

Alex's Anthology of Algorithms
Common Code for Contests in Concise C++
(Draft, December 2015)

Compiled, Edited, and Written by
Alex Li
University of Toronto
Email: alex@alexli.ca

This page is intentionally left blank.

Preface

Note: Visit <http://github.com/Alextrovert/Algorithm-Anthology> for the most up-to-date digital version of this codebook. The version you are reading is currently being reviewed, revised, and rewritten.

0.1 Introduction

This anthology started as a personal project to implement common algorithms in the most concise and "vanilla" way possible so that they're easily adaptable for use in algorithm competitions. To that end, several properties of the algorithm implementations should be satisfied, not limited to the following:

- Implementations must be clear. There is no time to write rigorous documentation within contests. This makes it all the more important to make class and variable names reflexive of what they represent. Clarity must also be carefully balanced with not making them too long-winded, since it can be just as time-consuming to type out long identifiers.
- Implementations must be generic. The more code that must be changed during the contest, the more room there is for mistakes. Thus, it should be easy to apply implementations to different purposes. C++ templates are often used to accomplish this at the slight cost of readability.
- Implementations must be portable. Different contest environments use different versions of C++ (though almost all of them use GCC), so in order to make programs as compatible as possible, non-standard features should be avoided. This is also why no features from C++0x or above are used, since many contest systems remain stuck on older versions of the language. Refer to the "Portability" section below for more information.
- Implementations must be efficient. The code cannot simply demonstrate an idea, it should also have the correct running time and a reasonably low constant overhead. This is sometimes challenging if concision is to be preserved. However, contest problem setters will often be understanding and set time limits liberally. If an implementation from here does not pass in time, chances are you are choosing the wrong algorithm.
- Implementations must be concise. During timed contests, code chunks are often moved around the file. To minimize the amount of scrolling, code design and formatting conventions should ensure as much code fits on the screen as possible (while not excessively sacrificing readability). It's a given that each algorithm should be placed within singleton files. Nearly all contest environments demand submissions to be contained within a single file.

A good trade-off between clarity, genericness, portability, efficiency, and concision is what comprises the ultimate goal of adaptability.

0.2 Portability

All programs are tested with version 4.7.3 of the GNU Compiler Collection (GCC) compiled for a 32-bit target system.

That means the following assumptions are made:

- 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 related features [`-Wlong-long`] (such as `LLONG_MIN` in `<climits>`), which are compliant in C99/C++0x or later. 64-bit integers are a must for many programming contest problems, so it is necessary to include these.
- Usage of variable sized arrays [`-Wvla`] (an easy fix using vectors, but I chose to keep it because it is simpler and because dynamic memory is generally good to avoid in contests)
- Usage of GCC's built-in functions like `__builtin_popcount()` and `__builtin_clz()`. These can be extremely convenient, and are easily implemented if they're not available. 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 is used in the anthology because it makes code much more concise by not having to define a constructor (which is trivial to do).
- Ad-hoc cases where bitwise hacks are intentionally used, such as functions for getting the signbit with type-punned pointers. If you are looking for these features, chances are you don't care about portability anyway.

0.3 Usage Notes

The primary purpose of this project is not to better your understanding of algorithms. To take advantage of this anthology, you must have prior understanding of the algorithms in question. In each source code file, you will find brief descriptions and simple examples to clarify how the functions and classes should be used (not so much how they work). This is why if you actually want to learn algorithms, you are better off researching the idea and trying to implement it independently. Directly using the code found here should be considered a last resort during the pressures of an actual contest.

All information from the comments (descriptions, complexities, etc.) come from Wikipedia and other online sources. Some programs here are direct implementations of pseudocode found online, while others are adapted and translated from informatics books and journals. If references for a program are not listed in its comments, you may assume that I have written them from scratch. You are free to use, modify, and distribute these programs in accordance to the license, but please first examine any corresponding references of each program for more details on usage and authorship.

Cheers and hope you enjoy!

— Alex Li

December, 2015

Contents

Preface	i
0.1 Introduction	i
0.2 Portability	ii
0.3 Usage Notes	ii
1 Elementary Algorithms	1
1.1 Array Transformations	1
1.1.1 Sorting Algorithms	1
1.1.2 Array Rotation	7
1.1.3 Counting Inversions	9
1.1.4 Coordinate Compression	12
1.1.5 Selection (Quickselect)	13
1.2 Array Queries	14
1.2.1 Longest Increasing Subsequence	14
1.2.2 Maximal Subarray Sum (Kadane's)	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	22
1.3.1 Discrete Binary Search	22
1.3.2 Ternary Search	24
1.3.3 Hill Climbing	26
1.3.4 Convex Hull Trick (Semi-Dynamic)	27
1.3.5 Convex Hull Trick (Fully Dynamic)	29
1.4 Cycle Detection	31
1.4.1 Floyd's Algorithm	31

1.4.2	Brent's Algorithm	33
1.5	Binary Exponentiation	35
2	Graph Theory	37
2.1	Depth-First Search	37
2.1.1	Graph Class and Depth-First Search	37
2.1.2	Topological Sorting	40
2.1.3	Eulerian Cycles	41
2.1.4	Unweighted Tree Centers	43
2.2	Shortest Paths	45
2.2.1	Breadth First Search	45
2.2.2	Dijkstra's Algorithm	47
2.2.3	Bellman-Ford Algorithm	48
2.2.4	Floyd-Warshall Algorithm	50
2.3	Connectivity	51
2.3.1	Strongly Connected Components (Kosaraju's Algorithm)	51
2.3.2	Strongly Connected Components (Tarjan's Algorithm)	53
2.3.3	Bridges, Cut-points, and Biconnectivity	55
2.4	Minimal Spanning Trees	58
2.4.1	Prim's Algorithm	58
2.4.2	Kruskal's Algorithm	60
2.5	Maximum Flow	61
2.5.1	Ford-Fulkerson Algorithm	61
2.5.2	Edmonds-Karp Algorithm	63
2.5.3	Dinic's Algorithm	64
2.5.4	Push-Relabel Algorithm	66
2.6	Backtracking	68
2.6.1	Max Clique (Bron-Kerbosch Algorithm)	68
2.6.2	Graph Coloring	70
2.7	Maximum Matching	72
2.7.1	Maximum Bipartite Matching (Kuhn's Algorithm)	72
2.7.2	Maximum Bipartite Matching (Hopcroft-Karp Algorithm)	73
2.7.3	Maximum Graph Matching (Edmonds's Algorithm)	75
2.8	Hamiltonian Path and Cycle	77
2.8.1	Shortest Hamiltonian Cycle (Travelling Salesman)	77

2.8.2	Shortest Hamiltonian Path	79
3	Data Structures	81
3.1	Disjoint Sets	81
3.1.1	Disjoint Set Forest (Simple)	81
3.1.2	Disjoint Set Forest	82
3.2	Fenwick Trees	84
3.2.1	Simple Fenwick Tree	84
3.2.2	Fenwick Tree	85
3.2.3	Fenwick Tree (Point Query)	86
3.2.4	Fenwick Tree (Range Update)	87
3.2.5	Fenwick Tree (Map)	89
3.2.6	2D Fenwick Tree	90
3.2.7	2D Fenwick Tree (Range Update)	91
3.3	1D Range Queries	93
3.3.1	Simple Segment Tree	93
3.3.2	Segment Tree	95
3.3.3	Segment Tree (Range Updates)	97
3.3.4	Segment Tree (Fast, Non-recursive)	100
3.3.5	Implicit Treap	102
3.3.6	Sparse Table	107
3.3.7	Square Root Decomposition	108
3.3.8	Interval Tree (Augmented Treap)	110
3.4	2D Range Queries	113
3.4.1	Quadtree (Simple)	113
3.4.2	Quadtree	114
3.4.3	2D Segment Tree	117
3.4.4	K-d Tree (2D Range Query)	119
3.4.5	K-d Tree (Nearest Neighbor)	121
3.4.6	R-Tree (Nearest Segment)	123
3.4.7	2D Range Tree	125
3.5	Search Trees and Alternatives	127
3.5.1	Binary Search Tree	127
3.5.2	Treap	130
3.5.3	Size Balanced Tree (Order Statistics)	133

3.5.4	HashMap (Chaining)	137
3.5.5	Skip List (Probabilistic)	139
3.6	Tree Data Structures	142
3.6.1	Heavy-Light Decomposition	142
3.6.2	Link-Cut Tree	146
3.7	Lowest Common Ancestor	151
3.7.1	Sparse Tables	151
3.7.2	Segment Trees	153
4	Mathematics	155
4.1	Mathematics Toolbox	155
4.2	Combinatorics	162
4.2.1	Combinatorial Calculations	162
4.2.2	Enumerating Arrangements	165
4.2.3	Enumerating Permutations	168
4.2.4	Enumerating Combinations	173
4.2.5	Enumerating Partitions	177
4.2.6	Enumerating Generic Combinatorial Sequences	180
4.3	Number Theory	183
4.3.1	GCD, LCM, Mod Inverse, Chinese Remainder	183
4.3.2	Generating Primes	187
4.3.3	Primality Testing	190
4.3.4	Integer Factorization	193
4.3.5	Euler's Totient Function	197
4.4	Arbitrary Precision Arithmetic	198
4.4.1	Big Integers (Simple)	198
4.4.2	Big Integer and Rational Class	200
4.4.3	FFT and Multiplication	211
4.5	Linear Algebra	214
4.5.1	Matrix Class	214
4.5.2	Determinant (Gauss)	217
4.5.3	Gaussian Elimination	220
4.5.4	LU Decomposition	222
4.5.5	Simplex Algorithm	226
4.6	Root-Finding	228

4.6.1	Real Root Finding (Differentiation)	228
4.6.2	Complex Root Finding (Laguerre's)	230
4.6.3	Complex Root Finding (RPOLY)	232
4.7	Integration	240
4.7.1	Simpson's Rule	240
5	Geometry	242
5.1	Geometric Classes	242
5.1.1	Point	242
5.1.2	Line	245
5.1.3	Circle	247
5.2	Geometric Calculations	250
5.2.1	Angles	250
5.2.2	Distances	252
5.2.3	Line Intersections	254
5.2.4	Circle Intersections	257
5.3	Common Geometric Computations	261
5.3.1	Polygon Sorting and Area	261
5.3.2	Point in Polygon Query	263
5.3.3	Convex Hull	264
5.3.4	Minimum Enclosing Circle	265
5.3.5	Diameter of Point Set	267
5.3.6	Closest Point Pair	269
5.3.7	Segment Intersection Finding	270
5.4	Advanced Geometric Computations	274
5.4.1	Convex Polygon Cut	274
5.4.2	Polygon Union and Intersection	275
5.4.3	Delaunay Triangulation (Simple)	279
5.4.4	Delaunay Triangulation (Fast)	281
6	Strings	292
6.1	Strings Toolbox	292
6.2	Expression Parsing	296
6.2.1	Recursive Descent	296
6.2.2	Recursive Descent (Simple)	298

6.2.3	Shunting Yard Algorithm	299
6.3	String Searching	301
6.3.1	Longest Common Substring	301
6.3.2	Longest Common Subsequence	302
6.3.3	Edit Distance	304
6.4	Dynamic Programming	305
6.4.1	Longest Common Substring	305
6.4.2	Longest Common Subsequence	306
6.4.3	Edit Distance	308
6.5	Suffix Array and LCP	309
6.5.1	$\mathcal{O}(N \log^2 N)$ Construction	309
6.5.2	$\mathcal{O}(N \log N)$ Construction	310
6.5.3	$\mathcal{O}(N \log N)$ Construction (DC3/Skew)	312
6.6	String Data Structures	315
6.5.1	Simple Trie	315
6.5.2	Radix Trie	317
6.5.3	Suffix Trie	319
6.5.4	Suffix Automaton	322

Chapter 1

Elementary Algorithms

1.1 Array Transformations

1.1.1 Sorting Algorithms

```
1  /*
2
3  The following functions are to be used like std::sort(), taking two
4  RandomAccessIterators as the range to be sorted, and optionally a
5  comparison function object to replace the default < operator.
6
7  They are not intended to compete with the standard library sorting
8  functions in terms of speed, but are merely demonstrations of how to
9  implement common sorting algorithms concisely in C++.
10
11 */
12
13 #include <algorithm> /* std::copy(), std::swap() */
14 #include <functional> /* std::less */
15 #include <iterator> /* std::iterator_traits */
16
17 /*
18
19 Quicksort
20
21 Quicksort repeatedly selects a pivot and "partitions" the range so that
22 all values comparing less than the pivot come before it, and all values
23 comparing greater comes after it. Divide and conquer is then applied to
24 both sides of the pivot until the original range is sorted. Despite
25 having a worst case of  $O(n^2)$ , quicksort is faster in practice than
26 merge sort and heapsort, which each has a worst case of  $O(n \log n)$ .
27
28 The pivot chosen in this implementation is always the middle element
29 of the range to be sorted. To reduce the likelihood of encountering the
30 worst case, the algorithm should be modified to select a random pivot,
31 or use the "median of three" method.
32
33 Time Complexity (Average):  $O(n \log n)$ 
34 Time Complexity (Worst):  $O(n^2)$ 
35 Space Complexity:  $O(\log n)$  auxiliary.
```

```

36 Stable?: No
37
38 */
39
40 template<class It, class Compare>
41 void quicksort(It lo, It hi, Compare comp) {
42     if (hi - lo < 2) return;
43     typedef typename std::iterator_traits<It>::value_type T;
44     T pivot = *(lo + (hi - lo) / 2);
45     It i, j;
46     for (i = lo, j = hi - 1; ; ++i, --j) {
47         while (comp(*i, pivot))
48             ++i;
49         while (comp(pivot, *j))
50             --j;
51         if (i >= j)
52             break;
53         std::swap(*i, *j);
54     }
55     quicksort(lo, i, comp);
56     quicksort(i, hi, comp);
57 }
58
59 template<class It> void quicksort(It lo, It hi) {
60     typedef typename std::iterator_traits<It>::value_type T;
61     quicksort(lo, hi, std::less<T>());
62 }
63
64 /*
65
66 Merge Sort
67
68 Merge sort works by first dividing a list into n sublists, each with
69 one element, then recursively merging sublists to produce new sorted
70 sublists until only a single sorted sublist remains. Merge sort has a
71 better worst case than quicksort, and is also stable, meaning that it
72 will preserve the relative ordering of elements considered equal by
73 the < operator or comparator (a < b and b < a both return false).
74
75 While std::stable_sort() is a corresponding function in the standard
76 library, the implementation below differs in that it will simply fail
77 if extra memory is not available. Meanwhile, std::stable_sort() will
78 not fail, but instead fall back to a time complexity of  $O(n \log^2 n)$ .
79
80 Time Complexity (Average):  $O(n \log n)$ 
81 Time Complexity (Worst):  $O(n \log n)$ 
82 Space Complexity:  $O(n)$  auxiliary.
83 Stable?: Yes
84
85 */
86
87 template<class It, class Compare>
88 void mergesort(It lo, It hi, Compare comp) {
89     if (hi - lo < 2) return;
90     It mid = lo + (hi - lo - 1) / 2, a = lo, c = mid + 1;
91     mergesort(lo, mid + 1, comp);
92     mergesort(mid + 1, hi, comp);
93     typedef typename std::iterator_traits<It>::value_type T;
94     T *buf = new T[hi - lo], *b = buf;

```

```

95     while (a <= mid && c < hi)
96         *(b++) = comp(*c, *a) ? *(c++) : *(a++);
97     if (a > mid) {
98         for (It k = c; k < hi; k++)
99             *(b++) = *k;
100    } else {
101        for (It k = a; k <= mid; k++)
102            *(b++) = *k;
103    }
104    for (int i = hi - lo - 1; i >= 0; i--)
105        *(lo + i) = buf[i];
106    delete[] buf;
107 }
108
109 template<class It> void mergesort(It lo, It hi) {
110     typedef typename std::iterator_traits<It>::value_type T;
111     mergesort(lo, hi, std::less<T>());
112 }
113
114 /*
115
116 Heapsort
117
118 Heapsort first rearranges an array to satisfy the heap property, and
119 then the max element of the heap is repeatedly removed and added to the
120 end of the resulting sorted list. A heapified array has the root node
121 at index 0. The two children of the node at index n are respectively
122 located at indices 2n + 1 and 2n + 2. Each node is greater than both
123 of its children. This leads to a structure that takes  $O(\log n)$  to
124 insert any element or remove the max element. Heapsort has a better
125 worst case complexity than quicksort, but a better space complexity
126 complexity than merge sort.
127
128 The standard library equivalent is calling std::make_heap(), followed
129 by std::sort_heap() on the input range.
130
131 Time Complexity (Average):  $O(n \log n)$ 
132 Time Complexity (Worst):  $O(n \log n)$ 
133 Space Complexity:  $O(1)$  auxiliary.
134 Stable?: No
135
136 */
137
138 template<class It, class Compare>
139 void heapsort(It lo, It hi, Compare comp) {
140     typename std::iterator_traits<It>::value_type t;
141     It i = lo + (hi - lo) / 2, j = hi, parent, child;
142     for (;;) {
143         if (i <= lo) {
144             if (--j == lo)
145                 return;
146             t = *j;
147             *j = *lo;
148         } else {
149             t = *(--i);
150         }
151         parent = i;
152         child = lo + 2 * (i - lo) + 1;
153         while (child < j) {

```

```

154     if (child + 1 < j && comp(*child, *(child + 1)))
155         child++;
156     if (!comp(t, *child))
157         break;
158     *parent = *child;
159     parent = child;
160     child = lo + 2 * (parent - lo) + 1;
161 }
162 *(lo + (parent - lo)) = t;
163 }
164 }
165
166 template<class It> void heapsort(It lo, It hi) {
167     typedef typename std::iterator_traits<It>::value_type T;
168     heapsort(lo, hi, std::less<T>());
169 }
170
171 /*
172
173 Comb Sort
174
175 Comb sort is an improved bubble sort. While bubble sort increases the
176 gap between swapped elements for every inner loop iteration, comb sort
177 uses a fixed gap for the inner loop and decreases the gap size by a
178 shrink factor for every iteration of the outer loop.
179
180 Even though the average time complexity is theoretically  $O(n^2)$ , if the
181 increments (gap sizes) are relatively prime and the shrink factor is
182 sensible (1.3 is empirically determined to be the best), then it will
183 require astronomically large  $n$  to make the algorithm exceed  $O(n \log n)$ 
184 steps. In practice, comb sort is only 2-3 times slower than merge sort.
185
186 Time Complexity (Average):  $O(n^2 / 2^p)$  for  $p$  increments.
187 Time Complexity (Worst):  $O(n^2)$ 
188 Space Complexity:  $O(1)$  auxiliary.
189 Stable?: No
190
191 */
192
193 template<class It, class Compare>
194 void combsort(It lo, It hi, Compare comp) {
195     int gap = hi - lo;
196     bool swapped = true;
197     while (gap > 1 || swapped) {
198         if (gap > 1)
199             gap = (int)((float)gap / 1.3f);
200         swapped = false;
201         for (It i = lo; i + gap < hi; i++)
202             if (comp(*(i + gap), *i)) {
203                 std::swap(*i, *(i + gap));
204                 swapped = true;
205             }
206     }
207 }
208
209 template<class It> void combsort(It lo, It hi) {
210     typedef typename std::iterator_traits<It>::value_type T;
211     combsort(lo, hi, std::less<T>());
212 }

```

```

213
214 /*
215
216 Radix Sort
217
218 Radix sort can be used to sort integer keys with a constant number of
219 bits in linear time. The keys are grouped by the individual digits of
220 a particular base which share the same significant position and value.
221
222 The implementation below only works on ranges pointing to unsigned
223 integer primitives (but can be modified to also work on signed values).
224 Note that the input range need not strictly be "unsigned" types, as
225 long as the values are all technically non-negative. A power of two is
226 chosen to be the base of the sort since bitwise operations may be used
227 to extract digits (instead of modulus and powers, which are much less
228 efficient). In practice, it's been demonstrated that 2^8 is the best
229 choice for sorting 32-bit integers (roughly 5 times faster than using
230 std::sort and 2 to 4 times faster than any other chosen power of two).
231
232 This implementation was adapted from: http://qr.ae/RbdDTa
233 Explanation of base 2^8 choice: http://qr.ae/RbdDcG
234
235 Time Complexity: O(n * w) for n integers of w bits.
236 Space Complexity: O(n + w) auxiliary.
237 Stable?: Yes
238
239 */
240
241 template<class UnsignedIt>
242 void radix_sort(UnsignedIt lo, UnsignedIt hi) {
243     if (hi - lo < 2)
244         return;
245     const int radix_bits = 8;
246     const int radix_base = 1 << radix_bits; //e.g. 2^8 = 256
247     const int radix_mask = radix_base - 1; //e.g. 2^8 - 1 = 0xFF
248     int num_bits = 8 * sizeof(*lo); //8 bits per byte
249     typedef typename std::iterator_traits<UnsignedIt>::value_type T;
250     T *l = new T[hi - lo];
251     for (int pos = 0; pos < num_bits; pos += radix_bits) {
252         int count[radix_base] = {0};
253         for (UnsignedIt it = lo; it != hi; it++)
254             count[(it >> pos) & radix_mask]++;
255         T *bucket[radix_base], *curr = l;
256         for (int i = 0; i < radix_base; curr += count[i++])
257             bucket[i] = curr;
258         for (UnsignedIt it = lo; it != hi; it++)
259             *bucket[(it >> pos) & radix_mask]++ = *it;
260         std::copy(l, l + (hi - lo), lo);
261     }
262     delete[] l;
263 }
264
265 /** Example Usage
266
267 Sample Output:
268
269 mergesort() with default comparisons: 1.32 1.41 1.62 1.73 2.58 2.72 3.14 4.67
270 mergesort() with 'compare_as_ints()': 1.41 1.73 1.32 1.62 2.72 2.58 3.14 4.67
271 -----

```

```

272 Sorting five million integers...
273 std::sort(): 0.429s
274 quicksort(): 0.498s
275 mergesort(): 1.437s
276 heapsort(): 1.179s
277 combsort(): 1.023s
278 radix_sort(): 0.078s
279
280 */
281
282 #include <cassert>
283 #include <cstdlib>
284 #include <ctime>
285 #include <iomanip>
286 #include <iostream>
287 #include <vector>
288 using namespace std;
289
290 template<class It> void print_range(It lo, It hi) {
291     while (lo != hi)
292         cout << *(lo++) << " ";
293     cout << endl;
294 }
295
296 template<class It> bool is_sorted(It lo, It hi) {
297     while (++lo != hi)
298         if (*(lo - 1) > *lo)
299             return false;
300     return true;
301 }
302
303 bool compare_as_ints(double i, double j) {
304     return (int)i < (int)j;
305 }
306
307 int main () {
308     { //can be used to sort arrays like std::sort()
309       int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
310       quicksort(a, a + 8);
311       assert(is_sorted(a, a + 8));
312     }
313     { //STL containers work too
314       int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
315       vector<int> v(a, a + 8);
316       quicksort(v.begin(), v.end());
317       assert(is_sorted(v.begin(), v.end()));
318     }
319     { //reverse iterators work as expected
320       int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
321       vector<int> v(a, a + 8);
322       heapsort(v.rbegin(), v.rend());
323       assert(is_sorted(v.rbegin(), v.rend()));
324     }
325     { //doubles are also fine
326       double a[] = {1.1, -5.0, 6.23, 4.123, 155.2};
327       vector<double> v(a, a + 5);
328       combsort(v.begin(), v.end());
329       assert(is_sorted(v.begin(), v.end()));
330     }

```



```

331 { //only unsigned ints work for radix_sort (but reverse works!)
332   int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
333   vector<int> v(a, a + 8);
334   radix_sort(v.rbegin(), v.rend());
335   assert(is_sorted(v.rbegin(), v.rend()));
336 }
337
338 //example from http://www.cplusplus.com/reference/algorithm/stable_sort
339 double a[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};
340 {
341   vector<double> v(a, a + 8);
342   cout << "mergesort() with default comparisons: ";
343   mergesort(v.begin(), v.end());
344   print_range(v.begin(), v.end());
345 }
346 {
347   vector<double> v(a, a + 8);
348   cout << "mergesort() with 'compare_as_ints()': ";
349   mergesort(v.begin(), v.end(), compare_as_ints);
350   print_range(v.begin(), v.end());
351 }
352 cout << "-----" << endl;
353
354 vector<int> v, v2;
355 for (int i = 0; i < 5000000; i++)
356   v.push_back((rand() & 0x7fff) | ((rand() & 0x7fff) << 15));
357 v2 = v;
358 cout << "Sorting five million integers..." << endl;
359 cout.precision(3);
360
361 #define test(sortfunc) { \
362   clock_t start = clock(); \
363   sortfunc(v.begin(), v.end()); \
364   double t = (double)(clock() - start) / CLOCKS_PER_SEC; \
365   cout << setw(14) << left << #sortfunc "(): "; \
366   cout << fixed << t << "s" << endl; \
367   assert(is_sorted(v.begin(), v.end())); \
368   v = v2; \
369 }
370 test(std::sort);
371 test(quick_sort);
372 test(mergesort);
373 test(heap_sort);
374 test(comb_sort);
375 test(radix_sort);
376
377 return 0;
378 }

```

1.1.2 Array Rotation

```

1  /*
2
3  The following functions are equivalent to std::rotate(), taking three
4  iterators lo, mid, hi, and swapping the elements in the range [lo, hi)
5  in such a way that the element at mid becomes the first element of the
6  new range and the element at mid - 1 becomes the last element.

```

```

7
8 All three versions achieve the same result using no temporary arrays.
9 Version 1 uses a straightforward swapping algorithm listed on many C++
10 reference sites, requiring only forward iterators. Version 2 requires
11 bidirectional iterators, employing the well-known technique of three
12 simple reversals. Version 3 applies a "juggling" algorithm which first
13 divides the range into gcd(n, k) sets (n = hi - lo and k = mid - lo)
14 and then rotates the corresponding elements in each set. This version
15 requires random access iterators.
16
17 Time Complexity: O(n) on the distance between lo and hi.
18 Space Complexity: O(1) auxiliary.
19
20 */
21
22 #include <algorithm> /* std::reverse(), std::rotate(), std::swap() */
23
24 template<class It> void rotate1(It lo, It mid, It hi) {
25     It next = mid;
26     while (lo != next) {
27         std::swap(*lo++, *next++);
28         if (next == hi)
29             next = mid;
30         else if (lo == mid)
31             mid = next;
32     }
33 }
34
35 template<class It> void rotate2(It lo, It mid, It hi) {
36     std::reverse(lo, mid);
37     std::reverse(mid, hi);
38     std::reverse(lo, hi);
39 }
40
41 int gcd(int a, int b) {
42     return b == 0 ? a : gcd(b, a % b);
43 }
44
45 template<class It> void rotate3(It lo, It mid, It hi) {
46     int n = hi - lo, jump = mid - lo;
47     int g = gcd(jump, n), cycle = n / g;
48     for (int i = 0; i < g; i++) {
49         int curr = i, next;
50         for (int j = 0; j < cycle - 1; j++) {
51             next = curr + jump;
52             if (next >= n)
53                 next -= n;
54             std::swap(*(lo + curr), *(lo + next));
55             curr = next;
56         }
57     }
58 }
59
60 /** Example Usage
61
62 Sample Output:
63
64 before sort:  2 4 2 0 5 10 7 3 7 1
65 after sort:   0 1 2 2 3 4 5 7 7 10

```

```

66 rotate left:  1 2 2 3 4 5 7 7 10 0
67 rotate right: 0 1 2 2 3 4 5 7 7 10
68
69 */
70
71 #include <algorithm>
72 #include <cassert>
73 #include <iostream>
74 #include <vector>
75 using namespace std;
76
77 int main() {
78     std::vector<int> v0, v1, v2, v3;
79     for (int i = 0; i < 10000; i++)
80         v0.push_back(i);
81     v1 = v2 = v3 = v0;
82     int mid = 5678;
83     std::rotate(v0.begin(), v0.begin() + mid, v0.end());
84     rotate1(v1.begin(), v1.begin() + mid, v1.end());
85     rotate2(v2.begin(), v2.begin() + mid, v2.end());
86     rotate3(v3.begin(), v3.begin() + mid, v3.end());
87     assert(v0 == v1 && v0 == v2 && v0 == v3);
88
89     //example from: http://en.cppreference.com/w/cpp/algorithm/rotate
90     int a[] = {2, 4, 2, 0, 5, 10, 7, 3, 7, 1};
91     vector<int> v(a, a + 10);
92     cout << "before_sort: ";
93     for (int i = 0; i < (int)v.size(); i++)
94         cout << v[i] << ' ';
95     cout << endl;
96
97     //insertion sort
98     for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)
99         rotate1(std::upper_bound(v.begin(), i, *i), i, i + 1);
100    cout << "after_sort: ";
101    for (int i = 0; i < (int)v.size(); i++)
102        cout << v[i] << ' ';
103    cout << endl;
104
105    //simple rotation to the left
106    rotate2(v.begin(), v.begin() + 1, v.end());
107    cout << "rotate_left: ";
108    for (int i = 0; i < (int)v.size(); i++)
109        cout << v[i] << ' ';
110    cout << endl;
111
112    //simple rotation to the right
113    rotate3(v.rbegin(), v.rbegin() + 1, v.rend());
114    cout << "rotate_right: ";
115    for (int i = 0; i < (int)v.size(); i++)
116        cout << v[i] << ' ';
117    cout << endl;
118    return 0;
119 }
```

1.1.3 Counting Inversions

```

1  /*
2
3  The number of inversions in an array a[] is the number of ordered pairs
4  (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 same as the minimum number
6  of swaps required to sort the array. If the array is sorted then the
7  inversion count is 0. If the array is sorted in decreasing order, then
8  the inversion count is maximal. The following are two methods of
9  efficiently counting the number of inversions.
10
11 */
12
13 #include <algorithm> /* std::fill(), std::max() */
14 #include <iterator> /* std::iterator_traits */
15
16 /*
17
18 Version 1: Merge sort
19
20 The input range [lo, hi) will become sorted after the function call,
21 and then the number of inversions will be returned. The iterator's
22 value type must have the less than < operator defined appropriately.
23
24 Explanation: http://www.geeksforgeeks.org/counting-inversions
25
26 Time Complexity:  $O(n \log n)$  on the distance between lo and hi.
27 Space Complexity:  $O(n)$  auxiliary.
28
29 */
30
31 template<class It> long long inversions(It lo, It hi) {
32     if (hi - lo < 2) return 0;
33     It mid = lo + (hi - lo - 1) / 2, a = lo, c = mid + 1;
34     long long res = 0;
35     res += inversions(lo, mid + 1);
36     res += inversions(mid + 1, hi);
37     typedef typename std::iterator_traits<It>::value_type T;
38     T *buf = new T[hi - lo], *ptr = buf;
39     while (a <= mid && c < hi) {
40         if (*c < *a) {
41             *(ptr++) = *(c++);
42             res += (mid - a) + 1;
43         } else {
44             *(ptr++) = *(a++);
45         }
46     }
47     if (a > mid) {
48         for (It k = c; k < hi; k++)
49             *(ptr++) = *k;
50     } else {
51         for (It k = a; k <= mid; k++)
52             *(ptr++) = *k;
53     }
54     for (int i = hi - lo - 1; i >= 0; i--)
55         *(lo + i) = buf[i];
56     delete[] buf;
57     return res;
58 }
59

```

```

60  /*
61
62  Version 2: Magic
63
64  The following magic is courtesy of misof, and works for any array of
65  nonnegative integers.
66
67  Explanation: http://codeforces.com/blog/entry/17881?comment=232099
68
69  The complexity depends on the magnitude of the maximum value in a[].
70  Coordinate compression should be applied on the values of a[] so that
71  they are strictly integers with magnitudes up to n for best results.
72  Note that after calling the function, a[] will be entirely set to 0.
73
74  Time Complexity:  $O(m \log m)$ , where m is maximum value in the array.
75  Space Complexity:  $O(m)$  auxiliary.
76
77  */
78
79  long long inversions(int n, int a[]) {
80      int mx = 0;
81      for (int i = 0; i < n; i++)
82          mx = std::max(mx, a[i]);
83      int *cnt = new int[mx];
84      long long res = 0;
85      while (mx > 0) {
86          std::fill(cnt, cnt + mx, 0);
87          for (int i = 0; i < n; i++) {
88              if (a[i] % 2 == 0)
89                  res += cnt[a[i] / 2];
90              else
91                  cnt[a[i] / 2]++;
92          }
93          mx = 0;
94          for (int i = 0; i < n; i++)
95              mx = std::max(mx, a[i] /= 2);
96      }
97      delete[] cnt;
98      return res;
99  }
100
101  /** Example Usage */
102
103  #include <cassert>
104
105  int main() {
106      {
107          int a[] = {6, 9, 1, 14, 8, 12, 3, 2};
108          assert(inversions(a, a + 8) == 16);
109      }
110      {
111          int a[] = {6, 9, 1, 14, 8, 12, 3, 2};
112          assert(inversions(8, a) == 16);
113      }
114      return 0;
115  }

```

1.1.4 Coordinate Compression

```

1  /*
2
3  Given an array a[] of size n, reassign integers to each value of a[]
4  such that the magnitude of each new value is no more than n, while the
5  relative order of each value as they were in the original array is
6  preserved. That is, if a[] is the original array and b[] is the result
7  array, then for every pair (i, j), the result of comparing a[i] < a[j]
8  will be exactly the same as the result of b[i] < b[j]. Furthermore,
9  no value of b[] will exceed the *number* of distinct values in a[].
10
11 In the following implementations, values in the range [lo, hi) will be
12 converted to integers in the range [0, d), where d is the number of
13 distinct values in the original range. lo and hi must be random access
14 iterators pointing to a numerical type that int can be assigned to.
15
16 Time Complexity: O(n log n) on the distance between lo and hi.
17 Space Complexity: O(n) auxiliary.
18
19 */
20
21 #include <algorithm> /* std::lower_bound(), std::sort(), std::unique() */
22 #include <iterator> /* std::iterator_traits */
23 #include <map>
24
25 //version 1 - using std::sort(), std::unique() and std::lower_bound()
26 template<class It> void compress1(It lo, It hi) {
27     typedef typename std::iterator_traits<It>::value_type T;
28     T *a = new T[hi - lo];
29     int n = 0;
30     for (It it = lo; it != hi; ++it)
31         a[n++] = *it;
32     std::sort(a, a + n);
33     int n2 = std::unique(a, a + n) - a;
34     for (It it = lo; it != hi; ++it)
35         *it = (int)(std::lower_bound(a, a + n2, *it) - a);
36     delete[] a;
37 }
38
39 //version 2 - using std::map
40 template<class It> void compress2(It lo, It hi) {
41     typedef typename std::iterator_traits<It>::value_type T;
42     std::map<T, int> m;
43     for (It it = lo; it != hi; ++it)
44         m[*it] = 0;
45     typename std::map<T, int>::iterator x = m.begin();
46     for (int i = 0; x != m.end(); x++)
47         x->second = i++;
48     for (It it = lo; it != hi; ++it)
49         *it = m[*it];
50 }
51
52 /** Example Usage
53
54 Sample Output:
55
56 0 4 4 1 3 2 5 5

```

```

57 0 4 4 1 3 2 5 5
58 1 0 2 0 3 1
59
60 ***/
61
62 #include <iostream>
63 using namespace std;
64
65 template<class It> void print_range(It lo, It hi) {
66     while (lo != hi)
67         cout << *(lo++) << " ";
68     cout << endl;
69 }
70
71 int main() {
72     {
73         int a[] = {1, 30, 30, 7, 9, 8, 99, 99};
74         compress1(a, a + 8);
75         print_range(a, a + 8);
76     }
77     {
78         int a[] = {1, 30, 30, 7, 9, 8, 99, 99};
79         compress2(a, a + 8);
80         print_range(a, a + 8);
81     }
82     { //works on doubles too
83         double a[] = {0.5, -1.0, 3, -1.0, 20, 0.5};
84         compress1(a, a + 6);
85         print_range(a, a + 6);
86     }
87     return 0;
88 }

```

1.1.5 Selection (Quickselect)

```

1  /*
2
3  Quickselect (also known as Hoare's algorithm) is a selection algorithm
4  which rearranges the elements in a sequence such that the element at
5  the nth position is the element that would be there if the sequence
6  were sorted. The other elements in the sequence are partitioned around
7  the nth element. That is, they are left in no particular order, except
8  that no element before the nth element is greater than it, and no
9  element after it is less.
10
11 The following implementation is equivalent to std::nth_element(),
12 taking in two random access iterators as the range and performing the
13 described operation in expected linear time.
14
15 Time Complexity (Average):  $O(n)$  on the distance between lo and hi.
16 Time Complexity (Worst):  $O(n^2)$ , although this *almost never* occurs.
17 Space Complexity:  $O(1)$  auxiliary.
18
19 */
20
21 #include <algorithm> /* std::swap() */
22 #include <cstdlib> /* rand() */

```

```

23 #include <iterator>  /* std::iterator_traits */
24
25 int rand32() {
26     return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
27 }
28
29 template<class It> It rand_partition(It lo, It hi) {
30     std::swap(*(lo + rand32() % (hi - lo)), *(hi - 1));
31     typename std::iterator_traits<It>::value_type mid = *(hi - 1);
32     It i = lo - 1;
33     for (It j = lo; j != hi; ++j)
34         if (*j <= mid)
35             std::swap(*(++i), *j);
36     return i;
37 }
38
39 template<class It> void nth_element2(It lo, It n, It hi) {
40     for (;;) {
41         It k = rand_partition(lo, hi);
42         if (n < k)
43             hi = k;
44         else if (n > k)
45             lo = k + 1;
46         else
47             return;
48     }
49 }
50
51 /** Example Usage
52
53 Sample Output:
54 2 3 1 5 4 6 8 7 9
55
56 ***/
57
58 #include <iostream>
59 using namespace std;
60
61 template<class It> void print_range(It lo, It hi) {
62     while (lo != hi)
63         cout << *(lo++) << " ";
64     cout << endl;
65 }
66
67 int main () {
68     int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
69     random_shuffle(a, a + 9);
70     nth_element2(a, a + 5, a + 9);
71     print_range(a, a + 9);
72     return 0;
73 }

```

1.2 Array Queries

1.2.1 Longest Increasing Subsequence


```

1  /*
2
3  Given an array a[] of size n, determine a longest subsequence of a[]
4  such that all of its elements are in ascending order. This subsequence
5  is not necessarily contiguous or unique, so only one such answer will
6  be found. The problem is efficiently solved using dynamic programming
7  and binary searching, since it has the following optimal substructure
8  with respect to the i-th position in the array:
9
10     LIS[i] = 1 + max(LIS[j] for all j < i and a[j] < a[i])
11     Otherwise if such a j does not exist, then LIS[i] = 1.
12
13 Explanation: https://en.wikipedia.org/wiki/Longest\_increasing\_subsequence
14
15 Time Complexity: O(n log n) on the size of the array.
16 Space Complexity: O(n) auxiliary.
17
18 */
19
20 #include <vector>
21
22 std::vector<int> tail, prev;
23
24 template<class T> int lower_bound(int len, T a[], int key) {
25     int lo = -1, hi = len;
26     while (hi - lo > 1) {
27         int mid = (lo + hi) / 2;
28         if (a[tail[mid]] < key)
29             lo = mid;
30         else
31             hi = mid;
32     }
33     return hi;
34 }
35
36 template<class T> std::vector<T> lis(int n, T a[]) {
37     tail.resize(n);
38     prev.resize(n);
39     int len = 0;
40     for (int i = 0; i < n; i++) {
41         int pos = lower_bound(len, a, a[i]);
42         if (len < pos + 1)
43             len = pos + 1;
44         prev[i] = pos > 0 ? tail[pos - 1] : -1;
45         tail[pos] = i;
46     }
47     std::vector<T> res(len);
48     for (int i = tail[len - 1]; i != -1; i = prev[i])
49         res[--len] = a[i];
50     return res;
51 }
52
53 /** Example Usage
54
55 Sample Output:
56 -5 1 9 10 11 13
57
58 */
59

```

```

60 #include <iostream>
61 using namespace std;
62
63 template<class It> void print_range(It lo, It hi) {
64     while (lo != hi)
65         cout << *(lo++) << " ";
66     cout << endl;
67 }
68
69 int main () {
70     int a[] = {-2, -5, 1, 9, 10, 8, 11, 10, 13, 11};
71     vector<int> res = lis(10, a);
72     print_range(res.begin(), res.end());
73     return 0;
74 }

```

1.2.2 Maximal Subarray Sum (Kadane's)

```

1  /*
2
3  Given a sequence of numbers (with at least one positive number), find
4  the maximum possible sum of any contiguous subarray. Kadane's algorithm
5  scans through the array, computing at each index the maximum (positive
6  sum) subarray ending at that position. This subarray is either empty
7  (in which case its sum is zero) or consists of one more element than
8  the maximum subarray ending at the previous position.
9
10 */
11
12 #include <algorithm> /* std::fill() */
13 #include <iterator> /* std::iterator_traits */
14 #include <limits> /* std::numeric_limits */
15 #include <vector>
16
17 /*
18
19 The following implementation takes two random access iterators as the
20 range of values to be considered. Optionally, two pointers to integers
21 may be passed to have the positions of the begin and end indices of
22 the maximal sum subarray stored. begin_idx will be inclusive while
23 end_idx will be exclusive (i.e. (lo + begin_idx) will reference the
24 first element of the max sum subarray and (lo + end_idx) will reference
25 the index just past the last element of the subarray. Note that the
26 following version does not allow empty subarrays to be returned, so the
27 the max element will simply be returned if the array is all negative.
28
29 Time Complexity: O(n) on the distance between lo and hi.
30 Space Complexity: O(1) auxiliary.
31
32 */
33
34 template<class It> typename std::iterator_traits<It>::value_type
35 max_subarray_sum(It lo, It hi, int *begin_idx = 0, int *end_idx = 0) {
36     typedef typename std::iterator_traits<It>::value_type T;
37     int curr_begin = 0, begin = 0, end = -1;
38     T sum = 0, max_sum = std::numeric_limits<T>::min();
39     for (It it = lo; it != hi; ++it) {

```

```

40     sum += *it;
41     if (sum < 0) {
42         sum = 0;
43         curr_begin = (it - lo) + 1;
44     } else if (max_sum < sum) {
45         max_sum = sum;
46         begin = curr_begin;
47         end = (it - lo) + 1;
48     }
49 }
50 if (end == -1) { //all negative, just return the max value
51     for (It it = lo; it != hi; ++it) {
52         if (max_sum < *it) {
53             max_sum = *it;
54             begin = it - lo;
55             end = begin + 1;
56         }
57     }
58 }
59 if (begin_idx != 0 && end_idx != 0) {
60     *begin_idx = begin;
61     *end_idx = end;
62 }
63 return max_sum;
64 }
65
66 /*
67
68 Maximal Submatrix Sum
69
70 In the 2-dimensional version of the problem, the largest sum of any
71 rectangular submatrix must be found for a matrix n rows by m columns.
72 Kadane's algorithm is applied to each interval [lcol, hcol] of columns
73 in the matrix, for an overall cubic time solution. The input must be a
74 two dimensional vector, where the outer vector must contain n vectors
75 each with m elements. Optionally, four int pointers begin_row, end_row,
76 begin_col, and end_col may be passed. If so, then their dereferenced
77 values will be set to the boundary indices of the max sum submatrix.
78 Note that begin_row and begin_col are inclusive indices, while end_row
79 and end_col are exclusive (referring to the index just past the end).
80
81 Time Complexity:  $O(m^2 * n)$  for a matrix with m columns and n rows.
82 Space Complexity:  $O(n)$  auxiliary.
83
84 */
85
86 template<class T>
87 T max_submatrix_sum(const std::vector< std::vector<T> > & mat,
88                     int *begin_row = 0, int *end_row = 0,
89                     int *begin_col = 0, int *end_col = 0) {
90     int n = mat.size(), m = mat[0].size();
91     std::vector<T> sums(n);
92     T sum, max_sum = std::numeric_limits<T>::min();
93     for (int lcol = 0; lcol < m; lcol++) {
94         std::fill(sums.begin(), sums.end(), 0);
95         for (int hcol = lcol; hcol < m; hcol++) {
96             for (int i = 0; i < n; i++)
97                 sums[i] += mat[i][hcol];
98             int begin, end;

```

```

99     sum = max_subarray_sum(sums.begin(), sums.end(), &begin, &end);
100     if (sum > max_sum) {
101         max_sum = sum;
102         if (begin_row != 0) {
103             *begin_row = begin;
104             *end_row = end;
105             *begin_col = lcol;
106             *end_col = hcol + 1;
107         }
108     }
109 }
110 }
111 return max_sum;
112 }
113
114 /** Example Usage
115 Sample Output:
116 1D example - the max sum subarray is
117 4 -1 2 1
118 2D example - the max sum submatrix is
119 9 2
120 -4 1
121 -1 8
122
123 */
124
125 #include <cassert>
126 #include <iostream>
127 using namespace std;
128
129 int main() {
130     {
131         int a[] = {-2, -1, -3, 4, -1, 2, 1, -5, 4};
132         int begin, end;
133         assert(max_subarray_sum(a, a + 3) == -1);
134         assert(max_subarray_sum(a, a + 9, &begin, &end) == 6);
135         cout << "1D example - the max sum subarray is" << endl;
136         for (int i = begin; i < end; i++)
137             cout << a[i] << " ";
138         cout << endl;
139     }
140     {
141         const int n = 4, m = 5;
142         int a[n][m] = {{ 0, -2, -7, 0, 5},
143                        { 9, 2, -6, 2, -4},
144                        {-4, 1, -4, 1, 0},
145                        {-1, 8, 0, -2, 3}};
146         vector< vector<int> > mat(n);
147         for (int i = 0; i < n; i++)
148             mat[i] = vector<int>(a[i], a[i] + m);
149         int lrow, hrow, lcol, hcol;
150         assert(max_submatrix_sum(mat, &lrow, &hrow, &lcol, &hcol) == 15);
151         cout << "2D example - The max sum submatrix is" << endl;
152         for (int i = lrow; i < hrow; i++) {
153             for (int j = lcol; j < hcol; j++)
154                 cout << mat[i][j] << " ";
155             cout << endl;
156         }
157     }

```

```

158     }
159     return 0;
160 }

```

1.2.3 Majority Element (Boyer-Moore)

```

1  /*
2
3  Given a sequence of n elements, the majority vote problem asks to find
4  an element that occurs more frequently than all others, or determine
5  that no such element exists. Formally, a value must occur strictly
6  greater than floor(n/2) times to be considered the majority element.
7  Boyer-Moore majority vote algorithm scans through the sequence and
8  keeps track of a running counter for the most likely candidate so far.
9  Whenever a value is equal to the current candidate, the counter is
10 incremented, otherwise the counter is decremented. When the counter is
11 zero, the candidate is eliminated and a new candidate is considered.
12
13 The following implementation takes two random access iterators as the
14 sequence [lo, hi) of elements and returns an iterator pointing to one
15 instance of the majority element if it exists, or the iterator hi if
16 there is no majority.
17
18 Time Complexity: O(n) on the size of the array.
19 Space Complexity: O(1) auxiliary.
20
21 */
22
23 template<class It> It majority(It lo, It hi) {
24     int cnt = 0;
25     It candidate = lo;
26     for (It it = lo; it != hi; ++it) {
27         if (cnt == 0) {
28             candidate = it;
29             cnt = 1;
30         } else if (*it == *candidate) {
31             cnt++;
32         } else {
33             cnt--;
34         }
35     }
36     cnt = 0;
37     for (It it = lo; it != hi; ++it) {
38         if (*it == *candidate)
39             cnt++;
40     }
41     if (cnt <= (hi - lo) / 2)
42         return hi;
43     return candidate;
44 }
45
46 /** Example Usage */
47
48 #include <cassert>
49
50 int main() {
51     int a[] = {3, 2, 3, 1, 3};

```

```

52  assert(*majority(a, a + 5) == 3);
53  int b[] = {2, 3, 3, 3, 2, 1};
54  assert(majority(b, b + 6) == b + 6);
55  return 0;
56 }

```

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

```

1  /*
2
3  Given a sequence of n (not necessarily unique) integers and a number v,
4  determine the minimum possible sum of any subset of the given sequence
5  that is not less than v. This is a generalization of a more well-known
6  version of the subset sum problem which asks whether a subset summing
7  to 0 exists (equivalent here to seeing if v = 0 yields an answer of 0).
8  Both problems are NP-complete. A meet-in-the-middle algorithm divides
9  the array in two equal parts. All possible sums of the lower and higher
10 parts are precomputed and sorted in a table. Finally, the table is
11 searched to find the lower bound.
12
13 The following implementation accepts two random access iterators as the
14 sequence [lo, hi) of integers, and the number v. Note that since the
15 sums can get large, 64-bit integers are necessary to avoid overflow.
16
17 Time Complexity:  $O(n * 2^{(n/2)})$  on the distance between lo and hi.
18 Space Complexity:  $O(n)$  auxiliary.
19
20 */
21
22 #include <algorithm> /* std::max(), std::sort() */
23 #include <limits>    /* std::numeric_limits */
24
25 template<class It>
26 long long sum_lower_bound(It lo, It hi, long long v) {
27     int n = hi - lo;
28     int llen = 1 << (n / 2);
29     int hlen = 1 << (n - n / 2);
30     long long *lsum = new long long[llen];
31     long long *hsum = new long long[hlen];
32     std::fill(lsum, lsum + llen, 0);
33     std::fill(hsum, hsum + hlen, 0);
34     for (int mask = 0; mask < llen; mask++) {
35         for (int i = 0; i < n / 2; i++) {
36             if ((mask >> i) & 1)
37                 lsum[mask] += *(lo + i);
38         }
39     }
40     for (int mask = 0; mask < hlen; mask++) {
41         for (int i = 0; i < n - n / 2; i++) {
42             if ((mask >> i) & 1)
43                 hsum[mask] += *(lo + i + n / 2);
44         }
45     }
46     std::sort(lsum, lsum + llen);
47     std::sort(hsum, hsum + hlen);
48     int l = 0, r = hlen - 1;
49     long long curr = std::numeric_limits<long long>::min();

```

```

50 while (l < llen && r >= 0) {
51     if (lsum[l] + hsum[r] <= v) {
52         curr = std::max(curr, lsum[l] + hsum[r]);
53         l++;
54     } else {
55         r--;
56     }
57 }
58 delete[] lsum;
59 delete[] hsum;
60 return curr;
61 }
62
63 /** Example Usage */
64
65 #include <cassert>
66
67 int main() {
68     int a[] = {9, 1, 5, 0, 1, 11, 5};
69     assert(sum_lower_bound(a, a + 7, 8) == 7);
70     int b[] = {-7, -3, -2, 5, 8};
71     assert(sum_lower_bound(b, b + 5, 0) == 0);
72     return 0;
73 }

```

1.2.5 Maximal Zero Submatrix

```

1  /*
2
3  Given an n by m rectangular matrix of 0's and 1's, determine the area
4  of the largest rectangular submatrix which contains only 0's. This can
5  be reduced the problem of finding the maximum rectangular area under a
6  histogram, which can be efficiently solved using a stack. The following
7  implementation accepts a 2-dimensional vector of bools and returns the
8  area of the maximum zero submatrix.
9
10 Explanation: http://stackoverflow.com/a/13657337
11
12 Time Complexity: O(n * m) for a matrix n rows by m columns.
13 Space Complexity: O(m) auxiliary.
14
15 */
16
17 #include <algorithm> /* std::max() */
18 #include <vector>
19
20 int max_zero_submatrix(const std::vector< std::vector<bool> > & mat) {
21     int n = mat.size(), m = mat[0].size(), res = 0;
22     std::vector<int> d(m, -1), d1(m), d2(m), stack;
23     for (int r = 0; r < n; r++) {
24         for (int c = 0; c < m; c++) {
25             if (mat[r][c])
26                 d[c] = r;
27         }
28         stack.clear();
29         for (int c = 0; c < m; c++) {
30             while (!stack.empty() && d[stack.back()] <= d[c])

```

```

31     stack.pop_back();
32     d1[c] = stack.empty() ? -1 : stack.back();
33     stack.push_back(c);
34 }
35 stack.clear();
36 for (int c = m - 1; c >= 0; c--) {
37     while (!stack.empty() && d[stack.back()] <= d[c])
38         stack.pop_back();
39     d2[c] = stack.empty() ? m : stack.back();
40     stack.push_back(c);
41 }
42 for (int j = 0; j < m; j++)
43     res = std::max(res, (r - d[j]) * (d2[j] - d1[j] - 1));
44 }
45 return res;
46 }
47
48 /** Example Usage */
49
50 #include <cassert>
51 using namespace std;
52
53 int main() {
54     const int n = 5, m = 6;
55     bool a[n][m] = {{1, 0, 1, 1, 0, 0},
56                     {1, 0, 0, 1, 0, 0},
57                     {0, 0, 0, 0, 0, 1},
58                     {1, 0, 0, 1, 0, 0},
59                     {1, 0, 1, 0, 0, 1}};
60     std::vector< std::vector<bool> > mat(n);
61     for (int i = 0; i < n; i++)
62         mat[i] = vector<bool>(a[i], a[i] + m);
63     assert(max_zero_submatrix(mat) == 6);
64     return 0;
65 }

```

1.3 Searching

1.3.1 Discrete Binary Search

```

1  /*
2
3  Not only can binary search be used to find the position of a given
4  element in a sorted array, it can also be used to find the input value
5  corresponding to any output value of a monotonic (either strictly
6  non-increasing or strictly non-decreasing) function in  $O(\log n)$  running
7  time with respect to the domain. This is a special case of finding
8  the exact point at which any given monotonic Boolean function changes
9  from true to false (or vice versa). Unlike searching through an array,
10 discrete binary search is not restricted by available memory, which is
11 especially important while handling infinitely large search spaces such
12 as the real numbers.
13
14 binary_search_first_true() takes two integers lo and hi as boundaries
15 for the search space [lo, hi) (i.e. including lo, but excluding hi),
16 and returns the least integer k ( $lo \leq k < hi$ ) for which the Boolean

```



```

17 predicate pred(k) tests true. This function is correct if and only if
18 there exists a constant k where the return value of pred(x) is false
19 for all  $x < k$  and true for all  $x \geq k$ .
20
21 binary_search_last_true() takes two integers lo and hi as boundaries
22 for the search space [lo, hi) (i.e. including lo, but excluding hi),
23 and returns the greatest integer k ( $lo \leq k < hi$ ) for which the Boolean
24 predicate pred(k) tests true. This function is correct if and only if
25 there exists a constant k where the return value of pred(x) is true
26 for all  $x \leq k$  and false for all  $x > k$ .
27
28 Time Complexity: At most  $O(\log n)$  calls to pred(), where n is the
29 distance between lo and hi.
30
31 Space Complexity:  $O(1)$  auxiliary.
32
33 */
34
35 //000[1]11
36 template<class Int, class IntPredicate>
37 Int binary_search_first_true(Int lo, Int hi, IntPredicate pred) {
38     Int mid, _hi = hi;
39     while (lo < hi) {
40         mid = lo + (hi - lo) / 2;
41         if (pred(mid))
42             hi = mid;
43         else
44             lo = mid + 1;
45     }
46     if (!pred(lo)) return _hi; //all false
47     return lo;
48 }
49
50 //11[1]000
51 template<class Int, class IntPredicate>
52 Int binary_search_last_true(Int lo, Int hi, IntPredicate pred) {
53     Int mid, _hi = hi;
54     while (lo < hi) {
55         mid = lo + (hi - lo + 1) / 2;
56         if (pred(mid))
57             lo = mid;
58         else
59             hi = mid - 1;
60     }
61     if (!pred(lo)) return _hi; //all true
62     return lo;
63 }
64
65 /*
66
67 fbinary_search() is the equivalent of binary_search_first_true() on
68 floating point predicates. Since any given range of reals numbers is
69 dense, it is clear that the exact target cannot be found. Instead, the
70 function will return a value that is very close to the border between
71 false and true. The precision of the answer depends on the number of
72 repetitions the function uses. Since each repetition bisects the search
73 space, for r repetitions, the absolute error of the answer will be
74  $1/(2^r)$  times the distance between lo and hi. Although it's possible to
75 control the error by looping while hi - lo is greater than an arbitrary

```

```

76 epsilon, it is much simpler to let the loop run for a sizable number of
77 iterations until floating point arithmetic breaks down. 100 iterations
78 is typically sufficient, reducing the search space to  $2^{-100} \sim 10^{-30}$ 
79 times its original size.
80
81 Note that the function can be modified to find the "last true" point
82 in the range by interchanging lo and hi in the if-else statement.
83
84 Time Complexity: At most  $O(\log n)$  calls to pred(), where n is the
85 distance between lo and hi divided by the desired absolute error.
86
87 Space Complexity:  $O(1)$  auxiliary.
88
89 */
90
91 //000[1]11
92 template<class DoublePredicate>
93 double fbinary_search(double lo, double hi, DoublePredicate pred) {
94     double mid;
95     for (int reps = 0; reps < 100; reps++) {
96         mid = (lo + hi) / 2.0;
97         if (pred(mid))
98             hi = mid;
99         else
100             lo = mid;
101     }
102     return lo;
103 }
104
105 /** Example Usage */
106
107 #include <cassert>
108 #include <cmath>
109
110 //Simple predicate examples:
111 bool pred1(int x) { return x >= 3; }
112 bool pred2(int x) { return false; }
113 bool pred3(int x) { return x <= 5; }
114 bool pred4(int x) { return true; }
115 bool pred5(double x) { return x >= 1.2345; }
116
117 int main() {
118     assert(binary_search_first_true(0, 7, pred1) == 3);
119     assert(binary_search_first_true(0, 7, pred2) == 7);
120     assert(binary_search_last_true(0, 7, pred3) == 5);
121     assert(binary_search_last_true(0, 7, pred4) == 7);
122     assert(fabs(fbinary_search(-10.0, 10.0, pred5) - 1.2345) < 1e-15);
123     return 0;
124 }

```

1.3.2 Ternary Search

```

1  /*
2
3  Given a unimodal function f(x), find its maximum or minimum point to a
4  an arbitrarily specified absolute error.
5

```

```

6  ternary_search_min() takes the domain [lo, hi] of a continuous function
7  f(x) and returns a number x such that f is strictly decreasing on the
8  interval [lo, x] and strictly increasing on the interval [x, hi]. For
9  ternary search to work, this x must exist and be unique.
10
11 ternary_search_max() takes the domain [lo, hi] of a continuous function
12 f(x) and returns a number x such that f is strictly increasing on the
13 interval [lo, x] and strictly decreasing on the interval [x, hi]. For
14 ternary search to work, this x must exist and be unique.
15
16 Time Complexity: At most  $O(\log n)$  calls to f, where n is the distance
17 between lo and hi divided by the desired absolute error (epsilon).
18
19 Space Complexity:  $O(1)$  auxiliary.
20
21 */
22
23 template<class UnimodalFunction>
24 double ternary_search_min(double lo, double hi, UnimodalFunction f) {
25     static const double EPS = 1e-9;
26     double lthird, hthird;
27     while (hi - lo > EPS) {
28         lthird = lo + (hi - lo) / 3;
29         hthird = hi - (hi - lo) / 3;
30         if (f(lthird) < f(hthird))
31             hi = hthird;
32         else
33             lo = lthird;
34     }
35     return lo;
36 }
37
38 template<class UnimodalFunction>
39 double ternary_search_max(double lo, double hi, UnimodalFunction f) {
40     static const double EPS = 1e-9;
41     double lthird, hthird;
42     while (hi - lo > EPS) {
43         lthird = lo + (hi - lo) / 3;
44         hthird = hi - (hi - lo) / 3;
45         if (f(lthird) < f(hthird))
46             lo = lthird;
47         else
48             hi = hthird;
49     }
50     return hi;
51 }
52
53 /** Example Usage */
54
55 #include <cmath>
56 #include <cassert>
57
58 bool eq(double a, double b) {
59     return fabs(a - b) < 1e-9;
60 }
61
62 //parabola opening up with vertex at (-2, -24)
63 double f1(double x) {
64     return 3*x*x + 12*x - 12;

```

```

65 }
66
67 //parabola opening down with vertex at (2/19, 8366/95)
68 double f2(double x) {
69     return -5.7*x*x + 1.2*x + 88;
70 }
71
72 //absolute value function shifted to the right by 30 units
73 double f3(double x) {
74     return fabs(x - 30);
75 }
76
77 int main() {
78     assert(eq(ternary_search_min(-1000, 1000, f1), -2));
79     assert(eq(ternary_search_max(-1000, 1000, f2), 2.0 / 19));
80     assert(eq(ternary_search_min(-1000, 1000, f3), 30));
81     return 0;
82 }

```

1.3.3 Hill Climbing

```

1  /*
2
3  Given a continuous function f on two real numbers, hill climbing is a
4  technique that can be used to find the local maximum or minimum point
5  based on some (possibly random) initial guess. Then, the algorithm
6  considers taking a single step in each of a fixed number of directions.
7  The direction with the best result is selected and steps are further
8  taken there until the answer no longer improves. When this happens, the
9  step size is reduced and the process repeats until a desired absolute
10 error is reached. The result is not necessarily the global extrema, and
11 the algorithm's success will heavily depend on the initial guess.
12
13 The following function find_min() takes the function f, any starting
14 guess (x0, y0), and optionally two pointers to double used for storing
15 the answer coordinates. find_min() returns a local minimum point near
16 the initial guess, and if the two pointers are given, then coordinates
17 will be stored into the variables pointed to by x_ans and y_ans.
18
19 Time Complexity: At most  $O(d \log n)$  calls to f, where d is the number
20 of directions considered at each position and n is the search space,
21 roughly proportional to the largest possible step size divided by the
22 smallest possible step size.
23
24 */
25
26 #include <cmath>
27 #include <iostream>
28 using namespace std;
29
30 template<class BinaryFunction>
31 double find_min(BinaryFunction f, double x0, double y0,
32                 double *x_ans = 0, double *y_ans = 0) {
33     static const double PI = acos(-1.0);
34     static const double STEP_MAX = 1000000;
35     static const double STEP_MIN = 1e-9;
36     static const int DIRECTIONS = 6;

```

```

37 double x = x0, y = y0, res = f(x0, y0);
38 for (double step = STEP_MAX; step > STEP_MIN; ) {
39     double best = res, best_x = x, best_y = y;
40     bool found = false;
41     for (int i = 0; i < DIRECTIONS; i++) {
42         double a = 2.0 * PI * i / DIRECTIONS;
43         double x2 = x + step * cos(a);
44         double y2 = y + step * sin(a);
45         double val = f(x2, y2);
46         if (best > val) {
47             best_x = x2;
48             best_y = y2;
49             best = val;
50             found = true;
51         }
52     }
53     if (!found) {
54         step /= 2.0;
55     } else {
56         x = best_x;
57         y = best_y;
58         res = best;
59     }
60 }
61 if (x_ans != 0 && y_ans != 0) {
62     *x_ans = x;
63     *y_ans = y;
64 }
65 return res;
66 }
67
68 /** Example Usage */
69
70 #include <cassert>
71 #include <cmath>
72
73 bool eq(double a, double b) {
74     return fabs(a - b) < 1e-8;
75 }
76
77 //minimized at f(2, 3) = 0
78 double f(double x, double y) {
79     return (x - 2)*(x - 2) + (y - 3)*(y - 3);
80 }
81
82 int main() {
83     double x, y;
84     assert(eq(find_min(f, 0, 0, &x, &y), 0));
85     assert(eq(x, 2) && eq(y, 3));
86     return 0;
87 }

```

1.3.4 Convex Hull Trick (Semi-Dynamic)

```

1  /*
2
3  Given a set of pairs (m, b) describing lines of the form  $y = mx + b$ ,

```

```

4 process a set of x-coordinate queries each asking to find the minimum
5 y-value of any of the given lines when evaluated at the specified x.
6 The convex hull optimization technique first ignores all lines which
7 never take on the maximum at any x value, then sorts the rest in order
8 of descending slope. The intersection points of adjacent lines in this
9 sorted list form the upper envelope of a convex hull, and line segments
10 connecting these points always take on the minimum y-value. The result
11 can be split up into x-intervals each mapped to the line which takes on
12 the minimum in that interval. The intervals can be binary searched to
13 solve each query in  $O(\log n)$  time on the number of lines.
14
15 Explanation: http://wcipeg.com/wiki/Convex\_hull\_trick
16
17 The following implementation is a concise, semi-dynamic version which
18 supports an an interlaced series of add_line and query operations.
19 However, two key preconditions are that each call to add_line(m, b)
20 must have m as the minimum slope of all lines added so far, and each
21 call to get_min(x) must have x as the maximum x of all queries so far.
22 As a result, pre-sorting the lines and queries may be necessary (in
23 which case the running time will be that of the sorting algorithm).
24
25 Time Complexity:  $O(n)$  on the number of calls to add_line(). Since the
26 number of steps taken by add_line() and get_min() are both bounded by
27 the number of lines added so far, their running times are respectively
28  $O(1)$  amortized.
29
30 Space Complexity:  $O(n)$  auxiliary on the number of calls to add_line().
31
32 */
33
34 #include <vector>
35
36 std::vector<long long> M, B;
37 int ptr = 0;
38
39 void add_line(long long m, long long b) {
40     int len = M.size();
41     while (len > 1 && (B[len - 2] - B[len - 1]) * (m - M[len - 1]) >=
42            (B[len - 1] - b) * (M[len - 1] - M[len - 2])) {
43         len--;
44     }
45     M.resize(len);
46     B.resize(len);
47     M.push_back(m);
48     B.push_back(b);
49 }
50
51 long long get_min(long long x) {
52     if (ptr >= (int)M.size())
53         ptr = (int)M.size() - 1;
54     while (ptr + 1 < (int)M.size() && M[ptr + 1] * x + B[ptr + 1] <=
55            M[ptr] * x + B[ptr]) {
56         ptr++;
57     }
58     return M[ptr] * x + B[ptr];
59 }
60
61 /** Example Usage */
62

```

```

63 #include <cassert>
64
65 int main() {
66     add_line(3, 0);
67     add_line(2, 1);
68     add_line(1, 2);
69     add_line(0, 6);
70     assert(get_min(0) == 0);
71     assert(get_min(1) == 3);
72     assert(get_min(2) == 4);
73     assert(get_min(3) == 5);
74     return 0;
75 }

```

1.3.5 Convex Hull Trick (Fully Dynamic)

```

1  /*
2
3  Given a set of pairs (m, b) describing lines of the form  $y = mx + b$ ,
4  process a set of x-coordinate queries each asking to find the minimum
5  y-value of any of the given lines when evaluated at the specified x.
6  The convex hull optimization technique first ignores all lines which
7  never take on the maximum at any x value, then sorts the rest in order
8  of descending slope. The intersection points of adjacent lines in this
9  sorted list form the upper envelope of a convex hull, and line segments
10 connecting these points always take on the minimum y-value. The result
11 can be split up into x-intervals each mapped to the line which takes on
12 the minimum in that interval. The intervals can be binary search to
13 solve each query in  $O(\log n)$  time on the number of lines.
14
15 Explanation: http://wcipeg.com/wiki/Convex\_hull\_trick
16
17 The following implementation is a fully dynamic version, using a
18 self-balancing binary search tree (std::set) to support calling line
19 addition and query operations in any desired order. In addition, one
20 may instead optimize for maximum y by setting QUERY_MAX to true.
21
22 Time Complexity:  $O(n \log n)$  for n calls to add_line(), where each call
23 is  $O(\log n)$  amortized on the number of lines added so far. Each call to
24 get_best() runs in  $O(\log n)$  on the number of lines added so far.
25
26 Space Complexity:  $O(n)$  auxiliary on the number of calls to add_line().
27
28 */
29
30 #include <set>
31
32 const bool QUERY_MAX = false;
33 const double INF = 1e30;
34
35 struct line {
36     long long m, b, val;
37     double xlo;
38     bool is_query;
39
40     line(long long m, long long b) {
41         this->m = m;

```

```

42     this->b = b;
43     val = 0;
44     xlo = -INF;
45     is_query = false;
46 }
47
48 long long evaluate(long long x) const {
49     return m * x + b;
50 }
51
52 bool parallel(const line & l) const {
53     return m == l.m;
54 }
55
56 double intersect(const line & l) const {
57     if (parallel(l))
58         return INF;
59     return (double)(l.b - b)/(m - l.m);
60 }
61
62 bool operator < (const line & l) const {
63     if (l.is_query)
64         return QUERY_MAX ? xlo < l.val : l.val < xlo;
65     return m < l.m;
66 }
67 };
68
69 std::set<line> hull;
70
71 typedef std::set<line>::iterator hulliter;
72
73 bool has_prev(hulliter it) {
74     return it != hull.begin();
75 }
76
77 bool has_next(hulliter it) {
78     return it != hull.end() && ++it != hull.end();
79 }
80
81 bool irrelevant(hulliter it) {
82     if (!has_prev(it) || !has_next(it))
83         return false;
84     hulliter prev = it; --prev;
85     hulliter next = it; ++next;
86     return QUERY_MAX ?
87         prev->intersect(*next) <= prev->intersect(*it) :
88         next->intersect(*prev) <= next->intersect(*it);
89 }
90
91 hulliter update_left_border(hulliter it) {
92     if ((QUERY_MAX && !has_prev(it)) || (!QUERY_MAX && !has_next(it)))
93         return it;
94     hulliter it2 = it;
95     double val = it->intersect(QUERY_MAX ? *--it2 : *++it2);
96     line buf(*it);
97     buf.xlo = val;
98     hull.erase(it++);
99     return hull.insert(it, buf);
100 }

```



```

101
102 void add_line(long long m, long long b) {
103     line l(m, b);
104     hulliter it = hull.lower_bound(l);
105     if (it != hull.end() && it->parallel(l)) {
106         if ((QUERY_MAX && it->b < b) || (!QUERY_MAX && b < it->b))
107             hull.erase(it++);
108         else
109             return;
110     }
111     it = hull.insert(it, l);
112     if (irrelevant(it)) {
113         hull.erase(it);
114         return;
115     }
116     while (has_prev(it) && irrelevant(--it))
117         hull.erase(it++);
118     while (has_next(it) && irrelevant(++it))
119         hull.erase(it--);
120     it = update_left_border(it);
121     if (has_prev(it))
122         update_left_border(--it);
123     if (has_next(++it))
124         update_left_border(++it);
125 }
126
127 long long get_best(long long x) {
128     line q(0, 0);
129     q.val = x;
130     q.is_query = true;
131     hulliter it = hull.lower_bound(q);
132     if (QUERY_MAX)
133         --it;
134     return it->evaluate(x);
135 }
136
137 /** Example Usage */
138
139 #include <cassert>
140
141 int main() {
142     add_line(3, 0);
143     add_line(0, 6);
144     add_line(1, 2);
145     add_line(2, 1);
146     assert(get_best(0) == 0);
147     assert(get_best(1) == 3);
148     assert(get_best(2) == 4);
149     assert(get_best(3) == 5);
150     return 0;
151 }

```

1.4 Cycle Detection

1.4.1 Floyd's Algorithm

```

1  /*
2
3  For a function f which maps a finite set S to itself and any initial
4  value x[0] in S, the same value must occur twice in the sequence below:
5
6      x[0], x[1] = f(x[0]), x[2] = f(x[1]), ..., x[i] = f(x[i - 1])
7
8  That is, there must exist numbers i, j (i < j) such that x[i] = x[j].
9  Once this happens, the sequence will continue periodically by repeating
10 the same sequence of values from x[i] to x[j - 1]. Cycle detection asks
11 to find i and j, given the function f and initial value x[0]. This is
12 also analogous to the problem of detecting a cycle in a linked list,
13 which will make it degenerate.
14
15 Floyd's cycle-finding algorithm, a.k.a. the "tortoise and the hare
16 algorithm", is a space-efficient algorithm that moves two pointers
17 through the sequence at different speeds. Each step in the algorithm
18 moves the "tortoise" one step forward and the "hare" two steps forward
19 in the sequence, comparing the sequence values at each step. The first
20 value which is simultaneously pointed to by both pointers is the start
21 of the sequence.
22
23 Time Complexity: O(mu + lambda), where mu is the smallest index of the
24 sequence on which a cycle starts, and lambda is the cycle's length.
25
26 Space Complexity: O(1) auxiliary.
27
28 */
29
30 #include <utility> /* std::pair */
31
32 template<class IntFunction>
33 std::pair<int, int> find_cycle(IntFunction f, int x0) {
34     int tortoise = f(x0), hare = f(f(x0));
35     while (tortoise != hare) {
36         tortoise = f(tortoise);
37         hare = f(f(hare));
38     }
39     int start = 0;
40     tortoise = x0;
41     while (tortoise != hare) {
42         tortoise = f(tortoise);
43         hare = f(hare);
44         start++;
45     }
46     int length = 1;
47     hare = f(tortoise);
48     while (tortoise != hare) {
49         hare = f(hare);
50         length++;
51     }
52     return std::make_pair(start, length);
53 }
54
55 /** Example Usage */
56
57 #include <cassert>
58 #include <set>
59 #include <iostream>

```

```

60 using namespace std;
61
62 const int x0 = 0;
63
64 int f(int x) {
65     return (123 * x * x + 4567890) % 1337;
66 }
67
68 void verify(int x0, int start, int length) {
69     set<int> s;
70     int x = x0;
71     for (int i = 0; i < start; i++) {
72         assert(!s.count(x));
73         s.insert(x);
74         x = f(x);
75     }
76     int startx = x;
77     s.clear();
78     for (int i = 0; i < length; i++) {
79         assert(!s.count(x));
80         s.insert(x);
81         x = f(x);
82     }
83     assert(startx == x);
84 }
85
86 int main () {
87     pair<int, int> res = find_cycle(f, x0);
88     assert(res == make_pair(4, 2));
89     verify(x0, res.first, res.second);
90     return 0;
91 }

```

1.4.2 Brent's Algorithm

```

1  /*
2
3  For a function f which maps a finite set S to itself and any initial
4  value x[0] in S, the same value must occur twice in the sequence below:
5
6      x[0], x[1] = f(x[0]), x[2] = f(x[1]), ..., x[i] = f(x[i - 1])
7
8  That is, there must exist numbers i, j (i < j) such that x[i] = x[j].
9  Once this happens, the sequence will continue periodically by repeating
10 the same sequence of values from x[i] to x[j - 1]. Cycle detection asks
11 to find i and j, given the function f and initial value x[0]. This is
12 also analogous to the problem of detecting a cycle in a linked list,
13 which will make it degenerate.
14
15 While Floyd's cycle-finding algorithm finds cycles by simultaneously
16 moving two pointers at different speeds, Brent's algorithm keeps the
17 tortoise pointer stationary and "teleports" it to the hare pointer
18 every power of two. The smallest power of two for which they meet is
19 the start of the first cycle. This improves upon the constant factor
20 of Floyd's algorithm by reducing the number of function calls.
21
22 Time Complexity: O(mu + lambda), where mu is the smallest index of the

```

23 sequence on which a cycle starts, and lambda is the cycle's length.

24

25 Space Complexity: $O(1)$ auxiliary.

26

27 */

28

29

30 #include <utility> /* std::pair */

31

32 template<class IntFunction>

33 std::pair<int, int> find_cycle(IntFunction f, int x0) {

34 int power = 1, length = 1;

35 int tortoise = x0, hare = f(x0);

36 while (tortoise != hare) {

37 if (power == length) {

38 tortoise = hare;

39 power *= 2;

40 length = 0;

41 }

42 hare = f(hare);

43 length++;

44 }

45 hare = x0;

46 for (int i = 0; i < length; i++)

47 hare = f(hare);

48 int start = 0;

49 tortoise = x0;

50 while (tortoise != hare) {

51 tortoise = f(tortoise);

52 hare = f(hare);

53 start++;

54 }

55 return std::make_pair(start, length);

56 }

57

58 /** Example Usage **/

59

60 #include <cassert>

61 #include <set>

62 using namespace std;

63

64 const int x0 = 0;

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     pair<int, int> res = find_cycle(f, x0);
90     assert(res == make_pair(4, 2));
91     verify(x0, res.first, res.second);
92     return 0;
93 }

```

1.5 Binary Exponentiation

```

1  /*
2
3  Given three positive, signed 64-bit integers, powmod() efficiently
4  computes the power of the first two integers, modulo the third integer.
5  Binary exponentiation, also known as "exponentiation by squaring,"
6  decomposes the computation with the observation that the exponent is
7  reduced by half whenever the base is squared. Odd-numbered exponents
8  can be dealt with by subtracting one and multiplying the overall
9  expression by the base of the power. This yields a logarithmic number
10 of multiplications while avoiding overflow. To further prevent overflow
11 in intermediate multiplications, multiplication can be done using the
12 similar principle of multiplication by adding. Despite using unsigned
13 64-bit integers for intermediate calculations and as parameter types,
14 each argument to powmod() must not exceed  $2^{63} - 1$ , the maximum value
15 of a signed 64-bit integer.
16
17 Time Complexity:  $O(\log n)$  on the exponent of the power.
18 Space Complexity:  $O(1)$  auxiliary.
19
20 */
21
22 typedef unsigned long long int64;
23
24 int64 mulmod(int64 a, int64 b, int64 m) {
25     int64 x = 0, y = a % m;
26     for (; b > 0; b >>= 1) {
27         if (b & 1)
28             x = (x + y) % m;
29         y = (y << 1) % m;
30     }
31     return x % m;
32 }
33
34 int64 powmod(int64 a, int64 b, int64 m) {
35     int64 x = 1, y = a;
36     for (; b > 0; b >>= 1) {
37         if (b & 1)
38             x = mulmod(x, y, m);
39         y = mulmod(y, y, m);
40     }
41     return x % m;

```

```
42 }
43
44 /** Example Usage **/
45
46 #include <cassert>
47
48 int main() {
49     assert(powmod(2, 10, 1000000007) == 1024);
50     assert(powmod(2, 62, 1000000) == 387904);
51     assert(powmod(10001, 10001, 100000) == 10001);
52     return 0;
53 }
```

Chapter 2

Graph Theory

2.1 Depth-First Search

2.1.1 Graph Class and Depth-First Search

```
1  /*
2
3  A graph can be represented as a set of objects (a.k.a. vertices, or
4  nodes) and connections (a.k.a. edges) between pairs of objects. It can
5  also be stored as an adjacency matrix or adjacency list, the latter of
6  which is more space efficient but less time efficient for particular
7  operations such as checking whether a connection exists. A fundamental
8  task to perform on graphs is traversal, where all reachable vertices
9  are visited and actions are performed. Given any arbitrary starting
10 node, depth-first search (DFS) recursively explores each "branch" from
11 the current node as deep as possible before backtracking and following
12 other branches. Depth-first search has many applications, including
13 detecting cycles and solving generic puzzles.
14
15 The following implements a simple graph class using adjacency lists,
16 along with with depth-first search and a few applications. The nodes of
17 the graph are identified by integers indices numbered consecutively
18 starting from 0. The total number nodes will automatically increase
19 based upon the maximum argument ever passed to add_edge().
20
21 Time Complexity:
22 - add_edge() is  $O(1)$  amortized per call, or  $O(n)$  for  $n$  calls where each
23   node index added is at most  $n$ .
24 - dfs(), has_cycle(), is_tree(), and is_dag() are each  $O(n)$  per call on
25   the number of edges added so far.
26 - All other public member functions are  $O(1)$ .
27
28 Space Complexity:
29 -  $O(n)$  to store a graph of  $n$  edges.
30 - dfs(), has_cycle(), is_tree(), and is_dag() each require  $O(n)$ 
31   auxiliary on the number of edges.
32 - All other public member functions require  $O(1)$  auxiliary.
33
34 */
35
```

```

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

```



```

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

```

```

154     g.add_edge(2, 3);
155     g.add_edge(2, 4);
156     g.add_edge(7, 8);
157     g.add_edge(7, 11);
158     g.add_edge(8, 9);
159     g.add_edge(8, 10);
160     cout << "DFS_order:␣";
161     g.dfs(0, print_node);
162     cout << endl;
163     assert(g[0].size() == 3);
164     assert(!g.has_cycle());
165 }
166 {
167     graph tree(false);
168     tree.add_edge(0, 1);
169     tree.add_edge(0, 2);
170     tree.add_edge(1, 3);
171     tree.add_edge(1, 4);
172     assert(tree.is_tree());
173     tree.add_edge(2, 3);
174     assert(!tree.is_tree());
175 }
176 return 0;
177 }

```

2.1.2 Topological Sorting

```

1  /*
2
3  Description: Given a directed acyclic graph (DAG), order the nodes
4  such that for every edge from a to b, a precedes b in the ordering.
5  Usually, there is more than one possible valid ordering. The
6  following program uses DFS to produce one possible ordering.
7  This can also be used to detect whether the graph is a DAG.
8  Note that the DFS algorithm here produces a reversed topological
9  ordering, so the output must be printed backwards. The graph is
10 stored in an adjacency list.
11
12 Complexity:  $O(V+E)$  on the number of vertices and edges.
13
14 ~=~=~= Sample Input ~=~=~=
15 8 9
16 0 3
17 0 4
18 1 3
19 2 4
20 2 7
21 3 5
22 3 6
23 3 7
24 4 6
25
26 ~=~=~= Sample Output ~=~=~=
27 The topological order: 2 1 0 4 3 7 6 5
28
29 */
30

```

```

31 #include <algorithm> /* std::fill(), std::reverse() */
32 #include <iostream>
33 #include <stdexcept> /* std::runtime_error() */
34 #include <vector>
35 using namespace std;
36
37 const int MAXN = 100;
38 vector<bool> vis(MAXN), done(MAXN);
39 vector<int> adj[MAXN], sorted;
40
41 void dfs(int u) {
42     if (vis[u])
43         throw std::runtime_error("Not a DAG.");
44     if (done[u]) return;
45     vis[u] = true;
46     for (int j = 0; j < (int)adj[u].size(); j++)
47         dfs(adj[u][j]);
48     vis[u] = false;
49     done[u] = true;
50     sorted.push_back(u);
51 }
52
53 void toposort(int nodes) {
54     fill(vis.begin(), vis.end(), false);
55     fill(done.begin(), done.end(), false);
56     sorted.clear();
57     for (int i = 0; i < nodes; i++)
58         if (!done[i]) dfs(i);
59     reverse(sorted.begin(), sorted.end());
60 }
61
62 int main() {
63     int nodes, edges, u, v;
64     cin >> nodes >> edges;
65     for (int i = 0; i < edges; i++) {
66         cin >> u >> v;
67         adj[u].push_back(v);
68     }
69     toposort(nodes);
70     cout << "The topological order:";
71     for (int i = 0; i < (int)sorted.size(); i++)
72         cout << " " << sorted[i];
73     cout << "\n";
74     return 0;
75 }

```

2.1.3 Eulerian Cycles

```

1  /*
2
3  Description: A Eulerian trail is a trail in a graph which
4  visits every edge exactly once. Similarly, an Eulerian circuit
5  or Eulerian cycle is an Eulerian trail which starts and ends
6  on the same vertex.
7
8  An undirected graph has an Eulerian cycle if and only if every
9  vertex has even degree, and all of its vertices with nonzero

```

```

10 degree belong to a single connected component.
11
12 A directed graph has an Eulerian cycle if and only if every
13 vertex has equal in degree and out degree, and all of its
14 vertices with nonzero degree belong to a single strongly
15 connected component.
16
17 Complexity:  $O(V+E)$  on the number of vertices and edges.
18
19 ~=~=~= Sample Input ~=~=~=
20 5 6
21 0 1
22 1 2
23 2 0
24 1 3
25 3 4
26 4 1
27
28 ~=~=~= Sample Output ~=~=~=
29 Eulerian cycle from 0 (directed): 0 1 3 4 1 2 0
30 Eulerian cycle from 2 (undirected): 2 1 3 4 1 0 2
31
32 */
33
34 #include <algorithm> /* std::reverse() */
35 #include <iostream>
36 #include <vector>
37 using namespace std;
38
39 const int MAXN = 100;
40
41 vector<int> euler_cycle_directed(vector<int> adj[], int u) {
42     vector<int> stack, res, cur_edge(MAXN);
43     stack.push_back(u);
44     while (!stack.empty()) {
45         u = stack.back();
46         stack.pop_back();
47         while (cur_edge[u] < (int)adj[u].size()) {
48             stack.push_back(u);
49             u = adj[u][cur_edge[u]++];
50         }
51         res.push_back(u);
52     }
53     reverse(res.begin(), res.end());
54     return res;
55 }
56
57 vector<int> euler_cycle_undirected(vector<int> adj[], int u) {
58     vector<vector<bool>> > used(MAXN, vector<bool>(MAXN, false));
59     vector<int> stack, res, cur_edge(MAXN);
60     stack.push_back(u);
61     while (!stack.empty()) {
62         u = stack.back();
63         stack.pop_back();
64         while (cur_edge[u] < (int)adj[u].size()) {
65             int v = adj[u][cur_edge[u]++];
66             if (!used[min(u, v)][max(u, v)]) {
67                 used[min(u, v)][max(u, v)] = 1;
68                 stack.push_back(u);

```

```

69     u = v;
70 }
71 }
72 res.push_back(u);
73 }
74 reverse(res.begin(), res.end());
75 return res;
76 }
77
78 int main() {
79     int nodes, edges, u, v;
80     vector<int> g1[5], g2[5], cycle;
81
82     cin >> nodes >> edges;
83     for (int i = 0; i < edges; i++) {
84         cin >> u >> v;
85         g1[u].push_back(v);
86         g2[u].push_back(v);
87         g2[v].push_back(u);
88     }
89
90     cycle = euler_cycle_directed(g1, 0);
91     cout << "Eulerian cycle from 0 (directed):";
92     for (int i = 0; i < (int)cycle.size(); i++)
93         cout << " " << cycle[i];
94     cout << "\n";
95
96     cycle = euler_cycle_undirected(g2, 2);
97     cout << "Eulerian cycle from 2 (undirected):";
98     for (int i = 0; i < (int)cycle.size(); i++)
99         cout << " " << cycle[i];
100    cout << "\n";
101    return 0;
102 }

```

2.1.4 Unweighted Tree Centers

```

1  /*
2
3  The following applies to unweighted, undirected trees only.
4
5  find_centers(): Returns 1 or 2 tree centers. The center
6  (or Jordan center) of a graph is the set of all vertices of
7  minimum eccentricity, that is, the set of all vertices A
8  where the max distance d(A,B) to other vertices B is minimal.
9
10 find_centroid(): Returns a vertex where all of its subtrees
11 have size <= N/2, where N is the number of nodes in the tree.
12
13 diameter(): The diameter of a tree is the greatest distance
14 d(A,B) between any two of the nodes in the tree.
15
16 Complexity: All three functions are O(V) on the number of
17 vertices in the tree.
18
19 ~=~=~= Sample Input ~=~=~=
20 6

```

```

21 0 1
22 1 2
23 1 4
24 3 4
25 4 5
26
27 ~=~=~= Sample Output ~=~=~=
28 Center(s): 1 4
29 Centroid: 4
30 Diameter: 3
31
32 */
33
34 #include <iostream>
35 #include <vector>
36 using namespace std;
37
38 const int MAXN = 100;
39 vector<int> adj[MAXN];
40
41 vector<int> find_centers(int n) {
42     vector<int> leaves, degree(n);
43     for (int i = 0; i < n; i++) {
44         degree[i] = adj[i].size();
45         if (degree[i] <= 1) leaves.push_back(i);
46     }
47     int removed = leaves.size();
48     while (removed < n) {
49         vector<int> nleaves;
50         for (int i = 0; i < (int)leaves.size(); i++) {
51             int u = leaves[i];
52             for (int j = 0; j < (int)adj[u].size(); j++) {
53                 int v = adj[u][j];
54                 if (--degree[v] == 1)
55                     nleaves.push_back(v);
56             }
57         }
58         leaves = nleaves;
59         removed += leaves.size();
60     }
61     return leaves;
62 }
63
64 int find_centroid(int n, int u = 0, int p = -1) {
65     int cnt = 1, v;
66     bool good_center = true;
67     for (int j = 0; j < (int)adj[u].size(); j++) {
68         if ((v = adj[u][j]) == p) continue;
69         int res = find_centroid(n, v, u);
70         if (res >= 0) return res;
71         int size = -res;
72         good_center &= (size <= n / 2);
73         cnt += size;
74     }
75     good_center &= (n - cnt <= n / 2);
76     return good_center ? u : -cnt;
77 }
78
79 pair<int, int> dfs(int u, int p, int depth) {

```

```

80     pair<int, int> res = 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     return res;
85 }
86
87 int diameter() {
88     int furthest_vertex = dfs(0, -1, 0).second;
89     return dfs(furthest_vertex, -1, 0).first;
90 }
91
92 int main() {
93     int nodes, u, v;
94     cin >> nodes;
95     for (int i = 0; i < nodes - 1; i++) {
96         cin >> u >> v;
97         adj[u].push_back(v);
98         adj[v].push_back(u);
99     }
100     vector<int> centers = find_centers(nodes);
101     cout << "Center(s):";
102     for (int i = 0; i < (int)centers.size(); i++)
103         cout << "□" << centers[i];
104     cout << "\nCentroid:□" << find_centroid(nodes);
105     cout << "\nDiameter:□" << diameter() << "\n";
106     return 0;
107 }

```

2.2 Shortest Paths

2.2.1 Breadth First Search

```

1  /*
2
3  Description: Given an unweighted graph, traverse all reachable
4  nodes from a source node and determine the shortest path.
5
6  Complexity:  $O(V+E)$  on the number of vertices and edges.
7
8  Note: The line "for (q.push(start); !q.empty(); q.pop())"
9  is simply a mnemonic for looping a BFS with a FIFO queue.
10 This will not work as intended with a priority queue, such as in
11 Dijkstra's algorithm for solving weighted shortest paths
12
13 ~=~=~= Sample Input ~=~=~=
14 4 5
15 0 1
16 0 3
17 1 2
18 1 3
19 2 3
20 0 3
21
22 ~=~=~= Sample Output ~=~=~=
23 The shortest distance from 0 to 3 is 2.

```

```

24 Take the path: 0->1->3.
25
26 */
27
28 #include <iostream>
29 #include <queue>
30 #include <vector>
31 using namespace std;
32
33 const int MAXN = 100, INF = 0x3f3f3f3f;
34 int dist[MAXN], pred[MAXN];
35 vector<int> adj[MAXN];
36
37 void bfs(int nodes, int start) {
38     vector<bool> vis(nodes, false);
39     for (int i = 0; i < nodes; i++) {
40         dist[i] = INF;
41         pred[i] = -1;
42     }
43     int u, v, d;
44     queue<pair<int, int> > q;
45     q.push(make_pair(start, 0));
46     while (!q.empty()) {
47         u = q.front().first;
48         d = q.front().second;
49         q.pop();
50         vis[u] = true;
51         for (int j = 0; j < (int)adj[u].size(); j++) {
52             if (vis[v = adj[u][j]]) continue;
53             dist[v] = d + 1;
54             pred[v] = u;
55             q.push(make_pair(v, d + 1));
56         }
57     }
58 }
59
60 //Use the precomputed pred[] array to print the path
61 void print_path(int dest) {
62     int i = 0, j = dest, path[MAXN];
63     while (pred[j] != -1) j = path[++i] = pred[j];
64     cout << "Take the path:";
65     while (i > 0) cout << path[i--] << "->";
66     cout << dest << ".\n";
67 }
68
69 int main() {
70     int nodes, edges, u, v, start, dest;
71     cin >> nodes >> edges;
72     for (int i = 0; i < edges; i++) {
73         cin >> u >> v;
74         adj[u].push_back(v);
75     }
76     cin >> start >> dest;
77     bfs(nodes, start);
78     cout << "The shortest distance from" << start;
79     cout << " to" << dest << " is" << dist[dest] << ".\n";
80     print_path(dest);
81     return 0;
82 }

```


2.2.2 Dijkstra's Algorithm

```

1  /*
2
3  Description: Given a directed graph with positive weights only, find
4  the shortest distance to all nodes from a single starting node.
5
6  Implementation Notes: The graph is stored using an adjacency list.
7  This implementation negates distances before adding them to the
8  priority queue, since the container is a max-heap by default. This
9  method is suggested in contests because it is easier than defining
10 special comparators. An alternative would be declaring the queue
11 with template parameters (clearly, this way is very verbose and ugly):
12     priority_queue< pair<int, int>, vector<pair<int, int> >,
13         greater<pair<int, int> > > pq;
14 If only the path between a single pair of nodes is needed, for speed,
15 we may break out of the loop as soon as the destination is reached
16 by inserting the line "if (a == dest) break;" after the line "pq.pop();"
17
18 Complexity: This version uses an adjacency list and priority queue
19 (internally a binary heap) and has a complexity of  $O((E+V) \log V) =$ 
20  $O(E \log V)$ . The priority queue and adjacency list improves the
21 simplest  $O(V^2)$  version of the algorithm, which uses looping and
22 an adjacency matrix. If the priority queue is implemented as a more
23 sophisticated Fibonacci heap, the complexity becomes  $O(E + V \log V)$ .
24
25 Modification to Shortest Path Faster Algorithm: The code for Dijkstra's
26 algorithm here can be easily modified to become the Shortest Path Faster
27 Algorithm (SPFA) by simply commenting out "visit[a] = true;" and changing
28 the priority queue to a FIFO queue like in BFS. SPFA is a faster version
29 of the Bellman-Ford algorithm, working on negative path lengths (whereas
30 Dijkstra's cannot). Certain graphs can be constructed to make SPFA slow.
31
32 ~=~=~= Sample Input ~=~=~=
33 4 5
34 0 1 2
35 0 3 8
36 1 2 2
37 1 3 4
38 2 3 1
39 0 3
40
41 ~=~=~= Sample Output ~=~=~=
42 The shortest distance from 0 to 3 is 5.
43 Take the path: 0->1->2->3.
44
45 */
46
47 #include <iostream>
48 #include <queue>
49 #include <vector>
50 using namespace std;
51
52 const int MAXN = 100, INF = 0x3f3f3f3f;
53 int dist[MAXN], pred[MAXN];
54 vector<pair<int, int> > adj[MAXN];
55
56 void dijkstra(int nodes, int start) {

```

```

57  vector<bool> vis(nodes, false);
58  for (int i = 0; i < nodes; i++) {
59      dist[i] = INF;
60      pred[i] = -1;
61  }
62  int u, v;
63  dist[start] = 0;
64  priority_queue<pair<int, int> > pq;
65  pq.push(make_pair(0, start));
66  while (!pq.empty()) {
67      u = pq.top().second;
68      pq.pop();
69      vis[u] = true;
70      for (int j = 0; j < (int)adj[u].size(); j++) {
71          if (vis[v = adj[u][j].first]) continue;
72          if (dist[v] > dist[u] + adj[u][j].second) {
73              dist[v] = dist[u] + adj[u][j].second;
74              pred[v] = u;
75              pq.push(make_pair(-dist[v], v));
76          }
77      }
78  }
79 }
80
81 //Use the precomputed pred[] array to print the path
82 void print_path(int dest) {
83     int i = 0, j = dest, path[MAXN];
84     while (pred[j] != -1) j = path[++i] = pred[j];
85     cout << "Take the path: ";
86     while (i > 0) cout << path[i--] << "->";
87     cout << dest << ".\n";
88 }
89
90 int main() {
91     int nodes, edges, u, v, w, start, dest;
92     cin >> nodes >> edges;
93     for (int i = 0; i < edges; i++) {
94         cin >> u >> v >> w;
95         adj[u].push_back(make_pair(v, w));
96     }
97     cin >> start >> dest;
98     dijkstra(nodes, start);
99     cout << "The shortest distance from " << start;
100    cout << " to " << dest << " is " << dist[dest] << ".\n";
101    print_path(dest);
102    return 0;
103 }

```

2.2.3 Bellman-Ford Algorithm

```

1  /*
2
3  Description: Given a directed graph with positive or negative weights
4  but no negative cycles, find the shortest distance to all nodes from
5  a single starting node. The input graph is stored using an edge list.
6
7  Complexity:  $O(V \cdot E)$  on the number of vertices and edges, respectively.

```

```

8
9  ~=~=~= Sample Input ~=~=~=
10 3 3
11 0 1 1
12 1 2 2
13 0 2 5
14 0 2
15
16 ~=~=~= Sample Output ~=~=~=
17 The shortest distance from 0 to 2 is 3.
18 Take the path: 0->1->2.
19
20 */
21
22 #include <iostream>
23 #include <stdexcept>
24 #include <vector>
25 using namespace std;
26
27 struct edge { int u, v, w; };
28
29 const int MAXN = 100, INF = 0x3f3f3f3f;
30 int dist[MAXN], pred[MAXN];
31 vector<edge> e;
32
33 void bellman_ford(int nodes, int start) {
34     for (int i = 0; i < nodes; i++) {
35         dist[i] = INF;
36         pred[i] = -1;
37     }
38     dist[start] = 0;
39     for (int i = 0; i < nodes; i++) {
40         for (int j = 0; j < (int)e.size(); j++) {
41             if (dist[e[j].v] > dist[e[j].u] + e[j].w) {
42                 dist[e[j].v] = dist[e[j].u] + e[j].w;
43                 pred[e[j].v] = e[j].u;
44             }
45         }
46     }
47     //optional: report negative-weight cycles
48     for (int i = 0; i < (int)e.size(); i++)
49         if (dist[e[i].v] > dist[e[i].u] + e[i].w)
50             throw std::runtime_error("Negative-weight found");
51 }
52
53 //Use the precomputed pred[] array to print the path
54 void print_path(int dest) {
55     int i = 0, j = dest, path[MAXN];
56     while (pred[j] != -1) j = path[++i] = pred[j];
57     cout << "Take the path: ";
58     while (i > 0) cout << path[i--] << "->";
59     cout << dest << ".\n";
60 }
61
62 int main() {
63     int nodes, edges, u, v, w, start, dest;
64     cin >> nodes >> edges;
65     for (int i = 0; i < edges; i++) {
66         cin >> u >> v >> w;

```

```

67     e.push_back((edge){u, v, w});
68 }
69 cin >> start >> dest;
70 bellman_ford(nodes, start);
71 cout << "The shortest distance from " << start;
72 cout << " to " << dest << " is " << dist[dest] << ".\n";
73 print_path(dest);
74 return 0;
75 }

```

2.2.4 Floyd-Warshall Algorithm

```

1  /*
2
3  Description: Given a directed graph with positive or negative
4  weights but no negative cycles, find the shortest distance
5  between all pairs of nodes. The input graph is stored using
6  an adjacency matrix. Note that the input adjacency matrix
7  is converted to the distance matrix afterwards. If you still
8  need the adjacencies afterwards, back it up at the beginning.
9
10 Complexity:  $O(V^3)$  on the number of vertices.
11
12 ~=~=~= Sample Input ~=~=~=
13 3 3
14 0 1 1
15 1 2 2
16 0 2 5
17 0 2
18
19 ~=~=~= Sample Output ~=~=~=
20 The shortest distance from 0 to 2 is 3.
21 Take the path: 0->1->2.
22
23 */
24
25 #include <iostream>
26 using namespace std;
27
28 const int MAXN = 100, INF = 0x3f3f3f3f;
29 int dist[MAXN][MAXN], next[MAXN][MAXN];
30
31 void initialize(int nodes) {
32     for (int i = 0; i < nodes; i++)
33         for (int j = 0; j < nodes; j++) {
34             dist[i][j] = (i == j) ? 0 : INF;
35             next[i][j] = -1;
36         }
37 }
38
39 void floyd_warshall(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                     next[i][j] = k;

```

```

46     }
47 }
48
49 void print_path(int u, int v) {
50     if (next[u][v] != -1) {
51         print_path(u, next[u][v]);
52         cout << next[u][v];
53         print_path(next[u][v], v);
54     } else cout << "->";
55 }
56
57 int main() {
58     int nodes, edges, u, v, w, start, dest;
59     cin >> nodes >> edges;
60     initialize(nodes);
61     for (int i = 0; i < edges; i++) {
62         cin >> u >> v >> w;
63         dist[u][v] = w;
64     }
65     cin >> start >> dest;
66     floyd_warshall(nodes);
67     cout << "The shortest distance from " << start;
68     cout << " to " << dest << " is ";
69     cout << dist[start][dest] << ".\n";
70
71     //Use next[][] to recursively print the path
72     cout << "Take the path " << start;
73     print_path(start, dest);
74     cout << dest << ".\n";
75     return 0;
76 }

```

2.3 Connectivity

2.3.1 Strongly Connected Components (Kosaraju's Algorithm)

```

1  /*
2
3  Description: Determines the strongly connected components (SCC)
4  from a given directed graph. Given a directed graph, its SCCs
5  are its maximal strongly connected sub-graphs. A graph is
6  strongly connected if there is a path from each node to every
7  other node. Condensing the strongly connected components of a
8  graph into single nodes will result in a directed acyclic graph.
9  The input is stored in an adjacency list.
10
11 Complexity:  $O(V+E)$  on the number of vertices and edges.
12
13 Comparison with other SCC algorithms:
14 The strongly connected components of a graph can be efficiently
15 computed using Kosaraju's algorithm, Tarjan's algorithm, or the
16 path-based strong component algorithm. Tarjan's algorithm can
17 be seen as an improved version of Kosaraju's because it performs
18 a single DFS rather than two. Though they both have the same
19 complexity, Tarjan's algorithm is much more efficient in
20 practice. However, Kosaraju's algorithm is conceptually simpler.

```

```

21
22  ~=~=~= Sample Input ~=~=~=
23  8 14
24  0 1
25  1 2
26  1 4
27  1 5
28  2 3
29  2 6
30  3 2
31  3 7
32  4 0
33  4 5
34  5 6
35  6 5
36  7 3
37  7 6
38
39  ~=~=~= Sample Output ~=~=~=
40  Component: 1 4 0
41  Component: 7 3 2
42  Component: 5 6
43
44  */
45
46  #include <algorithm> /* std::fill(), std::reverse() */
47  #include <iostream>
48  #include <vector>
49  using namespace std;
50
51  const int MAXN = 100;
52  vector<bool> vis(MAXN);
53  vector<int> adj[MAXN], order;
54  vector<vector<int>> > scc;
55
56  void dfs(vector<int> graph[], vector<int> & res, int u) {
57      vis[u] = true;
58      for (int j = 0; j < (int)graph[u].size(); j++)
59          if (!vis[graph[u][j]])
60              dfs(graph, res, graph[u][j]);
61      res.push_back(u);
62  }
63
64  void kosaraju(int nodes) {
65      scc.clear();
66      order.clear();
67      vector<int> rev[nodes];
68      fill(vis.begin(), vis.end(), false);
69      for (int i = 0; i < nodes; i++)
70          if (!vis[i]) dfs(adj, order, i);
71      for (int i = 0; i < nodes; i++)
72          for (int j = 0; j < (int)adj[i].size(); j++)
73              rev[adj[i][j]].push_back(i);
74      fill(vis.begin(), vis.end(), false);
75      reverse(order.begin(), order.end());
76      for (int i = 0; i < (int)order.size(); i++) {
77          if (vis[order[i]]) continue;
78          vector<int> component;
79          dfs(rev, component, order[i]);

```

```

80     scc.push_back(component);
81 }
82 }
83
84 int main() {
85     int nodes, edges, u, v;
86     cin >> nodes >> edges;
87     for (int i = 0; i < edges; i++) {
88         cin >> u >> v;
89         adj[u].push_back(v);
90     }
91     kosaraju(nodes);
92     for (int i = 0; i < (int)scc.size(); i++) {
93         cout << "Component:";
94         for (int j = 0; j < (int)scc[i].size(); j++)
95             cout << "□" << scc[i][j];
96         cout << "\n";
97     }
98     return 0;
99 }

```

2.3.2 Strongly Connected Components (Tarjan's Algorithm)

```

1  /*
2
3  Description: Determines the strongly connected components (SCC)
4  from a given directed graph. Given a directed graph, its SCCs
5  are its maximal strongly connected sub-graphs. A graph is
6  strongly connected if there is a path from each node to every
7  other node. Condensing the strongly connected components of a
8  graph into single nodes will result in a directed acyclic graph.
9  The input is stored in an adjacency list.
10
11  In this implementation, a vector is used to emulate a stack
12  for the sake of simplicity. One useful property of Tarjan's
13  algorithm is that, while there is nothing special about the
14  ordering of nodes within each component, the resulting DAG
15  is produced in reverse topological order.
16
17  Complexity:  $O(V+E)$  on the number of vertices and edges.
18
19  Comparison with other SCC algorithms:
20  The strongly connected components of a graph can be efficiently
21  computed using Kosaraju's algorithm, Tarjan's algorithm, or the
22  path-based strong component algorithm. Tarjan's algorithm can
23  be seen as an improved version of Kosaraju's because it performs
24  a single DFS rather than two. Though they both have the same
25  complexity, Tarjan's algorithm is much more efficient in
26  practice. However, Kosaraju's algorithm is conceptually simpler.
27
28  ~=~=~= Sample Input ~=~=~=
29  8 14
30  0 1
31  1 2
32  1 4
33  1 5
34  2 3

```

```

35  2 6
36  3 2
37  3 7
38  4 0
39  4 5
40  5 6
41  6 5
42  7 3
43  7 6
44
45  ==~=~=~= Sample Output ==~=~=~=
46  Component 1: 5 6
47  Component 2: 7 3 2
48  Component 3: 4 1 0
49
50  */
51
52  #include <algorithm> /* std::fill() */
53  #include <iostream>
54  #include <vector>
55  using namespace std;
56
57  const int MAXN = 100, INF = 0x3f3f3f3f;
58  int timer, lowlink[MAXN];
59  vector<bool> vis(MAXN);
60  vector<int> adj[MAXN], stack;
61  vector<vector<int>> > scc;
62
63  void dfs(int u) {
64      lowlink[u] = timer++;
65      vis[u] = true;
66      stack.push_back(u);
67      bool is_component_root = true;
68      int v;
69      for (int j = 0; j < (int)adj[u].size(); j++) {
70          if (!vis[v = adj[u][j]]) dfs(v);
71          if (lowlink[u] > lowlink[v]) {
72              lowlink[u] = lowlink[v];
73              is_component_root = false;
74          }
75      }
76      if (!is_component_root) return;
77      vector<int> component;
78      do {
79          vis[v = stack.back()] = true;
80          stack.pop_back();
81          lowlink[v] = INF;
82          component.push_back(v);
83      } while (u != v);
84      scc.push_back(component);
85  }
86
87  void tarjan(int nodes) {
88      scc.clear();
89      stack.clear();
90      fill(lowlink, lowlink + nodes, 0);
91      fill(vis.begin(), vis.end(), false);
92      timer = 0;
93      for (int i = 0; i < nodes; i++)

```



```

94     if (!vis[i]) dfs(i);
95 }
96
97 int main() {
98     int nodes, edges, u, v;
99     cin >> nodes >> edges;
100     for (int i = 0; i < edges; i++) {
101         cin >> u >> v;
102         adj[u].push_back(v);
103     }
104     tarjan(nodes);
105     for (int i = 0; i < (int)scc.size(); i++) {
106         cout << "Component:";
107         for (int j = 0; j < (int)scc[i].size(); j++)
108             cout << "□" << scc[i][j];
109         cout << "\n";
110     }
111     return 0;
112 }

```

2.3.3 Bridges, Cut-points, and Biconnectivity

```

1  /*
2
3  Description: The following operations apply to undirected graphs.
4
5  A bridge is an edge, when deleted, increases the number of
6  connected components. An edge is a bridge if and only if it is not
7  contained in any cycle.
8
9  A cut-point (i.e. cut-vertex or articulation point) is any vertex
10 whose removal increases the number of connected components.
11
12 A biconnected component of a graph is a maximally biconnected
13 subgraph. A biconnected graph is a connected and "nonseparable"
14 graph, meaning that if any vertex were to be removed, the graph
15 will remain connected. Therefore, a biconnected graph has no
16 articulation vertices.
17
18 Any connected graph decomposes into a tree of biconnected
19 components called the "block tree" of the graph. An unconnected
20 graph will thus decompose into a "block forest."
21
22 See: http://en.wikipedia.org/wiki/Biconnected\_component
23
24 Complexity:  $O(V+E)$  on the number of vertices and edges.
25
26 ~=~=~= Sample Input ~=~=~=
27 8 6
28 0 1
29 0 5
30 1 2
31 1 5
32 3 7
33 4 5
34
35 ~=~=~= Sample Output ~=~=~=

```

```

36 Cut Points: 5 1
37 Bridges:
38 1 2
39 5 4
40 3 7
41 Edge-Biconnected Components:
42 Component 1: 2
43 Component 2: 4
44 Component 3: 5 1 0
45 Component 4: 7
46 Component 5: 3
47 Component 6: 6
48 Adjacency List for Block Forest:
49 0 => 2
50 1 => 2
51 2 => 0 1
52 3 => 4
53 4 => 3
54 5 =>
55
56 */
57
58 #include <algorithm> /* std::fill(), std::min() */
59 #include <iostream>
60 #include <vector>
61 using namespace std;
62
63 const int MAXN = 100;
64 int timer, lowlink[MAXN], tin[MAXN], comp[MAXN];
65 vector<bool> vis(MAXN);
66 vector<int> adj[MAXN], bcc_forest[MAXN];
67 vector<int> stack, cutpoints;
68 vector<vector<int>> > bcc;
69 vector<pair<int, int>> > bridges;
70
71 void dfs(int u, int p) {
72     vis[u] = true;
73     lowlink[u] = tin[u] = timer++;
74     stack.push_back(u);
75     int v, children = 0;
76     bool cutpoint = false;
77     for (int j = 0; j < (int)adj[u].size(); j++) {
78         if ((v = adj[u][j]) == p) continue;
79         if (vis[v]) {
80             //lowlink[u] = min(lowlink[u], lowlink[v]);
81             lowlink[u] = min(lowlink[u], tin[v]);
82         } else {
83             dfs(v, u);
84             lowlink[u] = min(lowlink[u], lowlink[v]);
85             cutpoint |= (lowlink[v] >= tin[u]);
86             if (lowlink[v] > tin[u])
87                 bridges.push_back(make_pair(u, v));
88             children++;
89         }
90     }
91     if (p == -1) cutpoint = (children >= 2);
92     if (cutpoint) cutpoints.push_back(u);
93     if (lowlink[u] == tin[u]) {
94         vector<int> component;

```

```

95     do {
96         v = stack.back();
97         stack.pop_back();
98         component.push_back(v);
99     } while (u != v);
100     bcc.push_back(component);
101 }
102 }
103
104 void tarjan(int nodes) {
105     bcc.clear();
106     bridges.clear();
107     cutpoints.clear();
108     stack.clear();
109     fill(lowlink, lowlink + nodes, 0);
110     fill(tin, tin + nodes, 0);
111     fill(vis.begin(), vis.end(), false);
112     timer = 0;
113     for (int i = 0; i < nodes; i++)
114         if (!vis[i]) dfs(i, -1);
115 }
116
117 //condenses each bcc to a node and generates a tree
118 //global variables adj and bcc must be set beforehand
119 void get_block_tree(int nodes) {
120     fill(comp, comp + nodes, 0);
121     for (int i = 0; i < nodes; i++) bcc_forest[i].clear();
122     for (int i = 0; i < (int)bcc.size(); i++)
123         for (int j = 0; j < (int)bcc[i].size(); j++)
124             comp[bcc[i][j]] = i;
125     for (int i = 0; i < nodes; i++)
126         for (int j = 0; j < (int)adj[i].size(); j++)
127             if (comp[i] != comp[adj[i][j]])
128                 bcc_forest[comp[i]].push_back(comp[adj[i][j]]);
129 }
130
131 int main() {
132     int nodes, edges, u, v;
133     cin >> nodes >> edges;
134     for (int i = 0; i < edges; i++) {
135         cin >> u >> v;
136         adj[u].push_back(v);
137         adj[v].push_back(u);
138     }
139     tarjan(nodes);
140     cout << "Cut-points:";
141     for (int i = 0; i < (int)cutpoints.size(); i++)
142         cout << "□" << cutpoints[i];
143     cout << "\nBridges:\n";
144     for (int i = 0; i < (int)bridges.size(); i++)
145         cout << bridges[i].first << "□" << bridges[i].second << "\n";
146     cout << "Edge-Biconnected Components:\n";
147     for (int i = 0; i < (int)bcc.size(); i++) {
148         cout << "Component:";
149         for (int j = 0; j < (int)bcc[i].size(); j++)
150             cout << "□" << bcc[i][j];
151         cout << "\n";
152     }
153     get_block_tree(nodes);

```

```

154     cout << "Adjacency_List_for_Block_Forest:\n";
155     for (int i = 0; i < (int)bcc.size(); i++) {
156         cout << i << "=>";
157         for (int j = 0; j < (int)bcc_forest[i].size(); j++)
158             cout << " " << bcc_forest[i][j];
159         cout << "\n";
160     }
161     return 0;
162 }

```

2.4 Minimal Spanning Trees

2.4.1 Prim's Algorithm

```

1  /*
2
3  Description: Given an undirected graph, its minimum spanning
4  tree (MST) is a tree connecting all nodes with a subset of its
5  edges such that their total weight is minimized. Prim's algorithm
6  greedily selects edges from a priority queue, and is similar to
7  Dijkstra's algorithm, where instead of processing nodes, we
8  process individual edges. If the graph is not connected, Prim's
9  algorithm will produce the minimum spanning forest. The input
10 graph is stored in an adjacency list.
11
12 Note that the concept of the minimum spanning tree makes Prim's
13 algorithm work with negative weights. In fact, a big positive
14 constant added to all of the edge weights of the graph will not
15 change the resulting spanning tree.
16
17 Implementation Notes: Similar to the implementation of Dijkstra's
18 algorithm in the previous section, weights are negated before they
19 are added to the priority queue (and negated once again when they
20 are retrieved). To find the maximum spanning tree, simply skip the
21 two negation steps and the max weighted edges will be prioritized.
22
23 Complexity: This version uses an adjacency list and priority queue
24 (internally a binary heap) and has a complexity of  $O((E+V) \log V) =$ 
25  $O(E \log V)$ . The priority queue and adjacency list improves the
26 simplest  $O(V^2)$  version of the algorithm, which uses looping and
27 an adjacency matrix. If the priority queue is implemented as a more
28 sophisticated Fibonacci heap, the complexity becomes  $O(E + V \log V)$ .
29
30 ~=~=~= Sample Input ~=~=~=
31 7 7
32 0 1 4
33 1 2 6
34 2 0 3
35 3 4 1
36 4 5 2
37 5 6 3
38 6 4 4
39
40 ~=~=~= Sample Output ~=~=~=
41 Total distance: 13
42 0<->2

```

```

43 0<->1
44 3<->4
45 4<->5
46 5<->6
47
48 */
49
50 #include <algorithm> /* std::fill() */
51 #include <iostream>
52 #include <queue>
53 #include <vector>
54 using namespace std;
55
56 const int MAXN = 100;
57 vector<pair<int, int> > adj[MAXN], mst;
58
59 int prim(int nodes) {
60     mst.clear();
61     vector<bool> vis(nodes);
62     int u, v, w, total_dist = 0;
63     for (int i = 0; i < nodes; i++) {
64         if (vis[i]) continue;
65         vis[i] = true;
66         priority_queue<pair<int, pair<int, int> > > pq;
67         for (int j = 0; j < (int)adj[i].size(); j++)
68             pq.push(make_pair(-adj[i][j].second,
69                             make_pair(i, adj[i][j].first)));
70         while (!pq.empty()) {
71             w = -pq.top().first;
72             u = pq.top().second.first;
73             v = pq.top().second.second;
74             pq.pop();
75             if (vis[u] && !vis[v]) {
76                 vis[v] = true;
77                 if (v != i) {
78                     mst.push_back(make_pair(u, v));
79                     total_dist += w;
80                 }
81                 for (int j = 0; j < (int)adj[v].size(); j++)
82                     pq.push(make_pair(-adj[v][j].second,
83                                         make_pair(v, adj[v][j].first)));
84             }
85         }
86     }
87     return total_dist;
88 }
89
90 int main() {
91     int nodes, edges, u, v, w;
92     cin >> nodes >> edges;
93     for (int i = 0; i < edges; i++) {
94         cin >> u >> v >> w;
95         adj[u].push_back(make_pair(v, w));
96         adj[v].push_back(make_pair(u, w));
97     }
98     cout << "Total_distance:_" << prim(nodes) << "\n";
99     for (int i = 0; i < (int)mst.size(); i++)
100         cout << mst[i].first << "<->" << mst[i].second << "\n";
101     return 0;

```

102 }

2.4.2 Kruskal's Algorithm

```

1  /*
2
3  Description: Given an undirected graph, its minimum spanning
4  tree (MST) is a tree connecting all nodes with a subset of its
5  edges such that their total weight is minimized. If the graph
6  is not connected, Kruskal's algorithm will produce the minimum
7  spanning forest. The input graph is stored in an edge list.
8
9  Complexity:  $O(E \log V)$  on the number of edges and vertices.
10
11  ~=~=~=~= Sample Input ~=~=~=~=
12  7 7
13  0 1 4
14  1 2 6
15  2 0 3
16  3 4 1
17  4 5 2
18  5 6 3
19  6 4 4
20
21  ~=~=~=~= Sample Output ~=~=~=~=
22  Total distance: 13
23  3<->4
24  4<->5
25  2<->0
26  5<->6
27  0<->1
28
29  Note: If you already have a disjoint set data structure,
30  then the middle section of the program can be replaced by:
31
32  disjoint_set_forest<int> dsf;
33  for (int i = 0; i < nodes; i++) dsf.make_set(i);
34  for (int i = 0; i < E.size(); i++) {
35      a = E[i].second.first;
36      b = E[i].second.second;
37      if (!dsf.is_united(a, b)) {
38          ...
39          dsf.unite(a, b);
40      }
41  }
42
43  */
44
45  #include <algorithm> /* std::sort() */
46  #include <iostream>
47  #include <vector>
48  using namespace std;
49
50  const int MAXN = 100;
51  int root[MAXN];
52  vector<pair<int, pair<int, int> > > E;
53  vector<pair<int, int> > mst;

```

```

54
55 int find_root(int x) {
56     if (root[x] != x)
57         root[x] = find_root(root[x]);
58     return root[x];
59 }
60
61 int kruskal(int nodes) {
62     mst.clear();
63     sort(E.begin(), E.end());
64     int u, v, total_dist = 0;
65     for (int i = 0; i < nodes; i++) root[i] = i;
66     for (int i = 0; i < (int)E.size(); i++) {
67         u = find_root(E[i].second.first);
68         v = find_root(E[i].second.second);
69         if (u != v) {
70             mst.push_back(E[i].second);
71             total_dist += E[i].first;
72             root[u] = root[v];
73         }
74     }
75     return total_dist;
76 }
77
78 int main() {
79     int nodes, edges, u, v, w;
80     cin >> nodes >> edges;
81     for (int i = 0; i < edges; i++) {
82         cin >> u >> v >> w;
83         E.push_back(make_pair(w, make_pair(u, v)));
84     }
85     cout << "Total_distance:_" << kruskal(nodes) << "\n";
86     for (int i = 0; i < (int)mst.size(); i++)
87         cout << mst[i].first << "<->" << mst[i].second << "\n";
88     return 0;
89 }

```

2.5 Maximum Flow

2.5.1 Ford-Fulkerson Algorithm

```

1  /*
2
3  Description: Given a flow network, find a flow from a single
4  source node to a single sink node that is maximized. Note
5  that in this implementation, the adjacency matrix cap[][]
6  will be modified by the function ford_fulkerson() after it's
7  been called. Make a back-up if you require it afterwards.
8
9  Complexity:  $O(V^2 \cdot |F|)$ , where  $V$  is the number of
10 vertices and  $|F|$  is the magnitude of the max flow.
11
12 Real-valued capacities:
13 The Ford-Fulkerson algorithm is only optimal on graphs with
14 integer capacities; there exists certain real capacity inputs
15 for which it will never terminate. The Edmonds-Karp algorithm

```

16 is an improvement using BFS, supporting real number capacities.

```

17
18 ==~~~ Sample Input ==~~~
19 6 8
20 0 1 3
21 0 2 3
22 1 2 2
23 1 3 3
24 2 4 2
25 3 4 1
26 3 5 2
27 4 5 3
28 0 5
29
30 ==~~~ Sample Output ==~~~
31 5
32
33 */
34
35 #include <algorithm> /* std::fill() */
36 #include <iostream>
37 #include <vector>
38 using namespace std;
39
40 const int MAXN = 100, INF = 0x3f3f3f3f;
41 int nodes, source, sink, cap[MAXN][MAXN];
42 vector<bool> vis(MAXN);
43
44 int dfs(int u, int f) {
45     if (u == sink) return f;
46     vis[u] = true;
47     for (int v = 0; v < nodes; v++) {
48         if (!vis[v] && cap[u][v] > 0) {
49             int df = dfs(v, min(f, cap[u][v]));
50             if (df > 0) {
51                 cap[u][v] -= df;
52                 cap[v][u] += df;
53                 return df;
54             }
55         }
56     }
57     return 0;
58 }
59
60 int ford_fulkerson() {
61     int max_flow = 0;
62     for (;;) {
63         fill(vis.begin(), vis.end(), false);
64         int df = dfs(source, INF);
65         if (df == 0) break;
66         max_flow += df;
67     }
68     return max_flow;
69 }
70
71 int main() {
72     int edges, u, v, capacity;
73     cin >> nodes >> edges;
74     for (int i = 0; i < edges; i++) {

```



```

75     cin >> u >> v >> capacity;
76     cap[u][v] = capacity;
77 }
78 cin >> source >> sink;
79 cout << ford_fulkerson() << "\n";
80 return 0;
81 }

```

2.5.2 Edmonds-Karp Algorithm

```

1  /*
2
3  Description: Given a flow network, find a flow from a single
4  source node to a single sink node that is maximized. Note
5  that in this implementation, the adjacency list adj[] will
6  be modified by the function edmonds_karp() after it's been called.
7
8  Complexity:  $O(\min(V \cdot E^2, E \cdot |F|))$ , where  $V$  is the number of
9  vertices,  $E$  is the number of edges, and  $|F|$  is the magnitude of
10 the max flow. This improves the original Ford-Fulkerson algorithm,
11 which runs in  $O(E \cdot |F|)$ . As the Edmonds-Karp algorithm is also
12 bounded by  $O(E \cdot |F|)$ , it is guaranteed to be at least as fast as
13 Ford-Fulkerson. For an even faster algorithm, see Dinic's
14 algorithm in the next section, which runs in  $O(V^2 \cdot E)$ .
15
16 Real-valued capacities:
17 Although the Ford-Fulkerson algorithm is only optimal on graphs
18 with integer capacities, the Edmonds-Karp algorithm also works
19 correctly on real-valued capacities.
20
21 ~=~=~= Sample Input ~=~=~=
22 6 8
23 0 1 3
24 0 2 3
25 1 2 2
26 1 3 3
27 2 4 2
28 3 4 1
29 3 5 2
30 4 5 3
31 0 5
32
33 ~=~=~= Sample Output ~=~=~=
34 5
35
36 */
37
38 #include <algorithm> /* std::fill(), std::min() */
39 #include <iostream>
40 #include <vector>
41 using namespace std;
42
43 struct edge { int s, t, rev, cap, f; };
44
45 const int MAXN = 100, INF = 0x3f3f3f3f;
46 vector<edge> adj[MAXN];
47

```

```

48 void add_edge(int s, int t, int cap) {
49     adj[s].push_back((edge){s, t, (int)adj[t].size(), cap, 0});
50     adj[t].push_back((edge){t, s, (int)adj[s].size() - 1, 0, 0});
51 }
52
53 int edmonds_karp(int nodes, int source, int sink) {
54     static int q[MAXN];
55     int max_flow = 0;
56     for (;;) {
57         int qt = 0;
58         q[qt++] = source;
59         edge * pred[nodes];
60         fill(pred, pred + nodes, (edge*)0);
61         for (int qh = 0; qh < qt && !pred[sink]; qh++) {
62             int u = q[qh];
63             for (int j = 0; j < (int)adj[u].size(); j++) {
64                 edge * e = &adj[u][j];
65                 if (!pred[e->t] && e->cap > e->f) {
66                     pred[e->t] = e;
67                     q[qt++] = e->t;
68                 }
69             }
70         }
71         if (!pred[sink]) break;
72         int df = INF;
73         for (int u = sink; u != source; u = pred[u]->s)
74             df = min(df, pred[u]->cap - pred[u]->f);
75         for (int u = sink; u != source; u = pred[u]->s) {
76             pred[u]->f += df;
77             adj[pred[u]->t][pred[u]->rev].f -= df;
78         }
79         max_flow += df;
80     }
81     return max_flow;
82 }
83
84 int main() {
85     int nodes, edges, u, v, capacity, source, sink;
86     cin >> nodes >> edges;
87     for (int i = 0; i < edges; i++) {
88         cin >> u >> v >> capacity;
89         add_edge(u, v, capacity);
90     }
91     cin >> source >> sink;
92     cout << edmonds_karp(nodes, source, sink) << "\n";
93     return 0;
94 }

```

2.5.3 Dinic's Algorithm

```

1  /*
2
3  Description: Given a flow network, find a flow from a single
4  source node to a single sink node that is maximized. Note
5  that in this implementation, the adjacency list adj[] will
6  be modified by the function dinic() after it's been called.
7

```

```

8 Complexity:  $O(V^2 \cdot E)$  on the number of vertices and edges.
9
10 Comparison with Edmonds-Karp Algorithm:
11 Dinic's is similar to the Edmonds-Karp algorithm in that it
12 uses the shortest augmenting path. The introduction of the
13 concepts of the level graph and blocking flow enable Dinic's
14 algorithm to achieve its better performance. Hence, Dinic's
15 algorithm is also called Dinic's blocking flow algorithm.
16
17 ~=~=~= Sample Input ~=~=~=
18 6 8
19 0 1 3
20 0 2 3
21 1 2 2
22 1 3 3
23 2 4 2
24 3 4 1
25 3 5 2
26 4 5 3
27 0 5
28
29 ~=~=~= Sample Output ~=~=~=
30 5
31
32 */
33
34 #include <algorithm> /* std::fill(), std::min() */
35 #include <iostream>
36 #include <vector>
37 using namespace std;
38
39 struct edge { int to, rev, cap, f; };
40
41 const int MAXN = 100, INF = 0x3f3f3f3f;
42 int dist[MAXN], ptr[MAXN];
43 vector<edge> adj[MAXN];
44
45 void add_edge(int s, int t, int cap) {
46     adj[s].push_back((edge){t, (int)adj[t].size(), cap, 0});
47     adj[t].push_back((edge){s, (int)adj[s].size() - 1, 0, 0});
48 }
49
50 bool dinic_bfs(int nodes, int source, int sink) {
51     fill(dist, dist + nodes, -1);
52     dist[source] = 0;
53     int q[nodes], qh = 0, qt = 0;
54     q[qt++] = source;
55     while (qh < qt) {
56         int u = q[qh++];
57         for (int j = 0; j < (int)adj[u].size(); j++) {
58             edge & e = adj[u][j];
59             if (dist[e.to] < 0 && e.f < e.cap) {
60                 dist[e.to] = dist[u] + 1;
61                 q[qt++] = e.to;
62             }
63         }
64     }
65     return dist[sink] >= 0;
66 }

```

```

67
68 int dinic_dfs(int u, int f, int sink) {
69     if (u == sink) return f;
70     for (; ptr[u] < (int)adj[u].size(); ptr[u]++) {
71         edge &e = adj[u][ptr[u]];
72         if (dist[e.to] == dist[u] + 1 && e.f < e.cap) {
73             int df = dinic_dfs(e.to, min(f, e.cap - e.f), sink);
74             if (df > 0) {
75                 e.f += df;
76                 adj[e.to][e.rev].f -= df;
77                 return df;
78             }
79         }
80     }
81     return 0;
82 }
83
84 int dinic(int nodes, int source, int sink) {
85     int max_flow = 0, delta;
86     while (dinic_bfs(nodes, source, sink)) {
87         fill(ptr, ptr + nodes, 0);
88         while ((delta = dinic_dfs(source, INF, sink)) != 0)
89             max_flow += delta;
90     }
91     return max_flow;
92 }
93
94 int main() {
95     int nodes, edges, u, v, capacity, source, sink;
96     cin >> nodes >> edges;
97     for (int i = 0; i < edges; i++) {
98         cin >> u >> v >> capacity;
99         add_edge(u, v, capacity);
100     }
101     cin >> source >> sink;
102     cout << dinic(nodes, source, sink) << "\n";
103     return 0;
104 }

```

2.5.4 Push-Relabel Algorithm

```

1  /*
2
3  Description: Given a flow network, find a flow from a single
4  source node to a single sink node that is maximized. The push-
5  relabel algorithm is considered one of the most efficient
6  maximum flow algorithms. However, unlike the Ford-Fulkerson or
7  Edmonds-Karp algorithms, it cannot take advantage of the fact
8  if max flow itself has a small magnitude.
9
10 Complexity:  $O(V^3)$  on the number of vertices.
11
12 ~=~=~= Sample Input ~=~=~=
13 6 8
14 0 1 3
15 0 2 3
16 1 2 2

```

```

17  1 3 3
18  2 4 2
19  3 4 1
20  3 5 2
21  4 5 3
22  0 5
23
24  ~=~=~= Sample Output ~=~=~=
25  5
26
27  */
28
29  #include <algorithm> /* std::fill(), std::min() */
30  #include <iostream>
31  using namespace std;
32
33  const int MAXN = 100, INF = 0x3F3F3F3F;
34  int cap[MAXN][MAXN], f[MAXN][MAXN];
35
36  int push_relabel(int nodes, int source, int sink) {
37      int e[nodes], h[nodes], maxh[nodes];
38      fill(e, e + nodes, 0);
39      fill(h, h + nodes, 0);
40      fill(maxh, maxh + nodes, 0);
41      for (int i = 0; i < nodes; i++)
42          fill(f[i], f[i] + nodes, 0);
43      h[source] = nodes - 1;
44      for (int i = 0; i < nodes; i++) {
45          f[source][i] = cap[source][i];
46          f[i][source] = -f[source][i];
47          e[i] = cap[source][i];
48      }
49      int sz = 0;
50      for (;;) {
51          if (sz == 0) {
52              for (int i = 0; i < nodes; i++)
53                  if (i != source && i != sink && e[i] > 0) {
54                      if (sz != 0 && h[i] > h[maxh[0]]) sz = 0;
55                      maxh[sz++] = i;
56                  }
57          }
58          if (sz == 0) break;
59          while (sz != 0) {
60              int i = maxh[sz - 1];
61              bool pushed = false;
62              for (int j = 0; j < nodes && e[i] != 0; j++) {
63                  if (h[i] == h[j] + 1 && cap[i][j] - f[i][j] > 0) {
64                      int df = min(cap[i][j] - f[i][j], e[i]);
65                      f[i][j] += df;
66                      f[j][i] -= df;
67                      e[i] -= df;
68                      e[j] += df;
69                      if (e[i] == 0) sz--;
70                      pushed = true;
71                  }
72              }
73              if (!pushed) {
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         if (h[i] > h[maxh[0]]) {
79             sz = 0;
80             break;
81         }
82     }
83 }
84 }
85 int max_flow = 0;
86 for (int i = 0; i < nodes; i++)
87     max_flow += f[source][i];
88 return max_flow;
89 }
90
91 int main() {
92     int nodes, edges, u, v, capacity, source, sink;
93     cin >> nodes >> edges;
94     for (int i = 0; i < edges; i++) {
95         cin >> u >> v >> capacity;
96         cap[u][v] = capacity;
97     }
98     cin >> source >> sink;
99     cout << push_relabel(nodes, source, sink) << "\n";
100     return 0;
101 }

```

2.6 Backtracking

2.6.1 Max Clique (Bron-Kerbosch Algorithm)

```

1  /*
2
3  Description: Given an undirected graph, determine a subset of
4  the graph's vertices such that every pair of vertices in the
5  subset are connected by an edge, and that the subset is as
6  large as possible. For the weighted version, each vertex is
7  assigned a weight and the objective is to find the clique in
8  the graph that has maximum total weight.
9
10 Complexity:  $O(3^{(V/3)})$  where V is the number of vertices.
11
12 ~=~=~= Sample Input ~=~=~=
13 5 8
14 0 1
15 0 2
16 0 3
17 1 2
18 1 3
19 2 3
20 3 4
21 4 2
22 10 20 30 40 50
23
24 ~=~=~= Sample Output ~=~=~=
25 Max unweighted clique: 4

```

```

26 Max weighted clique: 120
27
28 */
29
30 #include <algorithm> /* std::fill(), std::max() */
31 #include <bitset>
32 #include <iostream>
33 #include <vector>
34 using namespace std;
35
36 const int MAXN = 35;
37 typedef bitset<MAXN> bits;
38 typedef unsigned long long ull;
39
40 int w[MAXN];
41 bool adj[MAXN][MAXN];
42
43 int rec(int nodes, bits & curr, bits & pool, bits & excl) {
44     if (pool.none() && excl.none()) return curr.count();
45     int ans = 0, u = 0;
46     for (int v = 0; v < nodes; v++)
47         if (pool[v] || excl[v]) u = v;
48     for (int v = 0; v < nodes; v++) {
49         if (!pool[v] || adj[u][v]) continue;
50         bits ncurr, npool, nexcl;
51         for (int i = 0; i < nodes; i++) ncurr[i] = curr[i];
52         ncurr[v] = true;
53         for (int j = 0; j < nodes; j++) {
54             npool[j] = pool[j] && adj[v][j];
55             nexcl[j] = excl[j] && adj[v][j];
56         }
57         ans = max(ans, rec(nodes, ncurr, npool, nexcl));
58         pool[v] = false;
59         excl[v] = true;
60     }
61     return ans;
62 }
63
64 int bron_kerbosch(int nodes) {
65     bits curr, excl, pool;
66     pool.flip();
67     return rec(nodes, curr, pool, excl);
68 }
69
70 //This is a fast implementation using bitmasks.
71 //Precondition: the number of nodes must be less than 64.
72 int bron_kerbosch_weighted(int nodes, ull g[], ull curr, ull pool, ull excl) {
73     if (pool == 0 && excl == 0) {
74         int res = 0, u = __builtin_ctzll(curr);
75         while (u < nodes) {
76             res += w[u];
77             u += __builtin_ctzll(curr >> (u + 1)) + 1;
78         }
79         return res;
80     }
81     if (pool == 0) return -1;
82     int res = -1, pivot = __builtin_ctzll(pool | excl);
83     ull z = pool & ~g[pivot];
84     int u = __builtin_ctzll(z);

```

```

85     while (u < nodes) {
86         res = max(res, bron_kerbosch_weighted(nodes, g, curr | (1LL << u),
87                                             pool & g[u], excl & g[u]));
88         pool ^= 1LL << u;
89         excl |= 1LL << u;
90         u += __builtin_ctzll(z >> (u + 1)) + 1;
91     }
92     return res;
93 }
94
95 int bron_kerbosch_weighted(int nodes) {
96     ull g[nodes];
97     for (int i = 0; i < nodes; i++) {
98         g[i] = 0;
99         for (int j = 0; j < nodes; j++)
100             if (adj[i][j]) g[i] |= 1LL << j;
101     }
102     return bron_kerbosch_weighted(nodes, g, 0, (1LL << nodes) - 1, 0);
103 }
104
105 int main() {
106     int nodes, edges, u, v;
107     cin >> nodes >> edges;
108     for (int i = 0; i < edges; i++) {
109         cin >> u >> v;
110         adj[u][v] = adj[v][u] = true;
111     }
112     for (int i = 0; i < nodes; i++) cin >> w[i];
113     cout << "Max_unweighted_clique: ";
114     cout << bron_kerbosch(nodes) << "\n";
115     cout << "Max_weighted_clique: ";
116     cout << bron_kerbosch_weighted(nodes) << "\n";
117     return 0;
118 }

```

2.6.2 Graph Coloring

```

1  /*
2
3  Description: Given an undirected graph, assign colors to each
4  of the vertices such that no pair of adjacent vertices have the
5  same color. Furthermore, do so using the minimum # of colors.
6
7  Complexity: Exponential on the number of vertices. The exact
8  running time is difficult to calculate due to several pruning
9  optimizations used here.
10
11  ~=~=~= Sample Input ~=~=~=
12  5 7
13  0 1
14  0 4
15  1 3
16  1 4
17  2 3
18  2 4
19  3 4
20

```



```

21  ~=~=~= Sample Output ~=~=~=
22  Colored using 3 color(s). The colorings are:
23  Color 1: 0 3
24  Color 2: 1 2
25  Color 3: 4
26
27  */
28
29  #include <algorithm> /* std::fill(), std::max() */
30  #include <iostream>
31  #include <vector>
32  using namespace std;
33
34  const int MAXN = 30;
35  int cols[MAXN], adj[MAXN][MAXN];
36  int id[MAXN + 1], deg[MAXN + 1];
37  int min_cols, best_cols[MAXN];
38
39  void dfs(int from, int to, int cur, int used_cols) {
40      if (used_cols >= min_cols) return;
41      if (cur == to) {
42          for (int i = from; i < to; i++)
43              best_cols[id[i]] = cols[i];
44          min_cols = used_cols;
45          return;
46      }
47      vector<bool> used(used_cols + 1);
48      for (int i = 0; i < cur; i++)
49          if (adj[id[cur]][id[i]]) used[cols[i]] = true;
50      for (int i = 0; i <= used_cols; i++) {
51          if (!used[i]) {
52              int tmp = cols[cur];
53              cols[cur] = i;
54              dfs(from, to, cur + 1, max(used_cols, i + 1));
55              cols[cur] = tmp;
56          }
57      }
58  }
59
60  int color_graph(int nodes) {
61      for (int i = 0; i <= nodes; i++) {
62          id[i] = i;
63          deg[i] = 0;
64      }
65      int res = 1;
66      for (int from = 0, to = 1; to <= nodes; to++) {
67          int best = to;
68          for (int i = to; i < nodes; i++) {
69              if (adj[id[to - 1]][id[i]]) deg[id[i]]++;
70              if (deg[id[best]] < deg[id[i]]) best = i;
71          }
72          int tmp = id[to];
73          id[to] = id[best];
74          id[best] = tmp;
75          if (deg[id[to]] == 0) {
76              min_cols = nodes + 1;
77              fill(cols, cols + nodes, 0);
78              dfs(from, to, from, 0);
79              from = to;

```

```

80     res = max(res, min_cols);
81 }
82 }
83 return res;
84 }
85
86 int main() {
87     int nodes, edges, u, v;
88     cin >> nodes >> edges;
89     for (int i = 0; i < edges; i++) {
90         cin >> u >> v;
91         adj[u][v] = adj[v][u] = true;
92     }
93     cout << "Colored using " << color_graph(nodes);
94     cout << " color(s). The colorings are:\n";
95     for (int i = 0; i < min_cols; i++) {
96         cout << "Color " << i + 1 << ":\n";
97         for (int j = 0; j < nodes; j++)
98             if (best_cols[j] == i) cout << " " << j;
99         cout << "\n";
100     }
101     return 0;
102 }

```

2.7 Maximum Matching

2.7.1 Maximum Bipartite Matching (Kuhn's Algorithm)

```

1  /*
2
3  Description: Given two sets of vertices A = {0, 1, ..., n1}
4  and B = {0, 1, ..., n2} as well as a set of edges E mapping
5  nodes from set A to set B, determine the largest possible
6  subset of E such that no pair of edges in the subset share
7  a common vertex. Precondition: n2 >= n1.
8
9  Complexity: O(V*E) on the number of vertices and edges.
10
11  ~=~=~= Sample Input ~=~=~=
12  3 4 6
13  0 1
14  1 0
15  1 1
16  1 2
17  2 2
18  2 3
19
20  ~=~=~= Sample Output ~=~=~=
21  Matched 3 pairs. Matchings are:
22  1 0
23  0 1
24  2 2
25
26  */
27
28  #include <algorithm> /* std::fill() */

```

```

29 #include <iostream>
30 #include <vector>
31 using namespace std;
32
33 const int MAXN = 100;
34 int match[MAXN];
35 vector<bool> vis(MAXN);
36 vector<int> adj[MAXN];
37
38 bool dfs(int u) {
39     vis[u] = true;
40     for (int j = 0; j < (int)adj[u].size(); j++) {
41         int v = match[adj[u][j]];
42         if (v == -1 || (!vis[v] && dfs(v))) {
43             match[adj[u][j]] = u;
44             return true;
45         }
46     }
47     return false;
48 }
49
50 int kuhn(int n1, int n2) {
51     fill(vis.begin(), vis.end(), false);
52     fill(match, match + n2, -1);
53     int matches = 0;
54     for (int i = 0; i < n1; i++) {
55         for (int j = 0; j < n1; j++) vis[j] = 0;
56         if (dfs(i)) matches++;
57     }
58     return matches;
59 }
60
61 int main() {
62     int n1, n2, edges, u, v;
63     cin >> n1 >> n2 >> edges;
64     for (int i = 0; i < edges; i++) {
65         cin >> u >> v;
66         adj[u].push_back(v);
67     }
68     cout << "Matched_" << kuhn(n1, n2);
69     cout << "_pair(s)_Matchings_are:\n";
70     for (int i = 0; i < n2; i++) {
71         if (match[i] == -1) continue;
72         cout << match[i] << "_" << i << "\n";
73     }
74     return 0;
75 }

```

2.7.2 Maximum Bipartite Matching (Hopcroft-Karp Algorithm)

```

1  /*
2
3  Description: Given two sets of vertices  $A = \{0, 1, \dots, n1\}$ 
4  and  $B = \{0, 1, \dots, n2\}$  as well as a set of edges  $E$  mapping
5  nodes from set  $A$  to set  $B$ , determine the largest possible
6  subset of  $E$  such that no pair of edges in the subset share
7  a common vertex. Precondition:  $n2 \geq n1$ .

```

```

8
9 Complexity:  $O(E \sqrt{V})$  on the number of edges and vertices.
10
11 ~=~=~= Sample Input ~=~=~=
12 3 4 6
13 0 1
14 1 0
15 1 1
16 1 2
17 2 2
18 2 3
19
20 ~=~=~= Sample Output ~=~=~=
21 Matched 3 pairs. Matchings are:
22 1 0
23 0 1
24 2 2
25
26 */
27
28 #include <algorithm> /* std::fill() */
29 #include <iostream>
30 #include <vector>
31 using namespace std;
32
33 const int MAXN = 100;
34 int match[MAXN], dist[MAXN];
35 vector<bool> used(MAXN), vis(MAXN);
36 vector<int> adj[MAXN];
37
38 void bfs(int n1, int n2) {
39     fill(dist, dist + n1, -1);
40     int q[n2], qb = 0;
41     for (int u = 0; u < n1; ++u) {
42         if (!used[u]) {
43             q[qb++] = u;
44             dist[u] = 0;
45         }
46     }
47     for (int i = 0; i < qb; i++) {
48         int u = q[i];
49         for (int j = 0; j < (int)adj[u].size(); j++) {
50             int v = match[adj[u][j]];
51             if (v >= 0 && dist[v] < 0) {
52                 dist[v] = dist[u] + 1;
53                 q[qb++] = v;
54             }
55         }
56     }
57 }
58
59 bool dfs(int u) {
60     vis[u] = true;
61     for (int j = 0; j < (int)adj[u].size(); j++) {
62         int v = match[adj[u][j]];
63         if (v < 0 || (!vis[v] && dist[v] == dist[u] + 1 && dfs(v))) {
64             match[adj[u][j]] = u;
65             used[u] = true;
66             return true;

```

```

67     }
68 }
69 return false;
70 }
71
72 int hopcroft_karp(int n1, int n2) {
73     fill(match, match + n2, -1);
74     fill(used.begin(), used.end(), false);
75     int res = 0;
76     for (;;) {
77         bfs(n1, n2);
78         fill(vis.begin(), vis.end(), false);
79         int f = 0;
80         for (int u = 0; u < n1; ++u)
81             if (!used[u] && dfs(u)) f++;
82         if (!f) return res;
83         res += f;
84     }
85     return res;
86 }
87
88 int main() {
89     int n1, n2, edges, u, v;
90     cin >> n1 >> n2 >> edges;
91     for (int i = 0; i < edges; i++) {
92         cin >> u >> v;
93         adj[u].push_back(v);
94     }
95     cout << "Matched_" << hopcroft_karp(n1, n2);
96     cout << "_pair(s)._Matchings_are:\n";
97     for (int i = 0; i < n2; i++) {
98         if (match[i] == -1) continue;
99         cout << match[i] << "_" << i << "\n";
100     }
101     return 0;
102 }

```

2.7.3 Maximum Graph Matching (Edmonds's Algorithm)

```

1  /*
2
3  Description: Given a general directed graph, determine a maximal
4  subset of the edges such that no vertex is repeated in the subset.
5
6  Complexity:  $O(V^3)$  on the number of vertices.
7
8  ~=~=~= Sample Input ~=~=~=
9  4 8
10 0 1
11 1 0
12 1 2
13 2 1
14 2 3
15 3 2
16 3 0
17 0 3
18

```

```

19  ~=~=~= Sample Output ~=~=~=
20  Matched 2 pair(s). Matchings are:
21  0 1
22  2 3
23
24  */
25
26  #include <iostream>
27  #include <vector>
28  using namespace std;
29
30  const int MAXN = 100;
31  int p[MAXN], base[MAXN], match[MAXN];
32  vector<int> adj[MAXN];
33
34  int lca(int nodes, int a, int b) {
35      vector<bool> used(nodes);
36      for (;;) {
37          a = base[a];
38          used[a] = true;
39          if (match[a] == -1) break;
40          a = p[match[a]];
41      }
42      for (;;) {
43          b = base[b];
44          if (used[b]) return b;
45          b = p[match[b]];
46      }
47  }
48
49  void mark_path(vector<bool> & blossom, int v, int b, int children) {
50      for (; base[v] != b; v = p[match[v]]) {
51          blossom[base[v]] = blossom[base[match[v]]] = true;
52          p[v] = children;
53          children = match[v];
54      }
55  }
56
57  int find_path(int nodes, int root) {
58      vector<bool> used(nodes);
59      for (int i = 0; i < nodes; ++i) {
60          p[i] = -1;
61          base[i] = i;
62      }
63      used[root] = true;
64      int q[nodes], qh = 0, qt = 0;
65      q[qt++] = root;
66      while (qh < qt) {
67          int v = q[qh++];
68          for (int j = 0, to; j < (int)adj[v].size(); j++) {
69              to = adj[v][j];
70              if (base[v] == base[to] || match[v] == to) continue;
71              if (to == root || (match[to] != -1 && p[match[to]] != -1)) {
72                  int curbase = lca(nodes, v, to);
73                  vector<bool> blossom(nodes);
74                  mark_path(blossom, v, curbase, to);
75                  mark_path(blossom, to, curbase, v);
76                  for (int i = 0; i < nodes; i++)
77                      if (blossom[base[i]]) {

```

```

78         base[i] = curbase;
79         if (!used[i]) {
80             used[i] = true;
81             q[qt++] = i;
82         }
83     }
84 } else if (p[to] == -1) {
85     p[to] = v;
86     if (match[to] == -1) return to;
87     to = match[to];
88     used[to] = true;
89     q[qt++] = to;
90 }
91 }
92 }
93 return -1;
94 }
95
96 int edmonds(int nodes) {
97     for (int i = 0; i < nodes; i++) match[i] = -1;
98     for (int i = 0; i < nodes; i++) {
99         if (match[i] == -1) {
100             int v, pv, ppv;
101             for (v = find_path(nodes, i); v != -1; v = ppv) {
102                 ppv = match[pv = p[v]];
103                 match[v] = pv;
104                 match[pv] = v;
105             }
106         }
107     }
108     int matches = 0;
109     for (int i = 0; i < nodes; i++)
110         if (match[i] != -1) matches++;
111     return matches / 2;
112 }
113
114 int main() {
115     int nodes, edges, u, v;
116     cin >> nodes >> edges;
117     for (int i = 0; i < edges; i++) {
118         cin >> u >> v;
119         adj[u].push_back(v);
120     }
121     cout << "Matched_" << edmonds(nodes);
122     cout << "_pair(s)_.Matchings_are:\n";
123     for (int i = 0; i < nodes; i++) {
124         if (match[i] != -1 && i < match[i])
125             cout << i << " " << match[i] << "\n";
126     }
127     return 0;
128 }

```

2.8 Hamiltonian Path and Cycle

2.8.1 Shortest Hamiltonian Cycle (Travelling Salesman)

```

1  /*
2
3  Description: Given a weighted, directed graph, the shortest
4  hamiltonian cycle is a cycle of minimum distance that visits
5  each vertex exactly once and returns to the original vertex.
6  This is also known as the traveling salesman problem (TSP).
7  Since this is a bitmasking solution with 32-bit integers,
8  the number of vertices must be less than 32.
9
10 Complexity:  $O(2^V * V^2)$  on the number of vertices.
11
12 ~=~=~= Sample Input ~=~=~=
13 5 10
14 0 1 1
15 0 2 10
16 0 3 1
17 0 4 10
18 1 2 10
19 1 3 10
20 1 4 1
21 2 3 1
22 2 4 1
23 3 4 10
24
25 ~=~=~= Sample Output ~=~=~=
26 The shortest hamiltonian cycle has length 5.
27 Take the path: 0->3->2->4->1->0
28
29 */
30
31 #include <algorithm> /* std::fill(), std::min() */
32 #include <iostream>
33 using namespace std;
34
35 const int MAXN = 20, INF = 0x3f3f3f3f;
36 int adj[MAXN][MAXN], order[MAXN];
37
38 int shortest_hamiltonian_cycle(int nodes) {
39     int dp[1 << nodes][nodes];
40     for (int i = 0; i < (1 << nodes); i++)
41         fill(dp[i], dp[i] + nodes, INF);
42     dp[1][0] = 0;
43     for (int mask = 1; mask < (1 << nodes); mask += 2) {
44         for (int i = 1; i < nodes; i++)
45             if ((mask & 1 << i) != 0)
46                 for (int j = 0; j < nodes; j++)
47                     if ((mask & 1 << j) != 0)
48                         dp[mask][i] = min(dp[mask][i], dp[mask ^ (1 << i)][j] + adj[j][i]);
49     }
50     int res = INF + INF;
51     for (int i = 1; i < nodes; i++)
52         res = min(res, dp[(1 << nodes) - 1][i] + adj[i][0]);
53     int cur = (1 << nodes) - 1, last = 0;
54     for (int i = nodes - 1; i >= 1; i--) {
55         int bj = -1;
56         for (int j = 1; j < nodes; j++) {
57             if ((cur & 1 << j) != 0 && (bj == -1 ||
58                 dp[cur][bj] + adj[bj][last] > dp[cur][j] + adj[j][last])) {
59                 bj = j;

```



```

60     }
61 }
62 order[i] = bj;
63 cur ^= 1 << bj;
64 last = bj;
65 }
66 return res;
67 }
68
69 int main() {
70     int nodes, edges, u, v, w;
71     cin >> nodes >> edges;
72     for (int i = 0; i < edges; i++) {
73         cin >> u >> v >> w;
74         adj[u][v] = adj[v][u] = w; //only set adj[u][v] if directed edges
75     }
76     cout << "The shortest hamiltonian cycle has length ";
77     cout << shortest_hamiltonian_cycle(nodes) << ".\n";
78     cout << "Take the path: ";
79     for (int i = 0; i < nodes; i++) cout << order[i] << "->";
80     cout << order[0] << "\n";
81     return 0;
82 }

```

2.8.2 Shortest Hamiltonian Path

```

1  /*
2
3  Description: Given a weighted, directed graph, the shortest
4  hamiltonian path is a path of minimum distance that visits
5  each vertex exactly once. Unlike the travelling salesman
6  problem, we don't have to return to the starting vertex.
7  Since this is a bitmasking solution with 32-bit integers,
8  the number of vertices must be less than 32.
9
10 Complexity:  $O(2^V * V^2)$  on the number of vertices.
11
12 ~=~=~= Sample Input ~=~=~=
13 3 6
14 0 1 1
15 0 2 1
16 1 0 7
17 1 2 2
18 2 0 3
19 2 1 5
20
21 ~=~=~= Sample Output ~=~=~=
22 The shortest hamiltonian path has length 3.
23 Take the path: 0->1->2
24
25 */
26
27 #include <algorithm> /* std::fill(), std::min() */
28 #include <iostream>
29 using namespace std;
30
31 const int MAXN = 20, INF = 0x3f3f3f3f;

```

```

32
33 int adj[MAXN][MAXN], order[MAXN];
34
35 int shortest_hamiltonian_path(int nodes) {
36     int dp[1 << nodes][nodes];
37     for (int i = 0; i < (1 << nodes); i++)
38         fill(dp[i], dp[i] + nodes, INF);
39     for (int i = 0; i < nodes; i++) dp[1 << i][i] = 0;
40     for (int mask = 1; mask < (1 << nodes); mask += 2) {
41         for (int i = 0; i < nodes; i++)
42             if ((mask & 1 << i) != 0)
43                 for (int j = 0; j < nodes; j++)
44                     if ((mask & 1 << j) != 0)
45                         dp[mask][i] = min(dp[mask][i], dp[mask ^ (1 << i)][j] + adj[j][i]);
46     }
47     int res = INF + INF;
48     for (int i = 1; i < nodes; i++)
49         res = min(res, dp[(1 << nodes) - 1][i]);
50     int cur = (1 << nodes) - 1, last = -1;
51     for (int i = nodes - 1; i >= 0; i--) {
52         int bj = -1;
53         for (int j = 0; j < nodes; j++) {
54             if ((cur & 1 << j) != 0 && (bj == -1 ||
55                 dp[cur][bj] + (last == -1 ? 0 : adj[bj][last]) >
56                 dp[cur][j] + (last == -1 ? 0 : adj[j][last]))) {
57                 bj = j;
58             }
59         }
60         order[i] = bj;
61         cur ^= 1 << bj;
62         last = bj;
63     }
64     return res;
65 }
66
67 int main() {
68     int nodes, edges, u, v, w;
69     cin >> nodes >> edges;
70     for (int i = 0; i < edges; i++) {
71         cin >> u >> v >> w;
72         adj[u][v] = w;
73     }
74     cout << "The shortest hamiltonian path has length ";
75     cout << shortest_hamiltonian_path(nodes) << ".\n";
76     cout << "Take the path: ";
77     for (int i = 1; i < nodes; i++) cout << "->" << order[i];
78     return 0;
79 }

```

Chapter 3

Data Structures

3.1 Disjoint Sets

3.1.1 Disjoint Set Forest (Simple)

```
1  /*
2
3  Description: This data structure dynamically keeps track
4  of items partitioned into non-overlapping sets (a disjoint
5  set forest). It is also known as a union-find data structure.
6
7  Time Complexity: Every function below is  $O(a(N))$  amortized
8  on the number of items in the set due to the optimizations
9  of union by rank and path compression. Here,  $a(N)$  is the
10 extremely slow growing inverse of the Ackermann function.
11 For all practical values of  $n$ ,  $a(n)$  is less than 5.
12
13 Space Complexity:  $O(N)$  total.
14
15 */
16
17 const int MAXN = 1000;
18 int num_sets = 0, root[MAXN+1], rank[MAXN+1];
19
20 int find_root(int x) {
21     if (root[x] != x) root[x] = find_root(root[x]);
22     return root[x];
23 }
24
25 void make_set(int x) {
26     root[x] = x;
27     rank[x] = 0;
28     num_sets++;
29 }
30
31 bool is_united(int x, int y) {
32     return find_root(x) == find_root(y);
33 }
34
35 void unite(int x, int y) {
```

```

36     int X = find_root(x), Y = find_root(y);
37     if (X == Y) return;
38     num_sets--;
39     if (rank[X] < rank[Y]) root[X] = Y;
40     else if (rank[X] > rank[Y]) root[Y] = X;
41     else rank[root[Y] = X]++;
42 }
43
44 /** Example Usage ***/
45
46 #include <cassert>
47 #include <iostream>
48 using namespace std;
49
50 int main() {
51     for (char c = 'a'; c <= 'g'; c++) make_set(c);
52     unite('a', 'b');
53     unite('b', 'f');
54     unite('d', 'e');
55     unite('e', 'g');
56     assert(num_sets == 3);
57     assert(is_united('a', 'b'));
58     assert(!is_united('a', 'c'));
59     assert(!is_united('b', 'g'));
60     assert(is_united('d', 'g'));
61     return 0;
62 }

```

3.1.2 Disjoint Set Forest

```

1  /*
2
3  Description: This data structure dynamically keeps track
4  of items partitioned into non-overlapping sets (a disjoint
5  set forest). It is also known as a union-find data structure.
6  This particular templated version employs an std::map for
7  built in storage and coordinate compression. That is, the
8  magnitude of values inserted is not limited.
9
10 Time Complexity: make_set(), unite() and is_united() are
11  $O(a(N) + \log N) = O(\log N)$  on the number of elements in the
12 disjoint set forest. get_all_sets() is  $O(N)$ . find() is is
13  $O(a(N))$  amortized on the number of items in the set due to
14 the optimizations of union by rank and path compression.
15 Here,  $a(N)$  is the extremely slow growing inverse of the
16 Ackermann function. For all practical values of  $n$ ,  $a(n)$  is
17 less than 5.
18
19 Space Complexity:  $O(N)$  storage and auxiliary.
20
21 ~=~=~= Sample Output ~=~=~=
22 Elements: 7, Sets: 3
23 [[a,b,f],[c],[d,e,g]]
24
25 */
26
27 #include <map>

```

```

28 #include <vector>
29
30 template<class T> class disjoint_set_forest {
31     int num_elements, num_sets;
32     std::map<T, int> ID;
33     std::vector<int> root, rank;
34
35     int find_root(int x) {
36         if (root[x] != x) root[x] = find_root(root[x]);
37         return root[x];
38     }
39
40 public:
41     disjoint_set_forest(): num_elements(0), num_sets(0) {}
42     int elements() { return num_elements; }
43     int sets() { return num_sets; }
44
45     bool is_united(const T & x, const T & y) {
46         return find_root(ID[x]) == find_root(ID[y]);
47     }
48
49     void make_set(const T & x) {
50         if (ID.find(x) != ID.end()) return;
51         root.push_back(ID[x] = num_elements++);
52         rank.push_back(0);
53         num_sets++;
54     }
55
56     void unite(const T & x, const T & y) {
57         int X = find_root(ID[x]), Y = find_root(ID[y]);
58         if (X == Y) return;
59         num_sets--;
60         if (rank[X] < rank[Y]) root[X] = Y;
61         else if (rank[X] > rank[Y]) root[Y] = X;
62         else rank[root[Y] = X]++;
63     }
64
65     std::vector<std::vector<T> > get_all_sets() {
66         std::map<int, std::vector<T> > tmp;
67         for (typename std::map<T, int>::iterator
68             it = ID.begin(); it != ID.end(); it++)
69             tmp[find_root(it->second)].push_back(it->first);
70         std::vector<std::vector<T> > ret;
71         for (typename std::map<int, std::vector<T> >::
72             iterator it = tmp.begin(); it != tmp.end(); it++)
73             ret.push_back(it->second);
74         return ret;
75     }
76 };
77
78 /** Example Usage */
79
80 #include <iostream>
81 using namespace std;
82
83 int main() {
84     disjoint_set_forest<char> d;
85     for (char c = 'a'; c <= 'g'; c++) d.make_set(c);
86     d.unite('a', 'b');

```

```

87  d.unite('b', 'f');
88  d.unite('d', 'e');
89  d.unite('e', 'g');
90  cout << "Elements:␣" << d.elements();
91  cout << "␣Sets:␣" << d.sets() << endl;
92  vector<vector<char>> > s = d.get_all_sets();
93  cout << "[";
94  for (int i = 0; i < (int)s.size(); i++) {
95      cout << (i > 0 ? ",[" : "[";
96      for (int j = 0; j < (int)s[i].size(); j++)
97          cout << (j > 0 ? "," : "") << s[i][j];
98      cout << "]";
99  }
100  cout << "]\n";
101  return 0;
102 }

```

3.2 Fenwick Trees

3.2.1 Simple Fenwick Tree

```

1  /*
2
3  Description: A Fenwick tree (a.k.a. binary indexed tree) is a
4  data structure that allows for the sum of an arbitrary range
5  of values in an array to be dynamically queried in logarithmic
6  time. Note that unlike the object-oriented versions of this
7  data structure found in later sections, the operations here
8  work on 1-based indices (i.e. between 1 and MAXN, inclusive).
9  The array a[] is always synchronized with the bit[] array and
10 should not be modified outside of the functions below.
11
12 Time Complexity: All functions are O(log MAXN).
13 Space Complexity: O(MAXN) storage and auxiliary.
14
15 */
16
17 const int MAXN = 1000;
18 int a[MAXN + 1], bit[MAXN + 1];
19
20 //a[i] += v
21 void add(int i, int v) {
22     a[i] += v;
23     for (; i <= MAXN; i += i & -i)
24         bit[i] += v;
25 }
26
27 //a[i] = v
28 void set(int i, int v) {
29     int inc = v - a[i];
30     add(i, inc);
31 }
32
33 //returns sum(a[i] for i = 1..hi inclusive)
34 int sum(int hi) {
35     int ret = 0;

```

```

36     for (; hi > 0; hi -= hi & -hi)
37         ret += bit[hi];
38     return ret;
39 }
40
41 //returns sum(a[i] for i = lo..hi inclusive)
42 int sum(int lo, int hi) {
43     return sum(hi) - sum(lo - 1);
44 }
45
46 /** Example Usage */
47
48 #include <iostream>
49 using namespace std;
50
51 int main() {
52     for (int i = 1; i <= 5; i++) set(i, i);
53     add(4, -5);
54     cout << "BIT values: ";
55     for (int i = 1; i <= 5; i++)
56         cout << a[i] << " "; //1 2 3 -1 5
57     cout << "\nSum of range [1,3] is ";
58     cout << sum(1, 3) << ".\n"; //6
59     return 0;
60 }

```

3.2.2 Fenwick Tree

```

1  /*
2
3  Description: A Fenwick tree (a.k.a. binary indexed tree) is a
4  data structure that allows for the sum of an arbitrary range
5  of values in an array to be dynamically queried in logarithmic
6  time. All methods below work on 0-based indices (i.e. indices
7  in the range from 0 to size() - 1, inclusive, are valid).
8
9  Time Complexity: add(), set(), and sum() are all O(log N) on
10 the length of the array. size() and at() are O(1).
11
12 Space Complexity: O(N) storage and O(N) auxiliary on size().
13
14 */
15
16 #include <vector>
17
18 template<class T> class fenwick_tree {
19     int len;
20     std::vector<int> a, bit;
21
22 public:
23     fenwick_tree(int n): len(n),
24         a(n + 1), bit(n + 1) {}
25
26     //a[i] += v
27     void add(int i, const T & v) {
28         a[++i] += v;
29         for (; i <= len; i += i & -i)

```

```

30     bit[i] += v;
31 }
32
33 //a[i] = v
34 void set(int i, const T & v) {
35     T inc = v - a[i + 1];
36     add(i, inc);
37 }
38
39 //returns sum(a[i] for i = 1..hi inclusive)
40 T sum(int hi) {
41     T res = 0;
42     for (hi++; hi > 0; hi -= hi & -hi)
43         res += bit[hi];
44     return res;
45 }
46
47 //returns sum(a[i] for i = lo..hi inclusive)
48 T sum(int lo, int hi) {
49     return sum(hi) - sum(lo - 1);
50 }
51
52 inline int size() { return len; }
53 inline T at(int i) { return a[i + 1]; }
54 };
55
56 /** Example Usage */
57
58 #include <iostream>
59 using namespace std;
60
61 int main() {
62     int a[] = {10, 1, 2, 3, 4};
63     fenwick_tree<int> t(5);
64     for (int i = 0; i < 5; i++) t.set(i, a[i]);
65     t.add(0, -5);
66     cout << "BIT values: ";
67     for (int i = 0; i < t.size(); i++)
68         cout << t.at(i) << " "; //5 1 2 3 4
69     cout << "\nSum of range [1, 3] is ";
70     cout << t.sum(1, 3) << ".\n"; //6
71     return 0;
72 }

```

3.2.3 Fenwick Tree (Point Query)

```

1  /*
2
3  Description: A Fenwick tree (a.k.a. binary indexed tree) is a
4  data structure that allows for the sum of an arbitrary range
5  of values in an array to be dynamically queried in logarithmic
6  time. Range updating in a Fenwick tree can only increment
7  values in a range, not set them all to the same value. This
8  version is a very concise version if only point queries are
9  needed. The functions below work on 1-based indices (between
10  1 and MAXN, inclusive).
11

```



```

12 Time Complexity: add() and at() are  $O(\log \text{MAXN})$ .
13 Space Complexity:  $O(N)$ .
14
15 */
16
17 const int MAXN = 1000;
18 int bit[MAXN + 1];
19
20 //a[i] += v
21 void add(int i, int v) {
22     for (i++; i <= MAXN; i += i & -i) bit[i] += v;
23 }
24
25 //a[i] += v for i = lo..hi, inclusive
26 void add(int lo, int hi, int v) {
27     add(lo, v);
28     add(hi + 1, -v);
29 }
30
31 //returns a[i]
32 int at(int i) {
33     int sum = 0;
34     for (i++; i > 0; i -= i & -i) sum += bit[i];
35     return sum;
36 }
37
38 /** Example Usage */
39
40 #include <iostream>
41 using namespace std;
42
43 int main() {
44     add(1, 2, 5);
45     add(2, 3, 5);
46     add(3, 5, 10);
47     cout << "BIT values: "; //5 10 15 10 10
48     for (int i = 1; i <= 5; i++)
49         cout << at(i) << " ";
50     cout << "\n";
51     return 0;
52 }

```

3.2.4 Fenwick Tree (Range Update)

```

1  /*
2
3  Description: Using two arrays, a Fenwick tree can be made to
4  support range updates and range queries simultaneously. However,
5  the range updates can only be used to add an increment to all
6  values in a range, not set them to the same value. The latter
7  problem may be solved using a segment tree + lazy propagation.
8  All methods below operate 0-based indices (i.e. indices in the
9  range from 0 to size() - 1, inclusive, are valid).
10
11 Time Complexity: add(), set(), at(), and sum() are all  $O(\log N)$ 
12 on the length of the array. size() is  $O(1)$ .
13

```

```

14 Space Complexity: O(N) storage and auxiliary.
15
16 ~=~=~= Sample Output ~=~=~=
17 BIT values: 15 6 7 -5 4
18 Sum of range [0, 4] is 27.
19
20 */
21
22 #include <vector>
23
24 template<class T> class fenwick_tree {
25     int len;
26     std::vector<T> b1, b2;
27
28     T sum(const std::vector<T> & b, int i) {
29         T res = 0;
30         for (; i != 0; i -= i & -i) res += b[i];
31         return res;
32     }
33
34     void add(std::vector<T> & b, int i, const T & v) {
35         for (; i <= len; i += i & -i) b[i] += v;
36     }
37
38 public:
39     fenwick_tree(int n):
40         len(n + 1), b1(n + 2), b2(n + 2) {}
41
42     //a[i] += v for i = lo..hi, inclusive
43     void add(int lo, int hi, const T & v) {
44         lo++, hi++;
45         add(b1, lo, v);
46         add(b1, hi + 1, -v);
47         add(b2, lo, v * (lo - 1));
48         add(b2, hi + 1, -v * hi);
49     }
50
51     //a[i] = v
52     void set(int i, const T & v) { add(i, i, v - at(i)); }
53
54     //returns sum(a[i] for i = 1..hi inclusive)
55     T sum(int hi) { return sum(b1, hi)*hi - sum(b2, hi); }
56
57     //returns sum(a[i] for i = lo..hi inclusive)
58     T sum(int lo, int hi) { return sum(hi + 1) - sum(lo); }
59
60     inline int size() const { return len - 1; }
61     inline T at(int i) { return sum(i, i); }
62 };
63
64 /** Example Usage */
65
66 #include <iostream>
67 using namespace std;
68
69 int main() {
70     int a[] = {10, 1, 2, 3, 4};
71     fenwick_tree<int> t(5);
72     for (int i = 0; i < 5; i++) t.set(i, a[i]);

```

```

73  t.add(0, 2, 5); //15 6 7 3 4
74  t.set(3, -5);  //15 6 7 -5 4
75  cout << "BIT_values:\n";
76  for (int i = 0; i < t.size(); i++)
77      cout << t.at(i) << "\n";
78  cout << "\nSum of range [0,4] is\n";
79  cout << t.sum(0, 4) << ".\n"; //27
80  return 0;
81  }

```

3.2.5 Fenwick Tree (Map)

```

1  /*
2
3  Description: Using two std::maps to represent the Fenwick tree,
4  there no longer needs to be a restriction on the magnitude of
5  queried indices. All indices in range [0, MAXN] are valid.
6
7  Time Complexity: All functions are  $O(\log^2 \text{MAXN})$ . If the
8  std::map is replaced with an std::unordered_map, then the
9  running time will become  $O(\log \text{MAXN})$  amortized.
10
11 Space Complexity:  $O(n)$  on the number of indices accessed.
12
13 */
14
15 #include <map>
16
17 const int MAXN = 1000000000;
18 std::map<int, int> tmul, tadd;
19
20 void _add(int at, int mul, int add) {
21     for (int i = at; i <= MAXN; i = (i | (i+1))) {
22         tmul[i] += mul;
23         tadd[i] += add;
24     }
25 }
26
27 //a[i] += v for all i = lo..hi, inclusive
28 void add(int lo, int hi, int v) {
29     _add(lo, v, -v * (lo - 1));
30     _add(hi, -v, v * hi);
31 }
32
33 //returns sum(a[i] for i = 1..hi inclusive)
34 int sum(int hi) {
35     int mul = 0, add = 0, start = hi;
36     for (int i = hi; i >= 0; i = (i & (i + 1)) - 1) {
37         if (tmul.find(i) != tmul.end())
38             mul += tmul[i];
39         if (tadd.find(i) != tadd.end())
40             add += tadd[i];
41     }
42     return mul*start + add;
43 }
44
45 //returns sum(a[i] for i = lo..hi inclusive)

```

```

46 int sum(int lo, int hi) {
47     return sum(hi) - sum(lo - 1);
48 }
49
50 //a[i] = v
51 void set(int i, int v) {
52     add(i, i, v - sum(i, i));
53 }
54
55 /** Example Usage */
56
57 #include <iostream>
58 using namespace std;
59
60 int main() {
61     add(500000001, 500000010, 3);
62     add(500000011, 500000015, 5);
63     set(500000000, 10);
64     cout << sum(500000000, 500000015) << "\n"; //65
65     return 0;
66 }

```

3.2.6 2D Fenwick Tree

```

1  /*
2
3  Description: A 2D Fenwick tree is abstractly a 2D array which also
4  supports efficient queries for the sum of values in the rectangle
5  with top-left (1, 1) and bottom-right (r, c). The implementation
6  below has indices accessible in the range [1..xmax][1..ymax].
7
8  Time Complexity: All functions are  $O(\log(xmax)*\log(ymax))$ .
9  Space Complexity:  $O(xmax*ymax)$  storage and auxiliary.
10
11 */
12
13 const int xmax = 100, ymax = 100;
14
15 int a[xmax+1][ymax+1], bit[xmax+1][ymax+1];
16
17 //a[x][y] += v
18 void add(int x, int y, int v) {
19     a[x][y] += v;
20     for (int i = x; i <= xmax; i += i & -i)
21         for (int j = y; j <= ymax; j += j & -j)
22             bit[i][j] += v;
23 }
24
25 //a[x][y] = v
26 void set(int x, int y, int v) {
27     int inc = v - a[x][y];
28     add(x, y, inc);
29 }
30
31 //returns sum(data[1..x][1..y], all inclusive)
32 int sum(int x, int y) {
33     int ret = 0;

```

```

34     for (int i = x; i > 0; i -= i & -i)
35         for (int j = y; j > 0; j -= j & -j)
36             ret += bit[i][j];
37     return ret;
38 }
39
40 //returns sum(data[x1..x2][y1..y2], all inclusive)
41 int sum(int x1, int y1, int x2, int y2) {
42     return sum(x2, y2) + sum(x1 - 1, y1 - 1) -
43         sum(x1 - 1, y2) - sum(x2, y1 - 1);
44 }
45
46 /** Example Usage **/
47
48 #include <cassert>
49 #include <iostream>
50 using namespace std;
51
52 int main() {
53     set(1, 1, 5);
54     set(1, 2, 6);
55     set(2, 1, 7);
56     add(3, 3, 9);
57     add(2, 1, -4);
58     /*
59     5 6 0
60     3 0 0
61     0 0 9
62     */
63     cout << "2D_BIT values:\n";
64     for (int i = 1; i <= 3; i++) {
65         for (int j = 1; j <= 3; j++)
66             cout << a[i][j] << " ";
67         cout << "\n";
68     }
69     assert(sum(1, 1, 1, 2) == 11);
70     assert(sum(1, 1, 2, 1) == 8);
71     assert(sum(1, 1, 3, 3) == 23);
72     return 0;
73 }

```

3.2.7 2D Fenwick Tree (Range Update)

```

1  /*
2
3  Description: A 2D Fenwick tree is abstractly a 2D array which also
4  supports efficient queries for the sum of values in the rectangle
5  with top-left (1, 1) and bottom-right (r, c). The implementation
6  below has indices accessible in the range [0..xmax][0..ymax].
7
8  Time Complexity: All functions are  $O(\log(xmax) * \log(ymax) * \log(N))$ 
9  where N is the number of indices operated on so far. Use an array
10 or an unordered_map instead of a map to remove the  $\log(N)$  factor.
11
12 Space Complexity:  $O(xmax * ymax)$  storage and auxiliary.
13
14 */

```

```

15
16 #include <map>
17 #include <utility>
18
19 template<class T> class fenwick_tree_2d {
20     static const int xmax = 1000000000;
21     static const int ymax = 1000000000;
22
23     std::map<std::pair<int, int>, T> t1, t2, t3, t4;
24
25     template<class Tree>
26     void add(Tree & t, int x, int y, const T & v) {
27         for (int i = x; i <= xmax; i += i & -i)
28             for (int j = y; j <= ymax; j += j & -j)
29                 t[std::make_pair(i, j)] += v;
30     }
31
32     //a[i][j] += v for i = [1,x], j = [1,y]
33     void add_pre(int x, int y, const T & v) {
34         add(t1, 1, 1, v);
35
36         add(t1, 1, y + 1, -v);
37         add(t2, 1, y + 1, v * y);
38
39         add(t1, x + 1, 1, -v);
40         add(t3, x + 1, 1, v * x);
41
42         add(t1, x + 1, y + 1, v);
43         add(t2, x + 1, y + 1, -v * y);
44         add(t3, x + 1, y + 1, -v * x);
45         add(t4, x + 1, y + 1, v * x * y);
46     }
47
48 public:
49     //a[i][j] += v for i = [x1,x2], j = [y1,y2]
50     void add(int x1, int y1, int x2, int y2, const T & v) {
51         x1++; y1++; x2++; y2++;
52         add_pre(x2, y2, v);
53         add_pre(x1 - 1, y2, -v);
54         add_pre(x2, y1 - 1, -v);
55         add_pre(x1 - 1, y1 - 1, v);
56     }
57
58     //a[x][y] += v
59     void add(int x, int y, const T & v) {
60         add(x, y, x, y, v);
61     }
62
63     //a[x][y] = v
64     void set(int x, int y, const T & v) {
65         add(x, y, v - at(x, y));
66     }
67
68     //returns sum(a[i][j] for i = [1,x], j = [1,y])
69     T sum(int x, int y) {
70         x++; y++;
71         T s1 = 0, s2 = 0, s3 = 0, s4 = 0;
72         for (int i = x; i > 0; i -= i & -i)
73             for (int j = y; j > 0; j -= j & -j) {

```

```

74         s1 += t1[std::make_pair(i, j)];
75         s2 += t2[std::make_pair(i, j)];
76         s3 += t3[std::make_pair(i, j)];
77         s4 += t4[std::make_pair(i, j)];
78     }
79     return s1 * x * y + s2 * x + s3 * y + s4;
80 }
81
82 //returns sum(a[i][j] for i = [x1,x2], j = [y1,y2])
83 T sum(int x1, int y1, int x2, int y2) {
84     return sum(x2, y2) + sum(x1 - 1, y1 - 1) -
85         sum(x1 - 1, y2) - sum(x2, y1 - 1);
86 }
87
88 T at(int x, int y) { return sum(x, y, x, y); }
89 };
90
91 /** Example Usage */
92
93 #include <cassert>
94 #include <iostream>
95 using namespace std;
96
97 int main() {
98     fenwick_tree_2d<long long> t;
99     t.set(0, 0, 5);
100    t.set(0, 1, 6);
101    t.set(1, 0, 7);
102    t.add(2, 2, 9);
103    t.add(1, 0, -4);
104    t.add(1, 1, 2, 2, 5);
105    /*
106     5 6 0
107     3 5 5
108     0 5 14
109    */
110    cout << "2D BIT values:\n";
111    for (int i = 0; i < 3; i++) {
112        for (int j = 0; j < 3; j++)
113            cout << t.at(i, j) << " ";
114        cout << "\n";
115    }
116    assert(t.sum(0, 0, 0, 1) == 11);
117    assert(t.sum(0, 0, 1, 0) == 8);
118    assert(t.sum(1, 1, 2, 2) == 29);
119    return 0;
120 }

```

3.3 1D Range Queries

3.3.1 Simple Segment Tree

```

1  /*
2
3  Description: A segment tree is a data structure used for
4  solving the dynamic range query problem, which asks to

```

```

5  determine the minimum (or maximum) value in any given
6  range in an array that is constantly being updated.
7
8  Time Complexity: Assuming merge() is O(1), build is O(n)
9  while query() and update() are O(log n). If merge() is
10 not O(1), then all running times are multiplied by a
11 factor of whatever complexity merge() runs in.
12
13 Space Complexity: O(MAXN). Note that a segment tree with
14 N leaves requires  $2^{(\log_2(N) - 1)} = 4*N$  total nodes.
15
16 Note: This implementation is 0-based, meaning that all
17 indices from 0 to MAXN - 1, inclusive, are accessible.
18
19 ~=~=~= Sample Input ~=~=~=
20 5 10
21 35232
22 390942
23 649675
24 224475
25 18709
26 Q 1 3
27 M 4 475689
28 Q 2 3
29 Q 1 3
30 Q 1 2
31 Q 3 3
32 Q 2 3
33 M 2 645514
34 M 2 680746
35 Q 0 4
36
37 ~=~=~= Sample Output ~=~=~=
38 224475
39 224475
40 224475
41 390942
42 224475
43 224475
44 35232
45
46 */
47
48 const int MAXN = 100000;
49 int N, M, a[MAXN], t[4*MAXN];
50
51 //define your custom nullv and merge() below.
52 //merge(x, nullv) must return x for all x
53
54 const int nullv = 1 << 30;
55
56 inline int merge(int a, int b) { return a < b ? a : b; }
57
58 void build(int n, int lo, int hi) {
59     if (lo == hi) {
60         t[n] = a[lo];
61         return;
62     }
63     build(2*n + 1, lo, (lo + hi)/2);

```



```

64     build(2*n + 2, (lo + hi)/2 + 1, hi);
65     t[n] = merge(t[2*n + 1], t[2*n + 2]);
66 }
67
68 //x and y must be manually set before each call to the
69 //functions below. For query(), [x, y] is the range to
70 //be considered. For update(), a[x] is to be set to y.
71 int x, y;
72
73 //merge(a[i] for i = x..y, inclusive)
74 int query(int n, int lo, int hi) {
75     if (hi < x || lo > y) return nullv;
76     if (lo >= x && hi <= y) return t[n];
77     return merge(query(2*n + 1, lo, (lo + hi) / 2),
78                 query(2*n + 2, (lo + hi) / 2 + 1, hi));
79 }
80
81 //a[x] = y
82 void update(int n, int lo, int hi) {
83     if (hi < x || lo > x) return;
84     if (lo == hi) {
85         t[n] = y;
86         return;
87     }
88     update(2*n + 1, lo, (lo + hi)/2);
89     update(2*n + 2, (lo + hi)/2 + 1, hi);
90     t[n] = merge(t[2*n + 1], t[2*n + 2]);
91 }
92
93 /** Example Usage (wcipeg.com/problem/segtree) */
94
95 #include <stdio>
96
97 int main() {
98     scanf("%d%d", &N, &M);
99     for (int i = 0; i < N; i++) scanf("%d", &a[i]);
100    build(0, 0, N - 1);
101    char op;
102    for (int i = 0; i < M; i++) {
103        scanf("_%c%d%d", &op, &x, &y);
104        if (op == 'Q') {
105            printf("%d\n", query(0, 0, N - 1));
106        } else if (op == 'M') {
107            update(0, 0, N - 1);
108        }
109    }
110    return 0;
111 }

```

3.3.2 Segment Tree

```

1  /*
2
3  Description: A segment tree is a data structure used for
4  solving the dynamic range query problem, which asks to
5  determine the minimum (or maximum) value in any given
6  range in an array that is constantly being updated.

```

```

7
8 Time Complexity: Assuming merge() is O(1), query(),
9 update(), and at() are O(log N). size() is O(1). If
10 merge() is not O(1), then all logarithmic running times
11 are multiplied by a factor of the complexity of merge().
12
13 Space Complexity: O(MAXN). Note that a segment tree with
14 N leaves requires  $2^{(\log_2(N) - 1)} = 4 * N$  total nodes.
15
16 Note: This implementation is 0-based, meaning that all
17 indices from 0 to N - 1, inclusive, are accessible.
18
19 */
20
21 #include <limits> /* std::numeric_limits<T>::min() */
22 #include <vector>
23
24 template<class T> class segment_tree {
25     int len, x, y;
26     std::vector<T> t;
27     T val, *init;
28
29     //define the following yourself. merge(x, nullv) must return x for all x
30     static inline T nullv() { return std::numeric_limits<T>::min(); }
31     static inline T merge(const T & a, const T & b) { return a > b ? a : b; }
32
33     void build(int n, int lo, int hi) {
34         if (lo == hi) {
35             t[n] = init[lo];
36             return;
37         }
38         build(n * 2 + 1, lo, (lo + hi) / 2);
39         build(n * 2 + 2, (lo + hi) / 2 + 1, hi);
40         t[n] = merge(t[n * 2 + 1], t[n * 2 + 2]);
41     }
42
43     void update(int n, int lo, int hi) {
44         if (x < lo || x > hi) return;
45         if (lo == hi) {
46             t[n] = val;
47             return;
48         }
49         update(n * 2 + 1, lo, (lo + hi) / 2);
50         update(n * 2 + 2, (lo + hi) / 2 + 1, hi);
51         t[n] = merge(t[n * 2 + 1], t[n * 2 + 2]);
52     }
53
54     T query(int n, int lo, int hi) {
55         if (hi < x || lo > y) return nullv();
56         if (lo >= x && hi <= y) return t[n];
57         return merge(query(n * 2 + 1, lo, (lo + hi) / 2),
58                     query(n * 2 + 2, (lo + hi) / 2 + 1, hi));
59     }
60
61 public:
62     segment_tree(int n, T * a = 0): len(n), t(4 * n, nullv()) {
63         if (a != 0) {
64             init = a;
65             build(0, 0, len - 1);
66         }
67     }

```

```

66     }
67 }
68
69 //a[i] = v
70 void update(int i, const T & v) {
71     x = i;
72     val = v;
73     update(0, 0, len - 1);
74 }
75
76 //merge(a[i] for i = lo..hi, inclusive)
77 T query(int lo, int hi) {
78     x = lo;
79     y = hi;
80     return query(0, 0, len - 1);
81 }
82
83 inline int size() { return len; }
84 inline T at(int i) { return query(i, i); }
85 };
86
87 /** Example Usage */
88
89 #include <iostream>
90 using namespace std;
91
92 int main() {
93     int arr[5] = {6, -2, 1, 8, 10};
94     segment_tree<int> T(5, arr);
95     T.update(1, 4);
96     cout << "Array contains:";
97     for (int i = 0; i < T.size(); i++)
98         cout << " " << T.at(i);
99     cout << "\nThe max value in the range [0,3] is ";
100     cout << T.query(0, 3) << ".\n"; //8
101     return 0;
102 }

```

3.3.3 Segment Tree (Range Updates)

```

1  /*
2
3  Description: A segment tree is a data structure used for
4  solving the dynamic range query problem, which asks to
5  determine the minimum (or maximum) value in any given
6  range in an array that is constantly being updated.
7  Lazy propagation is a technique applied to segment trees that
8  allows range updates to be carried out in  $O(\log N)$  time. The
9  range updating mechanism is less versatile than the one
10 implemented in the next section.
11
12 Time Complexity: Assuming merge() is  $O(1)$ , query(), update(),
13 at() are  $O(\log(N))$ . If merge() is not constant time, then all
14 running times are multiplied by whatever complexity the merge
15 function runs in.
16
17 Space Complexity:  $O(N)$  on the size of the array. A segment

```

```

18 tree for an array of size N needs  $2^{(\log_2(N)-1)} = 4N$  nodes.
19
20 Note: This implementation is 0-based, meaning that all
21 indices from 0 to size() - 1, inclusive, are accessible.
22
23 */
24
25 #include <limits> /* std::numeric_limits<T>::min() */
26 #include <vector>
27
28 template<class T> class segment_tree {
29     int len, x, y;
30     std::vector<T> tree, lazy;
31     T val, *init;
32
33     //define the following yourself. merge(x, nullv) must return x for all valid x
34     static inline T nullv() { return std::numeric_limits<T>::min(); }
35     static inline T merge(const T & a, const T & b) { return a > b ? a : b; }
36
37     void build(int n, int lo, int hi) {
38         if (lo == hi) {
39             tree[n] = init[lo];
40             return;
41         }
42         build(n * 2 + 1, lo, (lo + hi) / 2);
43         build(n * 2 + 2, (lo + hi) / 2 + 1, hi);
44         tree[n] = merge(tree[n * 2 + 1], tree[n * 2 + 2]);
45     }
46
47     T query(int n, int lo, int hi) {
48         if (x > hi || y < lo) return nullv();
49         if (x <= lo && hi <= y) {
50             if (lazy[n] == nullv()) return tree[n];
51             return tree[n] = lazy[n];
52         }
53         int lchild = n * 2 + 1, rchild = n * 2 + 2;
54         if (lazy[n] != nullv()) {
55             lazy[lchild] = lazy[rchild] = lazy[n];
56             lazy[n] = nullv();
57         }
58         return merge(query(lchild, lo, (lo + hi)/2),
59                     query(rchild, (lo + hi)/2 + 1, hi));
60     }
61
62     void _update(int n, int lo, int hi) {
63         if (x > hi || y < lo) return;
64         if (lo == hi) {
65             tree[n] = val;
66             return;
67         }
68         if (x <= lo && hi <= y) {
69             tree[n] = lazy[n] = merge(lazy[n], val);
70             return;
71         }
72         int lchild = n * 2 + 1, rchild = n * 2 + 2;
73         if (lazy[n] != nullv()) {
74             lazy[lchild] = lazy[rchild] = lazy[n];
75             lazy[n] = nullv();
76         }

```

```

77     _update(lchild, lo, (lo + hi) / 2);
78     _update(rchild, (lo + hi) / 2 + 1, hi);
79     tree[n] = merge(tree[lchild], tree[rchild]);
80 }
81
82 public:
83     segment_tree(int n, T * a = 0):
84         len(n), tree(4 * n, nullv()), lazy(4 * n, nullv()) {
85         if (a != 0) {
86             init = a;
87             build(0, 0, len - 1);
88         }
89     }
90
91     void update(int i, const T & v) {
92         x = y = i;
93         val = v;
94         _update(0, 0, len - 1);
95     }
96
97     //a[i] = v for i = lo..hi, inclusive
98     void update(int lo, int hi, const T & v) {
99         x = lo; y = hi;
100        val = v;
101        _update(0, 0, len - 1);
102    }
103
104    //returns merge(a[i] for i = lo..hi, inclusive)
105    T query(int lo, int hi) {
106        x = lo;
107        y = hi;
108        return query(0, 0, len - 1);
109    }
110
111    inline int size() { return len; }
112    inline T at(int i) { return query(i, i); }
113 };
114
115 /** Example Usage */
116
117 #include <iostream>
118 using namespace std;
119
120 int main() {
121     int arr[5] = {6, 4, 1, 8, 10};
122     segment_tree<int> T(5, arr);
123     cout << "Array contains: "; //6 4 1 8 10
124     for (int i = 0; i < T.size(); i++)
125         cout << " " << T.at(i);
126     cout << "\n";
127     T.update(2, 4, 12);
128     cout << "Array contains: "; //6 4 12 12 12
129     for (int i = 0; i < T.size(); i++)
130         cout << " " << T.at(i);
131     cout << "\nThe max value in the range [0, 3] is ";
132     cout << T.query(0, 3) << ".\n"; //12
133     return 0;
134 }

```

3.3.4 Segment Tree (Fast, Non-recursive)

```

1  /*
2
3  Description: A segment tree is a data structure used for
4  solving the dynamic range query problem, which asks to
5  determine the minimum (or maximum) value in any given
6  range in an array that is constantly being updated.
7  Lazy propagation is a technique applied to segment trees that
8  allows range updates to be carried out in  $O(\log N)$  time.
9
10 Time Complexity: Assuming merge() is  $O(1)$ , query(), update(),
11 at() are  $O(\log(N))$ . If merge() is not constant time, then all
12 running times are multiplied by whatever complexity the merge
13 function runs in.
14
15 Space Complexity:  $O(N)$  on the size of the array.
16
17 Note: This implementation is 0-based, meaning that all
18 indices from 0 to T.size() - 1, inclusive, are accessible.
19
20 */
21
22 #include <algorithm> /* std::fill(), std::max() */
23 #include <stdexcept> /* std::runtime_error */
24 #include <vector>
25
26 template<class T> class segment_tree {
27     //Modify the following 5 methods to implement your custom
28     //operations on the tree. This implements the Add/Max operations.
29     //Operations like Add/Sum, Set/Max can also be implemented.
30     static inline T modify_op(const T & x, const T & y) {
31         return x + y;
32     }
33
34     static inline T query_op(const T & x, const T & y) {
35         return std::max(x, y);
36     }
37
38     static inline T delta_on_segment(const T & delta, int seglen) {
39         if (delta == nullv()) return nullv();
40         //Here you must write a fast equivalent of following slow code:
41         // T result = delta;
42         // for (int i = 1; i < seglen; i++) result = query_op(result, delta);
43         // return result;
44         return delta;
45     }
46
47     static inline T nullv() { return 0; }
48     static inline T initv() { return 0; }
49
50     int length;
51     std::vector<T> value, delta;
52     std::vector<int> len;
53
54     static T join_value_with_delta(const T & val, const T & delta) {
55         return delta == nullv() ? val : modify_op(val, delta);
56     }

```

```

57
58 static T join_deltas(const T & delta1, const T & delta2) {
59     if (delta1 == nullv()) return delta2;
60     if (delta2 == nullv()) return delta1;
61     return modify_op(delta1, delta2);
62 }
63
64 T join_value_with_delta(int i) {
65     return join_value_with_delta(value[i], delta_on_segment(delta[i], len[i]));
66 }
67
68 void push_delta(int i) {
69     int d = 0;
70     while ((i >> d) > 0) d++;
71     for (d -= 2; d >= 0; d--) {
72         int x = i >> d;
73         value[x >> 1] = join_value_with_delta(x >> 1);
74         delta[x] = join_deltas(delta[x], delta[x >> 1]);
75         delta[x ^ 1] = join_deltas(delta[x ^ 1], delta[x >> 1]);
76         delta[x >> 1] = nullv();
77     }
78 }
79
80 public:
81 segment_tree(int n):
82     length(n), value(2 * n), delta(2 * n, nullv()), len(2 * n) {
83     std::fill(len.begin() + n, len.end(), 1);
84     for (int i = 0; i < n; i++) value[i + n] = initv();
85     for (int i = 2 * n - 1; i > 1; i -= 2) {
86         value[i >> 1] = query_op(value[i], value[i ^ 1]);
87         len[i >> 1] = len[i] + len[i ^ 1];
88     }
89 }
90
91 T query(int lo, int hi) {
92     if (lo < 0 || hi >= length || lo > hi)
93         throw std::runtime_error("Invalid_query_range.");
94     push_delta(lo += length);
95     push_delta(hi += length);
96     T res = 0;
97     bool found = false;
98     for (; lo <= hi; lo = (lo + 1) >> 1, hi = (hi - 1) >> 1) {
99         if ((lo & 1) != 0) {
100             res = found ? query_op(res, join_value_with_delta(lo)) :
101                 join_value_with_delta(lo);
102             found = true;
103         }
104         if ((hi & 1) == 0) {
105             res = found ? query_op(res, join_value_with_delta(hi)) :
106                 join_value_with_delta(hi);
107             found = true;
108         }
109     }
110     if (!found) throw std::runtime_error("Not_found.");
111     return res;
112 }
113
114 void modify(int lo, int hi, const T & delta) {
115     if (lo < 0 || hi >= length || lo > hi)

```

```

116         throw std::runtime_error("Invalid modify range.");
117     push_delta(lo += length);
118     push_delta(hi += length);
119     int ta = -1, tb = -1;
120     for (; lo <= hi; lo = (lo + 1) >> 1, hi = (hi - 1) >> 1) {
121         if ((lo & 1) != 0) {
122             this->delta[lo] = join_deltas(this->delta[lo], delta);
123             if (ta == -1) ta = lo;
124         }
125         if ((hi & 1) == 0) {
126             this->delta[hi] = join_deltas(this->delta[hi], delta);
127             if (tb == -1) tb = hi;
128         }
129     }
130     for (int i = ta; i > 1; i >>= 1)
131         value[i >> 1] = query_op(join_value_with_delta(i),
132                                 join_value_with_delta(i ^ 1));
133     for (int i = tb; i > 1; i >>= 1)
134         value[i >> 1] = query_op(join_value_with_delta(i),
135                                 join_value_with_delta(i ^ 1));
136 }
137
138 inline int size() { return length; }
139 inline T at(int i) { return query(i, i); }
140 };
141
142 /** Example Usage */
143
144 #include <iostream>
145 using namespace std;
146
147 int main() {
148     segment_tree<int> T(10);
149     T.modify(0, 0, 10);
150     T.modify(1, 1, 5);
151     T.modify(1, 1, 4);
152     T.modify(2, 2, 7);
153     T.modify(3, 3, 8);
154     cout << T.query(0, 3) << "\n"; //10
155     cout << T.query(1, 3) << "\n"; //9
156     T.modify(0, 9, 5);
157     cout << T.query(0, 9) << "\n"; //15
158     cout << "Array contains: "; //15 14 12 13 5 5 5 5 5 5
159     for (int i = 0; i < T.size(); i++)
160         cout << " " << T.at(i);
161     cout << "\n";
162     return 0;
163 }

```

3.3.5 Implicit Treap

```

1  /*
2
3  Description: A treap is a self-balancing binary search tree that
4  uses randomization to maintain a low height. In this version,
5  it is used emulate the operations of an std::vector with a tradeoff
6  of increasing the running time of push_back() and at() from O(1) to

```



```

7  O(log N), while decreasing the running time of insert() and erase()
8  from O(N) to O(log N). Furthermore, this version supports the same
9  operations as a segment tree with lazy propagation, allowing range
10 updates and queries to be performed in O(log N).
11
12 Time Complexity: Assuming the join functions have constant complexity:
13 insert(), push_back(), erase(), at(), modify(), and query() are all
14 O(log N), while walk() is O(N).
15
16 Space Complexity: O(N) on the size of the array.
17
18 Note: This implementation is 0-based, meaning that all
19 indices from 0 to size() - 1, inclusive, are accessible.
20
21 */
22
23 #include <climits> /* INT_MIN */
24 #include <cstdlib> /* srand(), rand() */
25 #include <ctime> /* time() */
26
27 template<class T> class implicit_treap {
28     //Modify the following 5 functions to implement your custom
29     //operations on the tree. This implements the Add/Max operations.
30     //Operations like Add/Sum, Set/Max can also be implemented.
31     static inline T join_values(const T & a, const T & b) {
32         return a > b ? a : b;
33     }
34
35     static inline T join_deltas(const T & d1, const T & d2) {
36         return d1 + d2;
37     }
38
39     static inline T join_value_with_delta(const T & v, const T & d, int len) {
40         return v + d;
41     }
42
43     static inline T null_delta() { return 0; }
44     static inline T null_value() { return INT_MIN; }
45
46     struct node_t {
47         static inline int rand32() {
48             return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
49         }
50
51         T value, subtree_value, delta;
52         int count, priority;
53         node_t *L, *R;
54
55         node_t(const T & val) {
56             value = subtree_value = val;
57             delta = null_delta();
58             count = 1;
59             L = R = 0;
60             priority = rand32();
61         }
62     } *root;
63
64     static int count(node_t * n) {
65         return n ? n->count : 0;

```

```

66  }
67
68  static T subtree_value(node_t * n) {
69      return n ? n->subtree_value : null_value();
70  }
71
72  static void update(node_t * n) {
73      if (n == 0) return;
74      n->subtree_value = join_values(join_values(subtree_value(n->L), n->value),
75                                   subtree_value(n->R));
76      n->count = 1 + count(n->L) + count(n->R);
77  }
78
79  static void apply_delta(node_t * n, const T & delta) {
80      if (n == 0) return;
81      n->delta = join_deltas(n->delta, delta);
82      n->value = join_value_with_delta(n->value, delta, 1);
83      n->subtree_value = join_value_with_delta(n->subtree_value, delta, n->count);
84  }
85
86  static void push_delta(node_t * n) {
87      if (n == 0) return;
88      apply_delta(n->L, n->delta);
89      apply_delta(n->R, n->delta);
90      n->delta = null_delta();
91  }
92
93  static void merge(node_t *& n, node_t * L, node_t * R) {
94      push_delta(L);
95      push_delta(R);
96      if (L == 0) n = R;
97      else if (R == 0) n = L;
98      else if (L->priority < R->priority)
99          merge(L->R, L->R, R), n = L;
100     else
101         merge(R->L, L, R->L), n = R;
102     update(n);
103 }
104
105 static void split(node_t * n, node_t *& L, node_t *& R, int key) {
106     push_delta(n);
107     if (n == 0) L = R = 0;
108     else if (key <= count(n->L))
109         split(n->L, L, n->L, key), R = n;
110     else
111         split(n->R, n->R, R, key - count(n->L) - 1), L = n;
112     update(n);
113 }
114
115 static void insert(node_t *& n, node_t * item, int idx) {
116     push_delta(n);
117     if (n == 0) n = item;
118     else if (item->priority < n->priority)
119         split(n, item->L, item->R, idx), n = item;
120     else if (idx <= count(n->L))
121         insert(n->L, item, idx);
122     else
123         insert(n->R, item, idx - count(n->L) - 1);
124     update(n);

```

```

125 }
126
127 static T get(node_t * n, int idx) {
128     push_delta(n);
129     if (idx < count(n->L))
130         return get(n->L, idx);
131     else if (idx > count(n->L))
132         return get(n->R, idx - count(n->L) - 1);
133     return n->value;
134 }
135
136 static void erase(node_t *& n, int idx) {
137     push_delta(n);
138     if (idx == count(n->L)) {
139         delete n;
140         merge(n, n->L, n->R);
141     } else if (idx < count(n->L)) {
142         erase(n->L, idx);
143     } else {
144         erase(n->R, idx - count(n->L) - 1);
145     }
146 }
147
148 template<class UnaryFunction>
149 void walk(node_t * n, UnaryFunction f) {
150     if (n == 0) return;
151     push_delta(n);
152     if (n->L) walk(n->L, f);
153     f(n->value);
154     if (n->R) walk(n->R, f);
155 }
156
157 void clean_up(node_t *& n) {
158     if (n == 0) return;
159     clean_up(n->L);
160     clean_up(n->R);
161     delete n;
162 }
163
164 public:
165 implicit_treap(): root(0) { srand(time(0)); }
166 ~implicit_treap() { clean_up(root); }
167
168 int size() const { return count(root); }
169 bool empty() const { return root == 0; }
170
171 //list.insert(list.begin() + idx, val)
172 void insert(int idx, const T & val) {
173     if (idx < 0 || idx > size()) return;
174     node_t * item = new node_t(val);
175     insert(root, item, idx);
176 }
177
178 void push_back(const T & val) {
179     insert(size(), val);
180 }
181
182 //list.erase(list.begin() + idx)
183 void erase(int idx) {

```

```

184     if (idx < 0 || idx >= size()) return;
185     erase(root, idx);
186 }
187
188 T at(int idx) {
189     if (root == 0 || idx < 0 || idx >= size())
190         return null_value();
191     return get(root, idx);
192 }
193
194 template<class UnaryFunction> void walk(UnaryFunction f) {
195     walk(root, f);
196 }
197
198 //for (i = a; i <= b; i++)
199 // list[i] = join_value_with_delta(list[i], delta)
200 void modify(int a, int b, const T & delta) {
201     if (a < 0 || b < 0 || a >= size() || b >= size() || a > b)
202         return;
203     node_t *l1, *r1;
204     split(root, l1, r1, b + 1);
205     node_t *l2, *r2;
206     split(l1, l2, r2, a);
207     apply_delta(r2, delta);
208     node_t *t;
209     merge(t, l2, r2);
210     merge(root, t, r1);
211 }
212
213 //return join_values(list[a..b])
214 T query(int a, int b) {
215     if (a < 0 || b < 0 || a >= size() || b >= size() || a > b)
216         return null_value();
217     node_t *l1, *r1;
218     split(root, l1, r1, b + 1);
219     node_t *l2, *r2;
220     split(l1, l2, r2, a);
221     int res = subtree_value(r2);
222     node_t *t;
223     merge(t, l2, r2);
224     merge(root, t, r1);
225     return res;
226 }
227 };
228
229 /** Example Usage **/
230
231 #include <iostream>
232 using namespace std;
233
234 void print(int x) { cout << x << "\n"; }
235
236 int main() {
237     implicit_treap<int> T;
238     T.push_back(7);
239     T.push_back(8);
240     T.push_back(9);
241     T.insert(1, 5);
242     T.erase(3);

```

```

243     T.walk(print); cout << "\n";    //7 5 8
244     T.modify(0, 2, 2);
245     T.walk(print); cout << "\n";    //9 7 10
246     cout << T.at(1) << "\n";        //7
247     cout << T.query(0, 2) << "\n";  //10
248     cout << T.size() << "\n";      //3
249     return 0;
250 }

```

3.3.6 Sparse Table

```

1  /*
2
3  Description: The static range minimum query problem can be solved
4  using a sparse table data structure. The RMQ for sub arrays of
5  length  $2^k$  is pre-processed using dynamic programming with formula:
6
7   $dp[i][j] = dp[i][j-1]$ , if  $A[dp[i][j-1]] \leq A[dp[i+2^{j-1}-1][j-1]]$ 
8   $dp[i+2^{j-1}-1][j-1]$ , otherwise
9
10 where  $dp[i][j]$  is the index of the minimum value in the sub array
11 starting at  $i$  having length  $2^j$ .
12
13 Time Complexity:  $O(N \log N)$  for build() and  $O(1)$  for min_idx()
14 Space Complexity:  $O(N \log N)$  on the size of the array.
15
16 Note: This implementation is 0-based, meaning that all
17 indices from 0 to  $N - 1$ , inclusive, are valid.
18
19 */
20
21 #include <vector>
22
23 const int MAXN = 100;
24 std::vector<int> logtable, dp[MAXN];
25
26 void build(int n, int a[]) {
27     logtable.resize(n + 1);
28     for (int i = 2; i <= n; i++)
29         logtable[i] = logtable[i >> 1] + 1;
30     for (int i = 0; i < n; i++) {
31         dp[i].resize(logtable[n] + 1);
32         dp[i][0] = i;
33     }
34     for (int k = 1; (1 << k) < n; k++) {
35         for (int i = 0; i + (1 << k) <= n; i++) {
36             int x = dp[i][k - 1];
37             int y = dp[i + (1 << (k - 1))][k - 1];
38             dp[i][k] = a[x] <= a[y] ? x : y;
39         }
40     }
41 }
42
43 //returns index of min element in [lo, hi]
44 int min_idx(int a[], int lo, int hi) {
45     int k = logtable[hi - lo];
46     int x = dp[lo][k];

```

```

47     int y = dp[hi - (1 << k) + 1][k];
48     return a[x] <= a[y] ? x : y;
49 }
50
51 /** Example Usage **/
52
53 #include <iostream>
54 using namespace std;
55
56 int main() {
57     int a[] = {7, -10, 5, 20};
58     build(4, a);
59     cout << min_idx(a, 0, 3) << "\n"; //1
60     return 0;
61 }

```

3.3.7 Square Root Decomposition

```

1  /*
2
3  Description: To solve the dynamic range query problem using
4  square root decomposition, we split an array of size N into
5  sqrt(N) buckets, each bucket of size sqrt(N). As a result,
6  each query and update operation will be sqrt(N) in running time.
7
8  Time Complexity: O(N*sqrt(N)) to construct the initial
9  decomposition. After, query() and update() are O(sqrt N)/call.
10
11 Space Complexity: O(N) for the array. O(sqrt N) for the buckets.
12
13 Note: This implementation is 0-based, meaning that all
14 indices from 0 to N - 1, inclusive, are accessible.
15
16 ~=~=~= Sample Input ~=~=~=
17 5 10
18 35232
19 390942
20 649675
21 224475
22 18709
23 Q 1 3
24 M 4 475689
25 Q 2 3
26 Q 1 3
27 Q 1 2
28 Q 3 3
29 Q 2 3
30 M 2 645514
31 M 2 680746
32 Q 0 4
33
34 ~=~=~= Sample Output ~=~=~=
35 224475
36 224475
37 224475
38 390942
39 224475

```

```

40 224475
41 35232
42
43 */
44
45 #include <cmath> /* sqrt() */
46 #include <limits> /* std::numeric_limits<T>::max() */
47 #include <vector>
48
49 template<class T> class sqrt_decomp {
50     //define the following yourself. merge(x, nullv) must return x for all x
51     static inline T nullv() { return std::numeric_limits<T>::max(); }
52     static inline T merge(const T & a, const T & b) { return a < b ? a : b; }
53
54     int len, blocklen, blocks;
55     std::vector<T> array, block;
56
57 public:
58     sqrt_decomp(int n, T * a = 0): len(n), array(n) {
59         blocklen = (int)sqrt(n);
60         blocks = (n + blocklen - 1) / blocklen;
61         block.resize(blocks);
62         for (int i = 0; i < n; i++)
63             array[i] = a ? a[i] : nullv();
64         for (int i = 0; i < blocks; i++) {
65             int h = (i + 1) * blocklen;
66             if (h > n) h = n;
67             block[i] = nullv();
68             for (int j = i * blocklen; j < h; j++)
69                 block[i] = merge(block[i], array[j]);
70         }
71     }
72
73     void update(int idx, const T & val) {
74         array[idx] = val;
75         int b = idx / blocklen;
76         int h = (b + 1) * blocklen;
77         if (h > len) h = len;
78         block[b] = nullv();
79         for (int i = b * blocklen; i < h; i++)
80             block[b] = merge(block[b], array[i]);
81     }
82
83     T query(int lo, int hi) {
84         T ret = nullv();
85         int lb = ceil((double)lo / blocklen);
86         int hb = (hi + 1) / blocklen - 1;
87         if (lb > hb) {
88             for (int i = lo; i <= hi; i++)
89                 ret = merge(ret, array[i]);
90         } else {
91             int l = lb * blocklen - 1;
92             int h = (hb + 1) * blocklen;
93             for (int i = lo; i <= l; i++)
94                 ret = merge(ret, array[i]);
95             for (int i = lb; i <= hb; i++)
96                 ret = merge(ret, block[i]);
97             for (int i = h; i <= hi; i++)
98                 ret = merge(ret, array[i]);

```

```

99     }
100     return ret;
101 }
102
103 inline int size() { return len; }
104 inline int at(int idx) { return array[idx]; }
105 };
106
107 /** Example Usage (wcipeg.com/problem/segtree) */
108
109 #include <cstdio>
110
111 int N, M, A, B, init[100005];
112
113 int main() {
114     scanf("%d%d", &N, &M);
115     for (int i = 0; i < N; i++) scanf("%d", &init[i]);
116     sqrt_decomp<int> a(N, init);
117     char op;
118     for (int i = 0; i < M; i++) {
119         scanf("%c%d%d", &op, &A, &B);
120         if (op == 'Q') {
121             printf("%d\n", a.query(A, B));
122         } else if (op == 'M') {
123             a.update(A, B);
124         }
125     }
126     return 0;
127 }

```

3.3.8 Interval Tree (Augmented Treap)

```

1  /*
2
3  Description: An interval tree is structure used to store and efficiently
4  query intervals. An interval may be dynamically inserted, and range
5  queries of [lo, hi] may be performed to have the tree report all intervals
6  that intersect with the queried interval. Augmented trees, described in
7  CLRS (2009, Section 14.3: pp. 348354), is one way to represent these
8  intervals. This implementation uses a treap to maintain balance.
9  See: http://en.wikipedia.org/wiki/Interval\_tree#Augmented\_tree
10
11  Time Complexity: On average  $O(\log N)$  for insert() and  $O(k)$  for query(),
12  where N is the number of intervals in the tree and k is the number of
13  intervals that will be reported by each query().
14
15  Space Complexity:  $O(N)$  on the number of intervals in the tree.
16
17  */
18
19 #include <cstdlib>    /* srand() */
20 #include <ctime>      /* time() */
21 #include <utility>    /* std::pair */
22
23 class interval_tree {
24     typedef std::pair<int, int> interval;
25

```



```

26 static bool overlap(const interval & a, const interval & b) {
27     return a.first <= b.second && b.first <= a.second;
28 }
29
30 struct node_t {
31     static inline int rand32() {
32         return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
33     }
34
35     interval i;
36     int maxh, priority;
37     node_t *L, *R;
38
39     node_t(const interval & i) {
40         this->i = i;
41         maxh = i.second;
42         L = R = 0;
43         priority = rand32();
44     }
45
46     void update() {
47         maxh = i.second;
48         if (L != 0 && L->maxh > maxh) maxh = L->maxh;
49         if (R != 0 && R->maxh > maxh) maxh = R->maxh;
50     }
51 } *root;
52
53 static void rotate_l(node_t *& k2) {
54     node_t *k1 = k2->R;
55     k2->R = k1->L;
56     k1->L = k2;
57     k2 = k1;
58     k2->update();
59     k1->update();
60 }
61
62 static void rotate_r(node_t *& k2) {
63     node_t *k1 = k2->L;
64     k2->L = k1->R;
65     k1->R = k2;
66     k2 = k1;
67     k2->update();
68     k1->update();
69 }
70
71 interval i; //temporary
72
73 void insert(node_t *& n) {
74     if (n == 0) { n = new node_t(i); return; }
75     if (i.first < (n->i).first) {
76         insert(n->L);
77         if (n->L->priority < n->priority) rotate_r(n);
78     } else {
79         insert(n->R);
80         if (n->R->priority < n->priority) rotate_l(n);
81     }
82     n->update();
83 }
84

```

```

85     template<class ReportFunction>
86     void query(node_t * n, ReportFunction f) {
87         if (n == 0 || n->maxh < i.first) return;
88         if (overlap(n->i, i)) f(n->i.first, n->i.second);
89         query(n->L, f);
90         query(n->R, f);
91     }
92
93     static void clean_up(node_t * n) {
94         if (n == 0) return;
95         clean_up(n->L);
96         clean_up(n->R);
97         delete n;
98     }
99
100 public:
101     interval_tree(): root(0) { srand(time(0)); }
102     ~interval_tree() { clean_up(root); }
103
104     void insert(int lo, int hi) {
105         i = interval(lo, hi);
106         insert(root);
107     }
108
109     template<class ReportFunction>
110     void query(int lo, int hi, ReportFunction f) {
111         i = interval(lo, hi);
112         query(root, f);
113     }
114 };
115
116 /** Example Usage */
117
118 #include <cassert>
119 #include <iostream>
120 using namespace std;
121
122 void print(int lo, int hi) {
123     cout << "[" << lo << "," << hi << "]\n";
124 }
125
126 int cnt;
127 void count(int lo, int hi) { cnt++; }
128
129 int main() {
130     int N = 6;
131     int intv[6][2] = {{15, 20}, {10, 30}, {17, 19}, {5, 20}, {12, 15}, {30, 40}};
132     interval_tree T;
133     for (int i = 0; i < N; i++) {
134         T.insert(intv[i][0], intv[i][1]);
135     }
136     T.query(10, 20, print); cout << "\n"; // [15,20] [10,30] [5,20] [12,15] [17,19]
137     T.query(0, 5, print); cout << "\n"; // [5,20]
138     T.query(25, 45, print); cout << "\n"; // [10,30] [30,40]
139     //check correctness
140     for (int l = 0; l <= 50; l++) {
141         for (int h = l; h <= 50; h++) {
142             cnt = 0;
143             T.query(l, h, count);

```

```

144     int cnt2 = 0;
145     for (int i = 0; i < N; i++)
146         if (intv[i][0] <= h && l <= intv[i][1])
147             cnt2++;
148     assert(cnt == cnt2);
149 }
150 }
151 return 0;
152 }

```

3.4 2D Range Queries

3.4.1 Quadtree (Simple)

```

1  /*
2
3  Description: A quadtree can be used to dynamically query values
4  of rectangles in a 2D array. In a quadtree, every node has exactly
5  4 children. The following uses a statically allocated array to
6  store the nodes. This is less efficient than a 2D segment tree.
7
8  Time Complexity: For update(), query() and at():  $O(\log(N*M))$  on
9  average and  $O(\sqrt{N*M})$  in the worst case, where N is the number
10 of rows and M is the number of columns in the 2D array.
11
12 Space Complexity:  $O(N*M)$ 
13
14 Note: This implementation is 0-based. Valid indices for
15 all operations are  $[0..xmax][0..ymax]$ 
16
17 */
18
19 #include <climits> /* INT_MIN */
20
21 const int xmax = 100, ymax = 100;
22 int tree[4 * xmax * ymax];
23 int X, Y, X1, X2, Y1, Y2, V; //temporary value to speed up recursion
24
25 //define the following yourself. merge(x, nullv) must return x for all valid x
26 inline int nullv() { return INT_MIN; }
27 inline int merge(int a, int b) { return a > b ? a : b; }
28
29 void update(int n, int x1, int x2, int y1, int y2) {
30     if (X < x1 || X > x2 || Y < y1 || Y > y2) return;
31     if (x1 == x2 && y1 == y2) {
32         tree[n] = V;
33         return;
34     }
35     update(n * 4 + 1, x1, (x1 + x2) / 2, y1, (y1 + y2) / 2);
36     update(n * 4 + 2, x1, (x1 + x2) / 2, (y1 + y2) / 2 + 1, y2);
37     update(n * 4 + 3, (x1 + x2) / 2 + 1, x2, y1, (y1 + y2) / 2);
38     update(n * 4 + 4, (x1 + x2) / 2 + 1, x2, (y1 + y2) / 2 + 1, y2);
39     tree[n] = merge(merge(tree[n * 4 + 1], tree[n * 4 + 2]),
40                     merge(tree[n * 4 + 3], tree[n * 4 + 4]));
41 }
42

```

```

43 void query(int n, int x1, int x2, int y1, int y2) {
44     if (x1 > X2 || x2 < X1 || y2 < Y1 || y1 > Y2 || merge(tree[n], V) == V)
45         return;
46     if (x1 >= X1 && x2 <= X2 && y1 >= Y1 && y2 <= Y2) {
47         V = merge(tree[n], V);
48         return;
49     }
50     query(n * 4 + 1, x1, (x1 + x2) / 2, y1, (y1 + y2) / 2);
51     query(n * 4 + 2, x1, (x1 + x2) / 2, (y1 + y2) / 2 + 1, y2);
52     query(n * 4 + 3, (x1 + x2) / 2 + 1, x2, y1, (y1 + y2) / 2);
53     query(n * 4 + 4, (x1 + x2) / 2 + 1, x2, (y1 + y2) / 2 + 1, y2);
54 }
55
56 void update(int x, int y, int v) {
57     X = x;
58     Y = y;
59     V = v;
60     update(0, 0, xmax - 1, 0, ymax - 1);
61 }
62
63 int query(int x1, int y1, int x2, int y2) {
64     X1 = x1;
65     X2 = x2;
66     Y1 = y1;
67     Y2 = y2;
68     V = nullv();
69     query(0, 0, xmax - 1, 0, ymax - 1);
70     return V;
71 }
72
73 /** Example Usage */
74
75 #include <iostream>
76 using namespace std;
77
78 int main() {
79     int arr[5][5] = {{1, 2, 3, 4, 5},
80                     {5, 4, 3, 2, 1},
81                     {6, 7, 8, 0, 0},
82                     {0, 1, 2, 3, 4},
83                     {5, 9, 9, 1, 2}};
84     for (int r = 0; r < 5; r++)
85         for (int c = 0; c < 5; c++)
86             update(r, c, arr[r][c]);
87     cout << "The maximum value in the rectangle with ";
88     cout << "upper left (0,2) and lower right (3,4) is ";
89     cout << query(0, 2, 3, 4) << ".\n"; //8
90     return 0;
91 }

```

3.4.2 Quadtree

```

1  /*
2
3  Description: A quadtree can be used to dynamically query values
4  of rectangles in a 2D array. In a quadtree, every node has exactly
5  4 children. The following uses dynamically allocated memory to

```

```

6 store the nodes, which allows arbitrarily large indices to exist
7 without affecting the performance of operations.
8
9 Time Complexity: For update(), query() and at():  $O(\log(N*M))$  on
10 average and  $O(\sqrt{N*M})$  in the worst case, where N is the number
11 of rows and M is the number of columns in the 2D array.
12
13 Space Complexity:  $O(N*M)$ 
14
15 Note: This implementation is 0-based. Valid indices for
16 all operations are [0..XMAX][0..YMAX]
17
18 */
19
20 #include <algorithm> /* std::max(), std::min() */
21 #include <limits> /* std::numeric_limits<T>::min() */
22
23 template<class T> class quadtree {
24     //these can be set to large values without affecting your memory usage!
25     static const int xmax = 1000000000;
26     static const int ymax = 1000000000;
27
28     //define the following yourself. merge(x, nullv) must return x for all valid x
29     static inline T nullv() { return std::numeric_limits<T>::min(); }
30     static inline T merge(const T & a, const T & b) { return a > b ? a : b; }
31
32     int X, Y, X1, X2, Y1, Y2; T V; //temp vals for speed
33
34     struct node_t {
35         node_t * child[4];
36         int x1, x2, y1, y2;
37         T value;
38
39         node_t(int x, int y) {
40             x1 = x2 = x;
41             y1 = y2 = y;
42             child[0] = child[1] = child[2] = child[3] = 0;
43             value = nullv();
44         }
45     } *root;
46
47     void update(node_t *& n, int x1, int x2, int y1, int y2) {
48         if (X < x1 || X > x2 || Y < y1 || Y > y2) return;
49         if (n == 0) n = new node_t(X, Y);
50         if (x1 == x2 && y1 == y2) {
51             n->value = V;
52             return;
53         }
54         int xmid = (x1 + x2)/2, ymid = (y1 + y2)/2;
55         update(n->child[0], x1, xmid, y1, ymid);
56         update(n->child[1], xmid + 1, x2, y1, ymid);
57         update(n->child[2], x1, xmid, ymid + 1, y2);
58         update(n->child[3], xmid + 1, x2, ymid + 1, y2);
59         for (int i = 0; i < 4; i++) {
60             if (!n->child[i] || n->child[i]->value == nullv()) continue;
61             n->x1 = std::min(n->x1, n->child[i]->x1);
62             n->x2 = std::max(n->x2, n->child[i]->x2);
63             n->y1 = std::min(n->y1, n->child[i]->y1);
64             n->y2 = std::max(n->y2, n->child[i]->y2);

```

```

65     n->value = merge(n->value, n->child[i]->value);
66 }
67 }
68
69 void query(node_t * n) {
70     if (n == 0 || n->x1 > X2 || n->x2 < X1 || n->y2 < Y1 || n->y1 > Y2 ||
71         merge(n->value, V) == V)
72         return;
73     if (n->x1 >= X1 && n->y1 >= Y1 && n->x2 <= X2 && n->y2 <= Y2) {
74         V = merge(V, n->value);
75         return;
76     }
77     for (int i = 0; i < 4; i++) query(n->child[i]);
78 }
79
80 static void clean_up(node_t * n) {
81     if (n == 0) return;
82     for (int i = 0; i < 4; i++) clean_up(n->child[i]);
83     delete n;
84 }
85
86 public:
87     quadtree() { root = 0; }
88     ~quadtree() { clean_up(root); }
89
90     void update(int x, int y, const T & v) {
91         X = x;
92         Y = y;
93         V = v;
94         update(root, 0, xmax - 1, 0, ymax - 1);
95     }
96
97     T query(int x1, int y1, int x2, int y2) {
98         X1 = x1;
99         X2 = x2;
100        Y1 = y1;
101        Y2 = y2;
102        V = nullv();
103        query(root);
104        return V;
105    }
106
107    T at(int x, int y) {
108        return query(x, y, x, y);
109    }
110 };
111
112 /** Example Usage */
113
114 #include <iostream>
115 using namespace std;
116
117 int main() {
118     int arr[5][5] = {{1, 2, 3, 4, 5},
119                     {5, 4, 3, 2, 1},
120                     {6, 7, 8, 0, 0},
121                     {0, 1, 2, 3, 4},
122                     {5, 9, 9, 1, 2}};
123     quadtree<int> T;

```

```

124     for (int r = 0; r < 5; r++)
125         for (int c = 0; c < 5; c++)
126             T.update(r, c, arr[r][c]);
127     cout << "The maximum value in the rectangle with ";
128     cout << "upper left (0,2) and lower right (3,4) is ";
129     cout << T.query(0, 2, 3, 4) << ".\n"; //8
130     return 0;
131 }

```

3.4.3 2D Segment Tree

```

1  /*
2
3  Description: A quadtree is a segment tree but with 4 children
4  per node, making its running time proportional to the square
5  root of the number of leaves. However, a 2D segment tree is a
6  segment tree of segment trees, making its running time
7  proportional to the log of its size. The following implementation
8  is a highly optimized implementation with features such as
9  coordinate compression and path compression.
10
11  Time Complexity: O(log(xmax)*log(ymax)) for update(), query(),
12  and at() operations. size() is O(1).
13
14  Space Complexity: Left as an exercise for the reader.
15
16  Note: This implementation is 0-based. Valid indices for
17  all operations are [0..xmax][0..ymax]
18
19  */
20
21  #include <limits> /* std::numeric_limits<T>::min() */
22
23  template<class T> class segment_tree_2d {
24      //these can be set to large values without affecting your memory usage!
25      static const int xmax = 1000000000;
26      static const int ymax = 1000000000;
27
28      //define the following yourself. merge(x, nullv) must return x for all valid x
29      static inline T nullv() { return std::numeric_limits<T>::min(); }
30      static inline T merge(const T & a, const T & b) { return a > b ? a : b; }
31
32      struct layer2_node {
33          int lo, hi;
34          layer2_node *L, *R;
35          T value;
36          layer2_node(int l, int h) : lo(l), hi(h), L(0), R(0) {}
37      };
38
39      struct layer1_node {
40          layer1_node *L, *R;
41          layer2_node l2;
42          layer1_node() : L(0), R(0), l2(0, ymax) {}
43      } *root;
44
45      void update2(layer2_node * node, int Q, const T & v) {
46          int lo = node->lo, hi = node->hi, mid = (lo + hi)/2;

```

```

47     if (lo + 1 == hi) {
48         node->value = v;
49         return;
50     }
51     layer2_node *tgt = Q < mid ? node->L : node->R;
52     if (tgt == 0) {
53         tgt = new layer2_node(Q, Q + 1);
54         tgt->value = v;
55     } else if (tgt->lo <= Q && Q < tgt->hi) {
56         update2(tgt, Q, v);
57     } else {
58         do {
59             (Q < mid ? hi : lo) = mid;
60             mid = (lo + hi)/2;
61         } while ((Q < mid) == (tgt->lo < mid));
62         layer2_node *nnode = new layer2_node(lo, hi);
63         (tgt->lo < mid ? nnode->L : nnode->R) = tgt;
64         tgt = nnode;
65         update2(nnode, Q, v);
66     }
67     node->value = merge(node->L ? node->L->value : nullv(),
68                       node->R ? node->R->value : nullv());
69 }
70
71 T query2(layer2_node * nd, int A, int B) {
72     if (nd == 0 || B <= nd->lo || nd->hi <= A) return nullv();
73     if (A <= nd->lo && nd->hi <= B) return nd->value;
74     return merge(query2(nd->L, A, B), query2(nd->R, A, B));
75 }
76
77 void update1(layer1_node * node, int lo, int hi, int x, int y, const T & v) {
78     if (lo + 1 == hi) update2(&node->l2, y, v);
79     else {
80         int mid = (lo + hi)/2;
81         layer1_node *& nnode = x < mid ? node->L : node->R;
82         (x < mid ? hi : lo) = mid;
83         if (nnode == 0) nnode = new layer1_node();
84         update1(nnode, lo, hi, x, y, v);
85         update2(&node->l2, y, merge(
86             node->L ? query2(&node->L->l2, y, y + 1) : nullv(),
87             node->R ? query2(&node->R->l2, y, y + 1) : nullv()
88         ));
89     }
90 }
91
92 T query1(layer1_node * nd, int lo, int hi, int A1, int B1, int A2, int B2) {
93     if (nd == 0 || B1 <= lo || hi <= A1) return nullv();
94     if (A1 <= lo && hi <= B1) return query2(&nd->l2, A2, B2);
95     int mid = (lo + hi) / 2;
96     return merge(query1(nd->L, lo, mid, A1, B1, A2, B2),
97                 query1(nd->R, mid, hi, A1, B1, A2, B2));
98 }
99
100 void clean_up2(layer2_node * n) {
101     if (n == 0) return;
102     clean_up2(n->L);
103     clean_up2(n->R);
104     delete n;
105 }

```



```

106
107 void clean_up1(layer1_node * n) {
108     if (n == 0) return;
109     clean_up2(n->l2.L);
110     clean_up2(n->l2.R);
111     clean_up1(n->L);
112     clean_up1(n->R);
113     delete n;
114 }
115
116 public:
117 segment_tree_2d() { root = new layer1_node(); }
118 ~segment_tree_2d() { clean_up1(root); }
119
120 void update(int x, int y, const T & v) {
121     update1(root, 0, xmax, x, y, v);
122 }
123
124 T query(int x1, int y1, int x2, int y2) {
125     return query1(root, 0, xmax, x1, x2 + 1, y1, y2 + 1);
126 }
127
128 T at(int x, int y) {
129     return query(x, y, x, y);
130 }
131 };
132
133 /** Example Usage */
134
135 #include <iostream>
136 using namespace std;
137
138 int main() {
139     int arr[5][5] = {{1, 2, 3, 4, 5},
140                     {5, 4, 3, 2, 1},
141                     {6, 7, 8, 0, 0},
142                     {0, 1, 2, 3, 4},
143                     {5, 9, 9, 1, 2}};
144     segment_tree_2d<int> T;
145     for (int r = 0; r < 5; r++)
146         for (int c = 0; c < 5; c++)
147             T.update(r, c, arr[r][c]);
148     cout << "The maximum value in the rectangle with ";
149     cout << "upper_left(0,2) and lower_right(3,4) is ";
150     cout << T.query(0, 2, 3, 4) << ".\n"; //8
151     return 0;
152 }

```

3.4.4 K-d Tree (2D Range Query)

```

1  /*
2
3  Description: k-d tree (short for k-dimensional tree) is a space-
4  partitioning data structure for organizing points in a k-
5  dimensional space. The following implementation supports
6  counting the number of points in rectangular ranges after the
7  tree has been build.

```

```

8
9 Time Complexity:  $O(N \log N)$  for build(), where N is the number
10 of points in the tree. count() is  $O(\sqrt{N})$ .
11
12 Space Complexity:  $O(N)$  on the number of points.
13
14 */
15
16 #include <algorithm> /* nth_element(), max(), min() */
17 #include <climits>   /* INT_MIN, INT_MAX */
18 #include <utility>   /* std::pair */
19 #include <vector>
20
21 class kd_tree {
22     typedef std::pair<int, int> point;
23
24     static inline bool cmp_x(const point & a, const point & b) {
25         return a.first < b.first;
26     }
27
28     static inline bool cmp_y(const point & a, const point & b) {
29         return a.second < b.second;
30     }
31
32     std::vector<int> tx, ty, cnt, minx, miny, maxx, maxy;
33     int x1, y1, x2, y2; //temporary values to speed up recursion
34
35     void build(int lo, int hi, bool div_x, point P[]) {
36         if (lo >= hi) return;
37         int mid = (lo + hi) >> 1;
38         std::nth_element(P + lo, P + mid, P + hi, div_x ? cmp_x : cmp_y);
39         tx[mid] = P[mid].first;
40         ty[mid] = P[mid].second;
41         cnt[mid] = hi - lo;
42         minx[mid] = INT_MAX; miny[mid] = INT_MAX;
43         maxx[mid] = INT_MIN; maxy[mid] = INT_MIN;
44         for (int i = lo; i < hi; i++) {
45             minx[mid] = std::min(minx[mid], P[i].first);
46             maxx[mid] = std::max(maxx[mid], P[i].first);
47             miny[mid] = std::min(miny[mid], P[i].second);
48             maxy[mid] = std::max(maxy[mid], P[i].second);
49         }
50         build(lo, mid, !div_x, P);
51         build(mid + 1, hi, !div_x, P);
52     }
53
54     int count(int lo, int hi) {
55         if (lo >= hi) return 0;
56         int mid = (lo + hi) >> 1;
57         int ax = minx[mid], ay = miny[mid];
58         int bx = maxx[mid], by = maxy[mid];
59         if (ax > x2 || x1 > bx || ay > y2 || y1 > by) return 0;
60         if (x1 <= ax && bx <= x2 && y1 <= ay && by <= y2) return cnt[mid];
61         int res = count(lo, mid) + count(mid + 1, hi);
62         res += (x1 <= tx[mid] && tx[mid] <= x2 && y1 <= ty[mid] && ty[mid] <= y2);
63         return res;
64     }
65
66 public:

```

```

67  kd_tree(int n, point P[]): tx(n), ty(n), cnt(n),
68      minx(n), miny(n), maxx(n), maxy(n) {
69      build(0, n, true, P);
70  }
71
72  int count(int x1, int y1, int x2, int y2) {
73      this->x1 = x1;
74      this->y1 = y1;
75      this->x2 = x2;
76      this->y2 = y2;
77      return count(0, tx.size());
78  }
79 };
80
81 /** Example Usage **/
82
83 #include <cassert>
84 using namespace std;
85
86 int main() {
87     pair<int, int> P[4];
88     P[0] = make_pair(0, 0);
89     P[1] = make_pair(10, 10);
90     P[2] = make_pair(0, 10);
91     P[3] = make_pair(10, 0);
92     kd_tree t(4, P);
93     assert(t.count(0, 0, 10, 9) == 2);
94     assert(t.count(0, 0, 10, 10) == 4);
95     return 0;
96 }

```

3.4.5 K-d Tree (Nearest Neighbor)

```

1  /*
2
3  Description: k-d tree (short for k-dimensional tree) is a space-
4  partitioning data structure for organizing points in a k-
5  dimensional space. The following implementation supports
6  querying the nearest neighboring point to (x, y) in terms of
7  Euclidean distance after the tree has been build. Note that
8  a point is not considered its own neighbour if it already exists
9  in the tree.
10
11 Time Complexity:  $O(N \log N)$  for build(), where N is the number of
12 points in the tree. nearest_neighbor_id() is  $O(\log(N))$  on average.
13
14 Space Complexity:  $O(N)$  on the number of points.
15
16 */
17
18 #include <algorithm> /* nth_element(), max(), min(), swap() */
19 #include <climits> /* INT_MIN, INT_MAX */
20 #include <utility>
21 #include <vector>
22
23 class kd_tree {
24     typedef std::pair<int, int> point;

```

```

25
26 static inline bool cmp_x(const point & a, const point & b) {
27     return a.first < b.first;
28 }
29
30 static inline bool cmp_y(const point & a, const point & b) {
31     return a.second < b.second;
32 }
33
34 std::vector<int> tx, ty;
35 std::vector<bool> div_x;
36
37 void build(int lo, int hi, point P[]) {
38     if (lo >= hi) return;
39     int mid = (lo + hi) >> 1;
40     int minx = INT_MAX, maxx = INT_MIN;
41     int miny = INT_MAX, maxy = INT_MIN;
42     for (int i = lo; i < hi; i++) {
43         minx = std::min(minx, P[i].first);
44         maxx = std::max(maxx, P[i].first);
45         miny = std::min(miny, P[i].second);
46         maxy = std::max(maxy, P[i].second);
47     }
48     div_x[mid] = (maxx - minx) >= (maxy - miny);
49     std::nth_element(P + lo, P + mid, P + hi, div_x[mid] ? cmp_x : cmp_y);
50     tx[mid] = P[mid].first;
51     ty[mid] = P[mid].second;
52     if (lo + 1 == hi) return;
53     build(lo, mid, P);
54     build(mid + 1, hi, P);
55 }
56
57 long long min_dist;
58 int min_dist_id, x, y;
59
60 void nearest_neighbor(int lo, int hi) {
61     if (lo >= hi) return;
62     int mid = (lo + hi) >> 1;
63     int dx = x - tx[mid], dy = y - ty[mid];
64     long long d = dx*(long long)dx + dy*(long long)dy;
65     if (min_dist > d && d) {
66         min_dist = d;
67         min_dist_id = mid;
68     }
69     if (lo + 1 == hi) return;
70     int delta = div_x[mid] ? dx : dy;
71     long long delta2 = delta*(long long)delta;
72     int l1 = lo, r1 = mid, l2 = mid + 1, r2 = hi;
73     if (delta > 0) std::swap(l1, l2), std::swap(r1, r2);
74     nearest_neighbor(l1, r1);
75     if (delta2 < min_dist) nearest_neighbor(l2, r2);
76 }
77
78 public:
79 kd_tree(int N, point P[]) {
80     tx.resize(N);
81     ty.resize(N);
82     div_x.resize(N);
83     build(0, N, P);

```

```

84     }
85
86     int nearest_neighbor_id(int x, int y) {
87         this->x = x; this->y = y;
88         min_dist = LLONG_MAX;
89         nearest_neighbor(0, tx.size());
90         return min_dist_id;
91     }
92 };
93
94 /** Example Usage */
95
96 #include <iostream>
97 using namespace std;
98
99 int main() {
100     pair<int, int> P[3];
101     P[0] = make_pair(0, 2);
102     P[1] = make_pair(0, 3);
103     P[2] = make_pair(-1, 0);
104     kd_tree T(3, P);
105     int res = T.nearest_neighbor_id(0, 0);
106     cout << P[res].first << "□" << P[res].second << "\n"; //-1, 0
107     return 0;
108 }

```

3.4.6 R-Tree (Nearest Segment)

```

1  /*
2
3  Description: R-trees are tree data structures used for spatial
4  access methods, i.e., for indexing multi-dimensional information
5  such as geographical coordinates, rectangles or polygons. The
6  following implementation supports querying of the nearest line
7  segment to a point after a tree of line segments have been built.
8
9  Time Complexity:  $O(N \log N)$  for build(), where N is the number of
10 points in the tree. nearest_neighbor_id() is  $O(\log(N))$  on average.
11
12 Space Complexity:  $O(N)$  on the number of points.
13
14 */
15
16 #include <algorithm> /* nth_element(), max(), min(), swap() */
17 #include <cfloat>    /* DBL_MAX */
18 #include <climits>  /* INT_MIN, INT_MAX */
19 #include <vector>
20
21 struct segment { int x1, y1, x2, y2; };
22
23 class r_tree {
24
25     static inline bool cmp_x(const segment & a, const segment & b) {
26         return a.x1 + a.x2 < b.x1 + b.x2;
27     }
28
29     static inline bool cmp_y(const segment & a, const segment & b) {

```

```

30     return a.y1 + a.y2 < b.y1 + b.y2;
31 }
32
33 std::vector<segment> s;
34 std::vector<int> minx, maxx, miny, maxy;
35
36 void build(int lo, int hi, bool div_x, segment s[]) {
37     if (lo >= hi) return;
38     int mid = (lo + hi) >> 1;
39     std::nth_element(s + lo, s + mid, s + hi, div_x ? cmp_x : cmp_y);
40     this->s[mid] = s[mid];
41     for (int i = lo; i < hi; i++) {
42         minx[mid] = std::min(minx[mid], std::min(s[i].x1, s[i].x2));
43         miny[mid] = std::min(miny[mid], std::min(s[i].y1, s[i].y2));
44         maxx[mid] = std::max(maxx[mid], std::max(s[i].x1, s[i].x2));
45         maxy[mid] = std::max(maxy[mid], std::max(s[i].y1, s[i].y2));
46     }
47     build(lo, mid, !div_x, s);
48     build(mid + 1, hi, !div_x, s);
49 }
50
51 double min_dist;
52 int min_dist_id, x, y;
53
54 void nearest_neighbor(int lo, int hi, bool div_x) {
55     if (lo >= hi) return;
56     int mid = (lo + hi) >> 1;
57     double pdist = point_to_segment_squared(x, y, s[mid]);
58     if (min_dist > pdist) {
59         min_dist = pdist;
60         min_dist_id = mid;
61     }
62     long long delta = div_x ? 2*x - s[mid].x1 - s[mid].x2 :
63                         2*y - s[mid].y1 - s[mid].y2;
64     if (delta <= 0) {
65         nearest_neighbor(lo, mid, !div_x);
66         if (mid + 1 < hi) {
67             int mid1 = (mid + hi + 1) >> 1;
68             long long dist = div_x ? seg_dist(x, minx[mid1], maxx[mid1]) :
69                                 seg_dist(y, miny[mid1], maxy[mid1]);
70             if (dist*dist < min_dist) nearest_neighbor(mid + 1, hi, !div_x);
71         }
72     } else {
73         nearest_neighbor(mid + 1, hi, !div_x);
74         if (lo < mid) {
75             int mid1 = (lo + mid) >> 1;
76             long long dist = div_x ? seg_dist(x, minx[mid1], maxx[mid1]) :
77                                 seg_dist(y, miny[mid1], maxy[mid1]);
78             if (dist*dist < min_dist) nearest_neighbor(lo, mid, !div_x);
79         }
80     }
81 }
82
83 static double point_to_segment_squared(int x, int y, const segment & s) {
84     long long dx = s.x2 - s.x1, dy = s.y2 - s.y1;
85     long long px = x - s.x1, py = y - s.y1;
86     long long square_dist = dx*dx + dy*dy;
87     long long dot_product = dx*px + dy*py;
88     if (dot_product <= 0 || square_dist == 0) return px*px + py*py;

```

```

89     if (dot_product >= square_dist)
90         return (px - dx)*(px - dx) + (py - dy)*(py - dy);
91     double q = (double)dot_product/square_dist;
92     return (px - q*dx)*(px - q*dx) + (py - q*dy)*(py - q*dy);
93 }
94
95 static inline int seg_dist(int v, int lo, int hi) {
96     return v <= lo ? lo - v : (v >= hi ? v - hi : 0);
97 }
98
99 public:
100 r_tree(int N, segment s[]) {
101     this->s.resize(N);
102     minx.assign(N, INT_MAX);
103     maxx.assign(N, INT_MIN);
104     miny.assign(N, INT_MAX);
105     maxy.assign(N, INT_MIN);
106     build(0, N, true, s);
107 }
108
109 int nearest_neighbor_id(int x, int y) {
110     min_dist = DBL_MAX;
111     this->x = x; this->y = y;
112     nearest_neighbor(0, s.size(), true);
113     return min_dist_id;
114 }
115 };
116
117 /** Example Usage */
118
119 #include <iostream>
120 using namespace std;
121
122 int main() {
123     segment s[4];
124     s[0] = (segment){0, 0, 0, 4};
125     s[1] = (segment){0, 4, 4, 4};
126     s[2] = (segment){4, 4, 4, 0};
127     s[3] = (segment){4, 0, 0, 0};
128     r_tree t(4, s);
129     int id = t.nearest_neighbor_id(-1, 2);
130     cout << s[id].x1 << " " << s[id].y1 << " " <<
131         s[id].x2 << " " << s[id].y2 << "\n"; //0 0 0 4
132     return 0;
133 }

```

3.4.7 2D Range Tree

```

1  /*
2
3  Description: A range tree is an ordered tree data structure to
4  hold a list of points. It allows all points within a given range
5  to be reported efficiently. Specifically, for a given query, a
6  range tree will report *all* points that lie in the given range.
7  Note that the initial array passed to construct the tree will be
8  sorted, and all resulting query reports will pertain to the
9  indices of points in the sorted array.

```

```

10
11 Time Complexity: A range tree can build() in  $O(N \log^{d-1}(N))$ 
12 and query() in  $O(\log^d(n) + k)$ , where  $N$  is the number of points
13 stored in the tree,  $d$  is the dimension of each point and  $k$  is the
14 number of points reported by a given query. Thus for this 2D case
15 build() is  $O(N \log N)$  and query() is  $O(\log^2(N) + k)$ .
16
17 Space Complexity:  $O(N \log^{d-1}(N))$  for a  $d$ -dimensional range tree.
18 Thus for this 2D case, the space complexity is  $O(N \log N)$ .
19
20 */
21
22 #include <algorithm> /* lower_bound(), merge(), sort() */
23 #include <utility>   /* std::pair */
24 #include <vector>
25
26 class range_tree_2d {
27     typedef std::pair<int, int> point;
28
29     std::vector<point> P;
30     std::vector<std::vector<point> > seg;
31
32     static inline bool comp1(const point & a, const point & b) {
33         return a.second < b.second;
34     }
35
36     static inline bool comp2(const point & a, int v) {
37         return a.second < v;
38     }
39
40     void build(int n, int lo, int hi) {
41         if (P[lo].first == P[hi].first) {
42             for (int i = lo; i <= hi; i++)
43                 seg[n].push_back(point(i, P[i].second));
44             return;
45         }
46         int l = n * 2 + 1, r = n * 2 + 2;
47         build(l, lo, (lo + hi)/2);
48         build(r, (lo + hi)/2 + 1, hi);
49         seg[n].resize(seg[l].size() + seg[r].size());
50         std::merge(seg[l].begin(), seg[l].end(), seg[r].begin(), seg[r].end(),
51                   seg[n].begin(), comp1);
52     }
53
54     int xl, xh, yl, yh;
55
56     template<class ReportFunction>
57     void query(int n, int lo, int hi, ReportFunction f) {
58         if (P[hi].first < xl || P[lo].first > xh) return;
59         if (xl <= P[lo].first && P[hi].first <= xh) {
60             if (!seg[n].empty() && yh >= yl) {
61                 std::vector<point>::iterator it;
62                 it = std::lower_bound(seg[n].begin(), seg[n].end(), yl, comp2);
63                 for (; it != seg[n].end(); ++it) {
64                     if (it->second > yh) break;
65                     f(it->first); //or report P[it->first], the actual point
66                 }
67             }
68             else if (lo != hi) {

```



```

69     query(n * 2 + 1, lo, (lo + hi) / 2, f);
70     query(n * 2 + 2, (lo + hi) / 2 + 1, hi, f);
71 }
72 }
73
74 public:
75     range_tree_2d(int n, point init[]): seg(4 * n + 1) {
76         std::sort(init, init + n);
77         P = std::vector<point>(init, init + n);
78         build(0, 0, n - 1);
79     }
80
81     template<class ReportFunction>
82     void query(int x1, int y1, int x2, int y2, ReportFunction f) {
83         x1 = x1; xh = x2;
84         y1 = y1; yh = y2;
85         query(0, 0, P.size() - 1, f);
86     }
87 };
88
89 /** Example Usage (wcipeg.com/problem/box1) */
90
91 #include <bitset>
92 #include <cstdio>
93 using namespace std;
94
95 int N, M; bitset<200005> b;
96 pair<int, int> pts[200005];
97 int x1[200005], y1[200005];
98 int x2[200005], y2[200005];
99
100 void mark(int i) {
101     b[i] = true;
102 }
103
104 int main() {
105     scanf("%d%d", &N, &M);
106     for (int i = 0; i < N; i++)
107         scanf("%d%d%d", x1 + i, y1 + i, x2 + i, y2 + i);
108     for (int i = 0; i < M; i++)
109         scanf("%d", &pts[i].first, &pts[i].second);
110     range_tree_2d t(M, pts);
111     for (int i = 0; i < N; i++)
112         t.query(x1[i], y1[i], x2[i], y2[i], mark);
113     printf("%d\n", b.count());
114     return 0;
115 }

```

3.5 Search Trees and Alternatives

3.5.1 Binary Search Tree

```

1  /*
2
3  Description: A binary search tree (BST) is a node-based binary tree data
4  structure where the left sub-tree of every node has keys less than the

```

5 node's key and the right sub-tree of every node has keys greater than the
 6 node's key. A BST may become degenerate like a linked list resulting in
 7 an $O(N)$ running time per operation. A self-balancing binary search tree
 8 such as a randomized treap prevents the occurrence of this known worst case.

9
 10 Note: The following implementation is used similar to an `std::map`. In order
 11 to make it behave like an `std::set`, modify the code to remove the value
 12 associated with each node. In order to make it behave like an `std::multiset`
 13 or `std::multimap`, make appropriate changes with key comparisons (e.g.
 14 change `(k < n->key)` to `(k <= n->key)` in search conditions).

15
 16 Time Complexity: `insert()`, `erase()` and `find()` are $O(\log(N))$ on average,
 17 but $O(N)$ at worst if the tree becomes degenerate. Speed can be improved
 18 by randomizing insertion order if it doesn't matter. `walk()` is $O(N)$.

19
 20 Space Complexity: $O(N)$ on the number of nodes.

21
 22 `*/`

```

23
24 template<class key_t, class val_t> class binary_search_tree {
25     struct node_t {
26         key_t key;
27         val_t val;
28         node_t *L, *R;
29
30         node_t(const key_t & k, const val_t & v) {
31             key = k;
32             val = v;
33             L = R = 0;
34         }
35     } *root;
36
37     int num_nodes;
38
39     static bool insert(node_t *& n, const key_t & k, const val_t & v) {
40         if (n == 0) {
41             n = new node_t(k, v);
42             return true;
43         }
44         if (k < n->key) return insert(n->L, k, v);
45         if (n->key < k) return insert(n->R, k, v);
46         return false; //already exists
47     }
48
49     static bool erase(node_t *& n, const key_t & key) {
50         if (n == 0) return false;
51         if (key < n->key) return erase(n->L, key);
52         if (n->key < key) return erase(n->R, key);
53         if (n->L == 0) {
54             node_t *temp = n->R;
55             delete n;
56             n = temp;
57         } else if (n->R == 0) {
58             node_t *temp = n->L;
59             delete n;
60             n = temp;
61         } else {
62             node_t *temp = n->R, *parent = 0;
63             while (temp->L != 0) {

```

```

64     parent = temp;
65     temp = temp->L;
66 }
67 n->key = temp->key;
68 n->val = temp->val;
69 if (parent != 0)
70     return erase(parent->L, parent->L->key);
71 return erase(n->R, n->R->key);
72 }
73 return true;
74 }
75
76 template<class BinaryFunction>
77 static void walk(node_t * n, BinaryFunction f) {
78     if (n == 0) return;
79     walk(n->L, f);
80     f(n->key, n->val);
81     walk(n->R, f);
82 }
83
84 static void clean_up(node_t * n) {
85     if (n == 0) return;
86     clean_up(n->L);
87     clean_up(n->R);
88     delete n;
89 }
90
91 public:
92     binary_search_tree(): root(0), num_nodes(0) {}
93     ~binary_search_tree() { clean_up(root); }
94     int size() const { return num_nodes; }
95     bool empty() const { return root == 0; }
96
97     bool insert(const key_t & key, const val_t & val) {
98         if (insert(root, key, val)) {
99             num_nodes++;
100             return true;
101         }
102         return false;
103     }
104
105     bool erase(const key_t & key) {
106         if (erase(root, key)) {
107             num_nodes--;
108             return true;
109         }
110         return false;
111     }
112
113     template<class BinaryFunction> void walk(BinaryFunction f) {
114         walk(root, f);
115     }
116
117     val_t * find(const key_t & key) {
118         for (node_t *n = root; n != 0; ) {
119             if (n->key == key) return &(n->val);
120             n = (key < n->key ? n->L : n->R);
121         }
122         return 0; //key not found

```

```

123     }
124 };
125
126 /** Example Usage */
127
128 #include <iostream>
129 using namespace std;
130
131 void printch(int k, char v) { cout << v; }
132
133 int main() {
134     binary_search_tree<int, char> T;
135     T.insert(2, 'b');
136     T.insert(1, 'a');
137     T.insert(3, 'c');
138     T.insert(5, 'e');
139     T.insert(4, 'x');
140     *T.find(4) = 'd';
141     cout << "In-order: ";
142     T.walk(printch); //abcde
143     cout << "\nRemoving node with key 3...";
144     cout << (T.erase(3) ? "Success!" : "Failed");
145     cout << "\n";
146     return 0;
147 }

```

3.5.2 Treap

```

1  /*
2
3  Description: A binary search tree (BST) is a node-based binary tree data
4  structure where the left sub-tree of every node has keys less than the
5  node's key and the right sub-tree of every node has keys greater than the
6  node's key. A BST may be come degenerate like a linked list resulting in
7  an O(N) running time per operation. A self-balancing binary search tree
8  such as a randomized treap prevents the occurrence of this known worst case.
9
10 Treaps use randomly generated priorities to reduce the height of the
11 tree. We assume that the rand() function in <stdlib> is 16-bits, and
12 call it twice to generate a 32-bit number. For the treap to be
13 effective, the range of the randomly generated numbers should be
14 between 0 and around the number of elements in the treap.
15
16 Note: The following implementation is used similar to an std::map. In order
17 to make it behave like an std::set, modify the code to remove the value
18 associated with each node. In order to make it behave like an std::multiset
19 or std::multimap, make appropriate changes with key comparisons (e.g.
20 change (k < n->key) to (k <= n->key) in search conditions).
21
22 Time Complexity: insert(), erase(), and find() are O(log(N)) on average
23 and O(N) in the worst case. Despite the technically O(N) worst case,
24 such cases are still extremely difficult to trigger, making treaps
25 very practice in many programming contest applications. walk() is O(N).
26
27 Space Complexity: O(N) on the number of nodes.
28
29 */

```

```

30
31 #include <cstdlib> /* srand(), rand() */
32 #include <ctime>   /* time() */
33
34 template<class key_t, class val_t> class treap {
35     struct node_t {
36         static inline int rand32() {
37             return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
38         }
39
40         key_t key;
41         val_t val;
42         int priority;
43         node_t *L, *R;
44
45         node_t(const key_t & k, const val_t & v): key(k), val(v), L(0), R(0) {
46             priority = rand32();
47         }
48     } *root;
49
50     int num_nodes;
51
52     static void rotate_l(node_t *& k2) {
53         node_t *k1 = k2->R;
54         k2->R = k1->L;
55         k1->L = k2;
56         k2 = k1;
57     }
58
59     static void rotate_r(node_t *& k2) {
60         node_t *k1 = k2->L;
61         k2->L = k1->R;
62         k1->R = k2;
63         k2 = k1;
64     }
65
66     static bool insert(node_t *& n, const key_t & k, const val_t & v) {
67         if (n == 0) {
68             n = new node_t(k, v);
69             return true;
70         }
71         if (k < n->key && insert(n->L, k, v)) {
72             if (n->L->priority < n->priority) rotate_r(n);
73             return true;
74         } else if (n->key < k && insert(n->R, k, v)) {
75             if (n->R->priority < n->priority) rotate_l(n);
76             return true;
77         }
78         return false;
79     }
80
81     static bool erase(node_t *& n, const key_t & k) {
82         if (n == 0) return false;
83         if (k < n->key) return erase(n->L, k);
84         if (k > n->key) return erase(n->R, k);
85         if (n->L == 0 || n->R == 0) {
86             node_t *temp = n;
87             n = (n->L != 0) ? n->L : n->R;
88             delete temp;

```

```

89     return true;
90 }
91 if (n->L->priority < n->R->priority) {
92     rotate_r(n);
93     return erase(n->R, k);
94 }
95 rotate_l(n);
96 return erase(n->L, k);
97 }
98
99 template<class BinaryFunction>
100 static void walk(node_t * n, BinaryFunction f) {
101     if (n == 0) return;
102     walk(n->L, f);
103     f(n->key, n->val);
104     walk(n->R, f);
105 }
106
107 static void clean_up(node_t * n) {
108     if (n == 0) return;
109     clean_up(n->L);
110     clean_up(n->R);
111     delete n;
112 }
113
114 public:
115     treap(): root(0), num_nodes(0) { srand(time(0)); }
116     ~treap() { clean_up(root); }
117     int size() const { return num_nodes; }
118     bool empty() const { return root == 0; }
119
120     bool insert(const key_t & key, const val_t & val) {
121         if (insert(root, key, val)) {
122             num_nodes++;
123             return true;
124         }
125         return false;
126     }
127
128     bool erase(const key_t & key) {
129         if (erase(root, key)) {
130             num_nodes--;
131             return true;
132         }
133         return false;
134     }
135
136     template<class BinaryFunction> void walk(BinaryFunction f) {
137         walk(root, f);
138     }
139
140     val_t * find(const key_t & key) {
141         for (node_t *n = root; n != 0; ) {
142             if (n->key == key) return &(n->val);
143             n = (key < n->key ? n->L : n->R);
144         }
145         return 0; //key not found
146     }
147 };

```

```

148
149 /** Example Usage **/
150
151 #include <cassert>
152 #include <iostream>
153 using namespace std;
154
155 void printch(int k, char v) { cout << v; }
156
157 int main() {
158     treap<int, char> T;
159     T.insert(2, 'b');
160     T.insert(1, 'a');
161     T.insert(3, 'c');
162     T.insert(5, 'e');
163     T.insert(4, 'x');
164     *T.find(4) = 'd';
165     cout << "In-order: ";
166     T.walk(printch); //abcde
167     cout << "\nRemoving node with key 3...";
168     cout << (T.erase(3) ? "Success!" : "Failed");
169     cout << "\n";
170
171     //stress test - runs in <0.5 seconds
172     //insert keys in an order that would break a normal BST
173     treap<int, int> T2;
174     for (int i = 0; i < 1000000; i++)
175         T2.insert(i, i*1337);
176     for (int i = 0; i < 1000000; i++)
177         assert(*T2.find(i) == i*1337);
178     return 0;
179 }

```

3.5.3 Size Balanced Tree (Order Statistics)

```

1  /*
2
3  Description: A binary search tree (BST) is a node-based binary tree data
4  structure where the left sub-tree of every node has keys less than the
5  node's key and the right sub-tree of every node has keys greater than the
6  node's key. A BST may be come degenerate like a linked list resulting in
7  an O(N) running time per operation. A self-balancing binary search tree
8  such as a randomized treap prevents the occurence of this known worst case.
9
10 The size balanced tree is a data structure first published in 2007 by
11 Chinese student Chen Qifeng. The tree is rebalanced by examining the sizes
12 of each node's subtrees. It is popular amongst Chinese OI competitors due
13 to its speed, simplicity to implement, and ability to double up as an
14 ordered statistics tree if necessary.
15 For more info, see: http://wcipeg.com/wiki/Size\_Balanced\_Tree
16
17 An ordered statistics tree is a BST that supports additional operations:
18 - Select(i): find the i-th smallest element stored in the tree
19 - Rank(x): find the rank of element x in the tree,
20 i.e. its index in the sorted list of elements of the tree
21 For more info, see: http://en.wikipedia.org/wiki/Order\_statistic\_tree
22

```

```

23 Note: The following implementation is used similar to an std::map. In order
24 to make it behave like an std::set, modify the code to remove the value
25 associated with each node. Making a size balanced tree behave like an
26 std::multiset or std::multimap is a more complex issue. Refer to the
27 articles above and determine the correct way to preserve the binary search
28 tree property with maintain() if equivalent keys are allowed.
29
30 Time Complexity: insert(), erase(), find(), select() and rank() are
31  $O(\log N)$  on the number of elements in the tree. walk() is  $O(N)$ .
32
33 Space Complexity:  $O(N)$  on the number of nodes in the tree.
34
35 */
36
37 #include <stdexcept> /* std::runtime_error */
38 #include <utility>   /* pair */
39
40 template<class key_t, class val_t> class size_balanced_tree {
41     struct node_t {
42         key_t key;
43         val_t val;
44         int size;
45         node_t * c[2];
46
47         node_t(const key_t & k, const val_t & v) {
48             key = k, val = v;
49             size = 1;
50             c[0] = c[1] = 0;
51         }
52
53         void update() {
54             size = 1;
55             if (c[0]) size += c[0]->size;
56             if (c[1]) size += c[1]->size;
57         }
58     } *root;
59
60     static inline int size(node_t * n) {
61         return n ? n->size : 0;
62     }
63
64     static void rotate(node_t *& n, bool d) {
65         node_t * p = n->c[d];
66         n->c[d] = p->c[!d];
67         p->c[!d] = n;
68         n->update();
69         p->update();
70         n = p;
71     }
72
73     static void maintain(node_t *& n, bool d) {
74         if (n == 0 || n->c[d] == 0) return;
75         node_t *& p = n->c[d];
76         if (size(p->c[d]) > size(n->c[!d])) {
77             rotate(n, d);
78         } else if (size(p->c[!d]) > size(n->c[!d])) {
79             rotate(p, !d);
80             rotate(n, d);
81         } else return;

```



```

82     maintain(n->c[0], 0);
83     maintain(n->c[1], 1);
84     maintain(n, 0);
85     maintain(n, 1);
86 }
87
88 static void insert(node_t *&n, const key_t &k, const val_t &v) {
89     if (n == 0) {
90         n = new node_t(k, v);
91         return;
92     }
93     if (k < n->key) {
94         insert(n->c[0], k, v);
95         maintain(n, 0);
96     } else if (n->key < k) {
97         insert(n->c[1], k, v);
98         maintain(n, 1);
99     } else return;
100     n->update();
101 }
102
103 static void erase(node_t *&n, const key_t &k) {
104     if (n == 0) return;
105     bool d = k < n->key;
106     if (k < n->key) {
107         erase(n->c[0], k);
108     } else if (n->key < k) {
109         erase(n->c[1], k);
110     } else {
111         if (n->c[1] == 0 || n->c[0] == 0) {
112             delete n;
113             n = n->c[1] == 0 ? n->c[0] : n->c[1];
114             return;
115         }
116         node_t *p = n->c[1];
117         while (p->c[0] != 0) p = p->c[0];
118         n->key = p->key;
119         erase(n->c[1], p->key);
120     }
121     maintain(n, d);
122     n->update();
123 }
124
125 template<class BinaryFunction>
126 static void walk(node_t *n, BinaryFunction f) {
127     if (n == 0) return;
128     walk(n->c[0], f);
129     f(n->key, n->val);
130     walk(n->c[1], f);
131 }
132
133 static std::pair<key_t, val_t> select(node_t *&n, int k) {
134     int r = size(n->c[0]);
135     if (k < r) return select(n->c[0], k);
136     if (k > r) return select(n->c[1], k - r - 1);
137     return std::make_pair(n->key, n->val);
138 }
139
140 static int rank(node_t *n, const key_t &k) {

```

```

141     if (n == 0)
142         throw std::runtime_error("Cannot rank key not in tree.");
143     int r = size(n->c[0]);
144     if (k < n->key) return rank(n->c[0], k);
145     if (n->key < k) return rank(n->c[1], k) + r + 1;
146     return r;
147 }
148
149 static void clean_up(node_t * n) {
150     if (n == 0) return;
151     clean_up(n->c[0]);
152     clean_up(n->c[1]);
153     delete n;
154 }
155
156 public:
157     size_balanced_tree() : root(0) {}
158     ~size_balanced_tree() { clean_up(root); }
159     int size() { return size(root); }
160     bool empty() const { return root == 0; }
161
162     void insert(const key_t & key, const val_t & val) {
163         insert(root, key, val);
164     }
165
166     void erase(const key_t & key) {
167         erase(root, key);
168     }
169
170     template<class BinaryFunction> void walk(BinaryFunction f) {
171         walk(root, f);
172     }
173
174     val_t * find(const key_t & key) {
175         for (node_t *n = root; n != 0; ) {
176             if (n->key == key) return &(n->val);
177             n = (key < n->key ? n->c[0] : n->c[1]);
178         }
179         return 0; //key not found
180     }
181
182     std::pair<key_t, val_t> select(int k) {
183         if (k >= size(root))
184             throw std::runtime_error("k must be smaller size of tree.");
185         return select(root, k);
186     }
187
188     int rank(const key_t & key) {
189         return rank(root, key);
190     }
191 };
192
193 /** Example Usage */
194
195 #include <cassert>
196 #include <iostream>
197 using namespace std;
198
199 void printch(int k, char v) { cout << v; }

```

```

200
201 int main() {
202     size_balanced_tree<int, char> T;
203     T.insert(2, 'b');
204     T.insert(1, 'a');
205     T.insert(3, 'c');
206     T.insert(5, 'e');
207     T.insert(4, 'x');
208     *T.find(4) = 'd';
209     cout << "In-order:\n";
210     T.walk(printch);           //abcde
211     T.erase(3);
212     cout << "\nRank_of_2:\n" << T.rank(2); //1
213     cout << "\nRank_of_5:\n" << T.rank(5); //3
214     cout << "\nValue_of_3rd_smallest_key:\n";
215     cout << T.select(2).second; //d
216     cout << "\n";
217
218     //stress test - runs in <1 second
219     //insert keys in an order that would break a normal BST
220     size_balanced_tree<int, int> T2;
221     for (int i = 0; i < 1000000; i++)
222         T2.insert(i, i*1337);
223     for (int i = 0; i < 1000000; i++)
224         assert(*T2.find(i) == i*1337);
225     return 0;
226 }

```

3.5.4 Hashmap (Chaining)

```

1  /*
2
3  Description: A hashmap (std::unordered_map in C++11) is an
4  alternative to a binary search tree. Hashmaps use more memory than
5  BSTs, but are usually more efficient. The following implementation
6  uses the chaining method to handle collisions. You can use the
7  hash algorithms provided in the example, or define your own.
8
9  Time Complexity: insert(), remove(), find(), are O(1) amortized.
10 rehash() is O(N).
11
12 Space Complexity: O(N) on the number of entries.
13
14 */
15
16 #include <list>
17
18 template<class key_t, class val_t, class Hash> class hashmap {
19     struct entry_t {
20         key_t key;
21         val_t val;
22         entry_t(const key_t & k, const val_t & v): key(k), val(v) {}
23     };
24
25     std::list<entry_t> * table;
26     int table_size, map_size;
27

```

```

28  /**
29   * This doubles the table size, then rehashes every entry.
30   * Rehashing is expensive; it is strongly suggested for the
31   * table to be constructed with a large size to avoid rehashing.
32   */
33  void rehash() {
34      std::list<entry_t> * old = table;
35      int old_size = table_size;
36      table_size = 2*table_size;
37      table = new std::list<entry_t>[table_size];
38      map_size = 0;
39      typename std::list<entry_t>::iterator it;
40      for (int i = 0; i < old_size; i++)
41          for (it = old[i].begin(); it != old[i].end(); ++it)
42              insert(it->key, it->val);
43      delete[] old;
44  }
45
46  public:
47      hashmap(int size = 1024): table_size(size), map_size(0) {
48          table = new std::list<entry_t>[table_size];
49      }
50
51      ~hashmap() { delete[] table; }
52      int size() const { return map_size; }
53
54      void insert(const key_t & key, const val_t & val) {
55          if (find(key) != 0) return;
56          if (map_size >= table_size) rehash();
57          unsigned int i = Hash()(key) % table_size;
58          table[i].push_back(entry_t(key, val));
59          map_size++;
60      }
61
62      void remove(const key_t & key) {
63          unsigned int i = Hash()(key) % table_size;
64          typename std::list<entry_t>::iterator it = table[i].begin();
65          while (it != table[i].end() && it->key != key) ++it;
66          if (it == table[i].end()) return;
67          table[i].erase(it);
68          map_size--;
69      }
70
71      val_t * find(const key_t & key) {
72          unsigned int i = Hash()(key) % table_size;
73          typename std::list<entry_t>::iterator it = table[i].begin();
74          while (it != table[i].end() && it->key != key) ++it;
75          if (it == table[i].end()) return 0;
76          return &(it->val);
77      }
78
79      val_t & operator [] (const key_t & key) {
80          val_t * ret = find(key);
81          if (ret != 0) return *ret;
82          insert(key, val_t());
83          return *find(key);
84      }
85  };
86

```

```

87  /*** Examples of Hash Algorithm Definitions ***/
88
89  #include <string>
90
91  struct class_hash {
92      unsigned int operator () (int key) {
93          return class_hash()((unsigned int)key);
94      }
95
96      unsigned int operator () (long long key) {
97          return class_hash()((unsigned long long)key);
98      }
99
100     //Knuth's multiplicative method (one-to-one)
101     unsigned int operator () (unsigned int key) {
102         return key * 2654435761u; //or just return key
103     }
104
105     //Jenkins's 64-bit hash
106     unsigned int operator () (unsigned long long key) {
107         key += ~(key << 32); key ^= (key >> 22);
108         key += ~(key << 13); key ^= (key >> 8);
109         key += (key << 3); key ^= (key >> 15);
110         key += ~(key << 27); key ^= (key >> 31);
111         return key;
112     }
113
114     //Jenkins's one-at-a-time hash
115     unsigned int operator () (const std::string & key) {
116         unsigned int hash = 0;
117         for (unsigned int i = 0; i < key.size(); i++) {
118             hash += ((hash += key[i]) << 10);
119             hash ^= (hash >> 6);
120         }
121         hash ^= ((hash += (hash << 3)) >> 11);
122         return hash + (hash << 15);
123     }
124 };
125
126 /*** Example Usage ***/
127
128 #include <iostream>
129 using namespace std;
130
131 int main() {
132     hashmap<string, int, class_hash> M;
133     M["foo"] = 1;
134     M.insert("bar", 2);
135     cout << M["foo"] << M["bar"] << endl; //prints 12
136     cout << M["baz"] << M["qux"] << endl; //prints 00
137     M.remove("foo");
138     cout << M.size() << endl; //prints 3
139     cout << M["foo"] << M["bar"] << endl; //prints 02
140     return 0;
141 }

```

3.5.5 Skip List (Probabilistic)

```

1  /*
2
3  Description: A skip list is an alternative to binary search trees.
4  Fast search is made possible by maintaining a linked hierarchy of
5  subsequences, each skipping over fewer elements. Searching starts
6  in the sparsest subsequence until two consecutive elements have
7  been found, one smaller and one larger than the element searched for.
8  Skip lists are generally slower than binary search trees, but can
9  be easier to implement. The following version uses randomized levels.
10
11 Time Complexity: insert(), erase(), count() and find() are  $O(\log(N))$ 
12 on average, but  $O(N)$  in the worst case. walk() is  $O(N)$ .
13
14 Space Complexity:  $O(N)$  on the number of elements inserted on average,
15 but  $O(N \log N)$  in the worst case.
16
17 */
18
19 #include <cmath>    /* log() */
20 #include <stdlib.h> /* rand(), srand() */
21 #include <cstring> /* memset() */
22 #include <ctime>   /* time() */
23
24 template<class key_t, class val_t> struct skip_list {
25     static const int MAX_LEVEL = 32; //~ log2(max # of keys)
26
27     static int random_level() { //geometric distribution
28         static const float P = 0.5;
29         int lvl = log((float)rand()/RAND_MAX)/log(1.0 - P);
30         return lvl < MAX_LEVEL ? lvl : MAX_LEVEL;
31     }
32
33     struct node_t {
34         key_t key;
35         val_t val;
36         node_t **next;
37
38         node_t(int level, const key_t & k, const val_t & v) {
39             next = new node_t * [level + 1];
40             memset(next, 0, sizeof(node_t)*(level + 1));
41             key = k;
42             val = v;
43         }
44
45         ~node_t() { delete[] next; }
46     } *head, *update[MAX_LEVEL + 1];
47
48     int level, num_nodes;
49
50     skip_list() {
51         srand(time(0));
52         head = new node_t(MAX_LEVEL, key_t(), val_t());
53         level = num_nodes = 0;
54     }
55
56     ~skip_list() { delete head; }
57     int size() { return num_nodes; }
58     bool empty() { return num_nodes == 0; }
59     int count(const key_t & k) { return find(k) != 0; }

```

```

60
61 void insert(const key_t & k, const val_t & v) {
62     node_t * n = head;
63     memset(update, 0, sizeof(node_t)*(MAX_LEVEL + 1));
64     for (int i = level; i >= 0; i--) {
65         while (n->next[i] && n->next[i]->key < k) n = n->next[i];
66         update[i] = n;
67     }
68     n = n->next[0];
69     if (!n || n->key != k) {
70         int lvl = random_level();
71         if (lvl > level) {
72             for (int i = level + 1; i <= lvl; i++) update[i] = head;
73             level = lvl;
74         }
75         n = new node_t(lvl, k, v);
76         num_nodes++;
77         for (int i = 0; i <= lvl; i++) {
78             n->next[i] = update[i]->next[i];
79             update[i]->next[i] = n;
80         }
81     } else if (n && n->key == k && n->val != v) {
82         n->val = v;
83     }
84 }
85
86 void erase(const key_t & k) {
87     node_t * n = head;
88     memset(update, 0, sizeof(node_t)*(MAX_LEVEL + 1));
89     for (int i = level; i >= 0; i--) {
90         while (n->next[i] && n->next[i]->key < k) n = n->next[i];
91         update[i] = n;
92     }
93     n = n->next[0];
94     if (n->key == k) {
95         for (int i = 0; i <= level; i++) {
96             if (update[i]->next[i] != n) break;
97             update[i]->next[i] = n->next[i];
98         }
99         delete n;
100         num_nodes--;
101         while (level > 0 && !head->next[level]) level--;
102     }
103 }
104
105 val_t * find(const key_t & k) {
106     node_t * n = head;
107     for (int i = level; i >= 0; i--)
108         while (n->next[i] && n->next[i]->key < k)
109             n = n->next[i];
110     n = n->next[0];
111     if (n && n->key == k) return &(n->val);
112     return 0; //not found
113 }
114
115 template<class BinaryFunction> void walk(BinaryFunction f) {
116     node_t *n = head->next[0];
117     while (n) {
118         f(n->key, n->val);

```

```

119     n = n->next[0];
120 }
121 }
122 };
123
124 /** Example Usage: Random Tests */
125
126 #include <cassert>
127 #include <iostream>
128 #include <map>
129 using namespace std;
130
131 int main() {
132     map<int, int> m;
133     skip_list<int, int> s;
134     for (int i = 0; i < 50000; i++) {
135         int op = rand() % 3;
136         int val1 = rand(), val2 = rand();
137         if (op == 0) {
138             m[val1] = val2;
139             s.insert(val1, val2);
140         } else if (op == 1) {
141             if (!m.count(val1)) continue;
142             m.erase(val1);
143             s.erase(val1);
144         } else if (op == 2) {
145             assert(s.count(val1) == (int)m.count(val1));
146             if (m.count(val1)) {
147                 assert(m[val1] == *s.find(val1));
148             }
149         }
150     }
151     return 0;
152 }

```

3.6 Tree Data Structures

3.6.1 Heavy-Light Decomposition

```

1  /*
2
3  Description: Given an undirected, connected graph that is a tree, the
4  heavy-light decomposition (HLD) on the graph is a partitioning of the
5  vertices into disjoint paths to later support dynamic modification and
6  querying of values on paths between pairs of vertices.
7  See: http://wcipeg.com/wiki/Heavy-light\_decomposition
8  and: http://blog.anudeep2011.com/heavy-light-decomposition/
9  To support dynamic adding and removal of edges, see link/cut tree.
10
11 Note: The adjacency list tree[] that is passed to the constructor must
12 not be changed afterwards in order for modify() and query() to work.
13
14 Time Complexity: O(N) for the constructor and O(log N) in the worst
15 case for both modify() and query(), where N is the number of vertices.
16
17 Space Complexity: O(N) on the number of vertices in the tree.

```



```

18
19 */
20
21 #include <algorithm> /* std::max(), std::min() */
22 #include <climits>   /* INT_MIN */
23 #include <vector>
24
25 template<class T> class heavy_light {
26     //true if you want values on edges, false if you want values on vertices
27     static const bool VALUES_ON_EDGES = true;
28
29     //Modify the following 6 functions to implement your custom
30     //operations on the tree. This implements the Add/Max operations.
31     //Operations like Add/Sum, Set/Max can also be implemented.
32     static inline T modify_op(const T & x, const T & y) {
33         return x + y;
34     }
35
36     static inline T query_op(const T & x, const T & y) {
37         return std::max(x, y);
38     }
39
40     static inline T delta_on_segment(const T & delta, int seglen) {
41         if (delta == null_delta()) return null_delta();
42         //Here you must write a fast equivalent of following slow code:
43         // T result = delta;
44         // for (int i = 1; i < seglen; i++) result = query_op(result, delta);
45         // return result;
46         return delta;
47     }
48
49     static inline T init_value() { return 0; }
50     static inline T null_delta() { return 0; }
51     static inline T null_value() { return INT_MIN; }
52
53     static inline T join_value_with_delta(const T & v, const T & delta) {
54         return delta == null_delta() ? v : modify_op(v, delta);
55     }
56
57     static T join_deltas(const T & delta1, const T & delta2) {
58         if (delta1 == null_delta()) return delta2;
59         if (delta2 == null_delta()) return delta1;
60         return modify_op(delta1, delta2);
61     }
62
63     int counter, paths;
64     std::vector<int> *adj;
65     std::vector<std::vector<T> > value, delta;
66     std::vector<std::vector<int> > len;
67     std::vector<int> size, parent, tin, tout;
68     std::vector<int> path, pathlen, pathpos, pathroot;
69
70     void precompute_dfs(int u, int p) {
71         tin[u] = counter++;
72         parent[u] = p;
73         size[u] = 1;
74         for (int j = 0, v; j < (int)adj[u].size(); j++) {
75             if ((v = adj[u][j]) == p) continue;
76             precompute_dfs(v, u);

```

```

77     size[u] += size[v];
78 }
79 tout[u] = counter++;
80 }
81
82 int new_path(int u) {
83     pathroot[paths] = u;
84     return paths++;
85 }
86
87 void build_paths(int u, int path) {
88     this->path[u] = path;
89     pathpos[u] = pathlen[path]++;
90     for (int j = 0, v; j < (int)adj[u].size(); j++) {
91         if ((v = adj[u][j]) == parent[u]) continue;
92         build_paths(v, 2*size[v] >= size[u] ? path : new_path(v));
93     }
94 }
95
96 inline T join_value_with_delta0(int path, int i) {
97     return join_value_with_delta(value[path][i],
98         delta_on_segment(delta[path][i], len[path][i]));
99 }
100
101 void push_delta(int path, int i) {
102     int d = 0;
103     while ((i >> d) > 0) d++;
104     for (d -= 2; d >= 0; d--) {
105         int x = i >> d;
106         value[path][x >> 1] = join_value_with_delta0(path, x >> 1);
107         delta[path][x] = join_deltas(delta[path][x], delta[path][x >> 1]);
108         delta[path][x ^ 1] = join_deltas(delta[path][x ^ 1], delta[path][x >> 1]);
109         delta[path][x >> 1] = null_delta();
110     }
111 }
112
113 T query(int path, int a, int b) {
114     push_delta(path, a += value[path].size() >> 1);
115     push_delta(path, b += value[path].size() >> 1);
116     T res = null_value();
117     for (; a <= b; a = (a + 1) >> 1, b = (b - 1) >> 1) {
118         if ((a & 1) != 0)
119             res = query_op(res, join_value_with_delta0(path, a));
120         if ((b & 1) == 0)
121             res = query_op(res, join_value_with_delta0(path, b));
122     }
123     return res;
124 }
125
126 void modify(int path, int a, int b, const T & delta) {
127     push_delta(path, a += value[path].size() >> 1);
128     push_delta(path, b += value[path].size() >> 1);
129     int ta = -1, tb = -1;
130     for (; a <= b; a = (a + 1) >> 1, b = (b - 1) >> 1) {
131         if ((a & 1) != 0) {
132             this->delta[path][a] = join_deltas(this->delta[path][a], delta);
133             if (ta == -1) ta = a;
134         }
135         if ((b & 1) == 0) {

```

```

136         this->delta[path][b] = join_deltas(this->delta[path][b], delta);
137         if (tb == -1) tb = b;
138     }
139 }
140 for (int i = ta; i > 1; i >>= 1)
141     value[path][i >> 1] = query_op(join_value_with_delta0(path, i),
142                                     join_value_with_delta0(path, i ^ 1));
143 for (int i = tb; i > 1; i >>= 1)
144     value[path][i >> 1] = query_op(join_value_with_delta0(path, i),
145                                     join_value_with_delta0(path, i ^ 1));
146 }
147
148 inline bool is_ancestor(int p, int ch) {
149     return tin[p] <= tin[ch] && tout[ch] <= tout[p];
150 }
151
152 public:
153 heavy_light(int N, std::vector<int> tree[]): size(N), parent(N),
154 tin(N), tout(N), path(N), pathlen(N), pathpos(N), pathroot(N) {
155     adj = tree;
156     counter = paths = 0;
157     precompute_dfs(0, -1);
158     build_paths(0, new_path(0));
159     value.resize(paths);
160     delta.resize(paths);
161     len.resize(paths);
162     for (int i = 0; i < paths; i++) {
163         int m = pathlen[i];
164         value[i].assign(2*m, init_value());
165         delta[i].assign(2*m, null_delta());
166         len[i].assign(2*m, 1);
167         for (int j = 2*m - 1; j > 1; j -= 2) {
168             value[i][j >> 1] = query_op(value[i][j], value[i][j ^ 1]);
169             len[i][j >> 1] = len[i][j] + len[i][j ^ 1];
170         }
171     }
172 }
173
174 T query(int a, int b) {
175     T res = null_value();
176     for (int root; !is_ancestor(root = pathroot[path[a]], b); a = parent[root])
177         res = query_op(res, query(path[a], 0, pathpos[a]));
178     for (int root; !is_ancestor(root = pathroot[path[b]], a); b = parent[root])
179         res = query_op(res, query(path[b], 0, pathpos[b]));
180     if (VALUES_ON_EDGES && a == b) return res;
181     return query_op(res, query(path[a], std::min(pathpos[a], pathpos[b]) +
182                                     VALUES_ON_EDGES, std::max(pathpos[a], pathpos[b])));
183 }
184
185 void modify(int a, int b, const T & delta) {
186     for (int root; !is_ancestor(root = pathroot[path[a]], b); a = parent[root])
187         modify(path[a], 0, pathpos[a], delta);
188     for (int root; !is_ancestor(root = pathroot[path[b]], a); b = parent[root])
189         modify(path[b], 0, pathpos[b], delta);
190     if (VALUES_ON_EDGES && a == b) return;
191     modify(path[a], std::min(pathpos[a], pathpos[b]) + VALUES_ON_EDGES,
192           std::max(pathpos[a], pathpos[b]), delta);
193 }
194 };

```

```

195
196 /** Example Usage */
197
198 #include <iostream>
199 using namespace std;
200
201 const int MAXN = 1000;
202 vector<int> adj[MAXN];
203
204 /*
205     w=10     w=20     w=40
206     0-----1-----2-----3
207                     \
208                     -----4
209                     w=30
210 */
211 int main() {
212     adj[0].push_back(1);
213     adj[1].push_back(0);
214     adj[1].push_back(2);
215     adj[2].push_back(1);
216     adj[2].push_back(3);
217     adj[3].push_back(2);
218     adj[2].push_back(4);
219     adj[4].push_back(2);
220     heavy_light<int> hld(5, adj);
221     hld.modify(0, 1, 10);
222     hld.modify(1, 2, 20);
223     hld.modify(2, 3, 40);
224     hld.modify(2, 4, 30);
225     cout << hld.query(0, 3) << "\n"; //40
226     cout << hld.query(2, 4) << "\n"; //30
227     hld.modify(3, 4, 50); //w[every edge from 3 to 4] += 50
228     cout << hld.query(1, 4) << "\n"; //80
229     return 0;
230 }

```

3.6.2 Link-Cut Tree

```

1  /*
2
3  Description: Given an unweighted forest of trees where each node
4  has an associated value, a link/cut tree can be used to dynamically
5  query and modify values on the path between pairs of nodes a tree.
6  This problem can be solved using heavy-light decomposition, which
7  also supports having values stored on edges rather than the nodes.
8  However in a link/cut tree, nodes in different trees may be
9  dynamically linked, edges between nodes in the same tree may be
10 dynamically split, and connectivity between two nodes (whether they
11 are in the same tree) may be checked.
12
13 Time Complexity:  $O(\log N)$  amortized for make_root(), link(), cut(),
14 connected(), modify(), and query(), where N is the number of nodes
15 in the forest.
16
17 Space Complexity:  $O(N)$  on the number of nodes in the forest.
18

```

```

19  */
20
21  #include <algorithm> /* std::max(), std::swap() */
22  #include <climits>   /* INT_MIN */
23  #include <map>
24  #include <stdexcept> /* std::runtime_error() */
25
26  template<class T> class linkcut_forest {
27      //Modify the following 5 functions to implement your custom
28      //operations on the tree. This implements the Add/Max operations.
29      //Operations like Add/Sum, Set/Max can also be implemented.
30      static inline T modify_op(const T & x, const T & y) {
31          return x + y;
32      }
33
34      static inline T query_op(const T & x, const T & y) {
35          return std::max(x, y);
36      }
37
38      static inline T delta_on_segment(const T & delta, int seglen) {
39          if (delta == null_delta()) return null_delta();
40          //Here you must write a fast equivalent of following slow code:
41          // T result = delta;
42          // for (int i = 1; i < seglen; i++) result = query_op(result, delta);
43          // return result;
44          return delta;
45      }
46
47      static inline T null_delta() { return 0; }
48      static inline T null_value() { return INT_MIN; }
49
50      static inline T join_value_with_delta(const T & v, const T & delta) {
51          return delta == null_delta() ? v : modify_op(v, delta);
52      }
53
54      static T join_deltas(const T & delta1, const T & delta2) {
55          if (delta1 == null_delta()) return delta2;
56          if (delta2 == null_delta()) return delta1;
57          return modify_op(delta1, delta2);
58      }
59
60      struct node_t {
61          T value, subtree_value, delta;
62          int size;
63          bool rev;
64          node_t *L, *R, *parent;
65
66          node_t(const T & v) {
67              value = subtree_value = v;
68              delta = null_delta();
69              size = 1;
70              rev = false;
71              L = R = parent = 0;
72          }
73
74          bool is_root() { //is this the root of a splay tree?
75              return parent == 0 || (parent->L != this && parent->R != this);
76          }
77

```

```

78 void push() {
79     if (rev) {
80         rev = false;
81         std::swap(L, R);
82         if (L != 0) L->rev = !L->rev;
83         if (R != 0) R->rev = !R->rev;
84     }
85     value = join_value_with_delta(value, delta);
86     subtree_value = join_value_with_delta(subtree_value,
87         delta_on_segment(delta, size));
88     if (L != 0) L->delta = join_deltas(L->delta, delta);
89     if (R != 0) R->delta = join_deltas(R->delta, delta);
90     delta = null_delta();
91 }
92
93 void update() {
94     subtree_value = query_op(query_op(get_subtree_value(L),
95         join_value_with_delta(value, delta)),
96         get_subtree_value(R));
97     size = 1 + get_size(L) + get_size(R);
98 }
99 };
100
101 static inline int get_size(node_t * n) {
102     return n == 0 ? 0 : n->size;
103 }
104
105 static inline int get_subtree_value(node_t * n) {
106     return n == 0 ? null_value() : join_value_with_delta(n->subtree_value,
107         delta_on_segment(n->delta, n->size));
108 }
109
110 static void connect(node_t * ch, node_t * p, char is_left) {
111     if (ch != 0) ch->parent = p;
112     if (is_left < 0) return;
113     (is_left ? p->L : p->R) = ch;
114 }
115
116 /** rotates edge (n, n.parent)
117 *
118 *      g          g
119 *     /          /
120 *    p          n
121 *   /\  -->  /\
122 *  n  p.r   n.l p
123 * /\       /\
124 * n.l n.r   n.r p.r
125 */
126 static void rotate(node_t * n) {
127     node_t *p = n->parent, *g = p->parent;
128     bool is_rootp = p->is_root(), is_left = (n == p->L);
129     connect(is_left ? n->R : n->L, p, is_left);
130     connect(p, n, !is_left);
131     connect(n, g, is_rootp ? -1 : (p == g->L));
132     p->update();
133 }
134
135 /** brings n to the root, balancing tree
136 *
137 * zig-zig case:

```

```

137 *           g           n
138 *       / \       / \       / \
139 *     p  g.r rot(p) / \   rot(n) n.l p
140 *   / \      -->  n   g   -->  / \
141 *  n  p.r      / \   / \      n.r g
142 *   / \      n.l n.r p.r g.r   / \
143 *  n.l n.r                p.r g.r
144 *
145 * zig-zag case:
146 *           g           n
147 *       / \       / \       / \
148 *     p  g.r rot(n) n  g.r rot(n) / \
149 *   / \      -->  / \      -->  p   g
150 * p.l n      p  n.r      / \   / \
151 *   / \      / \      p.l n.l n.r g.r
152 *  n.l n.r  p.l n.l
153 */
154 static void splay(node_t * n) {
155     while (!n->is_root()) {
156         node_t *p = n->parent, *g = p->parent;
157         if (!p->is_root()) g->push();
158         p->push();
159         n->push();
160         if (!p->is_root())
161             rotate((n == p->L) == (p == g->L) ? p/*zig-zig*/ : n/*zig-zag*/);
162         rotate(n);
163     }
164     n->push();
165     n->update();
166 }
167
168 //makes node n the root of the virtual tree,
169 //and also n becomes the leftmost node in its splay tree
170 static node_t * expose(node_t * n) {
171     node_t *prev = 0;
172     for (node_t *i = n; i != 0; i = i->parent) {
173         splay(i);
174         i->L = prev;
175         prev = i;
176     }
177     splay(n);
178     return prev;
179 }
180
181 std::map<int, node_t*> nodes; //use array if ID compression not required
182 node_t *u, *v; //temporary
183
184 void get_uv(int a, int b) {
185     static typename std::map<int, node_t*>::iterator it1, it2;
186     it1 = nodes.find(a);
187     it2 = nodes.find(b);
188     if (it1 == nodes.end() || it2 == nodes.end())
189         throw std::runtime_error("Error: a or b does not exist in forest.");
190     u = it1->second;
191     v = it2->second;
192 }
193
194 public:
195 ~linkcut_forest() {

```

```

196     static typename std::map<int, node_t*>::iterator it;
197     for (it = nodes.begin(); it != nodes.end(); ++it)
198         delete it->second;
199 }
200
201 void make_root(int id, const T & initv) {
202     if (nodes.find(id) != nodes.end())
203         throw std::runtime_error("Cannot make root(): ID already exists.");
204     node_t * n = new node_t(initv);
205     expose(n);
206     n->rev = !n->rev;
207     nodes[id] = n;
208 }
209
210 bool connected(int a, int b) {
211     get_uv(a, b);
212     if (a == b) return true;
213     expose(u);
214     expose(v);
215     return u->parent != 0;
216 }
217
218 void link(int a, int b) {
219     if (connected(a, b))
220         throw std::runtime_error("Error: a and b are already connected.");
221     get_uv(a, b);
222     expose(u);
223     u->rev = !u->rev;
224     u->parent = v;
225 }
226
227 void cut(int a, int b) {
228     get_uv(a, b);
229     expose(u);
230     u->rev = !u->rev;
231     expose(v);
232     if (v->R != u || u->L != 0)
233         throw std::runtime_error("Error: edge (a, b) does not exist.");
234     v->R->parent = 0;
235     v->R = 0;
236 }
237
238 T query(int a, int b) {
239     if (!connected(a, b))
240         throw std::runtime_error("Error: a and b are not connected.");
241     get_uv(a, b);
242     expose(u);
243     u->rev = !u->rev;
244     expose(v);
245     return get_subtree_value(v);
246 }
247
248 void modify(int a, int b, const T & delta) {
249     if (!connected(a, b))
250         throw std::runtime_error("Error: a and b are not connected.");
251     get_uv(a, b);
252     expose(u);
253     u->rev = !u->rev;
254     expose(v);

```



```

255     v->delta = join_deltas(v->delta, delta);
256 }
257 };
258
259 /** Example Usage */
260
261 #include <iostream>
262 using namespace std;
263
264 int main() {
265     linkcut_forest<int> F;
266     /*
267     v=10      v=40      v=20      v=10
268     0-----1-----2-----3
269                     \
270                     -----4
271                             v=30
272     */
273     F.make_root(0, 10);
274     F.make_root(1, 40);
275     F.make_root(2, 20);
276     F.make_root(3, 10);
277     F.make_root(4, 30);
278     F.link(0, 1);
279     F.link(1, 2);
280     F.link(2, 3);
281     F.link(2, 4);
282     cout << F.query(1, 4) << "\n"; //40
283     F.modify(1, 1, -10);
284     F.modify(3, 4, -10);
285     /*
286     v=10      v=30      v=10      v=0
287     0-----1-----2-----3
288                     \
289                     -----4
290                             v=20
291     */
292     cout << F.query(0, 4) << "\n"; //30
293     cout << F.query(3, 4) << "\n"; //20
294     F.cut(1, 2);
295     cout << F.connected(1, 2) << "\n"; //0
296     cout << F.connected(0, 4) << "\n"; //0
297     cout << F.connected(2, 3) << "\n"; //1
298     return 0;
299 }

```

3.7 Lowest Common Ancestor

3.7.1 Sparse Tables

```

1  /*
2
3  Description: Given an undirected graph that is a tree, the
4  lowest common ancestor (LCA) of two nodes v and w is the
5  lowest (i.e. deepest) node that has both v and w as descendants,
6  where we define each node to be a descendant of itself (so if

```

```

7  v has a direct connection from w, w is the lowest common
8  ancestor). The following program uses sparse tables to solve
9  the problem on an unchanging graph.
10
11  Time Complexity:  $O(N \log N)$  for build() and  $O(\log N)$  for lca(),
12  where N is the number of nodes in the tree.
13
14  Space Complexity:  $O(N \log N)$ .
15
16  */
17
18  #include <vector>
19
20  const int MAXN = 1000;
21  int len, timer, tin[MAXN], tout[MAXN];
22  std::vector<int> adj[MAXN], dp[MAXN];
23
24  void dfs(int u, int p) {
25      tin[u] = timer++;
26      dp[u][0] = p;
27      for (int i = 1; i < len; i++)
28          dp[u][i] = dp[dp[u][i - 1]][i - 1];
29      for (int j = 0, v; j < (int)adj[u].size(); j++)
30          if ((v = adj[u][j]) != p)
31              dfs(v, u);
32      tout[u] = timer++;
33  }
34
35  void build(int nodes, int root) {
36      len = 1;
37      while ((1 << len) <= nodes) len++;
38      for (int i = 0; i < nodes; i++)
39          dp[i].resize(len);
40      timer = 0;
41      dfs(root, root);
42  }
43
44  inline bool is_parent(int parent, int child) {
45      return tin[parent] <= tin[child] && tout[child] <= tout[parent];
46  }
47
48  int lca(int a, int b) {
49      if (is_parent(a, b)) return a;
50      if (is_parent(b, a)) return b;
51      for (int i = len - 1; i >= 0; i--)
52          if (!is_parent(dp[a][i], b))
53              a = dp[a][i];
54      return dp[a][0];
55  }
56
57  /** Example Usage */
58
59  #include <iostream>
60  using namespace std;
61
62  int main() {
63      adj[0].push_back(1);
64      adj[1].push_back(0);
65      adj[1].push_back(2);

```

```

66  adj[2].push_back(1);
67  adj[3].push_back(1);
68  adj[1].push_back(3);
69  adj[0].push_back(4);
70  adj[4].push_back(0);
71  build(5, 0);
72  cout << lca(3, 2) << "\n"; //1
73  cout << lca(2, 4) << "\n"; //0
74  return 0;
75 }

```

3.7.2 Segment Trees

```

1  /*
2
3  Description: Given a rooted tree, the lowest common ancestor (LCA)
4  of two nodes v and w is the lowest (i.e. deepest) node that has
5  both v and w as descendants, where we define each node to be a
6  descendant of itself (so if v has a direct connection from w, w
7  is the lowest common ancestor). This problem can be reduced to the
8  range minimum query problem using Eulerian tours.
9
10 Time Complexity: O(N log N) for build() and O(log N) for lca(),
11 where N is the number of nodes in the tree.
12
13 Space Complexity: O(N log N).
14
15 */
16
17 #include <algorithm> /* std::fill(), std::min(), std::max() */
18 #include <vector>
19
20 const int MAXN = 1000;
21 int len, counter;
22 int depth[MAXN], dfs_order[2*MAXN], first[MAXN], minpos[8*MAXN];
23 std::vector<int> adj[MAXN];
24
25 void dfs(int u, int d) {
26     depth[u] = d;
27     dfs_order[counter++] = u;
28     for (int j = 0, v; j < (int)adj[u].size(); j++) {
29         if (depth[v = adj[u][j]] == -1) {
30             dfs(v, d + 1);
31             dfs_order[counter++] = u;
32         }
33     }
34 }
35
36 void build_tree(int n, int l, int h) {
37     if (l == h) {
38         minpos[n] = dfs_order[l];
39         return;
40     }
41     int lchild = 2 * n + 1, rchild = 2 * n + 2;
42     build_tree(lchild, l, (l + h)/2);
43     build_tree(rchild, (l + h) / 2 + 1, h);
44     minpos[n] = depth[minpos[lchild]] < depth[minpos[rchild]] ?

```

```

45         minpos[lchild] : minpos[rchild];
46     }
47
48     void build(int nodes, int root) {
49         std::fill(depth, depth + nodes, -1);
50         std::fill(first, first + nodes, -1);
51         len = 2*nodes - 1;
52         counter = 0;
53         dfs(root, 0);
54         build_tree(0, 0, len - 1);
55         for (int i = 0; i < len; i++)
56             if (first[dfs_order[i]] == -1)
57                 first[dfs_order[i]] = i;
58     }
59
60     int get_minpos(int a, int b, int n, int l, int h) {
61         if (a == l && b == h) return minpos[n];
62         int mid = (l + h) >> 1;
63         if (a <= mid && b > mid) {
64             int p1 = get_minpos(a, std::min(b, mid), 2 * n + 1, l, mid);
65             int p2 = get_minpos(std::max(a, mid + 1), b, 2 * n + 2, mid + 1, h);
66             return depth[p1] < depth[p2] ? p1 : p2;
67         }
68         if (a <= mid) return get_minpos(a, std::min(b, mid), 2 * n + 1, l, mid);
69         return get_minpos(std::max(a, mid + 1), b, 2 * n + 2, mid + 1, h);
70     }
71
72     int lca(int a, int b) {
73         return get_minpos(std::min(first[a], first[b]),
74                         std::max(first[a], first[b]), 0, 0, len - 1);
75     }
76
77     /** Example Usage */
78
79     #include <iostream>
80     using namespace std;
81
82     int main() {
83         adj[0].push_back(1);
84         adj[1].push_back(0);
85         adj[1].push_back(2);
86         adj[2].push_back(1);
87         adj[3].push_back(1);
88         adj[1].push_back(3);
89         adj[0].push_back(4);
90         adj[4].push_back(0);
91         build(5, 0);
92         cout << lca(3, 2) << "\n"; //1
93         cout << lca(2, 4) << "\n"; //0
94         return 0;
95     }

```

Chapter 4

Mathematics

4.1 Mathematics Toolbox

```
1  /*
2
3  Useful math definitions. Excludes geometry (see next chapter).
4
5  */
6
7  #include <algorithm> /* std::reverse() */
8  #include <cfloat>    /* DBL_MAX */
9  #include <cmath>    /* a lot of things */
10 #include <string>
11 #include <vector>
12
13 /* Definitions for Common Floating Point Constants */
14
15 const double PI = acos(-1.0), E = exp(1.0), root2 = sqrt(2.0);
16 const double phi = (1.0 + sqrt(5.0)) / 2.0; //golden ratio
17
18 //Sketchy but working definitions of +infinity, -infinity and quiet NaN
19 //A better way is using functions of std::numeric_limits<T> from <limits>
20 //See main() for identities involving the following special values.
21 const double posinf = 1.0 / 0.0, neginf = -1.0 / 0.0, NaN = -(0.0 / 0.0);
22
23 /*
24
25 Epsilon Comparisons
26
27 The range of values for which X compares EQ() to is [X - eps, X + eps].
28 For values to compare LT() and GT() x, they must fall outside of the range.
29
30 e.g. if eps = 1e-7, then EQ(1e-8, 2e-8) is true and LT(1e-8, 2e-8) is false.
31
32 */
33
34 const double eps = 1e-7;
35 #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
36 #define NE(a, b) (fabs((a) - (b)) > eps)  /* not equal to */
37 #define LT(a, b) ((a) < (b) - eps)        /* less than */
```

```

38 #define GT(a, b) ((a) > (b) + eps)          /* greater than */
39 #define LE(a, b) ((a) <= (b) + eps)         /* less than or equal to */
40 #define GE(a, b) ((a) >= (b) - eps)         /* greater than or equal to */
41
42 /*
43
44 Sign Function:
45
46 Returns: -1 (if x < 0), 0 (if x == 0), or 1 (if x > 0)
47 Doesn't handle the sign of NaN like signbit() or copysign()
48
49 */
50
51 template<class T> int sgn(const T & x) {
52     return (T(0) < x) - (x < T(0));
53 }
54
55 /*
56
57 signbit() and copysign() functions, only in C++11 and later.
58
59 signbit() returns whether the sign bit of the floating point
60 number is set to true. If signbit(x), then x is "negative."
61 Note that signbit(0.0) == 0 but signbit(-0.0) == 1. This
62 also works as expected on NaN, -NaN, posinf, and neginf.
63
64 We implement this by casting the floating point value to an
65 integer type with the same number of bits so we can perform
66 shift operations on it, then we extract the sign bit.
67 Another way is using unions, but this is non-portable
68 depending on endianness of the platform. Unfortunately, we
69 cannot find the signbit of long doubles using the method
70 below because there is no corresponding 96-bit integer type.
71 Note that this will cause complaints with the compiler.
72
73 copysign(x, y) returns a number with the magnitude of x but
74 the sign of y.
75
76 Assumptions: sizeof(float) == sizeof(int) and
77               sizeof(long long) == sizeof(double)
78               CHAR_BITS == 8 (8 bits to a byte)
79
80 */
81
82 inline bool signbit(float x) {
83     return (*(int*)&x) >> (sizeof(float) * 8 - 1);
84 }
85
86 inline bool signbit(double x) {
87     return *(long long*)&x >> (sizeof(double) * 8 - 1);
88 }
89
90 template<class Double>
91 inline Double copysign(Double x, Double y) {
92     return signbit(y) ? -fabs(x) : fabs(x);
93 }
94
95 /*
96

```

```

97 Floating Point Rounding Functions
98
99 floor() in <cmath> asymmetrically rounds down, towards -infinity,
100 while ceil() in <cmath> asymmetrically rounds up, towards +infinity.
101 The following are common alternative ways to round.
102
103 */
104
105 //symmetric round down, bias: towards zero (same as trunc() in C++11)
106 template<class Double> Double floor0(const Double & x) {
107     Double res = floor(fabs(x));
108     return (x < 0.0) ? -res : res;
109 }
110
111 //symmetric round up, bias: away from zero
112 template<class Double> Double ceil0(const Double & x) {
113     Double res = ceil(fabs(x));
114     return (x < 0.0) ? -res : res;
115 }
116
117 //round half up, bias: towards +infinity
118 template<class Double> Double roundhalfup(const Double & x) {
119     return floor(x + 0.5);
120 }
121
122 //round half up, bias: towards -infinity
123 template<class Double> Double roundhalfdown(const Double & x) {
124     return ceil(x - 0.5);
125 }
126
127 //symmetric round half down, bias: towards zero
128 template<class Double> Double roundhalfdown0(const Double & x) {
129     Double res = roundhalfdown(fabs(x));
130     return (x < 0.0) ? -res : res;
131 }
132
133 //symmetric round half up, bias: away from zero
134 template<class Double> Double roundhalfup0(const Double & x) {
135     Double res = roundhalfup(fabs(x));
136     return (x < 0.0) ? -res : res;
137 }
138
139 //round half to even (banker's rounding), bias: none
140 template<class Double>
141 Double roundhalfeven(const Double & x, const Double & eps = 1e-7) {
142     if (x < 0.0) return -roundhalfeven(-x, eps);
143     Double ipart;
144     modf(x, &ipart);
145     if (x - (ipart + 0.5) < eps)
146         return (fmod(ipart, 2.0) < eps) ? ipart : ceil0(ipart + 0.5);
147     return roundhalfup0(x);
148 }
149
150 //round alternating up/down for ties, bias: none for sequential calls
151 template<class Double> Double roundalternate(const Double & x) {
152     static bool up = true;
153     return (up = !up) ? roundhalfup(x) : roundhalfdown(x);
154 }
155

```

```

156 //symmetric round alternate, bias: none for sequential calls
157 template<class Double> Double roundalternate0(const Double & x) {
158     static bool up = true;
159     return (up = !up) ? roundhalfup0(x) : roundhalfdown0(x);
160 }
161
162 //round randomly for tie-breaking, bias: generator's bias
163 template<class Double> Double roundrandom(const Double & x) {
164     return (rand() % 2 == 0) ? roundhalfup0(x) : roundhalfdown0(x);
165 }
166
167 //round x to N digits after the decimal using the specified round function
168 //e.g. roundplaces(-1.23456, 3, roundhalfdown0<double>) returns -1.235
169 template<class Double, class RoundFunction>
170 double roundplaces(const Double & x, unsigned int N, RoundFunction f) {
171     return f(x * pow(10, N)) / pow(10, N);
172 }
173
174 /*
175
176 Error Function (erf()) and erfc() in C++11
177
178 erf(x) = 2/sqrt(pi) * integral of exp(-t^2) dt from 0 to x
179 erfc(x) = 1 - erf(x)
180 Note that the functions are co-dependent.
181
182 Adapted from: http://www.digitalmars.com/archives/cplusplus/3634.html#N3655
183
184 */
185
186 //calculate 12 significant figs (don't ask for more than 1e-15)
187 static const double rel_error = 1e-12;
188
189 double erf(double x) {
190     if (signbit(x)) return -erf(-x);
191     if (fabs(x) > 2.2) return 1.0 - erfc(x);
192     double sum = x, term = x, xsqr = x * x;
193     int j = 1;
194     do {
195         term *= xsqr / j;
196         sum -= term / (2 * (j++) + 1);
197         term *= xsqr / j;
198         sum += term / (2 * (j++) + 1);
199     } while (fabs(term) / sum > rel_error);
200     return 1.128379167095512574 * sum; //1.128 ~ 2/sqrt(pi)
201 }
202
203 double erfc(double x) {
204     if (fabs(x) < 2.2) return 1.0 - erf(x);
205     if (signbit(x)) return 2.0 - erfc(-x);
206     double a = 1, b = x, c = x, d = x * x + 0.5, q1, q2 = 0, n = 1.0, t;
207     do {
208         t = a * n + b * x; a = b; b = t;
209         t = c * n + d * x; c = d; d = t;
210         n += 0.5;
211         q1 = q2;
212         q2 = b / d;
213     } while (fabs(q1 - q2) / q2 > rel_error);
214     return 0.564189583547756287 * exp(-x * x) * q2; //0.564 ~ 1/sqrt(pi)

```



```

215 }
216
217 /*
218
219 Gamma and Log-Gamma Functions (tgamma() and lgamma() in C++11)
220 Warning: unlike the actual standard C++ versions, the following
221 function only works on positive numbers (returns NaN if x <= 0).
222 Adapted from: http://www.johndcook.com/blog/cpp\_gamma/
223
224 */
225
226 double lgamma(double x);
227
228 double tgamma(double x) {
229     if (x <= 0.0) return NaN;
230     static const double gamma = 0.577215664901532860606512090;
231     if (x < 1e-3) return 1.0 / (x * (1.0 + gamma * x));
232     if (x < 12.0) {
233         double y = x;
234         int n = 0;
235         bool arg_was_less_than_one = (y < 1.0);
236         if (arg_was_less_than_one) y += 1.0;
237         else y -= (n = static_cast<int>(floor(y)) - 1);
238         static const double p[] = {
239             -1.71618513886549492533811E+0, 2.47656508055759199108314E+1,
240             -3.79804256470945635097577E+2, 6.29331155312818442661052E+2,
241             8.66966202790413211295064E+2, -3.14512729688483675254357E+4,
242             -3.61444134186911729807069E+4, 6.64561438202405440627855E+4
243         };
244         static const double q[] = {
245             -3.08402300119738975254353E+1, 3.15350626979604161529144E+2,
246             -1.01515636749021914166146E+3, -3.10777167157231109440444E+3,
247             2.25381184209801510330112E+4, 4.75584627752788110767815E+3,
248             -1.34659959864969306392456E+5, -1.15132259675553483497211E+5
249         };
250         double num = 0.0, den = 1.0, z = y - 1;
251         for (int i = 0; i < 8; i++) {
252             num = (num + p[i]) * z;
253             den = den * z + q[i];
254         }
255         double result = num / den + 1.0;
256         if (arg_was_less_than_one) result /= (y - 1.0);
257         else for (int i = 0; i < n; i++) result *= y++;
258         return result;
259     }
260     return (x > 171.624) ? DBL_MAX * 2.0 : exp(lgamma(x));
261 }
262
263 double lgamma(double x) {
264     if (x <= 0.0) return NaN;
265     if (x < 12.0) return log(fabs(tgamma(x)));
266     static const double c[8] = {
267         1.0/12.0, -1.0/360.0, 1.0/1260.0, -1.0/1680.0, 1.0/1188.0,
268         -691.0/360360.0, 1.0/156.0, -3617.0/122400.0
269     };
270     double z = 1.0 / (x * x), sum = c[7];
271     for (int i = 6; i >= 0; i--) sum = sum * z + c[i];
272     static const double halflog2pi = 0.91893853320467274178032973640562;
273     return (x - 0.5) * log(x) - x + halflog2pi + sum / x;

```

```

274 }
275
276 /*
277
278 Base Conversion - O(N) on the number of digits
279
280 Given the digits of an integer x in base a, returns x's digits in base b.
281 Precondition: the base-10 value of x must be able to fit within an unsigned
282 long long. In other words, the value of x must be between 0 and 2^64 - 1.
283
284 Note: vector[0] stores the most significant digit in all usages below.
285
286 e.g. if x = {1, 2, 3} and a = 5 (i.e. x = 123 in base 5 = 38 in base 10),
287 then convert_base(x, 5, 3) returns {1, 1, 0, 2} (1102 in base 2).
288
289 */
290
291 std::vector<int> convert_base(const std::vector<int> & x, int a, int b) {
292     unsigned long long base10 = 0;
293     for (int i = 0; i < (int)x.size(); i++)
294         base10 += x[i] * pow(a, x.size() - i - 1);
295     int N = ceil(log(base10 + 1) / log(b));
296     std::vector<int> baseb;
297     for (int i = 1; i <= N; i++)
298         baseb.push_back(int(base10 / pow(b, N - i)) % b);
299     return baseb;
300 }
301
302 //returns digits of a number in base b
303 std::vector<int> base_digits(int x, int b = 10) {
304     std::vector<int> baseb;
305     while (x != 0) {
306         baseb.push_back(x % b);
307         x /= b;
308     }
309     std::reverse(baseb.begin(), baseb.end());
310     return baseb;
311 }
312
313 /*
314
315 Integer to Roman Numerals Conversion
316
317 Given an integer x, this function returns the Roman numeral representation
318 of x as a C++ string. More 'M's are appended to the front of the resulting
319 string if x is greater than 1000. e.g. to_roman(1234) returns "MCCXXXIV"
320 and to_roman(5678) returns "MMMMDCLXXVIII".
321
322 */
323
324 std::string to_roman(unsigned int x) {
325     static std::string h[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};
326     static std::string t[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
327     static std::string o[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};
328     std::string res(x / 1000, 'M');
329     x %= 1000;
330     return res + h[x / 100] + t[x / 10 % 10] + o[x % 10];
331 }
332

```

```

333  /** Example Usage **/
334
335  #include <algorithm>
336  #include <cassert>
337  #include <iostream>
338  using namespace std;
339
340  int main() {
341      cout << "PI:␣" << PI << "\n";
342      cout << "E:␣" << E << "\n";
343      cout << "sqrt(2):␣" << root2 << "\n";
344      cout << "Golden␣ratio:␣" << phi << "\n";
345
346      //some properties of posinf, neginf, and NaN:
347      double x = -1234.567890; //any normal value of x will work
348      assert((posinf > x) && (neginf < x) && (posinf == -neginf));
349      assert((posinf + x == posinf) && (posinf - x == posinf));
350      assert((neginf + x == neginf) && (neginf - x == neginf));
351      assert((posinf + posinf == posinf) && (neginf - posinf == neginf));
352      assert((NaN != x) && (NaN != NaN) && (NaN != posinf) && (NaN != neginf));
353      assert(!(NaN < x) && !(NaN > x) && !(NaN <= x) && !(NaN >= x));
354      assert(isnan(0.0*posinf) && isnan(0.0*neginf) && isnan(posinf/neginf));
355      assert(isnan(NaN) && isnan(-NaN) && isnan(NaN*x + x - x/-NaN));
356      assert(isnan(neginf-neginf) && isnan(posinf-posinf) && isnan(posinf+neginf));
357      assert(!signbit(NaN) && signbit(-NaN) && !signbit(posinf) && signbit(neginf));
358
359      assert(copysign(1.0, +2.0) == +1.0 && copysign(posinf, -2.0) == neginf);
360      assert(copysign(1.0, -2.0) == -1.0 && signbit(copysign(NaN, -2.0)));
361      assert(sgn(-1.234) == -1 && sgn(0.0) == 0 && sgn(5678) == 1);
362
363      assert(EQ(floor0(1.5), 1.0) && EQ(floor0(-1.5), -1.0));
364      assert(EQ(ceil0(1.5), 2.0) && EQ(ceil0(-1.5), -2.0));
365      assert(EQ(roundhalfup(1.5), 2.0) && EQ(roundhalfup(-1.5), -1.0));
366      assert(EQ(roundhalfdown(1.5), 1.0) && EQ(roundhalfdown(-1.5), -2.0));
367      assert(EQ(roundhalfup0(1.5), 2.0) && EQ(roundhalfup0(-1.5), -2.0));
368      assert(EQ(roundhalfdown0(1.5), 1.0) && EQ(roundhalfdown0(-1.5), -1.0));
369      assert(EQ(roundhalfeven(1.5), 2.0) && EQ(roundhalfeven(-1.5), -2.0));
370      assert(NE(roundalternate(1.5), roundalternate(1.5)));
371      assert(EQ(roundplaces(-1.23456, 3, roundhalfdown0<double>), -1.235));
372
373      assert(EQ(erf(1.0), 0.8427007929) && EQ(erf(-1.0), -0.8427007929));
374      assert(EQ(tgamma(0.5), 1.7724538509) && EQ(tgamma(1.0), 1.0));
375      assert(EQ(lgamma(0.5), 0.5723649429) && EQ(lgamma(1.0), 0.0));
376
377      int base10digs[] = {1, 2, 3, 4, 5, 6}, a = 20, b = 10;
378      vector<int> basea = base_digits(123456, a);
379      vector<int> baseb = convert_base(basea, a, b);
380      assert(equal(baseb.begin(), baseb.end(), base10digs));
381
382      assert(to_roman(1234) == "MCCXXXIV");
383      assert(to_roman(5678) == "MMMMDCLXXVIII");
384      return 0;
385  }

```

4.2 Combinatorics

4.2.1 Combinatorial Calculations

```

1  /*
2
3  The meanings of the following functions can respectively be
4  found with quick searches online. All of them computes the
5  answer modulo m, since contest problems typically ask us for
6  this due to the actual answer being potentially very large.
7  All functions using tables to generate every answer below
8  n and k can be optimized using recursion and memoization.
9
10 Note: The following are only defined for nonnegative inputs.
11
12 */
13
14 #include <vector>
15
16 typedef std::vector<std::vector<long long> > table;
17
18 //n! mod m in O(n)
19 long long factorial(int n, int m = 1000000007) {
20     long long res = 1;
21     for (int i = 2; i <= n; i++) res = (res * i) % m;
22     return res % m;
23 }
24
25 //n! mod p, where p is a prime number, in O(p log n)
26 long long factorialp(long long n, long long p = 1000000007) {
27     long long res = 1, h;
28     while (n > 1) {
29         res = (res * ((n / p) % 2 == 1 ? p - 1 : 1)) % p;
30         h = n % p;
31         for (int i = 2; i <= h; i++) res = (res * i) % p;
32         n /= p;
33     }
34     return res % p;
35 }
36
37 //first n rows of pascal's triangle (mod m) in O(n^2)
38 table binomial_table(int n, long long m = 1000000007) {
39     table t(n + 1);
40     for (int i = 0; i <= n; i++)
41         for (int j = 0; j <= i; j++)
42             if (i < 2 || j == 0 || i == j)
43                 t[i].push_back(1);
44             else
45                 t[i].push_back((t[i - 1][j - 1] + t[i - 1][j]) % m);
46     return t;
47 }
48
49 //if the product of two 64-bit ints (a*a, a*b, or b*b) can
50 //overflow, you must use mulmod (multiplication by adding)
51 long long powmod(long long a, long long b, long long m) {
52     long long x = 1, y = a;
53     for (; b > 0; b >>= 1) {

```

```

54     if (b & 1) x = (x * y) % m;
55     y = (y * y) % m;
56 }
57 return x % m;
58 }
59
60 //n choose k (mod a prime number p) in O(min(k, n - k))
61 //powmod is used to find the mod inverse to get num / den % m
62 long long choose(int n, int k, long long p = 1000000007) {
63     if (n < k) return 0;
64     if (k > n - k) k = n - k;
65     long long num = 1, den = 1;
66     for (int i = 0; i < k; i++)
67         num = (num * (n - i)) % p;
68     for (int i = 1; i <= k; i++)
69         den = (den * i) % p;
70     return num * powmod(den, p - 2, p) % p;
71 }
72
73 //n multichoose k (mod a prime number p) in O(k)
74 long long multichoose(int n, int k, long long p = 1000000007) {
75     return choose(n + k - 1, k, p);
76 }
77
78 //n permute k (mod m) on O(k)
79 long long permute(int n, int k, long long m = 1000000007) {
80     if (n < k) return 0;
81     long long res = 1;
82     for (int i = 0; i < k; i++)
83         res = (res * (n - i)) % m;
84     return res % m;
85 }
86
87 //number of partitions of n (mod m) in O(n^2)
88 long long partitions(int n, long long m = 1000000007) {
89     std::vector<long long> p(n + 1, 0);
90     p[0] = 1;
91     for (int i = 1; i <= n; i++)
92         for (int j = i; j <= n; j++)
93             p[j] = (p[j] + p[j - i]) % m;
94     return p[n] % m;
95 }
96
97 //partitions of n into exactly k parts (mod m) in O(n * k)
98 long long partitions(int n, int k, long long m = 1000000007) {
99     table t(n + 1, std::vector<long long>(k + 1, 0));
100     t[0][1] = 1;
101     for (int i = 1; i <= n; i++)
102         for (int j = 1, h = k < i ? k : i; j <= h; j++)
103             t[i][j] = (t[i - 1][j - 1] + t[i - j][j]) % m;
104     return t[n][k] % m;
105 }
106
107 //unsigned Stirling numbers of the 1st kind (mod m) in O(n * k)
108 long long stirling1(int n, int k, long long m = 1000000007) {
109     table t(n + 1, std::vector<long long>(k + 1, 0));
110     t[0][0] = 1;
111     for (int i = 1; i <= n; i++)
112         for (int j = 1; j <= k; j++) {

```

```

113         t[i][j] = ((i - 1) * t[i - 1][j]) % m;
114         t[i][j] = (t[i][j] + t[i - 1][j - 1]) % m;
115     }
116     return t[n][k] % m;
117 }
118
119 //Stirling numbers of the 2nd kind (mod m) in O(n * k)
120 long long stirling2(int n, int k, long long m = 1000000007) {
121     table t(n + 1, std::vector<long long>(k + 1, 0));
122     t[0][0] = 1;
123     for (int i = 1; i <= n; i++)
124         for (int j = 1; j <= k; j++) {
125             t[i][j] = (j * t[i - 1][j]) % m;
126             t[i][j] = (t[i][j] + t[i - 1][j - 1]) % m;
127         }
128     return t[n][k] % m;
129 }
130
131 //Eulerian numbers of the 1st kind (mod m) in O(n * k)
132 //precondition: n > k
133 long long eulerian1(int n, int k, long long m = 1000000007) {
134     if (k > n - 1 - k) k = n - 1 - k;
135     table t(n + 1, std::vector<long long>(k + 1, 1));
136     for (int j = 1; j <= k; j++) t[0][j] = 0;
137     for (int i = 1; i <= n; i++)
138         for (int j = 1; j <= k; j++) {
139             t[i][j] = ((i - j) * t[i - 1][j - 1]) % m;
140             t[i][j] = (t[i][j] + ((j + 1) * t[i - 1][j]) % m) % m;
141         }
142     return t[n][k] % m;
143 }
144
145 //Eulerian numbers of the 2nd kind (mod m) in O(n * k)
146 //precondition: n > k
147 long long eulerian2(int n, int k, long long m = 1000000007) {
148     table t(n + 1, std::vector<long long>(k + 1, 1));
149     for (int i = 1; i <= n; i++)
150         for (int j = 1; j <= k; j++) {
151             if (i == j) {
152                 t[i][j] = 0;
153             } else {
154                 t[i][j] = ((j + 1) * t[i - 1][j]) % m;
155                 t[i][j] = (((2 * i - 1 - j) * t[i - 1][j - 1]) % m
156                     + t[i][j]) % m;
157             }
158         }
159     return t[n][k] % m;
160 }
161
162 //nth Catalan number (mod a prime number p) in O(n)
163 long long catalan(int n, long long p = 1000000007) {
164     return choose(2 * n, n, p) * powmod(n + 1, p - 2, p) % p;
165 }
166
167 /** Example Usage */
168
169 #include <cassert>
170 #include <iostream>
171 using namespace std;

```

```

172
173 int main() {
174     table t = binomial_table(10);
175     for (int i = 0; i < (int)t.size(); i++) {
176         for (int j = 0; j < (int)t[i].size(); j++)
177             cout << t[i][j] << " ";
178         cout << "\n";
179     }
180     assert(factorial(10) == 3628800);
181     assert(factorialp(123456) == 639390503);
182     assert(choose(20, 7) == 77520);
183     assert(multichoose(20, 7) == 657800);
184     assert(permute(10, 4) == 5040);
185     assert(partitions(4) == 5);
186     assert(partitions(100, 5) == 38225);
187     assert(stirling1(4, 2) == 11);
188     assert(stirling2(4, 3) == 6);
189     assert(eulerian1(9, 5) == 88234);
190     assert(eulerian2(8, 3) == 195800);
191     assert(catalan(10) == 16796);
192     return 0;
193 }

```

4.2.2 Enumerating Arrangements

```

1  /*
2
3  We shall consider an arrangement to be a permutation of
4  all the integers from 0 to n - 1. For our purposes, the
5  difference between an arrangement and a permutation is
6  simply that a permutation can pertain to a set of any
7  given values, not just distinct integers from 0 to n-1.
8
9  */
10
11 #include <algorithm> /* std::copy(), std::fill() */
12 #include <vector>
13
14 /*
15
16 Changes a[] to the next lexicographically greater
17 permutation of any k distinct integers in range [0, n).
18 The values of a[] that's passed should be k distinct
19 integers, each in range [0, n).
20
21 returns: whether the function could rearrange a[] to
22 a lexicographically greater arrangement.
23
24 examples:
25 next_arrangement(4, 3, {0, 1, 2}) => 1,  a[] = {0, 1, 3}
26 next_arrangement(4, 3, {0, 1, 3}) => 1,  a[] = {0, 2, 1}
27 next_arrangement(4, 3, {3, 2, 1}) => 0,  a[] unchanged
28
29 */
30
31 bool next_arrangement(int n, int k, int a[]) {
32     std::vector<bool> used(n);

```

```

33     for (int i = 0; i < k; i++) used[a[i]] = true;
34     for (int i = k - 1; i >= 0; i--) {
35         used[a[i]] = false;
36         for (int j = a[i] + 1; j < n; j++) {
37             if (!used[j]) {
38                 a[i++] = j;
39                 used[j] = true;
40                 for (int x = 0; x < k; x++)
41                     if (!used[x]) a[i++] = x;
42                 return true;
43             }
44         }
45     }
46     return false;
47 }
48
49 /*
50 Computes n permute k using formula:  $nPk = n!/(n - k)!$ 
51 Complexity:  $O(k)$ . E.g. n_permute_k(10, 7) = 604800
52
53 */
54
55 long long n_permute_k(int n, int k) {
56     long long res = 1;
57     for (int i = 0; i < k; i++) res *= n - i;
58     return res;
59 }
60
61 /*
62 Given an integer rank x in range [0, n permute k), returns
63 a vector of integers representing the x-th lexicographically
64 smallest permutation of any k distinct integers in [0, n).
65
66 examples: arrangement_by_rank(4, 3, 0) => {0, 1, 2}
67           arrangement_by_rank(4, 3, 5) => {0, 3, 2}
68
69 */
70
71 std::vector<int> arrangement_by_rank(int n, int k, long long x) {
72     std::vector<int> free(n), res(k);
73     for (int i = 0; i < n; i++) free[i] = i;
74     for (int i = 0; i < k; i++) {
75         long long cnt = n_permute_k(n - 1 - i, k - 1 - i);
76         int pos = (int)(x / cnt);
77         res[i] = free[pos];
78         std::copy(free.begin() + pos + 1, free.end(),
79                 free.begin() + pos);
80         x %= cnt;
81     }
82     return res;
83 }
84
85 /*
86 Given an array a[] of k integers each in range [0, n), returns
87 the (0-based) lexicographical rank (counting from least to
88 greatest) of the arrangement specified by a[] in all possible

```



```

92 permutations of k distinct integers in range [0, n).
93
94 examples: rank_by_arrangement(4, 3, {0, 1, 2}) => 0
95           rank_by_arrangement(4, 3, {0, 3, 2}) => 5
96
97 */
98
99 long long rank_by_arrangement(int n, int k, int a[]) {
100     long long res = 0;
101     std::vector<bool> used(n);
102     for (int i = 0; i < k; i++) {
103         int cnt = 0;
104         for (int j = 0; j < a[i]; j++)
105             if (!used[j]) cnt++;
106         res += n_permute_k(n - i - 1, k - i - 1) * cnt;
107         used[a[i]] = true;
108     }
109     return res;
110 }
111
112 /*
113
114 Changes a[] to the next lexicographically greater
115 permutation of k (not-necessarily distinct) integers in
116 range [0, n). The values of a[] should be in range [0, n).
117 If a[] was interpreted as a base-n integer that is k digits
118 long, this function would be equivalent to incrementing a.
119 Ergo, there are n^k arrangements if repeats are allowed.
120
121 returns: whether the function could rearrange a[] to a
122 lexicographically greater arrangement with repeats.
123
124 examples:
125 n_a_w_r(4, 3, {0, 0, 0}) => 1,  a[] = {0, 0, 1}
126 n_a_w_r(4, 3, {0, 1, 3}) => 1,  a[] = {0, 2, 0}
127 n_a_w_r(4, 3, {3, 3, 3}) => 0,  a[] unchanged
128
129 */
130
131 bool next_arrangement_with_repeats(int n, int k, int a[]) {
132     for (int i = k - 1; i >= 0; i--) {
133         if (a[i] < n - 1) {
134             a[i]++;
135             std::fill(a + i + 1, a + k, 0);
136             return true;
137         }
138     }
139     return false;
140 }
141
142 /** Example Usage */
143
144 #include <cassert>
145 #include <iostream>
146 using namespace std;
147
148 template<class it> void print(it lo, it hi) {
149     for (; lo != hi; ++lo) cout << *lo << " ";
150     cout << "\n";

```

```

151 }
152
153 int main() {
154     {
155         int n = 4, k = 3, a[] = {0, 1, 2};
156         cout << n << "└permute┘" << k << "└arrangements:\n";
157         int cnt = 0;
158         do {
159             print(a, a + k);
160             vector<int> b = arrangement_by_rank(n, k, cnt);
161             assert(equal(a, a + k, b.begin()));
162             assert(rank_by_arrangement(n, k, a) == cnt);
163             cnt++;
164         } while (next_arrangement(n, k, a));
165         cout << "\n";
166     }
167
168     {
169         int n = 4, k = 2, a[] = {0, 0};
170         cout << n << "└" << k << "└arrangements┘with┘repeats:\n";
171         do {
172             print(a, a + k);
173         } while (next_arrangement_with_repeats(n, k, a));
174     }
175     return 0;
176 }

```

4.2.3 Enumerating Permutations

```

1  /*
2
3  We shall consider a permutation of n objects to be an
4  ordered list of size n that contains all n elements,
5  where order is important. E.g. 1 1 2 0 and 0 1 2 1
6  are considered two different permutations of 0 1 1 2.
7  Compared to our prior definition of an arrangement, a
8  permutable range of size n may contain repeated values
9  of any type, not just the integers from 0 to n - 1.
10
11 */
12
13 #include <algorithm> /* copy, iter_swap, reverse, swap */
14 #include <vector>
15
16 //identical to std::next_permutation()
17 template<class It> bool _next_permutation(It lo, It hi) {
18     if (lo == hi) return false;
19     It i = lo;
20     if (++i == hi) return false;
21     i = hi; --i;
22     for (;;) {
23         It j = i; --i;
24         if (*i < *j) {
25             It k = hi;
26             while (!(*i < *--k)) /* pass */;
27             std::iter_swap(i, k);
28             std::reverse(j, hi);

```

```

29     return true;
30 }
31 if (i == lo) {
32     std::reverse(lo, hi);
33     return false;
34 }
35 }
36 }
37
38 //array version
39 template<class T> bool next_permutation(int n, T a[]) {
40     for (int i = n - 2; i >= 0; i--)
41         if (a[i] < a[i + 1])
42             for (int j = n - 1; j > i; j--)
43                 if (a[i] < a[j]) {
44                     std::swap(a[i++], a[j]);
45                     for (j = n - 1; i < j; i++, j--)
46                         std::swap(a[i], a[j]);
47                     return true;
48                 }
49     return false;
50 }
51
52 /*
53
54 Calls the custom function f(vector) on all permutations
55 of the integers from 0 to n - 1. This is more efficient
56 than making many consecutive calls to next_permutation(),
57 however, here, the permutations will not be printed in
58 lexicographically increasing order.
59
60 */
61
62 template<class ReportFunction>
63 void gen_permutations(int n, ReportFunction report,
64                      std::vector<int> & p, int d) {
65     if (d == n) {
66         report(p);
67         return;
68     }
69     for (int i = 0; i < n; i++) {
70         if (p[i] == 0) {
71             p[i] = d;
72             gen_permutations(n, report, p, d + 1);
73             p[i] = 0;
74         }
75     }
76 }
77
78 template<class ReportFunction>
79 void gen_permutations(int n, ReportFunction report) {
80     std::vector<int> perms(n, 0);
81     gen_permutations(n, report, perms, 0);
82 }
83
84 /*
85
86 Finds the next lexicographically greater permutation of
87 the binary digits of x. In other words, next_permutation()

```

```

88 simply returns the smallest integer greater than x which
89 has the same number of 1 bits (i.e. same popcount) as x.
90
91 examples: next_permutation(10101 base 2) = 10110
92           next_permutation(11100 base 2) = 100011
93
94 This can also be used to generate combinations as follows:
95 If we let k = popcount(x), then we can use this to generate
96 all possible masks to tell us which k items to take out of
97 n total items (represented by the first n bits of x).
98
99 */
100
101 long long next_permutation(long long x) {
102     long long s = x & -x, r = x + s;
103     return r | (((x ^ r) >> 2) / s);
104 }
105
106 /*
107
108 Given an integer rank x in range [0, n!), returns a vector
109 of integers representing the x-th lexicographically smallest
110 permutation of the integers in [0, n).
111
112 examples: permutation_by_rank(4, 0) => {0, 1, 2, 3}
113           permutation_by_rank(4, 5) => {0, 3, 2, 1}
114
115 */
116
117 std::vector<int> permutation_by_rank(int n, long long x) {
118     long long fact[n];
119     fact[0] = 1;
120     for (int i = 1; i < n; i++)
121         fact[i] = i * fact[i - 1];
122     std::vector<int> free(n), res(n);
123     for (int i = 0; i < n; i++) free[i] = i;
124     for (int i = 0; i < n; i++) {
125         int pos = x / fact[n - 1 - i];
126         res[i] = free[pos];
127         std::copy(free.begin() + pos + 1, free.end(),
128                 free.begin() + pos);
129         x %= fact[n - 1 - i];
130     }
131     return res;
132 }
133
134 /*
135
136 Given an array a[] of n integers each in range [0, n), returns
137 the (0-based) lexicographical rank (counting from least to
138 greatest) of the arrangement specified by a[] in all possible
139 permutations of the integers from 0 to n - 1.
140
141 examples: rank_by_permutation(3, {0, 1, 2}) => 0
142           rank_by_permutation(3, {2, 1, 0}) => 5
143
144 */
145
146 template<class T> long long rank_by_permutation(int n, T a[]) {

```

```

147     long long fact[n];
148     fact[0] = 1;
149     for (int i = 1; i < n; i++)
150         fact[i] = i * fact[i - 1];
151     long long res = 0;
152     for (int i = 0; i < n; i++) {
153         int v = a[i];
154         for (int j = 0; j < i; j++)
155             if (a[j] < a[i]) v--;
156         res += v * fact[n - 1 - i];
157     }
158     return res;
159 }
160
161 /*
162
163 Given a permutation a[] of the integers from 0 to n - 1,
164 returns a decomposition of the permutation into cycles.
165 A permutation cycle is a subset of a permutation whose
166 elements trade places with one another. For example, the
167 permutation {0, 2, 1, 3} decomposes to {0, 3, 2} and {1}.
168 Here, the notation {0, 3, 2} means that starting from the
169 original ordering {0, 1, 2, 3}, the 0th value is replaced
170 by the 3rd, the 3rd by the 2nd, and the 2nd by the first,
171 See: http://mathworld.wolfram.com/PermutationCycle.html
172
173 */
174
175 typedef std::vector<std::vector<int>> > cycles;
176
177 cycles decompose_into_cycles(int n, int a[]) {
178     std::vector<bool> vis(n);
179     cycles res;
180     for (int i = 0; i < n; i++) {
181         if (vis[i]) continue;
182         int j = i;
183         std::vector<int> cur;
184         do {
185             cur.push_back(j);
186             vis[j] = true;
187             j = a[j];
188         } while (j != i);
189         res.push_back(cur);
190     }
191     return res;
192 }
193
194 /** Example Usage */
195
196 #include <bitset>
197 #include <cassert>
198 #include <iostream>
199 using namespace std;
200
201 void printperm(const vector<int> & perm) {
202     for (int i = 0; i < (int)perm.size(); i++)
203         cout << perm[i] << " ";
204     cout << "\n";
205 }

```

```

206
207 template<class it> void print(it lo, it hi) {
208     for (; lo != hi; ++lo) cout << *lo << " ";
209     cout << "\n";
210 }
211
212 int main() {
213     { //method 1: ordered
214         int n = 4, a[] = {0, 1, 2, 3};
215         int b[n], c[n];
216         for (int i = 0; i < n; i++) b[i] = c[i] = a[i];
217         cout << "Ordered permutations of 0 to " << n-1 << ":\n";
218         int cnt = 0;
219         do {
220             print(a, a + n);
221             assert(equal(b, b + n, a));
222             assert(equal(c, c + n, a));
223             vector<int> d = permutation_by_rank(n, cnt);
224             assert(equal(d.begin(), d.end(), a));
225             assert(rank_by_permutation(n, a) == cnt);
226             cnt++;
227             std::next_permutation(b, b + n);
228             _next_permutation(c, c + n);
229         } while (next_permutation(n, a));
230         cout << "\n";
231     }
232
233     { //method 2: unordered
234         int n = 3;
235         cout << "Unordered permutations of 0 to " << n-1 << ":\n";
236         gen_permutations(n, printperm);
237         cout << "\n";
238     }
239
240     { //permuting binary digits
241         const int n = 5;
242         cout << "Permutations of 2 zeros and 3 ones:\n";
243         long long lo = 7; // 00111 in base 2
244         long long hi = 35; // 100011 in base 2
245         do {
246             cout << bitset<n>(lo).to_string() << "\n";
247         } while ((lo = next_permutation(lo)) != hi);
248         cout << "\n";
249     }
250
251     { //permutation cycles
252         int n = 4, a[] = {3, 1, 0, 2};
253         cout << "Decompose 0 2 1 3 into cycles:\n";
254         cycles c = decompose_into_cycles(n, a);
255         for (int i = 0; i < (int)c.size(); i++) {
256             cout << "Cycle " << i + 1 << ":\n";
257             for (int j = 0; j < (int)c[i].size(); j++)
258                 cout << " " << c[i][j];
259             cout << "\n";
260         }
261     }
262     return 0;
263 }

```

4.2.4 Enumerating Combinations

```

1  /*
2
3  We shall consider a combination n choose k to be an
4  set of k elements chosen from a total of n elements.
5  Unlike n permute k, the order here doesn't matter.
6  That is, 0 1 2 is considered the same as 0 2 1, so
7  we will consider the sorted representation of each
8  combination for purposes of the functions below.
9
10 */
11
12 #include <algorithm> /* iter_swap, rotate, swap, swap_ranges */
13 #include <iterator> /* std::iterator_traits */
14 #include <vector>
15
16 /*
17
18 Rearranges the values in the range [lo, hi) such that
19 elements in the range [lo, mid) becomes the next
20 lexicographically greater combination of the values from
21 [lo, hi) than it currently is, and returns whether the
22 function could rearrange [lo, hi) to a lexicographically
23 greater combination. If the range [lo, hi) contains n
24 elements and the range [lo, mid) contains k elements,
25 then starting off with a sorted range [lo, hi) and
26 calling next_combination() repeatedly will return true
27 for n choose k iterations before returning false.
28
29 */
30
31 template<class It>
32 bool next_combination(It lo, It mid, It hi) {
33     if (lo == mid || mid == hi) return false;
34     It l(mid - 1), h(hi - 1);
35     int sz1 = 1, sz2 = 1;
36     while (l != lo && !(*l < *h)) --l, ++sz1;
37     if (l == lo && !(*l < *h)) {
38         std::rotate(lo, mid, hi);
39         return false;
40     }
41     for (; mid < h; ++sz2) if (!(*l < *--h)) { ++h; break; }
42     if (sz1 == 1 || sz2 == 1) {
43         std::iter_swap(l, h);
44     } else if (sz1 == sz2) {
45         std::swap_ranges(l, mid, h);
46     } else {
47         std::iter_swap(l, h);
48         ++l; ++h; --sz1; --sz2;
49         int total = sz1 + sz2, gcd = total;
50         for (int i = sz1; i != 0; ) std::swap(gcd %= i, i);
51         int skip = total / gcd - 1;
52         for (int i = 0; i < gcd; i++) {
53             It curr(i < sz1 ? l + i : h + (i - sz1));
54             int k = i;
55             typename std::iterator_traits<It>::value_type v(*curr);
56             for (int j = 0; j < skip; j++) {

```

```

57         k = (k + sz1) % total;
58         It next(k < sz1 ? l + k : h + (k - sz1));
59         *curr = *next;
60         curr = next;
61     }
62     *curr = v;
63 }
64 }
65 return true;
66 }
67
68 /*
69
70 Changes a[] to the next lexicographically greater
71 combination of any k distinct integers in range [0, n).
72 The values of a[] that's passed should be k distinct
73 integers, each in range [0, n).
74
75 */
76
77 bool next_combination(int n, int k, int a[]) {
78     for (int i = k - 1; i >= 0; i--) {
79         if (a[i] < n - k + i) {
80             for (++a[i]; ++i < k; ) a[i] = a[i - 1] + 1;
81             return true;
82         }
83     }
84     return false;
85 }
86
87 /*
88
89 Finds the "mask" of the next combination of x. This is
90 equivalent to the next lexicographically greater permutation
91 of the binary digits of x. In other words, the function
92 simply returns the smallest integer greater than x which
93 has the same number of 1 bits (i.e. same popcount) as x.
94
95 examples: next_combination_mask(10101 base 2) = 10110
96           next_combination_mask(11100 base 2) = 100011
97
98 If we arbitrarily number the n items of our collection from
99 0 to n-1, then generating all combinations n choose k can
100 be done as follows: initialize x such that popcount(x) = k
101 and the first (least-significant) k bits are all set to 1
102 (e.g. to do 5 choose 3, start at x = 00111 (base 2) = 7).
103 Then, we repeatedly call x = next_combination_mask(x) until
104 we reach 11100 (the lexicographically greatest mask for 5
105 choose 3), after which we stop. At any point in the process,
106 we can say that the i-th item is being "taken" (0 <= i < n)
107 iff the i-th bit of x is set.
108
109 Note: this does not produce combinations in the same order
110 as next_combination, nor does it work if your n items have
111 repeated values (in that case, repeated combos will be
112 generated).
113
114 */
115

```



```

116 long long next_combination_mask(long long x) {
117     long long s = x & -x, r = x + s;
118     return r | (((x ^ r) >> 2) / s);
119 }
120
121 //n choose k in O(min(k, n - k))
122 long long n_choose_k(long long n, long long k) {
123     if (k > n - k) k = n - k;
124     long long res = 1;
125     for (int i = 0; i < k; i++)
126         res = res * (n - i) / (i + 1);
127     return res;
128 }
129
130 /*
131
132 Given an integer rank x in range [0, n choose k), returns
133 a vector of integers representing the x-th lexicographically
134 smallest combination k distinct integers in [0, n).
135
136 examples: combination_by_rank(4, 3, 0) => {0, 1, 2}
137           combination_by_rank(4, 3, 2) => {0, 2, 3}
138
139 */
140
141 std::vector<int> combination_by_rank(int n, int k, long long x) {
142     std::vector<int> res(k);
143     int cnt = n;
144     for (int i = 0; i < k; i++) {
145         int j = 1;
146         for (; j++) {
147             long long am = n_choose_k(cnt - j, k - 1 - i);
148             if (x < am) break;
149             x -= am;
150         }
151         res[i] = i > 0 ? (res[i - 1] + j) : (j - 1);
152         cnt -= j;
153     }
154     return res;
155 }
156
157 /*
158
159 Given an array a[] of k integers each in range [0, n), returns
160 the (0-based) lexicographical rank (counting from least to
161 greatest) of the combination specified by a[] in all possible
162 combination of k distinct integers in range [0, n).
163
164 examples: rank_by_combination(4, 3, {0, 1, 2}) => 0
165           rank_by_combination(4, 3, {0, 2, 3}) => 2
166
167 */
168
169 long long rank_by_combination(int n, int k, int a[]) {
170     long long res = 0;
171     int prev = -1;
172     for (int i = 0; i < k; i++) {
173         for (int j = prev + 1; j < a[i]; j++)
174             res += n_choose_k(n - 1 - j, k - 1 - i);

```

```

175     prev = a[i];
176 }
177 return res;
178 }
179
180 /*
181 Changes a[] to the next lexicographically greater
182 combination of any k (not necessarily distinct) integers
183 in range [0, n). The values of a[] that's passed should
184 be k integers, each in range [0, n). Note that there are
185 a total of n multichoose k combinations with repetition,
186 where n multichoose k = (n + k - 1) choose k
187
188 */
189
190 bool next_combination_with_repeats(int n, int k, int a[]) {
191     for (int i = k - 1; i >= 0; i--) {
192         if (a[i] < n - 1) {
193             for (++a[i]; ++i < k; ) a[i] = a[i - 1];
194             return true;
195         }
196     }
197     return false;
198 }
199
200
201 /** Example Usage */
202
203 #include <cassert>
204 #include <iostream>
205 using namespace std;
206
207 template<class it> void print(it lo, it hi) {
208     for (; lo != hi; ++lo) cout << *lo << " ";
209     cout << "\n";
210 }
211
212 int main() {
213     { //like std::next_permutation(), repeats in the range allowed
214         int k = 3;
215         string s = "11234";
216         cout << s << " choose " << k << ":\n";
217         do {
218             cout << s.substr(0, k) << "\n";
219         } while (next_combination(s.begin(), s.begin() + k, s.end()));
220         cout << "\n";
221     }
222
223     { //unordered combinations with masks
224         int n = 5, k = 3;
225         string s = "abcde"; //must be distinct values
226         cout << s << " choose " << k << " with masks:\n";
227         long long mask = 0, dest = 0;
228         for (int i = 0; i < k; i++) mask |= 1 << i;
229         for (int i = k - 1; i < n; i++) dest |= 1 << i;
230         do {
231             for (int i = 0; i < n; i++)
232                 if ((mask >> i) & 1) cout << s[i];
233             cout << "\n";

```

```

234     mask = next_combination_mask(mask);
235 } while (mask != dest);
236 cout << "\n";
237 }
238
239 { //only combinations of distinct integers from 0 to n - 1
240     int n = 5, k = 3, a[] = {0, 1, 2};
241     cout << n << " choose " << k << ":\n";
242     int cnt = 0;
243     do {
244         print(a, a + k);
245         vector<int> b = combination_by_rank(n, k, cnt);
246         assert(equal(a, a + k, b.begin()));
247         assert(rank_by_combination(n, k, a) == cnt);
248         cnt++;
249     } while (next_combination(n, k, a));
250     cout << "\n";
251 }
252
253 { //combinations with repetition
254     int n = 3, k = 2, a[] = {0, 0};
255     cout << n << " multichoose " << k << ":\n";
256     do {
257         print(a, a + k);
258     } while (next_combination_with_repeats(n, k, a));
259 }
260 return 0;
261 }

```

4.2.5 Enumerating Partitions

```

1  /*
2
3  We shall consider a partition of an integer n to be an
4  unordered multiset of positive integers that has a total
5  sum equal to n. Since both 2 1 1 and 1 2 1 represent the
6  same partition of 4, we shall consider only descending
7  sorted lists as "valid" partitions for functions below.
8
9  */
10
11 #include <vector>
12
13 /*
14
15 Given a vector representing a partition of some
16 integer n (the sum of all values in the vector),
17 changes p to the next lexicographically greater
18 partition of n and returns whether the change was
19 successful (whether a lexicographically greater
20 partition existed). Note that the "initial" value
21 of p must be a vector of size n, all initialized 1.
22
23 e.g. next_partition({2, 1, 1}) => 1, p becomes {2, 2}
24     next_partition({2, 2})      => 1, p becomes {3, 1}
25     next_partition({4})         => 0, p is unchanged
26

```

```

27  */
28
29  bool next_partition(std::vector<int> & p) {
30      int n = p.size();
31      if (n <= 1) return false;
32      int s = p[n - 1] - 1, i = n - 2;
33      p.pop_back();
34      for (; i > 0 && p[i] == p[i - 1]; i--) {
35          s += p[i];
36          p.pop_back();
37      }
38      for (p[i]++; s-- > 0; ) p.push_back(1);
39      return true;
40  }
41
42  /* Returns the number of partitions of n. */
43
44  long long count_partitions(int n) {
45      std::vector<long long> p(n + 1, 0);
46      p[0] = 1;
47      for (int i = 1; i <= n; i++)
48          for (int j = i; j <= n; j++)
49              p[j] += p[j - i];
50      return p[n];
51  }
52
53  /* Helper function for partitioning by rank */
54
55  std::vector< std::vector<long long> >
56  p(1, std::vector<long long>(1, 1)); //memoization
57
58  long long partition_function(int a, int b) {
59      if (a >= (int)p.size()) {
60          int old = p.size();
61          p.resize(a + 1);
62          p[0].resize(a + 1);
63          for (int i = 1; i <= a; i++) {
64              p[i].resize(a + 1);
65              for (int j = old; j <= i; j++)
66                  p[i][j] = p[i - 1][j - 1] + p[i - j][j];
67          }
68      }
69      return p[a][b];
70  }
71
72  /*
73
74  Given an integer n to partition and a 0-based rank x,
75  returns a vector of integers representing the x-th
76  lexicographically smallest partition of n (if values
77  in each partition were sorted in decreasing order).
78
79  examples: partition_by_rank(4, 0) => {1, 1, 1, 1}
80            partition_by_rank(4, 3) => {3, 1}
81
82  */
83
84  std::vector<int> partition_by_rank(int n, long long x) {
85      std::vector<int> res;

```

```

86     for (int i = n; i > 0; ) {
87         int j = 1;
88         for (; j++) {
89             long long cnt = partition_function(i, j);
90             if (x < cnt) break;
91             x -= cnt;
92         }
93         res.push_back(j);
94         i -= j;
95     }
96     return res;
97 }
98
99 /*
100
101 Given a partition of an integer n (sum of all values
102 in vector p), returns a 0-based rank x of the partition
103 represented by p, considering partitions from least to
104 greatest in lexicographical order (if each partition
105 had values sorted in descending order).
106
107 examples: rank_by_partition({1, 1, 1, 1}) => 0
108           rank_by_partition({3, 1})      => 3
109
110 */
111
112 long long rank_by_partition(const std::vector<int> & p) {
113     long long res = 0;
114     int sum = 0;
115     for (int i = 0; i < (int)p.size(); i++) sum += p[i];
116     for (int i = 0; i < (int)p.size(); i++) {
117         for (int j = 0; j < p[i]; j++)
118             res += partition_function(sum, j);
119         sum -= p[i];
120     }
121     return res;
122 }
123
124 /*
125
126 Calls the custom function f(vector) on all partitions
127 which consist of strictly *increasing* integers.
128 This will exclude partitions such as {1, 1, 1, 1}.
129
130 */
131
132 template<class ReportFunction>
133 void gen_increasing_partitions(int left, int prev, int i,
134                               ReportFunction f, std::vector<int> & p) {
135     if (left == 0) {
136         //warning: slow constructor - modify accordingly
137         f(std::vector<int>(p.begin(), p.begin() + i));
138         return;
139     }
140     for (p[i] = prev + 1; p[i] <= left; p[i]++)
141         gen_increasing_partitions(left - p[i], p[i], i + 1, f, p);
142 }
143
144 template<class ReportFunction>

```

```

145 void gen_increasing_partitions(int n, ReportFunction f) {
146     std::vector<int> partitions(n, 0);
147     gen_increasing_partitions(n, 0, 0, f, partitions);
148 }
149
150 /** Example Usage */
151
152 #include <cassert>
153 #include <iostream>
154 using namespace std;
155
156 void print(const vector<int> & v) {
157     for (int i = 0; i < (int)v.size(); i++)
158         cout << v[i] << " ";
159     cout << "\n";
160 }
161
162 int main() {
163     assert(count_partitions(5) == 7);
164     assert(count_partitions(20) == 627);
165     assert(count_partitions(30) == 5604);
166     assert(count_partitions(50) == 204226);
167     assert(count_partitions(100) == 190569292);
168
169     {
170         int n = 4;
171         vector<int> a(n, 1);
172         cout << "Partitions of " << n << ":\n";
173         int cnt = 0;
174         do {
175             print(a);
176             vector<int> b = partition_by_rank(n, cnt);
177             assert(equal(a.begin(), a.end(), b.begin()));
178             assert(rank_by_partition(a) == cnt);
179             cnt++;
180         } while (next_partition(a));
181         cout << "\n";
182     }
183
184     {
185         int n = 8;
186         cout << "Increasing partitions of " << n << ":\n";
187         gen_increasing_partitions(n, print);
188     }
189     return 0;
190 }

```

4.2.6 Enumerating Generic Combinatorial Sequences

```

1  /*
2
3  The follow provides a universal method for enumerating
4  abstract combinatorial sequences in  $O(n^2)$  time.
5
6  */
7
8  #include <vector>

```

```

9
10 class abstract_enumeration {
11     protected:
12         int range, length;
13
14     abstract_enumeration(int r, int l): range(r), length(l) {}
15
16     virtual long long count(const std::vector<int> & pre) {
17         return 0;
18     }
19
20     std::vector<int> next(std::vector<int> & seq) {
21         return from_number(to_number(seq) + 1);
22     }
23
24     long long total_count() {
25         return count(std::vector<int>(0));
26     }
27
28 public:
29     long long to_number(const std::vector<int> & seq) {
30         long long res = 0;
31         for (int i = 0; i < (int)seq.size(); i++) {
32             std::vector<int> pre(seq.begin(), seq.end());
33             pre.resize(i + 1);
34             for (pre[i] = 0; pre[i] < seq[i]; ++pre[i])
35                 res += count(pre);
36         }
37         return res;
38     }
39
40     std::vector<int> from_number(long long x) {
41         std::vector<int> seq(length);
42         for (int i = 0; i < (int)seq.size(); i++) {
43             std::vector<int> pre(seq.begin(), seq.end());
44             pre.resize(i + 1);
45             for (pre[i] = 0; pre[i] < range; ++pre[i]) {
46                 long long cur = count(pre);
47                 if (x < cur) break;
48                 x -= cur;
49             }
50             seq[i] = pre[i];
51         }
52         return seq;
53     }
54
55     template<class ReportFunction>
56     void enumerate(ReportFunction report) {
57         long long total = total_count();
58         for (long long i = 0; i < total; i++) {
59             //assert(i == to_number(from_number(i)));
60             report(from_number(i));
61         }
62     }
63 };
64
65 class arrangements: public abstract_enumeration {
66 public:
67     arrangements(int n, int k) : abstract_enumeration(n, k) {}

```

```

68
69     long long count(const std::vector<int> & pre) {
70         int sz = pre.size();
71         for (int i = 0; i < sz - 1; i++)
72             if (pre[i] == pre[sz - 1]) return 0;
73         long long res = 1;
74         for (int i = 0; i < length - sz; i++)
75             res *= range - sz - i;
76         return res;
77     }
78 };
79
80 class permutations: public arrangements {
81 public:
82     permutations(int n) : arrangements(n, n) {}
83 };
84
85 class combinations: public abstract_enumeration {
86     std::vector<std::vector<long long> > binomial;
87
88 public:
89     combinations(int n, int k) : abstract_enumeration(n, k),
90         binomial(n + 1, std::vector<long long>(n + 1, 0)) {
91         for (int i = 0; i <= n; i++)
92             for (int j = 0; j <= i; j++)
93                 binomial[i][j] = (j == 0) ? 1 :
94                     binomial[i - 1][j - 1] + binomial[i - 1][j];
95     }
96
97     long long count(const std::vector<int> & pre) {
98         int sz = pre.size();
99         if (sz >= 2 && pre[sz - 1] <= pre[sz - 2]) return 0;
100         int last = sz > 0 ? pre[sz - 1] : -1;
101         return binomial[range - 1 - last][length - sz];
102     }
103 };
104
105 class partitions: public abstract_enumeration {
106     std::vector<std::vector<long long> > p;
107
108 public:
109     partitions(int n) : abstract_enumeration(n + 1, n),
110         p(n + 1, std::vector<long long>(n + 1, 0)) {
111         std::vector<std::vector<long long> > pp(p);
112         pp[0][0] = 1;
113         for (int i = 1; i <= n; i++)
114             for (int j = 1; j <= i; j++)
115                 pp[i][j] = pp[i - 1][j - 1] + pp[i - j][j];
116         for (int i = 1; i <= n; i++)
117             for (int j = 1; j <= n; j++)
118                 p[i][j] = pp[i][j] + p[i][j - 1];
119     }
120
121     long long count(const std::vector<int> & pre) {
122         int size = pre.size(), sum = 0;
123         for (int i = 0; i < (int)pre.size(); i++) sum += pre[i];
124         if (sum == range - 1) return 1;
125         if (sum > range - 1 || (size > 0 && pre[size - 1] == 0) ||
126             (size >= 2 && pre[size - 1] > pre[size - 2])) return 0;

```



```

127     int last = size > 0 ? pre[size - 1] : range - 1;
128     return p[range - 1 - sum][last];
129 }
130 };
131
132 /** Example Usage */
133
134 #include <iostream>
135 using namespace std;
136
137 void print(const std::vector<int> & v) {
138     for (int i = 0; i < (int)v.size(); i++)
139         cout << v[i] << " ";
140     cout << "\n";
141 }
142
143 int main() {
144     cout << "Arrangement(3,2):\n";
145     arrangements arrg(3, 2);
146     arrg.enumerate(print);
147
148     cout << "Permutation(3):\n";
149     permutations perm(3);
150     perm.enumerate(print);
151
152     cout << "Combination(4,3):\n";
153     combinations comb(4, 3);
154     comb.enumerate(print);
155
156     cout << "Partition(4):\n";
157     partitions part(4);
158     part.enumerate(print);
159     return 0;
160 }

```

4.3 Number Theory

4.3.1 GCD, LCM, Mod Inverse, Chinese Remainder

```

1  /*
2
3  GCD, LCM, Modular Inverse, Chinese Remainder Theorem
4
5  */
6
7  #include <utility> /* std::pair */
8  #include <vector>
9
10 //C++98 does not have abs() declared for long long
11 template<class T> inline T _abs(const T & x) {
12     return x < 0 ? -x : x;
13 }
14
15 //GCD using Euclid's algorithm - O(log(a + b))
16 template<class Int> Int gcd(Int a, Int b) {
17     return b == 0 ? _abs(a) : gcd(b, a % b);

```

```

18 }
19
20 //non-recursive version
21 template<class Int> Int gcd2(Int a, Int b) {
22     while (b != 0) {
23         Int t = b;
24         b = a % b;
25         a = t;
26     }
27     return _abs(a);
28 }
29
30 template<class Int> Int lcm(Int a, Int b) {
31     return _abs(a / gcd(a, b) * b);
32 }
33
34 //returns <gcd(a, b), <x, y>> such that gcd(a, b) = ax + by
35 template<class Int>
36 std::pair<Int, std::pair<Int, Int> > euclid(Int a, Int b) {
37     Int x = 1, y = 0, x1 = 0, y1 = 1;
38     //invariant: a = a * x + b * y, b = a * x1 + b * y1
39     while (b != 0) {
40         Int q = a / b, _x1 = x1, _y1 = y1, _b = b;
41         x1 = x - q * x1;
42         y1 = y - q * y1;
43         b = a - q * b;
44         x = _x1;
45         y = _y1;
46         a = _b;
47     }
48     return a > 0 ? std::make_pair(a, std::make_pair(x, y)) :
49         std::make_pair(-a, std::make_pair(-x, -y));
50 }
51
52 //recursive version
53 template<class Int>
54 std::pair<Int, std::pair<Int, Int> > euclid2(Int a, Int b) {
55     if (b == 0) {
56         return a > 0 ? std::make_pair(a, std::make_pair(1, 0)) :
57             std::make_pair(-a, std::make_pair(-1, 0));
58     }
59     std::pair<Int, std::pair<Int, Int> > r = euclid2(b, a % b);
60     return std::make_pair(r.first, std::make_pair(r.second.second,
61         r.second.first - a / b * r.second.second));
62 }
63
64 /*
65
66 Modulo Operation - Euclidean Definition
67
68 The % operator in C/C++ returns the remainder of division (which
69 may be positive or negative) The true Euclidean definition of
70 modulo, however, defines the remainder to be always nonnegative.
71 For positive operators, % and mod are the same. But for negative
72 operands, they differ. The result here is consistent with the
73 Euclidean division algorithm.
74
75 e.g. -21 % 4 == -1 since -21 / 4 == -5 and 4 * -5 + (-1) == -21
76     however, -21 mod 4 is equal to 3 because -21 + 4 * 6 is 3.

```

```

77
78 */
79
80 template<class Int> Int mod(Int a, Int m) {
81     Int r = (Int)(a % m);
82     return r >= 0 ? r : r + m;
83 }
84
85 //returns x such that a * x = 1 (mod m)
86 //precondition: m > 0 && gcd(a, m) = 1
87 template<class Int> Int mod_inverse(Int a, Int m) {
88     a = mod(a, m);
89     return a == 0 ? 0 : mod((1 - m * mod_inverse(m % a, a)) / a, m);
90 }
91
92 //precondition: m > 0 && gcd(a, m) = 1
93 template<class Int> Int mod_inverse2(Int a, Int m) {
94     return mod(euclid(a, m).second.first, m);
95 }
96
97 //returns a vector where i*v[i] = 1 (mod p) in O(p) time
98 //precondition: p is prime
99 std::vector<int> generate_inverses(int p) {
100     std::vector<int> res(p);
101     res[1] = 1;
102     for (int i = 2; i < p; i++)
103         res[i] = (p - (p / i) * res[p % i] % p) % p;
104     return res;
105 }
106
107 /*
108
109 Chinese Remainder Theorem
110
111 Let r and s be positive integers which are relatively prime and
112 let a and b be any two integers. Then there exists an integer N
113 such that  $N \equiv a \pmod{r}$  and  $N \equiv b \pmod{s}$ . Moreover, N is
114 uniquely determined modulo rs.
115
116 More generally, given a set of simultaneous congruences for
117 which all values in p[] are pairwise relative prime:
118
119  $x \equiv a[i] \pmod{p[i]}$ , for  $i = 1..n$ 
120
121 the solution of the set of congruences is:
122
123  $x \equiv a[1] * b[1] * (M/p[1]) + \dots + a[n] * b[n] * (M/p[n]) \pmod{M}$ 
124
125 where  $M = p[1] * p[2] \dots * p[n]$  and the b[i] are determined for
126
127  $b[i] * (M/p[i]) \equiv 1 \pmod{p[i]}$ .
128
129 The following functions solves for this value of x, with the
130 first function computed using the method above while the
131 second function using a special case of Garner's algorithm.
132
133 http://e-maxx-eng.github.io/algebra/chinese-remainder-theorem.html
134
135 */

```

```

136
137 long long simple_restore(int n, int a[], int p[]) {
138     long long res = 0, m = 1;
139     for (int i = 0; i < n; i++) {
140         while (res % p[i] != a[i]) res += m;
141         m *= p[i];
142     }
143     return res;
144 }
145
146 long long garner_restore(int n, int a[], int p[]) {
147     int x[n];
148     for (int i = 0; i < n; i++) x[i] = a[i];
149     for (int i = 0; i < n; i++) {
150         for (int j = 0; j < i; j++)
151             x[i] = mod_inverse((long long)p[j], (long long)p[i]) *
152                     (long long)(x[i] - x[j]);
153         x[i] = (x[i] % p[i] + p[i]) % p[i];
154     }
155     long long res = x[0], m = 1;
156     for (int i = 1; i < n; i++) {
157         m *= p[i - 1];
158         res += x[i] * m;
159     }
160     return res;
161 }
162
163 /** Example Usage */
164
165 #include <cassert>
166 #include <cstdlib>
167 #include <ctime>
168 #include <iostream>
169 using namespace std;
170
171 int main() {
172     {
173         srand(time(0));
174         for (int steps = 0; steps < 10000; steps++) {
175             int a = rand() % 200 - 10;
176             int b = rand() % 200 - 10;
177             int g1 = gcd(a, b), g2 = gcd2(a, b);
178             assert(g1 == g2);
179             if (g1 == 1 && b > 1) {
180                 int inv1 = mod_inverse(a, b);
181                 int inv2 = mod_inverse2(a, b);
182                 assert(inv1 == inv2 && mod(a * inv1, b) == 1);
183             }
184             pair<int, pair<int, int> > euc1 = euclid(a, b);
185             pair<int, pair<int, int> > euc2 = euclid2(a, b);
186             assert(euc1.first == g1 && euc1 == euc2);
187             int x = euc1.second.first;
188             int y = euc1.second.second;
189             assert(g1 == a * x + b * y);
190         }
191     }
192
193     {
194         long long a = 6, b = 9;

```

```

195     pair<int, pair<int, int> > r = euclid(6, 9);
196     cout << r.second.first << " * " << a << " ) " << " + ";
197     cout << r.second.second << " * " << b << " ) = gcd ";
198     cout << a << " , " << b << " ) = " << r.first << " \n";
199 }
200
201 {
202     int prime = 17;
203     std::vector<int> res = generate_inverses(prime);
204     for (int i = 0; i < prime; i++) {
205         if (i > 0) assert(mod(i * res[i], prime) == 1);
206         cout << res[i] << " ";
207     }
208     cout << " \n";
209 }
210
211 {
212     int n = 3, a[] = {2, 3, 1}, m[] = {3, 4, 5};
213     //solves for x in the simultaneous congruences:
214     //x = 2 (mod 3)
215     //x = 3 (mod 4)
216     //x = 1 (mod 5)
217     int x1 = simple_restore(n, a, m);
218     int x2 = garner_restore(n, a, m);
219     assert(x1 == x2);
220     for (int i = 0; i < n; i++)
221         assert(mod(x1, m[i]) == a[i]);
222     cout << "Solution: " << x1 << " \n"; //11
223 }
224
225 return 0;
226 }

```

4.3.2 Generating Primes

```

1  /*
2
3  The following are three methods to generate primes.
4  Although the latter two functions are theoretically
5  linear, the former function with the sieve of
6  Eratosthenes is still significantly the fastest even
7  for n under 1 billion, since its constant factor is
8  so much better because of its minimal arithmetic
9  operations. For this reason, it should be favored
10 over the other two algorithms in most contest
11 applications. For the computation of larger primes,
12 you should replace int with long long or an arbitrary
13 precision class.
14
15 */
16
17 #include <cmath> /* ceil(), sqrt() */
18 #include <vector>
19
20 //Sieve of Eratosthenes in ~ O(n log log n)
21 //returns: a vector of all primes under n
22 std::vector<int> gen_primes(int n) {

```

```

23     std::vector<bool> prime(n + 1, true);
24     int sqrtn = (int)ceil(sqrt(n));
25     for (int i = 2; i <= sqrtn; i++) {
26         if (prime[i])
27             for (int j = i * i; j <= n; j += i)
28                 prime[j] = false;
29     }
30     std::vector<int> res;
31     for (int i = 2; i <= n; i++)
32         if (prime[i]) res.push_back(i);
33     return res;
34 }
35
36 //Technically O(n), but on -O2, this is about
37 //as fast as the above sieve for n = 100 million
38 std::vector<int> gen_primes_linear(int n) {
39     std::vector<int> lp(n + 1), res;
40     for (int i = 2; i <= n; i++) {
41         if (lp[i] == 0) {
42             lp[i] = i;
43             res.push_back(i);
44         }
45         for (int j = 0; j < (int)res.size(); j++) {
46             if (res[j] > lp[i] || i * res[j] > n)
47                 break;
48             lp[i * res[j]] = res[j];
49         }
50     }
51     return res;
52 }
53
54 //Sieve of Atkins in O(n), somewhat slow due to
55 //its heavier arithmetic compared to the above
56 std::vector<int> gen_primes_atkins(int n) {
57     std::vector<bool> prime(n + 1, false);
58     std::vector<int> res;
59     prime[2] = true;
60     prime[3] = true;
61     int num, lim = ceil(sqrt(n));
62     for (int x = 1; x <= lim; x++) {
63         for (int y = 1; y <= lim; y++) {
64             num = 4 * x * x + y * y;
65             if (num <= n && (num % 12 == 1 || num % 12 == 5))
66                 prime[num] = true;
67             num = 3 * x * x + y * y;
68             if (num <= n && (num % 12 == 7))
69                 prime[num] = true;
70             if (x > y) {
71                 num = (3 * x * x - y * y);
72                 if (num <= n && num % 12 == 11)
73                     prime[num] = true;
74             }
75         }
76     }
77     for (int i = 5; i <= lim; i++) {
78         if (prime[i])
79             for (int j = i * i; j <= n; j += i)
80                 prime[j] = false;
81     }

```

```

82     for (int i = 2; i <= n; i++)
83         if (prime[i]) res.push_back(i);
84     return res;
85 }
86
87 //Double sieve to find primes in [1, h]
88 //Approximately  $O(\sqrt{h} * \log \log(h - 1))$ 
89 std::vector<int> gen_primes(int l, int h) {
90     int sqrth = (int)ceil(sqrt(h));
91     int sqrtsqrth = (int)ceil(sqrt(sqrth));
92     std::vector<bool> prime1(sqrth + 1, true);
93     std::vector<bool> prime2(h - l + 1, true);
94     for (int i = 2; i <= sqrtsqrth; i++) {
95         if (prime1[i])
96             for (int j = i * i; j <= sqrth; j += i)
97                 prime1[j] = false;
98     }
99     for (int i = 2, n = h - l + 1; i <= sqrth; i++) {
100         if (prime1[i])
101             for (int j = l / i * i - l; j <= n; j += i)
102                 if (j >= 0 && j + 1 != i)
103                     prime2[j] = false;
104     }
105     std::vector<int> res;
106     for (int i = l > 1 ? l : 2; i <= h; i++)
107         if (prime2[i - l]) res.push_back(i);
108     return res;
109 }
110
111 /** Example Usage **/
112
113 #include <cassert>
114 #include <ctime>
115 #include <iostream>
116 using namespace std;
117
118 template<class It> void print(It lo, It hi) {
119     while (lo != hi) cout << *(lo++) << " ";
120     cout << "\n";
121 }
122
123 int main() {
124     int pmax = 10000000;
125     vector<int> p;
126     time_t start;
127     double delta;
128
129     cout << "Generating primes up to " << pmax << "... \n";
130     start = clock();
131     p = gen_primes(pmax);
132     delta = (double)(clock() - start)/CLOCKS_PER_SEC;
133     cout << "gen_primes() took " << delta << "s. \n";
134
135     start = clock();
136     p = gen_primes_linear(pmax);
137     delta = (double)(clock() - start)/CLOCKS_PER_SEC;
138     cout << "gen_primes_linear() took " << delta << "s. \n";
139
140     start = clock();

```

```

141 p = gen_primes_atkins(pmax);
142 delta = (double)(clock() - start)/CLOCKS_PER_SEC;
143 cout << "gen_primes_atkins()_took_" << delta << "s.\n";
144
145 cout << "Generated_" << p.size() << "_primes.\n";
146 //print(p.begin(), p.end());
147
148 for (int i = 0; i <= 1000; i++) {
149     assert(gen_primes(i) == gen_primes_linear(i));
150     assert(gen_primes(i) == gen_primes_atkins(i));
151 }
152
153 int l = 1000000000, h = 1000000500;
154 cout << "Generating_primes_in_" << l << ",_" << h << "]...\n";
155 start = clock();
156 p = gen_primes(l, h);
157 delta = (double)(clock() - start)/CLOCKS_PER_SEC;
158 cout << "Generated_" << p.size() << "_primes_in_" << delta << "s.\n";
159 print(p.begin(), p.end());
160 return 0;
161 }

```

4.3.3 Primality Testing

```

1  /*
2
3  Primality Testing
4
5  */
6
7  #include <cstdlib> /* rand(), srand() */
8  #include <ctime> /* time() */
9  #include <stdint.h> /* uint64_t */
10
11 /*
12
13 Trial division in O(sqrt(n)) to return whether n is prime
14 Applies an optimization based on the fact that all
15 primes greater than 3 take the form 6n + 1 or 6n - 1.
16
17 */
18
19 template<class Int> bool is_prime(Int n) {
20     if (n == 2 || n == 3) return true;
21     if (n < 2 || !(n % 2) || !(n % 3)) return false;
22     for (Int i = 5, w = 4; i * i <= n; i += (w = 6 - w))
23         if (n % i == 0) return false;
24     return true;
25 }
26
27 /*
28
29 Miller-Rabin Primality Test (Probabilistic)
30
31 Checks whether a number n is probably prime. If n is prime,
32 the function is guaranteed to return 1. If n is composite,
33 the function returns 1 with a probability of (1/4)^k,

```



```

34 where k is the number of iterations. With k = 1, the
35 probability of a composite being falsely predicted to be a
36 prime is 25%. If k = 5, the probability for this error is
37 just less than 0.1%. Thus, k = 18 to 20 is accurate enough
38 for most applications. All values of  $n < 2^{63}$  is supported.
39
40 Complexity:  $O(k \log^3(n))$ . In comparison to trial division,
41 the Miller-Rabin algorithm on 32-bit ints take ~45
42 operations for k = 10 iterations (~0.0001% error), while the
43 former takes ~10,000.
44
45 Warning: Due to the overflow of modular exponentiation,
46 this will only work on inputs less than  $2^{63}$ .
47
48 */
49
50 uint64_t mulmod(uint64_t a, uint64_t b, uint64_t m) {
51     uint64_t x = 0, y = a % m;
52     for (; b > 0; b >>= 1) {
53         if (b & 1) x = (x + y) % m;
54         y = (y << 1) % m;
55     }
56     return x % m;
57 }
58
59 uint64_t powmod(uint64_t a, uint64_t b, uint64_t m) {
60     uint64_t x = 1, y = a;
61     for (; b > 0; b >>= 1) {
62         if (b & 1) x = mulmod(x, y, m);
63         y = mulmod(y, y, m);
64     }
65     return x % m;
66 }
67
68 //5 calls to rand() is unnecessary if RAND_MAX is  $2^{31}-1$ 
69 uint64_t rand64u() {
70     return ((uint64_t)(rand() & 0xf) << 60) |
71         ((uint64_t)(rand() & 0x7fff) << 45) |
72         ((uint64_t)(rand() & 0x7fff) << 30) |
73         ((uint64_t)(rand() & 0x7fff) << 15) |
74         ((uint64_t)(rand() & 0x7fff));
75 }
76
77 bool is_probable_prime(long long n, int k = 20) {
78     if (n < 2 || (n != 2 && !(n & 1))) return false;
79     uint64_t s = n - 1, p = n - 1, x, r;
80     while (!(s & 1)) s >>= 1;
81     for (int i = 0; i < k; i++) {
82         r = powmod(rand64u() % p + 1, s, n);
83         for (x = s; x != p && r != 1 && r != p; x <<= 1)
84             r = mulmod(r, r, n);
85         if (r != p && !(x & 1)) return false;
86     }
87     return true;
88 }
89
90 /*
91
92 Miller-Rabin - Deterministic for all unsigned long long

```

```

93
94 Although Miller-Rabin is generally probabilistic, the seven
95 bases 2, 325, 9375, 28178, 450775, 9780504, 1795265022 have
96 been proven to deterministically test the primality of all
97 numbers under  $2^{64}$ . See: http://miller-rabin.appspot.com/
98
99 Complexity:  $O(\log^3(n))$ .
100 Warning: Due to the overflow of modular exponentiation,
101         this will only work on inputs less than  $2^{63}$ .
102
103 */
104
105 bool is_prime_fast(long long n) {
106     static const uint64_t witnesses[] =
107         {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
108     if (n <= 1) return false;
109     if (n <= 3) return true;
110     if ((n & 1) == 0) return false;
111     uint64_t d = n - 1;
112     int s = 0;
113     for (; ~d & 1; s++) d >>= 1;
114     for (int i = 0; i < 7; i++) {
115         if (witnesses[i] > (uint64_t)n - 2) break;
116         uint64_t x = powmod(witnesses[i], d, n);
117         if (x == 1 || x == (uint64_t)n - 1) continue;
118         bool flag = false;
119         for (int j = 0; j < s; j++) {
120             x = powmod(x, 2, n);
121             if (x == 1) return false;
122             if (x == (uint64_t)n - 1) {
123                 flag = true;
124                 break;
125             }
126         }
127         if (!flag) return false;
128     }
129     return true;
130 }
131
132 /** Example Usage */
133
134 #include <cassert>
135
136 int main() {
137     int len = 20;
138     unsigned long long v[] = {
139         0, 1, 2, 3, 4, 5, 11,
140         1000000ull,
141         772023803ull,
142         792904103ull,
143         813815117ull,
144         834753187ull,
145         855718739ull,
146         876717799ull,
147         897746119ull,
148         2147483647ull,
149         5705234089ull,
150         5914686649ull,
151         6114145249ull,

```

```

152     6339503641ull,
153     6548531929ull
154 };
155 for (int i = 0; i < len; i++) {
156     bool p = is_prime(v[i]);
157     assert(p == is_prime_fast(v[i]));
158     assert(p == is_probable_prime(v[i]));
159 }
160 return 0;
161 }

```

4.3.4 Integer Factorization

```

1  /*
2
3  Integer Factorization
4
5  */
6
7  #include <algorithm> /* std::sort() */
8  #include <cmath>     /* sqrt() */
9  #include <cstdlib>   /* rand(), srand() */
10 #include <stdint.h>  /* uint64_t */
11 #include <vector>
12
13 /*
14
15 Trial division in O(sqrt(n))
16
17 Returns a vector of pair<prime divisor, exponent>
18 e.g. prime_factorize(15435) => {(3,2),(5,1),(7,3)}
19 because 3^2 * 5^1 * 7^3 = 15435
20
21 */
22
23 template<class Int>
24 std::vector<std::pair<Int, int> > prime_factorize(Int n) {
25     std::vector<std::pair<Int, int> > res;
26     for (Int d = 2; ; d++) {
27         int power = 0, quot = n / d, rem = n - quot * d;
28         if (d > quot || (d == quot && rem > 0)) break;
29         for (; rem == 0; rem = n - quot * d) {
30             power++;
31             n = quot;
32             quot = n / d;
33         }
34         if (power > 0) res.push_back(std::make_pair(d, power));
35     }
36     if (n > 1) res.push_back(std::make_pair(n, 1));
37     return res;
38 }
39
40 /*
41
42 Trial division in O(sqrt(n))
43
44 Returns a sorted vector of all divisors of n.

```

```

45 e.g. get_all_divisors(28) => {1, 2, 4, 7, 14, 28}
46
47 */
48
49 template<class Int>
50 std::vector<Int> get_all_divisors(Int n) {
51     std::vector<Int> res;
52     for (int d = 1; d * d <= n; d++) {
53         if (n % d == 0) {
54             res.push_back(d);
55             if (d * d != n)
56                 res.push_back(n / d);
57         }
58     }
59     std::sort(res.begin(), res.end());
60     return res;
61 }
62
63 /*
64
65 Fermat's Method ~ O(sqrt(N))
66
67 Given a number n, returns one factor of n that is
68 not necessary prime. Fermat's algorithm is pretty
69 good when the number you wish to factor has two
70 factors very near to sqrt(n). Otherwise, it is just
71 as slow as the basic trial division algorithm.
72
73 e.g. 14917627 => 1 (it's a prime), or
74     1234567 => 127 (because 127*9721 = 1234567)
75
76 */
77
78 long long fermat(long long n) {
79     if (n % 2 == 0) return 2;
80     long long x = sqrt(n), y = 0;
81     long long r = x * x - y * y - n;
82     while (r != 0) {
83         if (r < 0) {
84             r += x + x + 1;
85             x++;
86         } else {
87             r -= y + y + 1;
88             y++;
89         }
90     }
91     return x != y ? x - y : x + y;
92 }
93
94 /*
95
96 Pollard's rho Algorithm with Brent's Optimization
97
98 Brent's algorithm is a much faster variant of Pollard's
99 rho algorithm using Brent's cycle-finding method. The
100 following function returns a (not necessarily prime) factor
101 of n, or n if n is prime. Note that this is not necessarily
102 guaranteed to always work perfectly. brent(9) may return 9
103 instead of 3. However, it works well when coupled with trial

```

```

104 division in the function prime_factorize_big() below.
105
106 */
107
108 uint64_t mulmod(uint64_t a, uint64_t b, uint64_t m) {
109     uint64_t x = 0, y = a % m;
110     for (; b > 0; b >>= 1) {
111         if (b & 1) x = (x + y) % m;
112         y = (y << 1) % m;
113     }
114     return x % m;
115 }
116
117 //5 calls to rand() is unnecessary if RAND_MAX is 2^31-1
118 uint64_t rand64u() {
119     return ((uint64_t)(rand() & 0xf) << 60) |
120           ((uint64_t)(rand() & 0x7fff) << 45) |
121           ((uint64_t)(rand() & 0x7fff) << 30) |
122           ((uint64_t)(rand() & 0x7fff) << 15) |
123           ((uint64_t)(rand() & 0x7fff));
124 }
125
126 uint64_t gcd(uint64_t a, uint64_t b) {
127     return b == 0 ? a : gcd(b, a % b);
128 }
129
130 long long brent(long long n) {
131     if (n % 2 == 0) return 2;
132     long long y = rand64u() % (n - 1) + 1;
133     long long c = rand64u() % (n - 1) + 1;
134     long long m = rand64u() % (n - 1) + 1;
135     long long g = 1, r = 1, q = 1, ys = 0, hi = 0, x = 0;
136     while (g == 1) {
137         x = y;
138         for (int i = 0; i < r; i++)
139             y = (mulmod(y, y, n) + c) % n;
140         for (long long k = 0; k < r && g == 1; k += m) {
141             ys = y;
142             hi = std::min(m, r - k);
143             for (int j = 0; j < hi; j++) {
144                 y = (mulmod(y, y, n) + c) % n;
145                 q = mulmod(q, x > y ? x - y : y - x, n);
146             }
147             g = gcd(q, n);
148         }
149         r *= 2;
150     }
151     if (g == n) do {
152         ys = (mulmod(ys, ys, n) + c) % n;
153         g = gcd(x > ys ? x - ys : ys - x, n);
154     } while (g <= 1);
155     return g;
156 }
157
158 /*
159
160 Combines Brent's method with trial division to efficiently
161 generate the prime factorization of large integers.
162

```

```

163 Returns a vector of prime divisors that multiply to n.
164 e.g. prime_factorize(15435) => {3, 3, 5, 7, 7, 7}
165     because 3^2 * 5^1 * 7^3 = 15435
166
167 */
168
169 std::vector<long long> prime_factorize_big(long long n) {
170     if (n <= 0) return std::vector<long long>(0);
171     if (n == 1) return std::vector<long long>(1, 1);
172     std::vector<long long> res;
173     for (; n % 2 == 0; n /= 2) res.push_back(2);
174     for (; n % 3 == 0; n /= 3) res.push_back(3);
175     int mx = 1000000; //trial division for factors <= 1M
176     for (int i = 5, w = 2; i <= mx; i += w, w = 6 - w) {
177         for (; n % i == 0; n /= i) res.push_back(i);
178     }
179     for (long long p = 0, p1; n > mx; n /= p1) { //brent
180         for (p1 = n; p1 != p; p1 = brent(p)) p = p1;
181         res.push_back(p1);
182     }
183     if (n != 1) res.push_back(n);
184     sort(res.begin(), res.end());
185     return res;
186 }
187
188 /** Example Usage */
189
190 #include <cassert>
191 #include <iostream>
192 #include <ctime>
193 using namespace std;
194
195 template<class It> void print(It lo, It hi) {
196     while (lo != hi) cout << *(lo++) << " ";
197     cout << "\n";
198 }
199
200 template<class It> void printp(It lo, It hi) {
201     for (; lo != hi; ++lo)
202         cout << "(" << lo->first << ", " << lo->second << ") ";
203     cout << "\n";
204 }
205
206 int main() {
207     srand(time(0));
208
209     vector< pair<int, int> > v1 = prime_factorize(15435);
210     printp(v1.begin(), v1.end());
211
212     vector<int> v2 = get_all_divisors(28);
213     print(v2.begin(), v2.end());
214
215     long long n = 100000311*10000003711;
216     assert(fermat(n) == 100000311);
217
218     vector<long long> v3 = prime_factorize_big(n);
219     print(v3.begin(), v3.end());
220
221     return 0;

```

222 }

4.3.5 Euler's Totient Function

```

1  /*
2
3  Euler's totient function (or Euler's phi function) counts
4  the positive integers less than or equal to n that are
5  relatively prime to n. (These integers are sometimes
6  referred to as totatives of n.) Thus, phi(n) is the number
7  of integers k in the range [1, n] for which gcd(n, k) = 1.
8
9  E.g. if n = 9. Then gcd(9, 3) = gcd(9, 6) = 3 and gcd(9, 9)
10 = 9. The other six numbers in the range [1, 9], i.e. 1, 2,
11 4, 5, 7 and 8 are relatively prime to 9. Thus, phi(9) = 6.
12
13 */
14
15 #include <vector>
16
17 int phi(int n) {
18     int res = n;
19     for (int i = 2; i * i <= n; i++)
20         if (n % i == 0) {
21             while (n % i == 0) n /= i;
22             res -= res / i;
23         }
24     if (n > 1) res -= res / n;
25     return res;
26 }
27
28 std::vector<int> phi_table(int n) {
29     std::vector<int> res(n + 1);
30     for (int i = 1; i <= n; i++)
31         res[i] = i;
32     for (int i = 1; i <= n; i++)
33         for (int j = i + i; j <= n; j += i)
34             res[j] -= res[i];
35     return res;
36 }
37
38 /** Example Usage */
39
40 #include <cassert>
41 #include <iostream>
42 using namespace std;
43
44 int main() {
45     cout << phi(1) << "\n";          //1
46     cout << phi(9) << "\n";          //6
47     cout << phi(1234567) << "\n";    //1224720
48
49     int n = 1000;
50     vector<int> v = phi_table(n);
51     for (int i = 0; i <= n; i++)
52         assert(v[i] == phi(i));
53     return 0;

```

54 }

4.4 Arbitrary Precision Arithmetic

4.4.1 Big Integers (Simple)

```

1  /*
2
3  Description: Integer arbitrary precision functions.
4  To use, pass bigints to the functions by addresses.
5  e.g. add(&a, &b, &c) stores the sum of a and b into c.
6
7  Complexity: comp(), to_string(), digit_shift(), add(),
8  and sub() are O(N) on the number of digits. mul() and
9  div() are O(N^2). zero_justify() is amortized constant.
10
11 */
12
13 #include <string>
14
15 struct bigint {
16     static const int maxdigits = 1000;
17
18     char dig[maxdigits], sign;
19     int last;
20
21     bigint(long long x = 0): sign(x < 0 ? -1 : 1) {
22         for (int i = 0; i < maxdigits; i++) dig[i] = 0;
23         if (x == 0) { last = 0; return; }
24         if (x < 0) x = -x;
25         for (last = -1; x > 0; x /= 10) dig[++last] = x % 10;
26     }
27
28     bigint(const std::string & s): sign(s[0] == '-' ? -1 : 1) {
29         for (int i = 0; i < maxdigits; i++) dig[i] = 0;
30         last = -1;
31         for (int i = s.size() - 1; i >= 0; i--)
32             dig[++last] = (s[i] - '0');
33         if (dig[last] + '0' == '-') dig[last--] = 0;
34     }
35 };
36
37 void zero_justify(bigint * x) {
38     while (x->last > 0 && !x->dig[x->last]) x->last--;
39     if (x->last == 0 && x->dig[0] == 0) x->sign = 1;
40 }
41
42 void add(bigint * a, bigint * b, bigint * c);
43 void sub(bigint * a, bigint * b, bigint * c);
44
45 //returns: -1 if a < b, 0 if a == b, or 1 if a > b
46 int comp(bigint * a, bigint * b) {
47     if (a->sign != b->sign) return b->sign;
48     if (b->last > a->last) return a->sign;
49     if (a->last > b->last) return -a->sign;
50     for (int i = a->last; i >= 0; i--) {

```



```

51     if (a->dig[i] > b->dig[i]) return -a->sign;
52     if (b->dig[i] > a->dig[i]) return a->sign;
53 }
54 return 0;
55 }
56
57 void add(bigint * a, bigint * b, bigint * c) {
58     if (a->sign != b->sign) {
59         if (a->sign == -1)
60             a->sign = 1, sub(b, a, c), a->sign = -1;
61         else
62             b->sign = 1, sub(a, b, c), b->sign = -1;
63         return;
64     }
65     c->sign = a->sign;
66     c->last = (a->last > b->last ? a->last : b->last) + 1;
67     for (int i = 0, carry = 0; i <= c->last; i++) {
68         c->dig[i] = (carry + a->dig[i] + b->dig[i]) % 10;
69         carry = (carry + a->dig[i] + b->dig[i]) / 10;
70     }
71     zero_justify(c);
72 }
73
74 void sub(bigint * a, bigint * b, bigint * c) {
75     if (a->sign == -1 || b->sign == -1) {
76         b->sign *= -1, add(a, b, c), b->sign *= -1;
77         return;
78     }
79     if (comp(a, b) == 1) {
80         sub(b, a, c), c->sign = -1;
81         return;
82     }
83     c->last = (a->last > b->last) ? a->last : b->last;
84     for (int i = 0, borrow = 0, v; i <= c->last; i++) {
85         v = a->dig[i] - borrow;
86         if (i <= b->last) v -= b->dig[i];
87         if (a->dig[i] > 0) borrow = 0;
88         if (v < 0) v += 10, borrow = 1;
89         c->dig[i] = v % 10;
90     }
91     zero_justify(c);
92 }
93
94 void digit_shift(bigint * x, int n) {
95     if (!x->last && !x->dig[0]) return;
96     for (int i = x->last; i >= 0; i--)
97         x->dig[i + n] = x->dig[i];
98     for (int i = 0; i < n; i++) x->dig[i] = 0;
99     x->last += n;
100 }
101
102 void mul(bigint * a, bigint * b, bigint * c) {
103     bigint row = *a, tmp;
104     for (int i = 0; i <= b->last; i++) {
105         for (int j = 1; j <= b->dig[i]; j++) {
106             add(c, &row, &tmp);
107             *c = tmp;
108         }
109         digit_shift(&row, 1);

```

```

110     }
111     c->sign = a->sign * b->sign;
112     zero_justify(c);
113 }
114
115 void div(bigint * a, bigint * b, bigint * c) {
116     bigint row, tmp;
117     int asign = a->sign, bsign = b->sign;
118     a->sign = b->sign = 1;
119     c->last = a->last;
120     for (int i = a->last; i >= 0; i--) {
121         digit_shift(&row, 1);
122         row.dig[0] = a->dig[i];
123         c->dig[i] = 0;
124         for (; comp(&row, b) != 1; row = tmp) {
125             c->dig[i]++;
126             sub(&row, b, &tmp);
127         }
128     }
129     c->sign = (a->sign = asign) * (b->sign = bsign);
130     zero_justify(c);
131 }
132
133 std::string to_string(bigint * x) {
134     std::string s(x->sign == -1 ? "-" : "");
135     for (int i = x->last; i >= 0; i--)
136         s += (char)('0' + x->dig[i]);
137     return s;
138 }
139
140 /** Example Usage */
141
142 #include <cassert>
143
144 int main() {
145     bigint a("-9899819294989142124"), b("12398124981294214");
146     bigint sum; add(&a, &b, &sum);
147     bigint dif; sub(&a, &b, &dif);
148     bigint prd; mul(&a, &b, &prd);
149     bigint quo; div(&a, &b, &quo);
150     assert(to_string(&sum) == "-9887421170007847910");
151     assert(to_string(&dif) == "-9912217419970436338");
152     assert(to_string(&prd) == "-122739196911503356525379735104870536");
153     assert(to_string(&quo) == "-798");
154     return 0;
155 }

```

4.4.2 Big Integer and Rational Class

```

1  /*
2
3  The following bigint class is implemented by storing "chunks"
4  of the big integer in a large base that is a power of 10 so
5  it can be efficiently stored, operated on, and printed.
6
7  It has extensive features including karatsuba multiplication,
8  exponentiation by squaring, and n-th root using binary search.

```

```

9  The class is thoroughly templated, so you can use it as
10 easily as you do for normal ints. For example, you may use
11 operators with a bigint and a string (e.g. bigint(1234)+"-567"
12 and the result will be correctly promoted to a bigint that has
13 a value of 667). I/O is done using <iostream>. For example:
14     bigint a, b; cin >> a >> b; cout << a + b << "\n";
15 adds two integers together and prints the result, just as you
16 would expect for a normal int, except with arbitrary precision.
17 The class also supports other streams such as fstream.
18
19 After the bigint class, a class for rational numbers is
20 implemented, using two bigints to store its numerators and
21 denominators. It is useful for when exact results of division
22 operations are needed.
23
24 */
25
26 #include <algorithm> /* std::max(), std::swap() */
27 #include <cmath>     /* sqrt() */
28 #include <cstdlib>   /* rand() */
29 #include <iomanip>    /* std::setw(), std::setfill() */
30 #include <istream>
31 #include <ostream>
32 #include <sstream>
33 #include <stdexcept> /* std::runtime_error() */
34 #include <string>
35 #include <utility>   /* std::pair */
36 #include <vector>
37
38 struct bigint {
39     //base should be a power of 10 for I/O to work
40     //base and base_digits should be consistent
41     static const int base = 1000000000, base_digits = 9;
42
43     typedef std::vector<int> vint;
44     typedef std::vector<long long> vll;
45
46     vint a; //a[0] stores right-most (least significant) base-digit
47     int sign;
48
49     bigint() : sign(1) {}
50     bigint(int v) { *this = (long long)v; }
51     bigint(long long v) { *this = v; }
52     bigint(const std::string & s) { read(s); }
53     bigint(const char * s) { read(std::string(s)); }
54
55     void trim() {
56         while (!a.empty() && a.back() == 0) a.pop_back();
57         if (a.empty()) sign = 1;
58     }
59
60     void read(const std::string & s) {
61         sign = 1;
62         a.clear();
63         int pos = 0;
64         while (pos < (int)s.size() && (s[pos] == '-' || s[pos] == '+')) {
65             if (s[pos] == '-') sign = -sign;
66             pos++;
67         }

```

```

68     for (int i = s.size() - 1; i >= pos; i -= base_digits) {
69         int x = 0;
70         for (int j = std::max(pos, i - base_digits + 1); j <= i; j++)
71             x = x * 10 + s[j] - '0';
72         a.push_back(x);
73     }
74     trim();
75 }
76
77 void operator = (const bigint & v) {
78     sign = v.sign;
79     a = v.a;
80 }
81
82 void operator = (long long v) {
83     sign = 1;
84     if (v < 0) sign = -1, v = -v;
85     a.clear();
86     for (; v > 0; v /= base) a.push_back(v % base);
87 }
88
89 bigint operator + (const bigint & v) const {
90     if (sign == v.sign) {
91         bigint res = v;
92         int carry = 0;
93         for (int i = 0; i < (int)std::max(a.size(), v.a.size()) || carry; i++) {
94             if (i == (int)res.a.size()) res.a.push_back(0);
95             res.a[i] += carry + (i < (int)a.size() ? a[i] : 0);
96             carry = res.a[i] >= base;
97             if (carry) res.a[i] -= base;
98         }
99         return res;
100     }
101     return *this - (-v);
102 }
103
104 bigint operator - (const bigint & v) const {
105     if (sign == v.sign) {
106         if (abs() >= v.abs()) {
107             bigint res(*this);
108             for (int i = 0, carry = 0; i < (int)v.a.size() || carry; i++) {
109                 res.a[i] -= carry + (i < (int)v.a.size() ? v.a[i] : 0);
110                 carry = res.a[i] < 0;
111                 if (carry) res.a[i] += base;
112             }
113             res.trim();
114             return res;
115         }
116         return -(v - *this);
117     }
118     return *this + (-v);
119 }
120
121 void operator *= (int v) {
122     if (v < 0) sign = -sign, v = -v;
123     for (int i = 0, carry = 0; i < (int)a.size() || carry; i++) {
124         if (i == (int)a.size()) a.push_back(0);
125         long long cur = a[i] * (long long)v + carry;
126         carry = (int)(cur / base);

```

```

127     a[i] = (int)(cur % base);
128     //asm("divl %%ecx" : "=a"(carry), "=d"(a[i]) : "A"(cur), "c"(base));
129 }
130 trim();
131 }
132
133 bigint operator * (int v) const {
134     bigint res(*this);
135     res *= v;
136     return res;
137 }
138
139 static vint convert_base(const vint & a, int l1, int l2) {
140     vll p(std::max(l1, l2) + 1);
141     p[0] = 1;
142     for (int i = 1; i < (int)p.size(); i++) p[i] = p[i - 1] * 10;
143     vint res;
144     long long cur = 0;
145     for (int i = 0, cur_digits = 0; i < (int)a.size(); i++) {
146         cur += a[i] * p[cur_digits];
147         cur_digits += l1;
148         while (cur_digits >= l2) {
149             res.push_back((int)(cur % p[l2]));
150             cur /= p[l2];
151             cur_digits -= l2;
152         }
153     }
154     res.push_back((int)cur);
155     while (!res.empty() && res.back() == 0) res.pop_back();
156     return res;
157 }
158
159 //complexity:  $O(3N^{\log_2(3)}) \sim O(3N^{1.585})$ 
160 static vll karatsuba_multiply(const vll & a, const vll & b) {
161     int n = a.size();
162     vll res(n + n);
163     if (n <= 32) {
164         for (int i = 0; i < n; i++)
165             for (int j = 0; j < n; j++)
166                 res[i + j] += a[i] * b[j];
167         return res;
168     }
169     int k = n >> 1;
170     vll a1(a.begin(), a.begin() + k), a2(a.begin() + k, a.end());
171     vll b1(b.begin(), b.begin() + k), b2(b.begin() + k, b.end());
172     vll a1b1 = karatsuba_multiply(a1, b1);
173     vll a2b2 = karatsuba_multiply(a2, b2);
174     for (int i = 0; i < k; i++) a2[i] += a1[i];
175     for (int i = 0; i < k; i++) b2[i] += b1[i];
176     vll r = karatsuba_multiply(a2, b2);
177     for (int i = 0; i < (int)a1b1.size(); i++) r[i] -= a1b1[i];
178     for (int i = 0; i < (int)a2b2.size(); i++) r[i] -= a2b2[i];
179     for (int i = 0; i < (int)r.size(); i++) res[i + k] += r[i];
180     for (int i = 0; i < (int)a1b1.size(); i++) res[i] += a1b1[i];
181     for (int i = 0; i < (int)a2b2.size(); i++) res[i + n] += a2b2[i];
182     return res;
183 }
184
185 bigint operator * (const bigint & v) const {

```

```

186 //if really big values cause overflow, use smaller _base
187 static const int _base = 10000, _base_digits = 4;
188 vint _a = convert_base(this->a, base_digits, _base_digits);
189 vint _b = convert_base(v.a, base_digits, _base_digits);
190 vll a(_a.begin(), _a.end());
191 vll b(_b.begin(), _b.end());
192 while (a.size() < b.size()) a.push_back(0);
193 while (b.size() < a.size()) b.push_back(0);
194 while (a.size() & (a.size() - 1)) {
195     a.push_back(0);
196     b.push_back(0);
197 }
198 vll c = karatsuba_multiply(a, b);
199 bigint res;
200 res.sign = sign * v.sign;
201 for (int i = 0, carry = 0; i < (int)c.size(); i++) {
202     long long cur = c[i] + carry;
203     res.a.push_back((int)(cur % _base));
204     carry = (int)(cur / _base);
205 }
206 res.a = convert_base(res.a, _base_digits, base_digits);
207 res.trim();
208 return res;
209 }
210
211 bigint operator ^ (const bigint & v) const {
212     if (v.sign == -1) return bigint(0);
213     bigint x(*this), n(v), res(1);
214     while (!n.is_zero()) {
215         if (n.a[0] % 2 == 1) res *= x;
216         x *= x;
217         n /= 2;
218     }
219     return res;
220 }
221
222 friend std::pair<bigint, bigint> divmod(const bigint & a1, const bigint & b1) {
223     int norm = base / (b1.a.back() + 1);
224     bigint a = a1.abs() * norm;
225     bigint b = b1.abs() * norm;
226     bigint q, r;
227     q.a.resize(a.a.size());
228     for (int i = a.a.size() - 1; i >= 0; i--) {
229         r *= base;
230         r += a.a[i];
231         int s1 = r.a.size() <= b.a.size() ? 0 : r.a[b.a.size()];
232         int s2 = r.a.size() <= b.a.size() - 1 ? 0 : r.a[b.a.size() - 1];
233         int d = ((long long)base * s1 + s2) / b.a.back();
234         for (r -= b * d; r < 0; r += b) d--;
235         q.a[i] = d;
236     }
237     q.sign = a1.sign * b1.sign;
238     r.sign = a1.sign;
239     q.trim();
240     r.trim();
241     return std::make_pair(q, r / norm);
242 }
243
244 bigint operator / (const bigint & v) const { return divmod(*this, v).first; }

```

```

245     bigint operator % (const bigint & v) const { return divmod(*this, v).second; }
246
247     bigint & operator /= (int v) {
248         if (v < 0) sign = -sign, v = -v;
249         for (int i = a.size() - 1, rem = 0; i >= 0; i--) {
250             long long cur = a[i] + rem * (long long)base;
251             a[i] = (int)(cur / v);
252             rem = (int)(cur % v);
253         }
254         trim();
255         return *this;
256     }
257
258     bigint operator / (int v) const {
259         bigint res(*this);
260         res /= v;
261         return res;
262     }
263
264     int operator % (int v) const {
265         if (v < 0) v = -v;
266         int m = 0;
267         for (int i = a.size() - 1; i >= 0; i--)
268             m = (a[i] + m * (long long)base) % v;
269         return m * sign;
270     }
271
272     bigint operator ++(int) { bigint t(*this); operator++(); return t; }
273     bigint operator --(int) { bigint t(*this); operator--(); return t; }
274     bigint & operator ++() { *this = *this + bigint(1); return *this; }
275     bigint & operator --() { *this = *this - bigint(1); return *this; }
276     bigint & operator += (const bigint & v) { *this = *this + v; return *this; }
277     bigint & operator -= (const bigint & v) { *this = *this - v; return *this; }
278     bigint & operator *= (const bigint & v) { *this = *this * v; return *this; }
279     bigint & operator /= (const bigint & v) { *this = *this / v; return *this; }
280     bigint & operator %= (const bigint & v) { *this = *this % v; return *this; }
281     bigint & operator ^= (const bigint & v) { *this = *this ^ v; return *this; }
282
283     bool operator < (const bigint & v) const {
284         if (sign != v.sign) return sign < v.sign;
285         if (a.size() != v.a.size())
286             return a.size() * sign < v.a.size() * v.sign;
287         for (int i = a.size() - 1; i >= 0; i--)
288             if (a[i] != v.a[i])
289                 return a[i] * sign < v.a[i] * v.sign;
290         return false;
291     }
292
293     bool operator > (const bigint & v) const { return v < *this; }
294     bool operator <= (const bigint & v) const { return !(v < *this); }
295     bool operator >= (const bigint & v) const { return !(*this < v); }
296     bool operator == (const bigint & v) const { return !(*this < v) && !(v < *this); }
297     bool operator != (const bigint & v) const { return *this < v || v < *this; }
298
299     int size() const {
300         if (a.empty()) return 1;
301         std::ostringstream oss;
302         oss << a.back();
303         return oss.str().length() + base_digits*(a.size() - 1);

```

```

304 }
305
306 bool is_zero() const {
307     return a.empty() || (a.size() == 1 && !a[0]);
308 }
309
310 bigint operator - () const {
311     bigint res(*this);
312     res.sign = -sign;
313     return res;
314 }
315
316 bigint abs() const {
317     bigint res(*this);
318     res.sign *= res.sign;
319     return res;
320 }
321
322 friend bigint abs(const bigint & a) {
323     return a.abs();
324 }
325
326 friend bigint gcd(const bigint & a, const bigint & b) {
327     return b.is_zero() ? a : gcd(b, a % b);
328 }
329
330 friend bigint lcm(const bigint & a, const bigint & b) {
331     return a / gcd(a, b) * b;
332 }
333
334 friend bigint sqrt(const bigint & x) {
335     bigint a = x;
336     while (a.a.empty() || a.a.size() % 2 == 1) a.a.push_back(0);
337     int n = a.a.size();
338     int firstdig = sqrt((double)a.a[n - 1] * base + a.a[n - 2]);
339     int norm = base / (firstdig + 1);
340     a *= norm;
341     a *= norm;
342     while (a.a.empty() || a.a.size() % 2 == 1) a.a.push_back(0);
343     bigint r = (long long)a.a[n - 1] * base + a.a[n - 2];
344     firstdig = sqrt((double)a.a[n - 1] * base + a.a[n - 2]);
345     int q = firstdig;
346     bigint res;
347     for (int j = n / 2 - 1; j >= 0; j--) {
348         for (; q-- > 0) {
349             bigint r1 = (r - (res * 2 * base + q) * q) * base * base + (j > 0 ?
350                 (long long)a.a[2 * j - 1] * base + a.a[2 * j - 2] : 0);
351             if (r1 >= 0) {
352                 r = r1;
353                 break;
354             }
355         }
356         res = (res * base) + q;
357         if (j > 0) {
358             int d1 = res.a.size() + 2 < r.a.size() ? r.a[res.a.size() + 2] : 0;
359             int d2 = res.a.size() + 1 < r.a.size() ? r.a[res.a.size() + 1] : 0;
360             int d3 = res.a.size() < r.a.size() ? r.a[res.a.size()] : 0;
361             q = ((long long)d1*base*base + (long long)d2*base + d3)/(firstdig * 2);
362         }

```



```

363     }
364     res.trim();
365     return res / norm;
366 }
367
368 friend bigint nthroot(const bigint & x, const bigint & n) {
369     bigint hi = 1;
370     while ((hi ^ n) <= x) hi *= 2;
371     bigint lo = hi / 2, mid, midn;
372     while (lo < hi) {
373         mid = (lo + hi) / 2;
374         midn = mid ^ n;
375         if (lo < mid && midn < x) {
376             lo = mid;
377         } else if (mid < hi && x < midn) {
378             hi = mid;
379         } else {
380             return mid;
381         }
382     }
383     return mid + 1;
384 }
385
386 friend std::istream & operator >> (std::istream & in, bigint & v) {
387     std::string s;
388     in >> s;
389     v.read(s);
390     return in;
391 }
392
393 friend std::ostream & operator << (std::ostream & out, const bigint & v) {
394     if (v.sign == -1) out << '-';
395     out << (v.a.empty() ? 0 : v.a.back());
396     for (int i = v.a.size() - 2; i >= 0; i--)
397         out << std::setw(base_digits) << std::setfill('0') << v.a[i];
398     return out;
399 }
400
401 std::string to_string() const {
402     std::ostringstream oss;
403     if (sign == -1) oss << '-';
404     oss << (a.empty() ? 0 : a.back());
405     for (int i = a.size() - 2; i >= 0; i--)
406         oss << std::setw(base_digits) << std::setfill('0') << a[i];
407     return oss.str();
408 }
409
410 long long to_llong() const {
411     long long res = 0;
412     for (int i = a.size() - 1; i >= 0; i--)
413         res = res * base + a[i];
414     return res * sign;
415 }
416
417 double to_double() const {
418     std::stringstream ss(to_string());
419     double res;
420     ss >> res;
421     return res;

```

```

422 }
423
424 long double to_ldouble() const {
425     std::stringstream ss(to_string());
426     long double res;
427     ss >> res;
428     return res;
429 }
430
431 static bigint rand(int len) {
432     if (len == 0) return bigint(0);
433     std::string s(1, '1' + (::rand() % 9));
434     for (int i = 1; i < len; i++) s += '0' + (::rand() % 10);
435     return bigint(s);
436 }
437 };
438
439 template<class T> bool operator > (const T & a, const bigint & b) { return bigint(a) > b; }
440 template<class T> bool operator < (const T & a, const bigint & b) { return bigint(a) < b; }
441 template<class T> bool operator >= (const T & a, const bigint & b) { return bigint(a) >= b; }
442 template<class T> bool operator <= (const T & a, const bigint & b) { return bigint(a) <= b; }
443 template<class T> bool operator == (const T & a, const bigint & b) { return bigint(a) == b; }
444 template<class T> bool operator != (const T & a, const bigint & b) { return bigint(a) != b; }
445 template<class T> bigint operator + (const T & a, const bigint & b) { return bigint(a) + b; }
446 template<class T> bigint operator - (const T & a, const bigint & b) { return bigint(a) - b; }
447 template<class T> bigint operator ^ (const T & a, const bigint & b) { return bigint(a) ^ b; }
448
449 /*
450
451 Exclude *, /, and % to force a user decision between int and bigint algorithms
452
453 bigint operator * (bigint a, bigint b) vs. bigint operator * (bigint a, int b)
454 bigint operator / (bigint a, bigint b) vs. bigint operator / (bigint a, int b)
455 bigint operator % (bigint a, bigint b) vs. int operator % (bigint a, int b)
456
457 */
458
459 struct rational {
460     bigint num, den;
461
462     rational(): num(0), den(1) {}
463     rational(long long n): num(n), den(1) {}
464     rational(const bigint & n) : num(n), den(1) {}
465
466     template<class T1, class T2>
467     rational(const T1 & n, const T2 & d): num(n), den(d) {
468         if (den == 0)
469             throw std::runtime_error("Rational_division_by_zero.");
470         if (den < 0) {
471             num = -num;
472             den = -den;
473         }
474         bigint a(num < 0 ? -num : num), b(den), tmp;
475         while (a != 0 && b != 0) {
476             tmp = a % b;
477             a = b;
478             b = tmp;
479         }
480         bigint gcd = (b == 0) ? a : b;

```

```

481     num /= gcd;
482     den /= gcd;
483 }
484
485 bool operator < (const rational & r) const {
486     return num * r.den < r.num * den;
487 }
488
489 bool operator > (const rational & r) const {
490     return r.num * den < num * r.den;
491 }
492
493 bool operator <= (const rational & r) const {
494     return !(r < *this);
495 }
496
497 bool operator >= (const rational & r) const {
498     return !(*this < r);
499 }
500
501 bool operator == (const rational & r) const {
502     return num == r.num && den == r.den;
503 }
504
505 bool operator != (const rational & r) const {
506     return num != r.num || den != r.den;
507 }
508
509 rational operator + (const rational & r) const {
510     return rational(num * r.den + r.num * den, den * r.den);
511 }
512
513 rational operator - (const rational & r) const {
514     return rational(num * r.den - r.num * den, r.den * den);
515 }
516
517 rational operator * (const rational & r) const {
518     return rational(num * r.num, r.den * den);
519 }
520
521 rational operator / (const rational & r) const {
522     return rational(num * r.den, den * r.num);
523 }
524
525 rational operator % (const rational & r) const {
526     return *this - r * rational(num * r.den / (r.num * den), 1);
527 }
528
529 rational operator ^ (const bigint & p) const {
530     return rational(num ^ p, den ^ p);
531 }
532
533 rational operator ++(int) { rational t(*this); operator++(); return t; }
534 rational operator --(int) { rational t(*this); operator--(); return t; }
535 rational & operator ++() { *this = *this + 1; return *this; }
536 rational & operator --() { *this = *this - 1; return *this; }
537 rational & operator += (const rational & r) { *this = *this + r; return *this; }
538 rational & operator -= (const rational & r) { *this = *this - r; return *this; }
539 rational & operator *= (const rational & r) { *this = *this * r; return *this; }

```

```

540 rational & operator /= (const rational & r) { *this = *this / r; return *this; }
541 rational & operator %= (const rational & r) { *this = *this % r; return *this; }
542 rational & operator ^= (const bigint & r) { *this = *this ^ r; return *this; }
543
544 rational operator - () const {
545     return rational(-num, den);
546 }
547
548 rational abs() const {
549     return rational(num.abs(), den);
550 }
551
552 long long to_llong() const {
553     return num.to_llong() / den.to_llong();
554 }
555
556 double to_double() const {
557     return num.to_double() / den.to_double();
558 }
559
560 friend rational abs(const rational & r) {
561     return rational(r.num.abs(), r.den);
562 }
563
564 friend std::istream & operator >> (std::istream & in, rational & r) {
565     std::string s;
566     in >> r.num;
567     r.den = 1;
568     return in;
569 }
570
571 friend std::ostream & operator << (std::ostream & out, const rational & r) {
572     out << r.num << "/" << r.den;
573     return out;
574 }
575
576 //rational in range [0, 1] with precision no greater than prec
577 static rational rand(int prec) {
578     rational r(bigint::rand(prec), bigint::rand(prec));
579     if (r.num > r.den) std::swap(r.num, r.den);
580     return r;
581 }
582 };
583
584 template<class T> bool operator > (const T & a, const rational & b) { return rational(a) > b; }
585 template<class T> bool operator < (const T & a, const rational & b) { return rational(a) < b; }
586 template<class T> bool operator >= (const T & a, const rational & b) { return rational(a) >= b; }
587 template<class T> bool operator <= (const T & a, const rational & b) { return rational(a) <= b; }
588 template<class T> bool operator == (const T & a, const rational & b) { return rational(a) == b; }
589 template<class T> bool operator != (const T & a, const rational & b) { return rational(a) != b; }
590 template<class T> rational operator + (const T & a, const rational & b) { return rational(a) + b; }
591 template<class T> rational operator - (const T & a, const rational & b) { return rational(a) - b; }
592 template<class T> rational operator * (const T & a, const rational & b) { return rational(a) * b; }
593 template<class T> rational operator / (const T & a, const rational & b) { return rational(a) / b; }
594 template<class T> rational operator % (const T & a, const rational & b) { return rational(a) % b; }
595 template<class T> rational operator ^ (const T & a, const rational & b) { return rational(a) ^ b; }
596
597 /** Example Usage */
598

```

```

599 #include <cassert>
600 #include <cstdio>
601 #include <ctime>
602 #include <iostream>
603 using namespace std;
604
605 int main() {
606     for (int i = 0; i < 20; i++) {
607         int n = rand() % 100 + 1;
608         bigint a = bigint::rand(n);
609         bigint res = sqrt(a);
610         bigint xx(res * res);
611         bigint yy(res + 1);
612         yy *= yy;
613         assert(xx <= a && yy > a);
614         int m = rand() % n + 1;
615         bigint b = bigint::rand(m) + 1;
616         res = a / b;
617         xx = res * b;
618         yy = b * (res + 1);
619         assert(a >= xx && a < yy);
620     }
621
622     assert("995291497" ==
623         nthroot(bigint("981298591892498189249182998429898124"), 4));
624
625     bigint x(5);
626     x = -6;
627     assert(x.to_llong() == -6ll);
628     assert(x.to_string() == "-6");
629
630     clock_t start;
631
632     start = clock();
633     bigint c = bigint::rand(10000) / bigint::rand(2000);
634     cout << "Div took " << (float)(clock() - start)/CLOCKS_PER_SEC << "s\n";
635
636     start = clock();
637     assert((20^bigint(12345)).size() == 16062);
638     cout << "Pow took " << (float)(clock() - start)/CLOCKS_PER_SEC << "s\n";
639
640     int mn = -21, dd = 2;
641     rational n(mn, 1), d(dd);
642     cout << (nn % dd) << "\n";
643     cout << (n % d) << "\n";
644     cout << fmod(-5.3, -1.7) << "\n";
645     cout << rational(-53, 10) % rational(-17, 10) << "\n";
646     cout << rational(-53, 10).abs() << "\n";
647     cout << (rational(-53, 10) ^ 20) << "\n";
648     cout << rational::rand(20) << "\n";
649     return 0;
650 }

```

4.4.3 FFT and Multiplication

```

1  /*
2

```

```

3  A discrete Fourier transform (DFT) converts a list of equally
4  spaced samples of a function into the list of coefficients of
5  a finite combination of complex sinusoids, ordered by their
6  frequencies, that has those same sample values. A Fast Fourier
7  Transform (FFT) rapidly computes the DFT by factorizing the
8  DFT matrix into a product of sparse (mostly zero) factors.
9  The FFT can be used to solve problems such as efficiently
10 multiplying big integers or polynomials
11
12 The fft() function below is a generic function that will
13 work well in many applications beyond just multiplying
14 big integers. While Karatsuba multiplication is  $\sim O(n^{1.58})$ ,
15 the complexity of the fft multiplication is only  $O(n \log n)$ .
16
17 Note that mul(string, string) in the following implementation
18 only works for strings of strictly digits from '0' to '9'.
19 It is also easy to adapt this for the bigint class in the
20 previous section. Simply replace the old bigint operator *
21 definition with the following modified version of mul():
22
23 bigint operator * (const bigint & v) const {
24     static const int _base = 10000, _base_digits = 4;
25     vint _a = convert_base(this->a, base_digits, _base_digits);
26     vint _b = convert_base(v.a, base_digits, _base_digits);
27     int len = 32 - __builtin_clz(std::max(_a.size(), _b.size()) - 1);
28     len = 1 << (len + 1);
29     vcd a(len), b(len);
30     for (int i = 0; i < _a.size(); i++) a[i] = cd(_a[i], 0);
31     for (int i = 0; i < _b.size(); i++) b[i] = cd(_b[i], 0);
32     a = fft(a);
33     b = fft(b);
34     for (int i = 0; i < len; i++) {
35         double real = a[i].real() * b[i].real() - a[i].imag() * b[i].imag();
36         a[i].imag() = a[i].imag() * b[i].real() + b[i].imag() * a[i].real();
37         a[i].real() = real;
38     }
39     a = fft(a, true);
40     vll c(len);
41     for (int i = 0; i < len; i++) c[i] = (long long)(a[i].real() + 0.5);
42     bigint res;
43     res.sign = sign * v.sign;
44     for (int i = 0, carry = 0; i < c.size(); i++) {
45         long long cur = c[i] + carry;
46         res.a.push_back((int)(cur % _base));
47         carry = (int)(cur / _base);
48     }
49     res.a = convert_base(res.a, _base_digits, base_digits);
50     res.trim();
51     return res;
52 }
53
54 */
55
56 #include <algorithm> /* std::max(), std::reverse() */
57 #include <cmath> /* M_PI, cos(), sin() */
58 #include <complex>
59 #include <iomanip> /* std::setw(), std::setfill() */
60 #include <sstream>
61 #include <string>

```

```

62 #include <vector>
63
64 typedef std::complex<double> cd;
65 typedef std::vector<cd> vcd;
66
67 vcd fft(const vcd & v, bool inverse = false) {
68     static const double PI = acos(-1.0);
69     int n = v.size(), k = 0, high1 = -1;
70     while ((1 << k) < n) k++;
71     std::vector<int> rev(n);
72     rev[0] = 0;
73     for (int i = 1; i < n; i++) {
74         if ((i & (i - 1)) == 0) high1++;
75         rev[i] = rev[i ^ (1 << high1)];
76         rev[i] |= (1 << (k - high1 - 1));
77     }
78     vcd roots(n), res(n);
79     for (int i = 0; i < n; i++) {
80         double alpha = 2 * PI * i / n;
81         roots[i] = cd(cos(alpha), sin(alpha));
82     }
83     for (int i = 0; i < n; i++) res[i] = v[rev[i]];
84     for (int len = 1; len < n; len <= 1) {
85         vcd tmp(n);
86         int rstep = roots.size() / (len * 2);
87         for (int pdest = 0; pdest < n; pdest++) {
88             int p1 = pdest;
89             for (int i = 0; i < len; i++) {
90                 cd val = roots[i * rstep] * res[p1 + len];
91                 tmp[pdest] = res[p1] + val;
92                 tmp[pdest + len] = res[p1] - val;
93                 pdest++, p1++;
94             }
95             pdest += len;
96         }
97         res.swap(tmp);
98     }
99     if (inverse) {
100         for (int i = 0; i < (int)res.size(); i++) res[i] /= v.size();
101         std::reverse(res.begin() + 1, res.end());
102     }
103     return res;
104 }
105
106 typedef std::vector<long long> vll;
107
108 vll mul(const vll & va, const vll & vb) {
109     int len = 32 - __builtin_clz(std::max(va.size(), vb.size()) - 1);
110     len = 1 << (len + 1);
111     vcd a(len), b(len);
112     for (int i = 0; i < (int)va.size(); i++) a[i] = cd(va[i], 0);
113     for (int i = 0; i < (int)vb.size(); i++) b[i] = cd(vb[i], 0);
114     a = fft(a);
115     b = fft(b);
116     for (int i = 0; i < len; i++) {
117         double real = a[i].real() * b[i].real() - a[i].imag() * b[i].imag();
118         a[i].imag() = a[i].imag() * b[i].real() + b[i].imag() * a[i].real();
119         a[i].real() = real;
120     }

```

```

121     a = fft(a, true);
122     vll res(len);
123     for (int i = 0; i < len; i++) res[i] = (long long)(a[i].real() + 0.5);
124     return res;
125 }
126
127 const int base = 10000, base_digits = 4;
128
129 std::string mul(const std::string & as, const std::string & bs) {
130     vll a, b;
131     for (int i = as.size() - 1; i >= 0; i -= base_digits) {
132         int x = 0;
133         for (int j = std::max(0, i - base_digits + 1); j <= i; j++)
134             x = x * 10 + as[j] - '0';
135         a.push_back(x);
136     }
137     for (int i = bs.size() - 1; i >= 0; i -= base_digits) {
138         int x = 0;
139         for (int j = std::max(0, i - base_digits + 1); j <= i; j++)
140             x = x * 10 + bs[j] - '0';
141         b.push_back(x);
142     }
143     vll c = mul(a, b);
144     long long carry = 0;
145     for (int i = 0; i < (int)c.size(); i++) {
146         c[i] += carry;
147         carry = c[i] / base;
148         c[i] %= base;
149     }
150     while (c.back() == 0) c.pop_back();
151     if (c.empty()) c.push_back(0);
152     std::ostringstream oss;
153     oss << (c.empty() ? 0 : c.back());
154     for (int i = c.size() - 2; i >= 0; i--)
155         oss << std::setw(base_digits) << std::setfill('0') << c[i];
156     return oss.str();
157 }
158
159 /** Example Usage */
160
161 #include <cassert>
162
163 int main() {
164     assert(mul("98904189", "244212") == "24153589804068");
165     return 0;
166 }

```

4.5 Linear Algebra

4.5.1 Matrix Class

```

1  /*
2
3  Basic matrix class with support for arithmetic operations
4  as well as matrix multiplication and exponentiation. You
5  can access/modify indices using m(r, c) or m[r][c]. You

```



```

6  can also treat it as a 2d vector, since the cast operator
7  to a reference to its internal 2d vector is defined. This
8  makes it compatible with the 2d vector functions such as
9  det() and lu_decompose() in later sections.
10
11  */
12
13  #include <ostream>
14  #include <stdexcept> /* std::runtime_error() */
15  #include <vector>
16
17  template<class val_t> class matrix {
18      int r, c;
19      std::vector<std::vector<val_t> > mat;
20
21  public:
22      matrix(int rows, int cols, val_t init = val_t()) {
23          r = rows;
24          c = cols;
25          mat.resize(r, std::vector<val_t>(c, init));
26      }
27
28      matrix(const std::vector<std::vector<val_t> > & m) {
29          r = m.size();
30          c = m[0].size();
31          mat = m;
32          mat.resize(r, std::vector<val_t>(c));
33      }
34
35      template<size_t rows, size_t cols>
36      matrix(val_t (&init)[rows][cols]) {
37          r = rows;
38          c = cols;
39          mat.resize(r, std::vector<val_t>(c));
40          for (int i = 0; i < r; i++)
41              for (int j = 0; j < c; j++)
42                  mat[i][j] = init[i][j];
43      }
44
45      operator std::vector<std::vector<val_t> > &() { return mat; }
46      val_t & operator() (int r, int c) { return mat[r][c]; }
47      std::vector<val_t> & operator[] (int r) { return mat[r]; }
48      val_t at(int r, int c) const { return mat[r][c]; }
49      int rows() const { return r; }
50      int cols() const { return c; }
51
52      friend bool operator < (const matrix & a, const matrix & b) { return a.mat < b.mat; }
53      friend bool operator > (const matrix & a, const matrix & b) { return a.mat > b.mat; }
54      friend bool operator <= (const matrix & a, const matrix & b) { return a.mat <= b.mat; }
55      friend bool operator >= (const matrix & a, const matrix & b) { return a.mat >= b.mat; }
56      friend bool operator == (const matrix & a, const matrix & b) { return a.mat == b.mat; }
57      friend bool operator != (const matrix & a, const matrix & b) { return a.mat != b.mat; }
58
59      friend matrix operator + (const matrix & a, const matrix & b) {
60          if (a.r != b.r || a.c != b.c)
61              throw std::runtime_error("Matrix_dimensions_don't_match.");
62          matrix res(a);
63          for (int i = 0; i < res.r; i++)
64              for (int j = 0; j < res.c; j++)

```

```

65         res.mat[i][j] += b.mat[i][j];
66     return res;
67 }
68
69 friend matrix operator - (const matrix & a, const matrix & b) {
70     if (a.r != b.r || a.c != b.c)
71         throw std::runtime_error("Matrix_dimensions_don't_match.");
72     matrix res(a);
73     for (int i = 0; i < a.r; i++)
74         for (int j = 0; j < a.c; j++)
75             res.mat[i][j] -= b.mat[i][j];
76     return res;
77 }
78
79 friend matrix operator * (const matrix & a, const matrix & b) {
80     if (a.c != b.r)
81         throw std::runtime_error("#_of_a_cols_must_equal_#_of_b_rows.");
82     matrix res(a.r, b.c, 0);
83     for (int i = 0; i < a.r; i++)
84         for (int j = 0; j < b.c; j++)
85             for (int k = 0; k < a.c; k++)
86                 res.mat[i][j] += a.mat[i][k] * b.mat[k][j];
87     return res;
88 }
89
90 friend matrix operator + (const matrix & a, const val_t & v) {
91     matrix res(a);
92     for (int i = 0; i < a.r; i++)
93         for (int j = 0; j < a.c; j++) res.mat[i][j] += v;
94     return res;
95 }
96
97 friend matrix operator - (const matrix & a, const val_t & v) {
98     matrix res(a);
99     for (int i = 0; i < a.r; i++)
100         for (int j = 0; j < a.c; j++) res.mat[i][j] -= v;
101     return res;
102 }
103
104 friend matrix operator * (const matrix & a, const val_t & v) {
105     matrix res(a);
106     for (int i = 0; i < a.r; i++)
107         for (int j = 0; j < a.c; j++) res.mat[i][j] *= v;
108     return res;
109 }
110
111 friend matrix operator / (const matrix & a, const val_t & v) {
112     matrix res(a);
113     for (int i = 0; i < a.r; i++)
114         for (int j = 0; j < a.c; j++)
115             res.mat[i][j] /= v;
116     return res;
117 }
118
119 //raise matrix to the n-th power. precondition: a must be a square matrix
120 friend matrix operator ^ (const matrix & a, unsigned int n) {
121     if (a.r != a.c)
122         throw std::runtime_error("Matrix_must_be_square_for_exponentiation.");
123     if (n == 0) return identity_matrix(a.r);

```

```

124     if (n % 2 == 0) return (a * a) ^ (n / 2);
125     return a * (a ^ (n - 1));
126 }
127
128 //returns a^1 + a^2 + ... + a^n
129 friend matrix powsum(const matrix & a, unsigned int n) {
130     if (n == 0) return matrix(a.r, a.r);
131     if (n % 2 == 0)
132         return powsum(a, n / 2) * (identity_matrix(a.r) + (a ^ (n / 2)));
133     return a + a * powsum(a, n - 1);
134 }
135
136 matrix & operator += (const matrix & m) { *this = *this + m; return *this; }
137 matrix & operator -= (const matrix & m) { *this = *this - m; return *this; }
138 matrix & operator *= (const matrix & m) { *this = *this * m; return *this; }
139 matrix & operator += (const val_t & v) { *this = *this + v; return *this; }
140 matrix & operator -= (const val_t & v) { *this = *this - v; return *this; }
141 matrix & operator *= (const val_t & v) { *this = *this * v; return *this; }
142 matrix & operator /= (const val_t & v) { *this = *this / v; return *this; }
143 matrix & operator ^= (unsigned int n) { *this = *this ^ n; return *this; }
144
145 static matrix identity_matrix(int n) {
146     matrix res(n, n);
147     for (int i = 0; i < n; i++) res[i][i] = 1;
148     return res;
149 }
150
151 friend std::ostream & operator << (std::ostream & out, const matrix & m) {
152     out << "[";
153     for (int i = 0; i < m.r; i++) {
154         out << (i > 0 ? ", [" : "[";
155         for (int j = 0; j < m.c; j++)
156             out << (j > 0 ? ", " : "") << m.mat[i][j];
157         out << "]";
158     }
159     out << "]";
160     return out;
161 }
162 };
163
164 /** Example Usage */
165
166 #include <cassert>
167 #include <iostream>
168 using namespace std;
169
170 int main() {
171     int a[2][2] = {{1,8}, {5,9}};
172     matrix<int> m(5, 5, 10), m2(a);
173     m += 10;
174     m[0][0] += 10;
175     assert(m[0][0] == 30 && m[1][1] == 20);
176     assert(powsum(m2, 3) == m2 + m2*m2 + (m2^3));
177     return 0;
178 }

```

4.5.2 Determinant (Gauss)

```

1  /*
2
3  The following are ways to compute the determinant of a
4  matrix directly using Gaussian elimination. See the
5  following section for a generalized solution using LU
6  decompositions. Since the determinant can get very large,
7  look out for overflows and floating-point inaccuracies.
8  Bignums are recommended for maximal correctness.
9
10 Complexity:  $O(N^3)$ , except for the adjustment for
11 overflow in the integer det() function.
12
13 Precondition: All input matrices must be square.
14
15 */
16
17 #include <algorithm> /* std::swap() */
18 #include <cassert>
19 #include <cmath>     /* fabs() */
20 #include <map>
21 #include <vector>
22
23 static const double eps = 1e-10;
24 typedef std::vector<std::vector<int> > vvi;
25 typedef std::vector<std::vector<double> > vvd;
26
27 double det(vvd a) {
28     int n = a.size();
29     assert(!a.empty() && n == (int)a[0].size());
30     double res = 1;
31     std::vector<bool> used(n, false);
32     for (int i = 0; i < n; i++) {
33         int p;
34         for (p = 0; p < n; p++)
35             if (!used[p] && fabs(a[p][i]) > eps)
36                 break;
37         if (p >= n) return 0;
38         res *= a[p][i];
39         used[p] = true;
40         double z = 1 / a[p][i];
41         for (int j = 0; j < n; j++) a[p][j] *= z;
42         for (int j = 0; j < n; j++) {
43             if (j == p) continue;
44             z = a[j][i];
45             for (int k = 0; k < n; k++)
46                 a[j][k] -= z * a[p][k];
47         }
48     }
49     return res;
50 }
51
52 /*
53
54 Determinant of Integer Matrix
55
56 This is prone to overflow, so it is recommended you use your
57 own bigint class instead of long long. At the end of this
58 function, the final answer is found as a product of powers.
59 You have two choices: change the "#if 0" to "#if 1" and use

```

```

60 the naive method to compute this product and risk overflow,
61 or keep it as "#if 0" and try to make the situation better
62 through prime factorization (less efficient). Note that
63 even in the prime factorization method, overflow may happen
64 if the final answer is too big for a long long.
65
66 */
67
68 //C++98 doesn't have an abs() for long long
69 template<class T> inline T _abs(const T & x) {
70     return x < 0 ? -x : x;
71 }
72
73 long long det(const vvi & a) {
74     int n = a.size();
75     assert(!a.empty() && n == (int)a[0].size());
76     long long b[n][n], det = 1;
77     for (int i = 0; i < n; i++)
78         for (int j = 0; j < n; j++) b[i][j] = a[i][j];
79     int sign = 1, exponent[n];
80     for (int i = 0; i < n; i++) {
81         exponent[i] = 0;
82         int k = i;
83         for (int j = i + 1; j < n; j++) {
84             if (b[k][i] == 0 || (b[j][i] != 0 && _abs(b[k][i]) > _abs(b[j][i])))
85                 k = j;
86         }
87         if (b[k][i] == 0) return 0;
88         if (i != k) {
89             sign = -sign;
90             for (int j = 0; j < n; j++)
91                 std::swap(b[i][j], b[k][j]);
92         }
93         exponent[i]++;
94         for (int j = i + 1; j < n; j++)
95             if (b[j][i] != 0) {
96                 for (int p = i + 1; p < n; ++p)
97                     b[j][p] = b[j][p] * b[i][i] - b[i][p] * b[j][i];
98                 exponent[i]--;
99             }
100     }
101
102     #if 0
103     for (int i = 0; i < n; i++)
104         for (; exponent[i] > 0; exponent[i]--)
105             det *= b[i][i];
106     for (int i = 0; i < n; i++)
107         for (; exponent[i] < 0; exponent[i]++)
108             det /= b[i][i];
109     #else
110     std::map<long long, int> m;
111     for (int i = 0; i < n; i++) {
112         long long x = b[i][i];
113         for (long long d = 2; ; d++) {
114             long long power = 0, quo = x / d, rem = x - quo * d;
115             if (d > quo || (d == quo && rem > 0)) break;
116             for (; rem == 0; rem = x - quo * d) {
117                 power++;
118                 x = quo;

```

```

119         quo = x / d;
120     }
121     if (power > 0) m[d] += power * exponent[i];
122 }
123 if (x > 1) m[x] += exponent[i];
124 }
125 std::map<long long, int>::iterator it;
126 for (it = m.begin(); it != m.end(); ++it)
127     for (int i = 0; i < it->second; i++)
128         det *= it->first;
129 #endif
130
131 return sign < 0 ? -det : det;
132 }
133
134 /** Example Usage */
135
136 #include <iostream>
137 using namespace std;
138
139 int main() {
140     const int n = 3;
141     int a[n][n] = {{6,1,1},{4,-2,5},{2,8,7}};
142     vvi v1(n);
143     vvd v2(n);
144     for (int i = 0; i < n; i++) {
145         v1[i] = vector<int>(a[i], a[i] + n);
146         v2[i] = vector<double>(a[i], a[i] + n);
147     }
148     int d1 = det(v1);
149     int d2 = (int)det(v2);
150     assert(d1 == d2 && d2 == -306);
151     return 0;
152 }

```

4.5.3 Gaussian Elimination

```

1  /*
2
3  Given a system of m linear equations with n unknowns:
4
5  A(1,1)*x(1) + A(1,2)*x(2) + ... + A(1,n)*x(n) = B(1)
6  A(2,1)*x(1) + A(2,2)*x(2) + ... + A(2,n)*x(n) = B(2)
7  ...
8  A(m,1)*x(1) + A(m,2)*x(2) + ... + A(m,n)*x(n) = B(m)
9
10 For any system of linear equations, there will either
11 be no solution (in 2d, lines are parallel), a single
12 solution (in 2d, the lines intersect at a point), or
13 or infinite solutions (in 2d, lines are the same).
14
15 Using Gaussian elimination in O(n^3), this program
16 solves for the values of x(1) ... x(n) or determines
17 that no unique solution of x() exists. Note that
18 the implementation below uses 0-based indices.
19
20 */

```

```

21
22 #include <algorithm> /* std::swap() */
23 #include <cmath>     /* fabs() */
24 #include <vector>
25
26 const double eps = 1e-9;
27 typedef std::vector<double> vd;
28 typedef std::vector<vd> vvd;
29
30 //note: A[i][n] stores B[i]
31 //if no unique solution found, returns empty vector
32 vd solve_system(vvd A) {
33     int m = A.size(), n = A[0].size() - 1;
34     vd x(n);
35     if (n > m) goto fail;
36     for (int k = 0; k < n; k++) {
37         double mv = 0;
38         int mi = -1;
39         for (int i = k; i < m; i++)
40             if (mv < fabs(A[i][k])) {
41                 mv = fabs(A[i][k]);
42                 mi = i;
43             }
44         if (mv < eps) goto fail;
45         for (int i = 0; i <= n; i++)
46             std::swap(A[mi][i], A[k][i]);
47         for (int i = k + 1; i < m; i++) {
48             double v = A[i][k] / A[k][k];
49             for (int j = k; j <= n; j++)
50                 A[i][j] -= v * A[k][j];
51             A[i][k] = 0;
52         }
53     }
54     for (int i = n; i < m; i++)
55         if (fabs(A[i][n]) > eps) goto fail;
56     for (int i = n - 1; i >= 0; i--) {
57         if (fabs(A[i][i]) < eps) goto fail;
58         double v = 0;
59         for (int j = i + 1; j < n; j++)
60             v += A[i][j] * x[j];
61         v = A[i][n] - v;
62         x[i] = v / A[i][i];
63     }
64     return x;
65 fail:
66     return vd();
67 }
68
69 /** Example Usage (wcipeg.com/problem/syssolve) */
70
71 #include <iostream>
72 using namespace std;
73
74 int main() {
75     int n, m;
76     cin >> n >> m;
77     vvd a(m, vd(n + 1));
78     for (int i = 0; i < m; i++)
79         for (int j = 0; j <= n; j++)

```

```

80     cin >> a[i][j];
81     vd x = solve_system(a);
82     if (x.empty()) {
83         cout << "NO_UNIQUE_SOLUTION\n";
84     } else {
85         cout.precision(6);
86         for (int i = 0; i < n; i++)
87             cout << fixed << x[i] << "\n";
88     }
89     return 0;
90 }

```

4.5.4 LU Decomposition

```

1  /*
2
3  The LU (lower upper) decomposition of a matrix is a factorization
4  of a matrix as the product of a lower triangular matrix and an
5  upper triangular matrix. With the LU decomposition, we can solve
6  many problems, including the determinant of the matrix, a systems
7  of linear equations, and the inverse of a matrix.
8
9  Note: in the following implementation, each call to det(),
10 solve_system(), and inverse() recomputes the lu decomposition.
11 For the same matrix, you should precompute the lu decomposition
12 and reuse it for several of these operations afterwards.
13
14 Complexity:  $O(n^3)$  for lu_decompose(). det() uses the running time
15 of lu_decompose(), plus an addition  $O(n)$  term. solve_system() and
16 inverse() both have the running time of lu_decompose(), plus an
17 additional  $O(n^3)$  term.
18
19 */
20
21 #include <algorithm> /* std::swap() */
22 #include <cassert>
23 #include <cmath>     /* fabs() */
24 #include <vector>
25
26 static const double eps = 1e-10;
27 typedef std::vector<double> vd;
28 typedef std::vector<vd> vvd;
29
30 /*
31
32 LU decomposition with Gauss-Jordan elimination. This is generalized
33 for rectangular matrices. Since the resulting L and U matrices have
34 all mutually exclusive 0's (except when  $i == j$ ), we can merge them
35 into a single LU matrix to save memory. Note:  $l[i][i] = 1$  for all  $i$ .
36
37 Optionally determine the permutation vector p. If an array p is
38 passed, p[i] will be populated such that p[i] is the only column of
39 the i-th row of the permutation matrix that is equal to 1.
40
41 Returns: a matrix m, the merged lower/upper triangular matrix:
42         m[i][j] = l[i][j] (for  $i > j$ ) or u[i][j] (for  $i \leq j$ )
43

```



```

44  */
45
46  vvd lu_decompose(vvd a, int * detsign = 0, int * p = 0) {
47      int n = a.size(), m = a[0].size();
48      int sign = 1;
49      if (p != 0)
50          for (int i = 0; i < n; i++) p[i] = i;
51      for (int r = 0, c = 0; r < n && c < m; r++, c++) {
52          int pr = r;
53          for (int i = r + 1; i < n; i++)
54              if (fabs(a[i][c]) > fabs(a[pr][c]))
55                  pr = i;
56          if (fabs(a[pr][c]) <= eps) {
57              r--;
58              continue;
59          }
60          if (pr != r) {
61              if (p != 0) std::swap(p[r], p[pr]);
62              sign = -sign;
63              for (int i = 0; i < m; i++)
64                  std::swap(a[r][i], a[pr][i]);
65          }
66          for (int s = r + 1; s < n; s++) {
67              a[s][c] /= a[r][c];
68              for (int d = c + 1; d < m; d++)
69                  a[s][d] -= a[s][c] * a[r][d];
70          }
71      }
72      if (detsign != 0) *detsign = sign;
73      return a;
74  }
75
76  double getl(const vvd & lu, int i, int j) {
77      if (i > j) return lu[i][j];
78      return i < j ? 0.0 : 1.0;
79  }
80
81  double getu(const vvd & lu, int i, int j) {
82      return i <= j ? lu[i][j] : 0.0;
83  }
84
85  //Precondition: A is square matrix.
86  double det(const vvd & a) {
87      int n = a.size(), detsign;
88      assert(!a.empty() && n == (int)a[0].size());
89      vvd lu = lu_decompose(a, &detsign);
90      double det = 1;
91      for (int i = 0; i < n; i++)
92          det *= lu[i][i];
93      return detsign < 0 ? -det : det;
94  }
95
96  /*
97
98  Solves system of linear equations with forward/backwards
99  substitution. Precondition: A must be n*n and B must be n*m.
100 Returns: an n by m matrix X such that A*X = B.
101
102  */

```

```

103
104 vvd solve_system(const vvd & a, const vvd & b) {
105     int n = b.size(), m = b[0].size();
106     assert(!a.empty() && n == (int)a.size() && n == (int)a[0].size());
107     int detsign, p[a.size()];
108     vvd lu = lu_decompose(a, &detsign, p);
109     //forward substitute for Y in L*Y = B
110     vvd y(n, vd(m));
111     for (int j = 0; j < m; j++) {
112         y[0][j] = b[p[0]][j] / getl(lu, 0, 0);
113         for (int i = 1; i < n; i++) {
114             double s = 0;
115             for (int k = 0; k < i; k++)
116                 s += getl(lu, i, k) * y[k][j];
117             y[i][j] = (b[p[i]][j] - s) / getl(lu, i, i);
118         }
119     }
120     //backward substitute for X in U*X = Y
121     vvd x(n, vd(m));
122     for (int j = 0; j < m; j++) {
123         x[n - 1][j] = y[n - 1][j] / getu(lu, n-1, n-1);
124         for (int i = n - 2; i >= 0; i--) {
125             double s = 0;
126             for (int k = i + 1; k < n; k++)
127                 s += getu(lu, i, k) * x[k][j];
128             x[i][j] = (y[i][j] - s) / getu(lu, i, i);
129         }
130     }
131     return x;
132 }
133
134 /*
135
136 Find the inverse A-1 of a matrix A. The inverse of a matrix
137 satisfies A * A-1 = I, where I is the identity matrix (for
138 all pairs (i, j), I[i][j] = 1 iff i = j, else I[i][j] = 0).
139 The inverse of a matrix exists if and only if det(a) is not 0.
140 We're lazy, so we just generate I and call solve_system().
141
142 Precondition: A is a square and det(A) != 0.
143
144 */
145
146 vvd inverse(const vvd & a) {
147     int n = a.size();
148     assert(!a.empty() && n == (int)a[0].size());
149     vvd I(n, vd(n));
150     for (int i = 0; i < n; i++) I[i][i] = 1;
151     return solve_system(a, I);
152 }
153
154 /** Example Usage */
155
156 #include <cstdio>
157 #include <iostream>
158 using namespace std;
159
160 void print(const vvd & m) {
161     cout << "[";

```

```

162     for (int i = 0; i < (int)m.size(); i++) {
163         cout << (i > 0 ? ",[" : "[");
164         for (int j = 0; j < (int)m[0].size(); j++)
165             cout << (j > 0 ? ", " : "") << m[i][j];
166         cout << "]";
167     }
168     cout << "\n";
169 }
170
171 void printlu(const vvd & lu) {
172     printf("L:\n");
173     for (int i = 0; i < (int)lu.size(); i++) {
174         for (int j = 0; j < (int)lu[0].size(); j++)
175             printf("%10.5f", getl(lu, i, j));
176         printf("\n");
177     }
178     printf("U:\n");
179     for (int i = 0; i < (int)lu.size(); i++) {
180         for (int j = 0; j < (int)lu[0].size(); j++)
181             printf("%10.5f", getu(lu, i, j));
182         printf("\n");
183     }
184 }
185
186 int main() {
187     { //determinant of 3x3
188         const int n = 3;
189         double a[n][n] = {{1,3,5},{2,4,7},{1,1,0}};
190         vvd v(n);
191         for (int i = 0; i < n; i++)
192             v[i] = vector<double>(a[i], a[i] + n);
193         printlu(lu_decompose(v));
194         cout << "determinant:_" << det(v) << "\n"; //4
195     }
196
197     { //determinant of 4x4
198         const int n = 4;
199         double a[n][n] = {{11,9,24,2},{1,5,2,6},{3,17,18,1},{2,5,7,1}};
200         vvd v(n);
201         for (int i = 0; i < n; i++)
202             v[i] = vector<double>(a[i], a[i] + n);
203         printlu(lu_decompose(v));
204         cout << "determinant:_" << det(v) << "\n"; //284
205     }
206
207     { //solve for [x, y] in x + 3y = 4 && 2x + 3y = 6
208         const int n = 2;
209         double a[n][n] = {{1,3},{2,3}};
210         double b[n] = {4, 6};
211         vvd va(n), vb(n);
212         for (int i = 0; i < n; i++) {
213             va[i] = vector<double>(a[i], a[i] + n);
214             vb[i] = vector<double>(1, b[i]);
215         }
216         vvd x = solve_system(va, vb);
217         for (int i = 0; i < n; i++) {
218             assert(fabs(a[i][0]*x[0][0] + a[i][1]*x[1][0] - b[i]) < eps);
219         }
220     }

```

```

221
222 { //find inverse by solving a system
223     const int n = 2;
224     double a[n][n] = {{2,3},{1,2}};
225     vvd v(n);
226     for (int i = 0; i < n; i++)
227         v[i] = vector<double>(a[i], a[i] + n);
228     print(inverse(v)); //[2,-3],[-1,2]]
229 }
230 return 0;
231 }

```

4.5.5 Simplex Algorithm

```

1  /*
2
3  Description: The canonical form of a linear programming
4  problem is to maximize  $c^T x$ , subject to  $Ax \leq b$ , and  $x \geq 0$ .
5  where  $x$  is the vector of variables (to be solved),  $c$  and  $b$ 
6  are vectors of (known) coefficients,  $A$  is a (known) matrix of
7  coefficients, and  $(.)^T$  is the matrix transpose. The following
8  implementation solves  $n$  variables in a system of  $m$  constraints.
9
10 Precondition:  $ab$  has dimensions  $m$  by  $n+1$  and  $c$  has length  $n+1$ .
11
12 Complexity: The simplex method is remarkably efficient in
13 practice, usually taking  $2m$  or  $3m$  iterations, converging in
14 expected polynomial time for certain distributions of random
15 inputs. However, its worst-case complexity is exponential,
16 and can be demonstrated with carefully constructed examples.
17
18 */
19
20 #include <algorithm> /* std::swap() */
21 #include <cfloat> /* DBL_MAX */
22 #include <cmath> /* fabs() */
23 #include <vector>
24
25 typedef std::vector<double> vd;
26 typedef std::vector<vd> vvd;
27
28 //ab[i][0..n-1] stores A and ab[i][n] stores B
29 vd simplex(const vvd & ab, const vd & c, bool max = true) {
30     const double eps = 1e-10;
31     int n = c.size() - 1, m = ab.size();
32     vvd ts(m + 2, vd(n + 2));
33     ts[1][1] = max ? c[n] : -c[n];
34     for (int j = 1; j <= n; j++)
35         ts[1][j + 1] = max ? c[j - 1] : -c[j - 1];
36     for (int i = 1; i <= m; i++) {
37         for (int j = 1; j <= n; j++)
38             ts[i + 1][j + 1] = -ab[i - 1][j - 1];
39         ts[i + 1][1] = ab[i - 1][n];
40     }
41     for (int j = 1; j <= n; j++)
42         ts[0][j + 1] = j;
43     for (int i = n + 1; i <= n + m; i++)

```

```

44     ts[i - n + 1][0] = i;
45     double p1 = 0.0, p2 = 0.0;
46     bool done = true;
47     do {
48         double mn = DBL_MAX, xmax = 0.0, v;
49         for (int j = 2; j <= n + 1; j++)
50             if (ts[1][j] > 0.0 && ts[1][j] > xmax) {
51                 p2 = j;
52                 xmax = ts[1][j];
53             }
54         for (int i = 2; i <= m + 1; i++) {
55             v = fabs(ts[i][1] / ts[i][p2]);
56             if (ts[i][p2] < 0.0 && mn > v) {
57                 mn = v;
58                 p1 = i;
59             }
60         }
61         std::swap(ts[p1][0], ts[0][p2]);
62         for (int i = 1; i <= m + 1; i++) {
63             if (i == p1) continue;
64             for (int j = 1; j <= n + 1; j++)
65                 if (j != p2)
66                     ts[i][j] -= ts[p1][j] * ts[i][p2] / ts[p1][p2];
67         }
68         ts[p1][p2] = 1.0 / ts[p1][p2];
69         for (int j = 1; j <= n + 1; j++) {
70             if (j != p2)
71                 ts[p1][j] *= fabs(ts[p1][p2]);
72         }
73         for (int i = 1; i <= m + 1; i++) {
74             if (i != p1)
75                 ts[i][p2] *= ts[p1][p2];
76         }
77         for (int i = 2; i <= m + 1; i++)
78             if (ts[i][1] < 0.0) return vd(); //no solution
79         done = true;
80         for (int j = 2; j <= n + 1; j++)
81             if (ts[1][j] > 0) done = false;
82     } while (!done);
83     vd res;
84     for (int i = 1; i <= n; i++)
85         for (int j = 2; j <= m + 1; j++)
86             if (fabs(ts[j][0] - i) <= eps)
87                 res.push_back(ts[j][1]);
88     //the solution is stored in ts[1][1]
89     return res;
90 }
91
92 /** Example Usage **/
93
94 #include <iostream>
95 using namespace std;
96
97 /*
98     Maximize  $3x + 4y + 5$ , subject to  $x, y \geq 0$  and:
99      $-2x + 1y \leq 0$ 
100      $1x + 0.85y \leq 9$ 
101      $1x + 2y \leq 14$ 
102

```

```

103     Note: The solution is 38.3043 at (5.30435, 4.34783).
104  */
105
106  int main() {
107      const int n = 2, m = 3;
108      double ab[m][n + 1] = {{-2, 1, 0}, {1, 0.85, 9}, {1, 2, 14}};
109      double c[n + 1] = {3, 4, 5};
110      vvd vab(m, vd(n + 1));
111      vd vc(c, c + n + 1);
112      for (int i = 0; i < m; i++) {
113          for (int j = 0; j <= n; j++)
114              vab[i][j] = ab[i][j];
115      }
116      vd x = simplex(vab, vc);
117      if (x.empty()) {
118          cout << "No solution.\n";
119      } else {
120          double solval = c[n];
121          for (int i = 0; i < (int)x.size(); i++)
122              solval += c[i] * x[i];
123          cout << "Solution=" << solval;
124          cout << " at (" << x[0];
125          for (int i = 1; i < (int)x.size(); i++)
126              cout << ", " << x[i];
127          cout << ").\n";
128      }
129      return 0;
130  }

```

4.6 Root-Finding

4.6.1 Real Root Finding (Differentiation)

```

1  /*
2
3  Real roots can be found via binary searching, a.k.a the bisection
4  method. If two x-coordinates evaluate to y-coordinates that have
5  opposite signs, a root must exist between them. For a polynomial
6  function, at most 1 root lies between adjacent local extrema.
7  Since local extrema exist where the derivative equals 0, we can
8  break root-finding into the subproblem of finding the roots of
9  the derivative. Recursively solve for local extrema until we get
10 to a base case of degree 0. For each set of local extrema found,
11 binary search between pairs of extrema for a root. This method is
12 easy, robust, and allows us to find the root to an arbitrary level
13 of accuracy. We're limited only by the precision of the arithmetic.
14
15 Complexity: For a degree N polynomial, repeatedly differentiating
16 it will take N + (N-1) + ... + 1 = O(N^2) operations. At each step
17 we binary search the number of times equal to the current degree.
18 If we want to make roots precise to eps=10^-P, each binary search
19 will take O(log P). Thus the overall complexity is O(N^2 log P).
20
21  */
22
23  #include <cmath>    /* fabs1(), pow1() */

```

```

24 #include <limits> /* std::numeric_limits<>::quiet_NaN() */
25 #include <utility> /* std::pair<> */
26 #include <vector>
27
28 typedef long double Double;
29 typedef std::vector<std::pair<Double, int> > poly;
30
31 const Double epsa = 1e-11; //required precision of roots in absolute error
32 const Double epsr = 1e-15; //required precision of roots in relative error
33 const Double eps0 = 1e-17; //x is considered a root if fabs(eval(x))<=eps0
34 const Double inf = 1e20;    //[-inf, inf] is the range of roots to consider
35 const Double NaN = std::numeric_limits<Double>::quiet_NaN();
36
37 Double eval(const poly & p, Double x) {
38     Double res = 0;
39     for (int i = 0; i < (int)p.size(); i++)
40         res += p[i].first * powl(x, p[i].second);
41     return res;
42 }
43
44 Double find_root(const poly & p, Double x1, Double x2) {
45     Double y1 = eval(p, x1), y2 = eval(p, x2);
46     if (fabsl(y1) <= eps0) return x1;
47     bool neg1 = (y1 < 0), neg2 = (y2 < 0);
48     if (fabsl(y2) <= eps0 || neg1 == neg2) return NaN;
49     while (x2 - x1 > epsa && x1 * (1 + epsr) < x2 && x2 * (1 + epsr) > x1) {
50         Double x = (x1 + x2) / 2;
51         ((eval(p, x) < 0) == neg1 ? x1 : x2) = x;
52     }
53     return x1;
54 }
55
56 std::vector<Double> find_all_roots(const poly & p) {
57     poly dif;
58     for (int i = 0; i < (int)p.size(); i++)
59         if (p[i].second > 0)
60             dif.push_back(std::make_pair(p[i].first * p[i].second, p[i].second - 1));
61     if (dif.empty()) return std::vector<Double>();
62     std::vector<Double> res, r = find_all_roots(dif);
63     r.insert(r.begin(), -inf);
64     r.push_back(inf);
65     for (int i = 0; i < (int)r.size() - 1; i++) {
66         Double root = find_root(p, r[i], r[i + 1]);
67         if (root != root) continue; //NaN, not found
68         if (res.empty() || root != res.back())
69             res.push_back(root);
70     }
71     return res;
72 }
73
74 /** Example Usage (http://wcipeg.com/problem/rootsolve) */
75
76 #include <iostream>
77 using namespace std;
78
79 int main() {
80     int n, d;
81     Double c;
82     poly p;

```

```

83     cin >> n;
84     for (int i = 0; i < n; i++) {
85         cin >> c >> d;
86         p.push_back(make_pair(c, d));
87     }
88     vector<Double> sol = find_all_roots(p);
89     if (sol.empty()) {
90         cout << "NO_REAL_ROOTS\n";
91     } else {
92         cout.precision(9);
93         for (int i = 0; i < (int)sol.size(); i++)
94             cout << fixed << sol[i] << "\n";
95     }
96     return 0;
97 }

```

4.6.2 Complex Root Finding (Laguerre's)

```

1  /*
2
3  Laguerre's method can be used to not only find complex roots of
4  a polynomial, the polynomial may also have complex coefficients.
5  From extensive empirical study, Laguerre's method is observed to
6  be very close to being a "sure-fire" method, as it is almost
7  guaranteed to always converge to some root of the polynomial
8  regardless of what initial guess is chosen.
9
10 */
11
12 #include <complex>
13 #include <cstdlib> /* rand(), RAND_MAX */
14 #include <vector>
15
16 typedef long double Double;
17 typedef std::complex<Double> cdouble;
18 typedef std::vector<cdouble> poly;
19
20 const Double eps = 1e-12;
21
22 std::pair<poly, cdouble> horner(const poly & a, const cdouble & x) {
23     int n = a.size();
24     poly b = poly(std::max(1, n - 1));
25     for (int i = n - 1; i > 0; i--)
26         b[i - 1] = a[i] + (i < n - 1 ? b[i] * x : 0);
27     return std::make_pair(b, a[0] + b[0] * x);
28 }
29
30 cdouble eval(const poly & p, const cdouble & x) {
31     return horner(p, x).second;
32 }
33
34 poly derivative(const poly & p) {
35     int n = p.size();
36     poly r(std::max(1, n - 1));
37     for (int i = 1; i < n; i++)
38         r[i - 1] = p[i] * cdouble(i);
39     return r;

```



```

40 }
41
42 int comp(const cdouble & x, const cdouble & y) {
43     Double diff = std::abs(x) - std::abs(y);
44     return diff < -eps ? -1 : (diff > eps ? 1 : 0);
45 }
46
47 cdouble find_one_root(const poly & p, cdouble x) {
48     int n = p.size() - 1;
49     poly p1 = derivative(p), p2 = derivative(p1);
50     for (int step = 0; step < 10000; step++) {
51         cdouble y0 = eval(p, x);
52         if (comp(y0, 0) == 0) break;
53         cdouble G = eval(p1, x) / y0;
54         cdouble H = G * G - eval(p2, x) / y0;
55         cdouble R = std::sqrt(cdouble(n - 1) * (H * cdouble(n) - G * G));
56         cdouble D1 = G + R, D2 = G - R;
57         cdouble a = cdouble(n) / (comp(D1, D2) > 0 ? D1 : D2);
58         x -= a;
59         if (comp(a, 0) == 0) break;
60     }
61     return x;
62 }
63
64 std::vector<cdouble> find_all_roots(const poly & p) {
65     std::vector<cdouble> res;
66     poly q = p;
67     while (q.size() > 2) {
68         cdouble z(rand()/Double(RAND_MAX), rand()/Double(RAND_MAX));
69         z = find_one_root(q, z);
70         z = find_one_root(p, z);
71         q = horner(q, z).first;
72         res.push_back(z);
73     }
74     res.push_back(-q[0] / q[1]);
75     return res;
76 }
77
78 /** Example Usage */
79
80 #include <cstdio>
81 #include <iostream>
82 using namespace std;
83
84 void print_roots(vector<cdouble> roots) {
85     for (int i = 0; i < (int)roots.size(); i++) {
86         printf("%9.5f, ", (double)roots[i].real());
87         printf("%9.5f\n", (double)roots[i].imag());
88     }
89 }
90
91 int main() {
92     { //  $x^3 - 8x^2 - 13x + 140 = (x + 4)(x - 5)(x - 7)$ 
93         printf("Roots of  $x^3 - 8x^2 - 13x + 140$ : \n");
94         poly p;
95         p.push_back(140);
96         p.push_back(-13);
97         p.push_back(-8);
98         p.push_back(1);

```

```

99     vector<cdouble> roots = find_all_roots(p);
100     print_roots(roots);
101 }
102
103 { //(-6+4i)x^4 + (-26+12i)x^3 + (-30+40i)x^2 + (-26+12i)x + (-24+36i)
104   // = ((2+3i)x + 6)*(x + i)*(2x + (6+4i))*(x*i + 1)
105   printf("Roots of ((2+3i)x+6)(x+i)(2x+(6+4i))(x*i+1):\n");
106   poly p;
107   p.push_back(cdouble(-24, 36));
108   p.push_back(cdouble(-26, 12));
109   p.push_back(cdouble(-30, 40));
110   p.push_back(cdouble(-26, 12));
111   p.push_back(cdouble(-6, 4));
112   vector<cdouble> roots = find_all_roots(p);
113   print_roots(roots);
114 }
115 return 0;
116 }

```

4.6.3 Complex Root Finding (RPOLY)

```

1  /*
2
3  Determine the complex roots of a polynomial with real coefficients.
4  This is the variant of the Jenkins-Traub algorithm for polynomials
5  with real coefficient, known as RPOLY. RPOLY follows follows the
6  same pattern as the CPOLY algorithm, but computes two roots at a
7  time, either two real roots or a pair of conjugate complex roots.
8  See: https://en.wikipedia.org/wiki/Jenkins%E2%80%93Traub\_algorithm
9
10 The following is a translation of TOMS493 (www.netlib.org/toms/)
11 from FORTRAN to C++, with a simple wrapper at the end for the C++
12 <complex> class. Although the code is not meant to be read, it is
13 extremely efficient and robust, capable of achieving an accuracy
14 of at least 5 decimal places for even the most strenuous inputs.
15
16 */
17
18 #include <cfloat> /* LDBL_EPSILON, LDBL_MAX, LDBL_MIN */
19 #include <cmath> /* cosl, expl, fabsl, logl, powl, sinl, sqrtl */
20
21 typedef long double LD;
22
23 void divide_quadratic(int n, LD u, LD v, LD p[], LD q[], LD * a, LD * b) {
24     q[0] = *b = p[0];
25     q[1] = *a = -((*b) * u) + p[1];
26     for (int i = 2; i < n; i++) {
27         q[i] = -((*a) * u + (*b) * v) + p[i];
28         *b = *a;
29         *a = q[i];
30     }
31 }
32
33 int get_flag(int n, LD a, LD b, LD * a1, LD * a3, LD * a7,
34             LD * c, LD * d, LD * e, LD * f, LD * g, LD * h,
35             LD k[], LD u, LD v, LD qk[]) {
36     divide_quadratic(n, u, v, k, qk, c, d);

```

```

37  if (fabs1(*c) <= 100.0 * LDBL_EPSILON * fabs1(k[n - 1]) &&
38      fabs1(*d) <= 100.0 * LDBL_EPSILON * fabs1(k[n - 2])) return 3;
39  *h = v * b;
40  if (fabs1(*d) >= fabs1(*c)) {
41      *e = a / (*d);
42      *f = (*c) / (*d);
43      *g = u * b;
44      *a1 = (*f) * b - a;
45      *a3 = (*e) * ((*g) + a) + (*h) * (b / (*d));
46      *a7 = (*h) + ((*f) + u) * a;
47      return 2;
48  }
49  *e = a / (*c);
50  *f = (*d) / (*c);
51  *g = (*e) * u;
52  *a1 = -(a * ((*d) / (*c))) + b;
53  *a3 = (*e) * a + ((*g) + (*h) / (*c)) * b;
54  *a7 = (*g) * (*d) + (*h) * (*f) + a;
55  return 1;
56 }
57
58 void find_polynomials(int n, int flag, LD a, LD b, LD a1, LD * a3,
59                      LD * a7, LD k[], LD qk[], LD qp[]) {
60     if (flag == 3) {
61         k[1] = k[0] = 0.0;
62         for (int i = 2; i < n; i++) k[i] = qk[i - 2];
63         return;
64     }
65     if (fabs1(a1) > 10.0 * LDBL_EPSILON * fabs1(flag == 1 ? b : a)) {
66         *a7 /= a1;
67         *a3 /= a1;
68         k[0] = qp[0];
69         k[1] = qp[1] - (*a7) * qp[0];
70         for (int i = 2; i < n; i++)
71             k[i] = qp[i] - ((*a7) * qp[i - 1]) + (*a3) * qk[i - 2];
72     } else {
73         k[0] = 0.0;
74         k[1] = -(*a7) * qp[0];
75         for (int i = 2; i < n; i++)
76             k[i] = (*a3) * qk[i - 2] - (*a7) * qp[i - 1];
77     }
78 }
79
80 void estimate_coeff(int flag, LD * uu, LD * vv, LD a, LD a1, LD a3, LD a7,
81                   LD b, LD c, LD d, LD f, LD g, LD h, LD u, LD v, LD k[],
82                   int n, LD p[]) {
83     LD a4, a5, b1, b2, c1, c2, c3, c4, temp;
84     *vv = *uu = 0.0;
85     if (flag == 3) return;
86     if (flag != 2) {
87         a4 = a + u * b + h * f;
88         a5 = c + (u + v * f) * d;
89     } else {
90         a4 = (a + g) * f + h;
91         a5 = (f + u) * c + v * d;
92     }
93     b1 = -k[n - 1] / p[n];
94     b2 = -(k[n - 2] + b1 * p[n - 1]) / p[n];
95     c1 = v * b2 * a1;

```

```

96     c2 = b1 * a7;
97     c3 = b1 * b1 * a3;
98     c4 = c1 - c2 - c3;
99     temp = b1 * a4 - c4 + a5;
100    if (temp != 0.0) {
101        *uu= u - (u * (c3 + c2) + v * (b1 * a1 + b2 * a7)) / temp;
102        *vv= v * (1.0 + c4 / temp);
103    }
104 }
105
106 void solve_quadratic(LD a, LD b1, LD c, LD * sr, LD * si, LD * lr, LD * li) {
107     LD b, d, e;
108     *sr = *si = *lr = *li = 0.0;
109     if (a == 0) {
110         *sr = (b1 != 0) ? -c / b1 : *sr;
111         return;
112     }
113     if (c == 0) {
114         *lr = -b1 / a;
115         return;
116     }
117     b = b1 / 2.0;
118     if (fabsl(b) < fabsl(c)) {
119         e = (c >= 0) ? a : -a;
120         e = b * (b / fabsl(c)) - e;
121         d = sqrtl(fabsl(e)) * sqrtl(fabsl(c));
122     } else {
123         e = 1.0 - (a / b) * (c / b);
124         d = sqrtl(fabsl(e)) * fabsl(b);
125     }
126     if (e >= 0) {
127         d = (b >= 0) ? -d : d;
128         *lr = (d - b) / a;
129         *sr = (*lr != 0) ? (c / *lr / a) : *sr;
130     } else {
131         *lr = *sr = -b / a;
132         *si = fabsl(d / a);
133         *li = -(*si);
134     }
135 }
136
137 void quadratic_iterate(int N, int * NZ, LD uu, LD vv,
138                      LD * szr, LD * szi, LD * lzt, LD * lzi, LD qp[],
139                      int n, LD * a, LD * b, LD p[], LD qk[],
140                      LD * a1, LD * a3, LD * a7, LD * c, LD * d, LD * e,
141                      LD * f, LD * g, LD * h, LD k[]) {
142     int steps = 0, flag, tried_flag = 0;
143     LD ee, mp, omp = 0.0, relstp = 0.0, t, u, ui, v, vi, zm;
144     *NZ = 0;
145     u = uu;
146     v = vv;
147     do {
148         solve_quadratic(1.0, u, v, szr, szi, lzt, lzi);
149         if (fabsl(fabsl(*szr) - fabsl(*lzt)) > 0.01 * fabsl(*lzt)) break;
150         divide_quadratic(n, u, v, p, qp, a, b);
151         mp = fabsl(-((*szr) * (*b)) + *a) + fabsl((*szi) * (*b));
152         zm = sqrtl(fabsl(v));
153         ee = 2.0 * fabsl(qp[0]);
154         t = -(*szr) * (*b);

```

```

155     for (int i = 1; i < N; i++) ee = ee * zm + fabsl(qp[i]);
156     ee = ee * zm + fabsl(*a + t);
157     ee = ee * 9.0 + 2.0 * fabsl(t) - 7.0 * (fabsl(*a + t) + zm * fabsl(*b));
158     ee *= LDBL_EPSILON;
159     if (mp <= 20.0 * ee) {
160         *NZ = 2;
161         break;
162     }
163     if (++steps > 20) break;
164     if (steps >= 2 && relstp <= 0.01 && mp >= omp && !tried_flag) {
165         relstp = (relstp < LDBL_EPSILON) ? sqrtl(LDBL_EPSILON) : sqrtl(relstp);
166         u -= u * relstp;
167         v += v * relstp;
168         divide_quadratic(n, u, v, p, qp, a, b);
169         for (int i = 0; i < 5; i++) {
170             flag = get_flag(N, *a, *b, a1, a3, a7, c, d, e, f, g, h, k, u, v, qk);
171             find_polynomials(N, flag, *a, *b, *a1, a3, a7, k, qk, qp);
172         }
173         tried_flag = 1;
174         steps = 0;
175     }
176     omp = mp;
177     flag = get_flag(N, *a, *b, a1, a3, a7, c, d, e, f, g, h, k, u, v, qk);
178     find_polynomials(N, flag, *a, *b, *a1, a3, a7, k, qk, qp);
179     flag = get_flag(N, *a, *b, a1, a3, a7, c, d, e, f, g, h, k, u, v, qk);
180     estimate_coeff(flag, &ui, &vi, *a, *a1, *a3, *a7, *b, *c, *d, *f, *g, *h,
181                   u, v, k, N, p);
182     if (vi != 0) {
183         relstp = fabsl((-v + vi) / vi);
184         u = ui;
185         v = vi;
186     }
187 } while (vi != 0);
188 }
189
190 void real_iterate(int * flag, int * nz, LD * sss, int n, LD p[],
191                 int nn, LD qp[], LD * szr, LD * szl, LD k[], LD qk[]) {
192     int steps = 0;
193     LD ee, kv, mp, ms, omp = 0.0, pv, s, t = 0.0;
194     *flag = *nz = 0;
195     for (s = *sss; ; s += t) {
196         pv = p[0];
197         qp[0] = pv;
198         for (int i = 1; i < nn; i++) qp[i] = pv = pv * s + p[i];
199         mp = fabsl(pv);
200         ms = fabsl(s);
201         ee = 0.5 * fabsl(qp[0]);
202         for (int i = 1; i < nn; i++) ee = ee * ms + fabsl(qp[i]);
203         if (mp <= 20.0 * LDBL_EPSILON * (2.0 * ee - mp)) {
204             *nz = 1;
205             *szr = s;
206             *szl = 0.0;
207             break;
208         }
209         if (++steps > 10) break;
210         if (steps >= 2 && fabsl(t) <= 0.001 * fabsl(s - t) && mp > omp) {
211             *flag = 1;
212             *sss = s;
213             break;

```

```

214     }
215     omp = mp;
216     qk[0] = kv = k[0];
217     for (int i = 1; i < n; i++) qk[i] = kv = kv * s + k[i];
218     if (fabs1(kv) > fabs1(k[n - 1]) * 10.0 * LDBL_EPSILON) {
219         t = -pv / kv;
220         k[0] = qp[0];
221         for (int i = 1; i < n; i++)
222             k[i] = t * qk[i - 1] + qp[i];
223     } else {
224         k[0] = 0.0;
225         for (int i = 1; i < n; i++)
226             k[i] = qk[i - 1];
227     }
228     kv = k[0];
229     for (int i = 1; i < n; i++) kv = kv * s + k[i];
230     t = fabs1(kv) > (fabs1(k[n - 1]) * 10.0 * LDBL_EPSILON) ? -pv / kv : 0.0;
231 }
232 }
233
234 void solve_fixedshift(int l2, int * nz, LD sr, LD v, LD k[], int n,
235                     LD p[], int nn, LD qp[], LD u, LD qk[], LD svk[],
236                     LD * lzi, LD * lzt, LD * szl, LD * szt) {
237     int flag, _flag, __flag = 1, spass, stry, vpass, vtry;
238     LD a, a1, a3, a7, b, betas, betav, c, d, e, f, g, h;
239     LD oss, ots = 0.0, otv = 0.0, ovv, s, ss, ts, tss, tv, tvv, ui, vi, vv;
240     *nz = 0;
241     betav = betas = 0.25;
242     oss = sr;
243     ovv = v;
244     divide_quadratic(nn, u, v, p, qp, &a, &b);
245     flag = get_flag(n, a, b, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h,
246                   k, u, v, qk);
247     for (int j = 0; j < 12; j++) {
248         _flag = 1;
249         find_polynomials(n, flag, a, b, a1, &a3, &a7, k, qk, qp);
250         flag = get_flag(n, a, b, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h,
251                       k, u, v, qk);
252         estimate_coeff(flag, &ui, &vi, a, a1, a3, a7, b, c, d, f, g, h,
253                      u, v, k, n, p);
254         vv = vi;
255         ss = k[n - 1] != 0.0 ? -p[n] / k[n - 1] : 0.0;
256         ts = tv = 1.0;
257         if (j != 0 && flag != 3) {
258             tv = (vv != 0.0) ? fabs1((vv - ovv) / vv) : tv;
259             ts = (ss != 0.0) ? fabs1((ss - oss) / ss) : ts;
260             tvv = (tv < otv) ? tv * otv : 1.0;
261             tss = (ts < ots) ? ts * ots : 1.0;
262             vpass = (tvv < betav) ? 1 : 0;
263             spass = (tss < betas) ? 1 : 0;
264             if (spass || vpass) {
265                 for (int i = 0; i < n; i++) svk[i] = k[i];
266                 s = ss; stry = vtry = 0;
267                 for (;;) {
268                     if (!(_flag && spass && (!vpass || tss < tvv))) {
269                         quadratic_iterate(n, nz, ui, vi, szr, szl, lzt, qp, nn,
270                                           &a, &b, p, qk, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h, k);
271                         if (*nz > 0) return;
272                         __flag = vtry = 1;

```

```

273         betav *= 0.25;
274         if (stry || !spass) {
275             __flag = 0;
276         } else {
277             for (int i = 0; i < n; i++) k[i] = svk[i];
278         }
279     }
280     _flag = 0;
281     if (__flag != 0) {
282         real_iterate(&__flag, nz, &s, n, p, nn, qp, szr, szl, k, qk);
283         if (*nz > 0) return;
284         stry = 1;
285         betas *= 0.25;
286         if (__flag != 0) {
287             ui = -(s + s);
288             vi = s * s;
289             continue;
290         }
291     }
292     for (int i = 0; i < n; i++) k[i] = svk[i];
293     if (!vpass || vtry) break;
294 }
295 divide_quadratic(nn, u, v, p, qp, &a, &b);
296 flag = get_flag(n, a, b, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h,
297               k, u, v, qk);
298 }
299 }
300 ovv = vv;
301 oss = ss;
302 otv = tv;
303 ots = ts;
304 }
305 }
306
307 void find_roots(int degree, LD co[], LD re[], LD im[]) {
308     int j, jj, n, nm1, nn, nz, zero, SZ = degree + 1;
309     LD k[SZ], p[SZ], pt[SZ], qp[SZ], temp[SZ], qk[SZ], svk[SZ];
310     LD bnd, df, dx, factor, ff, moduli_max, moduli_min, sc, x, xm;
311     LD aa, bb, cc, lzi, lzt, sr, szl, szr, t, u, xx, xxx, yy;
312     n = degree;
313     xx = sqrtl(0.5);
314     yy = -xx;
315     for (j = 0; co[n] == 0; n--, j++) re[j] = im[j] = 0.0;
316     nn = n + 1;
317     for (int i = 0; i < nn; i++) p[i] = co[i];
318     while (n >= 1) {
319         if (n <= 2) {
320             if (n < 2) {
321                 re[degree - 1] = -p[1] / p[0];
322                 im[degree - 1] = 0.0;
323             } else {
324                 solve_quadratic(p[0], p[1], p[2], &re[degree - 2], &im[degree - 2],
325                               &re[degree - 1], &im[degree - 1]);
326             }
327             break;
328         }
329         moduli_max = 0.0;
330         moduli_min = LDBL_MAX;
331         for (int i = 0; i < nn; i++) {

```

```

332     x = fabsl(p[i]);
333     if (x > moduli_max) moduli_max = x;
334     if (x != 0 && x < moduli_min) moduli_min = x;
335 }
336 sc = LDBL_MIN / LDBL_EPSILON / moduli_min;
337 if ((sc <= 1.0 && moduli_max >= 10) ||
338     (sc > 1.0 && LDBL_MAX / sc >= moduli_max)) {
339     sc = (sc == 0) ? LDBL_MIN : sc;
340     factor = powl(2.0, logl(sc) / logl(2.0));
341     if (factor != 1.0)
342         for (int i = 0; i < nn; i++) p[i] *= factor;
343 }
344 for (int i = 0; i < nn; i++) pt[i] = fabsl(p[i]);
345 pt[n] = -pt[n];
346 nm1 = n - 1;
347 x = expl((logl(-pt[n]) - logl(pt[0])) / (LD)n);
348 if (pt[nm1] != 0) {
349     xm = -pt[n] / pt[nm1];
350     if (xm < x) x = xm;
351 }
352 xm = x;
353 do {
354     x = xm;
355     xm = 0.1 * x;
356     ff = pt[0];
357     for (int i = 1; i < nn; i++) ff = ff * xm + pt[i];
358 } while (ff > 0);
359 dx = x;
360 do {
361     df = ff = pt[0];
362     for (int i = 1; i < n; i++) {
363         ff = x * ff + pt[i];
364         df = x * df + ff;
365     }
366     ff = x * ff + pt[n];
367     dx = ff / df;
368     x -= dx;
369 } while (fabsl(dx / x) > 0.005);
370 bnd = x;
371 for (int i = 1; i < n; i++)
372     k[i] = (LD)(n - i) * p[i] / (LD)n;
373 k[0] = p[0];
374 aa = p[n];
375 bb = p[nm1];
376 zero = (k[nm1] == 0) ? 1 : 0;
377 for (jj = 0; jj < 5; jj++) {
378     cc = k[nm1];
379     if (zero) {
380         for (int i = 0; i < nm1; i++) {
381             j = nm1 - i;
382             k[j] = k[j - 1];
383         }
384         k[0] = 0;
385         zero = (k[nm1] == 0) ? 1 : 0;
386     } else {
387         t = -aa / cc;
388         for (int i = 0; i < nm1; i++) {
389             j = nm1 - i;
390             k[j] = t * k[j - 1] + p[j];

```



```

391     }
392     k[0] = p[0];
393     zero = (fabsl(k[nm1]) <= fabsl(bb) * LDL_EPSILON * 10.0) ? 1 : 0;
394 }
395 }
396 for (int i = 0; i < n; i++) temp[i] = k[i];
397 static const LD DEG = 0.01745329251994329576923690768489L;
398 for (jj = 1; jj <= 20; jj++) {
399     xxx = -sinl(94.0 * DEG) * yy + cosl(94.0 * DEG) * xx;
400     yy = sinl(94.0 * DEG) * xx + cosl(94.0 * DEG) * yy;
401     xx = xxx;
402     sr = bnd * xx;
403     u = -2.0 * sr;
404     for (int i = 0; i < nn; i++) qk[i] = svk[i] = 0.0;
405     solve_fixedshift(20 * jj, &nz, sr, bnd, k, n, p, nn, qp, u,
406                     qk, svk, &lzi, &lzr, &szi, &szr);
407     if (nz != 0) {
408         j = degree - n;
409         re[j] = szr;
410         im[j] = szi;
411         nn = nn - nz;
412         n = nn - 1;
413         for (int i = 0; i < nn; i++) p[i] = qp[i];
414         if (nz != 1) {
415             re[j + 1] = lzr;
416             im[j + 1] = lzi;
417         }
418         break;
419     } else {
420         for (int i = 0; i < n; i++) k[i] = temp[i];
421     }
422 }
423 if (jj > 20) break;
424 }
425 }
426
427 /** Wrapper */
428
429 #include <algorithm> /* std::reverse(), std::sort() */
430 #include <complex>
431 #include <vector>
432
433 typedef std::complex<LD> root;
434
435 bool comp(const root & a, const root & b) {
436     if (real(a) != real(b)) return real(a) < real(b);
437     return imag(a) < imag(b);
438 }
439
440 std::vector<root> find_roots(int degree, LD coefficients[]) {
441     std::reverse(coefficients, coefficients + degree + 1);
442     LD re[degree], im[degree];
443     find_roots(degree, coefficients, re, im);
444     std::vector<root> res;
445     for (int i = 0; i < degree; i++)
446         res.push_back(root(re[i], im[i]));
447     std::sort(res.begin(), res.end(), comp);
448     return res;
449 }

```

```

450
451  /** Example Usage (http://wcipeg.com/problem/rootsolve) ***/
452
453  #include <iostream>
454  using namespace std;
455
456  int T, degree, p;
457  LD c, coeff[101];
458
459  int main() {
460      degree = 0;
461      cin >> T;
462      for (int i = 0; i < T; i++) {
463          cin >> c >> p;
464          if (p > degree) degree = p;
465          coeff[p] = c;
466      }
467      std::vector<root> roots = find_roots(degree, coeff);
468      bool printed = false;
469      cout.precision(6);
470      for (int i = 0; i < (int)roots.size(); i++) {
471          if (fabsl(roots[i].imag()) < LDBL_EPSILON) {
472              cout << fixed << roots[i].real() << "\n";
473              printed = true;
474          }
475      }
476      if (!printed) cout << "NO_REAL_ROOTS\n";
477      return 0;
478  }

```

4.7 Integration

4.7.1 Simpson's Rule

```

1  /*
2
3  Simpson's rule is a method for numerical integration, the
4  numerical approximation of definite integrals. The rule is:
5
6  Integral of f(x) dx from a to b ~=
7  [f(a) + 4*f((a + b)/2) + f(b)] * (b - a)/6
8
9  */
10
11  #include <cmath> /* fabs() */
12
13  template<class DoubleFunction>
14  double simpsons(DoubleFunction f, double a, double b) {
15      return (f(a) + 4 * f((a + b)/2) + f(b)) * (b - a)/6;
16  }
17
18  template<class DoubleFunction>
19  double integrate(DoubleFunction f, double a, double b) {
20      static const double eps = 1e-10;
21      double m = (a + b) / 2;
22      double am = simpsons(f, a, m);

```

```
23  double mb = simpsons(f, m, b);
24  double ab = simpsons(f, a, b);
25  if (fabs(am + mb - ab) < eps) return ab;
26  return integrate(f, a, m) + integrate(f, m, b);
27 }
28
29 /** Example Usage */
30
31 #include <iostream>
32 using namespace std;
33
34 double f(double x) { return sin(x); }
35
36 int main () {
37     double PI = acos(-1.0);
38     cout << integrate(f, 0.0, PI/2) << "\n"; //1
39     return 0;
40 }
```

Chapter 5

Geometry

5.1 Geometric Classes

5.1.1 Point

```
1  /*
2
3  This class is very similar to std::complex, except it uses epsilon
4  comparisons and also supports other operations such as reflection
5  and rotation. In addition, this class supports many arithmetic
6  operations (e.g. overloaded operators for vector addition, subtraction,
7  multiplication, and division; dot/cross products, etc.) pertaining to
8  2D cartesian vectors.
9
10 All operations are O(1) in time and space.
11
12 */
13
14 #include <cmath>    /* atan(), fabs(), sqrt() */
15 #include <ostream>
16 #include <utility> /* std::pair */
17
18 const double eps = 1e-9;
19
20 #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
21 #define LT(a, b) ((a) < (b) - eps)      /* less than */
22
23 struct point {
24
25     double x, y;
26
27     point() : x(0), y(0) {}
28     point(const point & p) : x(p.x), y(p.y) {}
29     point(const std::pair<double, double> & p) : x(p.first), y(p.second) {}
30     point(const double & a, const double & b) : x(a), y(b) {}
31
32     bool operator < (const point & p) const {
33         return EQ(x, p.x) ? LT(y, p.y) : LT(x, p.x);
34     }
35 }
```

```

36  bool operator > (const point & p) const {
37      return EQ(x, p.x) ? LT(p.y, y) : LT(p.x, x);
38  }
39
40  bool operator == (const point & p) const { return EQ(x, p.x) && EQ(y, p.y); }
41  bool operator != (const point & p) const { return !(*this == p); }
42  bool operator <= (const point & p) const { return !(*this > p); }
43  bool operator >= (const point & p) const { return !(*this < p); }
44  point operator + (const point & p) const { return point(x + p.x, y + p.y); }
45  point operator - (const point & p) const { return point(x - p.x, y - p.y); }
46  point operator + (const double & v) const { return point(x + v, y + v); }
47  point operator - (const double & v) const { return point(x - v, y - v); }
48  point operator * (const double & v) const { return point(x * v, y * v); }
49  point operator / (const double & v) const { return point(x / v, y / v); }
50  point & operator += (const point & p) { x += p.x; y += p.y; return *this; }
51  point & operator -= (const point & p) { x -= p.x; y -= p.y; return *this; }
52  point & operator += (const double & v) { x += v; y += v; return *this; }
53  point & operator -= (const double & v) { x -= v; y -= v; return *this; }
54  point & operator *= (const double & v) { x *= v; y *= v; return *this; }
55  point & operator /= (const double & v) { x /= v; y /= v; return *this; }
56  friend point operator + (const double & v, const point & p) { return p + v; }
57  friend point operator * (const double & v, const point & p) { return p * v; }
58
59  double norm() const { return x * x + y * y; }
60  double abs() const { return sqrt(x * x + y * y); }
61  double arg() const { return atan2(y, x); }
62  double dot(const point & p) const { return x * p.x + y * p.y; }
63  double cross(const point & p) const { return x * p.y - y * p.x; }
64  double proj(const point & p) const { return dot(p) / p.abs(); } //onto p
65  point rot90() const { return point(-y, x); }
66
67  //proportional unit vector of (x, y) such that x^2 + y^2 = 1
68  point normalize() const {
69      return (EQ(x, 0) && EQ(y, 0)) ? point(0, 0) : (point(x, y) / abs());
70  }
71
72  //rotate t radians CW about origin
73  point rotateCW(const double & t) const {
74      return point(x * cos(t) + y * sin(t), y * cos(t) - x * sin(t));
75  }
76
77  //rotate t radians CCW about origin
78  point rotateCCW(const double & t) const {
79      return point(x * cos(t) - y * sin(t), x * sin(t) + y * cos(t));
80  }
81
82  //rotate t radians CW about point p
83  point rotateCW(const point & p, const double & t) const {
84      return (*this - p).rotateCW(t) + p;
85  }
86
87  //rotate t radians CCW about point p
88  point rotateCCW(const point & p, const double & t) const {
89      return (*this - p).rotateCCW(t) + p;
90  }
91
92  //reflect across point p
93  point reflect(const point & p) const {
94      return point(2 * p.x - x, 2 * p.y - y);

```

```

95     }
96
97     //reflect across the line containing points p and q
98     point reflect(const point & p, const point & q) const {
99         if (p == q) return reflect(p);
100         point r(*this - p), s = q - p;
101         r = point(r.x * s.x + r.y * s.y, r.x * s.y - r.y * s.x) / s.norm();
102         r = point(r.x * s.x - r.y * s.y, r.x * s.y + r.y * s.x) + p;
103         return r;
104     }
105
106     friend double norm(const point & p) { return p.norm(); }
107     friend double abs(const point & p) { return p.abs(); }
108     friend double arg(const point & p) { return p.arg(); }
109     friend double dot(const point & p, const point & q) { return p.dot(q); }
110     friend double cross(const point & p, const point & q) { return p.cross(q); }
111     friend double proj(const point & p, const point & q) { return p.proj(q); }
112     friend point rot90(const point & p) { return p.rot90(); }
113     friend point normalize(const point & p) { return p.normalize(); }
114     friend point rotateCW(const point & p, const double & t) { return p.rotateCW(t); }
115     friend point rotateCCW(const point & p, const double & t) { return p.rotateCCW(t); }
116     friend point rotateCW(const point & p, const point & q, const double & t) { return p.rotateCW(q, t); }
117     friend point rotateCCW(const point & p, const point & q, const double & t) { return p.rotateCCW(q, t); }
118
119     friend point reflect(const point & p, const point & q) { return p.reflect(q); }
120     friend point reflect(const point & p, const point & a, const point & b) { return p.reflect(a, b); }
121
122     friend std::ostream & operator << (std::ostream & out, const point & p) {
123         out << "(";
124         out << (fabs(p.x) < eps ? 0 : p.x) << ",";
125         out << (fabs(p.y) < eps ? 0 : p.y) << ")";
126         return out;
127     }
128 };
129
130 /** Example Usage */
131 #include <cassert>
132 #define pt point
133
134 const double PI = acos(-1.0);
135
136 int main() {
137     pt p(-10, 3);
138     assert(pt(-18, 29) == p + pt(-3, 9) * 6 / 2 - pt(-1, 1));
139     assert(EQ(109, p.norm()));
140     assert(EQ(10.44030650891, p.abs()));
141     assert(EQ(2.850135859112, p.arg()));
142     assert(EQ(0, p.dot(pt(3, 10))));
143     assert(EQ(0, p.cross(pt(10, -3))));
144     assert(EQ(10, p.proj(pt(-10, 0))));
145     assert(EQ(1, p.normalize().abs()));
146     assert(pt(-3, -10) == p.rot90());
147     assert(pt(3, 12) == p.rotateCW(pt(1, 1), PI / 2));
148     assert(pt(1, -10) == p.rotateCCW(pt(2, 2), PI / 2));
149     assert(pt(10, -3) == p.reflect(pt(0, 0)));
150     assert(pt(-10, -3) == p.reflect(pt(-2, 0), pt(5, 0)));
151     return 0;
152 }

```

5.1.2 Line

```

1  /*
2
3  A 2D line is expressed in the form  $Ax + By + C = 0$ . All lines can be
4  "normalized" to a canonical form by insisting that the y-coefficient
5  equal 1 if it is non-zero. Otherwise, we set the x-coefficient to 1.
6  If B is non-zero, then we have the common case where the slope = -A
7  after normalization.
8
9  All operations are  $O(1)$  in time and space.
10
11 */
12
13 #include <cmath>    /* fabs() */
14 #include <limits>   /* std::numeric_limits */
15 #include <ostream>
16 #include <utility> /* std::pair */
17
18 const double eps = 1e-9, NaN = std::numeric_limits<double>::quiet_NaN();
19
20 #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
21 #define LT(a, b) ((a) < (b) - eps)       /* less than */
22
23 typedef std::pair<double, double> point;
24 #define x first
25 #define y second
26
27 struct line {
28
29     double a, b, c;
30
31     line(): a(0), b(0), c(0) {} //invalid or uninitialized line
32
33     line(const double & A, const double & B, const double & C) {
34         a = A;
35         b = B;
36         c = C;
37         if (!EQ(b, 0)) {
38             a /= b; c /= b; b = 1;
39         } else {
40             c /= a; a = 1; b = 0;
41         }
42     }
43
44     line(const double & slope, const point & p) {
45         a = -slope;
46         b = 1;
47         c = slope * p.x - p.y;
48     }
49
50     line(const point & p, const point & q): a(0), b(0), c(0) {
51         if (EQ(p.x, q.x)) {
52             if (EQ(p.y, q.y)) return; //invalid line
53             //vertical line
54             a = 1;
55             b = 0;
56             c = -p.x;

```

```

57     return;
58 }
59 a = -(p.y - q.y) / (p.x - q.x);
60 b = 1;
61 c = -(a * p.x) - (b * p.y);
62 }
63
64 bool operator == (const line & l) const {
65     return EQ(a, l.a) && EQ(b, l.b) && EQ(c, l.c);
66 }
67
68 bool operator != (const line & l) const {
69     return !(*this == l);
70 }
71
72 //whether the line is initialized and normalized
73 bool valid() const {
74     if (EQ(a, 0)) return !EQ(b, 0);
75     return EQ(b, 1) || (EQ(b, 0) && EQ(a, 1));
76 }
77
78 bool horizontal() const { return valid() && EQ(a, 0); }
79 bool vertical() const { return valid() && EQ(b, 0); }
80
81 double slope() const {
82     if (!valid() || EQ(b, 0)) return NaN; //vertical
83     return -a;
84 }
85
86 //solve for x, given y
87 //for horizontal lines, either +inf, -inf, or nan is returned
88 double x(const double & y) const {
89     if (!valid() || EQ(a, 0)) return NaN; //invalid or horizontal
90     return (-c - b * y) / a;
91 }
92
93 //solve for y, given x
94 //for vertical lines, either +inf, -inf, or nan is returned
95 double y(const double & x) const {
96     if (!valid() || EQ(b, 0)) return NaN; //invalid or vertical
97     return (-c - a * x) / b;
98 }
99
100 //returns whether p exists on the line
101 bool contains(const point & p) const {
102     return EQ(a * p.x + b * p.y + c, 0);
103 }
104
105 //returns whether the line is parallel to l
106 bool parallel(const line & l) const {
107     return EQ(a, l.a) && EQ(b, l.b);
108 }
109
110 //returns whether the line is perpendicular to l
111 bool perpendicular(const line & l) const {
112     return EQ(-a * l.a, b * l.b);
113 }
114
115 //return the parallel line passing through point p

```



```

116     line parallel(const point & p) const {
117         return line(a, b, -a * p.x - b * p.y);
118     }
119
120     //return the perpendicular line passing through point p
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         out << (fabs(l.a) < eps ? 0 : l.a) << "x" << std::showpos;
127         out << (fabs(l.b) < eps ? 0 : l.b) << "y";
128         out << (fabs(l.c) < eps ? 0 : l.c) << "=0" << std::noshowpos;
129         return out;
130     }
131 };
132
133 /** Example Usage */
134
135 #include <cassert>
136
137 int main() {
138     line l(2, -5, -8);
139     line para = line(2, -5, -8).parallel(point(-6, -2));
140     line perp = line(2, -5, -8).perpendicular(point(-6, -2));
141     assert(l.parallel(para) && l.perpendicular(perp));
142     assert(l.slope() == 0.4);
143     assert(para == line(-0.4, 1, -0.4)); // -0.4x+1y-0.4=0
144     assert(perp == line(2.5, 1, 17));    // 2.5x+1y+17=0
145     return 0;
146 }

```

5.1.3 Circle

```

1  /*
2
3  A 2D circle with center at (h, k) and a radius of r can be expressed by
4  the relation (x - h)^2 + (y - k)^2 = r^2. In the following definition,
5  the radius used to construct it is forced to be a positive number.
6
7  All operations are O(1) in time and space.
8
9  */
10
11 #include <cmath>      /* fabs(), sqrt() */
12 #include <ostream>
13 #include <stdexcept> /* std::runtime_error() */
14 #include <utility>    /* std::pair */
15
16 const double eps = 1e-9;
17
18 #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
19 #define GT(a, b) ((a) > (b) + eps)        /* greater than */
20 #define LE(a, b) ((a) <= (b) + eps)        /* less than or equal to */
21
22 typedef std::pair<double, double> point;
23 #define x first

```

```

24 #define y second
25
26 double norm(const point & a) { return a.x * a.x + a.y * a.y; }
27 double abs(const point & a) { return sqrt(norm(a)); }
28
29 struct circle {
30
31     double h, k, r;
32
33     circle(): h(0), k(0), r(0) {}
34     circle(const double & R): h(0), k(0), r(fabs(R)) {}
35     circle(const point & o, const double & R): h(o.x), k(o.y), r(fabs(R)) {}
36     circle(const double & H, const double & K, const double & R):
37         h(H), k(K), r(fabs(R)) {}
38
39     //circumcircle with the diameter equal to the distance from a to b
40     circle(const point & a, const point & b) {
41         h = (a.x + b.x) / 2.0;
42         k = (a.y + b.y) / 2.0;
43         r = abs(point(a.x - h, a.y - k));
44     }
45
46     //circumcircle of 3 points - throws exception if abc are collinear/equal
47     circle(const point & a, const point & b, const point & c) {
48         double an = norm(point(b.x - c.x, b.y - c.y));
49         double bn = norm(point(a.x - c.x, a.y - c.y));
50         double cn = norm(point(a.x - b.x, a.y - b.y));
51         double wa = an * (bn + cn - an);
52         double wb = bn * (an + cn - bn);
53         double wc = cn * (an + bn - cn);
54         double w = wa + wb + wc;
55         if (fabs(w) < eps)
56             throw std::runtime_error("No_circle_from_collinear_points.");
57         h = (wa * a.x + wb * b.x + wc * c.x) / w;
58         k = (wa * a.y + wb * b.y + wc * c.y) / w;
59         r = abs(point(a.x - h, a.y - k));
60     }
61
62     //circle from 2 points and a radius - many possible edge cases!
63     //in the "normal" case, there will be 2 possible circles, one
64     //centered at (h1, k1) and the other (h2, k2). Only one is used.
65     //note that (h1, k1) equals (h2, k2) if dist(a, b) = 2 * r = d
66     circle(const point & a, const point & b, const double & R) {
67         r = fabs(R);
68         if (LE(r, 0) && a == b) { //circle is a point
69             h = a.x;
70             k = a.y;
71             return;
72         }
73         double d = abs(point(b.x - a.x, b.y - a.y));
74         if (EQ(d, 0))
75             throw std::runtime_error("Identical_points,_infinite_circles.");
76         if (GT(d, r * 2.0))
77             throw std::runtime_error("Points_too_far_away_to_make_circle.");
78         double v = sqrt(r * r - d * d / 4.0) / d;
79         point m((a.x + b.x) / 2.0, (a.y + b.y) / 2.0);
80         h = m.x + (a.y - b.y) * v;
81         k = m.y + (b.x - a.x) * v;
82         //other answer is (h, k) = (m.x-(a.y-b.y)*v, m.y-(b.x-a.x)*v)

```

```

83     }
84
85     bool operator == (const circle & c) const {
86         return EQ(h, c.h) && EQ(k, c.k) && EQ(r, c.r);
87     }
88
89     bool operator != (const circle & c) const {
90         return !(*this == c);
91     }
92
93     bool contains(const point & p) const {
94         return LE(norm(point(p.x - h, p.y - k)), r * r);
95     }
96
97     bool on_edge(const point & p) const {
98         return EQ(norm(point(p.x - h, p.y - k)), r * r);
99     }
100
101     point center() const {
102         return point(h, k);
103     }
104
105     friend std::ostream & operator << (std::ostream & out, const circle & c) {
106         out << std::showpos;
107         out << "(x" << -(fabs(c.h) < eps ? 0 : c.h) << ")^2+";
108         out << "(y" << -(fabs(c.k) < eps ? 0 : c.k) << ")^2";
109         out << std::noshowpos;
110         out << " = " << (fabs(c.r) < eps ? 0 : c.r * c.r);
111         return out;
112     }
113 };
114
115 //circle inscribed within points a, b, and c
116 circle incircle(const point & a, const point & b, const point & c) {
117     double a1 = abs(point(b.x - c.x, b.y - c.y));
118     double b1 = abs(point(a.x - c.x, a.y - c.y));
119     double c1 = abs(point(a.x - b.x, a.y - b.y));
120     double p = a1 + b1 + c1;
121     if (EQ(p, 0)) return circle(a.x, a.y, 0);
122     circle res;
123     res.h = (a1 * a.x + b1 * b.x + c1 * c.x) / p;
124     res.k = (a1 * a.y + b1 * b.y + c1 * c.y) / p;
125     res.r = fabs((a.x - c.x) * (b.y - c.y) - (a.y - c.y) * (b.x - c.x)) / p;
126     return res;
127 }
128
129 /** Example Usage */
130
131 #include <cassert>
132
133 int main() {
134     circle c(-2, 5, sqrt(10)); //(x+2)^2+(y-5)^2=10
135     assert(c == circle(point(-2, 5), sqrt(10)));
136     assert(c == circle(point(1, 6), point(-5, 4)));
137     assert(c == circle(point(-3, 2), point(-3, 8), point(-1, 8)));
138     assert(c == incircle(point(-12, 5), point(3, 0), point(0, 9)));
139     assert(c.contains(point(-2, 8)) && !c.contains(point(-2, 9)));
140     assert(c.on_edge(point(-1, 2)) && !c.on_edge(point(-1.01, 2)));
141     return 0;

```

142 }

5.2 Geometric Calculations

5.2.1 Angles

```

1  /*
2
3  Angle calculations in 2 dimensions. All returned angles are in radians,
4  except for reduce_deg(). If x is an angle in radians, then you may use
5  x * DEG to convert x to degrees, and vice versa to radians with x * RAD.
6
7  All operations are O(1) in time and space.
8
9  */
10
11 #include <cmath>      /* acos(), fabs(), sqrt(), atan2() */
12 #include <utility>    /* std::pair */
13
14 typedef std::pair<double, double> point;
15 #define x first
16 #define y second
17
18 const double PI = acos(-1.0), RAD = 180 / PI, DEG = PI / 180;
19
20 double abs(const point & a) { return sqrt(a.x * a.x + a.y * a.y); }
21
22 //reduce angles to the range [0, 360) degrees. e.g. reduce_deg(-630) = 90
23 double reduce_deg(const double & t) {
24     if (t < -360) return reduce_deg(fmod(t, 360));
25     if (t < 0) return t + 360;
26     return t >= 360 ? fmod(t, 360) : t;
27 }
28
29 //reduce angles to the range [0, 2*pi) radians. e.g. reduce_rad(720.5) = 0.5
30 double reduce_rad(const double & t) {
31     if (t < -2 * PI) return reduce_rad(fmod(t, 2 * PI));
32     if (t < 0) return t + 2 * PI;
33     return t >= 2 * PI ? fmod(t, 2 * PI) : t;
34 }
35
36 //like std::polar(), but returns a point instead of an std::complex
37 point polar_point(const double & r, const double & theta) {
38     return point(r * cos(theta), r * sin(theta));
39 }
40
41 //angle of segment (0, 0) to p, relative (CCW) to the +ve x-axis in radians
42 double polar_angle(const point & p) {
43     double t = atan2(p.y, p.x);
44     return t < 0 ? t + 2 * PI : t;
45 }
46
47 //smallest angle formed by points aob (angle is at point o) in radians
48 double angle(const point & a, const point & o, const point & b) {
49     point u(o.x - a.x, o.y - a.y), v(o.x - b.x, o.y - b.y);
50     return acos((u.x * v.x + u.y * v.y) / (abs(u) * abs(v)));

```

```

51 }
52
53 //angle of line segment ab relative (CCW) to the +ve x-axis in radians
54 double angle_between(const point & a, const point & b) {
55     double t = atan2(a.x * b.y - a.y * b.x, a.x * b.x + a.y * b.y);
56     return t < 0 ? t + 2 * PI : t;
57 }
58
59 //Given the A, B values of two lines in Ax + By + C = 0 form, finds the
60 //minimum angle in radians between the two lines in the range [0, PI/2]
61 double angle_between(const double & a1, const double & b1,
62                     const double & a2, const double & b2) {
63     double t = atan2(a1 * b2 - a2 * b1, a1 * a2 + b1 * b2);
64     if (t < 0) t += PI; //force angle to be positive
65     if (t > PI / 2) t = PI - t; //force angle to be <= 90 degrees
66     return t;
67 }
68
69 //magnitude of the 3D cross product with Z component implicitly equal to 0
70 //the answer assumes the origin (0, 0) is instead shifted to point o.
71 //this is equal to 2x the signed area of the triangle from these 3 points.
72 double cross(const point & o, const point & a, const point & b) {
73     return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
74 }
75
76 //does the path a->o->b form:
77 // -1 ==> a left turn on the plane?
78 // 0 ==> a single straight line segment? (i.e. are a,o,b collinear?) or
79 // +1 ==> a right turn on the plane?
80 //warning: the order of parameters is a,o,b, and NOT o,a,b as in cross()
81 int turn(const point & a, const point & o, const point & b) {
82     double c = cross(o, a, b);
83     return c < 0 ? -1 : (c > 0 ? 1 : 0);
84 }
85
86 /** Example Usage */
87
88 #include <cassert>
89 #define pt point
90 #define EQ(a, b) (fabs((a) - (b)) <= 1e-9)
91
92 int main() {
93     assert(EQ(123, reduce_deg(-(8 * 360) + 123)));
94     assert(EQ(1.2345, reduce_rad(2 * PI * 8 + 1.2345)));
95     point p = polar_point(4, PI), q = polar_point(4, -PI / 2);
96     assert(EQ(p.x, -4) && EQ(p.y, 0));
97     assert(EQ(q.x, 0) && EQ(q.y, -4));
98     assert(EQ(45, polar_angle(pt(5, 5)) * RAD));
99     assert(EQ(135, polar_angle(pt(-4, 4)) * RAD));
100    assert(EQ(90, angle(pt(5, 0), pt(0, 5), pt(-5, 0)) * RAD));
101    assert(EQ(225, angle_between(pt(0, 5), pt(5, -5)) * RAD));
102    assert(EQ(90, angle_between(-1, 1, -1, -1) * RAD)); //y=x and y=-x
103    assert(-1 == cross(pt(0, 0), pt(0, 1), pt(1, 0)));
104    assert(+1 == turn(pt(0, 1), pt(0, 0), pt(-5, -5)));
105    return 0;
106 }

```

5.2.2 Distances

```

1  /*
2
3  Distance calculations in 2 dimensions between points, lines, and segments.
4  All operations are O(1) in time and space.
5
6  */
7
8  #include <algorithm> /* std::max(), std::min() */
9  #include <cmath>     /* fabs(), sqrt() */
10 #include <utility>    /* std::pair */
11
12 typedef std::pair<double, double> point;
13 #define x first
14 #define y second
15
16 const double eps = 1e-9;
17
18 #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
19 #define LE(a, b) ((a) <= (b) + eps)      /* less than or equal to */
20 #define GE(a, b) ((a) >= (b) - eps)      /* greater than or equal to */
21
22 double norm(const point & a) { return a.x * a.x + a.y * a.y; }
23 double abs(const point & a) { return sqrt(norm(a)); }
24
25 //distance from point a to point b
26 double dist(const point & a, const point & b) {
27     return abs(point(b.x - a.x, b.y - a.y));
28 }
29
30 //squared distance from point a to point b
31 double dist2(const point & a, const point & b) {
32     return norm(point(b.x - a.x, b.y - a.y));
33 }
34
35 //minimum distance from point p to line l denoted by ax + by + c = 0
36 //if a = b = 0, then -inf, nan, or +inf is returned depending on sgn(c)
37 double dist_line(const point & p,
38                 const double & a, const double & b, const double & c) {
39     return fabs(a * p.x + b * p.y + c) / sqrt(a * a + b * b);
40 }
41
42 //minimum distance from point p to the infinite line containing a and b
43 //if a = b, then the point distance from p to the single point is returned
44 double dist_line(const point & p, const point & a, const point & b) {
45     double ab2 = dist2(a, b);
46     if (EQ(ab2, 0)) return dist(p, a);
47     double u = ((p.x - a.x) * (b.x - a.x) + (p.y - a.y) * (b.y - a.y)) / ab2;
48     return abs(point(a.x + u * (b.x - a.x) - p.x, a.y + u * (b.y - a.y) - p.y));
49 }
50
51 //distance between two lines each denoted by the form ax + by + c = 0
52 //if the lines are nonparallel, then the distance is 0, otherwise
53 //it is the perpendicular distance from a point on one line to the other
54 double dist_lines(const double & a1, const double & b1, const double & c1,
55                  const double & a2, const double & b2, const double & c2) {
56     if (EQ(a1 * b2, a2 * b1)) {

```

```

57     double factor = EQ(b1, 0) ? (a1 / a2) : (b1 / b2);
58     if (EQ(c1, c2 * factor)) return 0;
59     return fabs(c2 * factor - c1) / sqrt(a1 * a1 + b1 * b1);
60 }
61 return 0;
62 }
63
64 //distance between two infinite lines respectively containing ab and cd
65 //same results as above, except we solve for the lines here first.
66 double dist_lines(const point & a, const point & b,
67                  const point & c, const point & d) {
68     double A1 = a.y - b.y, B1 = b.x - a.x;
69     double A2 = c.y - d.y, B2 = d.x - c.x;
70     double C1 = -A1 * a.x - B1 * a.y, C2 = -A2 * c.x - B2 * c.y;
71     return dist_lines(A1, B1, C1, A2, B2, C2);
72 }
73
74 //minimum distance from point p to any point on segment ab
75 double dist_seg(const point & p, const point & a, const point & b) {
76     if (a == b) return dist(p, a);
77     point ab(b.x - a.x, b.y - a.y), ap(p.x - a.x, p.y - a.y);
78     double n = norm(ab), d = ab.x * ap.x + ab.y * ap.y;
79     if (LE(d, 0) || EQ(n, 0)) return abs(ap);
80     if (GE(d, n)) return abs(point(ap.x - ab.x, ap.y - ab.y));
81     return abs(point(ap.x - ab.x * (d / n), ap.y - ab.y * (d / n)));
82 }
83
84 double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
85 double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }
86
87 //minimum distance from any point on segment ab to any point on segment cd
88 double dist_segs(const point & a, const point & b,
89                 const point & c, const point & d) {
90     //check if segments are touching or intersecting - if so, distance is 0
91     point ab(b.x - a.x, b.y - a.y);
92     point ac(c.x - a.x, c.y - a.y);
93     point cd(d.x - c.x, d.y - c.y);
94     double c1 = cross(ab, cd), c2 = cross(ac, ab);
95     if (EQ(c1, 0) && EQ(c2, 0)) {
96         double t0 = dot(ac, ab) / norm(ab);
97         double t1 = t0 + dot(cd, ab) / norm(ab);
98         if (LE(std::min(t0, t1), 1) && LE(0, std::max(t0, t1)))
99             return 0;
100     } else {
101         double t = cross(ac, cd) / c1, u = c2 / c1;
102         if (!EQ(c1, 0) && LE(0, t) && LE(t, 1) && LE(0, u) && LE(u, 1))
103             return 0;
104     }
105     //find min distances across each endpoint to opposing segment
106     return std::min(std::min(dist_seg(a, c, d), dist_seg(b, c, d)),
107                    std::min(dist_seg(c, a, b), dist_seg(d, a, b)));
108 }
109
110 /** Example Usage */
111
112 #include <cassert>
113 #define pt point
114
115 int main() {

```

```

116  assert(EQ(5, dist(pt(-1, -1), pt(2, 3))));
117  assert(EQ(25, dist2(pt(-1, -1), pt(2, 3))));
118  assert(EQ(1.2, dist_line(pt(2, 1), -4, 3, -1)));
119  assert(EQ(0.8, dist_line(pt(3, 3), pt(-1, -1), pt(2, 3))));
120  assert(EQ(1.2, dist_line(pt(2, 1), pt(-1, -1), pt(2, 3))));
121  assert(EQ(0.0, dist_lines(-4, 3, -1, 8, 6, 2)));
122  assert(EQ(0.8, dist_lines(-4, 3, -1, -8, 6, -10)));
123  assert(EQ(1.0, dist_seg(pt(3, 3), pt(-1, -1), pt(2, 3))));
124  assert(EQ(1.2, dist_seg(pt(2, 1), pt(-1, -1), pt(2, 3))));
125  assert(EQ(0.0, dist_segs(pt(0, 2), pt(3, 3), pt(-1, -1), pt(2, 3))));
126  assert(EQ(0.6, dist_segs(pt(-1, 0), pt(-2, 2), pt(-1, -1), pt(2, 3))));
127  return 0;
128 }

```

5.2.3 Line Intersections

```

1  /*
2
3  Intersections between straight lines, as well as between line segments
4  in 2 dimensions. Also included are functions to determine the closest
5  point to a line, which is done by finding the intersection through the
6  perpendicular. Note that you should modify the TOUCH_IS_INTERSECT flag
7  used for line segment intersection, depending on whether you wish for
8  the algorithm to consider barely touching segments to intersect.
9
10 All operations are O(1) in time and space.
11
12 */
13
14 #include <algorithm> /* std::min(), std::max() */
15 #include <cmath>     /* fabs(), sqrt() */
16 #include <utility>   /* std::pair */
17
18 typedef std::pair<double, double> point;
19 #define x first
20 #define y second
21
22 const double eps = 1e-9;
23
24 #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
25 #define LT(a, b) ((a) < (b) - eps)       /* less than */
26 #define LE(a, b) ((a) <= (b) + eps)      /* less than or equal to */
27
28 //intersection of line l1 and line l2, each in ax + by + c = 0 form
29 //returns: -1, if lines do not intersect,
30 //         0, if there is exactly one intersection point, or
31 //         +1, if there are infinite intersection
32 //in the 2nd case, the intersection point is optionally stored into p
33 int line_intersection(const double & a1, const double & b1, const double & c1,
34                     const double & a2, const double & b2, const double & c2,
35                     point * p = 0) {
36     if (EQ(a1 * b2, a2 * b1))
37         return (EQ(a1 * c2, a2 * c1) || EQ(b1 * c2, b2 * c1)) ? 1 : -1;
38     if (p != 0) {
39         p->x = (b1 * c1 - b1 * c2) / (a2 * b1 - a1 * b2);
40         if (!EQ(b1, 0)) p->y = -(a1 * p->x + c1) / b1;
41         else p->y = -(a2 * p->x + c2) / b2;

```



```

42     }
43     return 0;
44 }
45
46 //intersection of line through p1, p2, and line through p2, p3
47 //returns: -1, if lines do not intersect,
48 //          0, if there is exactly one intersection point, or
49 //          +1, if there are infinite intersections
50 //in the 2nd case, the intersection point is optionally stored into p
51 int line_intersection(const point & p1, const point & p2,
52                     const point & p3, const point & p4, point * p = 0) {
53     double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
54     double c1 = -(p1.x * p2.y - p2.x * p1.y);
55     double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
56     double c2 = -(p3.x * p4.y - p4.x * p3.y);
57     double x = -(c1 * b2 - c2 * b1), y = -(a1 * c2 - a2 * c1);
58     double det = a1 * b2 - a2 * b1;
59     if (EQ(det, 0))
60         return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
61     if (p != 0) *p = point(x / det, y / det);
62     return 0;
63 }
64
65 //Line Segment Intersection (http://stackoverflow.com/a/565282)
66
67 double norm(const point & a) { return a.x * a.x + a.y * a.y; }
68 double abs(const point & a) { return sqrt(norm(a)); }
69 double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
70 double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }
71
72 //should we consider barely touching segments an intersection?
73 const bool TOUCH_IS_INTERSECT = true;
74
75 //does [l, h] contain m?
76 //precondition: l <= h
77 bool contain(const double & l, const double & m, const double & h) {
78     if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
79     return LT(l, m) && LT(m, h);
80 }
81
82 //does [l1, h1] overlap with [l2, h2]?
83 //precondition: l1 <= h1 and l2 <= h2
84 bool overlap(const double & l1, const double & h1,
85             const double & l2, const double & h2) {
86     if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
87     return LT(l1, h2) && LT(l2, h1);
88 }
89
90 //intersection of line segment ab with line segment cd
91 //returns: -1, if segments do not intersect,
92 //          0, if there is exactly one intersection point
93 //          +1, if the intersection is another line segment
94 //In case 2, the intersection point is stored into p
95 //In case 3, the intersection segment is stored into p and q
96 int seg_intersection(const point & a, const point & b,
97                    const point & c, const point & d,
98                    point * p = 0, point * q = 0) {
99     point ab(b.x - a.x, b.y - a.y);
100    point ac(c.x - a.x, c.y - a.y);

```

```

101 point cd(d.x - c.x, d.y - c.y);
102 double c1 = cross(ab, cd), c2 = cross(ac, ab);
103 if (EQ(c1, 0) && EQ(c2, 0)) { //collinear
104     double t0 = dot(ac, ab) / norm(ab);
105     double t1 = t0 + dot(cd, ab) / norm(ab);
106     if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
107         point res1 = std::max(std::min(a, b), std::min(c, d));
108         point res2 = std::min(std::max(a, b), std::max(c, d));
109         if (res1 == res2) {
110             if (p != 0) *p = res1;
111             return 0; //collinear, meeting at an endpoint
112         }
113         if (p != 0 && q != 0) *p = res1, *q = res2;
114         return 1; //collinear and overlapping
115     } else {
116         return -1; //collinear and disjoint
117     }
118 }
119 if (EQ(c1, 0)) return -1; //parallel and disjoint
120 double t = cross(ac, cd) / c1, u = c2 / c1;
121 if (contain(0, t, 1) && contain(0, u, 1)) {
122     if (p != 0) *p = point(a.x + t * ab.x, a.y + t * ab.y);
123     return 0; //non-parallel with one intersection
124 }
125 return -1; //non-parallel with no intersections
126 }
127
128 //determines the point on line ax + by + c = 0 that is closest to point p
129 //this always lies on the line through p perpendicular to l.
130 point closest_point(const double & a, const double & b, const double & c,
131                     const point & p) {
132     if (EQ(a, 0)) return point(p.x, -c); //horizontal line
133     if (EQ(b, 0)) return point(-c, p.y); //vertical line
134     point res;
135     line_intersection(a, b, c, -b, a, b * p.x - a * p.y, &res);
136     return res;
137 }
138
139 //determines the point on segment ab closest to point p
140 point closest_point(const point & a, const point & b, const point & p) {
141     if (a == b) return a;
142     point ap(p.x - a.x, p.y - a.y), ab(b.x - a.x, b.y - a.y);
143     double t = dot(ap, ab) / norm(ab);
144     if (t <= 0) return a;
145     if (t >= 1) return b;
146     return point(a.x + t * ab.x, a.y + t * ab.y);
147 }
148
149 /** Example Usage */
150
151 #include <cassert>
152 #define pt point
153
154 int main() {
155     point p;
156     assert(line_intersection(-1, 1, 0, 1, 1, -3, &p) == 0);
157     assert(p == pt(1.5, 1.5));
158     assert(line_intersection(pt(0, 0), pt(1, 1), pt(0, 4), pt(4, 0), &p) == 0);
159     assert(p == pt(2, 2));

```

```

160
161 //tests for segment intersection (examples in order from link below)
162 //http://martin-thoma.com/how-to-check-if-two-line-segments-intersect/
163 {
164 #define test(a,b,c,d,e,f,g,h) seg_intersection(pt(a,b),pt(c,d),pt(e,f),pt(g,h),&p,&q)
165     pt p, q;
166     //intersection is a point
167     assert(0 == test(-4, 0, 4, 0, 0, -4, 0, 4));    assert(p == pt(0, 0));
168     assert(0 == test(0, 0, 10, 10, 2, 2, 16, 4));    assert(p == pt(2, 2));
169     assert(0 == test(-2, 2, -2, -2, -2, 0, 0, 0));    assert(p == pt(-2, 0));
170     assert(0 == test(0, 4, 4, 4, 4, 0, 4, 8));        assert(p == pt(4, 4));
171
172     //intersection is a segment
173     assert(1 == test(10, 10, 0, 0, 2, 2, 6, 6));
174     assert(p == pt(2, 2) && q == pt(6, 6));
175     assert(1 == test(6, 8, 14, -2, 14, -2, 6, 8));
176     assert(p == pt(6, 8) && q == pt(14, -2));
177
178     //no intersection
179     assert(-1 == test(6, 8, 8, 10, 12, 12, 4, 4));
180     assert(-1 == test(-4, 2, -8, 8, 0, 0, -4, 6));
181     assert(-1 == test(4, 4, 4, 6, 0, 2, 0, 0));
182     assert(-1 == test(4, 4, 6, 4, 0, 2, 0, 0));
183     assert(-1 == test(-2, -2, 4, 4, 10, 10, 6, 6));
184     assert(-1 == test(0, 0, 2, 2, 4, 0, 1, 4));
185     assert(-1 == test(2, 2, 2, 8, 4, 4, 6, 4));
186     assert(-1 == test(4, 2, 4, 4, 0, 8, 10, 0));
187 }
188 assert(pt(2.5, 2.5) == closest_point(-1, -1, 5, pt(0, 0)));
189 assert(pt(3, 0) == closest_point(1, 0, -3, pt(0, 0)));
190 assert(pt(0, 3) == closest_point(0, 1, -3, pt(0, 0)));
191
192 assert(pt(3, 0) == closest_point(pt(3, 0), pt(3, 3), pt(0, 0)));
193 assert(pt(2, -1) == closest_point(pt(2, -1), pt(4, -1), pt(0, 0)));
194 assert(pt(4, -1) == closest_point(pt(2, -1), pt(4, -1), pt(5, 0)));
195 return 0;
196 }

```

5.2.4 Circle Intersections

```

1  /*
2
3  Tangent lines to circles, circle-line intersections, and circle-circle
4  intersections (intersection point(s) as well as area) in 2 dimensions.
5
6  All operations are O(1) in time and space.
7
8  */
9
10 #include <algorithm> /* std::min(), std::max() */
11 #include <cmath>     /* acos(), fabs(), sqrt() */
12 #include <utility>   /* std::pair */
13
14 typedef std::pair<double, double> point;
15 #define x first
16 #define y second
17

```

```

18  const double eps = 1e-9;
19
20  #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
21  #define NE(a, b) (fabs((a) - (b)) > eps) /* not equal to */
22  #define LT(a, b) ((a) < (b) - eps)      /* less than */
23  #define GT(a, b) ((a) > (b) + eps)      /* greater than */
24  #define LE(a, b) ((a) <= (b) + eps)     /* less than or equal to */
25  #define GE(a, b) ((a) >= (b) - eps)     /* greater than or equal to */
26
27  struct circle {
28      double h, k, r;
29
30      circle(const double & h, const double & k, const double & r) {
31          this->h = h;
32          this->k = k;
33          this->r = r;
34      }
35  };
36
37  //note: this is a simplified version of line that is not canonicalized.
38  // e.g. comparing lines with == signs will not work as intended. For a
39  //      fully featured line class, see the whole geometry library.
40  struct line {
41      double a, b, c;
42
43      line() { a = b = c = 0; }
44
45      line(const double & a, const double & b, const double & c) {
46          this->a = a;
47          this->b = b;
48          this->c = c;
49      }
50
51      line(const point & p, const point & q) {
52          a = p.y - q.y,
53          b = q.x - p.x;
54          c = -a * p.x - b * p.y;
55      }
56  };
57
58  double norm(const point & a) { return a.x * a.x + a.y * a.y; }
59  double abs(const point & a) { return sqrt(norm(a)); }
60  double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
61
62  //tangent line(s) to circle c passing through p. there are 3 cases:
63  //returns: 0, if there are no lines (p is strictly inside c)
64  //         1, if there is 1 tangent line (p is on the edge)
65  //         2, if there are 2 tangent lines (p is strictly outside)
66  //If there is only 1 tangent, then the line will be stored in l1.
67  //If there are 2, then they will be stored in l1 and l2 respectively.
68  int tangents(const circle & c, const point & p, line * l1 = 0, line * l2 = 0) {
69      point vop(p.x - c.h, p.y - c.k);
70      if (EQ(norm(vop), c.r * c.r)) { //on an edge, get perpendicular through p
71          if (l1 != 0) {
72              *l1 = line(point(c.h, c.k), p);
73              *l1 = line(-l1->b, l1->a, l1->b * p.x - l1->a * p.y);
74          }
75          return 1;
76      }

```

```

77  if (LE(norm(vop), c.r * c.r)) return 0; //inside circle
78  point q(vop.x / c.r, vop.y / c.r);
79  double n = norm(q), d = q.y * sqrt(norm(q) - 1.0);
80  point t1((q.x - d) / n, c.k), t2((q.x + d) / n, c.k);
81  if (NE(q.y, 0)) { //common case
82      t1.y += c.r * (1.0 - t1.x * q.x) / q.y;
83      t2.y += c.r * (1.0 - t2.x * q.x) / q.y;
84  } else { //point at center horizontal, y = 0
85      d = c.r * sqrt(1.0 - t1.x * t1.x);
86      t1.y += d;
87      t2.y -= d;
88  }
89  t1.x = t1.x * c.r + c.h;
90  t2.x = t2.x * c.r + c.h;
91  //note: here, t1 and t2 are the two points of tangencies
92  if (l1 != 0) *l1 = line(p, t1);
93  if (l2 != 0) *l2 = line(p, t2);
94  return 2;
95 }
96
97 //determines the intersection(s) between a circle c and line l
98 //returns: 0, if the line does not intersect with the circle
99 //          1, if the line is tangent (one intersection)
100 //          2, if the line crosses through the circle
101 //If there is 1 intersection point, it will be stored in p
102 //If there are 2, they will be stored in p and q respectively
103 int intersection(const circle & c, const line & l,
104                 point * p = 0, point * q = 0) {
105     double v = c.h * l.a + c.k * l.b + l.c;
106     double aabb = l.a * l.a + l.b * l.b;
107     double disc = v * v / aabb - c.r * c.r;
108     if (disc > eps) return 0;
109     double x0 = -l.a * l.c / aabb, y0 = -l.b * v / aabb;
110     if (disc > -eps) {
111         if (p != 0) *p = point(x0 + c.h, y0 + c.k);
112         return 1;
113     }
114     double k = sqrt((disc /= -aabb) < 0 ? 0 : disc);
115     if (p != 0) *p = point(x0 + k * l.b + c.h, y0 - k * l.a + c.k);
116     if (q != 0) *q = point(x0 - k * l.b + c.h, y0 + k * l.a + c.k);
117     return 2;
118 }
119
120 //determines the intersection points between two circles c1 and c2
121 //returns: -2, if circle c2 completely encloses circle c1
122 //          -1, if circle c1 completely encloses circle c2
123 //          0, if the circles are completely disjoint
124 //          1, if the circles are tangent (one intersection point)
125 //          2, if the circles intersect at two points
126 //          3, if the circles intersect at infinite points (c1 = c2)
127 //If one intersection, the intersection point is stored in p
128 //If two, the intersection points are stored in p and q respectively
129 int intersection(const circle & c1, const circle & c2,
130                 point * p = 0, point * q = 0) {
131     if (EQ(c1.h, c2.h) && EQ(c1.k, c2.k))
132         return EQ(c1.r, c2.r) ? 3 : (c1.r > c2.r ? -1 : -2);
133     point d12(point(c2.h - c1.h, c2.k - c1.k));
134     double d = abs(d12);
135     if (GT(d, c1.r + c2.r)) return 0;

```

```

136     if (LT(d, fabs(c1.r - c2.r))) return c1.r > c2.r ? -1 : -2;
137     double a = (c1.r * c1.r - c2.r * c2.r + d * d) / (2 * d);
138     double x0 = c1.h + (d12.x * a / d);
139     double y0 = c1.k + (d12.y * a / d);
140     double s = sqrt(c1.r * c1.r - a * a);
141     double rx = -d12.y * s / d, ry = d12.x * s / d;
142     if (EQ(rx, 0) && EQ(ry, 0)) {
143         if (p != 0) *p = point(x0, y0);
144         return 1;
145     }
146     if (p != 0) *p = point(x0 - rx, y0 - ry);
147     if (q != 0) *q = point(x0 + rx, y0 + ry);
148     return 2;
149 }
150
151 const double PI = acos(-1.0);
152
153 //intersection area of circles c1 and c2
154 double intersection_area(const circle & c1, const circle c2) {
155     double r = std::min(c1.r, c2.r), R = std::max(c1.r, c2.r);
156     double d = abs(point(c2.h - c1.h, c2.k - c1.k));
157     if (LE(d, R - r)) return PI * r * r;
158     if (GE(d, R + r)) return 0;
159     return r * r * acos((d * d + r * r - R * R) / 2 / d / r) +
160         R * R * acos((d * d + R * R - r * r) / 2 / d / R) -
161         0.5 * sqrt((-d + r + R) * (d + r - R) * (d - r + R) * (d + r + R));
162 }
163
164 /** Example Usage */
165
166 #include <cassert>
167 #include <iostream>
168 using namespace std;
169 #define pt point
170
171 int main() {
172     line l1, l2;
173     assert(0 == tangents(circle(0, 0, 4), pt(1, 1), &l1, &l2));
174     assert(1 == tangents(circle(0, 0, sqrt(2)), pt(1, 1), &l1, &l2));
175     cout << l1.a << " " << l1.b << " " << l1.c << "\n"; // -x - y + 2 = 0
176     assert(2 == tangents(circle(0, 0, 2), pt(2, 2), &l1, &l2));
177     cout << l1.a << " " << l1.b << " " << l1.c << "\n"; // -2y + 4 = 0
178     cout << l2.a << " " << l2.b << " " << l2.c << "\n"; // 2x - 4 = 0
179
180     pt p, q;
181     assert(0 == intersection(circle(1, 1, 3), line(5, 3, -30), &p, &q));
182     assert(1 == intersection(circle(1, 1, 3), line(0, 1, -4), &p, &q));
183     assert(p == pt(1, 4));
184     assert(2 == intersection(circle(1, 1, 3), line(0, 1, -1), &p, &q));
185     assert(p == pt(4, 1));
186     assert(q == pt(-2, 1));
187
188     assert(-2 == intersection(circle(1, 1, 1), circle(0, 0, 3), &p, &q));
189     assert(-1 == intersection(circle(0, 0, 3), circle(1, 1, 1), &p, &q));
190     assert(0 == intersection(circle(5, 0, 4), circle(-5, 0, 4), &p, &q));
191     assert(1 == intersection(circle(-5, 0, 5), circle(5, 0, 5), &p, &q));
192     assert(p == pt(0, 0));
193     assert(2 == intersection(circle(-0.5, 0, 1), circle(0.5, 0, 1), &p, &q));
194     assert(p == pt(0, -sqrt(3) / 2));

```

```

195     assert(q == pt(0, sqrt(3) / 2));
196
197     //example where each circle passes through the other circle's center
198     //http://math.stackexchange.com/a/402891
199     double r = 3;
200     double a = intersection_area(circle(-r / 2, 0, r), circle(r / 2, 0, r));
201     assert(EQ(a, r * r * (2 * PI / 3 - sqrt(3) / 2)));
202     return 0;
203 }

```

5.3 Common Geometric Computations

5.3.1 Polygon Sorting and Area

```

1  /*
2
3  centroid() - Simply returns the geometric average point of all the
4  points given. This could be used to find the reference center point
5  for the following function. An empty range will result in (0, 0).
6  Complexity: O(n) on the number of points in the given range.
7
8  cw_comp() - Given a set of points, these points could possibly form
9  many different polygons. The following sorting comparators, when
10 used in conjunction with std::sort, will produce one such ordering
11 of points which is sorted in clockwise order relative to a custom-
12 defined center point that must be set beforehand. This could very
13 well be the result of mean_point(). ccw_comp() is the opposite
14 function, which produces the points in counterclockwise order.
15 Complexity: O(1) per call.
16
17 polygon_area() - A given range of points is interpreted as a polygon
18 based on the ordering they're given in. The shoelace formula is used
19 to determine its area. The polygon does not necessarily have to be
20 sorted using one of the functions above, but may be any ordering that
21 produces a valid polygon. You may optionally pass the last point in
22 the range equal to the first point and still expect the correct result.
23 Complexity: O(n) on the number of points in the range, assuming that
24 the points are already sorted in the order that specifies the polygon.
25
26 */
27
28 #include <algorithm> /* std::sort() */
29 #include <cmath>     /* fabs() */
30 #include <utility>   /* std::pair */
31
32 typedef std::pair<double, double> point;
33 #define x first
34 #define y second
35
36 const double eps = 1e-9;
37
38 #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
39 #define LT(a, b) ((a) < (b) - eps)        /* less than */
40 #define GE(a, b) ((a) >= (b) - eps)       /* greater than or equal to */
41
42 //magnitude of the 3D cross product with Z component implicitly equal to 0

```

```

43 //the answer assumes the origin (0, 0) is instead shifted to point o.
44 //this is equal to 2x the signed area of the triangle from these 3 points.
45 double cross(const point & o, const point & a, const point & b) {
46     return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
47 }
48
49 point ctr;
50
51 template<class It> point centroid(It lo, It hi) {
52     if (lo == hi) return point(0, 0);
53     double xtot = 0, ytot = 0, points = hi - lo;
54     for (; lo != hi; ++lo) {
55         xtot += lo->x;
56         ytot += lo->y;
57     }
58     return point(xtot / points, ytot / points);
59 }
60
61 //ctr must be defined beforehand
62 bool cw_comp(const point & a, const point & b) {
63     if (GE(a.x - ctr.x, 0) && LT(b.x - ctr.x, 0)) return true;
64     if (LT(a.x - ctr.x, 0) && GE(b.x - ctr.x, 0)) return false;
65     if (EQ(a.x - ctr.x, 0) && EQ(b.x - ctr.x, 0)) {
66         if (GE(a.y - ctr.y, 0) || GE(b.y - ctr.y, 0))
67             return a.y > b.y;
68         return b.y > a.y;
69     }
70     double det = cross(ctr, a, b);
71     if (EQ(det, 0))
72         return (a.x - ctr.x) * (a.x - ctr.x) + (a.y - ctr.y) * (a.y - ctr.y) >
73             (b.x - ctr.x) * (b.x - ctr.x) + (b.y - ctr.y) * (b.y - ctr.y);
74     return det < 0;
75 }
76
77 bool ccw_comp(const point & a, const point & b) {
78     return cw_comp(b, a);
79 }
80
81 //area of a polygon specified by range [lo, hi) - shoelace formula in O(n)
82 //[[lo, hi) must point to the polygon vertices, sorted in CW or CCW order
83 template<class It> double polygon_area(It lo, It hi) {
84     if (lo == hi) return 0;
85     double area = 0;
86     if (*lo != *--hi)
87         area += (lo->x - hi->x) * (lo->y + hi->y);
88     for (It i = hi, j = hi - 1; i != lo; --i, --j)
89         area += (i->x - j->x) * (i->y + j->y);
90     return fabs(area / 2.0);
91 }
92
93 /** Example Usage */
94
95 #include <cassert>
96 #include <vector>
97 using namespace std;
98 #define pt point
99
100 int main() {
101     //irregular pentagon with only (1, 2) not on the convex hull

```



```

102 //the ordering here is already sorted in ccw order around their centroid
103 //we will scramble them and see if our comparator works
104 pt pts[] = {pt(1, 3), pt(1, 2), pt(2, 1), pt(0, 0), pt(-1, 3)};
105 vector<pt> v(pts, pts + 5);
106 std::random_shuffle(v.begin(), v.end());
107 ctr = centroid(v.begin(), v.end()); //note: ctr is a global variable
108 assert(EQ(ctr.x, 0.6) && EQ(ctr.y, 1.8));
109 sort(v.begin(), v.end(), cw_comp);
110 for (int i = 0; i < (int)v.size(); i++) assert(v[i] == pts[i]);
111 assert(EQ(polygon_area(v.begin(), v.end()), 5));
112 return 0;
113 }

```

5.3.2 Point in Polygon Query

```

1  /*
2
3  Given a single point p and another range of points specifying a
4  polygon, determine whether p lies within the polygon. Note that
5  you should modify the EDGE_IS_INSIDE flag, depending on whether
6  you wish for the algorithm to consider points lying on an edge of
7  the polygon to be inside it.
8
9  Complexity: O(n) on the number of vertices in the polygon.
10
11 */
12
13 #include <algorithm> /* std::sort() */
14 #include <cmath> /* fabs() */
15 #include <utility> /* std::pair */
16
17 typedef std::pair<double, double> point;
18 #define x first
19 #define y second
20
21 const double eps = 1e-9;
22
23 #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
24 #define GT(a, b) ((a) > (b) + eps) /* greater than */
25 #define LE(a, b) ((a) <= (b) + eps) /* less than or equal to */
26
27 //should we consider points lying on an edge to be inside the polygon?
28 const bool EDGE_IS_INSIDE = true;
29
30 //magnitude of the 3D cross product with Z component implicitly equal to 0
31 //the answer assumes the origin (0, 0) is instead shifted to point o.
32 //this is equal to 2x the signed area of the triangle from these 3 points.
33 double cross(const point & o, const point & a, const point & b) {
34     return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
35 }
36
37 //return whether point p is in polygon specified by range [lo, hi) in O(n)
38 //[[lo, hi) must point to the polygon vertices, sorted in CW or CCW order
39 template<class It> bool point_in_polygon(const point & p, It lo, It hi) {
40     int cnt = 0;
41     for (It i = lo, j = hi - 1; i != hi; j = i++) {
42         if (EQ(i->y, p.y) && (EQ(i->x, p.x) ||

```

```

43         (EQ(j->y, p.y) && (LE(i->x, p.x) || LE(j->x, p.x))))
44     return EDGE_IS_INSIDE; //on an edge
45     if (GT(i->y, p.y) != GT(j->y, p.y)) {
46         double det = cross(p, *i, *j);
47         if (EQ(det, 0)) return EDGE_IS_INSIDE; //on an edge
48         if (GT(det, 0) != GT(j->y, i->y)) cnt++;
49     }
50 }
51 return cnt % 2 == 1;
52 }
53
54 /** Example Usage */
55
56 #include <cassert>
57 using namespace std;
58 #define pt point
59
60 int main() {
61     //irregular trapezoid
62     pt p[] = {pt(-1, 3), pt(1, 3), pt(2, 1), pt(0, 0)};
63     assert(point_in_polygon(pt(1, 2), p, p + 4));
64     assert(point_in_polygon(pt(0, 3), p, p + 4));
65     assert(!point_in_polygon(pt(0, 3.01), p, p + 4));
66     assert(!point_in_polygon(pt(2, 2), p, p + 4));
67     return 0;
68 }

```

5.3.3 Convex Hull

```

1  /*
2
3  Determines the convex hull from a range of points, that is, the
4  smallest convex polygon (a polygon such that every line which
5  crosses through it will only cross through it once) that contains
6  all of the points. This function uses the monotone chain algorithm
7  to compute the upper and lower hulls separately.
8
9  Returns: a vector of the convex hull points in clockwise order.
10 Complexity:  $O(n \log n)$  on the number of points given
11
12 Notes: To yield the hull points in counterclockwise order,
13         replace every usage of GE() in the function with LE().
14         To have the first point on the hull repeated as the last,
15         replace the last line of the function to res.resize(k);
16
17 */
18
19 #include <algorithm> /* std::sort() */
20 #include <cmath>     /* fabs() */
21 #include <utility>   /* std::pair */
22 #include <vector>
23
24 typedef std::pair<double, double> point;
25 #define x first
26 #define y second
27
28 //change < 0 comparisons to > 0 to produce hull points in CCW order

```

```

29 double cw(const point & o, const point & a, const point & b) {
30     return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x) < 0;
31 }
32
33 //convex hull from a range [lo, hi) of points
34 //monotone chain in O(n log n) to find hull points in CW order
35 //notes: the range of input points will be sorted lexicographically
36 template<class It> std::vector<point> convex_hull(It lo, It hi) {
37     int k = 0;
38     if (hi - lo <= 1) return std::vector<point>(lo, hi);
39     std::vector<point> res(2 * (int)(hi - lo));
40     std::sort(lo, hi); //compare by x, then by y if x-values are equal
41     for (It it = lo; it != hi; ++it) {
42         while (k >= 2 && !cw(res[k - 2], res[k - 1], *it)) k--;
43         res[k++] = *it;
44     }
45     int t = k + 1;
46     for (It it = hi - 2; it != lo - 1; --it) {
47         while (k >= t && !cw(res[k - 2], res[k - 1], *it)) k--;
48         res[k++] = *it;
49     }
50     res.resize(k - 1);
51     return res;
52 }
53
54 /** Example Usage */
55
56 #include <iostream>
57 using namespace std;
58
59 int main() {
60     //irregular pentagon with only (1, 2) not on the convex hull
61     vector<point> v;
62     v.push_back(point(1, 3));
63     v.push_back(point(1, 2));
64     v.push_back(point(2, 1));
65     v.push_back(point(0, 0));
66     v.push_back(point(-1, 3));
67     std::random_shuffle(v.begin(), v.end());
68     vector<point> h = convex_hull(v.begin(), v.end());
69     cout << "hull points:";
70     for (int i = 0; i < (int)h.size(); i++)
71         cout << " (" << h[i].x << ", " << h[i].y << ")";
72     cout << "\n";
73     return 0;
74 }

```

5.3.4 Minimum Enclosing Circle

```

1  /*
2
3  Given a range of points on the 2D cartesian plane, determine
4  the equation of the circle with smallest possible area which
5  encloses all of the points. Note: in an attempt to avoid the
6  worst case, the circles are randomly shuffled before the
7  algorithm is performed. This is not necessary to obtain the
8  correct answer, and may be removed if the input order must

```

```

9  be preserved.
10
11 Time Complexity: O(n) average on the number of points given.
12
13 */
14
15 #include <algorithm>
16 #include <cmath>
17 #include <stdexcept>
18 #include <utility>
19
20 const double eps = 1e-9;
21
22 #define LE(a, b) ((a) <= (b) + eps)      /* less than or equal to */
23
24 typedef std::pair<double, double> point;
25 #define x first
26 #define y second
27
28 double norm(const point & a) { return a.x * a.x + a.y * a.y; }
29 double abs(const point & a) { return sqrt(norm(a)); }
30
31 struct circle {
32
33     double h, k, r;
34
35     circle(): h(0), k(0), r(0) {}
36     circle(const double & H, const double & K, const double & R):
37         h(H), k(K), r(fabs(R)) {}
38
39     //circumcircle with the diameter equal to the distance from a to b
40     circle(const point & a, const point & b) {
41         h = (a.x + b.x) / 2.0;
42         k = (a.y + b.y) / 2.0;
43         r = abs(point(a.x - h, a.y - k));
44     }
45
46     //circumcircle of 3 points - throws exception if abc are collinear/equal
47     circle(const point & a, const point & b, const point & c) {
48         double an = norm(point(b.x - c.x, b.y - c.y));
49         double bn = norm(point(a.x - c.x, a.y - c.y));
50         double cn = norm(point(a.x - b.x, a.y - b.y));
51         double wa = an * (bn + cn - an);
52         double wb = bn * (an + cn - bn);
53         double wc = cn * (an + bn - cn);
54         double w = wa + wb + wc;
55         if (fabs(w) < eps)
56             throw std::runtime_error("No_circle_from_collinear_points.");
57         h = (wa * a.x + wb * b.x + wc * c.x) / w;
58         k = (wa * a.y + wb * b.y + wc * c.y) / w;
59         r = abs(point(a.x - h, a.y - k));
60     }
61
62     bool contains(const point & p) const {
63         return LE(norm(point(p.x - h, p.y - k)), r * r);
64     }
65
66 };
67

```

```

68 template<class It> circle smallest_circle(It lo, It hi) {
69     if (lo == hi) return circle(0, 0, 0);
70     if (lo + 1 == hi) return circle(lo->x, lo->y, 0);
71     std::random_shuffle(lo, hi);
72     circle res(*lo, *(lo + 1));
73     for (It i = lo + 2; i != hi; ++i) {
74         if (res.contains(*i)) continue;
75         res = circle(*lo, *i);
76         for (It j = lo + 1; j != i; ++j) {
77             if (res.contains(*j)) continue;
78             res = circle(*i, *j);
79             for (It k = lo; k != j; ++k)
80                 if (!res.contains(*k)) res = circle(*i, *j, *k);
81         }
82     }
83     return res;
84 }
85
86 /** Example Usage */
87
88 #include <iostream>
89 #include <vector>
90 using namespace std;
91
92 int main() {
93     vector<point> v;
94     v.push_back(point(0, 0));
95     v.push_back(point(0, 1));
96     v.push_back(point(1, 0));
97     v.push_back(point(1, 1));
98     circle res = smallest_circle(v.begin(), v.end());
99     cout << "center:␣" << res.h << ",␣" << res.k << "␣\n";
100    cout << "radius:␣" << res.r << "\n";
101    return 0;
102 }

```

5.3.5 Diameter of Point Set

```

1  /*
2
3  Determines the diametral pair of a range of points. The diameter
4  of a set of points is the largest distance between any two
5  points in the set. A diametral pair is a pair of points in the
6  set whose distance is equal to the set's diameter. The following
7  program uses rotating calipers method to find a solution.
8
9  Time Complexity:  $O(n \log n)$  on the number of points in the set.
10
11 */
12
13 #include <algorithm> /* std::sort() */
14 #include <cmath>     /* fabs(), sqrt() */
15 #include <utility>   /* std::pair */
16 #include <vector>
17
18 typedef std::pair<double, double> point;
19 #define x first

```

```

20 #define y second
21
22 double sqdist(const point & a, const point & b) {
23     double dx = a.x - b.x, dy = a.y - b.y;
24     return sqrt(dx * dx + dy * dy);
25 }
26
27 double cross(const point & o, const point & a, const point & b) {
28     return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
29 }
30
31 bool cw(const point & o, const point & a, const point & b) {
32     return cross(o, a, b) < 0;
33 }
34
35 double area(const point & o, const point & a, const point & b) {
36     return fabs(cross(o, a, b));
37 }
38
39 template<class It> std::vector<point> convex_hull(It lo, It hi) {
40     int k = 0;
41     if (hi - lo <= 1) return std::vector<point>(lo, hi);
42     std::vector<point> res(2 * (int)(hi - lo));
43     std::sort(lo, hi); //compare by x, then by y if x-values are equal
44     for (It it = lo; it != hi; ++it) {
45         while (k >= 2 && !cw(res[k - 2], res[k - 1], *it)) k--;
46         res[k++] = *it;
47     }
48     int t = k + 1;
49     for (It it = hi - 2; it != lo - 1; --it) {
50         while (k >= t && !cw(res[k - 2], res[k - 1], *it)) k--;
51         res[k++] = *it;
52     }
53     res.resize(k - 1);
54     return res;
55 }
56
57 template<class It> std::pair<point, point> diametral_pair(It lo, It hi) {
58     std::vector<point> h = convex_hull(lo, hi);
59     int m = h.size();
60     if (m == 1) return std::make_pair(h[0], h[0]);
61     if (m == 2) return std::make_pair(h[0], h[1]);
62     int k = 1;
63     while (area(h[m - 1], h[0], h[(k + 1) % m]) > area(h[m - 1], h[0], h[k]))
64         k++;
65     double maxdist = 0, d;
66     std::pair<point, point> res;
67     for (int i = 0, j = k; i <= k && j < m; i++) {
68         d = sqdist(h[i], h[j]);
69         if (d > maxdist) {
70             maxdist = d;
71             res = std::make_pair(h[i], h[j]);
72         }
73         while (j < m && area(h[i], h[(i + 1) % m], h[(j + 1) % m]) >
74             area(h[i], h[(i + 1) % m], h[j])) {
75             d = sqdist(h[i], h[(j + 1) % m]);
76             if (d > maxdist) {
77                 maxdist = d;
78                 res = std::make_pair(h[i], h[(j + 1) % m]);

```

```

79     }
80     j++;
81 }
82 }
83 return res;
84 }
85
86 /** Example Usage */
87
88 #include <iostream>
89 using namespace std;
90
91 int main() {
92     vector<point> v;
93     v.push_back(point(0, 0));
94     v.push_back(point(3, 0));
95     v.push_back(point(0, 3));
96     v.push_back(point(1, 1));
97     v.push_back(point(4, 4));
98     pair<point, point> res = diametral_pair(v.begin(), v.end());
99     cout << "diametral_pair: (" << res.first.x << ", " << res.first.y << ") ";
100    cout << "(" << res.second.x << ", " << res.second.y << ") \n";
101    cout << "diameter: " << sqrt(sqdist(res.first, res.second)) << " \n";
102    return 0;
103 }

```

5.3.6 Closest Point Pair

```

1  /*
2
3  Given a range containing distinct points on the Cartesian plane,
4  determine two points which have the closest possible distance.
5  A divide and conquer algorithm is used. Note that the ordering
6  of points in the input range may be changed by the function.
7
8  Time Complexity:  $O(n \log^2 n)$  where  $n$  is the number of points.
9
10 */
11
12 #include <algorithm> /* std::min, std::sort */
13 #include <cfloat>    /* DBL_MAX */
14 #include <cmath>    /* fabs */
15 #include <utility>  /* std::pair */
16
17 typedef std::pair<double, double> point;
18 #define x first
19 #define y second
20
21 double sqdist(const point & a, const point & b) {
22     double dx = a.x - b.x, dy = a.y - b.y;
23     return dx * dx + dy * dy;
24 }
25
26 bool cmp_x(const point & a, const point & b) { return a.x < b.x; }
27 bool cmp_y(const point & a, const point & b) { return a.y < b.y; }
28
29 template<class It>

```

```

30 double rec(It lo, It hi, std::pair<point, point> & res, double mindist) {
31     if (lo == hi) return DBL_MAX;
32     It mid = lo + (hi - lo) / 2;
33     double midx = mid->x;
34     double d1 = rec(lo, mid, res, mindist);
35     mindist = std::min(mindist, d1);
36     double d2 = rec(mid + 1, hi, res, mindist);
37     mindist = std::min(mindist, d2);
38     std::sort(lo, hi, cmp_y);
39     int size = 0;
40     It t[hi - lo];
41     for (It it = lo; it != hi; ++it)
42         if (fabs(it->x - midx) < mindist)
43             t[size++] = it;
44     for (int i = 0; i < size; i++) {
45         for (int j = i + 1; j < size; j++) {
46             point a = *t[i], b = *t[j];
47             if (b.y - a.y >= mindist) break;
48             double dist = sqdist(a, b);
49             if (mindist > dist) {
50                 mindist = dist;
51                 res = std::make_pair(a, b);
52             }
53         }
54     }
55     return mindist;
56 }
57
58 template<class It> std::pair<point, point> closest_pair(It lo, It hi) {
59     std::pair<point, point> res;
60     std::sort(lo, hi, cmp_x);
61     rec(lo, hi, res, DBL_MAX);
62     return res;
63 }
64
65 /** Example Usage */
66
67 #include <iostream>
68 #include <vector>
69 using namespace std;
70
71 int main() {
72     vector<point> v;
73     v.push_back(point(2, 3));
74     v.push_back(point(12, 30));
75     v.push_back(point(40, 50));
76     v.push_back(point(5, 1));
77     v.push_back(point(12, 10));
78     v.push_back(point(3, 4));
79     pair<point, point> res = closest_pair(v.begin(), v.end());
80     cout << "closest_pair: (" << res.first.x << ", " << res.first.y << ") ";
81     cout << "(" << res.second.x << ", " << res.second.y << ") \n";
82     cout << "dist: " << sqrt(sqdist(res.first, res.second)) << " \n"; //1.41421
83     return 0;
84 }

```

5.3.7 Segment Intersection Finding


```

1  /*
2
3  Given a range of segments on the Cartesian plane, identify one
4  pair of segments which intersect each other. This is done using
5  a sweep line algorithm.
6
7  Time Complexity:  $O(n \log n)$  where  $n$  is the number of segments.
8
9  */
10
11 #include <algorithm> /* std::min(), std::max(), std::sort() */
12 #include <cmath>     /* fabs() */
13 #include <set>
14 #include <utility>   /* std::pair */
15
16 typedef std::pair<double, double> point;
17 #define x first
18 #define y second
19
20 const double eps = 1e-9;
21
22 #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
23 #define LT(a, b) ((a) < (b) - eps)        /* less than */
24 #define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */
25
26 double norm(const point & a) { return a.x * a.x + a.y * a.y; }
27 double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
28 double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }
29 double cross(const point & o, const point & a, const point & b) {
30     return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
31 }
32
33 const bool TOUCH_IS_INTERSECT = true;
34
35 bool contain(const double & l, const double & m, const double & h) {
36     if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
37     return LT(l, m) && LT(m, h);
38 }
39
40 bool overlap(const double & l1, const double & h1,
41             const double & l2, const double & h2) {
42     if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
43     return LT(l1, h2) && LT(l2, h1);
44 }
45
46 int seg_intersection(const point & a, const point & b,
47                    const point & c, const point & d) {
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)) {
53         double t0 = dot(ac, ab) / norm(ab);
54         double t1 = t0 + dot(cd, ab) / norm(ab);
55         if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
56             point res1 = std::min(std::min(a, b), std::min(c, d));
57             point res2 = std::min(std::max(a, b), std::max(c, d));
58             return (res1 == res2) ? 0 : 1;
59         }
60     }

```

```

60     return -1;
61 }
62 if (EQ(c1, 0)) return -1;
63 double t = cross(ac, cd) / c1, u = c2 / c1;
64 if (contain(0, t, 1) && contain(0, u, 1)) return 0;
65 return -1;
66 }
67
68 struct segment {
69     point p, q;
70
71     segment() {}
72     segment(const point & p, const point & q) {
73         if (p < q) {
74             this->p = p;
75             this->q = q;
76         } else {
77             this->p = q;
78             this->q = p;
79         }
80     }
81
82     bool operator < (const segment & rhs) const {
83         if (p.x < rhs.p.x) {
84             double c = cross(p, q, rhs.p);
85             if (c != 0) return c > 0;
86         } else if (p.x > rhs.p.x) {
87             double c = cross(rhs.p, rhs.q, q);
88             if (c != 0) return c < 0;
89         }
90         return p.y < rhs.p.y;
91     }
92 };
93
94 template<class SegIt> struct event {
95     point p;
96     int type;
97     SegIt seg;
98
99     event() {}
100     event(const point & p, const int type, SegIt seg) {
101         this->p = p;
102         this->type = type;
103         this->seg = seg;
104     }
105
106     bool operator < (const event & rhs) const {
107         if (p.x != rhs.p.x) return p.x < rhs.p.x;
108         if (type != rhs.type) return type > rhs.type;
109         return p.y < rhs.p.y;
110     }
111 };
112
113 bool intersect(const segment & s1, const segment & s2) {
114     return seg_intersection(s1.p, s1.q, s2.p, s2.q) >= 0;
115 }
116
117 //returns whether any pair of segments in the range [lo, hi) intersect
118 //if the result is true, one such intersection pair will be stored

```

```

119 //into values pointed to by res1 and res2.
120 template<class It>
121 bool find_intersection(It lo, It hi, segment * res1, segment * res2) {
122     int cnt = 0;
123     event<It> e[2 * (hi - lo)];
124     for (It it = lo; it != hi; ++it) {
125         if (it->p > it->q) std::swap(it->p, it->q);
126         e[cnt++] = event<It>(it->p, 1, it);
127         e[cnt++] = event<It>(it->q, -1, it);
128     }
129     std::sort(e, e + cnt);
130     std::set<segment> s;
131     std::set<segment>::iterator it, next, prev;
132     for (int i = 0; i < cnt; i++) {
133         It seg = e[i].seg;
134         if (e[i].type == 1) {
135             it = s.lower_bound(*seg);
136             if (it != s.end() && intersect(*it, *seg)) {
137                 *res1 = *it; *res2 = *seg;
138                 return true;
139             }
140             if (it != s.begin() && intersect(*--it, *seg)) {
141                 *res1 = *it; *res2 = *seg;
142                 return true;
143             }
144             s.insert(*seg);
145         } else {
146             it = s.lower_bound(*seg);
147             next = prev = it;
148             prev = it;
149             if (it != s.begin() && it != --s.end()) {
150                 ++next;
151                 --prev;
152                 if (intersect(*next, *prev)) {
153                     *res1 = *next; *res2 = *prev;
154                     return true;
155                 }
156             }
157             s.erase(it);
158         }
159     }
160     return false;
161 }
162
163 /** Example Usage */
164
165 #include <iostream>
166 #include <vector>
167 using namespace std;
168
169 void print(const segment & s) {
170     cout << "(" << s.p.x << ", " << s.p.y << ")<->";
171     cout << "(" << s.q.x << ", " << s.q.y << ")\\n";
172 }
173
174 int main() {
175     vector<segment> v;
176     v.push_back(segment(point(0, 0), point(2, 2)));
177     v.push_back(segment(point(3, 0), point(0, -1)));

```

```

178     v.push_back(segment(point(0, 2), point(2, -2)));
179     v.push_back(segment(point(0, 3), point(9, 0)));
180     segment res1, res2;
181     bool res = find_intersection(v.begin(), v.end(), &res1, &res2);
182     if (res) {
183         print(res1);
184         print(res2);
185     } else {
186         cout << "No intersections.\n";
187     }
188     return 0;
189 }

```

5.4 Advanced Geometric Computations

5.4.1 Convex Polygon Cut

```

1  /*
2
3  Given a range of points specifying a polygon on the Cartesian
4  plane, as well as two points specifying an infinite line, "cut"
5  off the right part of the polygon with the line and return the
6  resulting polygon that is the left part.
7
8  Time Complexity: O(n) on the number of points in the poylgon.
9
10 */
11
12 #include <cmath>    /* fabs() */
13 #include <utility>  /* std::pair */
14 #include <vector>
15
16 typedef std::pair<double, double> point;
17 #define x first
18 #define y second
19
20 const double eps = 1e-9;
21
22 #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
23 #define LT(a, b) ((a) < (b) - eps)        /* less than */
24 #define GT(a, b) ((a) > (b) + eps)        /* greater than */
25
26 double cross(const point & o, const point & a, const point & b) {
27     return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
28 }
29
30 int orientation(const point & o, const point & a, const point & b) {
31     double c = cross(o, a, b);
32     return LT(c, 0) ? -1 : (GT(c, 0) ? 1 : 0);
33 }
34
35 int line_intersection(const point & p1, const point & p2,
36                      const point & p3, const point & p4, point * p = 0) {
37     double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
38     double c1 = -(p1.x * p2.y - p2.x * p1.y);
39     double a2 = p4.y - p3.y, b2 = p3.x - p4.x;

```

```

40 double c2 = -(p3.x * p4.y - p4.x * p3.y);
41 double x = -(c1 * b2 - c2 * b1), y = -(a1 * c2 - a2 * c1);
42 double det = a1 * b2 - a2 * b1;
43 if (EQ(det, 0))
44     return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
45 if (p != 0) *p = point(x / det, y / det);
46 return 0;
47 }
48
49 template<class It>
50 std::vector<point> convex_cut(It lo, It hi, const point & p, const point & q) {
51     std::vector<point> res;
52     for (It i = lo, j = hi - 1; i != hi; j = i++) {
53         int d1 = orientation(p, q, *j), d2 = orientation(p, q, *i);
54         if (d1 >= 0) res.push_back(*j);
55         if (d1 * d2 < 0) {
56             point r;
57             line_intersection(p, q, *j, *i, &r);
58             res.push_back(r);
59         }
60     }
61     return res;
62 }
63
64 /** Example Usage */
65
66 #include <iostream>
67 using namespace std;
68
69 int main() {
70     //irregular pentagon with only (1, 2) not on the convex hull
71     vector<point> v;
72     v.push_back(point(1, 3));
73     v.push_back(point(1, 2));
74     v.push_back(point(2, 1));
75     v.push_back(point(0, 0));
76     v.push_back(point(-1, 3));
77     //cut using the vertical line through (0, 0)
78     vector<point> res = convex_cut(v.begin(), v.end(), point(0, 0), point(0, 1));
79     cout << "left_cut:\n";
80     for (int i = 0; i < (int)res.size(); i++)
81         cout << "(" << res[i].x << ", " << res[i].y << ")\n";
82     return 0;
83 }

```

5.4.2 Polygon Union and Intersection

```

1  /*
2
3  Given two ranges of points respectively denoting the vertices of
4  two polygons, determine the intersection area of those polygons.
5  Using this, we can easily calculate their union with the formula:
6      union_area(A, B) = area(A) + area(B) - intersection_area(A, B)
7
8  Time Complexity:  $O(n^2 \log n)$ , where  $n$  is the total number of vertices.
9
10 */

```

```

11
12 #include <algorithm> /* std::sort() */
13 #include <cmath>     /* fabs(), sqrt() */
14 #include <set>
15 #include <utility>   /* std::pair */
16 #include <vector>
17
18 const double eps = 1e-9;
19
20 #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
21 #define LT(a, b) ((a) < (b) - eps)        /* less than */
22 #define LE(a, b) ((a) <= (b) + eps)      /* less than or equal to */
23
24 typedef std::pair<double, double> point;
25 #define x first
26 #define y second
27
28 inline int sgn(const double & x) {
29     return (0.0 < x) - (x < 0.0);
30 }
31
32 //Line and line segment intersection (see their own sections)
33
34 int line_intersection(const point & p1, const point & p2,
35                     const point & p3, const point & p4, point * p = 0) {
36     double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
37     double c1 = -(p1.x * p2.y - p2.x * p1.y);
38     double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
39     double c2 = -(p3.x * p4.y - p4.x * p3.y);
40     double x = -(c1 * b2 - c2 * b1), y = -(a1 * c2 - a2 * c1);
41     double det = a1 * b2 - a2 * b1;
42     if (EQ(det, 0))
43         return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
44     if (p != 0) *p = point(x / det, y / det);
45     return 0;
46 }
47
48 double norm(const point & a) { return a.x * a.x + a.y * a.y; }
49 double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
50 double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }
51
52 const bool TOUCH_IS_INTERSECT = true;
53
54 bool contain(const double & l, const double & m, const double & h) {
55     if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
56     return LT(l, m) && LT(m, h);
57 }
58
59 bool overlap(const double & l1, const double & h1,
60             const double & l2, const double & h2) {
61     if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
62     return LT(l1, h2) && LT(l2, h1);
63 }
64
65 int seg_intersection(const point & a, const point & b,
66                   const point & c, const point & d,
67                   point * p = 0, point * q = 0) {
68     point ab(b.x - a.x, b.y - a.y);
69     point ac(c.x - a.x, c.y - a.y);

```

```

70 point cd(d.x - c.x, d.y - c.y);
71 double c1 = cross(ab, cd), c2 = cross(ac, ab);
72 if (EQ(c1, 0) && EQ(c2, 0)) { //collinear
73     double t0 = dot(ac, ab) / norm(ab);
74     double t1 = t0 + dot(cd, ab) / norm(ab);
75     if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
76         point res1 = std::max(std::min(a, b), std::min(c, d));
77         point res2 = std::min(std::max(a, b), std::max(c, d));
78         if (res1 == res2) {
79             if (p != 0) *p = res1;
80             return 0; //collinear, meeting at an endpoint
81         }
82         if (p != 0 && q != 0) *p = res1, *q = res2;
83         return 1; //collinear and overlapping
84     } else {
85         return -1; //collinear and disjoint
86     }
87 }
88 if (EQ(c1, 0)) return -1; //parallel and disjoint
89 double t = cross(ac, cd) / c1, u = c2 / c1;
90 if (contain(0, t, 1) && contain(0, u, 1)) {
91     if (p != 0) *p = point(a.x + t * ab.x, a.y + t * ab.y);
92     return 0; //non-parallel with one intersection
93 }
94 return -1; //non-parallel with no intersections
95 }
96
97 struct event {
98     double y;
99     int mask_delta;
100
101     event(double y = 0, int mask_delta = 0) {
102         this->y = y;
103         this->mask_delta = mask_delta;
104     }
105
106     bool operator < (const event & e) const {
107         if (y != e.y) return y < e.y;
108         return mask_delta < e.mask_delta;
109     }
110 };
111
112 template<class It>
113 double intersection_area(It lo1, It hi1, It lo2, It hi2) {
114     It plo[2] = {lo1, lo2}, phi[] = {hi1, hi2};
115     std::set<double> xs;
116     for (It i1 = lo1; i1 != hi1; ++i1) xs.insert(i1->x);
117     for (It i2 = lo2; i2 != hi2; ++i2) xs.insert(i2->x);
118     for (It i1 = lo1, j1 = hi1 - 1; i1 != hi1; j1 = i1++) {
119         for (It i2 = lo2, j2 = hi2 - 1; i2 != hi2; j2 = i2++) {
120             point p;
121             if (seg_intersection(*i1, *j1, *i2, *j2, &p) == 0)
122                 xs.insert(p.x);
123         }
124     }
125     std::vector<double> xsa(xs.begin(), xs.end());
126     double res = 0;
127     for (int k = 0; k < (int)xsa.size() - 1; k++) {
128         double x = (xsa[k] + xsa[k + 1]) / 2;

```

```

129     point sweep0(x, 0), sweep1(x, 1);
130     std::vector<event> events;
131     for (int poly = 0; poly < 2; poly++) {
132         It lo = plo[poly], hi = phi[poly];
133         double area = 0;
134         for (It i = lo, j = hi - 1; i != hi; j = i++)
135             area += (j->x - i->x) * (j->y + i->y);
136         for (It j = lo, i = hi - 1; j != hi; i = j++) {
137             point p;
138             if (line_intersection(*j, *i, sweep0, sweep1, &p) == 0) {
139                 double y = p.y, x0 = i->x, x1 = j->x;
140                 if (x0 < x && x1 > x) {
141                     events.push_back(event(y, sgn(area) * (1 << poly)));
142                 } else if (x0 > x && x1 < x) {
143                     events.push_back(event(y, -sgn(area) * (1 << poly)));
144                 }
145             }
146         }
147     }
148     std::sort(events.begin(), events.end());
149     double a = 0.0;
150     int mask = 0;
151     for (int j = 0; j < (int)events.size(); j++) {
152         if (mask == 3)
153             a += events[j].y - events[j - 1].y;
154         mask += events[j].mask_delta;
155     }
156     res += a * (xsa[k + 1] - xsa[k]);
157 }
158 return res;
159 }
160
161 template<class It> double polygon_area(It lo, It hi) {
162     if (lo == hi) return 0;
163     double area = 0;
164     if (*lo != *--hi)
165         area += (lo->x - hi->x) * (lo->y + hi->y);
166     for (It i = hi, j = hi - 1; i != lo; --i, --j)
167         area += (i->x - j->x) * (i->y + j->y);
168     return fabs(area / 2.0);
169 }
170
171 template<class It>
172 double union_area(It lo1, It hi1, It lo2, It hi2) {
173     return polygon_area(lo1, hi1) + polygon_area(lo2, hi2) -
174         intersection_area(lo1, hi1, lo2, hi2);
175 }
176
177 /** Example Usage */
178
179 #include <cassert>
180 using namespace std;
181
182 int main() {
183     vector<point> p1, p2;
184
185     //irregular pentagon with area 1.5 triangle in quadrant 2
186     p1.push_back(point(1, 3));
187     p1.push_back(point(1, 2));

```



```

188  p1.push_back(point(2, 1));
189  p1.push_back(point(0, 0));
190  p1.push_back(point(-1, 3));
191  //a big square in quadrant 2
192  p2.push_back(point(0, 0));
193  p2.push_back(point(0, 3));
194  p2.push_back(point(-3, 3));
195  p2.push_back(point(-3, 0));
196
197  assert(EQ(1.5, intersection_area(p1.begin(), p1.end(),
198                                p2.begin(), p2.end())));
199  assert(EQ(12.5, union_area(p1.begin(), p1.end(),
200                             p2.begin(), p2.end())));
201  return 0;
202 }

```

5.4.3 Delaunay Triangulation (Simple)

```

1  /*
2
3  Given a range of points P on the Cartesian plane, the Delaunay
4  Triangulation of said points is a set of non-overlapping triangles
5  covering the entire convex hull of P, such that no point in P lies
6  within the circumcircle of any of the resulting triangles. The
7  triangulation maximizes the minimum angle of all the angles of the
8  triangles in the triangulation. In addition, for any point p in the
9  convex hull (not necessarily in P), the nearest point is guaranteed
10 to be a vertex of the enclosing triangle from the triangulation.
11 See: https://en.wikipedia.org/wiki/Delaunay\_triangulation
12
13 The triangulation may not exist (e.g. for a set of collinear points)
14 or it may not be unique (multiple possible triangulations may exist).
15 The triangulation may not exist (e.g. for a set of collinear points)
16 or it may not be unique (multiple possible triangulations may exist).
17 The following program assumes that a triangulation exists, and
18 produces one such valid result using one of the simplest algorithms
19 to solve this problem. It involves encasing the simplex in a circle
20 and rejecting the simplex if another point in the tessellation is
21 within the generalized circle.
22
23 Time Complexity:  $O(n^4)$  on the number of input points.
24
25 */
26
27 #include <algorithm> /* std::sort() */
28 #include <cmath>     /* fabs(), sqrt() */
29 #include <utility>   /* std::pair */
30 #include <vector>
31
32 const double eps = 1e-9;
33
34 #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
35 #define LT(a, b) ((a) < (b) - eps)        /* less than */
36 #define GT(a, b) ((a) > (b) + eps)        /* greater than */
37 #define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */
38 #define GE(a, b) ((a) >= (b) - eps)       /* greater than or equal to */
39

```

```

40 typedef std::pair<double, double> point;
41 #define x first
42 #define y second
43
44 double norm(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) { return a.x * b.y - a.y * b.x; }
47
48 const bool TOUCH_IS_INTERSECT = false;
49
50 bool contain(const double & l, const double & m, const double & h) {
51     if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
52     return LT(l, m) && LT(m, h);
53 }
54
55 bool overlap(const double & l1, const double & h1,
56             const double & l2, const double & h2) {
57     if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
58     return LT(l1, h2) && LT(l2, h1);
59 }
60
61 int seg_intersection(const point & a, const point & b,
62                    const point & c, const point & d) {
63     point ab(b.x - a.x, b.y - a.y);
64     point ac(c.x - a.x, c.y - a.y);
65     point cd(d.x - c.x, d.y - c.y);
66     double c1 = cross(ab, cd), c2 = cross(ac, ab);
67     if (EQ(c1, 0) && EQ(c2, 0)) {
68         double t0 = dot(ac, ab) / norm(ab);
69         double t1 = t0 + dot(cd, ab) / norm(ab);
70         if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
71             point res1 = std::max(std::min(a, b), std::min(c, d));
72             point res2 = std::min(std::max(a, b), std::max(c, d));
73             return (res1 == res2) ? 0 : 1;
74         }
75         return -1;
76     }
77     if (EQ(c1, 0)) return -1;
78     double t = cross(ac, cd) / c1, u = c2 / c1;
79     if (contain(0, t, 1) && contain(0, u, 1)) return 0;
80     return -1;
81 }
82
83 struct triangle { point a, b, c; };
84
85 template<class It>
86 std::vector<triangle> delaunay_triangulation(It lo, It hi) {
87     int n = hi - lo;
88     std::vector<double> x, y, z;
89     for (It it = lo; it != hi; ++it) {
90         x.push_back(it->x);
91         y.push_back(it->y);
92         z.push_back((it->x) * (it->x) + (it->y) * (it->y));
93     }
94     std::vector<triangle> res;
95     for (int i = 0; i < n - 2; i++) {
96         for (int j = i + 1; j < n; j++) {
97             for (int k = i + 1; k < n; k++) {
98                 if (j == k) continue;

```

```

99     double nx = (y[j] - y[i]) * (z[k] - z[i]) - (y[k] - y[i]) * (z[j] - z[i]);
100    double ny = (x[k] - x[i]) * (z[j] - z[i]) - (x[j] - x[i]) * (z[k] - z[i]);
101    double nz = (x[j] - x[i]) * (y[k] - y[i]) - (x[k] - x[i]) * (y[j] - y[i]);
102    if (GE(nz, 0)) continue;
103    bool done = false;
104    for (int m = 0; m < n; m++)
105        if (x[m] - x[i]) * nx + (y[m] - y[i]) * ny + (z[m] - z[i]) * nz > 0) {
106            done = true;
107            break;
108        }
109    if (!done) { //handle 4 points on a circle
110        point s1[] = { *(lo + i), *(lo + j), *(lo + k), *(lo + i) };
111        for (int t = 0; t < (int)res.size(); t++) {
112            point s2[] = { res[t].a, res[t].b, res[t].c, res[t].a };
113            for (int u = 0; u < 3; u++)
114                for (int v = 0; v < 3; v++)
115                    if (seg_intersection(s1[u], s1[u + 1], s2[v], s2[v + 1]) == 0)
116                        goto skip;
117        }
118        res.push_back((triangle){*(lo + i), *(lo + j), *(lo + k)});
119    }
120    skip::;
121    }
122    }
123    }
124    return res;
125    }
126
127    /** Example Usage **/
128
129    #include <iostream>
130    using namespace std;
131
132    int main() {
133        vector<point> v;
134        v.push_back(point(1, 3));
135        v.push_back(point(1, 2));
136        v.push_back(point(2, 1));
137        v.push_back(point(0, 0));
138        v.push_back(point(-1, 3));
139        vector<triangle> dt = delaunay_triangulation(v.begin(), v.end());
140        for (int i = 0; i < (int)dt.size(); i++) {
141            cout << "Triangle: ";
142            cout << "(" << dt[i].a.x << "," << dt[i].a.y << ") ";
143            cout << "(" << dt[i].b.x << "," << dt[i].b.y << ") ";
144            cout << "(" << dt[i].c.x << "," << dt[i].c.y << ") \n";
145        }
146        return 0;
147    }

```

5.4.4 Delaunay Triangulation (Fast)

```

1  /*
2
3  Given a range of points P on the Cartesian plane, the Delaunay
4  Triangulation of said points is a set of non-overlapping triangles
5  covering the entire convex hull of P, such that no point in P lies

```

```

6  within the circumcircle of any of the resulting triangles. The
7  triangulation maximizes the minimum angle of all the angles of the
8  triangles in the triangulation. In addition, for any point p in the
9  convex hull (not necessarily in P), the nearest point is guaranteed
10 to be a vertex of the enclosing triangle from the triangulation.
11 See: https://en.wikipedia.org/wiki/Delaunay\_triangulation
12
13 The triangulation may not exist (e.g. for a set of collinear points)
14 or it may not be unique (multiple possible triangulations may exist).
15 The following program assumes that a triangulation exists, and
16 produces one such valid result. The following is a C++ adaptation of
17 a FORTRAN90 program, which applies a divide and conquer algorithm
18 with complex linear-time merging. The original program can be found
19 via the following link. It contains more thorough documentation,
20 comments, and debugging messages associated with the current asserts().
21 http://people.sc.fsu.edu/~burkardt/f\_src/table\_delaunay/table\_delaunay.html
22
23 Time Complexity:  $O(n \log n)$  on the number of input points.
24
25 */
26
27 #include <algorithm> /* std::min(), std::max() */
28 #include <cassert>
29 #include <cmath>     /* fabs(), sqrt() */
30 #include <utility>   /* std::pair */
31 #include <vector>
32
33 int wrap(int ival, int ilo, int ihi) {
34     int jlo = std::min(ilo, ihi), jhi = std::max(ilo, ihi);
35     int wide = jhi + 1 - jlo, res = jlo;
36     if (wide != 1) {
37         assert(wide != 0);
38         int tmp = (ival - jlo) % wide;
39         if (tmp < 0) res += abs(wide);
40         res += tmp;
41     }
42     return res;
43 }
44
45 double epsilon() {
46     double r = 1;
47     while (1 < (double)(r + 1)) r /= 2;
48     return 2 * r;
49 }
50
51 void permute(int n, double a[][2], int p[]) {
52     for (int istart = 1; istart <= n; istart++) {
53         if (p[istart - 1] < 0) continue;
54         if (p[istart - 1] == istart) {
55             p[istart - 1] = -p[istart - 1];
56             continue;
57         }
58         double tmp0 = a[istart - 1][0];
59         double tmp1 = a[istart - 1][1];
60         int iget = istart;
61         for (;;) {
62             int iput = iget;
63             iget = p[iget - 1];
64             p[iput - 1] = -p[iput - 1];

```

```

65     assert(!(iget < 1 || n < iget));
66     if (iget == istart) {
67         a[iput - 1][0] = tmp0;
68         a[iput - 1][1] = tmp1;
69         break;
70     }
71     a[iput - 1][0] = a[iget - 1][0];
72     a[iput - 1][1] = a[iget - 1][1];
73 }
74 }
75 for (int i = 0; i < n; i++) p[i] = -p[i];
76 return;
77 }
78
79 int * sort_heap(int n, double a[][2]) {
80     double aval[2];
81     int i, ir, j, l, idxt;
82     int *idx;
83     if (n < 1) return NULL;
84     if (n == 1) {
85         idx = new int[1];
86         idx[0] = 1;
87         return idx;
88     }
89     idx = new int[n];
90     for (int i = 0; i < n; i++) idx[i] = i + 1;
91     l = n / 2 + 1;
92     ir = n;
93     for (;;) {
94         if (l < 1) {
95             l--;
96             idxt = idx[l - 1];
97             aval[0] = a[idxt - 1][0];
98             aval[1] = a[idxt - 1][1];
99         } else {
100             idxt = idx[ir - 1];
101             aval[0] = a[idxt - 1][0];
102             aval[1] = a[idxt - 1][1];
103             idx[ir - 1] = idx[0];
104             if (--ir == 1) {
105                 idx[0] = idxt;
106                 break;
107             }
108         }
109         i = l;
110         j = 2 * l;
111         while (j <= ir) {
112             if (j < ir && (a[idx[j - 1] - 1][0] < a[idx[j] - 1][0] ||
113                 (a[idx[j - 1] - 1][0] == a[idx[j] - 1][0] &&
114                 a[idx[j - 1] - 1][1] < a[idx[j] - 1][1]))) {
115                 j++;
116             }
117             if (aval[0] < a[idx[j - 1] - 1][0] ||
118                 (aval[0] == a[idx[j - 1] - 1][0] &&
119                 aval[1] < a[idx[j - 1] - 1][1])) {
120                 idx[i - 1] = idx[j - 1];
121                 i = j;
122                 j *= 2;
123             } else {

```

```

124         j = ir + 1;
125     }
126 }
127 idx[i - 1] = idxt;
128 }
129 return idx;
130 }
131
132 int lrline(double xu, double yu, double xv1, double yv1,
133           double xv2, double yv2, double dv) {
134     double tol = 1e-7;
135     double dx = xv2 - xv1, dy = yv2 - yv1;
136     double dxu = xu - xv1, dyu = yu - yv1;
137     double t = dy * dxu - dx * dyu + dv * sqrt(dx * dx + dy * dy);
138     double tolabs = tol * std::max(std::max(fabs(dx), fabs(dy)),
139                                   std::max(fabs(dxu), std::max(fabs(dyu), fabs(dv))));
140     if (tolabs < t) return 1;
141     if (-tolabs <= t) return 0;
142     return -1;
143 }
144
145 void vbedg(double x, double y, int point_num, double point_xy[][2],
146           int tri_num, int tri_nodes[][3], int tri_neigh[][3],
147           int *ltri, int *ledg, int *rtri, int *redg) {
148     int a, b;
149     double ax, ay, bx, by;
150     bool done;
151     int e, l, t;
152     if (*ltri == 0) {
153         done = false;
154         *ltri = *rtri;
155         *ledg = *redg;
156     } else {
157         done = true;
158     }
159     for (;;) {
160         l = -tri_neigh[*rtri - 1][*redg - 1];
161         t = l / 3;
162         e = l % 3 + 1;
163         a = tri_nodes[t - 1][e - 1];
164         if (e <= 2) {
165             b = tri_nodes[t - 1][e];
166         } else {
167             b = tri_nodes[t - 1][0];
168         }
169         ax = point_xy[a - 1][0];
170         ay = point_xy[a - 1][1];
171         bx = point_xy[b - 1][0];
172         by = point_xy[b - 1][1];
173         if (lrline(x, y, ax, ay, bx, by, 0.0) <= 0) break;
174         *rtri = t;
175         *redg = e;
176     }
177     if (done) return;
178     t = *ltri;
179     e = *ledg;
180     for (;;) {
181         b = tri_nodes[t - 1][e - 1];
182         e = wrap(e - 1, 1, 3);

```

```

183     while (0 < tri_neigh[t - 1][e - 1]) {
184         t = tri_neigh[t - 1][e - 1];
185         if (tri_nodes[t - 1][0] == b) {
186             e = 3;
187         } else if (tri_nodes[t - 1][1] == b) {
188             e = 1;
189         } else {
190             e = 2;
191         }
192     }
193     a = tri_nodes[t - 1][e - 1];
194     ax = point_xy[a - 1][0];
195     ay = point_xy[a - 1][1];
196     bx = point_xy[b - 1][0];
197     by = point_xy[b - 1][1];
198     if (lrline(x, y, ax, ay, bx, by, 0.0) <= 0) break;
199 }
200 *ltri = t;
201 *ledg = e;
202 return;
203 }
204
205 int diaedg(double x0, double y0, double x1, double y1,
206           double x2, double y2, double x3, double y3) {
207     double ca, cb, s, tol, tola, tolb;
208     int value;
209     tol = 100.0 * epsilon();
210     double dx10 = x1 - x0, dy10 = y1 - y0;
211     double dx12 = x1 - x2, dy12 = y1 - y2;
212     double dx30 = x3 - x0, dy30 = y3 - y0;
213     double dx32 = x3 - x2, dy32 = y3 - y2;
214     tola = tol * std::max(std::max(fabs(dx10), fabs(dy10)),
215                          std::max(fabs(dx30), fabs(dy30)));
216     tolb = tol * std::max(std::max(fabs(dx12), fabs(dy12)),
217                          std::max(fabs(dx32), fabs(dy32)));
218     ca = dx10 * dx30 + dy10 * dy30;
219     cb = dx12 * dx32 + dy12 * dy32;
220     if (tola < ca && tolb < cb) {
221         value = -1;
222     } else if (ca < -tola && cb < -tolb) {
223         value = 1;
224     } else {
225         tola = std::max(tola, tolb);
226         s = (dx10 * dy30 - dx30 * dy10) * cb + (dx32 * dy12 - dx12 * dy32) * ca;
227         if (tola < s) {
228             value = -1;
229         } else if (s < -tola) {
230             value = 1;
231         } else {
232             value = 0;
233         }
234     }
235     return value;
236 }
237
238 int swapec(int i, int *top, int *btri, int *bedg,
239           int point_num, double point_xy[][2],
240           int tri_num, int tri_nodes[][3], int tri_neigh[][3], int stack[]) {
241     int a, b, c, e, ee, em1, ep1, f, fm1, fp1, l, r, s, swap, t, tt, u;

```

```

242 double x = point_xy[i - 1][0];
243 double y = point_xy[i - 1][1];
244 for (;;) {
245     if (*top <= 0) break;
246     t = stack[*top - 1];
247     *top = *top - 1;
248     if (tri_nodes[t - 1][0] == i) {
249         e = 2;
250         b = tri_nodes[t - 1][2];
251     } else if (tri_nodes[t - 1][1] == i) {
252         e = 3;
253         b = tri_nodes[t - 1][0];
254     } else {
255         e = 1;
256         b = tri_nodes[t - 1][1];
257     }
258     a = tri_nodes[t - 1][e - 1];
259     u = tri_neigh[t - 1][e - 1];
260     if (tri_neigh[u - 1][0] == t) {
261         f = 1;
262         c = tri_nodes[u - 1][2];
263     } else if (tri_neigh[u - 1][1] == t) {
264         f = 2;
265         c = tri_nodes[u - 1][0];
266     } else {
267         f = 3;
268         c = tri_nodes[u - 1][1];
269     }
270     swap = diaedg(x, y, point_xy[a - 1][0], point_xy[a - 1][1],
271                 point_xy[c - 1][0], point_xy[c - 1][1],
272                 point_xy[b - 1][0], point_xy[b - 1][1]);
273     if (swap == 1) {
274         em1 = wrap(e - 1, 1, 3);
275         ep1 = wrap(e + 1, 1, 3);
276         fm1 = wrap(f - 1, 1, 3);
277         fp1 = wrap(f + 1, 1, 3);
278         tri_nodes[t - 1][ep1 - 1] = c;
279         tri_nodes[u - 1][fp1 - 1] = i;
280         r = tri_neigh[t - 1][ep1 - 1];
281         s = tri_neigh[u - 1][fp1 - 1];
282         tri_neigh[t - 1][ep1 - 1] = u;
283         tri_neigh[u - 1][fp1 - 1] = t;
284         tri_neigh[t - 1][e - 1] = s;
285         tri_neigh[u - 1][f - 1] = r;
286         if (0 < tri_neigh[u - 1][fm1 - 1]) {
287             *top = *top + 1;
288             stack[*top - 1] = u;
289         }
290         if (0 < s) {
291             if (tri_neigh[s - 1][0] == u) {
292                 tri_neigh[s - 1][0] = t;
293             } else if (tri_neigh[s - 1][1] == u) {
294                 tri_neigh[s - 1][1] = t;
295             } else {
296                 tri_neigh[s - 1][2] = t;
297             }
298             *top = *top + 1;
299             if (point_num < *top) return 8;
300             stack[*top - 1] = t;

```



```

301     } else {
302         if (u == *btri && fp1 == *bedg) {
303             *btri = t;
304             *bedg = e;
305         }
306         l = - (3 * t + e - 1);
307         tt = t;
308         ee = em1;
309         while (0 < tri_neigh[tt - 1][ee - 1]) {
310             tt = tri_neigh[tt - 1][ee - 1];
311             if (tri_nodes[tt - 1][0] == a) {
312                 ee = 3;
313             } else if (tri_nodes[tt - 1][1] == a) {
314                 ee = 1;
315             } else {
316                 ee = 2;
317             }
318         }
319         tri_neigh[tt - 1][ee - 1] = 1;
320     }
321     if (0 < r) {
322         if (tri_neigh[r - 1][0] == t) {
323             tri_neigh[r - 1][0] = u;
324         } else if (tri_neigh[r - 1][1] == t) {
325             tri_neigh[r - 1][1] = u;
326         } else {
327             tri_neigh[r - 1][2] = u;
328         }
329     } else {
330         if (t == *btri && ep1 == *bedg) {
331             *btri = u;
332             *bedg = f;
333         }
334         l = -(3 * u + f - 1);
335         tt = u;
336         ee = fm1;
337         while (0 < tri_neigh[tt - 1][ee - 1]) {
338             tt = tri_neigh[tt - 1][ee - 1];
339             if (tri_nodes[tt - 1][0] == b) {
340                 ee = 3;
341             } else if (tri_nodes[tt - 1][1] == b) {
342                 ee = 1;
343             } else {
344                 ee = 2;
345             }
346         }
347         tri_neigh[tt - 1][ee - 1] = 1;
348     }
349 }
350 }
351 return 0;
352 }
353
354 void perm_inv(int n, int p[]) {
355     int i, i0, i1, i2;
356     assert(n > 0);
357     for (i = 1; i <= n; i++) {
358         i1 = p[i - 1];
359         while (i < i1) {

```

```

360     i2 = p[i1 - 1];
361     p[i1 - 1] = -i2;
362     i1 = i2;
363 }
364 p[i - 1] = -p[i - 1];
365 }
366 for (i = 1; i <= n; i++) {
367     i1 = -p[i - 1];
368     if (0 <= i1) {
369         i0 = i;
370         for (;;) {
371             i2 = p[i1 - 1];
372             p[i1 - 1] = i0;
373             if (i2 < 0) break;
374             i0 = i1;
375             i1 = i2;
376         }
377     }
378 }
379 return;
380 }
381
382 int dtris2(int point_num, double point_xy[][2],
383           int tri_nodes[][3], int tri_neigh[][3]) {
384     double cmax;
385     int e, error;
386     int i, j, k, l, m, m1, m2, n;
387     int ledg, lr, ltri, redg, rtri, t, top;
388     double tol;
389     int *stack = new int[point_num];
390     tol = 100.0 * epsilon();
391     int *idx = sort_heap(point_num, point_xy);
392     permute(point_num, point_xy, idx);
393     m1 = 0;
394     for (i = 1; i < point_num; i++) {
395         m = m1;
396         m1 = i;
397         k = -1;
398         for (j = 0; j <= 1; j++) {
399             cmax = std::max(fabs(point_xy[m][j]), fabs(point_xy[m1][j]));
400             if (tol * (cmax + 1.0) < fabs(point_xy[m][j] - point_xy[m1][j])) {
401                 k = j;
402                 break;
403             }
404         }
405         assert(k != -1);
406     }
407     m1 = 1;
408     m2 = 2;
409     j = 3;
410     for (;;) {
411         assert(point_num >= j);
412         m = j;
413         lr = lrline(point_xy[m - 1][0], point_xy[m - 1][1],
414                   point_xy[m1 - 1][0], point_xy[m1 - 1][1],
415                   point_xy[m2 - 1][0], point_xy[m2 - 1][1], 0.0);
416         if (lr != 0) break;
417         j++;
418     }

```

```

419 int tri_num = j - 2;
420 if (lr == -1) {
421     tri_nodes[0][0] = m1;
422     tri_nodes[0][1] = m2;
423     tri_nodes[0][2] = m;
424     tri_neigh[0][2] = -3;
425     for (i = 2; i <= tri_num; i++) {
426         m1 = m2;
427         m2 = i + 1;
428         tri_nodes[i - 1][0] = m1;
429         tri_nodes[i - 1][1] = m2;
430         tri_nodes[i - 1][2] = m;
431         tri_neigh[i - 1][0] = -3 * i;
432         tri_neigh[i - 1][1] = i;
433         tri_neigh[i - 1][2] = i - 1;
434     }
435     tri_neigh[tri_num - 1][0] = -3 * tri_num - 1;
436     tri_neigh[tri_num - 1][1] = -5;
437     ledg = 2;
438     ltri = tri_num;
439 } else {
440     tri_nodes[0][0] = m2;
441     tri_nodes[0][1] = m1;
442     tri_nodes[0][2] = m;
443     tri_neigh[0][0] = -4;
444     for (i = 2; i <= tri_num; i++) {
445         m1 = m2;
446         m2 = i+1;
447         tri_nodes[i - 1][0] = m2;
448         tri_nodes[i - 1][1] = m1;
449         tri_nodes[i - 1][2] = m;
450         tri_neigh[i - 2][2] = i;
451         tri_neigh[i - 1][0] = -3 * i - 3;
452         tri_neigh[i - 1][1] = i - 1;
453     }
454     tri_neigh[tri_num - 1][2] = -3 * (tri_num);
455     tri_neigh[0][1] = -3 * (tri_num) - 2;
456     ledg = 2;
457     ltri = 1;
458 }
459 top = 0;
460 for (i = j + 1; i <= point_num; i++) {
461     m = i;
462     m1 = tri_nodes[ltri - 1][ledg - 1];
463     if (ledg <= 2) {
464         m2 = tri_nodes[ltri - 1][ledg];
465     } else {
466         m2 = tri_nodes[ltri - 1][0];
467     }
468     lr = lrline(point_xy[m - 1][0], point_xy[m - 1][1],
469                point_xy[m1 - 1][0], point_xy[m1 - 1][1],
470                point_xy[m2 - 1][0], point_xy[m2 - 1][1], 0.0);
471     if (0 < lr) {
472         rtri = ltri;
473         redg = ledg;
474         ltri = 0;
475     } else {
476         l = -tri_neigh[ltri - 1][ledg - 1];
477         rtri = l / 3;

```

```

478     redg = (l % 3) + 1;
479 }
480 vbedg(point_xy[m - 1][0], point_xy[m - 1][1],
481       point_num, point_xy, tri_num, tri_nodes, tri_neigh,
482       &ltri, &ledg, &rtri, &redg);
483 n = tri_num + 1;
484 l = -tri_neigh[ltri - 1][ledg - 1];
485 for (;;) {
486     t = l / 3;
487     e = (l % 3) + 1;
488     l = -tri_neigh[t - 1][e - 1];
489     m2 = tri_nodes[t - 1][e - 1];
490     if (e <= 2) {
491         m1 = tri_nodes[t - 1][e];
492     } else {
493         m1 = tri_nodes[t - 1][0];
494     }
495     tri_num++;
496     tri_neigh[t - 1][e - 1] = tri_num;
497     tri_nodes[tri_num - 1][0] = m1;
498     tri_nodes[tri_num - 1][1] = m2;
499     tri_nodes[tri_num - 1][2] = m;
500     tri_neigh[tri_num - 1][0] = t;
501     tri_neigh[tri_num - 1][1] = tri_num - 1;
502     tri_neigh[tri_num - 1][2] = tri_num + 1;
503     top++;
504     assert(point_num >= top);
505     stack[top - 1] = tri_num;
506     if (t == rtri && e == redg) break;
507 }
508 tri_neigh[ltri - 1][ledg - 1] = -3 * n - 1;
509 tri_neigh[n - 1][1] = -3 * tri_num - 2;
510 tri_neigh[tri_num - 1][2] = -1;
511 ltri = n;
512 ledg = 2;
513 error = swapec(m, &top, &ltri, &ledg, point_num, point_xy,
514               tri_num, tri_nodes, tri_neigh, stack);
515 assert(error == 0);
516 }
517 for (i = 0; i < 3; i++)
518     for (j = 0; j < tri_num; j++)
519         tri_nodes[j][i] = idx[tri_nodes[j][i] - 1];
520 perm_inv(point_num, idx);
521 permute(point_num, point_xy, idx);
522 delete[] idx;
523 delete[] stack;
524 return tri_num;
525 }
526
527 /** C++ Wrapper */
528
529 typedef std::pair<double, double> point;
530 #define x first
531 #define y second
532
533 struct triangle { point a, b, c; };
534
535 template<class It>
536 std::vector<triangle> delaunay_triangulation(It lo, It hi) {

```

```

537     int n = hi - lo;
538     double points[n][2];
539     int tri_nodes[3 * n][3], tri_neigh[3 * n][3];
540     int curr = 0;
541     for (It it = lo; it != hi; ++curr, ++it) {
542         points[curr][0] = it->x;
543         points[curr][1] = it->y;
544     }
545     int m = dtris2(n, points, tri_nodes, tri_neigh);
546     std::vector<triangle> res;
547     for (int i = 0; i < m; i++)
548         res.push_back((triangle){*(lo + (tri_nodes[i][0] - 1)),
549                                 *(lo + (tri_nodes[i][1] - 1)),
550                                 *(lo + (tri_nodes[i][2] - 1))});
551     return res;
552 }
553
554 /** Example Usage */
555
556 #include <iostream>
557 using namespace std;
558
559 int main() {
560     vector<point> v;
561     v.push_back(point(1, 3));
562     v.push_back(point(1, 2));
563     v.push_back(point(2, 1));
564     v.push_back(point(0, 0));
565     v.push_back(point(-1, 3));
566     vector<triangle> dt = delaunay_triangulation(v.begin(), v.end());
567     for (int i = 0; i < (int)dt.size(); i++) {
568         cout << "Triangle: ";
569         cout << "(" << dt[i].a.x << "," << dt[i].a.y << ") ";
570         cout << "(" << dt[i].b.x << "," << dt[i].b.y << ") ";
571         cout << "(" << dt[i].c.x << "," << dt[i].c.y << ") \n";
572     }
573     return 0;
574 }

```

Chapter 6

Strings

6.1 Strings Toolbox

```
1  /*
2
3  Useful or trivial string operations. These functions are not particularly
4  algorithmic. They are typically naive implementations using C++ features.
5  They depend on many features of the C++ <string> library, which tend to
6  have an unspecified complexity. They may not be optimally efficient.
7
8  */
9
10 #include <cstdlib>
11 #include <sstream>
12 #include <string>
13 #include <vector>
14
15 //integer to string conversion and vice versa using C++ features
16
17 //note that a similar std::to_string is introduced in C++0x
18 template<class Int>
19 std::string to_string(const Int & i) {
20     std::ostringstream oss;
21     oss << i;
22     return oss.str();
23 }
24
25 //like atoi, except during special cases like overflows
26 int to_int(const std::string & s) {
27     std::istringstream iss(s);
28     int res;
29     if (!(iss >> res)) /* complain */;
30     return res;
31 }
32
33 /*
34
35 itoa implementation (fast)
36 documentation: http://www.cplusplus.com/reference/cstdlib/itoa/
37 taken from: http://www.jb.man.ac.uk/~slowe/cpp/itoa.html
```

```

38
39 */
40
41 char* itoa(int value, char * str, int base = 10) {
42     if (base < 2 || base > 36) {
43         *str = '\0';
44         return str;
45     }
46     char *ptr = str, *ptr1 = str, tmp_c;
47     int tmp_v;
48     do {
49         tmp_v = value;
50         value /= base;
51         *ptr++ = "zyxwvutsrqponmlkjihgfedcba9876543210123456789"
52             "abcdefghijklmnopqrstuvwxyz"[35 + (tmp_v - value * base)];
53     } while (value);
54     if (tmp_v < 0) *ptr++ = '-';
55     for (*ptr-- = '\0'; ptr1 < ptr; *ptr1++ = tmp_c) {
56         tmp_c = *ptr;
57         *ptr-- = *ptr1;
58     }
59     return str;
60 }
61
62 /*
63
64 Trimming functions (in place). Given a string and optionally a series
65 of characters to be considered for trimming, trims the string's ends
66 (left, right, or both) and returns the string. Note that the ORIGINAL
67 string is trimmed as it's passed by reference, despite the original
68 reference being returned for convenience.
69
70 */
71
72 std::string& ltrim(std::string & s, const std::string & delim = "\n\t\v\f\r") {
73     unsigned int pos = s.find_first_not_of(delim);
74     if (pos != std::string::npos) s.erase(0, pos);
75     return s;
76 }
77
78 std::string& rtrim(std::string & s, const std::string & delim = "\n\t\v\f\r") {
79     unsigned int pos = s.find_last_not_of(delim);
80     if (pos != std::string::npos) s.erase(pos);
81     return s;
82 }
83
84 std::string& trim(std::string & s, const std::string & delim = "\n\t\v\f\r") {
85     return ltrim(rtrim(s));
86 }
87
88 /*
89
90 Returns a copy of the string s with all occurrences of the given
91 string search replaced with the given string replace.
92
93 Time Complexity: Unspecified, but proportional to the number of times
94 the search string occurs and the complexity of std::string::replace,
95 which is unspecified.
96

```

```

97  */
98
99  std::string replace(std::string s,
100                    const std::string & search,
101                    const std::string & replace) {
102      if (search.empty()) return s;
103      unsigned int pos = 0;
104      while ((pos = s.find(search, pos)) != std::string::npos) {
105          s.replace(pos, search.length(), replace);
106          pos += replace.length();
107      }
108      return s;
109  }
110
111  /*
112
113  Tokenizes the string s based on single character delimiters.
114
115  Version 1: Simpler. Only one delimiter character allowed, and this will
116  not skip empty tokens.
117      e.g. split("a:b", ":") yields {"a", "b"}, not {"a", "", "b"}.
118
119  Version 2: All of the characters in the delim parameter that also exists
120  in s will be removed from s, and the token(s) of s that are left over will
121  be added sequentially to a vector and returned. Empty tokens are skipped.
122      e.g. split("a:b", ":") yields {"a", "b"}, not {"a", "", "b"}.
123
124  Time Complexity: O(s.length() * delim.length())
125
126  */
127
128  std::vector<std::string> split(const std::string & s, char delim) {
129      std::vector<std::string> res;
130      std::stringstream ss(s);
131      std::string curr;
132      while (std::getline(ss, curr, delim))
133          res.push_back(curr);
134      return res;
135  }
136
137  std::vector<std::string> split(const std::string & s,
138                                const std::string & delim = "\n\t\v\f\r") {
139      std::vector<std::string> res;
140      std::string curr;
141      for (int i = 0; i < (int)s.size(); i++) {
142          if (delim.find(s[i]) == std::string::npos) {
143              curr += s[i];
144          } else if (!curr.empty()) {
145              res.push_back(curr);
146              curr = "";
147          }
148      }
149      if (!curr.empty()) res.push_back(curr);
150      return res;
151  }
152
153  /*
154
155  Like the explode() function in PHP, the string s is tokenized based

```



```

156 on delim, which is considered as a whole boundary string, not just a
157 sequence of possible boundary characters like the split() function above.
158 This will not skip empty tokens.
159 e.g. explode("a::b", ":") yields {"a", "", "b"}, not {"a", "b"}.
160
161 Time Complexity: O(s.length() * delim.length())
162
163 */
164
165 std::vector<std::string> explode(const std::string & s,
166                               const std::string & delim) {
167     std::vector<std::string> res;
168     unsigned int last = 0, next = 0;
169     while ((next = s.find(delim, last)) != std::string::npos) {
170         res.push_back(s.substr(last, next - last));
171         last = next + delim.size();
172     }
173     res.push_back(s.substr(last));
174     return res;
175 }
176
177 /** Example Usage */
178
179 #include <cassert>
180 #include <cstdio>
181 #include <iostream>
182 using namespace std;
183
184 void print(const vector<string> & v) {
185     cout << "[";
186     for (int i = 0; i < (int)v.size(); i++)
187         cout << (i ? "\",_" : "\"") << v[i];
188     cout << "\"]\n";
189 }
190
191 int main() {
192     assert(to_string(123) + "4" == "1234");
193     assert(to_int("1234") == 1234);
194     char buffer[50];
195     assert(string(itoa(1750, buffer, 10)) == "1750");
196     assert(string(itoa(1750, buffer, 16)) == "6d6");
197     assert(string(itoa(1750, buffer, 2)) == "11011010110");
198
199     string s("___abc_\n");
200     string t = s;
201     assert(ltrim(s) == "abc_\n");
202     assert(rtrim(s) == trim(t));
203     assert(replace("abcdabba", "ab", "00") == "00cd00ba");
204
205     vector<string> tokens;
206
207     tokens = split("a\nb\ncde\nf", '\n');
208     cout << "split_v1:_" ;
209     print(tokens); //[ "a", "b", "cde", "f" ]
210
211     tokens = split("a::b,cde:,f", ":",");
212     cout << "split_v2:_" ;
213     print(tokens); //[ "a", "b", "cde", "f" ]
214

```

```

215     tokens = explode("a..b.cde...f", "..");
216     cout << "explode:␣";
217     print(tokens); //[ "a", ".b.cde", "", ".f" ]
218     return 0;
219 }

```

6.2 Expression Parsing

6.2.1 Recursive Descent

```

1  /*
2
3  Evaluate a mathematical expression in accordance to the order
4  of operations (parentheses, exponents, multiplication, division,
5  addition, subtraction). Does not handle unary operators like '-'.
6
7  */
8
9  /** Example Usage **/
10
11 #include <cctype>
12 #include <cmath>
13 #include <sstream>
14 #include <stdexcept>
15 #include <string>
16
17 class parser {
18     int pos;
19     double tokval;
20     std::string s;
21
22     bool is_dig_or_dot(char c) {
23         return isdigit(c) || c == '.';
24     }
25
26     double to_double(const std::string & s) {
27         std::stringstream ss(s);
28         double res;
29         ss >> res;
30         return res;
31     }
32
33 public:
34     char token;
35
36     parser(const std::string & s) {
37         this->s = s;
38         pos = 0;
39     }
40
41     int next() {
42         for (;;) {
43             if (pos == (int)s.size())
44                 return token = -1;
45             char c = s[pos++];
46             if (std::string("+-*/^()\n").find(c) != std::string::npos)

```

```

47     return token = c;
48     if (isspace(c)) continue;
49     if (isdigit(c) || c == '.') {
50         std::string operand(1, c);
51         while (pos < (int)s.size() && is_dig_or_dot(s[pos]))
52             operand += (c = s[pos++]);
53         tokval = to_double(operand);
54         return token = 'n';
55     }
56     throw std::runtime_error(std::string("Bad character: ") + c);
57 }
58 }
59
60 void skip(int ch) {
61     if (token != ch)
62         throw std::runtime_error(std::string("Bad character: ") + token + std::string(", expected: ") +
63             (char)ch);
64     next();
65 }
66
67 double number() {
68     if (token == 'n') {
69         double v = tokval;
70         skip('n');
71         return v;
72     }
73     skip('(');
74     double v = expression();
75     skip(')');
76     return v;
77 }
78
79 // factor ::= number | number '^' factor
80 double factor() {
81     double v = number();
82     if (token == '^') {
83         skip('^');
84         v = pow(v, factor());
85     }
86     return v;
87 }
88
89 // term ::= factor | term '*' factor | term '/' factor
90 double term() {
91     double v = factor();
92     for (;;) {
93         if (token == '*') {
94             skip('*');
95             v *= factor();
96         } else if (token == '/') {
97             skip('/');
98             v /= factor();
99         } else {
100             return v;
101         }
102     }
103 }
104
105 // expression ::= term | expression '+' term | expression '-' term

```

```

105     double expression() {
106         double v = term();
107         for (;;) {
108             if (token == '+') {
109                 skip('+');
110                 v += term();
111             } else if (token == '-') {
112                 skip('-');
113                 v -= term();
114             } else {
115                 return v;
116             }
117         }
118     }
119 };
120
121 #include <iostream>
122 using namespace std;
123
124 int main() {
125     parser p("1+2*3*4+3*(2+2)-100\n");
126     p.next();
127     while (p.token != -1) {
128         if (p.token == '\n') {
129             p.skip('\n');
130             continue;
131         }
132         cout << p.expression() << "\n";
133     }
134     return 0;
135 }

```

6.2.2 Recursive Descent (Simple)

```

1  /*
2
3  Evaluate a mathematica expression in accordance to the order
4  of operations (parentheses, exponents, multiplication, division,
5  addition, subtraction). This handles unary operators like '-'.
6
7  */
8
9  #include <string>
10
11 template<class It> int eval(It & it, int prec) {
12     if (prec == 0) {
13         int sign = 1, ret = 0;
14         for (; *it == '+'; it++) sign *= -1;
15         if (*it == '(') {
16             ret = eval(++it, 2);
17             it++;
18         } else while (*it >= '0' && *it <= '9') {
19             ret = 10 * ret + (*(it++) - '0');
20         }
21         return sign * ret;
22     }
23     int num = eval(it, prec - 1);

```

```

24     while (!(prec == 2 && *it != '+' && *it != '-') ||
25            (prec == 1 && *it != '*' && *it != '/')) {
26         switch (*(it++)) {
27             case '+': num += eval(it, prec - 1); break;
28             case '-': num -= eval(it, prec - 1); break;
29             case '*': num *= eval(it, prec - 1); break;
30             case '/': num /= eval(it, prec - 1); break;
31         }
32     }
33     return num;
34 }
35
36 /** Wrapper Function */
37
38 int eval(const std::string & s) {
39     std::string::iterator it = std::string(s).begin();
40     return eval(it, 2);
41 }
42
43 /** Example Usage */
44
45 #include <iostream>
46 using namespace std;
47
48 int main() {
49     cout << eval("1+2*3*4+3*(2+2)-100") << "\n";
50     return 0;
51 }

```

6.2.3 Shunting Yard Algorithm

```

1  /*
2
3  Evaluate a mathematica expression in accordance to the order
4  of operations (parentheses, exponents, multiplication, division,
5  addition, subtraction). This also handles unary operators like '-'.
6  We use strings for operators so we can even define things like "sqrt"
7  and "mod" as unary operators by changing prec() and split_expr()
8  accordingly.
9
10 Time Complexity: O(n) on the total number of operators and operands.
11
12 */
13
14 #include <cstdlib>    /* strtol() */
15 #include <stack>
16 #include <stdexcept> /* std::runtime_error */
17 #include <string>
18 #include <vector>
19
20 // Classify the precedences of operators here.
21 inline int prec(const std::string & op, bool unary) {
22     if (unary) {
23         if (op == "+" || op == "-") return 3;
24         return 0; // not a unary operator
25     }
26     if (op == "*" || op == "/") return 2;

```

```

27     if (op == "+" || op == "-") return 1;
28     return 0; // not a binary operator
29 }
30
31 inline int calc1(const std::string & op, int val) {
32     if (op == "+") return +val;
33     if (op == "-") return -val;
34     throw std::runtime_error("Invalid unary operator: " + op);
35 }
36
37 inline int calc2(const std::string & op, int L, int R) {
38     if (op == "+") return L + R;
39     if (op == "-") return L - R;
40     if (op == "*") return L * R;
41     if (op == "/") return L / R;
42     throw std::runtime_error("Invalid binary operator: " + op);
43 }
44
45 inline bool is_operand(const std::string & s) {
46     return s != "(" && s != ")" && !prec(s, 0) && !prec(s, 1);
47 }
48
49 int eval(std::vector<std::string> E) { // E stores the tokens
50     E.insert(E.begin(), "(");
51     E.push_back(")");
52     std::stack<std::pair<std::string, bool> > ops;
53     std::stack<int> vals;
54     for (int i = 0; i < (int)E.size(); i++) {
55         if (is_operand(E[i])) {
56             vals.push(strtol(E[i].c_str(), 0, 10)); // convert to int
57             continue;
58         }
59         if (E[i] == "(") {
60             ops.push(std::make_pair("(", 0));
61             continue;
62         }
63         if (prec(E[i], 1) && (i == 0 || E[i - 1] == "(" || prec(E[i - 1], 0))) {
64             ops.push(std::make_pair(E[i], 1));
65             continue;
66         }
67         while(prec(ops.top().first, ops.top().second) >= prec(E[i], 0)) {
68             std::string op = ops.top().first;
69             bool is_unary = ops.top().second;
70             ops.pop();
71             if (op == "(") break;
72             int y = vals.top(); vals.pop();
73             if (is_unary) {
74                 vals.push(calc1(op, y));
75             } else {
76                 int x = vals.top(); vals.pop();
77                 vals.push(calc2(op, x, y));
78             }
79         }
80         if (E[i] != ")") ops.push(std::make_pair(E[i], 0));
81     }
82     return vals.top();
83 }
84
85 /*

```

```

86
87 Split a string expression to tokens, ignoring whitespace delimiters.
88 A vector of tokens is a more flexible format since you can decide to
89 parse the expression however you wish just by modifying this function.
90 e.g. "1+(51 * -100)" converts to {"1","+","(", "51","*","-", "100","")"}
91
92 */
93
94 std::vector<std::string> split_expr(const std::string &s,
95                                   const std::string &delim = "\n\t\v\f\r") {
96     std::vector<std::string> ret;
97     std::string acc = "";
98     for (int i = 0; i < (int)s.size(); i++)
99         if (s[i] >= '0' && s[i] <= '9') {
100             acc += s[i];
101         } else {
102             if (i > 0 && s[i - 1] >= '0' && s[i - 1] <= '9')
103                 ret.push_back(acc);
104             acc = "";
105             if (delim.find(s[i]) != std::string::npos) continue;
106             ret.push_back(std::string("") + s[i]);
107         }
108     if (s[s.size() - 1] >= '0' && s[s.size() - 1] <= '9')
109         ret.push_back(acc);
110     return ret;
111 }
112
113 int eval(const std::string & s) {
114     return eval(split_expr(s));
115 }
116
117 /** Example Usage */
118
119 #include <iostream>
120 using namespace std;
121
122 int main() {
123     cout << eval("1+2*3*4+3*(2+2)-100") << endl;
124     return 0;
125 }

```

6.3 String Searching

6.3.1 Longest Common Substring

```

1  /*
2
3  Given an text and a pattern to be searched for within the text,
4  determine the first position in which the pattern occurs in
5  the text. The KMP algorithm is much faster than the naive,
6  quadratic time, string searching algorithm that is found in
7  string.find() in the C++ standard library.
8
9  KMP generates a table using a prefix function of the pattern.
10 Then, the precomputed table of the pattern can be used indefinitely
11 for any number of texts.

```

```

12
13 Time Complexity:  $O(n + m)$  where  $n$  is the length of the text
14 and  $m$  is the length of the pattern.
15
16 Space Complexity:  $O(m)$  auxiliary on the length of the pattern.
17
18 */
19
20 #include <string>
21 #include <vector>
22
23 int find(const std::string & text, const std::string & pattern) {
24     if (pattern.empty()) return 0;
25     //generate table using pattern
26     std::vector<int> p(pattern.size());
27     for (int i = 0, j = p[0] = -1; i < (int)pattern.size(); ) {
28         while (j >= 0 && pattern[i] != pattern[j])
29             j = p[j];
30         i++;
31         j++;
32         p[i] = (pattern[i] == pattern[j]) ? p[j] : j;
33     }
34     //use the precomputed table to search within text
35     //the following can be repeated on many different texts
36     for (int i = 0, j = 0; j < (int)text.size(); ) {
37         while (i >= 0 && pattern[i] != text[j])
38             i = p[i];
39         i++;
40         j++;
41         if (i >= (int)pattern.size())
42             return j - i;
43     }
44     return std::string::npos;
45 }
46
47 /** Example Usage */
48
49 #include <cassert>
50
51 int main() {
52     assert(15 == find("ABC_ABCDAB_ABCDABCDABDE", "ABCDABD"));
53     return 0;
54 }

```

6.3.2 Longest Common Subsequence

```

1  /*
2
3  Given a text and multiple patterns to be searched for within the
4  text, simultaneously determine the position of all matches.
5  All of the patterns will be first required for precomputing
6  the automata, after which any input text may be given without
7  having to recompute the automata for the pattern.
8
9  Time Complexity:  $O(n)$  for build_automata(), where  $n$  is the sum of
10 all pattern lengths, and  $O(1)$  amortized for next_state(). However,
11 since it must be called  $m$  times for an input text of length  $m$ , and

```



```

12 if there are z matches throughout the entire text, then the entire
13 algorithm will have a running time of  $O(n + m + z)$ .
14
15 Note that in this implementation, a bitset is used to speed up
16 build_automata() at the cost of making the later text search cost
17  $O(n * m)$ . To truly make the algorithm  $O(n + m + z)$ , bitset must be
18 substituted for an unordered_set, which will not encounter any
19 blank spaces during iteration of the bitset. However, for simply
20 counting the number of matches, bitsets are clearly advantages.
21
22 Space Complexity:  $O(l * c)$ , where l is the sum of all pattern
23 lengths and c is the size of the alphabet.
24
25 */
26
27 #include <bitset>
28 #include <cstring>
29 #include <queue>
30 #include <string>
31 #include <vector>
32
33 const int MAXP = 1000; //maximum number of patterns
34 const int MAXL = 10000; //max possible sum of all pattern lengths
35 const int MAXC = 26;    //size of the alphabet (e.g. 'a'..'z')
36
37 //This function should be customized to return a mapping from
38 //the input alphabet (e.g. 'a'..'z') to the integers 0..MAXC-1
39 inline int map_alphabet(char c) {
40     return (int)(c - 'a');
41 }
42
43 std::bitset<MAXP> out[MAXL]; //std::unordered_set<int> out[MAXL]
44 int fail[MAXL], g[MAXL][MAXC + 1];
45
46 int build_automata(const std::vector<std::string> & patterns) {
47     memset(fail, -1, sizeof fail);
48     memset(g, -1, sizeof g);
49     for (int i = 0; i < MAXL; i++)
50         out[i].reset(); //out[i].clear();
51     int states = 1;
52     for (int i = 0; i < (int)patterns.size(); i++) {
53         const std::string & pattern = patterns[i];
54         int curr = 0;
55         for (int j = 0; j < (int)pattern.size(); j++) {
56             int c = map_alphabet(pattern[j]);
57             if (g[curr][c] == -1)
58                 g[curr][c] = states++;
59             curr = g[curr][c];
60         }
61         out[curr][i] = out[curr][i] | 1; //out[curr].insert(i);
62     }
63     for (int c = 0; c < MAXC; c++)
64         if (g[0][c] == -1) g[0][c] = 0;
65     std::queue<int> q;
66     for (int c = 0; c <= MAXC; c++) {
67         if (g[0][c] != -1 && g[0][c] != 0) {
68             fail[g[0][c]] = 0;
69             q.push(g[0][c]);
70         }

```

```

71     }
72     while (!q.empty()) {
73         int s = q.front(), t;
74         q.pop();
75         for (int c = 0; c <= MAXC; c++) {
76             t = g[s][c];
77             if (t != -1) {
78                 int f = fail[s];
79                 while (g[f][c] == -1)
80                     f = fail[f];
81                 f = g[f][c];
82                 fail[t] = f;
83                 out[t] |= out[f]; //out[t].insert(out[f].begin(), out[f].end());
84                 q.push(t);
85             }
86         }
87     }
88     return states;
89 }
90
91 int next_state(int curr, char ch) {
92     int next = curr, c = map_alphabet(ch);
93     while (g[next][c] == -1)
94         next = fail[next];
95     return g[next][c];
96 }
97
98 /** Example Usage (en.wikipedia.org/wiki/AhoCorasick_algorithm) */
99
100 #include <iostream>
101 using namespace std;
102
103 int main() {
104     vector<string> patterns;
105     patterns.push_back("a");
106     patterns.push_back("ab");
107     patterns.push_back("bab");
108     patterns.push_back("bc");
109     patterns.push_back("bca");
110     patterns.push_back("c");
111     patterns.push_back("caa");
112     build_automata(patterns);
113
114     string text("abccab");
115     int state = 0;
116     for (int i = 0; i < (int)text.size(); i++) {
117         state = next_state(state, text[i]);
118         cout << "Matches_ending_at_position_" << i << " " << endl;
119         if (out[state].any())
120             for (int j = 0; j < (int)out[state].size(); j++)
121                 if (out[state][j])
122                     cout << " " << patterns[j] << " " << endl;
123     }
124     return 0;
125 }

```

6.3.3 Edit Distance

```

1  /*
2
3  Given an text and a pattern to be searched for within the text,
4  determine the positions of all patterns within the text. This
5  is as efficient as KMP, but does so through computing the
6  "Z function." For a string S, Z[i] stores the length of the longest
7  substring starting from S[i] which is also a prefix of S, i.e. the
8  maximum k such that S[j] = S [ i + j ] for all 0 <= j < k .
9
10 Time Complexity: O(n + m) where n is the length of the text
11 and m is the length of the pattern.
12
13 Space Complexity: O(m) auxiliary on the length of the pattern.
14
15 */
16
17 #include <algorithm>
18 #include <string>
19 #include <vector>
20
21 std::vector<int> z_function(const std::string & s) {
22     std::vector<int> z(s.size());
23     for (int i = 1, l = 0, r = 0; i < (int)z.size(); i++) {
24         if (i <= r)
25             z[i] = std::min(r - i + 1, z[i - l]);
26         while (i + z[i] < (int)z.size() && s[z[i]] == s[i + z[i]])
27             z[i]++;
28         if (r < i + z[i] - 1) {
29             l = i;
30             r = i + z[i] - 1;
31         }
32     }
33     return z;
34 }
35
36 /** Example Usage **/
37
38 #include <iostream>
39 using namespace std;
40
41 int main() {
42     string text = "abcabaaaababab";
43     string pattern = "aba";
44     vector<int> z = z_function(pattern + "$" + text);
45     for (int i = (int)pattern.size() + 1; i < (int)z.size(); i++) {
46         if (z[i] == (int)pattern.size())
47             cout << "Pattern found starting at index "
48                 << (i - (int)pattern.size() - 1) << "." << endl;
49     }
50     return 0;
51 }

```

6.4 Dynamic Programming

6.4.1 Longest Common Substring

```

1  /*
2
3  A substring is a consecutive part of a longer string (e.g. "ABC" is
4  a substring of "ABCDE" but "ABD" is not). Using dynamic programming,
5  determine the longest string which is a substring common to any two
6  input strings.
7
8  Time Complexity:  $O(n * m)$  where  $n$  and  $m$  are the lengths of the two
9  input strings, respectively.
10
11 Space Complexity:  $O(\min(n, m))$  auxiliary.
12
13 */
14
15 #include <string>
16
17 std::string longest_common_substring
18 (const std::string & s1, const std::string & s2) {
19     if (s1.empty() || s2.empty()) return "";
20     if (s1.size() < s2.size())
21         return longest_common_substring(s2, s1);
22     int * A = new int[s2.size()];
23     int * B = new int[s2.size()];
24     int startpos = 0, maxlen = 0;
25     for (int i = 0; i < (int)s1.size(); i++) {
26         for (int j = 0; j < (int)s2.size(); j++) {
27             if (s1[i] == s2[j]) {
28                 A[j] = (i > 0 && j > 0) ? 1 + B[j - 1] : 1;
29                 if (maxlen < A[j]) {
30                     maxlen = A[j];
31                     startpos = i - A[j] + 1;
32                 }
33             } else {
34                 A[j] = 0;
35             }
36         }
37         int * temp = A;
38         A = B;
39         B = temp;
40     }
41     delete[] A;
42     delete[] B;
43     return s1.substr(startpos, maxlen);
44 }
45
46 /** Example Usage */
47
48 #include <cassert>
49
50 int main() {
51     assert(longest_common_substring("bbbabca", "aababcd") == "babc");
52     return 0;
53 }

```

6.4.2 Longest Common Subsequence

```

1  /*

```

```

2
3 A subsequence is a sequence that can be derived from another sequence
4 by deleting some elements without changing the order of the remaining
5 elements (e.g. "ACE" is a subsequence of "ABCDE", but "BAE" is not).
6 Using dynamic programming, determine the longest string which
7 is a subsequence common to any two input strings.
8
9 In addition, the shortest common supersequence between two strings is
10 a closely related problem, which involves finding the shortest string
11 which has both input strings as subsequences (e.g. "ABBC" and "BCB" has
12 the shortest common supersequence of "ABBCB"). The answer is simply:
13     (sum of lengths of s1 and s2) - (length of LCS of s1 and s2)
14
15 Time Complexity:  $O(n * m)$  where n and m are the lengths of the two
16 input strings, respectively.
17
18 Space Complexity:  $O(n * m)$  auxiliary.
19
20 */
21
22 #include <string>
23 #include <vector>
24
25 std::string longest_common_subsequence
26 (const std::string & s1, const std::string & s2) {
27     int n = s1.size(), m = s2.size();
28     std::vector< std::vector<int> > dp;
29     dp.resize(n + 1, std::vector<int>(m + 1, 0));
30     for (int i = 0; i < n; i++) {
31         for (int j = 0; j < m; j++) {
32             if (s1[i] == s2[j]) {
33                 dp[i + 1][j + 1] = dp[i][j] + 1;
34             } else if (dp[i + 1][j] > dp[i][j + 1]) {
35                 dp[i + 1][j + 1] = dp[i + 1][j];
36             } else {
37                 dp[i + 1][j + 1] = dp[i][j + 1];
38             }
39         }
40     }
41     std::string ret;
42     for (int i = n, j = m; i > 0 && j > 0; ) {
43         if (s1[i - 1] == s2[j - 1]) {
44             ret = s1[i - 1] + ret;
45             i--;
46             j--;
47         } else if (dp[i - 1][j] < dp[i][j - 1]) {
48             j--;
49         } else {
50             i--;
51         }
52     }
53     return ret;
54 }
55
56 /** Example Usage */
57
58 #include <cassert>
59
60 int main() {

```

```

61     assert(longest_common_subsequence("xmjyauz", "mzjawxu") == "mjau");
62     return 0;
63 }

```

6.4.3 Edit Distance

```

1  /*
2
3  Given two strings s1 and s2, the edit distance between them is the
4  minimum number of operations required to transform s1 into s2,
5  where each operation can be any one of the following:
6      - insert a letter anywhere into the current string
7      - delete any letter from the current string
8      - replace any letter of the current string with any other letter
9
10 Time Complexity: O(n * m) where n and m are the lengths of the two
11 input strings, respectively.
12
13 Space Complexity: O(n * m) auxiliary.
14
15 */
16
17 #include <algorithm>
18 #include <string>
19 #include <vector>
20
21 int edit_distance(const std::string & s1, const std::string & s2) {
22     int n = s1.size(), m = s2.size();
23     std::vector< std::vector<int> > dp;
24     dp.resize(n + 1, std::vector<int>(m + 1, 0));
25     for (int i = 0; i <= n; i++) dp[i][0] = i;
26     for (int j = 0; j <= m; j++) dp[0][j] = j;
27     for (int i = 0; i < n; i++) {
28         for (int j = 0; j < m; j++) {
29             if (s1[i] == s2[j]) {
30                 dp[i + 1][j + 1] = dp[i][j];
31             } else {
32                 dp[i + 1][j + 1] = 1 + std::min(dp[i][j],           //replace
33                                                std::min(dp[i + 1][j], //insert
34                                                         dp[i][j + 1])); //delete
35             }
36         }
37     }
38     return dp[n][m];
39 }
40
41 /** Example Usage */
42
43 #include <cassert>
44
45 int main() {
46     assert(edit_distance("abxdef", "abcdefg") == 2);
47     return 0;
48 }

```

6.5 Suffix Array and LCP

6.5.1 $\mathcal{O}(N \log^2 N)$ Construction

```

1  /*
2
3  A suffix array SA of a string S[1..n] is a sorted array of indices of
4  all the suffixes of S ("abc" has suffixes "abc", "bc", and "c").
5  SA[i] contains the starting position of the i-th smallest suffix in S,
6  ensuring that for all  $1 < i \leq n$ ,  $S[SA[i - 1], n] < S[A[i], n]$  holds.
7  It is a simple, space efficient alternative to suffix trees.
8  By binary searching on a suffix array, one can determine whether a
9  substring exists in a string in  $\mathcal{O}(\log n)$  time per query.
10
11 The longest common prefix array (LCP array) stores the lengths of the
12 longest common prefixes between all pairs of consecutive suffixes in
13 a sorted suffix array and can be found in  $\mathcal{O}(n)$  given the suffix array.
14
15 The following algorithm uses a "gap" partitioning algorithm
16 explained here: http://stackoverflow.com/a/17763563
17
18 Time Complexity:  $\mathcal{O}(n \log^2 n)$  for suffix_array() and  $\mathcal{O}(n)$  for
19 lcp_array(), where n is the length of the input string.
20
21 Space Complexity:  $\mathcal{O}(n)$  auxiliary.
22
23 */
24
25 #include <algorithm>
26 #include <string>
27 #include <vector>
28
29 std::vector<long long> rank2;
30
31 bool comp(const int & a, const int & b) {
32     return rank2[a] < rank2[b];
33 }
34
35 std::vector<int> suffix_array(const std::string & s) {
36     int n = s.size();
37     std::vector<int> sa(n), rank(n);
38     for (int i = 0; i < n; i++) {
39         sa[i] = i;
40         rank[i] = (int)s[i];
41     }
42     rank2.resize(n);
43     for (int len = 1; len < n; len *= 2) {
44         for (int i = 0; i < n; i++)
45             rank2[i] = ((long long)rank[i] << 32) +
46                 (i + len < n ? rank[i + len] + 1 : 0);
47         std::sort(sa.begin(), sa.end(), comp);
48         for (int i = 0; i < n; i++)
49             rank[sa[i]] = (i > 0 && rank2[sa[i - 1]] == rank2[sa[i]]) ?
50                 rank[sa[i - 1]] : i;
51     }
52     return sa;
53 }

```

```

54
55 std::vector<int> lcp_array(const std::string & s,
56                          const std::vector<int> & sa) {
57     int n = sa.size();
58     std::vector<int> rank(n), lcp(n - 1);
59     for (int i = 0; i < n; i++)
60         rank[sa[i]] = i;
61     for (int i = 0, h = 0; i < n; i++) {
62         if (rank[i] < n - 1) {
63             int j = sa[rank[i] + 1];
64             while (std::max(i, j) + h < n && s[i + h] == s[j + h])
65                 h++;
66             lcp[rank[i]] = h;
67             if (h > 0) h--;
68         }
69     }
70     return lcp;
71 }
72
73 /** Example Usage **/
74
75 #include <cassert>
76 using namespace std;
77
78 int main() {
79     string s("banana");
80     vector<int> sa = suffix_array(s);
81     vector<int> lcp = lcp_array(s, sa);
82     int sa_ans[] = {5, 3, 1, 0, 4, 2};
83     int lcp_ans[] = {1, 3, 0, 0, 2};
84     assert(equal(sa.begin(), sa.end(), sa_ans));
85     assert(equal(lcp.begin(), lcp.end(), lcp_ans));
86     return 0;
87 }

```

6.5.2 $\mathcal{O}(N \log N)$ Construction

```

1  /*
2
3  A suffix array SA of a string S[1..n] is a sorted array of indices of
4  all the suffixes of S ("abc" has suffixes "abc", "bc", and "c").
5  SA[i] contains the starting position of the i-th smallest suffix in S,
6  ensuring that for all  $1 < i \leq n$ ,  $S[SA[i - 1], n] < S[SA[i], n]$  holds.
7  It is a simple, space efficient alternative to suffix trees.
8  By binary searching on a suffix array, one can determine whether a
9  substring exists in a string in  $\mathcal{O}(\log n)$  time per query.
10
11 The longest common prefix array (LCP array) stores the lengths of the
12 longest common prefixes between all pairs of consecutive suffixes in
13 a sorted suffix array and can be found in  $\mathcal{O}(n)$  given the suffix array.
14
15 The following algorithm uses a "gap" partitioning algorithm
16 explained here: http://stackoverflow.com/a/17763563, except that the
17  $\mathcal{O}(n \log n)$  comparison-based sort is substituted for an  $\mathcal{O}(n)$  counting
18 sort to reduce the running time by an order of  $\log n$ .
19
20 Time Complexity:  $\mathcal{O}(n \log n)$  for suffix_array() and  $\mathcal{O}(n)$  for

```



```

21 lcp_array(), where n is the length of the input string.
22
23 Space Complexity:  $O(n)$  auxiliary.
24
25 */
26
27 #include <algorithm>
28 #include <string>
29 #include <vector>
30
31 const std::string * str;
32
33 bool comp(const int & a, const int & b) {
34     return (*str)[a] < (*str)[b];
35 }
36
37 std::vector<int> suffix_array(const std::string & s) {
38     int n = s.size();
39     std::vector<int> sa(n), order(n), rank(n);
40     for (int i = 0; i < n; i++)
41         order[i] = n - 1 - i;
42     str = &s;
43     std::stable_sort(order.begin(), order.end(), comp);
44     for (int i = 0; i < n; i++) {
45         sa[i] = order[i];
46         rank[i] = (int)s[i];
47     }
48     std::vector<int> r(n), cnt(n), _sa(n);
49     for (int len = 1; len < n; len *= 2) {
50         r = rank;
51         _sa = sa;
52         for (int i = 0; i < n; i++)
53             cnt[i] = i;
54         for (int i = 0; i < n; i++) {
55             if (i > 0 && r[sa[i - 1]] == r[sa[i]] && sa[i - 1] + len < n &&
56                 r[sa[i - 1] + len / 2] == r[sa[i] + len / 2]) {
57                 rank[sa[i]] = rank[sa[i - 1]];
58             } else {
59                 rank[sa[i]] = i;
60             }
61         }
62         for (int i = 0; i < n; i++) {
63             int s1 = _sa[i] - len;
64             if (s1 >= 0)
65                 sa[cnt[rank[s1]]++] = s1;
66         }
67     }
68     return sa;
69 }
70
71 std::vector<int> lcp_array(const std::string & s,
72                           const std::vector<int> & sa) {
73     int n = sa.size();
74     std::vector<int> rank(n), lcp(n - 1);
75     for (int i = 0; i < n; i++)
76         rank[sa[i]] = i;
77     for (int i = 0, h = 0; i < n; i++) {
78         if (rank[i] < n - 1) {
79             int j = sa[rank[i] + 1];

```

```

80     while (std::max(i, j) + h < n && s[i + h] == s[j + h])
81         h++;
82     lcp[rank[i]] = h;
83     if (h > 0) h--;
84 }
85 }
86 return lcp;
87 }
88
89 /** Example Usage */
90
91 #include <cassert>
92 using namespace std;
93
94 int main() {
95     string s("banana");
96     vector<int> sa = suffix_array(s);
97     vector<int> lcp = lcp_array(s, sa);
98     int sa_ans[] = {5, 3, 1, 0, 4, 2};
99     int lcp_ans[] = {1, 3, 0, 0, 2};
100     assert(equal(sa.begin(), sa.end(), sa_ans));
101     assert(equal(lcp.begin(), lcp.end(), lcp_ans));
102     return 0;
103 }

```

6.5.3 $O(N \log N)$ Construction (DC3/Skew)

```

1  /*
2
3  A suffix array SA of a string S[1, n] is a sorted array of indices of
4  all the suffixes of S ("abc" has suffixes "abc", "bc", and "c").
5  SA[i] contains the starting position of the i-th smallest suffix in S,
6  ensuring that for all  $1 < i \leq n$ ,  $S[SA[i - 1], n] < S[SA[i], n]$  holds.
7  It is a simple, space efficient alternative to suffix trees.
8  By binary searching on a suffix array, one can determine whether a
9  substring exists in a string in  $O(\log n)$  time per query.
10
11 The longest common prefix array (LCP array) stores the lengths of the
12 longest common prefixes between all pairs of consecutive suffixes in
13 a sorted suffix array and can be found in  $O(n)$  given the suffix array.
14
15 The following implementation uses the sophisticated DC3/skew algorithm
16 by Karkkainen & Sanders (2003), using radix sort on integer alphabets
17 for linear construction. The function suffix_array(s, SA, n, K) takes
18 in s, an array [0, n - 1] of ints with n values in the range [1, K].
19 It stores the indices defining the suffix array into SA. The last value
20 of the input array s[n - 1] must be equal 0, the sentinel character. A
21 C++ wrapper function suffix_array(std::string) is implemented below it.
22
23 Time Complexity:  $O(n)$  for suffix_array() and lcp_array(), where n is
24 the length of the input string.
25
26 Space Complexity:  $O(n)$  auxiliary.
27
28 */
29
30 inline bool leq(int a1, int a2, int b1, int b2) {

```

```

31     return a1 < b1 || (a1 == b1 && a2 <= b2);
32 }
33
34 inline bool leq(int a1, int a2, int a3, int b1, int b2, int b3) {
35     return a1 < b1 || (a1 == b1 && leq(a2, a3, b2, b3));
36 }
37
38 static void radix_pass(int * a, int * b, int * r, int n, int K) {
39     int *c = new int[K + 1];
40     for (int i = 0; i <= K; i++)
41         c[i] = 0;
42     for (int i = 0; i < n; i++)
43         c[r[a[i]]]++;
44     for (int i = 0, sum = 0; i <= K; i++) {
45         int tmp = c[i];
46         c[i] = sum;
47         sum += tmp;
48     }
49     for (int i = 0; i < n; i++)
50         b[c[r[a[i]]]++] = a[i];
51     delete[] c;
52 }
53
54 void suffix_array(int * s, int * sa, int n, int K) {
55     int n0 = (n + 2) / 3, n1 = (n + 1) / 3, n2 = n / 3, n02 = n0 + n2;
56     int *s12 = new int[n02 + 3], *SA12 = new int[n02 + 3];
57     s12[n02] = s12[n02 + 1] = s12[n02 + 2] = 0;
58     SA12[n02] = SA12[n02 + 1] = SA12[n02 + 2] = 0;
59     int *s0 = new int[n0], *SA0 = new int[n0];
60     for (int i = 0, j = 0; i < n + n0 - n1; i++)
61         if (i % 3 != 0) s12[j++] = i;
62     radix_pass(s12, SA12, s + 2, n02, K);
63     radix_pass(SA12, s12, s + 1, n02, K);
64     radix_pass(s12, SA12, s, n02, K);
65     int name = 0, c0 = -1, c1 = -1, c2 = -1;
66     for (int i = 0; i < n02; i++) {
67         if (s[SA12[i]] != c0 || s[SA12[i] + 1] != c1 || s[SA12[i] + 2] != c2) {
68             name++;
69             c0 = s[SA12[i]];
70             c1 = s[SA12[i] + 1];
71             c2 = s[SA12[i] + 2];
72         }
73         if (SA12[i] % 3 == 1)
74             s12[SA12[i] / 3] = name;
75         else
76             s12[SA12[i] / 3 + n0] = name;
77     }
78     if (name < n02) {
79         suffix_array(s12, SA12, n02, name);
80         for (int i = 0; i < n02; i++)
81             s12[SA12[i]] = i + 1;
82     } else {
83         for (int i = 0; i < n02; i++)
84             SA12[s12[i] - 1] = i;
85     }
86     for (int i = 0, j = 0; i < n02; i++)
87         if (SA12[i] < n0)
88             s0[j++] = 3 * SA12[i];
89     radix_pass(s0, SA0, s, n0, K);

```

```

90 #define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
91 for (int p = 0, t = n0 - n1, k = 0; k < n; k++) {
92     int i = GetI(), j = SA0[p];
93     if (SA12[t] < n0 ? leq(s[i], s12[SA12[t] + n0], s[j], s12[j/3]) :
94         leq(s[i], s[i + 1], s12[SA12[t] - n0 + 1], s[j], s[j + 1], s12[j / 3 + n0])) {
95         sa[k] = i;
96         if (++t == n02)
97             for (k++; p < n0; p++, k++)
98                 sa[k] = SA0[p];
99     } else {
100         sa[k] = j;
101         if (++p == n0)
102             for (k++; t < n02; t++, k++)
103                 sa[k] = GetI();
104     }
105 }
106 #undef GetI
107 delete[] s12;
108 delete[] SA12;
109 delete[] SA0;
110 delete[] s0;
111 }
112
113 #include <string>
114 #include <vector>
115
116 // C++ wrapper function
117 std::vector<int> suffix_array(const std::string & s) {
118     int n = s.size();
119     int *str = new int[n + 5], *sa = new int[n + 1];
120     for (int i = 0; i < n + 5; i++) str[i] = 0;
121     for (int i = 0; i < n; i++) str[i] = (int)s[i];
122     suffix_array(str, sa, n + 1, 256);
123     return std::vector<int>(sa + 1, sa + n + 1);
124 }
125
126 std::vector<int> lcp_array(const std::string & s,
127                          const std::vector<int> & sa) {
128     int n = sa.size();
129     std::vector<int> rank(n), lcp(n - 1);
130     for (int i = 0; i < n; i++)
131         rank[sa[i]] = i;
132     for (int i = 0, h = 0; i < n; i++) {
133         if (rank[i] < n - 1) {
134             int j = sa[rank[i] + 1];
135             while (std::max(i, j) + h < n && s[i + h] == s[j + h])
136                 h++;
137             lcp[rank[i]] = h;
138             if (h > 0) h--;
139         }
140     }
141     return lcp;
142 }
143
144 /** Example Usage */
145
146 #include <cassert>
147 using namespace std;
148

```

```

149 int main() {
150     string s("banana");
151     vector<int> sa = suffix_array(s);
152     vector<int> lcp = lcp_array(s, sa);
153     int sa_ans[] = {5, 3, 1, 0, 4, 2};
154     int lcp_ans[] = {1, 3, 0, 0, 2};
155     assert(equal(sa.begin(), sa.end(), sa_ans));
156     assert(equal(lcp.begin(), lcp.end(), lcp_ans));
157     return 0;
158 }

```

6.6 String Data Structures

6.5.1 Simple Trie

```

1  /*
2
3  A trie, digital tree, or prefix tree, is an ordered tree data
4  structure that is used to store a dynamic set or associative array
5  where the keys are strings. Each leaf node represents a string that
6  has been inserted into the trie. This makes tries easier to implement
7  than balanced binary search trees, and also potentially faster.
8
9  Time Complexity:  $O(n)$  for insert(), contains(), and erase(), where
10 n is the length of the string being inserted, searched, or erased.
11
12 Space Complexity: At worst  $O(l * \text{ALPHABET\_SIZE})$ , where l is the
13 sum of all lengths of strings that have been inserted so far.
14
15 */
16
17 #include <string>
18
19 class trie {
20     static const int ALPHABET_SIZE = 26;
21
22     static int map_alphabet(char c) {
23         return (int)(c - 'a');
24     }
25
26     struct node_t {
27         bool leaf;
28
29         node_t * children[ALPHABET_SIZE];
30
31         node_t(): leaf(false) {
32             for (int i = 0; i < ALPHABET_SIZE; i++)
33                 children[i] = 0;
34         }
35
36         bool is_free() {
37             for (int i = 0; i < ALPHABET_SIZE; i++)
38                 if (this->children[i] != 0) return true;
39             return false;
40         }
41     } *root;

```

```

42
43 bool erase(const std::string & s, node_t * n, int depth) {
44     if (n == 0) return false;
45     if (depth == (int)s.size()) {
46         if (n->leaf) {
47             n->leaf = false;
48             return n->is_free();
49         }
50     } else {
51         int idx = map_alphabet(s[depth]);
52         if (erase(s, n->children[idx], depth + 1)) {
53             delete n->children[idx];
54             return !n->leaf && n->is_free();
55         }
56     }
57     return false;
58 }
59
60 static void clean_up(node_t * n) {
61     if (n == 0 || n->leaf) return;
62     for (int i = 0; i < ALPHABET_SIZE; i++)
63         clean_up(n->children[i]);
64     delete n;
65 }
66
67 public:
68     trie() { root = new node_t(); }
69     ~trie() { clean_up(root); }
70
71     void insert(const std::string & s) {
72         node_t * n = root;
73         for (int i = 0; i < (int)s.size(); i++) {
74             int c = map_alphabet(s[i]);
75             if (n->children[c] == 0)
76                 n->children[c] = new node_t();
77             n = n->children[c];
78         }
79         n->leaf = true;
80     }
81
82     bool contains(const std::string & s) {
83         node_t *n = root;
84         for (int i = 0; i < (int)s.size(); i++) {
85             int c = map_alphabet(s[i]);
86             if (n->children[c] == 0)
87                 return false;
88             n = n->children[c];
89         }
90         return n != 0 && n->leaf;
91     }
92
93     bool erase(const std::string & s) {
94         return erase(s, root, 0);
95     }
96 };
97
98 /** Example Usage */
99
100 #include <cassert>

```

```

101 using namespace std;
102
103 int main() {
104     string s[8] = {"a", "to", "tea", "ted", "ten", "i", "in", "inn"};
105     trie t;
106     for (int i = 0; i < 8; i++)
107         t.insert(s[i]);
108     assert(t.contains("ten"));
109     t.erase("tea");
110     assert(!t.contains("tea"));
111     return 0;
112 }

```

6.5.2 Radix Trie

```

1  /*
2
3  A radix tree, radix trie, patricia trie, or compressed trie is a
4  data structure that is used to store a dynamic set or associative
5  array where the keys are strings. Each leaf node represents a string
6  that has been inserted into the trie. Unlike simple tries, radix
7  tries are space-optimized by merging each node that is an only child
8  with its parent.
9
10 Time Complexity:  $O(n)$  for insert(), contains(), and erase(), where
11 n is the length of the string being inserted, searched, or erased.
12
13 Space Complexity: At worst  $O(l)$ , where l is the sum of all lengths
14 of strings that have been inserted so far.
15
16 */
17
18 #include <string>
19 #include <vector>
20
21 class radix_trie {
22     struct node_t {
23         std::string label;
24         std::vector<node_t*> children;
25
26         node_t(const std::string & s = "") {
27             label = s;
28         }
29     } *root;
30
31     unsigned int lcpLen(const std::string & s, const std::string & t) {
32         int minsize = (t.size() < s.size()) ? t.size() : s.size();
33         if (minsize == 0) return 0;
34         unsigned int res = 0;
35         for (int i = 0; i < minsize && s[i] == t[i]; i++)
36             res++;
37         return res;
38     }
39
40     void insert(const std::string & s, node_t * n) {
41         unsigned int lcp = lcpLen(s, n->label);
42         if (lcp == 0 || n == root ||

```

```

43     (lcp > 0 && lcp < s.size() && lcp >= n->label.size())) {
44     bool inserted = false;
45     std::string newstr = s.substr(lcp, s.size() - lcp);
46     for (int i = 0; i < (int)n->children.size(); i++) {
47         if (n->children[i]->label[0] == newstr[0]) {
48             inserted = true;
49             insert(newstr, n->children[i]);
50         }
51     }
52     if (!inserted)
53         n->children.push_back(new node_t(newstr));
54 } else if (lcp < s.size()) {
55     node_t * t = new node_t();
56     t->label = n->label.substr(lcp, n->label.size() - lcp);
57     t->children.assign(n->children.begin(), n->children.end());
58     n->label = s.substr(0, lcp);
59     n->children.assign(1, t);
60     n->children.push_back(new node_t(s.substr(lcp, s.size() - lcp)));
61 }
62 }
63
64 void erase(const std::string & s, node_t * n) {
65     unsigned int lcp = lcplen(s, n->label);
66     if (lcp == 0 || n == root ||
67         (lcp > 0 && lcp < s.size() && lcp >= n->label.size())) {
68         std::string newstr = s.substr(lcp, s.size() - lcp);
69         for (int i = 0; i < (int)n->children.size(); i++) {
70             if (n->children[i]->label[0] == newstr[0]) {
71                 if (newstr == n->children[i]->label &&
72                     n->children[i]->children.empty()) {
73                     n->children.erase(n->children.begin() + i);
74                     return;
75                 }
76                 erase(newstr, n->children[i]);
77             }
78         }
79     }
80 }
81
82 bool contains(const std::string & s, node_t * n) {
83     unsigned int lcp = lcplen(s, n->label);
84     if (lcp == 0 || n == root ||
85         (lcp > 0 && lcp < s.size() && lcp >= n->label.size())) {
86         std::string newstr = s.substr(lcp, s.size() - lcp);
87         for (int i = 0; i < (int)n->children.size(); i++)
88             if (n->children[i]->label[0] == newstr[0])
89                 return contains(newstr, n->children[i]);
90         return false;
91     }
92     return lcp == n->label.size();
93 }
94
95 static void clean_up(node_t * n) {
96     if (n == 0) return;
97     for (int i = 0; i < (int)n->children.size(); i++)
98         clean_up(n->children[i]);
99     delete n;
100 }
101

```



```

102 public:
103     template <class UnaryFunction>
104     void walk(node_t * n, UnaryFunction f) {
105         if (n == 0) return;
106         if (n != root) f(n->label);
107         for (int i = 0; i < (int)n->children.size(); i++)
108             walk(n->children[i], f);
109     }
110
111     radix_trie() { root = new node_t(); }
112     ~radix_trie() { clean_up(root); }
113
114     void insert(const std::string & s) { insert(s, root); }
115     void erase(const std::string & s) { erase(s, root); }
116     bool contains(const std::string & s) { return contains(s, root); }
117
118     template <class UnaryFunction> void walk(UnaryFunction f) {
119         walk(root, f);
120     }
121 };
122
123 /** Example Usage */
124
125 #include <cassert>
126 using namespace std;
127
128 string preorder;
129
130 void concat(const string & s) {
131     preorder += (s + "_");
132 }
133
134 int main() {
135     {
136         string s[8] = {"a", "to", "tea", "ted", "ten", "i", "in", "inn"};
137         radix_trie t;
138         for (int i = 0; i < 8; i++)
139             t.insert(s[i]);
140         assert(t.contains("ten"));
141         t.erase("tea");
142         assert(!t.contains("tea"));
143     }
144     {
145         radix_trie t;
146         t.insert("test");
147         t.insert("toaster");
148         t.insert("toasting");
149         t.insert("slow");
150         t.insert("slowly");
151         preorder = "";
152         t.walk(concat);
153         assert(preorder == "t_est_oast_er_ing_slow_ly_");
154     }
155     return 0;
156 }

```

6.5.3 Suffix Trie


```

60         node = ch;
61         ch = ch->children[c[i - tail]];
62     }
63     if (ch == 0) {
64         node->children[c[i]] = new node_t(i, n,
65                                         node->depth + node->end - node->begin, node);
66         if (last != 0) last->suffix_link = node;
67         last = 0;
68     } else {
69         int aftertail = c[ch->begin + tail];
70         if (aftertail == c[i]) {
71             if (last != 0) last->suffix_link = node;
72             break;
73         } else {
74             node_t *split = new node_t(ch->begin, ch->begin + tail,
75                                       node->depth + node->end - node->begin, node);
76             split->children[c[i]] = new node_t(i, n, ch->depth + tail, split);
77             split->children[aftertail] = ch;
78             ch->begin += tail;
79             ch->depth += tail;
80             ch->parent = split;
81             node->children[c[i - tail]] = split;
82             if (last != 0)
83                 last->suffix_link = split;
84             last = split;
85         }
86     }
87     if (node == root) {
88         tail--;
89     } else {
90         node = node->suffix_link;
91     }
92 }
93 }
94 }
95 };
96
97 int lcs_begin, lcs_len;
98
99 int lcs_rec(suffix_tree::node_t * n, int i1, int i2) {
100     if (n->begin <= i1 && i1 < n->end) return 1;
101     if (n->begin <= i2 && i2 < n->end) return 2;
102     int mask = 0;
103     for (int i = 0; i < suffix_tree::ALPHABET_SIZE; i++) {
104         if (n->children[i] != 0)
105             mask |= lcs_rec(n->children[i], i1, i2);
106     }
107     if (mask == 3) {
108         int curr_len = n->depth + n->end - n->begin;
109         if (lcs_len < curr_len) {
110             lcs_len = curr_len;
111             lcs_begin = n->begin;
112         }
113     }
114     return mask;
115 }
116
117 std::string longest_common_substring
118 (const std::string & s1, const std::string & s2) {

```

```

119     std::string s(s1 + '\01' + s2 + '\02');
120     suffix_tree tree(s);
121     lcs_begin = lcs_len = 0;
122     lcs_rec(tree.root, s1.size(), s1.size() + s2.size() + 1);
123     return s.substr(lcs_begin - 1, lcs_len);
124 }
125
126 /** Example Usage */
127
128 #include <cassert>
129
130 int main() {
131     assert(longest_common_substring("bbbabca", "aababcd") == "babc");
132     return 0;
133 }

```

6.5.4 Suffix Automaton

```

1  /*
2
3  A suffix automaton is a data structure to efficiently represent the
4  suffixes of a string. It can be considered a compressed version of
5  a suffix tree. The data structure supports querying for substrings
6  within the text from with the automaton is constructed in linear
7  time. It also supports computation of the longest common substring
8  in linear time.
9
10 Time Complexity: O(n * ALPHABET_SIZE) for construction, and O(n)
11 for find_all(), as well as longest_common_substring().
12
13 Space Complexity: O(n * ALPHABET_SIZE) auxiliary.
14
15 */
16
17 #include <algorithm>
18 #include <queue>
19 #include <string>
20 #include <vector>
21
22 struct suffix_automaton {
23
24     static const int ALPHABET_SIZE = 26;
25
26     static int map_alphabet(char c) {
27         return (int)(c - 'a');
28     }
29
30     struct state_t {
31         int length, suffix_link;
32         int firstpos, next[ALPHABET_SIZE];
33         std::vector<int> invlinks;
34
35         state_t() {
36             length = 0;
37             suffix_link = 0;
38             firstpos = -1;
39             for (int i = 0; i < ALPHABET_SIZE; i++)

```

```

40     next[i] = -1;
41 }
42 };
43
44 std::vector<state_t> states;
45
46 suffix_automaton(const std::string & s) {
47     int n = s.size();
48     states.resize(std::max(2, 2 * n - 1));
49     states[0].suffix_link = -1;
50     int last = 0;
51     int size = 1;
52     for (int i = 0; i < n; i++) {
53         int c = map_alphabet(s[i]);
54         int curr = size++;
55         states[curr].length = i + 1;
56         states[curr].firstpos = i;
57         int p = last;
58         while (p != -1 && states[p].next[c] == -1) {
59             states[p].next[c] = curr;
60             p = states[p].suffix_link;
61         }
62         if (p == -1) {
63             states[curr].suffix_link = 0;
64         } else {
65             int q = states[p].next[c];
66             if (states[p].length + 1 == states[q].length) {
67                 states[curr].suffix_link = q;
68             } else {
69                 int clone = size++;
70                 states[clone].length = states[p].length + 1;
71                 for (int i = 0; i < ALPHABET_SIZE; i++)
72                     states[clone].next[i] = states[q].next[i];
73                 states[clone].suffix_link = states[q].suffix_link;
74                 while (p != -1 && states[p].next[c] == q) {
75                     states[p].next[c] = clone;
76                     p = states[p].suffix_link;
77                 }
78                 states[q].suffix_link = clone;
79                 states[curr].suffix_link = clone;
80             }
81         }
82         last = curr;
83     }
84     for (int i = 1; i < size; i++)
85         states[states[i].suffix_link].invlinks.push_back(i);
86     states.resize(size);
87 }
88
89 std::vector<int> find_all(const std::string & s) {
90     std::vector<int> res;
91     int node = 0;
92     for (int i = 0; i < (int)s.size(); i++) {
93         int next = states[node].next[map_alphabet(s[i])];
94         if (next == -1) return res;
95         node = next;
96     }
97     std::queue<int> q;
98     q.push(node);

```

```

99     while (!q.empty()) {
100         int curr = q.front();
101         q.pop();
102         if (states[curr].firstpos != -1)
103             res.push_back(states[curr].firstpos - (int)s.size() + 1);
104         for (int j = 0; j < (int)states[curr].invlinks.size(); j++)
105             q.push(states[curr].invlinks[j]);
106     }
107     return res;
108 }
109
110 std::string longest_common_substring(const std::string & s) {
111     int len = 0, bestlen = 0, bestpos = -1;
112     for (int i = 0, cur = 0; i < (int)s.size(); i++) {
113         int c = map_alphabet(s[i]);
114         if (states[cur].next[c] == -1) {
115             while (cur != -1 && states[cur].next[c] == -1)
116                 cur = states[cur].suffix_link;
117             if (cur == -1) {
118                 cur = len = 0;
119                 continue;
120             }
121             len = states[cur].length;
122         }
123         len++;
124         cur = states[cur].next[c];
125         if (bestlen < len) {
126             bestlen = len;
127             bestpos = i;
128         }
129     }
130     return s.substr(bestpos - bestlen + 1, bestlen);
131 }
132 };
133
134 /** Example Usage */
135
136 #include <algorithm>
137 #include <cassert>
138 using namespace std;
139
140 int main() {
141     {
142         suffix_automaton sa("bananas");
143         vector<int> pos_a, pos_an, pos_ana;
144         int ans_a[] = {1, 3, 5};
145         int ans_an[] = {1, 3};
146         int ans_ana[] = {1, 3};
147         pos_a = sa.find_all("a");
148         pos_an = sa.find_all("an");
149         pos_ana = sa.find_all("ana");
150         assert(equal(pos_a.begin(), pos_a.end(), ans_a));
151         assert(equal(pos_an.begin(), pos_an.end(), ans_an));
152         assert(equal(pos_ana.begin(), pos_ana.end(), ans_ana));
153     }
154     {
155         suffix_automaton sa("bbbabca");
156         assert(sa.longest_common_substring("aababcd") == "babc");
157     }

```

```
158     return 0;  
159 }
```