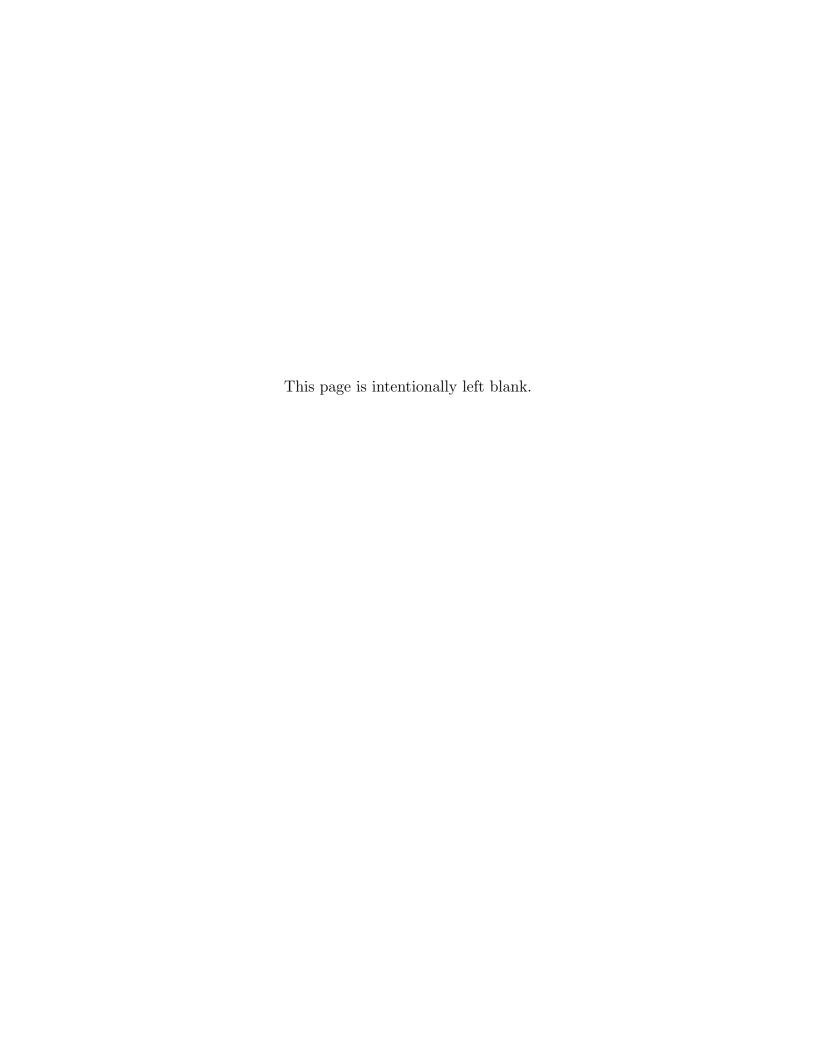# Alex's Anthology of Algorithms

Common Code for Contests in Concise C++

(Incomplete Draft, October 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/Algorithms-Anthology` for the most up-to-date digital version of this code book. The current version you are reading has yet to be completed in terms of content (the final chapter 6 is in progress). It is also yet to be thoroughly proofread for mistakes.

0.1   Introduction

This anthology started as a personal project to implement classic algorithms in the most concise and "vanilla" way possible while minimizing code redundancy and obfuscation. I tried to make the most appropriate trade-off between clarity, portability, efficiency, and concision. Short descriptions of what the programs do along with their time (and space, where necessary) complexities are included. If these are insufficient to help you understand, then you should be able to find extra information by referring to the code's comments and looking online. I do not guarantee that the programs are bug-free - it is always best to carefully study the usage notes and examples first, especially before you make your own modifications.

C++ is my language of choice because of its predominance in competitions. The International Olympiad in Informatics (and practically every other programming contest) accepts solutions to tasks in C++. C++ is a fast, flexible language with a sizable standard library and support for useful features like built-in memory management and object-oriented programming. In an attempt to focus less on reinventing the wheel and more on the essence of the algorithms themselves, these implementations will attempt to take advantage of useful features of the C++ language and library (iterators, templates, functions in algorithm library, STL data structures, etc.) whenever possible.

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-puned pointers. If you are looking for these features, chances are you don't care about portability anyway.

0.3   Usage Guidelines

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 by yourself. Treat this as a last resort, if you will, when you just need a concise and working implementation in the heat of a programming 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 examine any corresponding references to each program for more details on usage and authorship before you use it.

Cheers and hope you enjoy!

— Alex Li

October, 2015

Contents

4   Mathematics   147

5 Geometry

6 Strings

Chapter 1

Elementary Algorithms

1.1 Array Transformations

1.1.1 Sorting Algorithms

```
1   /*
2
3   1.1.1 - Sorting Algorithms
4
5   The following functions behave just like the sorting functions
6   found in the C++ standard library, taking RandomAccessIterators
7   as the range to be sorted, and optionally a comparator.
8
9   */
10
11  #include <algorithm>  /* std::(stable_)partition, std::swap */
12  #include <functional> /* std::less<T> */
13  #include <iterator>   /* std::iterator_traits<T> */
14
15  /*
16
17  Quicksort
18
19  Best: O(N*log(N))    Average: O(N*log(N))    Worst: O(N^2)
20       Auxiliary Space: O(log(N))          Stable?: No
21
22  Notes: The pivot value chosen here is always half way between
23  lo and hi. Choosing random pivot values reduces the chances of
24  encountering the worst case performance of O(N^2). Quicksort is
25  faster in practice than other O(N*log(N)) algorithms.
26
27  */
28
29  template<class It, class Compare>
30  void quicksort(It lo, It hi, Compare cmp) {
31    if (lo + 1 >= hi) return;
32    It i, j, pivot = lo + (hi - lo) / 2;
33    for (i = lo, j = hi - 1; ; ++i, ++j) {
34      while (cmp(*i, *pivot)) ++i;
35      while (cmp(*pivot, *j)) --j;
```

```
36       if (i >= j) break;
37       std::swap(*i, *j);
38     }
39     quicksort(lo, i, cmp);
40     quicksort(i, hi, cmp);
41   }
42
43   template<class It> void quicksort(It lo, It hi) {
44     typedef typename std::iterator_traits<It>::value_type T;
45     quicksort(lo, hi, std::less<T>());
46   }
47
48   /*
49
50   Merge Sort
51
52   Best: O(N*log(N))    Average: O(N*log(N))    Worst: O(N*log(N))
53       Auxiliary Space: O(N)               Stable?: Yes
54
55   Notes: Merge sort has a better worse case complexity than
56   quicksort and is stable (i.e. it maintains the relative order
57   of equivalent values after the sort). The implementation below
58   is different from std::stable_sort() in that it will simply fail
59   if extra memory is not available. Meanwhile, std::stable_sort()
60   will fall back to a run time of O(N*log^2(N)) if out of memory.
61
62   */
63
64   template<class It, class Compare>
65   void mergesort(It lo, It hi, Compare cmp) {
66     typedef typename std::iterator_traits<It>::value_type T;
67     if (lo >= hi - 1) return;
68     It mid = lo + (hi - lo - 1) / 2, a = lo, c = mid + 1;
69     mergesort(lo, mid + 1, cmp);
70     mergesort(mid + 1, hi, cmp);
71     T buf[hi - lo], *b = buf;
72     while (a <= mid && c < hi)
73       *(b++) = cmp(*c, *a) ? *(c++) : *(a++);
74     if (a > mid)
75       for (It k = c; k < hi; k++) *(b++) = *k;
76     else
77       for (It k = a; k <= mid; k++) *(b++) = *k;
78     for (int i = hi - lo - 1; i >= 0; i--)
79       *(lo + i) = buf[i];
80   }
81
82   template<class It> void mergesort(It lo, It hi) {
83     typedef typename std::iterator_traits<It>::value_type T;
84     mergesort(lo, hi, std::less<T>());
85   }
86
87   /*
88
89   Heap Sort
90
91   Best: O(N*log(N))    Average: O(N*log(N))    Worst: O(N*log(N))
92       Auxiliary Space: O(1)               Stable?: No
93
94   Heap sort has a better worst case complexity than quicksort,
```

```
95    but also a better space complexity than merge sort. On average,
96    this will likely run slower than a well implemented quicksort.
97
98    */
99
100   template<class It, class Compare>
101   void heapsort(It lo, It hi, Compare cmp) {
102     typename std::iterator_traits<It>::value_type t;
103     It n = hi, i = lo + (n - lo) / 2, parent, child;
104     for (;;) {
105       if (i <= lo) {
106         if (--n == lo) return;
107         t = *n, *n = *lo;
108       } else t = *(--i);
109       parent = i, child = lo + 2 * (i - lo) + 1;
110       while (child < n) {
111         if (child + 1 < n && cmp(*child, *(child + 1))) child++;
112         if (!cmp(t, *child)) break;
113         *parent = *child, parent = child;
114         child = lo + 2 * (parent - lo) + 1;
115       }
116       *(lo + (parent - lo)) = t;
117     }
118   }
119
120   template<class It> void heapsort(It lo, It hi) {
121     typedef typename std::iterator_traits<It>::value_type T;
122     heapsort(lo, hi, std::less<T>());
123   }
124
125   /*
126
127   Comb Sort
128
129   Best: O(N)        Average: O(N^2/(2^p)) for p increments of gap
130      Worst: O(N^2)     Auxiliary Space: O(1)      Stable?: No
131
132   Comb sort is an improved bubble sort that's simple to memorize.
133
134   */
135
136   template<class It, class Compare>
137   void combsort(It lo, It hi, Compare cmp) {
138     int gap = hi - lo, swapped = 1;
139     while (gap > 1 || swapped) {
140       if (gap > 1) gap = (int)((float)gap / 1.3f);
141       swapped = 0;
142       for (It i = lo; i + gap < hi; i++)
143         if (cmp(*(i + gap), *i)) {
144           std::swap(*i, *(i + gap));
145           swapped = 1;
146         }
147     }
148   }
149
150   template<class It> void combsort(It lo, It hi) {
151     typedef typename std::iterator_traits<It>::value_type T;
152     combsort(lo, hi, std::less<T>());
153   }
```

```
154
155  /*
156
157  Radix Sort (32-bit Signed Integers Only!)
158
159  Worst: O(d*N), where d is the number of bits/digits in each
160  of the N values. Since d is constant (32), one can say that
161  radix sort runs in linear time.
162
163  Auxiliary Space: O(d)
164
165  Stable?: The LSD (least significant digit) radix sort is
166  stable, but because std::stable_partition() is slower than
167  the unstable std::partition(), the MSD (most significant
168  digit) radix sort is much faster. Both are given here, with
169  the MSD version implemented recursively.
170
171  Notes: Radix sort usually employs a bucket sort or counting
172  sort. Here, we take advantage of the partition functions
173  found in the C++ library.
174
175  */
176
177  struct radix_comp { //UnaryPredicate for std::partition
178    const int p; //bit position [0..31] to examine
179    radix_comp(int offset): p(offset) {}
180    bool operator () (int v) const {
181      return (p == 31) ? (v < 0) : !(v & (1 << p));
182    }
183  };
184
185  void lsd_radix_sort(int * lo, int * hi) {
186    for (int lsb = 0; lsb < 32; lsb++)
187      std::stable_partition(lo, hi, radix_comp(lsb));
188  }
189
190  void msd_radix_sort(int * lo, int * hi, int msb = 31) {
191    if (lo == hi || msb < 0) return;
192    int * mid = std::partition(lo, hi, radix_comp(msb--));
193    msd_radix_sort(lo, mid, msb);
194    msd_radix_sort(mid, hi, msb);
195  }
196
197  /*** Example Usage ***/
198
199  #include <iostream>
200  #include <vector>
201  using namespace std;
202
203  bool compare_as_ints(double i, double j) {
204    return (int)i < (int)j;
205  }
206
207  template<class T> void print_range(T lo, T hi) {
208    while (lo != hi) cout << *(lo++) << "␣";
209    cout << "\n";
210  }
211
212  int main () {
```

```
213
214    { //can be used to sort arrays like std::sort()
215      int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
216      quicksort(a, a + 8);
217      print_range(a, a + 8);
218    }
219
220    { //STL contains works too
221      int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
222      vector<int> v(a, a + 8);
223      quicksort(v.begin(), v.end());
224      print_range(v.begin(), v.end());
225    }
226
227    { //reverse iterators work as expected
228      int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
229      vector<int> v(a, a + 8);
230      heapsort(v.rbegin(), v.rend());
231      print_range(v.begin(), v.end());
232    }
233
234    { //doubles are fine also
235      double a[] = {1.1, -5.0, 6.23, 4.123, 155.2};
236      vector<double> v(a, a + 5);
237      combsort(v.begin(), v.end());
238      print_range(v.begin(), v.end());
239    }
240
241    { //radix sort only works on 32-bit signed integers
242      int a[] = {1, 886, 6, -50, 7, -11, 123, 4};
243      msd_radix_sort(a, a + 8);
244      print_range(a, a + 8);
245    }
246
247    //example from cplusplus.com/reference/algorithm/stable_sort/
248    double a[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};
249
250    {
251      vector<double> v(a, a + 8);
252      cout << "mergesort() with default comparison: ";
253      mergesort(v.begin(), v.end());
254      print_range(v.begin(), v.end());
255    }
256
257    {
258      vector<double> v(a, a + 8);
259      cout << "mergesort() with 'compare_as_ints' : ";
260      mergesort(v.begin(), v.end(), compare_as_ints);
261      print_range(v.begin(), v.end());
262    }
263
264    return 0;
265  }
```

1.1.2   Array Rotation

```
1    /*
```

```
 2
 3   1.1.2 - Array Rotation
 4
 5   All of the rotate functions below are equivalent in
 6   behavior to std::rotate(). Even though the function is
 7   already implemented in the standard library, this is just
 8   a demo of how it could be done in other languages.
 9
10   Time Complexity: O(n) on the distance between lo and hi.
11   Space Complexity: O(1) auxiliary.
12
13   */
14
15   #include <algorithm> /* reverse(), rotate(), swap() */
16
17   template<class It> void rotate1(It lo, It mid, It hi) {
18     It next = mid;
19     while (lo != next) {
20       std::swap(*lo++, *next++);
21       if (next == hi)
22         next = mid;
23       else if (lo == mid)
24         mid = next;
25     }
26   }
27
28   template<class It> void rotate2(It lo, It mid, It hi) {
29     std::reverse(lo, mid);
30     std::reverse(mid, hi);
31     std::reverse(lo, hi);
32   }
33
34   int gcd(int a, int b) {
35     return b == 0 ? a : gcd(b, a % b);
36   }
37
38   template<class It> void rotate3(It lo, It mid, It hi) {
39     int n = hi - lo, jump = mid - lo;
40     int g = gcd(jump, n), cycle = n / g;
41     for (int i = 0; i < g; i++) {
42       int cur = i, next;
43       for (int j = 0; j < cycle - 1; j++) {
44         next = cur + jump;
45         if (next >= n) next -= n;
46         std::swap(*(lo + cur), *(lo + next));
47         cur = next;
48       }
49     }
50   }
51
52   /*** Example Usage ***/
53
54   #include <cassert>
55   #include <iostream>
56   #include <vector>
57   using namespace std;
58
59   int main () {
60     int a[] = {-10, 11, 5, 7, 9, 4, 8, 2};
```

```
61    rotate3(a, a + 5, a + 8); //4, 8, 2, -10, 11, 5, 7, 9
62    for (int i = 0; i < 8; i++)
63      cout << a[i] << "␣";
64    cout << "\n";
65
66    std::vector<int> v, v1, v2, v3;
67    for (int i = 0; i < 10000; i++) {
68      v.push_back(i);
69      v1.push_back(i);
70      v2.push_back(i);
71      v3.push_back(i);
72    }
73    int mid = 5678;
74    std::rotate(v.begin(), v.begin() + mid, v.end());
75    rotate1(v1.begin(), v1.begin() + mid, v1.end());
76    rotate2(v2.begin(), v2.begin() + mid, v2.end());
77    rotate3(v3.begin(), v3.begin() + mid, v3.end());
78
79    assert(v == v1 && v == v2 && v == v3);
80
81    return 0;
82  }
```

1.1.3   Counting Inversions

```
1   /*
2
3   1.1.3 - Counting Inversions
4
5   Counts how far (or close) an array is from being sorted. If
6   the array is already sorted then inversion count is 0. If
7   the array is sorted in reverse, then the inversion count is
8   maximal. Formally, two elements a[i] and a[j] form an
9   inversion if a[i] > a[j] and i < j. Note that the inversion
10  count is *not* the same as the minimum number of swaps
11  required to sort the array. This problem can be solved using
12  merge sort, or a bit of magic...
13
14  Time Complexity: O(n log n) on the size of the array.
15  Space Complexty: O(n) auxiliary.
16
17  */
18
19  #include <iterator> /* iterator_traits */
20
21  //sorts [lo, hi) and returns the inversions count
22  template<class RAI>
23  long long mergesort_inversions(RAI lo, RAI hi) {
24    typedef typename std::iterator_traits<RAI>::value_type T;
25    if (lo >= hi - 1) return 0;
26    RAI mid = lo + (hi - lo - 1) / 2, a = lo, c = mid + 1;
27    long long res = 0;
28    res += mergesort_inversions(lo, mid + 1);
29    res += mergesort_inversions(mid + 1, hi);
30    T buf[hi - lo], *b = buf;
31    while (a <= mid && c < hi) {
32      if (*c < *a) {
```

```
33         *(b++) = *(c++);
34         res += (mid - a) + 1;
35       } else {
36         *(b++) = *(a++);
37       }
38     }
39     if (a > mid)
40       for (RAI k = c; k < hi; k++) *(b++) = *k;
41     else
42       for (RAI k = a; k <= mid; k++) *(b++) = *k;
43     for (int i = hi - lo - 1; i >= 0; i--)
44       *(lo + i) = buf[i];
45     return res;
46  }
47
48  /*
49
50  The following magic is courtesy of misof, and works for any
51  array of nonnegative integers. See the link below:
52  http://codeforces.com/blog/entry/17881?#comment-232099
53
54  The complexity depends on the magnitude of the maximum value
55  in a[]. You should coordinate compress the values of a[]
56  into integers up to n for the best results. Note that after
57  calling the function, a[] will be zeroed out.
58
59  Time Complexity: O(m log m), where m is the magnitude of the
60  maximum value in the array.
61
62  Space Complexity: O(m) auxiliary on the magnitude of the
63  maximum value in the array.
64
65  */
66
67  long long magic_inversions(int n, int a[]) {
68    int mx = 0;
69    for (int i = 0; i < n; i++)
70      if (a[i] > mx) mx = a[i];
71    int c[mx];
72    long long res = 0;
73    while (mx > 0) {
74      for (int i = 0; i < mx; i++) c[i] = 0;
75      for (int i = 0; i < n; i++) {
76        if (a[i] % 2 == 0) res += c[a[i] / 2];
77        else c[a[i] / 2]++;
78      }
79      mx = 0;
80      for (int i = 0; i < n; i++)
81        if ((a[i] /= 2) > mx) mx = a[i];
82    }
83    return res;
84  }
85
86  /*** Example Usage ***/
87
88  #include <cassert>
89
90  int main() {
91    {
```

```
92      int a[] = {6, 9, 1, 14, 8, 12, 3, 2};
93      assert(mergesort_inversions(a, a + 8) == 16);
94    }
95    {
96      int a[] = {6, 9, 1, 14, 8, 12, 3, 2};
97      assert(magic_inversions(8, a) == 16);
98    }
99    return 0;
100 }
```

1.1.4   Coordinate Compression

```
1   /*
2
3   1.1.4 - Coordinate Compression
4
5   Given an array of length n, reassigns values to the array elements
6   such that the magnitude of each new value is no more than n, while
7   preserving the relative order of each element as they existed in
8   the original array. Formally, if a[] is the original array and b[]
9   is the compressed array, for every pair i, j (0 <= i, j < n), the
10  result of comparing a[i] compared to a[j] (<, >=, ==, etc.) will
11  be exactly the same as the result of b[i] compared to b[j].
12
13  For the following implementations, values in the range [lo, hi)
14  will be changed to integers in the range [0, D), where D is the
15  number of distinct values in the range. lo and hi must be iterators
16  pointing to a numerical type that int can be assigned to.
17
18  Time Complexity: O(n log n) on the size of the array.
19  Space Complexity: O(n) auxiliary on the size of the array.
20
21  */
22
23  #include <algorithm> /* lower_bound(), sort(), unique() */
24  #include <iterator>  /* std::iterator_traits */
25  #include <map>
26
27  //version 1 - using std::unique() and std::lower_bound()
28  template<class It> void compress1(It lo, It hi) {
29    typename std::iterator_traits<It>::value_type a[hi - lo];
30    int n = 0;
31    for (It it = lo; it != hi; ++it) 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  }
37
38  //version 2 - using std::map
39  template<class It> void compress2(It lo, It hi) {
40    typedef typename std::iterator_traits<It>::value_type T;
41    std::map<T, int> m;
42    for (It it = lo; it != hi; ++it) m[*it] = 0;
43    typename std::map<T, int>::iterator x = m.begin();
44    for (int i = 0; x != m.end(); x++) x->second = i++;
45    for (It it = lo; it != hi; ++it) *it = m[*it];
```

```
46  }
47
48  /*** Example Usage ***/
49
50  #include <iostream>
51  using namespace std;
52
53  template<class It> void print(It lo, It hi) {
54    while (lo != hi) cout << *lo++ << "␣";
55    cout << "\n";
56  }
57
58  int main() {
59    {
60      int a[] = {1, 30, 30, 7, 9, 8, 99, 99};
61      compress1(a, a + 8);
62      print(a, a + 8); //0 4 4 1 3 2 5 5
63    }
64
65    {
66      int a[] = {1, 30, 30, 7, 9, 8, 99, 99};
67      compress2(a, a + 8);
68      print(a, a + 8); //0 4 4 1 3 2 5 5
69    }
70
71    { //works on doubles too
72      double a[] = {0.5, -1.0, 3, -1.0, 20, 0.5};
73      compress1(a, a + 6);
74      print(a, a + 6); //1 0 2 0 3 1
75    }
76    return 0;
77  }
```

1.1.5   Kth Order Statistics

```
 1  /*
 2
 3  1.1.5 - Kth Order Statistics
 4
 5  The following is equivalent in behavior to the std::nth_element()
 6  function in the C++ standard library. They rearrange the elements
 7  in the range [lo, hi), in such a way that the element at the nth
 8  position is the element that would be in that position in a sorted
 9  sequence. The other elements are left without any specific order,
10  except that none of the elements preceding nth are greater than
11  it, and none of the elements following it are less than it.
12
13  Time Complexity: O(n) expected time on the distance between
14  lo and hi. However, its worst case is technically O(n^2).
15  This has the same running time as std::nth_element().
16
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(int n) {
26     return ((rand() & 0x7fff) | ((rand() & 0x7fff) << 15)) % n;
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) std::swap(*(++i), *j);
35     return i;
36   }
37
38   template<class It> void nth_element2(It lo, It n, It hi) {
39     for (;;) {
40       It k = rand_partition(lo, hi);
41       if (n < k) hi = k;
42       else if (n > k) lo = k + 1;
43       else return;
44     }
45   }
46
47   /*** Example Usage ***/
48
49   #include <iostream>
50   using namespace std;
51
52   template<class It> void print(It lo, It hi) {
53     while (lo != hi) cout << *lo++ << "␣";
54     cout << "\n";
55   }
56
57   int main () {
58     int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
59     random_shuffle(a, a + 9);
60     nth_element2(a, a + 5, a + 9);
61     print(a, a + 9);
62     return 0;
63   }
```

1.2   Array Queries

1.2.1   Longest Increasing Subsequence

```
1    /*
2
3    1.2.1 - Longest Increasing Subsequence
4
5    Find a subarray of a given array n such that the subarray's
6    elements are in sorted order, lowest to highest, and in which
7    the subsequence is as long as possible. This subsequence is not
8    necessarily contiguous, or unique, so the following function
9    will find one such sequence.
10
```

```
11  Time Complexity: O(n log n) on the size of the array.
12  Space Complexity: O(n) auxiliary.
13
14  */
15
16  #include <vector>
17
18  int lower_bound(int a[], int tail[], int len, int key) {
19    int lo = -1, hi = len;
20    while (hi - lo > 1) {
21      int mid = (lo + hi) / 2;
22      if (a[tail[mid]] < key) lo = mid;
23      else hi = mid;
24    }
25    return hi;
26  }
27
28  template<class T> std::vector<T> lis(int n, T a[]) {
29    int tail[n], prev[n];
30    int len = 0;
31    for (int i = 0; i < n; i++) {
32      int pos = lower_bound(a, tail, len, a[i]);
33      if (len < pos + 1) len = pos + 1;
34      prev[i] = pos > 0 ? tail[pos - 1] : -1;
35      tail[pos] = i;
36    }
37    std::vector<T> res(len);
38    for (int i = tail[len - 1]; i != -1; i = prev[i])
39      res[--len] = a[i];
40    return res;
41  }
42
43  /*** Example Usage ***/
44
45  #include <iostream>
46  using namespace std;
47
48  template<class It> void print(It lo, It hi) {
49    while (lo != hi) cout << *lo++ << "␣";
50    cout << "\n";
51  }
52
53  int main () {
54    int a[] = {-2, -5, 1, 9, 10, 8, 11, 10, 13, 11};
55    vector<int> res = lis(10, a);
56    print(res.begin(), res.end()); //-5 1 9 10 11 13
57    return 0;
58  }
```

1.2.2   Maximal Subarray Sum (Kadane's)

```
1  /*
2
3  1.2.2 - Maximal Subarray Sum (Kadane's Algorithm)
4
5  Given an array of integers that may be negative, determine
6  the maximum sum of any contiguous subarray. Additionally,
```

```
 7   do so optimally without using extra memory.
 8
 9   Time Complexity: O(n) on the size of the array
10   Space Complexty: O(1) auxiliary.
11
12   */
13
14   template<class It> int max_subarray_sum(It lo, It hi) {
15     int max_ending_here = 0, max_so_far = 0, max_overall;
16     for (max_overall = *lo; lo != hi; ++lo) {
17       max_ending_here += *lo;
18       if (max_overall < *lo) max_overall = *lo;
19       if (max_ending_here < 0) {
20         max_ending_here = 0;
21       } else if (max_so_far < max_ending_here) {
22         max_so_far = max_ending_here;
23       }
24     }
25     if (max_overall < 0) return max_overall; //all negative
26     return max_so_far;
27   }
28
29   /*** Example Usage ***/
30
31   #include <iostream>
32   using namespace std;
33
34   int main() {
35     int a[] = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
36     //answer is 4 + (-1) + 2 + 1 = 6;
37     cout << max_subarray_sum(a, a + 9) << "\n";
38     return 0;
39   }
```

1.2.3   Majority Element (Boyer-Moore)

```
 1   /*
 2
 3   1.2.3 - Majority Element (Boyer-Moore Algorithm)
 4
 5   Given an array of size n, the majority vote problem asks
 6   to determine whether there exists a value with greater
 7   occurrences than all the others, and if so, to determine
 8   this value. In other words, a value must occur more than
 9   floor(n/2) times to be the majority. Additionally, do so
10   optimally without using extra memory.
11
12   Time Complexity: O(n) on the size of the array.
13   Space Complexty: O(1) auxiliary space.
14
15   */
16
17   #include <iterator> /* std::iterator_traits */
18
19   //returns an iterator to the majority element in
20   //the range [lo, hi), or hi if none exists.
21   template<class It> It majority(It lo, It hi) {
```

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

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

```
1   /*
2
3   1.2.4 - Subset Sum (Meet-in-the-Middle)
4
5   Given an unsorted array of integers and a value v, determine
6   the minimum sum of any subset of values in the array that is
7   not less than v. A meet-in-the-middle algorithm divides the
8   array in two equal parts. All possible sums are precomputed
9   and stored in a table and sorted by the sum. Finally, to
10  find the answer, the table is searched.
11
12  Time Complexity: O(n * 2^(n / 2)) on the size of the array.
13  Space Complexity: O(n) auxiliary on the size of the array.
14
15  */
16
17  #include <algorithm> /* std::max(), std::sort() */
18  #include <climits>   /* LLONG_MIN */
19
20  template<class It>
21  long long sum_lower_bound(It lo, It hi, long long v) {
22    int n = hi - lo;
23    int llen = 1 << (n / 2);
24    int rlen = 1 << (n - n / 2);
25    long long lsum[llen], rsum[rlen];
```

```
26    std::fill(lsum, lsum + llen, 0);
27    std::fill(rsum, rsum + rlen, 0);
28    for (int i = 0; i < llen; ++i)
29      for (int j = 0; j < n / 2; ++j)
30        if ((i & (1 << j)) > 0)
31          lsum[i] += *(lo + j);
32    for (int i = 0; i < rlen; ++i)
33      for (int j = 0; j < n - n / 2; ++j)
34        if ((i & (1 << j)) > 0)
35          rsum[i] += *(lo + j + n / 2);
36    std::sort(lsum, lsum + llen);
37    std::sort(rsum, rsum + llen);
38    int l = 0, r = rlen - 1;
39    long long cur = LLONG_MIN;
40    while (l < llen && r >= 0) {
41      if (lsum[l] + rsum[r] <= v) {
42        cur = std::max(cur, lsum[l] + rsum[r]);
43        l++;
44      } else {
45        r--;
46      }
47    }
48    return cur;
49  }
50
51  /*** Example Usage ***/
52
53  #include <iostream>
54  using namespace std;
55
56  int main() {
57    long long a[] = {9, 1, 5, 0, 1, 11, 5};
58    cout << sum_lower_bound(a, a + 7, 8) << "\n"; //7
59    return 0;
60  }
```

1.2.5   Maximal Zero Submatrix

```
1   /*
2
3   1.2.5 - Maximal Zero Submatrix
4
5   Given a rectangular matrix of 0's and 1's, determine the
6   area of the largest submatrix which contains only 0's.
7
8   Time Complexity: O(n * m), where n is the number of rows
9   and m is the number of columns in the matrix.
10
11  Space Complexity: O(m) auxiliary on the number of columns.
12
13  */
14
15  #include <algorithm> /* std::max() */
16  #include <vector>
17
18  typedef std::vector<std::vector<bool> > matrix;
19
```

```
20  int max_zero_submatrix(const matrix & a) {
21    int n = a.size(), m = a[0].size(), res = 0;
22    std::vector<int> d(m, -1), d1(m), d2(m), st(m);
23    for (int r = 0; r < n; r++) {
24      for (int c = 0; c < m; c++)
25        if (a[r][c]) d[c] = r;
26      int sz = 0;
27      for (int c = 0; c < m; c++) {
28        while (sz > 0 && d[st[sz - 1]] <= d[c]) sz--;
29        d1[c] = sz == 0 ? -1 : st[sz - 1];
30        st[sz++] = c;
31      }
32      sz = 0;
33      for (int c = m - 1; c >= 0; c--) {
34        while (sz > 0 && d[st[sz - 1]] <= d[c]) sz--;
35        d2[c] = sz == 0 ? m : st[sz - 1];
36        st[sz++] = c;
37      }
38      for (int j = 0; j < m; j++)
39        res = std::max(res, (r - d[j]) * (d2[j] - d1[j] - 1));
40    }
41    return res;
42  }
43
44  /*** Example Usage ***/
45
46  #include <iostream>
47  using namespace std;
48
49  int main() {
50    const int n = 5, m = 6;
51    bool a[n][m] = {{1, 0, 1, 1, 0, 0},
52                    {1, 0, 0, 1, 0, 0},
53                    {0, 0, 0, 0, 0, 1},
54                    {1, 0, 0, 1, 0, 0},
55                    {1, 0, 1, 0, 0, 1}};
56    matrix v(n);
57    for (int i = 0; i < 5; i++)
58      v[i] = vector<bool>(a[i], a[i] + m);
59    cout << max_zero_submatrix(v) << "\n";
60    return 0;
61  }
```

1.3   Searching

1.3.1   Discrete Binary Search

```
1  /*
2
3  1.3.1 - Discrete Binary Search
4
5  Not only can binary search be used on arrays, it can also be used
6  on monotonic functions. In this case we aren't restricted by
7  tangible quantities such as available memory, as is the case with
8  storing values explicitly in a sorted array.
9
```

```
10  binary_search_first_true() returns the smallest integer k in the
11  range [lo, hi) - (i.e. including lo, but excluding hi) for which
12  the boolean function query(k) is true. It can be used if and only
13  if there exists a constant k where the return value of query(x)
14  is false for all x < k and true for all x >= k.
15
16  binary_search_last_true() returns the greatest integer k in the
17  range [lo, hi) - (i.e. including lo, but excluding hi) for which
18  the boolean function query(k) is true. It can be used if and only
19  if there exists a constant k where the return value of query(x)
20  is true for all x <= k and false for all x > k.
21
22  Complexity: At most O(log N) calls to query(), where N is the
23  search space (the distance between lo and hi).
24
25  fbinary_search() works like binary_search_first_true(), but on
26  floating point query functions. Since the set of reals numbers is
27  dense, it is clear that we cannot find the exact target. Instead,
28  fbinary_search() will return a value that is very close to the
29  border between false and true. Note that the number of iterations
30  performed depends on how precise we want the answer to be. Although
31  one can loop while (hi - lo > epsilon) until the distance between
32  lo and hi is really small, we can instead just keep it simple and
33  let the loop run for a few hundred iterations. In most contest
34  applications, 100 iterations is more than sufficient, reducing the
35  search space to 10^-30 times its original size.
36
37  */
38
39  //000[1]11
40  template<class Int, class BoolFunction>
41  Int binary_search_first_true(Int lo, Int hi, BoolFunction query) {
42    Int mid, _hi = hi;
43    while (lo < hi) {
44      mid = lo + (hi - lo) / 2;
45      if (query(mid)) hi = mid;
46      else lo = mid + 1;
47    }
48    if (!query(lo)) return _hi; //all [lo, hi) is false
49    return lo;
50  }
51
52  //11[1]000
53  template<class Int, class BoolFunction>
54  Int binary_search_last_true(Int lo, Int hi, BoolFunction query) {
55    Int mid, _hi = hi;
56    while (lo < hi) {
57      mid = lo + (hi - lo + 1) / 2;
58      if (query(mid)) lo = mid;
59      else hi = mid - 1;
60    }
61    if (!query(lo)) return _hi; //all [lo, hi) is true
62    return lo;
63  }
64
65  //000[1]11
66  template<class Double, class BoolFunction>
67  Double fbinary_search(Double lo, Double hi, BoolFunction query) {
68    Double mid;
```

```
69     for (int reps = 0; reps < 100; reps++) {
70       mid = (lo + hi) / 2.0;
71       //simply switch hi and lo in the line below to find last true
72       (query(mid) ? hi : lo) = mid;
73     }
74     return lo;
75   }
76
77   /*** Example Usage ***/
78
79   #include <iostream>
80   using namespace std;
81
82   //Define your own black-box query function here. Simple examples:
83   bool query1(int x) { return x >= 3; }
84   bool query2(int x) { return false; }
85   bool query3(int x) { return x <= 5; }
86   bool query4(int x) { return true; }
87   bool query5(double x) { return x >= 1.2345; }
88
89   int main() {
90     cout << binary_search_first_true(0, 7, query1) << "\n"; //3
91     cout << binary_search_first_true(0, 7, query2) << "\n"; //7
92     cout << binary_search_last_true(0, 7, query3)  << "\n"; //5
93     cout << binary_search_last_true(0, 7, query4)  << "\n"; //7
94     cout << fbinary_search(-10.0, 10.0, query5)    << "\n"; //1.2345
95     return 0;
96   }
```

1.3.2   Ternary Search

```
1   /*
2
3   1.3.2 - Ternary Search
4
5   Given a unimodal function f(x), find its maximum or minimum.
6
7   Assume we are looking for a maximum of f(x) and that we know
8   the maximum lies somewhere between A and B. For the algorithm
9   to be applicable, there must be some value x such that:
10  >>> for all a,b with A <= a < b <= x, we have f(a) < f(b); and
11  >>> for all a,b with x <= a < b <= B, we have f(a) > f(b).
12
13  In other words, for the graph of f(x) must look like a "valley"
14  with a single point at its trough (lowest point) and no flat
15  regions on the left side or right side for a ternary search to
16  be able to find its minimum point. Similarly the graph of f(x)
17  must look like a "hill" with a single peak for a ternary search
18  to able able to find its maximum point.
19
20  The answer for both functions will be really close to the peak
21  or trough by no further than +/-epsilon (0.00000001 by default).
22
23  */
24
25  template<class UnimodalFunction>
26  double ternary_search_min(double lo, double hi,
```

```
27                                UnimodalFunction f, double eps = 1e-8) {
28     double lthird, hthird;
29     while (hi - lo > eps) {
30       lthird = lo + (hi - lo) / 3;
31       hthird = hi - (hi - lo) / 3;
32       if (f(lthird) < f(hthird)) hi = hthird;
33       else lo = lthird;
34     }
35     return lo;
36   }
37
38   template<class UnimodalFunction>
39   double ternary_search_max(double lo, double hi,
40                                UnimodalFunction f, double eps = 1e-8) {
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)) lo = lthird;
46       else hi = hthird;
47     }
48     return lo;
49   }
50
51   /*** Example Usage ***/
52
53   #include <iostream>
54   using namespace std;
55
56   //parabola opening up with vertex at (-2, -24)
57   double f1(double x) { return 3 * x * x + 12 * x - 12; }
58
59   //parabola opening down with vertex at (0.105, 88.1)
60   double f2(double x) { return -5.7 * x * x + 1.2 * x + 88; }
61
62   int main() {
63     cout << ternary_search_min(-99, 99, f1) << endl; //-2
64     cout << ternary_search_max(-99, 99, f2) << endl; //0.105263
65     return 0;
66   }
```

1.3.3   Hill Climbing

```
1   /*
2
3   1.3.3 - Hill Climbing
4
5   Hill climbing is an optimization technique which finds the
6   local extrema of a function. It is an iterative algorithm
7   that starts with an arbitrary solution, then attempts to
8   find a better solution by incrementally changing a single
9   element of the solution. If the change produces a better
10  solution, an incremental change is made to the new solution,
11  repeating until no further improvements can be found.
12
13  Complexity: O(log SIZE) where SIZE is the search space.
14
```

```
15  */
16
17  #include <cmath>
18  #include <iostream>
19  using namespace std;
20
21  template<class BinaryFunction>
22  double find_minimum(BinaryFunction f) {
23    static const double PI = acos(-1.0);
24    double x = 0, y = 0, res = f(x, y);
25    for (double step = 1e6; step > 1e-7; ) {
26      double best = res, bestx = x, besty = y;
27      bool found = false;
28      for (int i = 0; i < 6; i++) {
29        double a = 2.0 * PI * i / 6.0;
30        double nextx = x + step * cos(a);
31        double nexty = y + step * sin(a);
32        double next = f(nextx, nexty);
33        if (best > next) {
34          bestx = nextx;
35          besty = nexty;
36          best = next;
37          found = true;
38        }
39      }
40      if (!found) {
41        step /= 2.0;
42      } else {
43        x = bestx;
44        y = besty;
45        res = best;
46      }
47    }
48    cout << fixed << x << "␣" << y << "\n";
49    return res;
50  }
51
52  /*** Example Usage ***/
53
54  //minimized at f(2, 3) = 0
55  double f(double x, double y) {
56    return (x - 2)*(x - 2) + (y - 3)*(y - 3);
57  }
58
59  int main() {
60    cout.precision(1);
61    cout << fixed << find_minimum(f) << "\n";
62    return 0;
63  }
```

1.3.4   Convex Hull Optimization

```
1  /*
2
3  1.3.4 - Convex Hull Optimization
4
5  Given a set of lines in the form y = mx + b process queries
```

```
6    of x-coordinates to find the minimum y-value for any of the
7    lines when evaluated at x. For the following implementation,
8    the lines must be added in order of descending slope (m),
9    and queries must be called in nondecreasing order of x.
10   See: http://wcipeg.com/wiki/Convex_hull_optimization
11
12   Time Complexity: O(n log n) overall, where n is the number
13   of lines, for any number of calls to the two functions that
14   satisfy the preconditions.
15
16   Space Complexity: O(n) auxiliary on the number of lines.
17
18   */
19
20   #include <vector>
21
22   std::vector<long long> M, B;
23   int ptr = 0;
24
25   //precondition: m must be the minimum of all lines added so far
26   void add_line(long long m, long long b) {
27     int len = M.size();
28     while (len >= 2 && (B[len - 2] - B[len - 1]) * (m - M[len - 1]) >=
29                        (B[len - 1] - b) * (M[len - 1] - M[len - 2])) {
30       len--;
31     }
32     M.resize(len);
33     B.resize(len);
34     M.push_back(m);
35     B.push_back(b);
36   }
37
38   //precondition: x must be the maximum of all x queried so far
39   long long min_y(long long x) {
40     if (ptr >= (int)M.size()) ptr = (int)M.size() - 1;
41     while (ptr + 1 < (int)M.size() && M[ptr + 1] * x + B[ptr + 1] <=
42                                       M[ptr] * x + B[ptr]) {
43       ptr++;
44     }
45     return M[ptr] * x + B[ptr];
46   }
47
48   /*** Example Usage ***/
49
50   #include <cassert>
51
52   int main() {
53     add_line(3, 0);
54     add_line(2, 1);
55     add_line(3, 2);
56     add_line(0, 6);
57     assert(min_y(0) == 0);
58     assert(min_y(1) == 3);
59     assert(min_y(2) == 5);
60     assert(min_y(3) == 6);
61     return 0;
62   }
```

1.4   Cycle Detection

1.4.1   Floyd's Algorithm

```
/*

1.4.1 - Cycle Detection (Floyd's Algorithm)

For a function f which maps a finite set S to itself and any
initial value x0 in S, the sequence of iterated values:

x_0, x_1 = f(x_0), x_2 = f(x_1), ... x_i = f(x_(i-1))

must eventually use the same value twice: there must be some
i <> j such that xi = xj. Once this happens, the sequence
must continue periodically, by repeating the same sequence
of values from x_i to x_(j1). Cycle detection asks to find
i and j, given the function f(x) and x_0.

Floyd's cycle-finding algorithm, a.k.a. the "tortoise and
the hare algorithm", is a pointer algorithm that uses only
2 pointers, moving through the sequence at different speeds.

Time Complexity: O(lambda + mu), where lambda is the length
of the cycle and mu is the first index of x for which the
cycle starts to occur.

Space Complexity: O(1) auxiliary.

*/

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

//returns pair<mu, lambda> (as described above)
template<class IntFunction>
std::pair<int, int> floyd(IntFunction f, int x0) {
  int tortoise = f(x0), hare = f(f(x0));
  while (tortoise != hare) {
    tortoise = f(tortoise);
    hare = f(f(hare));
  }
  int start = 0;
  tortoise = x0;
  while (tortoise != hare) {
    tortoise = f(tortoise);
    hare = f(hare);
    start++;
  }
  int length = 1;
  hare = f(tortoise);
  while (tortoise != hare) {
    hare = f(hare);
    length++;
  }
  return std::make_pair(start, length);
}

```

```
54  /*** Example Usage ***/
55
56  #include <cassert>
57  #include <iostream>
58  #include <set>
59  using namespace std;
60
61  int f(int x) {
62    return (123 * x * x + 4567890) % 1337;
63  }
64
65  void verify(int x0, int start, int length) {
66    set<int> s;
67    int x = x0;
68    for (int i = 0; i < start; i++) {
69      assert(!s.count(x));
70      s.insert(x);
71      x = f(x);
72    }
73    int startx = x;
74    s.clear();
75    for (int i = 0; i < length; i++) {
76      assert(!s.count(x));
77      s.insert(x);
78      x = f(x);
79    }
80    assert(startx == x);
81  }
82
83  int main () {
84    int x0 = 0;
85    pair<int, int> res = floyd(f, x0);
86    cout << "Found cycle of length " << res.second;
87    cout << " starting at x_" << res.first << ".\n";
88    verify(x0, res.first, res.second);
89    return 0;
90  }
```

1.4.2   Brent's Algorithm

```
1   /*
2
3   1.4.2 - Cycle Detection (Brent's Algorithm)
4
5   For a function f which maps a finite set S to itself and any
6   initial value x0 in S, the sequence of iterated values:
7
8   x_0, x_1 = f(x_0), x_2 = f(x_1), ... x_i = f(x_(i-1))
9
10  must eventually use the same value twice: there must be some
11  i <> j such that xi = xj. Once this happens, the sequence
12  must continue periodically, by repeating the same sequence
13  of values from x_i to x_(j1). Cycle detection asks to find
14  i and j, given the function f(x) and x_0.
15
16  Brent's cycle-finding algorithm is based on a different idea
17  than Floyd's: searching for the smallest power of two, 2^i
```

```
18   that is larger than both lambda and mu.
19
20   Time Complexity: O(lambda + mu), where lambda is the length
21   of the cycle and mu is the first index of x for which the
22   cycle starts to occur. Brent claims that, on average, his
23   cycle finding algorithm runs around 36% more quickly than
24   Floyd's and that it speeds up the Pollard rho algorithm by
25   around 24% (see mathematics chapter).
26
27   Space Complexity: O(1) auxiliary.
28
29   */
30
31   #include <utility> /* std::pair */
32
33   //returns pair<mu, lambda> (as described above)
34   template<class IntFunction>
35   std::pair<int, int> brent(IntFunction f, int x0) {
36     int power = 1, length = 1;
37     int tortoise = x0, hare = f(x0);
38     while (tortoise != hare) {
39       if (power == length) {
40         tortoise = hare;
41         power *= 2;
42         length = 0;
43       }
44       hare = f(hare);
45       length++;
46     }
47     hare = x0;
48     for (int i = 0; i < length; i++)
49       hare = f(hare);
50     int start = 0;
51     tortoise = x0;
52     while (tortoise != hare) {
53       tortoise = f(tortoise);
54       hare = f(hare);
55       start++;
56     }
57     return std::make_pair(start, length);
58   }
59
60   /*** Example Usage ***/
61
62   #include <cassert>
63   #include <iostream>
64   #include <set>
65   using namespace std;
66
67   int f(int x) {
68     return (123 * x * x + 4567890) % 1337;
69   }
70
71   void verify(int x0, int start, int length) {
72     set<int> s;
73     int x = x0;
74     for (int i = 0; i < start; i++) {
75       assert(!s.count(x));
76       s.insert(x);
```

```
77        x = f(x);
78      }
79      int startx = x;
80      s.clear();
81      for (int i = 0; i < length; i++) {
82        assert(!s.count(x));
83        s.insert(x);
84        x = f(x);
85      }
86      assert(startx == x);
87    }
88
89    int main () {
90      int x0 = 0;
91      pair<int, int> res = brent(f, x0);
92      cout << "Found cycle of length " << res.second;
93      cout << " starting at x_" << res.first << ".\n";
94      verify(x0, res.first, res.second);
95      return 0;
96    }
```

1.5   Binary Exponentiation

```
1     /*
2
3     1.5.1 - Binary Exponentiation
4
5     Exponentiation by squaring is a general method for fast
6     computation of large positive integer powers of a number.
7     This method is also known as "square-and-multiply." The
8     following version computes x^n modulo m. Remove the "% m"
9     if you do not want the answer to be modded - but as this
10    causes overflow, you should switch to a bigger data type.
11
12    Complexity: O(log n) on the exponent of the computation.
13
14    */
15
16    #include <stdint.h>
17
18    uint64_t mulmod(uint64_t a, uint64_t b, uint64_t m) {
19      uint64_t x = 0, y = a % m;
20      for (; b > 0; b >>= 1) {
21        if (b & 1) x = (x + y) % m;
22        y = (y << 1) % m;
23      }
24      return x % m;
25    }
26
27    uint64_t powmod(uint64_t a, uint64_t b, uint64_t m) {
28      uint64_t x = 1, y = a;
29      for (; b > 0; b >>= 1) {
30        if (b & 1) x = mulmod(x, y, m);
31        y = mulmod(y, y, m);
32      }
33      return x % m;
```

```
34  }
35
36  /*** Example Usage ***/
37
38  #include <cassert>
39
40  int main() {
41    assert(powmod(2, 10, 1000000007) == 1024);
42    return 0;
43  }
```

Chapter 2

Graph Theory

2.1 Depth-First Search

2.1.1 Depth-First Search

```
1   /*
2
3   2.1.1 - Depth First Search
4
5   Description: Given an unweighted graph, traverse all reachable
6   nodes from a source node. Each branch is explored as deep as
7   possible before more branches are visited. DFS only uses as
8   much space as the length of the longest branch. When DFS'ing
9   recursively, the internal call-stack could overflow, so
10  sometimes it is safer to use an explicit stack data structure.
11
12  Complexity: O(V+E) on the number of vertices and edges for
13  explicit graphs traversed without repetition. O(b^d) for
14  implicit graphs with a branch factor of b, searched to depth d.
15
16  =~=~=~=~= Sample Input =~=~=~=~=
17  12 11 0
18  0 1
19  0 6
20  0 7
21  1 2
22  1 5
23  2 3
24  2 4
25  7 8
26  7 11
27  8 9
28  8 10
29
30  =~=~=~=~= Sample Output =~=~=~=~=
31  Nodes visited: 0 1 2 3 4 5 6 7 8 9 10 11
32
33  */
34
35  #include <iostream>
```

27

```
36   #include <vector>
37   using namespace std;
38
39   const int MAXN = 100;
40   vector<bool> vis(MAXN);
41   vector<int> adj[MAXN];
42
43   void dfs(int u) {
44     vis[u] = true;
45     cout << "␣" << u;
46     for (int j = 0; j < (int)adj[u].size(); j++)
47       if (!vis[adj[u][j]])
48         dfs(adj[u][j]);
49   }
50
51   int main() {
52     int nodes, edges, start, u, v;
53     cin >> nodes >> edges >> start;
54     for (int i = 0; i < edges; i++) {
55       cin >> u >> v;
56       adj[u].push_back(v);
57     }
58     cout << "Nodes␣visited:";
59     dfs(start);
60     cout << "\n";
61     return 0;
62   }
```

2.1.2   Floodfill

```
1    /*
2
3    2.1.2 - Floodfill (DFS)
4
5    Description: Given a directed graph and a source node,
6    traverse to all reachable nodes from the source and determine
7    the total area traveled. Conceptually, the order that nodes
8    are visited in a "flooding" of the graph should resemble a
9    BFS ordering. However, if the objective is simply to visit
10   all reachable nodes without regard for the order (as is the
11   case with most applications of floodfill in contests), it is
12   much simpler to DFS because an extra queue is not needed.
13   The input graph is stored in an adjacency list.
14
15   Complexity: O(V+E) on the number of vertices and edges.
16
17   =~=~=~=~= Sample Input =~=~=~=~=
18   8 8 0
19   0 1
20   0 5
21   1 2
22   1 3
23   3 2
24   4 0
25   4 3
26   6 7
27
```

```
28   =~=~=~=~= Sample Output =~=~=~=~=
29   Visited 5 nodes starting from 0
30
31   */
32
33   #include <iostream>
34   #include <vector>
35   using namespace std;
36
37   const int MAXN = 100;
38   vector<bool> vis(MAXN);
39   vector<int> adj[MAXN];
40
41   int dfs(int u) {
42     vis[u] = true;
43     int area = 1;
44     for (int j = 0; j < (int)adj[u].size(); j++)
45       if (!vis[adj[u][j]])
46         area += dfs(adj[u][j]);
47     return area;
48   }
49
50   int main() {
51     int nodes, edges, start, u, v;
52     cin >> nodes >> edges >> start;
53     for (int i = 0; i < edges; i++) {
54       cin >> u >> v;
55       adj[u].push_back(v);
56     }
57     cout << "Visited␣" << dfs(start) << "␣nodes";
58     cout << "␣starting␣from␣" << start << "\n";
59     return 0;
60   }
```

2.1.3   Topological Sorting

```
1    /*
2
3    2.1.3 - Topological Sorting (DFS)
4
5    Description: Given a directed acyclic graph (DAG), order the nodes
6    such that for every edge from a to b, a precedes b in the ordering.
7    Usually, there is more than one possible valid ordering. The
8    following program uses DFS to produce one possible ordering.
9    This can also be used to detect whether the graph is a DAG.
10   Note that the DFS algorithm here produces a reversed topological
11   ordering, so the output must be printed backwards. The graph is
12   stored in an adjacency list.
13
14   Complexity: O(V+E) on the number of vertices and edges.
15
16   =~=~=~=~= Sample Input =~=~=~=~=
17   8 9
18   0 3
19   0 4
20   1 3
21   2 4
```

```
22  2 7
23  3 5
24  3 6
25  3 7
26  4 6
27
28  =~=~=~=~= Sample Output =~=~=~=~=
29  The topological order: 2 1 0 4 3 7 6 5
30
31  */
32
33  #include <algorithm> /* std::fill(), std::reverse() */
34  #include <iostream>
35  #include <stdexcept> /* std::runtime_error() */
36  #include <vector>
37  using namespace std;
38
39  const int MAXN = 100;
40  vector<bool> vis(MAXN), done(MAXN);
41  vector<int> adj[MAXN], sorted;
42
43  void dfs(int u) {
44    if (vis[u])
45      throw std::runtime_error("Not a DAG.");
46    if (done[u]) return;
47    vis[u] = true;
48    for (int j = 0; j < (int)adj[u].size(); j++)
49      dfs(adj[u][j]);
50    vis[u] = false;
51    done[u] = true;
52    sorted.push_back(u);
53  }
54
55  void toposort(int nodes) {
56    fill(vis.begin(), vis.end(), false);
57    fill(done.begin(), done.end(), false);
58    sorted.clear();
59    for (int i = 0; i < nodes; i++)
60      if (!done[i]) dfs(i);
61    reverse(sorted.begin(), sorted.end());
62  }
63
64  int main() {
65    int nodes, edges, u, v;
66    cin >> nodes >> edges;
67    for (int i = 0; i < edges; i++) {
68      cin >> u >> v;
69      adj[u].push_back(v);
70    }
71    toposort(nodes);
72    cout << "The topological order:";
73    for (int i = 0; i < (int)sorted.size(); i++)
74      cout << " " << sorted[i];
75    cout << "\n";
76    return 0;
77  }
```

2.1.4 Eulerian Cycles

```
1   /*
2
3   2.1.4 - Eulerian Cycles (DFS)
4
5   Description: A Eulerian trail is a trail in a graph which
6   visits every edge exactly once. Similarly, an Eulerian circuit
7   or Eulerian cycle is an Eulerian trail which starts and ends
8   on the same vertex.
9
10  An undirected graph has an Eulerian cycle if and only if every
11  vertex has even degree, and all of its vertices with nonzero
12  degree belong to a single connected component.
13
14  A directed graph has an Eulerian cycle if and only if every
15  vertex has equal in degree and out degree, and all of its
16  vertices with nonzero degree belong to a single strongly
17  connected component.
18
19  Complexity: O(V+E) on the number of vertices and edges.
20
21  =~=~=~=~= Sample Input =~=~=~=~=
22  5 6
23  0 1
24  1 2
25  2 0
26  1 3
27  3 4
28  4 1
29
30  =~=~=~=~= Sample Output =~=~=~=~=
31  Eulerian cycle from 0 (directed): 0 1 3 4 1 2 0
32  Eulerian cycle from 2 (undirected): 2 1 3 4 1 0 2
33
34  */
35
36  #include <algorithm> /* std::reverse() */
37  #include <iostream>
38  #include <vector>
39  using namespace std;
40
41  const int MAXN = 100;
42
43  vector<int> euler_cycle_directed(vector<int> adj[], int u) {
44    vector<int> stack, res, cur_edge(MAXN);
45    stack.push_back(u);
46    while (!stack.empty()) {
47      u = stack.back();
48      stack.pop_back();
49      while (cur_edge[u] < (int)adj[u].size()) {
50        stack.push_back(u);
51        u = adj[u][cur_edge[u]++];
52      }
53      res.push_back(u);
54    }
55    reverse(res.begin(), res.end());
56    return res;
```

```
57  }
58
59  vector<int> euler_cycle_undirected(vector<int> adj[], int u) {
60    vector<vector<bool> > used(MAXN, vector<bool>(MAXN, false));
61    vector<int> stack, res, cur_edge(MAXN);
62    stack.push_back(u);
63    while (!stack.empty()) {
64      u = stack.back();
65      stack.pop_back();
66      while (cur_edge[u] < (int)adj[u].size()) {
67        int v = adj[u][cur_edge[u]++];
68        if (!used[min(u, v)][max(u, v)]) {
69          used[min(u, v)][max(u, v)] = 1;
70          stack.push_back(u);
71          u = v;
72        }
73      }
74      res.push_back(u);
75    }
76    reverse(res.begin(), res.end());
77    return res;
78  }
79
80  int main() {
81    int nodes, edges, u, v;
82    vector<int> g1[5], g2[5], cycle;
83
84    cin >> nodes >> edges;
85    for (int i = 0; i < edges; i++) {
86      cin >> u >> v;
87      g1[u].push_back(v);
88      g2[u].push_back(v);
89      g2[v].push_back(u);
90    }
91
92    cycle = euler_cycle_directed(g1, 0);
93    cout << "Eulerian␣cycle␣from␣0␣(directed):";
94    for (int i = 0; i < (int)cycle.size(); i++)
95      cout << "␣" << cycle[i];
96    cout << "\n";
97
98    cycle = euler_cycle_undirected(g2, 2);
99    cout << "Eulerian␣cycle␣from␣2␣(undirected):";
100   for (int i = 0; i < (int)cycle.size(); i++)
101     cout << "␣" << cycle[i];
102   cout << "\n";
103   return 0;
104 }
```

2.1.5   Unweighted Tree Centers

```
1  /*
2
3  2.1.5 - Unweighted Tree Centers, Centroid, and Diameter
4
5  The following applies to unweighted, undirected trees only.
6
```

```
 7  find_centers(): Returns 1 or 2 tree centers. The center
 8  (or Jordan center) of a graph is the set of all vertices of
 9  minimum eccentricity, that is, the set of all vertices A
10  where the max distance d(A,B) to other vertices B is minimal.
11
12  find_centroid(): Returns a vertex where all of its subtrees
13  have size <= N/2, where N is the number of nodes in the tree.
14
15  diameter(): The diameter of a tree is the greatest distance
16  d(A,B) between any two of the nodes in the tree.
17
18  Complexity: All three functions are O(V) on the number of
19  vertices in the tree.
20
21  =~=~=~=~= Sample Input =~=~=~=~=
22  6
23  0 1
24  1 2
25  1 4
26  3 4
27  4 5
28
29  =~=~=~=~= Sample Output =~=~=~=~=
30  Center(s): 1 4
31  Centroid: 4
32  Diameter: 3
33
34  */
35
36  #include <iostream>
37  #include <vector>
38  using namespace std;
39
40  const int MAXN = 100;
41  vector<int> adj[MAXN];
42
43  vector<int> find_centers(int n) {
44    vector<int> leaves, degree(n);
45    for (int i = 0; i < n; i++) {
46      degree[i] = adj[i].size();
47      if (degree[i] <= 1) leaves.push_back(i);
48    }
49    int removed = leaves.size();
50    while (removed < n) {
51      vector<int> nleaves;
52      for (int i = 0; i < (int)leaves.size(); i++) {
53        int u = leaves[i];
54        for (int j = 0; j < (int)adj[u].size(); j++) {
55          int v = adj[u][j];
56          if (--degree[v] == 1)
57            nleaves.push_back(v);
58        }
59      }
60      leaves = nleaves;
61      removed += leaves.size();
62    }
63    return leaves;
64  }
65
```

```
66   int find_centroid(int n, int u = 0, int p = -1) {
67     int cnt = 1, v;
68     bool good_center = true;
69     for (int j = 0; j < (int)adj[u].size(); j++) {
70       if ((v = adj[u][j]) == p) continue;
71       int res = find_centroid(n, v, u);
72       if (res >= 0) return res;
73       int size = -res;
74       good_center &= (size <= n / 2);
75       cnt += size;
76     }
77     good_center &= (n - cnt <= n / 2);
78     return good_center ? u : -cnt;
79   }
80
81   pair<int, int> dfs(int u, int p, int depth) {
82     pair<int, int> res = make_pair(depth, u);
83     for (int j = 0; j < (int)adj[u].size(); j++)
84       if (adj[u][j] != p)
85         res = max(res, dfs(adj[u][j], u, depth + 1));
86     return res;
87   }
88
89   int diameter() {
90     int furthest_vertex = dfs(0, -1, 0).second;
91     return dfs(furthest_vertex, -1, 0).first;
92   }
93
94   int main() {
95     int nodes, u, v;
96     cin >> nodes;
97     for (int i = 0; i < nodes - 1; i++) {
98       cin >> u >> v;
99       adj[u].push_back(v);
100      adj[v].push_back(u);
101    }
102    vector<int> centers = find_centers(nodes);
103    cout << "Center(s):";
104    for (int i = 0; i < (int)centers.size(); i++)
105      cout << "␣" << centers[i];
106    cout << "\nCentroid:␣" << find_centroid(nodes);
107    cout << "\nDiameter:␣" << diameter() << "\n";
108    return 0;
109  }
```

2.2   Shortest Paths

2.2.1   Breadth First Search

```
1   /*
2
3   2.2.1 - Shortest Path (Breadth First Search)
4
5   Description: Given an unweighted graph, traverse all reachable
6   nodes from a source node and determine the shortest path.
7
```

```
 8  Complexity: O(V+E) on the number of vertices and edges.
 9
10  Note: The line "for (q.push(start); !q.empty(); q.pop())"
11  is simply a mnemonic for looping a BFS with a FIFO queue.
12  This will not work as intended with a priority queue, such as in
13  Dijkstra's algorithm for solving weighted shortest paths
14
15  =~=~=~=~= Sample Input =~=~=~=~=
16  4 5
17  0 1
18  0 3
19  1 2
20  1 3
21  2 3
22  0 3
23
24  =~=~=~=~= Sample Output =~=~=~=~=
25  The shortest distance from 0 to 3 is 2.
26  Take the path: 0->1->3.
27
28  */
29
30  #include <iostream>
31  #include <queue>
32  #include <vector>
33  using namespace std;
34
35  const int MAXN = 100, INF = 0x3f3f3f3f;
36  int dist[MAXN], pred[MAXN];
37  vector<int> adj[MAXN];
38
39  void bfs(int nodes, int start) {
40    vector<bool> vis(nodes, false);
41    for (int i = 0; i < nodes; i++) {
42      dist[i] = INF;
43      pred[i] = -1;
44    }
45    int u, v, d;
46    queue<pair<int, int> > q;
47    q.push(make_pair(start, 0));
48    while (!q.empty()) {
49      u = q.front().first;
50      d = q.front().second;
51      q.pop();
52      vis[u] = true;
53      for (int j = 0; j < (int)adj[u].size(); j++) {
54        if (vis[v = adj[u][j]]) continue;
55        dist[v] = d + 1;
56        pred[v] = u;
57        q.push(make_pair(v, d + 1));
58      }
59    }
60  }
61
62  //Use the precomputed pred[] array to print the path
63  void print_path(int dest) {
64    int i = 0, j = dest, path[MAXN];
65    while (pred[j] != -1) j = path[++i] = pred[j];
66    cout << "Take the path: ";
```

```
67     while (i > 0) cout << path[i--] << "->";
68     cout << dest << ".\n";
69   }
70
71   int main() {
72     int nodes, edges, u, v, start, dest;
73     cin >> nodes >> edges;
74     for (int i = 0; i < edges; i++) {
75       cin >> u >> v;
76       adj[u].push_back(v);
77     }
78     cin >> start >> dest;
79     bfs(nodes, start);
80     cout << "The shortest distance from " << start;
81     cout << " to " << dest << " is " << dist[dest] << ".\n";
82     print_path(dest);
83     return 0;
84   }
```

2.2.2   Dijkstra's Algorithm

```
1   /*
2
3   2.2.2 - Dijkstra's Algorithm (Single Source Shortest Path)
4
5   Description: Given a directed graph with positive weights only, find
6   the shortest distance to all nodes from a single starting node.
7
8   Implementation Notes: The graph is stored using an adjacency list.
9   This implementation negates distances before adding them to the
10  priority queue, since the container is a max-heap by default. This
11  method is suggested in contests because it is easier than defining
12  special comparators. An alternative would be declaring the queue
13  with template parameters (clearly, this way is very verbose and ugly):
14    priority_queue< pair<int, int>, vector<pair<int, int> >,
15                  greater<pair<int, int> > > pq;
16  If only the path between a single pair of nodes is needed, for speed,
17  we may break out of the loop as soon as the destination is reached
18  by inserting the line "if (a == dest) break;" after the line "pq.pop();"
19
20  Complexity: This version uses an adjacency list and priority queue
21  (internally a binary heap) and has a complexity of O((E+V) log V) =
22  O(E log V). The priority queue and adjacency list improves the
23  simplest O(V^2) version of the algorithm, which uses looping and
24  an adjacency matrix. If the priority queue is implemented as a more
25  sophisticated Fibonacci heap, the complexity becomes O(E + V log V).
26
27  Modification to Shortest Path Faster Algorithm: The code for Dijkstra's
28  algorithm here can be easily modified to become the Shortest Path Faster
29  Algorithm (SPFA) by simply commenting out "visit[a] = true;" and changing
30  the priority queue to a FIFO queue like in BFS. SPFA is a faster version
31  of the Bellman-Ford algorithm, working on negative path lengths (whereas
32  Dijkstra's cannot). Certain graphs can be constructed to make SPFA slow.
33
34  =~=~=~=~= Sample Input =~=~=~=~=
35  4 5
36  0 1 2
```

```
37  0 3 8
38  1 2 2
39  1 3 4
40  2 3 1
41  0 3
42
43  =~=~=~=~= Sample Output =~=~=~=~=
44  The shortest distance from 0 to 3 is 5.
45  Take the path: 0->1->2->3.
46
47  */
48
49  #include <iostream>
50  #include <queue>
51  #include <vector>
52  using namespace std;
53
54  const int MAXN = 100, INF = 0x3f3f3f3f;
55  int dist[MAXN], pred[MAXN];
56  vector<pair<int, int> > adj[MAXN];
57
58  void dijkstra(int nodes, int start) {
59    vector<bool> vis(nodes, false);
60    for (int i = 0; i < nodes; i++) {
61      dist[i] = INF;
62      pred[i] = -1;
63    }
64    int u, v;
65    dist[start] = 0;
66    priority_queue<pair<int, int> > pq;
67    pq.push(make_pair(0, start));
68    while (!pq.empty()) {
69      u = pq.top().second;
70      pq.pop();
71      vis[u] = true;
72      for (int j = 0; j < (int)adj[u].size(); j++) {
73        if (vis[v = adj[u][j].first]) continue;
74        if (dist[v] > dist[u] + adj[u][j].second) {
75          dist[v] = dist[u] + adj[u][j].second;
76          pred[v] = u;
77          pq.push(make_pair(-dist[v], v));
78        }
79      }
80    }
81  }
82
83  //Use the precomputed pred[] array to print the path
84  void print_path(int dest) {
85    int i = 0, j = dest, path[MAXN];
86    while (pred[j] != -1) j = path[++i] = pred[j];
87    cout << "Take␣the␣path:␣";
88    while (i > 0) cout << path[i--] << "->";
89    cout << dest << ".\n";
90  }
91
92  int main() {
93    int nodes, edges, u, v, w, start, dest;
94    cin >> nodes >> edges;
95    for (int i = 0; i < edges; i++) {
```

```
96      cin >> u >> v >> w;
97      adj[u].push_back(make_pair(v, w));
98    }
99    cin >> start >> dest;
100    dijkstra(nodes, start);
101    cout << "The␣shortest␣distance␣from␣" << start;
102    cout << "␣to␣" << dest << "␣is␣" << dist[dest] << ".\n";
103    print_path(dest);
104    return 0;
105  }
```

2.2.3   Bellman-Ford Algorithm

```
1  /*
2
3  2.2.3 - Bellman-Ford Algorithm (Single-Source Shortest Path)
4
5  Description: Given a directed graph with positive or negative weights
6  but no negative cycles, find the shortest distance to all nodes from
7  a single starting node. The input graph is stored using an edge list.
8
9  Complexity: O(V*E) on the number of vertices and edges, respectively.
10
11  =~=~=~=~= Sample Input =~=~=~=~=
12  3 3
13  0 1 1
14  1 2 2
15  0 2 5
16  0 2
17
18  =~=~=~=~= Sample Output =~=~=~=~=
19  The shortest distance from 0 to 2 is 3.
20  Take the path: 0->1->2.
21
22  */
23
24  #include <iostream>
25  #include <stdexcept>
26  #include <vector>
27  using namespace std;
28
29  struct edge { int u, v, w; };
30
31  const int MAXN = 100, INF = 0x3f3f3f3f;
32  int dist[MAXN], pred[MAXN];
33  vector<edge> e;
34
35  void bellman_ford(int nodes, int start) {
36    for (int i = 0; i < nodes; i++) {
37      dist[i] = INF;
38      pred[i] = -1;
39    }
40    dist[start] = 0;
41    for (int i = 0; i < nodes; i++) {
42      for (int j = 0; j < (int)e.size(); j++) {
43        if (dist[e[j].v] > dist[e[j].u] + e[j].w) {
44          dist[e[j].v] = dist[e[j].u] + e[j].w;
```

```
45          pred[e[j].v] = e[j].u;
46        }
47      }
48    }
49    //optional: report negative-weight cycles
50    for (int i = 0; i < (int)e.size(); i++)
51      if (dist[e[i].v] > dist[e[i].u] + e[i].w)
52        throw std::runtime_error("Negative-weight found");
53  }
54
55  //Use the precomputed pred[] array to print the path
56  void print_path(int dest) {
57    int i = 0, j = dest, path[MAXN];
58    while (pred[j] != -1) j = path[++i] = pred[j];
59    cout << "Take the path: ";
60    while (i > 0) cout << path[i--] << "->";
61    cout << dest << ".\n";
62  }
63
64  int main() {
65    int nodes, edges, u, v, w, start, dest;
66    cin >> nodes >> edges;
67    for (int i = 0; i < edges; i++) {
68      cin >> u >> v >> w;
69      e.push_back((edge){u, v, w});
70    }
71    cin >> start >> dest;
72    bellman_ford(nodes, start);
73    cout << "The shortest distance from " << start;
74    cout << " to " << dest << " is " << dist[dest] << ".\n";
75    print_path(dest);
76    return 0;
77  }
```

2.2.4   Floyd-Warshall Algorithm

```
1   /*
2
3   2.2.4 - Floyd-Warshall Algorithm (All-Pairs Shortest Path)
4
5   Description: Given a directed graph with positive or negative
6   weights but no negative cycles, find the shortest distance
7   between all pairs of nodes. The input graph is stored using
8   an adjacency matrix. Note that the input adjacency matrix
9   is converted to the distance matrix afterwards. If you still
10  need the adjacencies afterwards, back it up at the beginning.
11
12  Complexity: O(V^3) on the number of vertices.
13
14  =~=~=~=~= Sample Input =~=~=~=~=
15  3 3
16  0 1 1
17  1 2 2
18  0 2 5
19  0 2
20
21  =~=~=~=~= Sample Output =~=~=~=~=
```

```
22  The shortest distance from 0 to 2 is 3.
23  Take the path: 0->1->2.
24
25  */
26
27  #include <iostream>
28  using namespace std;
29
30  const int MAXN = 100, INF = 0x3f3f3f3f;
31  int dist[MAXN][MAXN], next[MAXN][MAXN];
32
33  void initialize(int nodes) {
34    for (int i = 0; i < nodes; i++)
35      for (int j = 0; j < nodes; j++) {
36        dist[i][j] = (i == j) ? 0 : INF;
37        next[i][j] = -1;
38      }
39  }
40
41  void floyd_warshall(int nodes) {
42    for (int k = 0; k < nodes; k++)
43     for (int i = 0; i < nodes; i++)
44      for (int j = 0; j < nodes; j++)
45        if (dist[i][j] > dist[i][k] + dist[k][j]) {
46          dist[i][j] = dist[i][k] + dist[k][j];
47          next[i][j] = k;
48        }
49  }
50
51  void print_path(int u, int v) {
52    if (next[u][v] != -1) {
53      print_path(u, next[u][v]);
54      cout << next[u][v];
55      print_path(next[u][v], v);
56    } else cout << "->";
57  }
58
59  int main() {
60    int nodes, edges, u, v, w, start, dest;
61    cin >> nodes >> edges;
62    initialize(nodes);
63    for (int i = 0; i < edges; i++) {
64      cin >> u >> v >> w;
65      dist[u][v] = w;
66    }
67    cin >> start >> dest;
68    floyd_warshall(nodes);
69    cout << "The shortest distance from " << start;
70    cout << " to " << dest << " is ";
71    cout << dist[start][dest] << ".\n";
72
73    //Use next[][] to recursively print the path
74    cout << "Take the path " << start;
75    print_path(start, dest);
76    cout << dest << ".\n";
77    return 0;
78  }
```

2.3   Connectivity

2.3.1   Strongly Connected Components (Kosaraju's Algorithm)

```
1   /*
2
3   2.3.1 - Strongly Connected Components (Kosaraju's Algorithm)
4
5   Description: Determines the strongly connected components (SCC)
6   from a given directed graph. Given a directed graph, its SCCs
7   are its maximal strongly connected sub-graphs. A graph is
8   strongly connected if there is a path from each node to every
9   other node. Condensing the strongly connected components of a
10  graph into single nodes will result in a directed acyclic graph.
11  The input is stored in an adjacency list.
12
13  Complexity: O(V+E) on the number of vertices and edges.
14
15  Comparison with other SCC algorithms:
16  The strongly connected components of a graph can be efficiently
17  computed using Kosaraju's algorithm, Tarjan's algorithm, or the
18  path-based strong component algorithm. Tarjan's algorithm can
19  be seen as an improved version of Kosaraju's because it performs
20  a single DFS rather than two. Though they both have the same
21  complexity, Tarjan's algorithm is much more efficient in
22  practice. However, Kosaraju's algorithm is conceptually simpler.
23
24  =~=~=~=~= Sample Input =~=~=~=~=
25  8 14
26  0 1
27  1 2
28  1 4
29  1 5
30  2 3
31  2 6
32  3 2
33  3 7
34  4 0
35  4 5
36  5 6
37  6 5
38  7 3
39  7 6
40
41  =~=~=~=~= Sample Output =~=~=~=~=
42  Component: 1 4 0
43  Component: 7 3 2
44  Component: 5 6
45
46  */
47
48  #include <algorithm> /* std::fill(), std::reverse() */
49  #include <iostream>
50  #include <vector>
51  using namespace std;
52
53  const int MAXN = 100;
```

```cpp
54   vector<bool> vis(MAXN);
55   vector<int> adj[MAXN], order;
56   vector<vector<int> > scc;
57
58   void dfs(vector<int> graph[], vector<int> & res, int u) {
59     vis[u] = true;
60     for (int j = 0; j < (int)graph[u].size(); j++)
61       if (!vis[graph[u][j]])
62         dfs(graph, res, graph[u][j]);
63     res.push_back(u);
64   }
65
66   void kosaraju(int nodes) {
67     scc.clear();
68     order.clear();
69     vector<int> rev[nodes];
70     fill(vis.begin(), vis.end(), false);
71     for (int i = 0; i < nodes; i++)
72       if (!vis[i]) dfs(adj, order, i);
73     for (int i = 0; i < nodes; i++)
74       for (int j = 0; j < (int)adj[i].size(); j++)
75         rev[adj[i][j]].push_back(i);
76     fill(vis.begin(), vis.end(), false);
77     reverse(order.begin(), order.end());
78     for (int i = 0; i < (int)order.size(); i++) {
79       if (vis[order[i]]) continue;
80       vector<int> component;
81       dfs(rev, component, order[i]);
82       scc.push_back(component);
83     }
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].push_back(v);
92     }
93     kosaraju(nodes);
94     for (int i = 0; i < (int)scc.size(); i++) {
95       cout << "Component:";
96       for (int j = 0; j < (int)scc[i].size(); j++)
97         cout << "␣" << scc[i][j];
98       cout << "\n";
99     }
100    return 0;
101  }
```

2.3.2   Strongly Connected Components (Tarjan's Algorithm)

```
1   /*
2
3   2.3.2 - Strongly Connected Components (Tarjan's Algorithm)
4
5   Description: Determines the strongly connected components (SCC)
6   from a given directed graph. Given a directed graph, its SCCs
```

```
 7   are its maximal strongly connected sub-graphs. A graph is
 8   strongly connected if there is a path from each node to every
 9   other node. Condensing the strongly connected components of a
10   graph into single nodes will result in a directed acyclic graph.
11   The input is stored in an adjacency list.
12
13   In this implementation, a vector is used to emulate a stack
14   for the sake of simplicity. One useful property of Tarjans
15   algorithm is that, while there is nothing special about the
16   ordering of nodes within each component, the resulting DAG
17   is produced in reverse topological order.
18
19   Complexity: O(V+E) on the number of vertices and edges.
20
21   Comparison with other SCC algorithms:
22   The strongly connected components of a graph can be efficiently
23   computed using Kosaraju's algorithm, Tarjan's algorithm, or the
24   path-based strong component algorithm. Tarjan's algorithm can
25   be seen as an improved version of Kosaraju's because it performs
26   a single DFS rather than two. Though they both have the same
27   complexity, Tarjan's algorithm is much more efficient in
28   practice. However, Kosaraju's algorithm is conceptually simpler.
29
30   =~=~=~=~= Sample Input =~=~=~=~=
31   8 14
32   0 1
33   1 2
34   1 4
35   1 5
36   2 3
37   2 6
38   3 2
39   3 7
40   4 0
41   4 5
42   5 6
43   6 5
44   7 3
45   7 6
46
47   =~=~=~=~= Sample Output =~=~=~=~=
48   Component 1: 5 6
49   Component 2: 7 3 2
50   Component 3: 4 1 0
51
52   */
53
54   #include <algorithm> /* std::fill() */
55   #include <iostream>
56   #include <vector>
57   using namespace std;
58
59   const int MAXN = 100, INF = 0x3f3f3f3f;
60   int timer, lowlink[MAXN];
61   vector<bool> vis(MAXN);
62   vector<int> adj[MAXN], stack;
63   vector<vector<int> > scc;
64
65   void dfs(int u) {
```

```
66      lowlink[u] = timer++;
67      vis[u] = true;
68      stack.push_back(u);
69      bool is_component_root = true;
70      int v;
71      for (int j = 0; j < (int)adj[u].size(); j++) {
72        if (!vis[v = adj[u][j]]) dfs(v);
73        if (lowlink[u] > lowlink[v]) {
74          lowlink[u] = lowlink[v];
75          is_component_root = false;
76        }
77      }
78      if (!is_component_root) return;
79      vector<int> component;
80      do {
81        vis[v = stack.back()] = true;
82        stack.pop_back();
83        lowlink[v] = INF;
84        component.push_back(v);
85      } while (u != v);
86      scc.push_back(component);
87    }
88
89    void tarjan(int nodes) {
90      scc.clear();
91      stack.clear();
92      fill(lowlink, lowlink + nodes, 0);
93      fill(vis.begin(), vis.end(), false);
94      timer = 0;
95      for (int i = 0; i < nodes; i++)
96        if (!vis[i]) dfs(i);
97    }
98
99    int main() {
100     int nodes, edges, u, v;
101     cin >> nodes >> edges;
102     for (int i = 0; i < edges; i++) {
103       cin >> u >> v;
104       adj[u].push_back(v);
105     }
106     tarjan(nodes);
107     for (int i = 0; i < (int)scc.size(); i++) {
108       cout << "Component:";
109       for (int j = 0; j < (int)scc[i].size(); j++)
110         cout << "␣" << scc[i][j];
111       cout << "\n";
112     }
113     return 0;
114   }
```

2.3.3   Bridges, Cut-points, and Biconnectivity

```
1    /*
2
3    2.3.3 - Bridges, Cut-points, and Biconnectivity (Tarjan's)
4
5    Description: The following operations apply to undirected graphs.
```

```
6
7    A bridge is an edge, when deleted, increases the number of
8    connected components. An edge is a bridge if and only if it is not
9    contained in any cycle.
10
11   A cut-point (i.e. cut-vertex or articulation point) is any vertex
12   whose removal increases the number of connected components.
13
14   A biconnected component of a graph is a maximally biconnected
15   subgraph. A biconnected graph is a connected and "nonseparable"
16   graph, meaning that if any vertex were to be removed, the graph
17   will remain connected. Therefore, a biconnected graph has no
18   articulation vertices.
19
20   Any connected graph decomposes into a tree of biconnected
21   components called the "block tree" of the graph. An unconnected
22   graph will thus decompose into a "block forest."
23
24   See: http://en.wikipedia.org/wiki/Biconnected_component
25
26   Complexity: O(V+E) on the number of vertices and edges.
27
28   =~=~=~=~= Sample Input =~=~=~=~=
29   8 6
30   0 1
31   0 5
32   1 2
33   1 5
34   3 7
35   4 5
36
37   =~=~=~=~= Sample Output =~=~=~=~=
38   Cut Points: 5 1
39   Bridges:
40   1 2
41   5 4
42   3 7
43   Edge-Biconnected Components:
44   Component 1: 2
45   Component 2: 4
46   Component 3: 5 1 0
47   Component 4: 7
48   Component 5: 3
49   Component 6: 6
50   Adjacency List for Block Forest:
51   0 => 2
52   1 => 2
53   2 => 0 1
54   3 => 4
55   4 => 3
56   5 =>
57
58   */
59
60   #include <algorithm> /* std::fill(), std::min() */
61   #include <iostream>
62   #include <vector>
63   using namespace std;
64
```

```
65   const int MAXN = 100;
66   int timer, lowlink[MAXN], tin[MAXN], comp[MAXN];
67   vector<bool> vis(MAXN);
68   vector<int> adj[MAXN], bcc_forest[MAXN];
69   vector<int> stack, cutpoints;
70   vector<vector<int> > bcc;
71   vector<pair<int, int> > bridges;
72
73   void dfs(int u, int p) {
74     vis[u] = true;
75     lowlink[u] = tin[u] = timer++;
76     stack.push_back(u);
77     int v, children = 0;
78     bool cutpoint = false;
79     for (int j = 0; j < (int)adj[u].size(); j++) {
80       if ((v = adj[u][j]) == p) continue;
81       if (vis[v]) {
82         //lowlink[u] = min(lowlink[u], lowlink[v]);
83         lowlink[u] = min(lowlink[u], tin[v]);
84       } else {
85         dfs(v, u);
86         lowlink[u] = min(lowlink[u], lowlink[v]);
87         cutpoint |= (lowlink[v] >= tin[u]);
88         if (lowlink[v] > tin[u])
89           bridges.push_back(make_pair(u, v));
90         children++;
91       }
92     }
93     if (p == -1) cutpoint = (children >= 2);
94     if (cutpoint) cutpoints.push_back(u);
95     if (lowlink[u] == tin[u]) {
96       vector<int> component;
97       do {
98         v = stack.back();
99         stack.pop_back();
100        component.push_back(v);
101      } while (u != v);
102      bcc.push_back(component);
103    }
104  }
105
106  void tarjan(int nodes) {
107    bcc.clear();
108    bridges.clear();
109    cutpoints.clear();
110    stack.clear();
111    fill(lowlink, lowlink + nodes, 0);
112    fill(tin, tin + nodes, 0);
113    fill(vis.begin(), vis.end(), false);
114    timer = 0;
115    for (int i = 0; i < nodes; i++)
116      if (!vis[i]) dfs(i, -1);
117  }
118
119  //condenses each bcc to a node and generates a tree
120  //global variables adj and bcc must be set beforehand
121  void get_block_tree(int nodes) {
122    fill(comp, comp + nodes, 0);
123    for (int i = 0; i < nodes; i++) bcc_forest[i].clear();
```

```
124    for (int i = 0; i < (int)bcc.size(); i++)
125      for (int j = 0; j < (int)bcc[i].size(); j++)
126        comp[bcc[i][j]] = i;
127    for (int i = 0; i < nodes; i++)
128      for (int j = 0; j < (int)adj[i].size(); j++)
129        if (comp[i] != comp[adj[i][j]])
130          bcc_forest[comp[i]].push_back(comp[adj[i][j]]);
131  }
132
133  int main() {
134    int nodes, edges, u, v;
135    cin >> nodes >> edges;
136    for (int i = 0; i < edges; i++) {
137      cin >> u >> v;
138      adj[u].push_back(v);
139      adj[v].push_back(u);
140    }
141    tarjan(nodes);
142    cout << "Cut-points:";
143    for (int i = 0; i < (int)cutpoints.size(); i++)
144      cout << "␣" << cutpoints[i];
145    cout << "\nBridges:\n";
146    for (int i = 0; i < (int)bridges.size(); i++)
147      cout << bridges[i].first << "␣" << bridges[i].second << "\n";
148    cout << "Edge-Biconnected␣Components:\n";
149    for (int i = 0; i < (int)bcc.size(); i++) {
150      cout << "Component:";
151      for (int j = 0; j < (int)bcc[i].size(); j++)
152        cout << "␣" << bcc[i][j];
153      cout << "\n";
154    }
155    get_block_tree(nodes);
156    cout << "Adjacency␣List␣for␣Block␣Forest:\n";
157    for (int i = 0; i < (int)bcc.size(); i++) {
158      cout << i << "␣=>";
159      for (int j = 0; j < (int)bcc_forest[i].size(); j++)
160        cout << "␣" << bcc_forest[i][j];
161      cout << "\n";
162    }
163    return 0;
164  }
```

2.4 Minimal Spanning Trees

2.4.1 Prim's Algorithm

```
1   /*
2
3   2.4.1 - Minimum Spanning Tree (Prim's Algorithm)
4
5   Description: Given an undirected graph, its minimum spanning
6   tree (MST) is a tree connecting all nodes with a subset of its
7   edges such that their total weight is minimized. Prim's algorithm
8   greedily selects edges from a priority queue, and is similar to
9   Dijkstra's algorithm, where instead of processing nodes, we
10  process individual edges. If the graph is not connected, Prim's
```

```
11  algorithm will produce the minimum spanning forest. The input
12  graph is stored in an adjacency list.
13
14  Note that the concept of the minimum spanning tree makes Prim's
15  algorithm work with negative weights. In fact, a big positive
16  constant added to all of the edge weights of the graph will not
17  change the resulting spanning tree.
18
19  Implementation Notes: Similar to the implementation of Dijkstra's
20  algorithm in the previous section, weights are negated before they
21  are added to the priority queue (and negated once again when they
22  are retrieved). To find the maximum spanning tree, simply skip the
23  two negation steps and the max weighted edges will be prioritized.
24
25  Complexity: This version uses an adjacency list and priority queue
26  (internally a binary heap) and has a complexity of O((E+V) log V) =
27  O(E log V). The priority queue and adjacency list improves the
28  simplest O(V^2) version of the algorithm, which uses looping and
29  an adjacency matrix. If the priority queue is implemented as a more
30  sophisticated Fibonacci heap, the complexity becomes O(E + V log V).
31
32  =~=~=~=~= Sample Input =~=~=~=~=
33  7 7
34  0 1 4
35  1 2 6
36  2 0 3
37  3 4 1
38  4 5 2
39  5 6 3
40  6 4 4
41
42  =~=~=~=~= Sample Output =~=~=~=~=
43  Total distance: 13
44  0<->2
45  0<->1
46  3<->4
47  4<->5
48  5<->6
49
50  */
51
52  #include <algorithm> /* std::fill() */
53  #include <iostream>
54  #include <queue>
55  #include <vector>
56  using namespace std;
57
58  const int MAXN = 100;
59  vector<pair<int, int> > adj[MAXN], mst;
60
61  int prim(int nodes) {
62    mst.clear();
63    vector<bool> vis(nodes);
64    int u, v, w, total_dist = 0;
65    for (int i = 0; i < nodes; i++) {
66      if (vis[i]) continue;
67      vis[i] = true;
68      priority_queue<pair<int, pair<int, int> > > pq;
69      for (int j = 0; j < (int)adj[i].size(); j++)
```

```
70        pq.push(make_pair(-adj[i][j].second,
71                  make_pair(i, adj[i][j].first)));
72      while (!pq.empty()) {
73        w = -pq.top().first;
74        u = pq.top().second.first;
75        v = pq.top().second.second;
76        pq.pop();
77        if (vis[u] && !vis[v]) {
78          vis[v] = true;
79          if (v != i) {
80            mst.push_back(make_pair(u, v));
81            total_dist += w;
82          }
83          for (int j = 0; j < (int)adj[v].size(); j++)
84            pq.push(make_pair(-adj[v][j].second,
85                      make_pair(v, adj[v][j].first)));
86        }
87      }
88    }
89    return total_dist;
90  }
91
92  int main() {
93    int nodes, edges, u, v, w;
94    cin >> nodes >> edges;
95    for (int i = 0; i < edges; i++) {
96      cin >> u >> v >> w;
97      adj[u].push_back(make_pair(v, w));
98      adj[v].push_back(make_pair(u, w));
99    }
100   cout << "Total distance: " << prim(nodes) << "\n";
101   for (int i = 0; i < (int)mst.size(); i++)
102     cout << mst[i].first << "<->" << mst[i].second << "\n";
103   return 0;
104 }
```

2.4.2   Kruskal's Algorithm

```
1  /*
2
3  2.4.2 - Minimum Spanning Tree (Kruskal's Algorithm)
4
5  Description: Given an undirected graph, its minimum spanning
6  tree (MST) is a tree connecting all nodes with a subset of its
7  edges such that their total weight is minimized. If the graph
8  is not connected, Kruskal's algorithm will produce the minimum
9  spanning forest. The input graph is stored in an edge list.
10
11 Complexity: O(E log V) on the number of edges and vertices.
12
13 =~=~=~=~= Sample Input =~=~=~=~=
14 7 7
15 0 1 4
16 1 2 6
17 2 0 3
18 3 4 1
19 4 5 2
```

```
20  5 6 3
21  6 4 4
22
23  =~=~=~=~= Sample Output =~=~=~=~=
24  Total distance: 13
25  3<->4
26  4<->5
27  2<->0
28  5<->6
29  0<->1
30
31  Note: If you already have a disjoint set data structure,
32  then the middle section of the program can be replaced by:
33
34  disjoint_set_forest<int> dsf;
35  for (int i = 0; i < nodes; i++) dsf.make_set(i);
36  for (int i = 0; i < E.size(); i++) {
37    a = E[i].second.first;
38    b = E[i].second.second;
39    if (!dsf.is_united(a, b)) {
40       ...
41       dsf.unite(a, b);
42    }
43  }
44
45  */
46
47  #include <algorithm> /* std::sort() */
48  #include <iostream>
49  #include <vector>
50  using namespace std;
51
52  const int MAXN = 100;
53  int root[MAXN];
54  vector<pair<int, pair<int, int> > > E;
55  vector<pair<int, int> > mst;
56
57  int find_root(int x) {
58    if (root[x] != x)
59      root[x] = find_root(root[x]);
60    return root[x];
61  }
62
63  int kruskal(int nodes) {
64    mst.clear();
65    sort(E.begin(), E.end());
66    int u, v, total_dist = 0;
67    for (int i = 0; i < nodes; i++) root[i] = i;
68    for (int i = 0; i < (int)E.size(); i++) {
69      u = find_root(E[i].second.first);
70      v = find_root(E[i].second.second);
71      if (u != v) {
72        mst.push_back(E[i].second);
73        total_dist += E[i].first;
74        root[u] = root[v];
75      }
76    }
77    return total_dist;
78  }
```

```
79
80  int main() {
81    int nodes, edges, u, v, w;
82    cin >> nodes >> edges;
83    for (int i = 0; i < edges; i++) {
84      cin >> u >> v >> w;
85      E.push_back(make_pair(w, make_pair(u, v)));
86    }
87    cout << "Total distance: " << kruskal(nodes) << "\n";
88    for (int i = 0; i < (int)mst.size(); i++)
89      cout << mst[i].first << "<->" << mst[i].second << "\n";
90    return 0;
91  }
```

2.5 Maximum Flow

2.5.1 Ford-Fulkerson Algorithm

```
1   /*
2
3   2.5.1 - Maximum Flow (Ford-Fulkerson Algorithm)
4
5   Description: Given a flow network, find a flow from a single
6   source node to a single sink node that is maximized. Note
7   that in this implementation, the adjacency matrix cap[][]
8   will be modified by the function ford_fulkerson() after it's
9   been called. Make a back-up if you require it afterwards.
10
11  Complexity: O(V^2*|F|), where V is the number of
12  vertices and |F| is the magnitude of the max flow.
13
14  Real-valued capacities:
15  The Ford-Fulkerson algorithm is only optimal on graphs with
16  integer capacities; there exists certain real capacity inputs
17  for which it will never terminate. The Edmonds-Karp algorithm
18  is an improvement using BFS, supporting real number capacities.
19
20  =~=~=~=~= Sample Input =~=~=~=~=
21  6 8
22  0 1 3
23  0 2 3
24  1 2 2
25  1 3 3
26  2 4 2
27  3 4 1
28  3 5 2
29  4 5 3
30  0 5
31
32  =~=~=~=~= Sample Output =~=~=~=~=
33  5
34
35  */
36
37  #include <algorithm> /* std::fill() */
38  #include <iostream>
```

```
39   #include <vector>
40   using namespace std;
41
42   const int MAXN = 100, INF = 0x3f3f3f3f;
43   int nodes, source, sink, cap[MAXN][MAXN];
44   vector<bool> vis(MAXN);
45
46   int dfs(int u, int f) {
47     if (u == sink) return f;
48     vis[u] = true;
49     for (int v = 0; v < nodes; v++) {
50       if (!vis[v] && cap[u][v] > 0) {
51         int df = dfs(v, min(f, cap[u][v]));
52         if (df > 0) {
53           cap[u][v] -= df;
54           cap[v][u] += df;
55           return df;
56         }
57       }
58     }
59     return 0;
60   }
61
62   int ford_fulkerson() {
63     int max_flow = 0;
64     for (;;) {
65       fill(vis.begin(), vis.end(), false);
66       int df = dfs(source, INF);
67       if (df == 0) break;
68       max_flow += df;
69     }
70     return max_flow;
71   }
72
73   int main() {
74     int edges, u, v, capacity;
75     cin >> nodes >> edges;
76     for (int i = 0; i < edges; i++) {
77       cin >> u >> v >> capacity;
78       cap[u][v] = capacity;
79     }
80     cin >> source >> sink;
81     cout << ford_fulkerson() << "\n";
82     return 0;
83   }
```

2.5.2   Edmonds-Karp Algorithm

```
1   /*
2
3   2.5.2 - Maximum Flow (Edmonds-Karp Algorithm)
4
5   Description: Given a flow network, find a flow from a single
6   source node to a single sink node that is maximized. Note
7   that in this implementation, the adjacency list adj[] will
8   be modified by the function edmonds_karp() after it's been called.
9
```

```
10  Complexity: O(min(V*E^2, E*|F|)), where V is the number of
11  vertices, E is the number of edges, and |F| is the magnitude of
12  the max flow. This improves the original Ford-Fulkerson algorithm,
13  which runs in O(E*|F|). As the Edmonds-Karp algorithm is also
14  bounded by O(E*|F|), it is guaranteed to be at least as fast as
15  Ford-Fulkerson. For an even faster algorithm, see Dinic's
16  algorithm in the next section, which runs in O(V^2*E).
17
18  Real-valued capacities:
19  Although the Ford-Fulkerson algorithm is only optimal on graphs
20  with integer capacities, the Edmonds-Karp algorithm also works
21  correctly on real-valued capacities.
22
23  =~=~=~=~= Sample Input =~=~=~=~=
24  6 8
25  0 1 3
26  0 2 3
27  1 2 2
28  1 3 3
29  2 4 2
30  3 4 1
31  3 5 2
32  4 5 3
33  0 5
34
35  =~=~=~=~= Sample Output =~=~=~=~=
36  5
37
38  */
39
40  #include <algorithm> /* std::fill(), std::min() */
41  #include <iostream>
42  #include <vector>
43  using namespace std;
44
45  struct edge { int s, t, rev, cap, f; };
46
47  const int MAXN = 100, INF = 0x3f3f3f3f;
48  vector<edge> adj[MAXN];
49
50  void add_edge(int s, int t, int cap) {
51    adj[s].push_back((edge){s, t, (int)adj[t].size(), cap, 0});
52    adj[t].push_back((edge){t, s, (int)adj[s].size() - 1, 0, 0});
53  }
54
55  int edmonds_karp(int nodes, int source, int sink) {
56    static int q[MAXN];
57    int max_flow = 0;
58    for (;;) {
59      int qt = 0;
60      q[qt++] = source;
61      edge * pred[nodes];
62      fill(pred, pred + nodes, (edge*)0);
63      for (int qh = 0; qh < qt && !pred[sink]; qh++) {
64        int u = q[qh];
65        for (int j = 0; j < (int)adj[u].size(); j++) {
66          edge * e = &adj[u][j];
67          if (!pred[e->t] && e->cap > e->f) {
68            pred[e->t] = e;
```

```
69            q[qt++] = e->t;
70          }
71        }
72      }
73      if (!pred[sink]) break;
74      int df = INF;
75      for (int u = sink; u != source; u = pred[u]->s)
76        df = min(df, pred[u]->cap - pred[u]->f);
77      for (int u = sink; u != source; u = pred[u]->s) {
78        pred[u]->f += df;
79        adj[pred[u]->t][pred[u]->rev].f -= df;
80      }
81      max_flow += df;
82    }
83    return max_flow;
84  }
85
86  int main() {
87    int nodes, edges, u, v, capacity, source, sink;
88    cin >> nodes >> edges;
89    for (int i = 0; i < edges; i++) {
90      cin >> u >> v >> capacity;
91      add_edge(u, v, capacity);
92    }
93    cin >> source >> sink;
94    cout << edmonds_karp(nodes, source, sink) << "\n";
95    return 0;
96  }
```

2.5.3   Dinic's Algorithm

```
1  /*
2
3  2.5.3 - Maximum Flow (Dinic's Blocking Flow Algorithm)
4
5  Description: Given a flow network, find a flow from a single
6  source node to a single sink node that is maximized. Note
7  that in this implementation, the adjacency list adj[] will
8  be modified by the function dinic() after it's been called.
9
10 Complexity: O(V^2*E) on the number of vertices and edges.
11
12 Comparison with Edmonds-Karp Algorithm:
13 Dinic's is similar to the Edmonds-Karp algorithm in that it
14 uses the shortest augmenting path. The introduction of the
15 concepts of the level graph and blocking flow enable Dinic's
16 algorithm to achieve its better performance. Hence, Dinic's
17 algorithm is also called Dinic's blocking flow algorithm.
18
19 =~=~=~=~= Sample Input =~=~=~=~=
20 6 8
21 0 1 3
22 0 2 3
23 1 2 2
24 1 3 3
25 2 4 2
26 3 4 1
```

```
27  3 5 2
28  4 5 3
29  0 5
30
31  =~=~=~=~= Sample Output =~=~=~=~=
32  5
33
34  */
35
36  #include <algorithm> /* std::fill(), std::min() */
37  #include <iostream>
38  #include <vector>
39  using namespace std;
40
41  struct edge { int to, rev, cap, f; };
42
43  const int MAXN = 100, INF = 0x3f3f3f3f;
44  int dist[MAXN], ptr[MAXN];
45  vector<edge> adj[MAXN];
46
47  void add_edge(int s, int t, int cap) {
48    adj[s].push_back((edge){t, (int)adj[t].size(), cap, 0});
49    adj[t].push_back((edge){s, (int)adj[s].size() - 1, 0, 0});
50  }
51
52  bool dinic_bfs(int nodes, int source, int sink) {
53    fill(dist, dist + nodes, -1);
54    dist[source] = 0;
55    int q[nodes], qh = 0, qt = 0;
56    q[qt++] = source;
57    while (qh < qt) {
58      int u = q[qh++];
59      for (int j = 0; j < (int)adj[u].size(); j++) {
60        edge & e = adj[u][j];
61        if (dist[e.to] < 0 && e.f < e.cap) {
62          dist[e.to] = dist[u] + 1;
63          q[qt++] = e.to;
64        }
65      }
66    }
67    return dist[sink] >= 0;
68  }
69
70  int dinic_dfs(int u, int f, int sink) {
71    if (u == sink) return f;
72    for (; ptr[u] < (int)adj[u].size(); ptr[u]++) {
73      edge &e = adj[u][ptr[u]];
74      if (dist[e.to] == dist[u] + 1 && e.f < e.cap) {
75        int df = dinic_dfs(e.to, min(f, e.cap - e.f), sink);
76        if (df > 0) {
77          e.f += df;
78          adj[e.to][e.rev].f -= df;
79          return df;
80        }
81      }
82    }
83    return 0;
84  }
85
```

```
86  int dinic(int nodes, int source, int sink) {
87    int max_flow = 0, delta;
88    while (dinic_bfs(nodes, source, sink)) {
89      fill(ptr, ptr + nodes, 0);
90      while ((delta = dinic_dfs(source, INF, sink)) != 0)
91        max_flow += delta;
92    }
93    return max_flow;
94  }
95
96  int main() {
97    int nodes, edges, u, v, capacity, source, sink;
98    cin >> nodes >> edges;
99    for (int i = 0; i < edges; i++) {
100     cin >> u >> v >> capacity;
101     add_edge(u, v, capacity);
102   }
103   cin >> source >> sink;
104   cout << dinic(nodes, source, sink) << "\n";
105   return 0;
106 }
```

2.5.4   Push-Relabel Algorithm

```
1   /*
2
3   2.5.4 - Max Flow (Push-Relabel Algorithm)
4
5   Description: Given a flow network, find a flow from a single
6   source node to a single sink node that is maximized. The push-
7   relabel algorithm is considered one of the most efficient
8   maximum flow algorithms. However, unlike the Ford-Fulkerson or
9   Edmonds-Karp algorithms, it cannot take advantage of the fact
10  if max flow itself has a small magnitude.
11
12  Complexity: O(V^3) on the number of vertices.
13
14  =~=~=~=~= Sample Input =~=~=~=~=
15  6 8
16  0 1 3
17  0 2 3
18  1 2 2
19  1 3 3
20  2 4 2
21  3 4 1
22  3 5 2
23  4 5 3
24  0 5
25
26  =~=~=~=~= Sample Output =~=~=~=~=
27  5
28
29  */
30
31  #include <algorithm> /* std::fill(), std::min() */
32  #include <iostream>
33  using namespace std;
```

```
34
35   const int MAXN = 100, INF = 0x3F3F3F3F;
36   int cap[MAXN][MAXN], f[MAXN][MAXN];
37
38   int push_relabel(int nodes, int source, int sink) {
39     int e[nodes], h[nodes], maxh[nodes];
40     fill(e, e + nodes, 0);
41     fill(h, h + nodes, 0);
42     fill(maxh, maxh + nodes, 0);
43     for (int i = 0; i < nodes; i++)
44       fill(f[i], f[i] + nodes, 0);
45     h[source] = nodes - 1;
46     for (int i = 0; i < nodes; i++) {
47       f[source][i] = cap[source][i];
48       f[i][source] = -f[source][i];
49       e[i] = cap[source][i];
50     }
51     int sz = 0;
52     for (;;) {
53       if (sz == 0) {
54         for (int i = 0; i < nodes; i++)
55           if (i != source && i != sink && e[i] > 0) {
56             if (sz != 0 && h[i] > h[maxh[0]]) sz = 0;
57             maxh[sz++] = i;
58           }
59       }
60       if (sz == 0) break;
61       while (sz != 0) {
62         int i = maxh[sz - 1];
63         bool pushed = false;
64         for (int j = 0; j < nodes && e[i] != 0; j++) {
65           if (h[i] == h[j] + 1 && cap[i][j] - f[i][j] > 0) {
66             int df = min(cap[i][j] - f[i][j], e[i]);
67             f[i][j] += df;
68             f[j][i] -= df;
69             e[i] -= df;
70             e[j] += df;
71             if (e[i] == 0) sz--;
72             pushed = true;
73           }
74         }
75         if (!pushed) {
76           h[i] = INF;
77           for (int j = 0; j < nodes; j++)
78             if (h[i] > h[j] + 1 && cap[i][j] - f[i][j] > 0)
79               h[i] = h[j] + 1;
80           if (h[i] > h[maxh[0]]) {
81             sz = 0;
82             break;
83           }
84         }
85       }
86     }
87     int max_flow = 0;
88     for (int i = 0; i < nodes; i++)
89       max_flow += f[source][i];
90     return max_flow;
91   }
92
```

```
93   int main() {
94     int nodes, edges, u, v, capacity, source, sink;
95     cin >> nodes >> edges;
96     for (int i = 0; i < edges; i++) {
97       cin >> u >> v >> capacity;
98       cap[u][v] = capacity;
99     }
100    cin >> source >> sink;
101    cout << push_relabel(nodes, source, sink) << "\n";
102    return 0;
103  }
```

2.6   Backtracking

2.6.1   Max Clique (Bron-Kerbosch Algorithm)

```
1    /*
2
3    2.6.1 - Backtracking: Maximum Clique (Bron-Kerbosch Algorithm)
4
5    Description: Given an undirected graph, determine a subset of
6    the graph's vertices such that every pair of vertices in the
7    subset are connected by an edge, and that the subset is as
8    large as possible. For the weighted version, each vertex is
9    assigned a weight and the objective is to find the clique in
10   the graph that has maximum total weight.
11
12   Complexity: O(3^(V/3)) where V is the number of vertices.
13
14   =~=~=~=~= Sample Input =~=~=~=~=
15   5 8
16   0 1
17   0 2
18   0 3
19   1 2
20   1 3
21   2 3
22   3 4
23   4 2
24   10 20 30 40 50
25
26   =~=~=~=~= Sample Output =~=~=~=~=
27   Max unweighted clique: 4
28   Max weighted clique: 120
29
30   */
31
32   #include <algorithm> /* std::fill(), std::max() */
33   #include <bitset>
34   #include <iostream>
35   #include <vector>
36   using namespace std;
37
38   const int MAXN = 35;
39   typedef bitset<MAXN> bits;
40   typedef unsigned long long ull;
```

```
41
42   int w[MAXN];
43   bool adj[MAXN][MAXN];
44
45   int rec(int nodes, bits & curr, bits & pool, bits & excl) {
46     if (pool.none() && excl.none()) return curr.count();
47     int ans = 0, u = 0;
48     for (int v = 0; v < nodes; v++)
49       if (pool[v] || excl[v]) u = v;
50     for (int v = 0; v < nodes; v++) {
51       if (!pool[v] || adj[u][v]) continue;
52       bits ncurr, npool, nexcl;
53       for (int i = 0; i < nodes; i++) ncurr[i] = curr[i];
54       ncurr[v] = true;
55       for (int j = 0; j < nodes; j++) {
56         npool[j] = pool[j] && adj[v][j];
57         nexcl[j] = excl[j] && adj[v][j];
58       }
59       ans = max(ans, rec(nodes, ncurr, npool, nexcl));
60       pool[v] = false;
61       excl[v] = true;
62     }
63     return ans;
64   }
65
66   int bron_kerbosch(int nodes) {
67     bits curr, excl, pool;
68     pool.flip();
69     return rec(nodes, curr, pool, excl);
70   }
71
72   //This is a fast implementation using bitmasks.
73   //Precondition: the number of nodes must be less than 64.
74   int bron_kerbosch_weighted(int nodes, ull g[], ull curr, ull pool, ull excl) {
75     if (pool == 0 && excl == 0) {
76       int res = 0, u = __builtin_ctzll(curr);
77       while (u < nodes) {
78         res += w[u];
79         u += __builtin_ctzll(curr >> (u + 1)) + 1;
80       }
81       return res;
82     }
83     if (pool == 0) return -1;
84     int res = -1, pivot = __builtin_ctzll(pool | excl);
85     ull z = pool & ~g[pivot];
86     int u = __builtin_ctzll(z);
87     while (u < nodes) {
88       res = max(res, bron_kerbosch_weighted(nodes, g, curr | (1LL << u),
89                                             pool & g[u], excl & g[u]));
90       pool ^= 1LL << u;
91       excl |= 1LL << u;
92       u += __builtin_ctzll(z >> (u + 1)) + 1;
93     }
94     return res;
95   }
96
97   int bron_kerbosch_weighted(int nodes) {
98     ull g[nodes];
99     for (int i = 0; i < nodes; i++) {
```

```
100        g[i] = 0;
101        for (int j = 0; j < nodes; j++)
102          if (adj[i][j]) g[i] |= 1LL << j;
103      }
104      return bron_kerbosch_weighted(nodes, g, 0, (1LL << nodes) - 1, 0);
105    }
106
107    int main() {
108      int nodes, edges, u, v;
109      cin >> nodes >> edges;
110      for (int i = 0; i < edges; i++) {
111        cin >> u >> v;
112        adj[u][v] = adj[v][u] = true;
113      }
114      for (int i = 0; i < nodes; i++) cin >> w[i];
115      cout << "Max unweighted clique: ";
116      cout << bron_kerbosch(nodes) << "\n";
117      cout << "Max weighted clique: ";
118      cout << bron_kerbosch_weighted(nodes) << "\n";
119      return 0;
120    }
```

2.6.2   Graph Coloring

```
1    /*
2
3    2.6.2 - Backtracking - Graph Coloring
4
5    Description: Given an undirected graph, assign colors to each
6    of the vertices such that no pair of adjacent vertices have the
7    same color. Furthermore, do so using the minimum # of colors.
8
9    Complexity: Exponential on the number of vertices. The exact
10   running time is difficult to calculate due to several pruning
11   optimizations used here.
12
13   =~=~=~=~= Sample Input =~=~=~=~=
14   5 7
15   0 1
16   0 4
17   1 3
18   1 4
19   2 3
20   2 4
21   3 4
22
23   =~=~=~=~= Sample Output =~=~=~=~=
24   Colored using 3 color(s). The colorings are:
25   Color 1: 0 3
26   Color 2: 1 2
27   Color 3: 4
28
29   */
30
31   #include <algorithm> /* std::fill(), std::max() */
32   #include <iostream>
33   #include <vector>
```

```
34  using namespace std;
35
36  const int MAXN = 30;
37  int cols[MAXN], adj[MAXN][MAXN];
38  int id[MAXN + 1], deg[MAXN + 1];
39  int min_cols, best_cols[MAXN];
40
41  void dfs(int from, int to, int cur, int used_cols) {
42    if (used_cols >= min_cols) return;
43    if (cur == to) {
44      for (int i = from; i < to; i++)
45        best_cols[id[i]] = cols[i];
46      min_cols = used_cols;
47      return;
48    }
49    vector<bool> used(used_cols + 1);
50    for (int i = 0; i < cur; i++)
51      if (adj[id[cur]][id[i]]) used[cols[i]] = true;
52    for (int i = 0; i <= used_cols; i++) {
53      if (!used[i]) {
54        int tmp = cols[cur];
55        cols[cur] = i;
56        dfs(from, to, cur + 1, max(used_cols, i + 1));
57        cols[cur] = tmp;
58      }
59    }
60  }
61
62  int color_graph(int nodes) {
63    for (int i = 0; i <= nodes; i++) {
64      id[i] = i;
65      deg[i] = 0;
66    }
67    int res = 1;
68    for (int from = 0, to = 1; to <= nodes; to++) {
69      int best = to;
70      for (int i = to; i < nodes; i++) {
71        if (adj[id[to - 1]][id[i]]) deg[id[i]]++;
72        if (deg[id[best]] < deg[id[i]]) best = i;
73      }
74      int tmp = id[to];
75      id[to] = id[best];
76      id[best] = tmp;
77      if (deg[id[to]] == 0) {
78        min_cols = nodes + 1;
79        fill(cols, cols + nodes, 0);
80        dfs(from, to, from, 0);
81        from = to;
82        res = max(res, min_cols);
83      }
84    }
85    return res;
86  }
87
88  int main() {
89    int nodes, edges, u, v;
90    cin >> nodes >> edges;
91    for (int i = 0; i < edges; i++) {
92      cin >> u >> v;
```

```
 93       adj[u][v] = adj[v][u] = true;
 94     }
 95     cout << "Colored␣using␣" << color_graph(nodes);
 96     cout << "␣color(s).␣The␣colorings␣are:\n";
 97     for (int i = 0; i < min_cols; i++) {
 98       cout << "Color␣" << i + 1 << ":";
 99       for (int j = 0; j < nodes; j++)
100         if (best_cols[j] == i) cout << "␣" << j;
101       cout << "\n";
102     }
103     return 0;
104   }
```

2.7   Maximum Matching

2.7.1   Maximum Bipartite Matching (Kuhn's Algorithm)

```
 1   /*
 2
 3   2.7.1 - Maximum Bipartite Matching (Kuhn's Algorithm)
 4
 5   Description: Given two sets of vertices A = {0, 1, ..., n1}
 6   and B = {0, 1, ..., n2} as well as a set of edges E mapping
 7   nodes from set A to set B, determine the largest possible
 8   subset of E such that no pair of edges in the subset share
 9   a common vertex. Precondition: n2 >= n1.
10
11   Complexity: O(V*E) on the number of vertices and edges.
12
13   =~=~=~=~= Sample Input =~=~=~=~=
14   3 4 6
15   0 1
16   1 0
17   1 1
18   1 2
19   2 2
20   2 3
21
22   =~=~=~=~= Sample Output =~=~=~=~=
23   Matched 3 pairs. Matchings are:
24   1 0
25   0 1
26   2 2
27
28   */
29
30   #include <algorithm> /* std::fill() */
31   #include <iostream>
32   #include <vector>
33   using namespace std;
34
35   const int MAXN = 100;
36   int match[MAXN];
37   vector<bool> vis(MAXN);
38   vector<int> adj[MAXN];
39
```

```
40   bool dfs(int u) {
41     vis[u] = true;
42     for (int j = 0; j < (int)adj[u].size(); j++) {
43       int v = match[adj[u][j]];
44       if (v == -1 || (!vis[v] && dfs(v))) {
45         match[adj[u][j]] = u;
46         return true;
47       }
48     }
49     return false;
50   }
51
52   int kuhn(int n1, int n2) {
53     fill(vis.begin(), vis.end(), false);
54     fill(match, match + n2, -1);
55     int matches = 0;
56     for (int i = 0; i < n1; i++) {
57       for (int j = 0; j < n1; j++) vis[j] = 0;
58       if (dfs(i)) matches++;
59     }
60     return matches;
61   }
62
63   int main() {
64     int n1, n2, edges, u, v;
65     cin >> n1 >> n2 >> edges;
66     for (int i = 0; i < edges; i++) {
67       cin >> u >> v;
68       adj[u].push_back(v);
69     }
70     cout << "Matched " << kuhn(n1, n2);
71     cout << " pair(s). Matchings are:\n";
72     for (int i = 0; i < n2; i++) {
73       if (match[i] == -1) continue;
74       cout << match[i] << " " << i << "\n";
75     }
76     return 0;
77   }
```

2.7.2   Maximum Bipartite Matching (Hopcroft-Karp Algorithm)

```
1    /*
2
3    2.7.2 - Maximum Bipartite Matching (Hopcroft-Karp Algorithm)
4
5    Description: Given two sets of vertices A = {0, 1, ..., n1}
6    and B = {0, 1, ..., n2} as well as a set of edges E mapping
7    nodes from set A to set B, determine the largest possible
8    subset of E such that no pair of edges in the subset share
9    a common vertex. Precondition: n2 >= n1.
10
11   Complexity: O(E sqrt V) on the number of edges and vertices.
12
13   =~=~=~=~= Sample Input =~=~=~=~=
14   3 4 6
15   0 1
16   1 0
```

```
17   1 1
18   1 2
19   2 2
20   2 3
21
22   =~=~=~=~= Sample Output =~=~=~=~=
23   Matched 3 pairs. Matchings are:
24   1 0
25   0 1
26   2 2
27
28   */
29
30   #include <algorithm> /* std::fill() */
31   #include <iostream>
32   #include <vector>
33   using namespace std;
34
35   const int MAXN = 100;
36   int match[MAXN], dist[MAXN];
37   vector<bool> used(MAXN), vis(MAXN);
38   vector<int> adj[MAXN];
39
40   void bfs(int n1, int n2) {
41     fill(dist, dist + n1, -1);
42     int q[n2], qb = 0;
43     for (int u = 0; u < n1; ++u) {
44       if (!used[u]) {
45         q[qb++] = u;
46         dist[u] = 0;
47       }
48     }
49     for (int i = 0; i < qb; i++) {
50       int u = q[i];
51       for (int j = 0; j < (int)adj[u].size(); j++) {
52         int v = match[adj[u][j]];
53         if (v >= 0 && dist[v] < 0) {
54           dist[v] = dist[u] + 1;
55           q[qb++] = v;
56         }
57       }
58     }
59   }
60
61   bool dfs(int u) {
62     vis[u] = true;
63     for (int j = 0; j < (int)adj[u].size(); j++) {
64       int v = match[adj[u][j]];
65       if (v < 0 || (!vis[v] && dist[v] == dist[u] + 1 && dfs(v))) {
66         match[adj[u][j]] = u;
67         used[u] = true;
68         return true;
69       }
70     }
71     return false;
72   }
73
74   int hopcroft_karp(int n1, int n2) {
75     fill(match, match + n2, -1);
```

```
76    fill(used.begin(), used.end(), false);
77    int res = 0;
78    for (;;) {
79      bfs(n1, n2);
80      fill(vis.begin(), vis.end(), false);
81      int f = 0;
82      for (int u = 0; u < n1; ++u)
83        if (!used[u] && dfs(u)) f++;
84      if (!f) return res;
85      res += f;
86    }
87    return res;
88  }
89
90  int main() {
91    int n1, n2, edges, u, v;
92    cin >> n1 >> n2 >> edges;
93    for (int i = 0; i < edges; i++) {
94      cin >> u >> v;
95      adj[u].push_back(v);
96    }
97    cout << "Matched␣" << hopcroft_karp(n1, n2);
98    cout << "␣pair(s).␣Matchings␣are:\n";
99    for (int i = 0; i < n2; i++) {
100     if (match[i] == -1) continue;
101     cout << match[i] << "␣" << i << "\n";
102   }
103   return 0;
104 }
```

2.7.3   Maximum Graph Matching (Edmonds's Algorithm)

```
1   /*
2
3   2.7.3 - Maximum Matching for General Graphs (Edmonds's Algorithm)
4
5   Description: Given a general directed graph, determine a maximal
6   subset of the edges such that no vertex is repeated in the subset.
7
8   Complexity: O(V^3) on the number of vertices.
9
10  =~=~=~=~= Sample Input =~=~=~=~=
11  4 8
12  0 1
13  1 0
14  1 2
15  2 1
16  2 3
17  3 2
18  3 0
19  0 3
20
21  =~=~=~=~= Sample Output =~=~=~=~=
22  Matched 2 pair(s). Matchings are:
23  0 1
24  2 3
25
```

```cpp
26  */
27
28  #include <iostream>
29  #include <vector>
30  using namespace std;
31
32  const int MAXN = 100;
33  int p[MAXN], base[MAXN], match[MAXN];
34  vector<int> adj[MAXN];
35
36  int lca(int nodes, int a, int b) {
37    vector<bool> used(nodes);
38    for (;;) {
39      a = base[a];
40      used[a] = true;
41      if (match[a] == -1) break;
42      a = p[match[a]];
43    }
44    for (;;) {
45      b = base[b];
46      if (used[b]) return b;
47      b = p[match[b]];
48    }
49  }
50
51  void mark_path(vector<bool> & blossom, int v, int b, int children) {
52    for (; base[v] != b; v = p[match[v]]) {
53      blossom[base[v]] = blossom[base[match[v]]] = true;
54      p[v] = children;
55      children = match[v];
56    }
57  }
58
59  int find_path(int nodes, int root) {
60    vector<bool> used(nodes);
61    for (int i = 0; i < nodes; ++i) {
62      p[i] = -1;
63      base[i] = i;
64    }
65    used[root] = true;
66    int q[nodes], qh = 0, qt = 0;
67    q[qt++] = root;
68    while (qh < qt) {
69      int v = q[qh++];
70      for (int j = 0, to; j < (int)adj[v].size(); j++) {
71        to = adj[v][j];
72        if (base[v] == base[to] || match[v] == to) continue;
73        if (to == root || (match[to] != -1 && p[match[to]] != -1)) {
74          int curbase = lca(nodes, v, to);
75          vector<bool> blossom(nodes);
76          mark_path(blossom, v, curbase, to);
77          mark_path(blossom, to, curbase, v);
78          for (int i = 0; i < nodes; i++)
79            if (blossom[base[i]]) {
80              base[i] = curbase;
81              if (!used[i]) {
82                used[i] = true;
83                q[qt++] = i;
84              }
```

```
 85            }
 86        } else if (p[to] == -1) {
 87          p[to] = v;
 88          if (match[to] == -1) return to;
 89          to = match[to];
 90          used[to] = true;
 91          q[qt++] = to;
 92        }
 93      }
 94    }
 95    return -1;
 96  }
 97
 98  int edmonds(int nodes) {
 99    for (int i = 0; i < nodes; i++) match[i] = -1;
100    for (int i = 0; i < nodes; i++) {
101      if (match[i] == -1) {
102        int v, pv, ppv;
103        for (v = find_path(nodes, i); v != -1; v = ppv) {
104          ppv = match[pv = p[v]];
105          match[v] = pv;
106          match[pv] = v;
107        }
108      }
109    }
110    int matches = 0;
111    for (int i = 0; i < nodes; i++)
112      if (match[i] != -1) matches++;
113    return matches / 2;
114  }
115
116  int main() {
117    int nodes, edges, u, v;
118    cin >> nodes >> edges;
119    for (int i = 0; i < edges; i++) {
120      cin >> u >> v;
121      adj[u].push_back(v);
122    }
123    cout << "Matched␣" << edmonds(nodes);
124    cout << "␣pair(s).␣Matchings␣are:\n";
125    for (int i = 0; i < nodes; i++) {
126      if (match[i] != -1 && i < match[i])
127        cout << i << "␣" << match[i] << "\n";
128    }
129    return 0;
130  }
```

2.8   Hamiltonian Path and Cycle

2.8.1   Shortest Hamiltonian Cycle (Travelling Salesman)

```
1  /*
2
3  2.8.1 - Shortest Hamiltonian Cycle (TSP)
4
5  Description: Given a weighted, directed graph, the shortest
```

```
 6   hamiltonian cycle is a cycle of minimum distance that visits
 7   each vertex exactly once and returns to the original vertex.
 8   This is also known as the traveling salesman problem (TSP).
 9   Since this is a bitmasking solution with 32-bit integers,
10   the number of vertices must be less than 32.
11
12   Complexity: O(2^V * V^2) on the number of vertices.
13
14   =~=~=~=~= Sample Input =~=~=~=~=
15   5 10
16   0 1 1
17   0 2 10
18   0 3 1
19   0 4 10
20   1 2 10
21   1 3 10
22   1 4 1
23   2 3 1
24   2 4 1
25   3 4 10
26
27   =~=~=~=~= Sample Output =~=~=~=~=
28   The shortest hamiltonian cycle has length 5.
29   Take the path: 0->3->2->4->1->0
30
31   */
32
33   #include <algorithm> /* std::fill(), std::min() */
34   #include <iostream>
35   using namespace std;
36
37   const int MAXN = 20, INF = 0x3f3f3f3f;
38   int adj[MAXN][MAXN], order[MAXN];
39
40   int shortest_hamiltonian_cycle(int nodes) {
41     int dp[1 << nodes][nodes];
42     for (int i = 0; i < (1 << nodes); i++)
43       fill(dp[i], dp[i] + nodes, INF);
44     dp[1][0] = 0;
45     for (int mask = 1; mask < (1 << nodes); mask += 2) {
46       for (int i = 1; i < nodes; i++)
47         if ((mask & 1 << i) != 0)
48           for (int j = 0; j < nodes; j++)
49             if ((mask & 1 << j) != 0)
50               dp[mask][i] = min(dp[mask][i], dp[mask ^ (1 << i)][j] + adj[j][i]);
51     }
52     int res = INF + INF;
53     for (int i = 1; i < nodes; i++)
54       res = min(res, dp[(1 << nodes) - 1][i] + adj[i][0]);
55     int cur = (1 << nodes) - 1, last = 0;
56     for (int i = nodes - 1; i >= 1; i--) {
57       int bj = -1;
58       for (int j = 1; j < nodes; j++) {
59         if ((cur & 1 << j) != 0 && (bj == -1 ||
60             dp[cur][bj] + adj[bj][last] > dp[cur][j] + adj[j][last])) {
61           bj = j;
62         }
63       }
64       order[i] = bj;
```

```
65        cur ^= 1 << bj;
66        last = bj;
67      }
68      return res;
69    }
70
71    int main() {
72      int nodes, edges, u, v, w;
73      cin >> nodes >> edges;
74      for (int i = 0; i < edges; i++) {
75        cin >> u >> v >> w;
76        adj[u][v] = adj[v][u] = w; //only set adj[u][v] if directed edges
77      }
78      cout << "The shortest hamiltonian cycle has length ";
79      cout << shortest_hamiltonian_cycle(nodes) << ".\n";
80      cout << "Take the path: ";
81      for (int i = 0; i < nodes; i++) cout << order[i] << "->";
82      cout << order[0] << "\n";
83      return 0;
84    }
```

2.8.2   Shortest Hamiltonian Path

```
1    /*
2
3    2.8.2 - Shortest Hamiltonian Path
4
5    Description: Given a weighted, directed graph, the shortest
6    hamiltonian path is a path of minimum distance that visits
7    each vertex exactly once. Unlike the travelling salesman
8    problem, we don't have to return to the starting vertex.
9    Since this is a bitmasking solution with 32-bit integers,
10   the number of vertices must be less than 32.
11
12   Complexity: O(2^V * V^2) on the number of vertices.
13
14   =~=~=~=~= Sample Input =~=~=~=~=
15   3 6
16   0 1 1
17   0 2 1
18   1 0 7
19   1 2 2
20   2 0 3
21   2 1 5
22
23   =~=~=~=~= Sample Output =~=~=~=~=
24   The shortest hamiltonian path has length 3.
25   Take the path: 0->1->2
26
27   */
28
29   #include <algorithm> /* std::fill(), std::min() */
30   #include <iostream>
31   using namespace std;
32
33   const int MAXN = 20, INF = 0x3f3f3f3f;
34
```

```cpp
35  int adj[MAXN][MAXN], order[MAXN];
36
37  int shortest_hamiltonian_path(int nodes) {
38    int dp[1 << nodes][nodes];
39    for (int i = 0; i < (1 << nodes); i++)
40      fill(dp[i], dp[i] + nodes, INF);
41    for (int i = 0; i < nodes; i++) dp[1 << i][i] = 0;
42    for (int mask = 1; mask < (1 << nodes); mask += 2) {
43      for (int i = 0; i < nodes; i++)
44        if ((mask & 1 << i) != 0)
45          for (int j = 0; j < nodes; j++)
46            if ((mask & 1 << j) != 0)
47              dp[mask][i] = min(dp[mask][i], dp[mask ^ (1 << i)][j] + adj[j][i]);
48    }
49    int res = INF + INF;
50    for (int i = 1; i < nodes; i++)
51      res = min(res, dp[(1 << nodes) - 1][i]);
52    int cur = (1 << nodes) - 1, last = -1;
53    for (int i = nodes - 1; i >= 0; i--) {
54      int bj = -1;
55      for (int j = 0; j < nodes; j++) {
56        if ((cur & 1 << j) != 0 && (bj == -1 ||
57            dp[cur][bj] + (last == -1 ? 0 : adj[bj][last]) >
58            dp[cur][j]  + (last == -1 ? 0 : 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] = w;
75    }
76    cout << "The shortest hamiltonian path has length ";
77    cout << shortest_hamiltonian_path(nodes) << ".\n";
78    cout << "Take the path: " << order[0];
79    for (int i = 1; i < nodes; i++) cout << "->" << order[i];
80    return 0;
81  }
```

Chapter 3

Data Structures

3.1 Disjoint Sets

3.1.1 Disjoint Set Forest (Simple)

```
1  /*
2
3  3.1.1 - Disjoint Set Forest (Simple)
4
5  Description: This data structure dynamically keeps track
6  of items partitioned into non-overlapping sets (a disjoint
7  set forest). It is also known as a union-find data structure.
8
9  Time Complexity: Every function below is O(a(N)) amortized
10 on the number of items in the set due to the optimizations
11 of union by rank and path compression. Here, a(N) is the
12 extremely slow growing inverse of the Ackermann function.
13 For all practical values of n, a(n) is less than 5.
14
15 Space Complexity: O(N) total.
16
17 */
18
19 const int MAXN = 1000;
20 int num_sets = 0, root[MAXN+1], rank[MAXN+1];
21
22 int find_root(int x) {
23   if (root[x] != x) root[x] = find_root(root[x]);
24   return root[x];
25 }
26
27 void make_set(int x) {
28   root[x] = x;
29   rank[x] = 0;
30   num_sets++;
31 }
32
33 bool is_united(int x, int y) {
34   return find_root(x) == find_root(y);
35 }
```

```
36
37   void unite(int x, int y) {
38     int X = find_root(x), Y = find_root(y);
39     if (X == Y) return;
40     num_sets--;
41     if (rank[X] < rank[Y]) root[X] = Y;
42     else if (rank[X] > rank[Y]) root[Y] = X;
43     else rank[root[Y] = X]++;
44   }
45
46   /*** Example Usage ***/
47
48   #include <cassert>
49   #include <iostream>
50   using namespace std;
51
52   int main() {
53     for (char c = 'a'; c <= 'g'; c++) make_set(c);
54     unite('a', 'b');
55     unite('b', 'f');
56     unite('d', 'e');
57     unite('e', 'g');
58     assert(num_sets == 3);
59     assert(is_united('a', 'b'));
60     assert(!is_united('a', 'c'));
61     assert(!is_united('b', 'g'));
62     assert(is_united('d', 'g'));
63     return 0;
64   }
```

3.1.2   Disjoint Set Forest

```
1   /*
2
3   3.1.2 - Disjoint Set Forest
4
5   Description: This data structure dynamically keeps track
6   of items partitioned into non-overlapping sets (a disjoint
7   set forest). It is also known as a union-find data structure.
8   This particular templatized version employs an std::map for
9   built in storage and coordinate compression. That is, the
10  magnitude of values inserted is not limited.
11
12  Time Complexity: make_set(), unite() and is_united() are
13  O(a(N) + log N) = O(log N) on the number of elements in the
14  disjoint set forest. get_all_sets() is O(N). find() is is
15  O(a(N)) amortized on the number of items in the set due to
16  the optimizations of union by rank and path compression.
17  Here, a(N) is the extremely slow growing inverse of the
18  Ackermann function. For all practical values of n, a(n) is
19  less than 5.
20
21  Space Complexity: O(N) storage and auxiliary.
22
23  =~=~=~=~= Sample Output =~=~=~=~=
24  Elements: 7, Sets: 3
25  [[a,b,f],[c],[d,e,g]]
```

```
26
27  */
28
29  #include <map>
30  #include <vector>
31
32  template<class T> class disjoint_set_forest {
33    int num_elements, num_sets;
34    std::map<T, int> ID;
35    std::vector<int> root, rank;
36
37    int find_root(int x) {
38      if (root[x] != x) root[x] = find_root(root[x]);
39      return root[x];
40    }
41
42   public:
43    disjoint_set_forest(): num_elements(0), num_sets(0) {}
44    int elements() { return num_elements; }
45    int sets() { return num_sets; }
46
47    bool is_united(const T & x, const T & y) {
48      return find_root(ID[x]) == find_root(ID[y]);
49    }
50
51    void make_set(const T & x) {
52      if (ID.find(x) != ID.end()) return;
53      root.push_back(ID[x] = num_elements++);
54      rank.push_back(0);
55      num_sets++;
56    }
57
58    void unite(const T & x, const T & y) {
59      int X = find_root(ID[x]), Y = find_root(ID[y]);
60      if (X == Y) return;
61      num_sets--;
62      if (rank[X] < rank[Y]) root[X] = Y;
63      else if (rank[X] > rank[Y]) root[Y] = X;
64      else rank[root[Y] = X]++;
65    }
66
67    std::vector<std::vector<T> > get_all_sets() {
68      std::map<int, std::vector<T> > tmp;
69      for (typename std::map<T, int>::iterator
70          it = ID.begin(); it != ID.end(); it++)
71        tmp[find_root(it->second)].push_back(it->first);
72      std::vector<std::vector<T> > ret;
73      for (typename std::map<int, std::vector<T> >::
74          iterator it = tmp.begin(); it != tmp.end(); it++)
75        ret.push_back(it->second);
76      return ret;
77    }
78  };
79
80  /*** Example Usage ***/
81
82  #include <iostream>
83  using namespace std;
84
```

```
85  int main() {
86    disjoint_set_forest<char> d;
87    for (char c = 'a'; c <= 'g'; c++) d.make_set(c);
88    d.unite('a', 'b');
89    d.unite('b', 'f');
90    d.unite('d', 'e');
91    d.unite('e', 'g');
92    cout << "Elements:␣" << d.elements();
93    cout << ",␣Sets:␣" << d.sets() << endl;
94    vector<vector<char> > s = d.get_all_sets();
95    cout << "[";
96    for (int i = 0; i < (int)s.size(); i++) {
97      cout << (i > 0 ? ",[" : "[");
98      for (int j = 0; j < (int)s[i].size(); j++)
99        cout << (j > 0 ? "," : "") << s[i][j];
100     cout << "]";
101   }
102   cout << "]\n";
103   return 0;
104 }
```

3.2   Fenwick Trees

3.2.1   Simple Fenwick Tree

```
1   /*
2
3   3.2.1 - Fenwick Tree (Simple)
4
5   Description: A Fenwick tree (a.k.a. binary indexed tree) is a
6   data structure that allows for the sum of an arbitrary range
7   of values in an array to be dynamically queried in logarithmic
8   time. Note that unlike the object-oriented versions of this
9   data structure found in later sections, the operations here
10  work on 1-based indices (i.e. between 1 and MAXN, inclusive).
11  The array a[] is always synchronized with the bit[] array and
12  should not be modified outside of the functions below.
13
14  Time Complexity: All functions are O(log MAXN).
15  Space Complexity: O(MAXN) storage and auxiliary.
16
17  */
18
19  const int MAXN = 1000;
20  int a[MAXN + 1], bit[MAXN + 1];
21
22  //a[i] += v
23  void add(int i, int v) {
24    a[i] += v;
25    for (; i <= MAXN; i += i & -i)
26      bit[i] += v;
27  }
28
29  //a[i] = v
30  void set(int i, int v) {
31    int inc = v - a[i];
```

```
32      add(i, inc);
33    }
34
35    //returns sum(a[i] for i = 1..hi inclusive)
36    int sum(int hi) {
37      int ret = 0;
38      for (; hi > 0; hi -= hi & -hi)
39        ret += bit[hi];
40      return ret;
41    }
42
43    //returns sum(a[i] for i = lo..hi inclusive)
44    int sum(int lo, int hi) {
45      return sum(hi) - sum(lo - 1);
46    }
47
48    /*** Example Usage ***/
49
50    #include <iostream>
51    using namespace std;
52
53    int main() {
54      for (int i = 1; i <= 5; i++) set(i, i);
55      add(4, -5);
56      cout << "BIT_values:_";
57      for (int i = 1; i <= 5; i++)
58        cout << a[i] << "_"; //1 2 3 -1 5
59      cout << "\nSum_of_range_[1,3]_is_";
60      cout << sum(1, 3) << ".\n"; //6
61      return 0;
62    }
```

3.2.2   Fenwick Tree

```
1    /*
2
3    3.2.2 - Fenwick Tree (Point Update, Range Query)
4
5    Description: A Fenwick tree (a.k.a. binary indexed tree) is a
6    data structure that allows for the sum of an arbitrary range
7    of values in an array to be dynamically queried in logarithmic
8    time. All methods below work on 0-based indices (i.e. indices
9    in the range from 0 to size() - 1, inclusive, are valid).
10
11   Time Complexity: add(), set(), and sum() are all O(log N) on
12   the length of the array. size() and at() are O(1).
13
14   Space Complexity: O(N) storage and O(N) auxiliary on size().
15
16   */
17
18   #include <vector>
19
20   template<class T> class fenwick_tree {
21     int len;
22     std::vector<int> a, bit;
23
```

```cpp
24   public:
25     fenwick_tree(int n): len(n),
26      a(n + 1), bit(n + 1) {}
27
28     //a[i] += v
29     void add(int i, const T & v) {
30       a[++i] += v;
31       for (; i <= len; i += i & -i)
32         bit[i] += v;
33     }
34
35     //a[i] = v
36     void set(int i, const T & v) {
37       T inc = v - a[i + 1];
38       add(i, inc);
39     }
40
41     //returns sum(a[i] for i = 1..hi inclusive)
42     T sum(int hi) {
43       T res = 0;
44       for (hi++; hi > 0; hi -= hi & -hi)
45         res += bit[hi];
46       return res;
47     }
48
49     //returns sum(a[i] for i = lo..hi inclusive)
50     T sum(int lo, int hi) {
51       return sum(hi) - sum(lo - 1);
52     }
53
54     inline int size() { return len; }
55     inline T at(int i) { return a[i + 1]; }
56   };
57
58   /*** Example Usage ***/
59
60   #include <iostream>
61   using namespace std;
62
63   int main() {
64     int a[] = {10, 1, 2, 3, 4};
65     fenwick_tree<int> t(5);
66     for (int i = 0; i < 5; i++) t.set(i, a[i]);
67     t.add(0, -5);
68     cout << "BIT values: ";
69     for (int i = 0; i < t.size(); i++)
70       cout << t.at(i) << " "; //5 1 2 3 4
71     cout << "\nSum of range [1, 3] is ";
72     cout << t.sum(1, 3) << ".\n"; //6
73     return 0;
74   }
```

3.2.3   Fenwick Tree (Point Query)

```
1   /*
2
3   3.2.3 - Fenwick Tree (Range Update, Point Query)
```

```
4
5    Description: A Fenwick tree (a.k.a. binary indexed tree) is a
6    data structure that allows for the sum of an arbitrary range
7    of values in an array to be dynamically queried in logarithmic
8    time. Range updating in a Fenwick tree can only increment
9    values in a range, not set them all to the same value. This
10   version is a very concise version if only point queries are
11   needed. The functions below work on 1-based indices (between
12   1 and MAXN, inclusive).
13
14   Time Complexity: add() and at() are O(log MAXN).
15   Space Complexity: O(N).
16
17   */
18
19   const int MAXN = 1000;
20   int bit[MAXN + 1];
21
22   //a[i] += v
23   void add(int i, int v) {
24     for (i++; i <= MAXN; i += i & -i) bit[i] += v;
25   }
26
27   //a[i] += v for i = lo..hi, inclusive
28   void add(int lo, int hi, int v) {
29     add(lo, v);
30     add(hi + 1, -v);
31   }
32
33   //returns a[i]
34   int at(int i) {
35     int sum = 0;
36     for (i++; i > 0; i -= i & -i) sum += bit[i];
37     return sum;
38   }
39
40   /*** Example Usage ***/
41
42   #include <iostream>
43   using namespace std;
44
45   int main() {
46     add(1, 2, 5);
47     add(2, 3, 5);
48     add(3, 5, 10);
49     cout << "BIT values: "; //5 10 15 10 10
50     for (int i = 1; i <= 5; i++)
51       cout << at(i) << " ";
52     cout << "\n";
53     return 0;
54   }
```

3.2.4   Fenwick Tree (Range Update)

```
1    /*
2
3    3.2.4 - Fenwick Tree (Range Update/Query)
```

```
4
5    Description: Using two arrays, a Fenwick tree can be made to
6    support range updates and range queries simultaneously. However,
7    the range updates can only be used to add an increment to all
8    values in a range, not set them to the same value. The latter
9    problem may be solved using a segment tree + lazy propagation.
10   All methods below operate 0-based indices (i.e. indices in the
11   range from 0 to size() - 1, inclusive, are valid).
12
13   Time Complexity: add(), set(), at(), and sum() are all O(log N)
14   on the length of the array. size() is O(1).
15
16   Space Complexity: O(N) storage and auxiliary.
17
18   =~=~=~=~= Sample Output =~=~=~=~=
19   BIT values: 15 6 7 -5 4
20   Sum of range [0, 4] is 27.
21
22   */
23
24   #include <vector>
25
26   template<class T> class fenwick_tree {
27     int len;
28     std::vector<T> b1, b2;
29
30     T sum(const std::vector<T> & b, int i) {
31       T res = 0;
32       for (; i != 0; i -= i & -i) res += b[i];
33       return res;
34     }
35
36     void add(std::vector<T> & b, int i, const T & v) {
37       for (; i <= len; i += i & -i) b[i] += v;
38     }
39
40    public:
41     fenwick_tree(int n):
42       len(n + 1), b1(n + 2), b2(n + 2) {}
43
44     //a[i] += v for i = lo..hi, inclusive
45     void add(int lo, int hi, const T & v) {
46       lo++, hi++;
47       add(b1, lo, v);
48       add(b1, hi + 1, -v);
49       add(b2, lo, v * (lo - 1));
50       add(b2, hi + 1, -v * hi);
51     }
52
53     //a[i] = v
54     void set(int i, const T & v) { add(i, i, v - at(i)); }
55
56     //returns sum(a[i] for i = 1..hi inclusive)
57     T sum(int hi) { return sum(b1, hi)*hi - sum(b2, hi); }
58
59     //returns sum(a[i] for i = lo..hi inclusive)
60     T sum(int lo, int hi) { return sum(hi + 1) - sum(lo); }
61
62     inline int size() const { return len - 1; }
```

```
63    inline T at(int i) { return sum(i, i); }
64  };
65
66  /*** Example Usage ***/
67
68  #include <iostream>
69  using namespace std;
70
71  int main() {
72    int a[] = {10, 1, 2, 3, 4};
73    fenwick_tree<int> t(5);
74    for (int i = 0; i < 5; i++) t.set(i, a[i]);
75    t.add(0, 2, 5); //15 6 7 3 4
76    t.set(3, -5);   //15 6 7 -5 4
77    cout << "BIT values: ";
78    for (int i = 0; i < t.size(); i++)
79      cout << t.at(i) << " ";
80    cout << "\nSum of range [0, 4] is ";
81    cout << t.sum(0, 4) << ".\n"; //27
82    return 0;
83  }
```

3.2.5   Fenwick Tree (Map)

```
1   /*
2
3   3.2.5 - Fenwick Tree (Range Updates and Range Query
4                         with Co-ordinate Compression)
5
6   Description: Using two std::maps to represent the Fenwick tree,
7   there no longer needs to be a restriction on the magnitude of
8   queried indices. All indices in range [0, MAXN] are valid.
9
10  Time Complexity: All functions are O(log^2 MAXN). If the
11  std::map is replaced with an std::unordered_map, then the
12  running time will become O(log MAXN) amortized.
13
14  Space Complexity: O(n) on the number of indices accessed.
15
16  */
17
18  #include <map>
19
20  const int MAXN = 1000000000;
21  std::map<int, int> tmul, tadd;
22
23  void _add(int at, int mul, int add) {
24    for (int i = at; i <= MAXN; i = (i | (i+1))) {
25      tmul[i] += mul;
26      tadd[i] += add;
27    }
28  }
29
30  //a[i] += v for all i = lo..hi, inclusive
31  void add(int lo, int hi, int v) {
32    _add(lo, v, -v * (lo - 1));
33    _add(hi, -v, v * hi);
```

```
34  }
35
36  //returns sum(a[i] for i = 1..hi inclusive)
37  int sum(int hi) {
38    int mul = 0, add = 0, start = hi;
39    for (int i = hi; i >= 0; i = (i & (i + 1)) - 1) {
40      if (tmul.find(i) != tmul.end())
41        mul += tmul[i];
42      if (tadd.find(i) != tadd.end())
43        add += tadd[i];
44    }
45    return mul*start + add;
46  }
47
48  //returns sum(a[i] for i = lo..hi inclusive)
49  int sum(int lo, int hi) {
50    return sum(hi) - sum(lo - 1);
51  }
52
53  //a[i] = v
54  void set(int i, int v) {
55    add(i, i, v - sum(i, i));
56  }
57
58  /*** Example Usage ***/
59
60  #include <iostream>
61  using namespace std;
62
63  int main() {
64    add(500000001, 500000010, 3);
65    add(500000011, 500000015, 5);
66    set(500000000, 10);
67    cout << sum(500000000, 500000015) << "\n"; //65
68    return 0;
69  }
```

3.2.6   2D Fenwick Tree

```
1   /*
2
3   3.2.6 - 2D Fenwick Tree (Point Update, Range Query)
4
5   Description: A 2D Fenwick tree is abstractly a 2D array which also
6   supports efficient queries for the sum of values in the rectangle
7   with top-left (1, 1) and bottom-right (r, c). The implementation
8   below has indices accessible in the range [1...xmax][1...ymax].
9
10  Time Complexity: All functions are O(log(xmax)*log(ymax)).
11  Space Complexity: O(xmax*ymax) storage and auxiliary.
12
13  */
14
15  const int xmax = 100, ymax = 100;
16
17  int a[xmax+1][ymax+1], bit[xmax+1][ymax+1];
18
```

```
19   //a[x][y] += v
20   void add(int x, int y, int v) {
21     a[x][y] += v;
22     for (int i = x; i <= xmax; i += i & -i)
23       for (int j = y; j <= ymax; j += j & -j)
24         bit[i][j] += v;
25   }
26
27    //a[x][y] = v
28   void set(int x, int y, int v) {
29     int inc = v - a[x][y];
30     add(x, y, inc);
31   }
32
33   //returns sum(data[1..x][1..y], all inclusive)
34   int sum(int x, int y) {
35     int ret = 0;
36     for (int i = x; i > 0; i -= i & -i)
37       for (int j = y; j > 0; j -= j & -j)
38         ret += bit[i][j];
39     return ret;
40   }
41
42   //returns sum(data[x1..x2][y1..y2], all inclusive)
43   int sum(int x1, int y1, int x2, int y2) {
44     return sum(x2, y2) + sum(x1 - 1, y1 - 1) -
45            sum(x1 - 1, y2) - sum(x2, y1 - 1);
46   }
47
48   /*** Example Usage ***/
49
50   #include <cassert>
51   #include <iostream>
52   using namespace std;
53
54   int main() {
55     set(1, 1, 5);
56     set(1, 2, 6);
57     set(2, 1, 7);
58     add(3, 3, 9);
59     add(2, 1, -4);
60   /*
61     5 6 0
62     3 0 0
63     0 0 9
64   */
65     cout << "2D BIT values:\n";
66     for (int i = 1; i <= 3; i++) {
67       for (int j = 1; j <= 3; j++)
68         cout << a[i][j] << " ";
69       cout << "\n";
70     }
71     assert(sum(1, 1, 1, 2) == 11);
72     assert(sum(1, 1, 2, 1) == 8);
73     assert(sum(1, 1, 3, 3) == 23);
74     return 0;
75   }
```

3.2.7   2D Fenwick Tree (Range Update)

```
1   /*
2
3   3.2.7 - 2D Fenwick Tree (Range Update, Range Query,
4                            with Coordinate Compression)
5
6   Description: A 2D Fenwick tree is abstractly a 2D array which also
7   supports efficient queries for the sum of values in the rectangle
8   with top-left (1, 1) and bottom-right (r, c). The implementation
9   below has indices accessible in the range [0..xmax][0...ymax].
10
11  Time Complexity: All functions are O(log(xmax)*log(ymax)*log(N))
12  where N is the number of indices operated on so far. Use an array
13  or an unordered_map instead of a map to remove the log(N) factor.
14
15  Space Complexity: O(xmax*ymax) storage and auxiliary.
16
17  */
18
19  #include <map>
20  #include <utility>
21
22  template<class T> class fenwick_tree_2d {
23    static const int xmax = 1000000000;
24    static const int ymax = 1000000000;
25
26    std::map<std::pair<int, int>, T> t1, t2, t3, t4;
27
28    template<class Tree>
29    void add(Tree & t, int x, int y, const T & v) {
30      for (int i = x; i <= xmax; i += i & -i)
31        for (int j = y; j <= ymax; j += j & -j)
32          t[std::make_pair(i, j)] += v;
33    }
34
35    //a[i][j] += v for i = [1,x], j = [1,y]
36    void add_pre(int x, int y, const T & v) {
37      add(t1, 1, 1, v);
38
39      add(t1, 1, y + 1, -v);
40      add(t2, 1, y + 1, v * y);
41
42      add(t1, x + 1, 1, -v);
43      add(t3, x + 1, 1, v * x);
44
45      add(t1, x + 1, y + 1, v);
46      add(t2, x + 1, y + 1, -v * y);
47      add(t3, x + 1, y + 1, -v * x);
48      add(t4, x + 1, y + 1, v * x * y);
49    }
50
51   public:
52    //a[i][j] += v for i = [x1,x2], j = [y1,y2]
53    void add(int x1, int y1, int x2, int y2, const T & v) {
54      x1++; y1++; x2++; y2++;
55      add_pre(x2, y2, v);
56      add_pre(x1 - 1, y2, -v);
```

```
57        add_pre(x2, y1 - 1, -v);
58        add_pre(x1 - 1, y1 - 1, v);
59      }
60
61      //a[x][y] += v
62      void add(int x, int y, const T & v) {
63        add(x, y, x, y, v);
64      }
65
66      //a[x][y] = v
67      void set(int x, int y, const T & v) {
68        add(x, y, v - at(x, y));
69      }
70
71      //returns sum(a[i][j] for i = [1,x], j = [1,y])
72      T sum(int x, int y) {
73        x++; y++;
74        T s1 = 0, s2 = 0, s3 = 0, s4 = 0;
75        for (int i = x; i > 0; i -= i & -i)
76          for (int j = y; j > 0; j -= j & -j) {
77            s1 += t1[std::make_pair(i, j)];
78            s2 += t2[std::make_pair(i, j)];
79            s3 += t3[std::make_pair(i, j)];
80            s4 += t4[std::make_pair(i, j)];
81          }
82        return s1 * x * y + s2 * x + s3 * y + s4;
83      }
84
85      //returns sum(a[i][j] for i = [x1,x2], j = [y1,y2])
86      T sum(int x1, int y1, int x2, int y2) {
87        return sum(x2, y2) + sum(x1 - 1, y1 - 1) -
88               sum(x1 - 1, y2) - sum(x2, y1 - 1);
89      }
90
91      T at(int x, int y) { return sum(x, y, x, y); }
92    };
93
94    /*** Example Usage ***/
95
96    #include <cassert>
97    #include <iostream>
98    using namespace std;
99
100   int main() {
101     fenwick_tree_2d<long long> t;
102     t.set(0, 0, 5);
103     t.set(0, 1, 6);
104     t.set(1, 0, 7);
105     t.add(2, 2, 9);
106     t.add(1, 0, -4);
107     t.add(1, 1, 2, 2, 5);
108   /*
109     5 6 0
110     3 5 5
111     0 5 14
112   */
113     cout << "2D BIT values:\n";
114     for (int i = 0; i < 3; i++) {
115       for (int j = 0; j < 3; j++)
```

```
116        cout << t.at(i, j) << "␣";
117      cout << "\n";
118    }
119    assert(t.sum(0, 0, 0, 1) == 11);
120    assert(t.sum(0, 0, 1, 0) == 8);
121    assert(t.sum(1, 1, 2, 2) == 29);
122    return 0;
123  }
```

3.3   1D Range Queries

3.3.1   Simple Segment Tree

```
1   /*
2
3   3.3.1 - 1D Segment Tree (Simple Version for ints)
4
5   Description: A segment tree is a data structure used for
6   solving the dynamic range query problem, which asks to
7   determine the minimum (or maximum) value in any given
8   range in an array that is constantly being updated.
9
10  Time Complexity: Assuming merge() is O(1), build is O(n)
11  while query() and update() are O(log n). If merge() is
12  not O(1), then all running times are multiplied by a
13  factor of whatever complexity merge() runs in.
14
15  Space Complexity: O(MAXN). Note that a segment tree with
16  N leaves requires 2^(log2(N) - 1) = 4*N total nodes.
17
18  Note: This implementation is 0-based, meaning that all
19  indices from 0 to MAXN - 1, inclusive, are accessible.
20
21  =~=~=~=~= Sample Input =~=~=~=~=
22  5 10
23  35232
24  390942
25  649675
26  224475
27  18709
28  Q 1 3
29  M 4 475689
30  Q 2 3
31  Q 1 3
32  Q 1 2
33  Q 3 3
34  Q 2 3
35  M 2 645514
36  M 2 680746
37  Q 0 4
38
39  =~=~=~=~= Sample Output =~=~=~=~=
40  224475
41  224475
42  224475
43  390942
```

```
44   224475
45   224475
46   35232
47
48   */
49
50   const int MAXN = 100000;
51   int N, M, a[MAXN], t[4*MAXN];
52
53   //define your custom nullv and merge() below.
54   //merge(x, nullv) must return x for all x
55
56   const int nullv = 1 << 30;
57
58   inline int merge(int a, int b) { return a < b ? a : b; }
59
60   void build(int n, int lo, int hi) {
61     if (lo == hi) {
62       t[n] = a[lo];
63       return;
64     }
65     build(2*n + 1, lo, (lo + hi)/2);
66     build(2*n + 2, (lo + hi)/2 + 1, hi);
67     t[n] = merge(t[2*n + 1], t[2*n + 2]);
68   }
69
70   //x and y must be manually set before each call to the
71   //functions below. For query(), [x, y] is the range to
72   //be considered. For update(), a[x] is to be set to y.
73   int x, y;
74
75   //merge(a[i] for i = x..y, inclusive)
76   int query(int n, int lo, int hi) {
77     if (hi < x || lo > y) return nullv;
78     if (lo >= x && hi <= y) return t[n];
79     return merge(query(2*n + 1, lo, (lo + hi) / 2),
80                  query(2*n + 2, (lo + hi) / 2 + 1, hi));
81   }
82
83   //a[x] = y
84   void update(int n, int lo, int hi) {
85     if (hi < x || lo > x) return;
86     if (lo == hi) {
87       t[n] = y;
88       return;
89     }
90     update(2*n + 1, lo, (lo + hi)/2);
91     update(2*n + 2, (lo + hi)/2 + 1, hi);
92     t[n] = merge(t[2*n + 1], t[2*n + 2]);
93   }
94
95   /*** Example Usage (wcipeg.com/problem/segtree) ***/
96
97   #include <cstdio>
98
99   int main() {
100    scanf("%d%d", &N, &M);
101    for (int i = 0; i < N; i++) scanf("%d", &a[i]);
102    build(0, 0, N - 1);
```

```
103    char op;
104    for (int i = 0; i < M; i++) {
105      scanf(" %c%d%d", &op, &x, &y);
106      if (op == 'Q') {
107        printf("%d\n", query(0, 0, N - 1));
108      } else if (op == 'M') {
109        update(0, 0, N - 1);
110      }
111    }
112    return 0;
113  }
```

3.3.2   Segment Tree

```
1   /*
2
3   3.3.2 - 1D Segment Tree Class
4
5   Description: A segment tree is a data structure used for
6   solving the dynamic range query problem, which asks to
7   determine the minimum (or maximum) value in any given
8   range in an array that is constantly being updated.
9
10  Time Complexity: Assuming merge() is O(1), query(),
11  update(), and at() are O(log N). size() is O(1). If
12  merge() is not O(1), then all logarithmic running times
13  are multiplied by a factor of the complexity of merge().
14
15  Space Complexity: O(MAXN). Note that a segment tree with
16  N leaves requires 2^(log2(N) - 1) = 4*N total nodes.
17
18  Note: This implementation is 0-based, meaning that all
19  indices from 0 to N - 1, inclusive, are accessible.
20
21  */
22
23  #include <limits> /* std::numeric_limits<T>::min() */
24  #include <vector>
25
26  template<class T> class segment_tree {
27    int len, x, y;
28    std::vector<T> t;
29    T val, *init;
30
31    //define the following yourself. merge(x, nullv) must return x for all x
32    static inline T nullv() { return std::numeric_limits<T>::min(); }
33    static inline T merge(const T & a, const T & b) { return a > b ? a : b; }
34
35    void build(int n, int lo, int hi) {
36      if (lo == hi) {
37        t[n] = init[lo];
38        return;
39      }
40      build(n * 2 + 1, lo, (lo + hi) / 2);
41      build(n * 2 + 2, (lo + hi) / 2 + 1, hi);
42      t[n] = merge(t[n * 2 + 1], t[n * 2 + 2]);
43    }
```

```cpp
44
45    void update(int n, int lo, int hi) {
46      if (x < lo || x > hi) return;
47      if (lo == hi) {
48        t[n] = val;
49        return;
50      }
51      update(n * 2 + 1, lo, (lo + hi) / 2);
52      update(n * 2 + 2, (lo + hi) / 2 + 1, hi);
53      t[n] = merge(t[n * 2 + 1], t[n * 2 + 2]);
54    }
55
56    T query(int n, int lo, int hi) {
57      if (hi < x || lo > y) return nullv();
58      if (lo >= x && hi <= y) return t[n];
59      return merge(query(n * 2 + 1, lo, (lo + hi) / 2),
60                   query(n * 2 + 2, (lo + hi) / 2 + 1, hi));
61    }
62
63   public:
64    segment_tree(int n, T * a = 0): len(n), t(4 * n, nullv()) {
65      if (a != 0) {
66        init = a;
67        build(0, 0, len - 1);
68      }
69    }
70
71    //a[i] = v
72    void update(int i, const T & v) {
73      x = i;
74      val = v;
75      update(0, 0, len - 1);
76    }
77
78    //merge(a[i] for i = lo..hi, inclusive)
79    T query(int lo, int hi) {
80      x = lo;
81      y = hi;
82      return query(0, 0, len - 1);
83    }
84
85    inline int size() { return len; }
86    inline T at(int i) { return query(i, i); }
87  };
88
89  /*** Example Usage ***/
90
91  #include <iostream>
92  using namespace std;
93
94  int main() {
95    int arr[5] = {6, -2, 1, 8, 10};
96    segment_tree<int> T(5, arr);
97    T.update(1, 4);
98    cout << "Array contains:";
99    for (int i = 0; i < T.size(); i++)
100       cout << " " << T.at(i);
101     cout << "\nThe max value in the range [0, 3] is ";
102     cout << T.query(0, 3) << ".\n"; //8
```

```
103      return 0;
104  }
```

3.3.3   Segment Tree (Range Updates)

```
1   /*
2
3   3.3.3 - 1D Segment Tree with Range Updates
4
5   Description: A segment tree is a data structure used for
6   solving the dynamic range query problem, which asks to
7   determine the minimum (or maximum) value in any given
8   range in an array that is constantly being updated.
9   Lazy propagation is a technique applied to segment trees that
10  allows range updates to be carried out in O(log N) time. The
11  range updating mechanism is less versatile than the one
12  implemented in the next section.
13
14  Time Complexity: Assuming merge() is O(1), query(), update(),
15  at() are O(log(N)). If merge() is not constant time, then all
16  running times are multiplied by whatever complexity the merge
17  function runs in.
18
19  Space Complexity: O(N) on the size of the array. A segment
20  tree for an array of size N needs 2^(log2(N)-1) = 4N nodes.
21
22  Note: This implementation is 0-based, meaning that all
23  indices from 0 to size() - 1, inclusive, are accessible.
24
25  */
26
27  #include <limits> /* std::numeric_limits<T>::min() */
28  #include <vector>
29
30  template<class T> class segment_tree {
31    int len, x, y;
32    std::vector<T> tree, lazy;
33    T val, *init;
34
35    //define the following yourself. merge(x, nullv) must return x for all valid x
36    static inline T nullv() { return std::numeric_limits<T>::min(); }
37    static inline T merge(const T & a, const T & b) { return a > b ? a : b; }
38
39    void build(int n, int lo, int hi) {
40      if (lo == hi) {
41        tree[n] = init[lo];
42        return;
43      }
44      build(n * 2 + 1, lo, (lo + hi) / 2);
45      build(n * 2 + 2, (lo + hi) / 2 + 1, hi);
46      tree[n] = merge(tree[n * 2 + 1], tree[n * 2 + 2]);
47    }
48
49    T query(int n, int lo, int hi) {
50      if (x > hi || y < lo) return nullv();
51      if (x <= lo && hi <= y) {
52        if (lazy[n] == nullv()) return tree[n];
```

```
53        return tree[n] = lazy[n];
54      }
55      int lchild = n * 2 + 1, rchild = n * 2 + 2;
56      if (lazy[n] != nullv()) {
57        lazy[lchild] = lazy[rchild] = lazy[n];
58        lazy[n] = nullv();
59      }
60      return merge(query(lchild, lo, (lo + hi)/2),
61                   query(rchild, (lo + hi)/2 + 1, hi));
62    }
63
64    void _update(int n, int lo, int hi) {
65      if (x > hi || y < lo) return;
66      if (lo == hi) {
67        tree[n] = val;
68        return;
69      }
70      if (x <= lo && hi <= y) {
71        tree[n] = lazy[n] = merge(lazy[n], val);
72        return;
73      }
74      int lchild = n * 2 + 1, rchild = n * 2 + 2;
75      if (lazy[n] != nullv()) {
76        lazy[lchild] = lazy[rchild] = lazy[n];
77        lazy[n] = nullv();
78      }
79      _update(lchild, lo, (lo + hi) / 2);
80      _update(rchild, (lo + hi) / 2 + 1, hi);
81      tree[n] = merge(tree[lchild], tree[rchild]);
82    }
83
84  public:
85    segment_tree(int n, T * a = 0):
86     len(n), tree(4 * n, nullv()), lazy(4 * n, nullv()) {
87      if (a != 0) {
88        init = a;
89        build(0, 0, len - 1);
90      }
91    }
92
93    void update(int i, const T & v) {
94      x = y = i;
95      val = v;
96      _update(0, 0, len - 1);
97    }
98
99    //a[i] = v for i = lo..hi, inclusive
100   void update(int lo, int hi, const T & v) {
101     x = lo; y = hi;
102     val = v;
103     _update(0, 0, len - 1);
104   }
105
106   //returns merge(a[i] for i = lo..hi, inclusive)
107   T query(int lo, int hi) {
108     x = lo;
109     y = hi;
110     return query(0, 0, len - 1);
111   }
```

```
112
113    inline int size() { return len; }
114    inline T at(int i) { return query(i, i); }
115  };
116
117  /*** Example Usage ***/
118
119  #include <iostream>
120  using namespace std;
121
122  int main() {
123    int arr[5] = {6, 4, 1, 8, 10};
124    segment_tree<int> T(5, arr);
125    cout << "Array␣contains:"; //6 4 1 8 10
126    for (int i = 0; i < T.size(); i++)
127      cout << "␣" << T.at(i);
128    cout << "\n";
129    T.update(2, 4, 12);
130    cout << "Array␣contains:"; //6 4 12 12 12
131    for (int i = 0; i < T.size(); i++)
132      cout << "␣" << T.at(i);
133    cout << "\nThe␣max␣value␣in␣the␣range␣[0,␣3]␣is␣";
134    cout << T.query(0, 3) << ".\n"; //12
135    return 0;
136  }
```

3.3.4   Segment Tree (Fast, Non-recursive)

```
1   /*
2
3   3.3.4 - 1D Segment Tree with Range Updates (Fast, No Recursion)
4
5   Description: A segment tree is a data structure used for
6   solving the dynamic range query problem, which asks to
7   determine the minimum (or maximum) value in any given
8   range in an array that is constantly being updated.
9   Lazy propagation is a technique applied to segment trees that
10  allows range updates to be carried out in O(log N) time.
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.
18
19  Note: This implementation is 0-based, meaning that all
20  indices from 0 to T.size() - 1, inclusive, are accessible.
21
22  */
23
24  #include <algorithm> /* std::fill(), std::max() */
25  #include <stdexcept> /* std::runtime_error */
26  #include <vector>
27
28  template<class T> class segment_tree {
29    //Modify the following 5 methods 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 == nullv()) return nullv();
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 nullv() { return 0; }
50     static inline T initv() { return 0; }
51
52     int length;
53     std::vector<T> value, delta;
54     std::vector<int> len;
55
56     static T join_value_with_delta(const T & val, const T & delta) {
57       return delta == nullv() ? val : modify_op(val, delta);
58     }
59
60     static T join_deltas(const T & delta1, const T & delta2) {
61       if (delta1 == nullv()) return delta2;
62       if (delta2 == nullv()) return delta1;
63       return modify_op(delta1, delta2);
64     }
65
66     T join_value_with_delta(int i) {
67       return join_value_with_delta(value[i], delta_on_segment(delta[i], len[i]));
68     }
69
70     void push_delta(int i) {
71       int d = 0;
72       while ((i >> d) > 0) d++;
73       for (d -= 2; d >= 0; d--) {
74         int x = i >> d;
75         value[x >> 1] = join_value_with_delta(x >> 1);
76         delta[x] = join_deltas(delta[x], delta[x >> 1]);
77         delta[x ^ 1] = join_deltas(delta[x ^ 1], delta[x >> 1]);
78         delta[x >> 1] = nullv();
79       }
80     }
81
82   public:
83     segment_tree(int n):
84     length(n), value(2 * n), delta(2 * n, nullv()), len(2 * n) {
85       std::fill(len.begin() + n, len.end(), 1);
86       for (int i = 0; i < n; i++) value[i + n] = initv();
87       for (int i = 2 * n - 1; i > 1; i -= 2) {
88         value[i >> 1] = query_op(value[i], value[i ^ 1]);
```

```
89        len[i >> 1] = len[i] + len[i ^ 1];
90      }
91    }
92
93    T query(int lo, int hi) {
94      if (lo < 0 || hi >= length || lo > hi)
95        throw std::runtime_error("Invalid query range.");
96      push_delta(lo += length);
97      push_delta(hi += length);
98      T res = 0;
99      bool found = false;
100     for (; lo <= hi; lo = (lo + 1) >> 1, hi = (hi - 1) >> 1) {
101       if ((lo & 1) != 0) {
102         res = found ? query_op(res, join_value_with_delta(lo)) :
103                       join_value_with_delta(lo);
104         found = true;
105       }
106       if ((hi & 1) == 0) {
107         res = found ? query_op(res, join_value_with_delta(hi)) :
108                       join_value_with_delta(hi);
109         found = true;
110       }
111     }
112     if (!found) throw std::runtime_error("Not found.");
113     return res;
114   }
115
116   void modify(int lo, int hi, const T & delta) {
117     if (lo < 0 || hi >= length || lo > hi)
118       throw std::runtime_error("Invalid modify range.");
119     push_delta(lo += length);
120     push_delta(hi += length);
121     int ta = -1, tb = -1;
122     for (; lo <= hi; lo = (lo + 1) >> 1, hi = (hi - 1) >> 1) {
123       if ((lo & 1) != 0) {
124         this->delta[lo] = join_deltas(this->delta[lo], delta);
125         if (ta == -1) ta = lo;
126       }
127       if ((hi & 1) == 0) {
128         this->delta[hi] = join_deltas(this->delta[hi], delta);
129         if (tb == -1) tb = hi;
130       }
131     }
132     for (int i = ta; i > 1; i >>= 1)
133       value[i >> 1] = query_op(join_value_with_delta(i),
134                                join_value_with_delta(i ^ 1));
135     for (int i = tb; i > 1; i >>= 1)
136       value[i >> 1] = query_op(join_value_with_delta(i),
137                                join_value_with_delta(i ^ 1));
138   }
139
140   inline int size() { return length; }
141   inline T at(int i) { return query(i, i); }
142 };
143
144 /*** Example Usage ***/
145
146 #include <iostream>
147 using namespace std;
```

```
148
149   int main() {
150     segment_tree<int> T(10);
151     T.modify(0, 0, 10);
152     T.modify(1, 1, 5);
153     T.modify(1, 1, 4);
154     T.modify(2, 2, 7);
155     T.modify(3, 3, 8);
156     cout << T.query(0, 3) << "\n"; //10
157     cout << T.query(1, 3) << "\n"; //9
158     T.modify(0, 9, 5);
159     cout << T.query(0, 9) << "\n"; //15
160     cout << "Array contains:"; //15 14 12 13 5 5 5 5 5 5
161     for (int i = 0; i < T.size(); i++)
162       cout << " " << T.at(i);
163     cout << "\n";
164     return 0;
165   }
```

3.3.5   Implicit Treap

```
1    /*
2
3    3.3.5 - Implicit Treap for Range Operations
4
5    Description: A treap is a self-balancing binary search tree that
6    uses randomization to maintain a low height. In this version,
7    it is used emulate the operations of an std::vector with a tradeoff
8    of increasing the running time of push_back() and at() from O(1) to
9    O(log N), while decreasing the running time of insert() and erase()
10   from O(N) to O(log N). Furthermore, this version supports the same
11   operations as a segment tree with lazy propagation, allowing range
12   updates and queries to be performed in O(log N).
13
14   Time Complexity: Assuming the join functions have constant complexity:
15   insert(), push_back(), erase(), at(), modify(), and query() are all
16   O(log N), while walk() is O(N).
17
18   Space Complexity: O(N) on the size of the array.
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 <climits> /* INT_MIN */
26   #include <cstdlib> /* srand(), rand() */
27   #include <ctime>   /* time() */
28
29   template<class T> class implicit_treap {
30     //Modify the following 5 functions to implement your custom
31     //operations on the tree. This implements the Add/Max operations.
32     //Operations like Add/Sum, Set/Max can also be implemented.
33     static inline T join_values(const T & a, const T & b) {
34       return a > b ? a : b;
35     }
36
```

```
37     static inline T join_deltas(const T & d1, const T & d2) {
38       return d1 + d2;
39     }
40
41     static inline T join_value_with_delta(const T & v, const T & d, int len) {
42       return v + d;
43     }
44
45     static inline T null_delta() { return 0; }
46     static inline T null_value() { return INT_MIN; }
47
48     struct node_t {
49       static inline int rand32() {
50         return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
51       }
52
53       T value, subtree_value, delta;
54       int count, priority;
55       node_t *L, *R;
56
57       node_t(const T & val) {
58         value = subtree_value = val;
59         delta = null_delta();
60         count = 1;
61         L = R = 0;
62         priority = rand32();
63       }
64     } *root;
65
66     static int count(node_t * n) {
67       return n ? n->count : 0;
68     }
69
70     static T subtree_value(node_t * n) {
71       return n ? n->subtree_value : null_value();
72     }
73
74     static void update(node_t * n) {
75       if (n == 0) return;
76       n->subtree_value = join_values(join_values(subtree_value(n->L), n->value),
77                                      subtree_value(n->R));
78       n->count = 1 + count(n->L) + count(n->R);
79     }
80
81     static void apply_delta(node_t * n, const T & delta) {
82       if (n == 0) return;
83       n->delta = join_deltas(n->delta, delta);
84       n->value = join_value_with_delta(n->value, delta, 1);
85       n->subtree_value = join_value_with_delta(n->subtree_value, delta, n->count);
86     }
87
88     static void push_delta(node_t * n) {
89       if (n == 0) return;
90       apply_delta(n->L, n->delta);
91       apply_delta(n->R, n->delta);
92       n->delta = null_delta();
93     }
94
95     static void merge(node_t *& n, node_t * L, node_t * R) {
```

```
 96        push_delta(L);
 97        push_delta(R);
 98        if (L == 0) n = R;
 99        else if (R == 0) n = L;
100        else if (L->priority < R->priority)
101          merge(L->R, L->R, R), n = L;
102        else
103          merge(R->L, L, R->L), n = R;
104        update(n);
105      }
106
107      static void split(node_t * n, node_t *& L, node_t *& R, int key) {
108        push_delta(n);
109        if (n == 0) L = R = 0;
110        else if (key <= count(n->L))
111          split(n->L, L, n->L, key), R = n;
112        else
113          split(n->R, n->R, R, key - count(n->L) - 1), L = n;
114        update(n);
115      }
116
117      static void insert(node_t *& n, node_t * item, int idx) {
118        push_delta(n);
119        if (n == 0) n = item;
120        else if (item->priority < n->priority)
121          split(n, item->L, item->R, idx), n = item;
122        else if (idx <= count(n->L))
123          insert(n->L, item, idx);
124        else
125          insert(n->R, item, idx - count(n->L) - 1);
126        update(n);
127      }
128
129      static T get(node_t * n, int idx) {
130        push_delta(n);
131        if (idx < count(n->L))
132          return get(n->L, idx);
133        else if (idx > count(n->L))
134          return get(n->R, idx - count(n->L) - 1);
135        return n->value;
136      }
137
138      static void erase(node_t *& n, int idx) {
139        push_delta(n);
140        if (idx == count(n->L)) {
141          delete n;
142          merge(n, n->L, n->R);
143        } else if (idx < count(n->L)) {
144          erase(n->L, idx);
145        } else {
146          erase(n->R, idx - count(n->L) - 1);
147        }
148      }
149
150      template<class UnaryFunction>
151      void walk(node_t * n, UnaryFunction f) {
152        if (n == 0) return;
153        push_delta(n);
154        if (n->L) walk(n->L, f);
```

```
155        f(n->value);
156        if (n->R) walk(n->R, f);
157      }
158
159      void clean_up(node_t *& n) {
160        if (n == 0) return;
161        clean_up(n->L);
162        clean_up(n->R);
163        delete n;
164      }
165
166    public:
167      implicit_treap(): root(0) { srand(time(0)); }
168      ~implicit_treap() { clean_up(root); }
169
170      int size() const { return count(root); }
171      bool empty() const { return root == 0; }
172
173      //list.insert(list.begin() + idx, val)
174      void insert(int idx, const T & val) {
175        if (idx < 0 || idx > size()) return;
176        node_t * item = new node_t(val);
177        insert(root, item, idx);
178      }
179
180      void push_back(const T & val) {
181        insert(size(), val);
182      }
183
184      //list.erase(list.begin() + idx)
185      void erase(int idx) {
186        if (idx < 0 || idx >= size()) return;
187        erase(root, idx);
188      }
189
190      T at(int idx) {
191        if (root == 0 || idx < 0 || idx >= size())
192          return null_value();
193        return get(root, idx);
194      }
195
196      template<class UnaryFunction> void walk(UnaryFunction f) {
197        walk(root, f);
198      }
199
200      //for (i = a; i <= b; i++)
201      //  list[i] = join_value_with_delta(list[i], delta)
202      void modify(int a, int b, const T & delta) {
203        if (a < 0 || b < 0 || a >= size() || b >= size() || a > b)
204          return;
205        node_t *l1, *r1;
206        split(root, l1, r1, b + 1);
207        node_t *l2, *r2;
208        split(l1, l2, r2, a);
209        apply_delta(r2, delta);
210        node_t *t;
211        merge(t, l2, r2);
212        merge(root, t, r1);
213      }
```

```
214
215    //return join_values(list[a..b])
216    T query(int a, int b) {
217      if (a < 0 || b < 0 || a >= size() || b >= size() || a > b)
218        return null_value();
219      node_t *l1, *r1;
220      split(root, l1, r1, b + 1);
221      node_t *l2, *r2;
222      split(l1, l2, r2, a);
223      int res = subtree_value(r2);
224      node_t *t;
225      merge(t, l2, r2);
226      merge(root, t, r1);
227      return res;
228    }
229  };
230
231  /*** Example Usage ***/
232
233  #include <iostream>
234  using namespace std;
235
236  void print(int x) { cout << x << "␣"; }
237
238  int main() {
239    implicit_treap<int> T;
240    T.push_back(7);
241    T.push_back(8);
242    T.push_back(9);
243    T.insert(1, 5);
244    T.erase(3);
245    T.walk(print); cout << "\n";   //7 5 8
246    T.modify(0, 2, 2);
247    T.walk(print); cout << "\n";   //9 7 10
248    cout << T.at(1) << "\n";       //7
249    cout << T.query(0, 2) << "\n"; //10
250    cout << T.size() << "\n";      //3
251    return 0;
252  }
```

3.3.6   Sparse Table

```
1   /*
2
3   3.3.6 - Range Minimum Query using a Sparse Table
4
5   Description: The static range minimum query problem can be solved
6   using a sparse table data structure. The RMQ for sub arrays of
7   length 2^k is pre-processed using dynamic programming with formula:
8
9   dp[i][j] = dp[i][j-1], if A[dp[i][j-1]] <= A[dp[i+2^(j-1)-1][j-1]]
10             dp[i+2^(j-1)-1][j-1], otherwise
11
12  where dp[i][j] is the index of the minimum value in the sub array
13  starting at i having length 2^j.
14
15  Time Complexity: O(N log N) for build() and O(1) for min_idx()
```

```
16   Space Complexity: O(N log N) on the size of the array.
17
18   Note: This implementation is 0-based, meaning that all
19   indices from 0 to N - 1, inclusive, are valid.
20
21   */
22
23   #include <vector>
24
25   const int MAXN = 100;
26   std::vector<int> logtable, dp[MAXN];
27
28   void build(int n, int a[]) {
29     logtable.resize(n + 1);
30     for (int i = 2; i <= n; i++)
31       logtable[i] = logtable[i >> 1] + 1;
32     for (int i = 0; i < n; i++) {
33       dp[i].resize(logtable[n] + 1);
34       dp[i][0] = i;
35     }
36     for (int k = 1; (1 << k) < n; k++) {
37       for (int i = 0; i + (1 << k) <= n; i++) {
38         int x = dp[i][k - 1];
39         int y = dp[i + (1 << (k - 1))][k - 1];
40         dp[i][k] = a[x] <= a[y] ? x : y;
41       }
42     }
43   }
44
45   //returns index of min element in [lo, hi]
46   int min_idx(int a[], int lo, int hi) {
47     int k = logtable[hi - lo];
48     int x = dp[lo][k];
49     int y = dp[hi - (1 << k) + 1][k];
50     return a[x] <= a[y] ? x : y;
51   }
52
53   /*** Example Usage ***/
54
55   #include <iostream>
56   using namespace std;
57
58   int main() {
59     int a[] = {7, -10, 5, 20};
60     build(4, a);
61     cout << min_idx(a, 0, 3) << "\n"; //1
62     return 0;
63   }
```

3.3.7   Square Root Decomposition

```
1   /*
2
3   3.3.7 - Square Root Decomposition
4
5   Description: To solve the dynamic range query problem using
6   square root decomposition, we split an array of size N into
```

```
 7   sqrt(N) buckets, each bucket of size sqrt(N). As a result,
 8   each query and update operation will be sqrt(N) in running time.
 9
10   Time Complexity: O(N*sqrt(N)) to construct the initial
11   decomposition. After, query() and update() are O(sqrt N)/call.
12
13   Space Complexity: O(N) for the array. O(sqrt N) for the buckets.
14
15   Note: This implementation is 0-based, meaning that all
16   indices from 0 to N - 1, inclusive, are accessible.
17
18   =~=~=~=~= Sample Input =~=~=~=~=
19   5 10
20   35232
21   390942
22   649675
23   224475
24   18709
25   Q 1 3
26   M 4 475689
27   Q 2 3
28   Q 1 3
29   Q 1 2
30   Q 3 3
31   Q 2 3
32   M 2 645514
33   M 2 680746
34   Q 0 4
35
36   =~=~=~=~= Sample Output =~=~=~=~=
37   224475
38   224475
39   224475
40   390942
41   224475
42   224475
43   35232
44
45   */
46
47   #include <cmath> /* sqrt() */
48   #include <limits> /* std::numeric_limits<T>::max() */
49   #include <vector>
50
51   template<class T> class sqrt_decomp {
52     //define the following yourself. merge(x, nullv) must return x for all x
53     static inline T nullv() { return std::numeric_limits<T>::max(); }
54     static inline T merge(const T & a, const T & b) { return a < b ? a : b; }
55
56     int len, blocklen, blocks;
57     std::vector<T> array, block;
58
59    public:
60     sqrt_decomp(int n, T * a = 0): len(n), array(n) {
61       blocklen = (int)sqrt(n);
62       blocks = (n + blocklen - 1) / blocklen;
63       block.resize(blocks);
64       for (int i = 0; i < n; i++)
65         array[i] = a ? a[i] : nullv();
```

```
66       for (int i = 0; i < blocks; i++) {
67         int h = (i + 1) * blocklen;
68         if (h > n) h = n;
69         block[i] = nullv();
70         for (int j = i * blocklen; j < h; j++)
71           block[i] = merge(block[i], array[j]);
72       }
73     }
74
75     void update(int idx, const T & val) {
76       array[idx] = val;
77       int b = idx / blocklen;
78       int h = (b + 1) * blocklen;
79       if (h > len) h = len;
80       block[b] = nullv();
81       for (int i = b * blocklen; i < h; i++)
82         block[b] = merge(block[b], array[i]);
83     }
84
85     T query(int lo, int hi) {
86       T ret = nullv();
87       int lb = ceil((double)lo / blocklen);
88       int hb = (hi + 1) / blocklen - 1;
89       if (lb > hb) {
90         for (int i = lo; i <= hi; i++)
91           ret = merge(ret, array[i]);
92       } else {
93         int l = lb * blocklen - 1;
94         int h = (hb + 1) * blocklen;
95         for (int i = lo; i <= l; i++)
96           ret = merge(ret, array[i]);
97         for (int i = lb; i <= hb; i++)
98           ret = merge(ret, block[i]);
99         for (int i = h; i <= hi; i++)
100          ret = merge(ret, array[i]);
101      }
102      return ret;
103    }
104
105    inline int size() { return len; }
106    inline int at(int idx) { return array[idx]; }
107  };
108
109  /*** Example Usage (wcipeg.com/problem/segtree) ***/
110
111  #include <cstdio>
112
113  int N, M, A, B, init[100005];
114
115  int main() {
116    scanf("%d%d", &N, &M);
117    for (int i = 0; i < N; i++) scanf("%d", &init[i]);
118    sqrt_decomp<int> a(N, init);
119    char op;
120    for (int i = 0; i < M; i++) {
121      scanf(" %c%d%d", &op, &A, &B);
122      if (op == 'Q') {
123        printf("%d\n", a.query(A, B));
124      } else if (op == 'M') {
```

```
125        a.update(A, B);
126      }
127    }
128    return 0;
129  }
```

3.3.8 Interval Tree (Augmented Treap)

```
1   /*
2
3   3.3.8 - 1D Interval Tree (Augmented Treap)
4
5   Description: An interval tree is structure used to store and efficiently
6   query intervals. An interval may be dynamically inserted, and range
7   queries of [lo, hi] may be performed to have the tree report all intervals
8   that intersect with the queried interval. Augmented trees, described in
9   CLRS (2009, Section 14.3: pp. 348354), is one way to represent these
10  intervals. This implementation uses a treap to maintain balance.
11  See: http://en.wikipedia.org/wiki/Interval_tree#Augmented_tree
12
13  Time Complexity: On average O(log N) for insert() and O(k) for query(),
14  where N is the number of intervals in the tree and k is the number of
15  intervals that will be reported by each query().
16
17  Space Complexity: O(N) on the number of intervals in the tree.
18
19  */
20
21  #include <cstdlib>   /* srand() */
22  #include <ctime>     /* time() */
23  #include <utility>   /* std:pair */
24
25  class interval_tree {
26    typedef std::pair<int, int> interval;
27
28    static bool overlap(const interval & a, const interval & b) {
29      return a.first <= b.second && b.first <= a.second;
30    }
31
32    struct node_t {
33      static inline int rand32() {
34        return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
35      }
36
37      interval i;
38      int maxh, priority;
39      node_t *L, *R;
40
41      node_t(const interval & i) {
42        this->i = i;
43        maxh = i.second;
44        L = R = 0;
45        priority = rand32();
46      }
47
48      void update() {
49        maxh = i.second;
```

```
50          if (L != 0 && L->maxh > maxh) maxh = L->maxh;
51          if (R != 0 && R->maxh > maxh) maxh = R->maxh;
52      }
53    } *root;
54
55    static void rotate_l(node_t *& k2) {
56      node_t *k1 = k2->R;
57      k2->R = k1->L;
58      k1->L = k2;
59      k2 = k1;
60      k2->update();
61      k1->update();
62    }
63
64    static void rotate_r(node_t *& k2) {
65      node_t *k1 = k2->L;
66      k2->L = k1->R;
67      k1->R = k2;
68      k2 = k1;
69      k2->update();
70      k1->update();
71    }
72
73    interval i; //temporary
74
75    void insert(node_t *& n) {
76      if (n == 0) { n = new node_t(i); return; }
77      if (i.first < (n->i).first) {
78        insert(n->L);
79        if (n->L->priority < n->priority) rotate_r(n);
80      } else {
81        insert(n->R);
82        if (n->R->priority < n->priority) rotate_l(n);
83      }
84      n->update();
85    }
86
87    template<class ReportFunction>
88    void query(node_t * n, ReportFunction f) {
89      if (n == 0 || n->maxh < i.first) return;
90      if (overlap(n->i, i)) f(n->i.first, n->i.second);
91      query(n->L, f);
92      query(n->R, f);
93    }
94
95    static void clean_up(node_t * n) {
96      if (n == 0) return;
97      clean_up(n->L);
98      clean_up(n->R);
99      delete n;
100   }
101
102  public:
103    interval_tree(): root(0) { srand(time(0)); }
104    ~interval_tree() { clean_up(root); }
105
106    void insert(int lo, int hi) {
107      i = interval(lo, hi);
108      insert(root);
```

```
109     }
110
111     template<class ReportFunction>
112     void query(int lo, int hi, ReportFunction f) {
113       i = interval(lo, hi);
114       query(root, f);
115     }
116   };
117
118   /*** Example Usage ***/
119
120   #include <cassert>
121   #include <iostream>
122   using namespace std;
123
124   void print(int lo, int hi) {
125     cout << "[" << lo << "," << hi << "]␣";
126   }
127
128   int cnt;
129   void count(int lo, int hi) { cnt++; }
130
131   int main() {
132     int N = 6;
133     int intv[6][2] = {{15, 20}, {10, 30}, {17, 19}, {5, 20}, {12, 15}, {30, 40}};
134     interval_tree T;
135     for (int i = 0; i < N; i++) {
136       T.insert(intv[i][0], intv[i][1]);
137     }
138     T.query(10, 20, print); cout << "\n"; //[15,20] [10,30] [5,20] [12,15] [17,19]
139     T.query(0, 5, print); cout << "\n";   //[5,20]
140     T.query(25, 45, print); cout << "\n"; //[10,30] [30,40]
141     //check correctness
142     for (int l = 0; l <= 50; l++) {
143       for (int h = l; h <= 50; h++) {
144         cnt = 0;
145         T.query(l, h, count);
146         int cnt2 = 0;
147         for (int i = 0; i < N; i++)
148           if (intv[i][0] <= h && l <= intv[i][1])
149             cnt2++;
150         assert(cnt == cnt2);
151       }
152     }
153     return 0;
154   }
```

3.4   2D Range Queries

3.4.1   Quadtree (Simple)

```
1   /*
2
3   3.4.1 - Quadtree (Simple)
4
5   Description: A quadtree can be used to dynamically query values
```

```
 6   of rectangles in a 2D array. In a quadtree, every node has exactly
 7   4 children. The following uses a statically allocated array to
 8   store the nodes. This is less efficient than a 2D segment tree.
 9
10   Time Complexity: For update(), query() and at(): O(log(N*M)) on
11   average and O(sqrt(N*M)) in the worst case, where N is the number
12   of rows and M is the number of columns in the 2D array.
13
14   Space Complexity: O(N*M)
15
16   Note: This implementation is 0-based. Valid indices for
17   all operations are [0..xmax][0..ymax]
18
19   */
20
21   #include <climits> /* INT_MIN */
22
23   const int xmax = 100, ymax = 100;
24   int tree[4 * xmax * ymax];
25   int X, Y, X1, X2, Y1, Y2, V; //temporary value to speed up recursion
26
27   //define the following yourself. merge(x, nullv) must return x for all valid x
28   inline int nullv() { return INT_MIN; }
29   inline int merge(int a, int b) { return a > b ? a : b; }
30
31   void update(int n, int x1, int x2, int y1, int y2) {
32     if (X < x1 || X > x2 || Y < y1 || Y > y2) return;
33     if (x1 == x2 && y1 == y2) {
34       tree[n] = V;
35       return;
36     }
37     update(n * 4 + 1, x1, (x1 + x2) / 2, y1, (y1 + y2) / 2);
38     update(n * 4 + 2, x1, (x1 + x2) / 2, (y1 + y2) / 2 + 1, y2);
39     update(n * 4 + 3, (x1 + x2) / 2 + 1, x2, y1, (y1 + y2) / 2);
40     update(n * 4 + 4, (x1 + x2) / 2 + 1, x2, (y1 + y2) / 2 + 1, y2);
41     tree[n] = merge(merge(tree[n * 4 + 1], tree[n * 4 + 2]),
42                     merge(tree[n * 4 + 3], tree[n * 4 + 4]));
43   }
44
45   void query(int n, int x1, int x2, int y1, int y2) {
46     if (x1 > X2 || x2 < X1 || y2 < Y1 || y1 > Y2 || merge(tree[n], V) == V)
47        return;
48     if (x1 >= X1 && x2 <= X2 && y1 >= Y1 && y2 <= Y2) {
49       V = merge(tree[n], V);
50        return;
51     }
52     query(n * 4 + 1, x1, (x1 + x2) / 2, y1, (y1 + y2) / 2);
53     query(n * 4 + 2, x1, (x1 + x2) / 2, (y1 + y2) / 2 + 1, y2);
54     query(n * 4 + 3, (x1 + x2) / 2 + 1, x2, y1, (y1 + y2) / 2);
55     query(n * 4 + 4, (x1 + x2) / 2 + 1, x2, (y1 + y2) / 2 + 1, y2);
56   }
57
58   void update(int x, int y, int v) {
59     X = x;
60     Y = y;
61     V = v;
62     update(0, 0, xmax - 1, 0, ymax - 1);
63   }
64
```

```
65   int query(int x1, int y1, int x2, int y2) {
66     X1 = x1;
67     X2 = x2;
68     Y1 = y1;
69     Y2 = y2;
70     V = nullv();
71     query(0, 0, xmax - 1, 0, ymax - 1);
72     return V;
73   }
74
75   /*** Example Usage ***/
76
77   #include <iostream>
78   using namespace std;
79
80   int main() {
81     int arr[5][5] = {{1, 2, 3, 4, 5},
82                      {5, 4, 3, 2, 1},
83                      {6, 7, 8, 0, 0},
84                      {0, 1, 2, 3, 4},
85                      {5, 9, 9, 1, 2}};
86     for (int r = 0; r < 5; r++)
87       for (int c = 0; c < 5; c++)
88         update(r, c, arr[r][c]);
89     cout << "The␣maximum␣value␣in␣the␣rectangle␣with␣";
90     cout << "upper␣left␣(0,2)␣and␣lower␣right␣(3,4)␣is␣";
91     cout << query(0, 2, 3, 4) << ".\n"; //8
92     return 0;
93   }
```

3.4.2   Quadtree

```
1    /*
2
3    3.4.2 - Quadtree with Compression
4
5    Description: A quadtree can be used to dynamically query values
6    of rectangles in a 2D array. In a quadtree, every node has exactly
7    4 children. The following uses dynamically allocated memory to
8    store the nodes, which allows arbitrarily large indices to exist
9    without affecting the performance of operations.
10
11   Time Complexity: For update(), query() and at(): O(log(N*M)) on
12   average and O(sqrt(N*M)) in the worst case, where N is the number
13   of rows and M is the number of columns in the 2D array.
14
15   Space Complexity: O(N*M)
16
17   Note: This implementation is 0-based. Valid indices for
18   all operations are [0..XMAX][0..YMAX]
19
20   */
21
22   #include <algorithm> /* std::max(), std::min() */
23   #include <limits>    /* std::numeric_limits<T>::min() */
24
25   template<class T> class quadtree {
```

```
26     //these can be set to large values without affecting your memory usage!
27     static const int xmax = 1000000000;
28     static const int ymax = 1000000000;
29
30     //define the following yourself. merge(x, nullv) must return x for all valid x
31     static inline T nullv() { return std::numeric_limits<T>::min(); }
32     static inline T merge(const T & a, const T & b) { return a > b ? a : b; }
33
34     int X, Y, X1, X2, Y1, Y2;  T V; //temp vals for speed
35
36     struct node_t {
37       node_t * child[4];
38       int x1, x2, y1, y2;
39       T value;
40
41       node_t(int x, int y) {
42         x1 = x2 = x;
43         y1 = y2 = y;
44         child[0] = child[1] = child[2] = child[3] = 0;
45         value = nullv();
46       }
47     } *root;
48
49     void update(node_t *& n, int x1, int x2, int y1, int y2) {
50       if (X < x1 || X > x2 || Y < y1 || Y > y2) return;
51       if (n == 0) n = new node_t(X, Y);
52       if (x1 == x2 && y1 == y2) {
53         n->value = V;
54         return;
55       }
56       int xmid = (x1 + x2)/2, ymid = (y1 + y2)/2;
57       update(n->child[0], x1, xmid, y1, ymid);
58       update(n->child[1], xmid + 1, x2, y1, ymid);
59       update(n->child[2], x1, xmid, ymid + 1, y2);
60       update(n->child[3], xmid + 1, x2, ymid + 1, y2);
61       for (int i = 0; i < 4; i++) {
62         if (!n->child[i] || n->child[i]->value == nullv()) continue;
63         n->x1 = std::min(n->x1, n->child[i]->x1);
64         n->x2 = std::max(n->x2, n->child[i]->x2);
65         n->y1 = std::min(n->y1, n->child[i]->y1);
66         n->y2 = std::max(n->y2, n->child[i]->y2);
67         n->value = merge(n->value, n->child[i]->value);
68       }
69     }
70
71     void query(node_t * n) {
72       if (n == 0 || n->x1 > X2 || n->x2 < X1 || n->y2 < Y1 || n->y1 > Y2 ||
73           merge(n->value, V) == V)
74         return;
75       if (n->x1 >= X1 && n->y1 >= Y1 && n->x2 <= X2 && n->y2 <= Y2) {
76         V = merge(V, n->value);
77         return;
78       }
79       for (int i = 0; i < 4; i++) query(n->child[i]);
80     }
81
82     static void clean_up(node_t * n) {
83       if (n == 0) return;
84       for (int i = 0; i < 4; i++) clean_up(n->child[i]);
```

```
85      delete n;
86    }
87
88  public:
89    quadtree() { root = 0; }
90    ~quadtree() { clean_up(root); }
91
92    void update(int x, int y, const T & v) {
93      X = x;
94      Y = y;
95      V = v;
96      update(root, 0, xmax - 1, 0, ymax - 1);
97    }
98
99    T query(int x1, int y1, int x2, int y2) {
100     X1 = x1;
101     X2 = x2;
102     Y1 = y1;
103     Y2 = y2;
104     V = nullv();
105     query(root);
106     return V;
107   }
108
109   T at(int x, int y) {
110     return query(x, y, x, y);
111   }
112 };
113
114 /*** Example Usage ***/
115
116 #include <iostream>
117 using namespace std;
118
119 int main() {
120   int arr[5][5] = {{1, 2, 3, 4, 5},
121                    {5, 4, 3, 2, 1},
122                    {6, 7, 8, 0, 0},
123                    {0, 1, 2, 3, 4},
124                    {5, 9, 9, 1, 2}};
125   quadtree<int> T;
126   for (int r = 0; r < 5; r++)
127     for (int c = 0; c < 5; c++)
128       T.update(r, c, arr[r][c]);
129   cout << "The maximum value in the rectangle with ";
130   cout << "upper left (0,2) and lower right (3,4) is ";
131   cout << T.query(0, 2, 3, 4) << ".\n"; //8
132   return 0;
133 }
```

3.4.3   2D Segment Tree

```
1  /*
2
3  3.4.3 - 2D Segment Tree
4
5  Description: A quadtree is a segment tree but with 4 children
```

```
 6  per node, making its running time proportional to the square
 7  root of the number of leaves. However, a 2D segment tree is a
 8  segment tree of segment trees, making its running time
 9  proportional to the log of its size. The following implementation
10  is a highly optimized implementation with features such as
11  coordinate compression and path compression.
12
13  Time Complexity: O(log(xmax)*log(ymax)) for update(), query(),
14  and at() operations. size() is O(1).
15
16  Space Complexity: Left as an exercise for the reader.
17
18  Note: This implementation is 0-based. Valid indices for
19  all operations are [0..xmax][0..ymax]
20
21  */
22
23  #include <limits> /* std::numeric_limits<T>::min() */
24
25  template<class T> class segment_tree_2d {
26    //these can be set to large values without affecting your memory usage!
27    static const int xmax = 1000000000;
28    static const int ymax = 1000000000;
29
30    //define the following yourself. merge(x, nullv) must return x for all valid x
31    static inline T nullv() { return std::numeric_limits<T>::min(); }
32    static inline T merge(const T & a, const T & b) { return a > b ? a : b; }
33
34    struct layer2_node {
35      int lo, hi;
36      layer2_node *L, *R;
37      T value;
38      layer2_node(int l, int h) : lo(l), hi(h), L(0), R(0) {}
39    };
40
41    struct layer1_node {
42      layer1_node *L, *R;
43      layer2_node l2;
44      layer1_node() : L(0), R(0), l2(0, ymax) {}
45    } *root;
46
47    void update2(layer2_node * node, int Q, const T & v) {
48      int lo = node->lo, hi = node->hi, mid = (lo + hi)/2;
49      if (lo + 1 == hi) {
50        node->value = v;
51        return;
52      }
53      layer2_node *& tgt = Q < mid ? node->L : node->R;
54      if (tgt == 0) {
55        tgt = new layer2_node(Q, Q + 1);
56        tgt->value = v;
57      } else if (tgt->lo <= Q && Q < tgt->hi) {
58        update2(tgt, Q, v);
59      } else {
60        do {
61          (Q < mid ? hi : lo) = mid;
62          mid = (lo + hi)/2;
63        } while ((Q < mid) == (tgt->lo < mid));
64        layer2_node *nnode = new layer2_node(lo, hi);
```

```
65        (tgt->lo < mid ? nnode->L : nnode->R) = tgt;
66        tgt = nnode;
67        update2(nnode, Q, v);
68      }
69      node->value = merge(node->L ? node->L->value : nullv(),
70                          node->R ? node->R->value : nullv());
71    }
72
73    T query2(layer2_node * nd, int A, int B) {
74      if (nd == 0 || B <= nd->lo || nd->hi <= A) return nullv();
75      if (A <= nd->lo && nd->hi <= B) return nd->value;
76      return merge(query2(nd->L, A, B), query2(nd->R, A, B));
77    }
78
79    void update1(layer1_node * node, int lo, int hi, int x, int y, const T & v) {
80      if (lo + 1 == hi) update2(&node->l2, y, v);
81      else {
82        int mid = (lo + hi)/2;
83        layer1_node *& nnode = x < mid ? node->L : node->R;
84        (x < mid ? hi : lo) = mid;
85        if (nnode == 0) nnode = new layer1_node();
86        update1(nnode, lo, hi, x, y, v);
87        update2(&node->l2, y, merge(
88          node->L ? query2(&node->L->l2, y, y + 1) : nullv(),
89          node->R ? query2(&node->R->l2, y, y + 1) : nullv())
90        );
91      }
92    }
93
94    T query1(layer1_node * nd, int lo, int hi, int A1, int B1, int A2, int B2) {
95      if (nd == 0 || B1 <= lo || hi <= A1) return nullv();
96      if (A1 <= lo && hi <= B1) return query2(&nd->l2, A2, B2);
97      int mid = (lo + hi) / 2;
98      return merge(query1(nd->L, lo, mid, A1, B1, A2, B2),
99                   query1(nd->R, mid, hi, A1, B1, A2, B2));
100   }
101
102   void clean_up2(layer2_node * n) {
103     if (n == 0) return;
104     clean_up2(n->L);
105     clean_up2(n->R);
106     delete n;
107   }
108
109   void clean_up1(layer1_node * n) {
110     if (n == 0) return;
111     clean_up2(n->l2.L);
112     clean_up2(n->l2.R);
113     clean_up1(n->L);
114     clean_up1(n->R);
115     delete n;
116   }
117
118  public:
119   segment_tree_2d() { root = new layer1_node(); }
120   ~segment_tree_2d() { clean_up1(root); }
121
122   void update(int x, int y, const T & v) {
123     update1(root, 0, xmax, x, y, v);
```

```
124     }
125
126     T query(int x1, int y1, int x2, int y2) {
127       return query1(root, 0, xmax, x1, x2 + 1, y1, y2 + 1);
128     }
129
130     T at(int x, int y) {
131       return query(x, y, x, y);
132     }
133   };
134
135   /*** Example Usage ***/
136
137   #include <iostream>
138   using namespace std;
139
140   int main() {
141     int arr[5][5] = {{1, 2, 3, 4, 5},
142                      {5, 4, 3, 2, 1},
143                      {6, 7, 8, 0, 0},
144                      {0, 1, 2, 3, 4},
145                      {5, 9, 9, 1, 2}};
146     segment_tree_2d<int> T;
147     for (int r = 0; r < 5; r++)
148       for (int c = 0; c < 5; c++)
149         T.update(r, c, arr[r][c]);
150     cout << "The maximum value in the rectangle with ";
151     cout << "upper left (0,2) and lower right (3,4) is ";
152     cout << T.query(0, 2, 3, 4) << ".\n"; //8
153     return 0;
154   }
```

3.4.4   K-d Tree (2D Range Query)

```
1   /*
2
3   3.4.4 - K-d Tree for 2D Rectangular Queries
4
5   Description: k-d tree (short for k-dimensional tree) is a space-
6   partitioning data structure for organizing points in a k-
7   dimensional space. The following implementation supports
8   counting the number of points in rectangular ranges after the
9   tree has been build.
10
11  Time Complexity: O(N log N) for build(), where N is the number
12  of points in the tree. count() is O(sqrt N).
13
14  Space Complexity: O(N) on the number of points.
15
16  */
17
18  #include <algorithm> /* nth_element(), max(), min() */
19  #include <climits>   /* INT_MIN, INT_MAX */
20  #include <utility>   /* std::pair */
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, cnt, minx, miny, maxx, maxy;
35    int x1, y1, x2, y2; //temporary values to speed up recursion
36
37    void build(int lo, int hi, bool div_x, point P[]) {
38      if (lo >= hi) return;
39      int mid = (lo + hi) >> 1;
40      std::nth_element(P + lo, P + mid, P + hi, div_x ? cmp_x : cmp_y);
41      tx[mid] = P[mid].first;
42      ty[mid] = P[mid].second;
43      cnt[mid] = hi - lo;
44      minx[mid] = INT_MAX; miny[mid] = INT_MAX;
45      maxx[mid] = INT_MIN; maxy[mid] = INT_MIN;
46      for (int i = lo; i < hi; i++) {
47        minx[mid] = std::min(minx[mid], P[i].first);
48        maxx[mid] = std::max(maxx[mid], P[i].first);
49        miny[mid] = std::min(miny[mid], P[i].second);
50        maxy[mid] = std::max(maxy[mid], P[i].second);
51      }
52      build(lo, mid, !div_x, P);
53      build(mid + 1, hi, !div_x, P);
54    }
55
56    int count(int lo, int hi) {
57      if (lo >= hi) return 0;
58      int mid = (lo + hi) >> 1;
59      int ax = minx[mid], ay = miny[mid];
60      int bx = maxx[mid], by = maxy[mid];
61      if (ax > x2 || x1 > bx || ay > y2 || y1 > by) return 0;
62      if (x1 <= ax && bx <= x2 && y1 <= ay && by <= y2) return cnt[mid];
63      int res = count(lo, mid) + count(mid + 1, hi);
64      res += (x1 <= tx[mid] && tx[mid] <= x2 && y1 <= ty[mid] && ty[mid] <= y2);
65      return res;
66    }
67
68  public:
69    kd_tree(int n, point P[]): tx(n), ty(n), cnt(n),
70      minx(n), miny(n), maxx(n), maxy(n) {
71        build(0, n, true, P);
72    }
73
74    int count(int x1, int y1, int x2, int y2) {
75      this->x1 = x1;
76      this->y1 = y1;
77      this->x2 = x2;
78      this->y2 = y2;
79      return count(0, tx.size());
80    }
81  };
82
```

```
83  /*** Example Usage ***/
84
85  #include <cassert>
86  using namespace std;
87
88  int main() {
89    pair<int, int> P[4];
90    P[0] = make_pair(0, 0);
91    P[1] = make_pair(10, 10);
92    P[2] = make_pair(0, 10);
93    P[3] = make_pair(10, 0);
94    kd_tree t(4, P);
95    assert(t.count(0, 0, 10, 9) == 2);
96    assert(t.count(0, 0, 10, 10) == 4);
97    return 0;
98  }
```

3.4.5   K-d Tree (Nearest Neighbor)

```
1   /*
2
3   3.4.5 - K-d Tree for Nearest Neighbour Queries
4
5   Description: k-d tree (short for k-dimensional tree) is a space-
6   partitioning data structure for organizing points in a k-
7   dimensional space. The following implementation supports
8   querying the nearest neighboring point to (x, y) in terms of
9   Euclidean distance after the tree has been build. Note that
10  a point is not considered its own neighbour if it already exists
11  in the tree.
12
13  Time Complexity: O(N log N) for build(), where N is the number of
14  points in the tree. nearest_neighbor_id() is O(log(N)) on average.
15
16  Space Complexity: O(N) on the number of points.
17
18  */
19
20  #include <algorithm> /* nth_element(), max(), min(), swap() */
21  #include <climits>   /* INT_MIN, INT_MAX */
22  #include <utility>
23  #include <vector>
24
25  class kd_tree {
26    typedef std::pair<int, int> point;
27
28    static inline bool cmp_x(const point & a, const point & b) {
29      return a.first < b.first;
30    }
31
32    static inline bool cmp_y(const point & a, const point & b) {
33      return a.second < b.second;
34    }
35
36    std::vector<int> tx, ty;
37    std::vector<bool> div_x;
38
```

```
39    void build(int lo, int hi, point P[]) {
40      if (lo >= hi) return;
41      int mid = (lo + hi) >> 1;
42      int minx = INT_MAX, maxx = INT_MIN;
43      int miny = INT_MAX, maxy = INT_MIN;
44      for (int i = lo; i < hi; i++) {
45        minx = std::min(minx, P[i].first);
46        maxx = std::max(maxx, P[i].first);
47        miny = std::min(miny, P[i].second);
48        maxy = std::max(maxy, P[i].second);
49      }
50      div_x[mid] = (maxx - minx) >= (maxy - miny);
51      std::nth_element(P + lo, P + mid, P + hi, div_x[mid] ? cmp_x : cmp_y);
52      tx[mid] = P[mid].first;
53      ty[mid] = P[mid].second;
54      if (lo + 1 == hi) return;
55      build(lo, mid, P);
56      build(mid + 1, hi, P);
57    }
58
59    long long min_dist;
60    int min_dist_id, x, y;
61
62    void nearest_neighbor(int lo, int hi) {
63      if (lo >= hi) return;
64      int mid = (lo + hi) >> 1;
65      int dx = x - tx[mid], dy = y - ty[mid];
66      long long d = dx*(long long)dx + dy*(long long)dy;
67      if (min_dist > d && d) {
68        min_dist = d;
69        min_dist_id = mid;
70      }
71      if (lo + 1 == hi) return;
72      int delta = div_x[mid] ? dx : dy;
73      long long delta2 = delta*(long long)delta;
74      int l1 = lo, r1 = mid, l2 = mid + 1, r2 = hi;
75      if (delta > 0) std::swap(l1, l2), std::swap(r1, r2);
76      nearest_neighbor(l1, r1);
77      if (delta2 < min_dist) nearest_neighbor(l2, r2);
78    }
79
80  public:
81    kd_tree(int N, point P[]) {
82      tx.resize(N);
83      ty.resize(N);
84      div_x.resize(N);
85      build(0, N, P);
86    }
87
88    int nearest_neighbor_id(int x, int y) {
89      this->x = x; this->y = y;
90      min_dist = LLONG_MAX;
91      nearest_neighbor(0, tx.size());
92      return min_dist_id;
93    }
94  };
95
96  /*** Example Usage ***/
97
```

```
98   #include <iostream>
99   using namespace std;
100
101  int main() {
102    pair<int, int> P[3];
103    P[0] = make_pair(0, 2);
104    P[1] = make_pair(0, 3);
105    P[2] = make_pair(-1, 0);
106    kd_tree T(3, P);
107    int res = T.nearest_neighbor_id(0, 0);
108    cout << P[res].first << "␣" << P[res].second << "\n"; //-1, 0
109    return 0;
110  }
```

3.4.6   R-Tree (Nearest Segment)

```
1    /*
2
3    3.4.6 - R-Tree for Nearest Neighbouring Line Segment Queries
4
5    Description: R-trees are tree data structures used for spatial
6    access methods, i.e., for indexing multi-dimensional information
7    such as geographical coordinates, rectangles or polygons. The
8    following implementation supports querying of the nearing line
9    segment to a point after a tree of line segments have been built.
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 <cfloat>     /* DBL_MAX */
20   #include <climits>   /* INT_MIN, INT_MAX */
21   #include <vector>
22
23   struct segment { int x1, y1, x2, y2; };
24
25   class r_tree {
26
27     static inline bool cmp_x(const segment & a, const segment & b) {
28       return a.x1 + a.x2 < b.x1 + b.x2;
29     }
30
31     static inline bool cmp_y(const segment & a, const segment & b) {
32       return a.y1 + a.y2 < b.y1 + b.y2;
33     }
34
35     std::vector<segment> s;
36     std::vector<int> minx, maxx, miny, maxy;
37
38     void build(int lo, int hi, bool div_x, segment s[]) {
39       if (lo >= hi) return;
40       int mid = (lo + hi) >> 1;
41       std::nth_element(s + lo, s + mid, s + hi, div_x ? cmp_x : cmp_y);
```

```
42        this->s[mid] = s[mid];
43        for (int i = lo; i < hi; i++) {
44          minx[mid] = std::min(minx[mid], std::min(s[i].x1, s[i].x2));
45          miny[mid] = std::min(miny[mid], std::min(s[i].y1, s[i].y2));
46          maxx[mid] = std::max(maxx[mid], std::max(s[i].x1, s[i].x2));
47          maxy[mid] = std::max(maxy[mid], std::max(s[i].y1, s[i].y2));
48        }
49        build(lo, mid, !div_x, s);
50        build(mid + 1, hi, !div_x, s);
51      }
52
53      double min_dist;
54      int min_dist_id, x, y;
55
56      void nearest_neighbor(int lo, int hi, bool div_x) {
57        if (lo >= hi) return;
58        int mid = (lo + hi) >> 1;
59        double pdist = point_to_segment_squared(x, y, s[mid]);
60        if (min_dist > pdist) {
61          min_dist = pdist;
62          min_dist_id = mid;
63        }
64        long long delta = div_x ? 2*x - s[mid].x1 - s[mid].x2 :
65                                  2*y - s[mid].y1 - s[mid].y2;
66        if (delta <= 0) {
67          nearest_neighbor(lo, mid, !div_x);
68          if (mid + 1 < hi) {
69            int mid1 = (mid + hi + 1) >> 1;
70            long long dist = div_x ? seg_dist(x, minx[mid1], maxx[mid1]) :
71                                     seg_dist(y, miny[mid1], maxy[mid1]);
72            if (dist*dist < min_dist) nearest_neighbor(mid + 1, hi, !div_x);
73          }
74        } else {
75          nearest_neighbor(mid + 1, hi, !div_x);
76          if (lo < mid) {
77            int mid1 = (lo + mid) >> 1;
78            long long dist = div_x ? seg_dist(x, minx[mid1], maxx[mid1]) :
79                                     seg_dist(y, miny[mid1], maxy[mid1]);
80            if (dist*dist < min_dist) nearest_neighbor(lo, mid, !div_x);
81          }
82        }
83      }
84
85      static double point_to_segment_squared(int x, int y, const segment & s) {
86        long long dx = s.x2 - s.x1, dy = s.y2 - s.y1;
87        long long px = x - s.x1, py = y - s.y1;
88        long long square_dist = dx*dx + dy*dy;
89        long long dot_product = dx*px + dy*py;
90        if (dot_product <= 0 || square_dist == 0) return px*px + py*py;
91        if (dot_product >= square_dist)
92          return (px - dx)*(px - dx) + (py - dy)*(py - dy);
93        double q = (double)dot_product/square_dist;
94        return (px - q*dx)*(px - q*dx) + (py - q*dy)*(py - q*dy);
95      }
96
97      static inline int seg_dist(int v, int lo, int hi) {
98        return v <= lo ? lo - v : (v >= hi ? v - hi : 0);
99      }
100
```

```
101    public:
102      r_tree(int N, segment s[]) {
103        this->s.resize(N);
104        minx.assign(N, INT_MAX);
105        maxx.assign(N, INT_MIN);
106        miny.assign(N, INT_MAX);
107        maxy.assign(N, INT_MIN);
108        build(0, N, true, s);
109      }
110
111      int nearest_neighbor_id(int x, int y) {
112        min_dist = DBL_MAX;
113        this->x = x; this->y = y;
114        nearest_neighbor(0, s.size(), true);
115        return min_dist_id;
116      }
117    };
118
119    /*** Example Usage ***/
120
121    #include <iostream>
122    using namespace std;
123
124    int main() {
125      segment s[4];
126      s[0] = (segment){0, 0, 0, 4};
127      s[1] = (segment){0, 4, 4, 4};
128      s[2] = (segment){4, 4, 4, 0};
129      s[3] = (segment){4, 0, 0, 0};
130      r_tree t(4, s);
131      int id = t.nearest_neighbor_id(-1, 2);
132      cout << s[id].x1 << "␣" << s[id].y1 << "␣" <<
133              s[id].x2 << "␣" << s[id].y2 << "\n"; //0 0 0 4
134      return 0;
135    }
```

3.4.7   2D Range Tree

```
1    /*
2
3    3.4.7 - 2D Range Tree for Rectangular Queries
4
5    Description: A range tree is an ordered tree data structure to
6    hold a list of points. It allows all points within a given range
7    to be reported efficiently. Specifically, for a given query, a
8    range tree will report *all* points that lie in the given range.
9    Note that the initial array passed to construct the tree will be
10   sorted, and all resulting query reports will pertain to the
11   indices of points in the sorted array.
12
13   Time Complexity: A range tree can build() in O(N log^(d-1)(N))
14   and query() in O(log^d(n) + k), where N is the number of points
15   stored in the tree, d is the dimension of each point and k is the
16   number of points reported by a given query. Thus for this 2D case
17   build() is O(N log N) and query() is O(log^2(N) + k).
18
19   Space Complexity: O(N log^(d-1)(N)) for a d-dimensional range tree.
```

```
20   Thus for this 2D case, the space complexity is O(N log N).
21
22   */
23
24   #include <algorithm> /* lower_bound(), merge(), sort() */
25   #include <utility>   /* std::pair */
26   #include <vector>
27
28   class range_tree_2d {
29     typedef std::pair<int, int> point;
30
31     std::vector<point> P;
32     std::vector<std::vector<point> > seg;
33
34     static inline bool comp1(const point & a, const point & b) {
35       return a.second < b.second;
36     }
37
38     static inline bool comp2(const point & a, int v) {
39       return a.second < v;
40     }
41
42     void build(int n, int lo, int hi) {
43       if (P[lo].first == P[hi].first) {
44         for (int i = lo; i <= hi; i++)
45           seg[n].push_back(point(i, P[i].second));
46         return;
47       }
48       int l = n * 2 + 1, r = n * 2 + 2;
49       build(l, lo, (lo + hi)/2);
50       build(r, (lo + hi)/2 + 1, hi);
51       seg[n].resize(seg[l].size() + seg[r].size());
52       std::merge(seg[l].begin(), seg[l].end(), seg[r].begin(), seg[r].end(),
53                  seg[n].begin(), comp1);
54     }
55
56     int xl, xh, yl, yh;
57
58     template<class ReportFunction>
59     void query(int n, int lo, int hi, ReportFunction f) {
60       if (P[hi].first < xl || P[lo].first > xh) return;
61       if (xl <= P[lo].first && P[hi].first <= xh) {
62         if (!seg[n].empty() && yh >= yl) {
63           std::vector<point>::iterator it;
64           it = std::lower_bound(seg[n].begin(), seg[n].end(), yl, comp2);
65           for (; it != seg[n].end(); ++it) {
66             if (it->second > yh) break;
67             f(it->first); //or report P[it->first], the actual point
68           }
69         }
70       } else if (lo != hi) {
71         query(n * 2 + 1, lo, (lo + hi) / 2, f);
72         query(n * 2 + 2, (lo + hi) / 2 + 1, hi, f);
73       }
74     }
75
76    public:
77     range_tree_2d(int n, point init[]): seg(4 *n + 1) {
78       std::sort(init, init + n);
```

```
79       P = std::vector<point>(init, init + n);
80       build(0, 0, n - 1);
81     }
82
83     template<class ReportFunction>
84     void query(int x1, int y1, int x2, int y2, ReportFunction f) {
85       xl = x1; xh = x2;
86       yl = y1; yh = y2;
87       query(0, 0, P.size() - 1, f);
88     }
89   };
90
91   /*** Example Usage (wcipeg.com/problem/boxl) ***/
92
93   #include <bitset>
94   #include <cstdio>
95   using namespace std;
96
97   int N, M; bitset<200005> b;
98   pair<int, int> pts[200005];
99   int x1[200005], y1[200005];
100  int x2[200005], y2[200005];
101
102  void mark(int i) {
103    b[i] = true;
104  }
105
106  int main() {
107    scanf("%d%d", &N, &M);
108    for (int i = 0; i < N; i++)
109      scanf("%d%d%d%d", x1 + i, y1 + i, x2 + i, y2 + i);
110    for (int i = 0; i < M; i++)
111      scanf("%d%d", &pts[i].first, &pts[i].second);
112    range_tree_2d t(M, pts);
113    for (int i = 0; i < N; i++)
114      t.query(x1[i], y1[i], x2[i], y2[i], mark);
115    printf("%d\n", b.count());
116    return 0;
117  }
```

3.5   Search Trees and Alternatives

3.5.1   Binary Search Tree

```
1   /*
2
3   3.5.1 - Binary Search Tree
4
5   Description: A binary search tree (BST) is a node-based binary tree data
6   structure where the left sub-tree of every node has keys less than the
7   node's key and the right sub-tree of every node has keys greater than the
8   node's key. A BST may be come degenerate like a linked list resulting in
9   an O(N) running time per operation. A self-balancing binary search tree
10  such as a randomized treap prevents the occurence of this known worst case.
11
12  Note: The following implementation is used similar to an std::map. In order
```

```
13   to make it behave like an std::set, modify the code to remove the value
14   associated with each node. In order to make it behave like an std::multiset
15   or std::multimap, make appropriate changes with key comparisons (e.g.
16   change (k < n->key) to (k <= n->key) in search conditions).
17
18   Time Complexity: insert(), erase() and find() are O(log(N)) on average,
19   but O(N) at worst if the tree becomes degenerate. Speed can be improved
20   by randomizing insertion order if it doesn't matter. walk() is O(N).
21
22   Space Complexity: O(N) on the number of nodes.
23
24   */
25
26   template<class key_t, class val_t> class binary_search_tree {
27     struct node_t {
28       key_t key;
29       val_t val;
30       node_t *L, *R;
31
32       node_t(const key_t & k, const val_t & v) {
33         key = k;
34         val = v;
35         L = R = 0;
36       }
37     } *root;
38
39     int num_nodes;
40
41     static bool insert(node_t *& n, const key_t & k, const val_t & v) {
42       if (n == 0) {
43         n = new node_t(k, v);
44         return true;
45       }
46       if (k < n->key) return insert(n->L, k, v);
47       if (n->key < k) return insert(n->R, k, v);
48       return false; //already exists
49     }
50
51     static bool erase(node_t *& n, const key_t & key) {
52       if (n == 0) return false;
53       if (key < n->key) return erase(n->L, key);
54       if (n->key < key) return erase(n->R, key);
55       if (n->L == 0) {
56         node_t *temp = n->R;
57         delete n;
58         n = temp;
59       } else if (n->R == 0) {
60         node_t *temp = n->L;
61         delete n;
62         n = temp;
63       } else {
64         node_t *temp = n->R, *parent = 0;
65         while (temp->L != 0) {
66           parent = temp;
67           temp = temp->L;
68         }
69         n->key = temp->key;
70         n->val = temp->val;
71         if (parent != 0)
```

```
72            return erase(parent->L, parent->L->key);
73          return erase(n->R, n->R->key);
74        }
75        return true;
76      }
77
78      template<class BinaryFunction>
79      static void walk(node_t * n, BinaryFunction f) {
80        if (n == 0) return;
81        walk(n->L, f);
82        f(n->key, n->val);
83        walk(n->R, f);
84      }
85
86      static void clean_up(node_t * n) {
87        if (n == 0) return;
88        clean_up(n->L);
89        clean_up(n->R);
90        delete n;
91      }
92
93    public:
94      binary_search_tree(): root(0), num_nodes(0) {}
95      ~binary_search_tree() { clean_up(root); }
96      int size() const { return num_nodes; }
97      bool empty() const { return root == 0; }
98
99      bool insert(const key_t & key, const val_t & val) {
100        if (insert(root, key, val)) {
101          num_nodes++;
102          return true;
103        }
104        return false;
105      }
106
107      bool erase(const key_t & key) {
108        if (erase(root, key)) {
109          num_nodes--;
110          return true;
111        }
112        return false;
113      }
114
115      template<class BinaryFunction> void walk(BinaryFunction f) {
116        walk(root, f);
117      }
118
119      val_t * find(const key_t & key) {
120        for (node_t *n = root; n != 0; ) {
121          if (n->key == key) return &(n->val);
122          n = (key < n->key ? n->L : n->R);
123        }
124        return 0; //key not found
125      }
126    };
127
128    /*** Example Usage ***/
129
130    #include <iostream>
```

```
131  using namespace std;
132
133  void printch(int k, char v) { cout << v; }
134
135  int main() {
136    binary_search_tree<int, char> T;
137    T.insert(2, 'b');
138    T.insert(1, 'a');
139    T.insert(3, 'c');
140    T.insert(5, 'e');
141    T.insert(4, 'x');
142    *T.find(4) = 'd';
143    cout << "In-order:␣";
144    T.walk(printch);  //abcde
145    cout << "\nRemoving␣node␣with␣key␣3...";
146    cout << (T.erase(3) ? "Success!" : "Failed");
147    cout << "\n";
148    return 0;
149  }
```

3.5.2 Treap

```
 1  /*
 2
 3  3.5.2 - Treap
 4
 5  Description: A binary search tree (BST) is a node-based binary tree data
 6  structure where the left sub-tree of every node has keys less than the
 7  node's key and the right sub-tree of every node has keys greater than the
 8  node's key. A BST may be come degenerate like a linked list resulting in
 9  an O(N) running time per operation. A self-balancing binary search tree
10  such as a randomized treap prevents the occurence of this known worst case.
11
12  Treaps use randomly generated priorities to reduce the height of the
13  tree. We assume that the rand() function in <cstdlib> is 16-bits, and
14  call it twice to generate a 32-bit number. For the treap to be
15  effective, the range of the randomly generated numbers should be
16  between 0 and around the number of elements in the treap.
17
18  Note: The following implementation is used similar to an std::map. In order
19  to make it behave like an std::set, modify the code to remove the value
20  associated with each node. In order to make it behave like an std::multiset
21  or std::multimap, make appropriate changes with key comparisons (e.g.
22  change (k < n->key) to (k <= n->key) in search conditions).
23
24  Time Complexity: insert(), erase(), and find() are O(log(N)) on average
25  and O(N) in the worst case. Despite the technically O(N) worst case,
26  such cases are still extremely difficult to trigger, making treaps
27  very practice in many programming contest applications. walk() is O(N).
28
29  Space Complexity: O(N) on the number of nodes.
30
31  */
32
33  #include <cstdlib> /* srand(), rand() */
34  #include <ctime>   /* time() */
35
```

```
36   template<class key_t, class val_t> class treap {
37     struct node_t {
38       static inline int rand32() {
39         return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
40       }
41
42       key_t key;
43       val_t val;
44       int priority;
45       node_t *L, *R;
46
47       node_t(const key_t & k, const val_t & v): key(k), val(v), L(0), R(0) {
48         priority = rand32();
49       }
50     } *root;
51
52     int num_nodes;
53
54     static void rotate_l(node_t *& k2) {
55       node_t *k1 = k2->R;
56       k2->R = k1->L;
57       k1->L = k2;
58       k2 = k1;
59     }
60
61     static void rotate_r(node_t *& k2) {
62       node_t *k1 = k2->L;
63       k2->L = k1->R;
64       k1->R = k2;
65       k2 = k1;
66     }
67
68     static bool insert(node_t *& n, const key_t & k, const val_t & v) {
69       if (n == 0) {
70         n = new node_t(k, v);
71         return true;
72       }
73       if (k < n->key && insert(n->L, k, v)) {
74         if (n->L->priority < n->priority) rotate_r(n);
75         return true;
76       } else if (n->key < k && insert(n->R, k, v)) {
77         if (n->R->priority < n->priority) rotate_l(n);
78         return true;
79       }
80       return false;
81     }
82
83     static bool erase(node_t *& n, const key_t & k) {
84       if (n == 0) return false;
85       if (k < n->key) return erase(n->L, k);
86       if (k > n->key) return erase(n->R, k);
87       if (n->L == 0 || n->R == 0) {
88         node_t *temp = n;
89         n = (n->L != 0) ? n->L : n->R;
90         delete temp;
91         return true;
92       }
93       if (n->L->priority < n->R->priority) {
94         rotate_r(n);
```

```
 95        return erase(n->R, k);
 96      }
 97      rotate_l(n);
 98      return erase(n->L, k);
 99    }
100
101    template<class BinaryFunction>
102    static void walk(node_t * n, BinaryFunction f) {
103      if (n == 0) return;
104      walk(n->L, f);
105      f(n->key, n->val);
106      walk(n->R, f);
107    }
108
109    static void clean_up(node_t * n) {
110      if (n == 0) return;
111      clean_up(n->L);
112      clean_up(n->R);
113      delete n;
114    }
115
116  public:
117    treap(): root(0), num_nodes(0) { srand(time(0)); }
118    ~treap() { clean_up(root); }
119    int size() const { return num_nodes; }
120    bool empty() const { return root == 0; }
121
122    bool insert(const key_t & key, const val_t & val) {
123      if (insert(root, key, val)) {
124        num_nodes++;
125        return true;
126      }
127      return false;
128    }
129
130    bool erase(const key_t & key) {
131      if (erase(root, key)) {
132        num_nodes--;
133        return true;
134      }
135      return false;
136    }
137
138    template<class BinaryFunction> void walk(BinaryFunction f) {
139      walk(root, f);
140    }
141
142    val_t * find(const key_t & key) {
143      for (node_t *n = root; n != 0; ) {
144        if (n->key == key) return &(n->val);
145        n = (key < n->key ? n->L : n->R);
146      }
147      return 0; //key not found
148    }
149  };
150
151  /*** Example Usage ***/
152
153  #include <cassert>
```

```
154  #include <iostream>
155  using namespace std;
156
157  void printch(int k, char v) { cout << v; }
158
159  int main() {
160    treap<int, char> T;
161    T.insert(2, 'b');
162    T.insert(1, 'a');
163    T.insert(3, 'c');
164    T.insert(5, 'e');
165    T.insert(4, 'x');
166    *T.find(4) = 'd';
167    cout << "In-order:␣";
168    T.walk(printch);   //abcde
169    cout << "\nRemoving␣node␣with␣key␣3...";
170    cout << (T.erase(3) ? "Success!" : "Failed");
171    cout << "\n";
172
173    //stress test - runs in <0.5 seconds
174    //insert keys in an order that would break a normal BST
175    treap<int, int> T2;
176    for (int i = 0; i < 1000000; i++)
177      T2.insert(i, i*1337);
178    for (int i = 0; i < 1000000; i++)
179      assert(*T2.find(i) == i*1337);
180    return 0;
181  }
```

3.5.3   Size Balanced Tree (Order Statistics)

```
1   /*
2
3   3.5.3 - Size Balanced Tree with Order Statistics
4
5   Description: A binary search tree (BST) is a node-based binary tree data
6   structure where the left sub-tree of every node has keys less than the
7   node's key and the right sub-tree of every node has keys greater than the
8   node's key. A BST may be come degenerate like a linked list resulting in
9   an O(N) running time per operation. A self-balancing binary search tree
10  such as a randomized treap prevents the occurence of this known worst case.
11
12  The size balanced tree is a data structure first published in 2007 by
13  Chinese student Chen Qifeng. The tree is rebalanced by examining the sizes
14  of each node's subtrees. It is popular amongst Chinese OI competitors due
15  to its speed, simplicity to implement, and ability to double up as an
16  ordered statistics tree if necessary.
17  For more info, see: http://wcipeg.com/wiki/Size_Balanced_Tree
18
19  An ordered statistics tree is a BST that supports additional operations:
20  - Select(i): find the i-th smallest element stored in the tree
21  - Rank(x):   find the rank of element x in the tree,
22               i.e. its index in the sorted list of elements of the tree
23  For more info, see: http://en.wikipedia.org/wiki/Order_statistic_tree
24
25  Note: The following implementation is used similar to an std::map. In order
26  to make it behave like an std::set, modify the code to remove the value
```

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

```
86        maintain(n, 0);
87        maintain(n, 1);
88      }
89
90      static void insert(node_t *& n, const key_t & k, const val_t & v) {
91        if (n == 0) {
92          n = new node_t(k, v);
93          return;
94        }
95        if (k < n->key) {
96          insert(n->c[0], k, v);
97          maintain(n, 0);
98        } else if (n->key < k) {
99          insert(n->c[1], k, v);
100         maintain(n, 1);
101       } else return;
102       n->update();
103     }
104
105     static void erase(node_t *& n, const key_t & k) {
106       if (n == 0) return;
107       bool d = k < n->key;
108       if (k < n->key) {
109         erase(n->c[0], k);
110       } else if (n->key < k) {
111         erase(n->c[1], k);
112       } else {
113         if (n->c[1] == 0 || n->c[0] == 0) {
114           delete n;
115           n = n->c[1] == 0 ? n->c[0] : n->c[1];
116           return;
117         }
118         node_t * p = n->c[1];
119         while (p->c[0] != 0) p = p->c[0];
120         n->key = p->key;
121         erase(n->c[1], p->key);
122       }
123       maintain(n, d);
124       n->update();
125     }
126
127     template<class BinaryFunction>
128     static void walk(node_t * n, BinaryFunction f) {
129       if (n == 0) return;
130       walk(n->c[0], f);
131       f(n->key, n->val);
132       walk(n->c[1], f);
133     }
134
135     static std::pair<key_t, val_t> select(node_t *& n, int k) {
136       int r = size(n->c[0]);
137       if (k < r) return select(n->c[0], k);
138       if (k > r) return select(n->c[1], k - r - 1);
139       return std::make_pair(n->key, n->val);
140     }
141
142     static int rank(node_t * n, const key_t & k) {
143       if (n == 0)
144         throw std::runtime_error("Cannot rank key not in tree.");
```

```
145        int r = size(n->c[0]);
146        if (k < n->key) return rank(n->c[0], k);
147        if (n->key < k) return rank(n->c[1], k) + r + 1;
148        return r;
149      }
150
151      static void clean_up(node_t * n) {
152        if (n == 0) return;
153        clean_up(n->c[0]);
154        clean_up(n->c[1]);
155        delete n;
156      }
157
158    public:
159      size_balanced_tree() : root(0) {}
160      ~size_balanced_tree() { clean_up(root); }
161      int size() { return size(root); }
162      bool empty() const { return root == 0; }
163
164      void insert(const key_t & key, const val_t & val) {
165        insert(root, key, val);
166      }
167
168      void erase(const key_t & key) {
169        erase(root, key);
170      }
171
172      template<class BinaryFunction> void walk(BinaryFunction f) {
173        walk(root, f);
174      }
175
176      val_t * find(const key_t & key) {
177        for (node_t *n = root; n != 0; ) {
178          if (n->key == key) return &(n->val);
179          n = (key < n->key ? n->c[0] : n->c[1]);
180        }
181        return 0; //key not found
182      }
183
184      std::pair<key_t, val_t> select(int k) {
185        if (k >= size(root))
186          throw std::runtime_error("k must be smaller size of tree.");
187        return select(root, k);
188      }
189
190      int rank(const key_t & key) {
191        return rank(root, key);
192      }
193    };
194
195    /*** Example Usage ***/
196
197    #include <cassert>
198    #include <iostream>
199    using namespace std;
200
201    void printch(int k, char v) { cout << v; }
202
203    int main() {
```

```
204    size_balanced_tree<int, char> T;
205    T.insert(2, 'b');
206    T.insert(1, 'a');
207    T.insert(3, 'c');
208    T.insert(5, 'e');
209    T.insert(4, 'x');
210    *T.find(4) = 'd';
211    cout << "In-order:␣";
212    T.walk(printch);                    //abcde
213    T.erase(3);
214    cout << "\nRank␣of␣2:␣" << T.rank(2); //1
215    cout << "\nRank␣of␣5:␣" << T.rank(5); //3
216    cout << "\nValue␣of␣3rd␣smallest␣key:␣";
217    cout << T.select(2).second;          //d
218    cout << "\n";
219
220    //stress test - runs in <1 second
221    //insert keys in an order that would break a normal BST
222    size_balanced_tree<int, int> T2;
223    for (int i = 0; i < 1000000; i++)
224      T2.insert(i, i*1337);
225    for (int i = 0; i < 1000000; i++)
226      assert(*T2.find(i) == i*1337);
227    return 0;
228  }
```

3.5.4   Hashmap (Chaining)

```
1   /*
2
3   3.5.4 - Hashmap (Chaining)
4
5   Description: A hashmap (std::unordered_map in C++11) is an
6   alternative to a binary search tree. Hashmaps use more memory than
7   BSTs, but are usually more efficient. The following implementation
8   uses the chaining method to handle collisions. You can use the
9   hash algorithms provided in the example, or define your own.
10
11  Time Complexity: insert(), remove(), find(), are O(1) amortized.
12  rehash() is O(N).
13
14  Space Complexity: O(N) on the number of entries.
15
16  */
17
18  #include <list>
19
20  template<class key_t, class val_t, class Hash> class hashmap {
21    struct entry_t {
22      key_t key;
23      val_t val;
24      entry_t(const key_t & k, const val_t & v): key(k), val(v) {}
25    };
26
27    std::list<entry_t> * table;
28    int table_size, map_size;
29
```

```
30     /**
31      * This doubles the table size, then rehashes every entry.
32      * Rehashing is expensive; it is strongly suggested for the
33      * table to be constructed with a large size to avoid rehashing.
34      */
35     void rehash() {
36       std::list<entry_t> * old = table;
37       int old_size = table_size;
38       table_size = 2*table_size;
39       table = new std::list<entry_t>[table_size];
40       map_size = 0;
41       typename std::list<entry_t>::iterator it;
42       for (int i = 0; i < old_size; i++)
43         for (it = old[i].begin(); it != old[i].end(); ++it)
44           insert(it->key, it->val);
45       delete[] old;
46     }
47
48   public:
49     hashmap(int size = 1024): table_size(size), map_size(0) {
50       table = new std::list<entry_t>[table_size];
51     }
52
53     ~hashmap() { delete[] table; }
54     int size() const { return map_size; }
55
56     void insert(const key_t & key, const val_t & val) {
57       if (find(key) != 0) return;
58       if (map_size >= table_size) rehash();
59       unsigned int i = Hash()(key) % table_size;
60       table[i].push_back(entry_t(key, val));
61       map_size++;
62     }
63
64     void remove(const key_t & key) {
65       unsigned int i = Hash()(key) % table_size;
66       typename std::list<entry_t>::iterator it = table[i].begin();
67       while (it != table[i].end() && it->key != key) ++it;
68       if (it == table[i].end()) return;
69       table[i].erase(it);
70       map_size--;
71     }
72
73     val_t * find(const key_t & key) {
74       unsigned int i = Hash()(key) % table_size;
75       typename std::list<entry_t>::iterator it = table[i].begin();
76       while (it != table[i].end() && it->key != key) ++it;
77       if (it == table[i].end()) return 0;
78       return &(it->val);
79     }
80
81     val_t & operator [] (const key_t & key) {
82       val_t * ret = find(key);
83       if (ret != 0) return *ret;
84       insert(key, val_t());
85       return *find(key);
86     }
87   };
88
```

```
89   /*** Examples of Hash Algorithm Definitions ***/
90
91   #include <string>
92
93   struct class_hash {
94     unsigned int operator () (int key) {
95       return class_hash()((unsigned int)key);
96     }
97
98     unsigned int operator () (long long key) {
99       return class_hash()((unsigned long long)key);
100    }
101
102    //Knuth's multiplicative method (one-to-one)
103    unsigned int operator () (unsigned int key) {
104      return key * 2654435761u; //or just return key
105    }
106
107    //Jenkins's 64-bit hash
108    unsigned int operator () (unsigned long long key) {
109      key += ~(key << 32); key ^= (key >> 22);
110      key += ~(key << 13); key ^= (key >>  8);
111      key +=  (key <<  3); key ^= (key >> 15);
112      key += ~(key << 27); key ^= (key >> 31);
113      return key;
114    }
115
116    //Jenkins's one-at-a-time hash
117    unsigned int operator () (const std::string & key) {
118      unsigned int hash = 0;
119      for (unsigned int i = 0; i < key.size(); i++) {
120        hash += ((hash += key[i]) << 10);
121        hash ^= (hash >> 6);
122      }
123      hash ^= ((hash += (hash << 3)) >> 11);
124      return hash + (hash << 15);
125    }
126  };
127
128  /*** Example Usage ***/
129
130  #include <iostream>
131  using namespace std;
132
133  int main() {
134    hashmap<string, int, class_hash> M;
135    M["foo"] = 1;
136    M.insert("bar", 2);
137    cout << M["foo"] << M["bar"] << endl; //prints 12
138    cout << M["baz"] << M["qux"] << endl; //prints 00
139    M.remove("foo");
140    cout << M.size() << endl;             //prints 3
141    cout << M["foo"] << M["bar"] << endl; //prints 02
142    return 0;
143  }
```

3.5.5   Skip List (Probabilistic)

```
1   /*
2
3   3.5.5 - Skip List (Probabilistic)
4
5   Description: A skip list is an alternative to binary search trees.
6   Fast search is made possible by maintaining a linked hierarchy of
7   subsequences, each skipping over fewer elements. Searching starts
8   in the sparsest subsequence until two consecutive elements have
9   been found, one smaller and one larger than the element searched for.
10  Skip lists are generally slower than binary search trees, but can
11  be easier to implement. The following version uses randomized levels.
12
13  Time Complexity: insert(), erase(), count() and find() are O(log(N))
14  on average, but O(N) in the worst case. walk() is O(N).
15
16  Space Complexity: O(N) on the number of elements inserted on average,
17  but O(N log N) in the worst case.
18
19  */
20
21  #include <cmath>   /* log() */
22  #include <cstdlib> /* rand(), srand() */
23  #include <cstring> /* memset() */
24  #include <ctime>   /* time() */
25
26  template<class key_t, class val_t> struct skip_list {
27    static const int MAX_LEVEL = 32; //~ log2(max # of keys)
28
29    static int random_level() { //geometric distribution
30      static const float P = 0.5;
31      int lvl = log((float)rand()/RAND_MAX)/log(1.0 - P);
32      return lvl < MAX_LEVEL ? lvl : MAX_LEVEL;
33    }
34
35    struct node_t {
36      key_t key;
37      val_t val;
38      node_t **next;
39
40      node_t(int level, const key_t & k, const val_t & v) {
41        next = new node_t * [level + 1];
42        memset(next, 0, sizeof(node_t*)*(level + 1));
43        key = k;
44        val = v;
45      }
46
47      ~node_t() { delete[] next; }
48    } *head, *update[MAX_LEVEL + 1];
49
50    int level, num_nodes;
51
52    skip_list() {
53      srand(time(0));
54      head = new node_t(MAX_LEVEL, key_t(), val_t());
55      level = num_nodes = 0;
56    }
57
58    ~skip_list() { delete head; }
59    int size() { return num_nodes; }
```

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

```
119      while (n) {
120        f(n->key, n->val);
121        n = n->next[0];
122      }
123    }
124  };
125
126  /*** Example Usage: Random Tests ***/
127
128  #include <cassert>
129  #include <iostream>
130  #include <map>
131  using namespace std;
132
133  int main() {
134    map<int, int> m;
135    skip_list<int, int> s;
136    for (int i = 0; i < 50000; i++) {
137      int op = rand() % 3;
138      int val1 = rand(), val2 = rand();
139      if (op == 0) {
140        m[val1] = val2;
141        s.insert(val1, val2);
142      } else if (op == 1) {
143        if (!m.count(val1)) continue;
144        m.erase(val1);
145        s.erase(val1);
146      } else if (op == 2) {
147        assert(s.count(val1) == (int)m.count(val1));
148        if (m.count(val1)) {
149          assert(m[val1] == *s.find(val1));
150        }
151      }
152    }
153    return 0;
154  }
```

3.6   Tree Data Structures

3.6.1   Heavy-Light Decomposition

```
1   /*
2
3   3.6.1 - Heavy-Light Decomposition for Dynamic Path Queries
4
5   Description: Given an undirected, connected graph that is a tree, the
6   heavy-light decomposition (HLD) on the graph is a partitioning of the
7   vertices into disjoint paths to later support dynamic modification and
8   querying of values on paths between pairs of vertices.
9   See: http://wcipeg.com/wiki/Heavy-light_decomposition
10  and: http://blog.anudeep2011.com/heavy-light-decomposition/
11  To support dynamic adding and removal of edges, see link/cut tree.
12
13  Note: The adjacency list tree[] that is passed to the constructor must
14  not be changed afterwards in order for modify() and query() to work.
15
```

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

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

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

```
193    modify(path[a], std::min(pathpos[a], pathpos[b]) + VALUES_ON_EDGES,
194        std::max(pathpos[a], pathpos[b]), delta);
195    }
196  };
197
198  /*** Example Usage ***/
199
200  #include <iostream>
201  using namespace std;
202
203  const int MAXN = 1000;
204  vector<int> adj[MAXN];
205
206  /*
207      w=10        w=20        w=40
208    0---------1---------2---------3
209                        \
210                         ---------4
211                        w=30
212  */
213  int main() {
214    adj[0].push_back(1);
215    adj[1].push_back(0);
216    adj[1].push_back(2);
217    adj[2].push_back(1);
218    adj[2].push_back(3);
219    adj[3].push_back(2);
220    adj[2].push_back(4);
221    adj[4].push_back(2);
222    heavy_light<int> hld(5, adj);
223    hld.modify(0, 1, 10);
224    hld.modify(1, 2, 20);
225    hld.modify(2, 3, 40);
226    hld.modify(2, 4, 30);
227    cout << hld.query(0, 3) << "\n"; //40
228    cout << hld.query(2, 4) << "\n"; //30
229    hld.modify(3, 4, 50); //w[every edge from 3 to 4] += 50
230    cout << hld.query(1, 4) << "\n"; //80
231    return 0;
232  }
```

3.6.2   Link-Cut Tree

```
1   /*
2
3   3.6.2 - Link/Cut Tree for Dynamic Path Queries and Connectivity
4
5   Description: Given an unweighted forest of trees where each node
6   has an associated value, a link/cut tree can be used to dynamically
7   query and modify values on the path between pairs of nodes a tree.
8   This problem can be solved using heavy-light decomposition, which
9   also supports having values stored on edges rather than the nodes.
10  However in a link/cut tree, nodes in different trees may be
11  dynamically linked, edges between nodes in the same tree may be
12  dynamically split, and connectivity between two nodes (whether they
13  are in the same tree) may be checked.
14
```

```
15   Time Complexity: O(log N) amortized for make_root(), link(), cut(),
16   connected(), modify(), and query(), where N is the number of nodes
17   in the forest.
18
19   Space Complexity: O(N) on the number of nodes in the forest.
20
21   */
22
23   #include <algorithm> /* std::max(), std::swap() */
24   #include <climits>   /* INT_MIN */
25   #include <map>
26   #include <stdexcept> /* std::runtime_error() */
27
28   template<class T> class linkcut_forest {
29     //Modify the following 5 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 null_delta() { return 0; }
50     static inline T null_value() { return INT_MIN; }
51
52     static inline T join_value_with_delta(const T & v, const T & delta) {
53       return delta == null_delta() ? v : modify_op(v, delta);
54     }
55
56     static T join_deltas(const T & delta1, const T & delta2) {
57       if (delta1 == null_delta()) return delta2;
58       if (delta2 == null_delta()) return delta1;
59       return modify_op(delta1, delta2);
60     }
61
62     struct node_t {
63       T value, subtree_value, delta;
64       int size;
65       bool rev;
66       node_t *L, *R, *parent;
67
68       node_t(const T & v) {
69         value = subtree_value = v;
70         delta = null_delta();
71         size = 1;
72         rev = false;
73         L = R = parent = 0;
```

```
74        }
75
76        bool is_root() { //is this the root of a splay tree?
77          return parent == 0 || (parent->L != this && parent->R != this);
78        }
79
80        void push() {
81          if (rev) {
82            rev = false;
83            std::swap(L, R);
84            if (L != 0) L->rev = !L->rev;
85            if (R != 0) R->rev = !R->rev;
86          }
87          value = join_value_with_delta(value, delta);
88          subtree_value = join_value_with_delta(subtree_value,
89                            delta_on_segment(delta, size));
90          if (L != 0) L->delta = join_deltas(L->delta, delta);
91          if (R != 0) R->delta = join_deltas(R->delta, delta);
92          delta = null_delta();
93        }
94
95        void update() {
96          subtree_value = query_op(query_op(get_subtree_value(L),
97                                      join_value_with_delta(value, delta)),
98                               get_subtree_value(R));
99          size = 1 + get_size(L) + get_size(R);
100       }
101     };
102
103     static inline int get_size(node_t * n) {
104       return n == 0 ? 0 : n->size;
105     }
106
107     static inline int get_subtree_value(node_t * n) {
108       return n == 0 ? null_value() : join_value_with_delta(n->subtree_value,
109                                 delta_on_segment(n->delta, n->size));
110     }
111
112     static void connect(node_t * ch, node_t * p, char is_left) {
113       if (ch != 0) ch->parent = p;
114       if (is_left < 0) return;
115       (is_left ? p->L : p->R) = ch;
116     }
117
118     /** rotates edge (n, n.parent)
119      *         g           g
120      *        /           /
121      *       p           n
122      *      / \   -->   / \
123      *     n  p.r     n.l  p
124      *    / \             / \
125      *  n.l n.r        n.r p.r
126      */
127     static void rotate(node_t * n) {
128       node_t *p = n->parent, *g = p->parent;
129       bool is_rootp = p->is_root(), is_left = (n == p->L);
130       connect(is_left ? n->R : n->L, p, is_left);
131       connect(p, n, !is_left);
132       connect(n, g, is_rootp ? -1 : (p == g->L));
```

```
133       p->update();
134     }
135
136     /** brings n to the root, balancing tree
137      *
138      *  zig-zig case:
139      *         g                                    n
140      *        / \                 p                / \
141      *       p  g.r rot(p)    /    \     rot(n) n.l  p
142      *      / \      -->    n      g    -->        / \
143      *     n  p.r          / \    / \            n.r  g
144      *    / \            n.l n.r p.r g.r             / \
145      *  n.l n.r                                    p.r g.r
146      *
147      *  zig-zag case:
148      *        g                  g
149      *       / \                / \                  n
150      *      p  g.r rot(n)    n  g.r rot(n)     /   \
151      *     / \       -->    / \        -->    p       g
152      *   p.l  n            p  n.r            / \     / \
153      *       / \          / \              p.l n.l n.r g.r
154      *     n.l n.r      p.l n.l
155      */
156     static void splay(node_t * n) {
157       while (!n->is_root()) {
158         node_t *p = n->parent, *g = p->parent;
159         if (!p->is_root()) g->push();
160         p->push();
161         n->push();
162         if (!p->is_root())
163           rotate((n == p->L) == (p == g->L) ? p/*zig-zig*/ : n/*zig-zag*/);
164         rotate(n);
165       }
166       n->push();
167       n->update();
168     }
169
170     //makes node n the root of the virtual tree,
171     //and also n becomes the leftmost node in its splay tree
172     static node_t * expose(node_t * n) {
173       node_t *prev = 0;
174       for (node_t *i = n; i != 0; i = i->parent) {
175         splay(i);
176         i->L = prev;
177         prev = i;
178       }
179       splay(n);
180       return prev;
181     }
182
183     std::map<int, node_t*> nodes; //use array if ID compression not required
184     node_t *u, *v; //temporary
185
186     void get_uv(int a, int b) {
187       static typename std::map<int, node_t*>::iterator it1, it2;
188       it1 = nodes.find(a);
189       it2 = nodes.find(b);
190       if (it1 == nodes.end() || it2 == nodes.end())
191         throw std::runtime_error("Error: a or b does not exist in forest.");
```

```
192      u = it1->second;
193      v = it2->second;
194    }
195
196  public:
197    ~linkcut_forest() {
198      static typename std::map<int, node_t*>::iterator it;
199      for (it = nodes.begin(); it != nodes.end(); ++it)
200        delete it->second;
201    }
202
203    void make_root(int id, const T & initv) {
204      if (nodes.find(id) != nodes.end())
205        throw std::runtime_error("Cannot_make_root():_ID_already_exists.");
206      node_t * n = new node_t(initv);
207      expose(n);
208      n->rev = !n->rev;
209      nodes[id] = n;
210    }
211
212    bool connected(int a, int b) {
213      get_uv(a, b);
214      if (a == b) return true;
215      expose(u);
216      expose(v);
217      return u->parent != 0;
218    }
219
220    void link(int a, int b) {
221      if (connected(a, b))
222        throw std::runtime_error("Error:_a_and_b_are_already_connected.");
223      get_uv(a, b);
224      expose(u);
225      u->rev = !u->rev;
226      u->parent = v;
227    }
228
229    void cut(int a, int b) {
230      get_uv(a, b);
231      expose(u);
232      u->rev = !u->rev;
233      expose(v);
234      if (v->R != u || u->L != 0)
235        throw std::runtime_error("Error:_edge_(a,_b)_does_not_exist.");
236      v->R->parent = 0;
237      v->R = 0;
238    }
239
240    T query(int a, int b) {
241      if (!connected(a, b))
242        throw std::runtime_error("Error:_a_and_b_are_not_connected.");
243      get_uv(a, b);
244      expose(u);
245      u->rev = !u->rev;
246      expose(v);
247      return get_subtree_value(v);
248    }
249
250    void modify(int a, int b, const T & delta) {
```

```
251       if (!connected(a, b))
252         throw std::runtime_error("Error: a and b are not connected.");
253       get_uv(a, b);
254       expose(u);
255       u->rev = !u->rev;
256       expose(v);
257       v->delta = join_deltas(v->delta, delta);
258     }
259 };
260
261 /*** Example Usage ***/
262
263 #include <iostream>
264 using namespace std;
265
266 int main() {
267   linkcut_forest<int> F;
268 /*
269  v=10        v=40        v=20        v=10
270    0---------1---------2---------3
271                          \
272                           ---------4
273                                  v=30
274 */
275   F.make_root(0, 10);
276   F.make_root(1, 40);
277   F.make_root(2, 20);
278   F.make_root(3, 10);
279   F.make_root(4, 30);
280   F.link(0, 1);
281   F.link(1, 2);
282   F.link(2, 3);
283   F.link(2, 4);
284   cout << F.query(1, 4) << "\n"; //40
285   F.modify(1, 1, -10);
286   F.modify(3, 4, -10);
287 /*
288  v=10        v=30        v=10        v=0
289    0---------1---------2---------3
290                          \
291                           ---------4
292                                  v=20
293 */
294   cout << F.query(0, 4) << "\n"; //30
295   cout << F.query(3, 4) << "\n"; //20
296   F.cut(1, 2);
297   cout << F.connected(1, 2) << "\n"; //0
298   cout << F.connected(0, 4) << "\n"; //0
299   cout << F.connected(2, 3) << "\n"; //1
300   return 0;
301 }
```

3.7   Lowest Common Ancestor

3.7.1   Sparse Tables

```
1   /*
2
3   3.7.1 - Sparse Tables for Lowest Common Ancestor
4
5   Description: Given an undirected graph that is a tree, the
6   lowest common ancestor (LCA) of two nodes v and w is the
7   lowest (i.e. deepest) node that has both v and w as descendants,
8   where we define each node to be a descendant of itself (so if
9   v has a direct connection from w, w is the lowest common
10  ancestor). The following program uses sparse tables to solve
11  the problem on an unchanging graph.
12
13  Time Complexity: O(N log N) for build() and O(log N) for lca(),
14  where N is the number of nodes in the tree.
15
16  Space Complexity: O(N log N).
17
18  */
19
20  #include <vector>
21
22  const int MAXN = 1000;
23  int len, timer, tin[MAXN], tout[MAXN];
24  std::vector<int> adj[MAXN], dp[MAXN];
25
26  void dfs(int u, int p) {
27    tin[u] = timer++;
28    dp[u][0] = p;
29    for (int i = 1; i < len; i++)
30      dp[u][i] = dp[dp[u][i - 1]][i - 1];
31    for (int j = 0, v; j < (int)adj[u].size(); j++)
32      if ((v = adj[u][j]) != p)
33        dfs(v, u);
34    tout[u] = timer++;
35  }
36
37  void build(int nodes, int root) {
38    len = 1;
39    while ((1 << len) <= nodes) len++;
40    for (int i = 0; i < nodes; i++)
41      dp[i].resize(len);
42    timer = 0;
43    dfs(root, root);
44  }
45
46  inline bool is_parent(int parent, int child) {
47    return tin[parent] <= tin[child] && tout[child] <= tout[parent];
48  }
49
50  int lca(int a, int b) {
51    if (is_parent(a, b)) return a;
52    if (is_parent(b, a)) return b;
53    for (int i = len - 1; i >= 0; i--)
54      if (!is_parent(dp[a][i], b))
55        a = dp[a][i];
56    return dp[a][0];
57  }
58
59  /*** Example Usage ***/
```

```
60
61  #include <iostream>
62  using namespace std;
63
64  int main() {
65    adj[0].push_back(1);
66    adj[1].push_back(0);
67    adj[1].push_back(2);
68    adj[2].push_back(1);
69    adj[3].push_back(1);
70    adj[1].push_back(3);
71    adj[0].push_back(4);
72    adj[4].push_back(0);
73    build(5, 0);
74    cout << lca(3, 2) << "\n"; //1
75    cout << lca(2, 4) << "\n"; //0
76    return 0;
77  }
```

3.7.2   Segment Trees

```
1   /*
2
3   3.7.2 - Segment Trees for Lowest Common Ancestor
4
5   Description: Given a rooted tree, the lowest common ancestor (LCA)
6   of two nodes v and w is the lowest (i.e. deepest) node that has
7   both v and w as descendants, where we define each node to be a
8   descendant of itself (so if v has a direct connection from w, w
9   is the lowest common ancestor). This problem can be reduced to the
10  range minimum query problem using Eulerian tours.
11
12  Time Complexity: O(N log N) for build() and O(log N) for lca(),
13  where N is the number of nodes in the tree.
14
15  Space Complexity: O(N log N).
16
17  */
18
19  #include <algorithm> /* std::fill(), std::min(), std::max() */
20  #include <vector>
21
22  const int MAXN = 1000;
23  int len, counter;
24  int depth[MAXN], dfs_order[2*MAXN], first[MAXN], minpos[8*MAXN];
25  std::vector<int> adj[MAXN];
26
27  void dfs(int u, int d) {
28    depth[u] = d;
29    dfs_order[counter++] = u;
30    for (int j = 0, v; j < (int)adj[u].size(); j++) {
31      if (depth[v = adj[u][j]] == -1) {
32        dfs(v, d + 1);
33        dfs_order[counter++] = u;
34      }
35    }
36  }
```

```
37
38  void build_tree(int n, int l, int h) {
39    if (l == h) {
40      minpos[n] = dfs_order[l];
41      return;
42    }
43    int lchild = 2 * n + 1, rchild = 2 * n + 2;
44    build_tree(lchild, l, (l + h)/2);
45    build_tree(rchild, (l + h) / 2 + 1, h);
46    minpos[n] = depth[minpos[lchild]] < depth[minpos[rchild]] ?
47               minpos[lchild] : minpos[rchild];
48  }
49
50  void build(int nodes, int root) {
51    std::fill(depth, depth + nodes, -1);
52    std::fill(first, first + nodes, -1);
53    len = 2*nodes - 1;
54    counter = 0;
55    dfs(root, 0);
56    build_tree(0, 0, len - 1);
57    for (int i = 0; i < len; i++)
58      if (first[dfs_order[i]] == -1)
59        first[dfs_order[i]] = i;
60  }
61
62  int get_minpos(int a, int b, int n, int l, int h) {
63    if (a == l && b == h) return minpos[n];
64    int mid = (l + h) >> 1;
65    if (a <= mid && b > mid) {
66      int p1 = get_minpos(a, std::min(b, mid), 2 * n + 1, l, mid);
67      int p2 = get_minpos(std::max(a, mid + 1), b, 2 * n + 2, mid + 1, h);
68      return depth[p1] < depth[p2] ? p1 : p2;
69    }
70    if (a <= mid) return get_minpos(a, std::min(b, mid), 2 * n + 1, l, mid);
71    return get_minpos(std::max(a, mid + 1), b, 2 * n + 2, mid + 1, h);
72  }
73
74  int lca(int a, int b) {
75    return get_minpos(std::min(first[a], first[b]),
76                      std::max(first[a], first[b]), 0, 0, len - 1);
77  }
78
79  /*** Example Usage ***/
80
81  #include <iostream>
82  using namespace std;
83
84  int main() {
85    adj[0].push_back(1);
86    adj[1].push_back(0);
87    adj[1].push_back(2);
88    adj[2].push_back(1);
89    adj[3].push_back(1);
90    adj[1].push_back(3);
91    adj[0].push_back(4);
92    adj[4].push_back(0);
93    build(5, 0);
94    cout << lca(3, 2) << "\n"; //1
95    cout << lca(2, 4) << "\n"; //0
```

```
96     return 0;
97  }
```

Chapter 4

Mathematics

4.1   Mathematics Toolbox

```
1   /*
2
3   4.1 - Mathematics Toolbox
4
5   Useful math definitions. Excludes geometry (see next chapter).
6
7   */
8
9   #include <algorithm> /* std::reverse() */
10  #include <cfloat>    /* DBL_MAX */
11  #include <cmath>     /* a lot of things */
12  #include <string>
13  #include <vector>
14
15  /* Definitions for Common Floating Point Constants */
16
17  const double PI = acos(-1.0), E = exp(1.0), root2 = sqrt(2.0);
18  const double phi = (1.0 + sqrt(5.0)) / 2.0; //golden ratio
19
20  //Sketchy but working defintions of +infinity, -infinity and quiet NaN
21  //A better way is using functions of std::numeric_limits<T> from <limits>
22  //See main() for identities involving the following special values.
23  const double posinf = 1.0 / 0.0, neginf = -1.0 / 0.0, NaN = -(0.0 / 0.0);
24
25  /*
26
27  Epsilon Comparisons
28
29  The range of values for which X compares EQ() to is [X - eps, X + eps].
30  For values to compare LT() and GT() x, they must fall outside of the range.
31
32  e.g. if eps = 1e-7, then EQ(1e-8, 2e-8) is true and LT(1e-8, 2e-8) is false.
33
34  */
35
36  const double eps = 1e-7;
37  #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
```

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

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

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

```
215      } while (fabs(q1 - q2) / q2 > rel_error);
216      return 0.564189583547756287 * exp(-x * x) * q2; //0.564 ~ 1/sqrt(pi)
217    }
218
219    /*
220
221    Gamma and Log-Gamma Functions (tgamma() and lgamma() in C++11)
222    Warning: unlike the actual standard C++ versions, the following
223    function only works on positive numbers (returns NaN if x <= 0).
224    Adapted from: http://www.johndcook.com/blog/cpp_gamma/
225
226    */
227
228    double lgamma(double x);
229
230    double tgamma(double x) {
231      if (x <= 0.0) return NaN;
232      static const double gamma = 0.577215664901532860606512090;
233      if (x < 1e-3) return 1.0 / (x * (1.0 + gamma * x));
234      if (x < 12.0) {
235        double y = x;
236        int n = 0;
237        bool arg_was_less_than_one = (y < 1.0);
238        if (arg_was_less_than_one) y += 1.0;
239        else y -= (n = static_cast<int>(floor(y)) - 1);
240        static const double p[] = {
241          -1.71618513886549492533811E+0, 2.47656508055759199108314E+1,
242          -3.79804256470945635097577E+2, 6.29331155312818442661052E+2,
243           8.66966202790413211295064E+2,-3.14512729688483675254357E+4,
244          -3.61444134186911729807069E+4, 6.64561438202405440627855E+4
245        };
246        static const double q[] = {
247          -3.08402300119738975254353E+1, 3.15350626979604161529144E+2,
248          -1.01515636749021914166146E+3,-3.10777167157231109440444E+3,
249           2.25381184209801510330112E+4, 4.75584627752788110767815E+3,
250          -1.34659959864969306392456E+5,-1.15132259675553483497211E+5
251        };
252        double num = 0.0, den = 1.0, z = y - 1;
253        for (int i = 0; i < 8; i++) {
254          num = (num + p[i]) * z;
255          den = den * z + q[i];
256        }
257        double result = num / den + 1.0;
258        if (arg_was_less_than_one) result /= (y - 1.0);
259        else for (int i = 0; i < n; i++) result *= y++;
260        return result;
261      }
262      return (x > 171.624) ? DBL_MAX * 2.0 : exp(lgamma(x));
263    }
264
265    double lgamma(double x) {
266      if (x <= 0.0) return NaN;
267      if (x < 12.0) return log(fabs(tgamma(x)));
268      static const double c[8] = {
269        1.0/12.0, -1.0/360.0, 1.0/1260.0, -1.0/1680.0, 1.0/1188.0,
270        -691.0/360360.0, 1.0/156.0, -3617.0/122400.0
271      };
272      double z = 1.0 / (x * x), sum = c[7];
273      for (int i = 6; i >= 0; i--) sum = sum * z + c[i];
```

```
274    static const double halflog2pi = 0.9189385332046727417803297364056;
275    return (x - 0.5) * log(x) - x + halflog2pi + sum / x;
276  }
277
278  /*
279
280  Base Conversion - O(N) on the number of digits
281
282  Given the digits of an integer x in base a, returns x's digits in base b.
283  Precondition: the base-10 value of x must be able to fit within an unsigned
284  long long. In other words, the value of x must be between 0 and 2^64 - 1.
285
286  Note: vector[0] stores the most significant digit in all usages below.
287
288  e.g. if x = {1, 2, 3} and a = 5 (i.e. x = 123 in base 5 = 38 in base 10),
289  then convert_base(x, 5, 3) returns {1, 1, 0, 2} (1102 in base 2).
290
291  */
292
293  std::vector<int> convert_base(const std::vector<int> & x, int a, int b) {
294    unsigned long long base10 = 0;
295    for (int i = 0; i < (int)x.size(); i++)
296      base10 += x[i] * pow(a, x.size() - i - 1);
297    int N = ceil(log(base10 + 1) / log(b));
298    std::vector<int> baseb;
299    for (int i = 1; i <= N; i++)
300      baseb.push_back(int(base10 / pow(b, N - i)) % b);
301    return baseb;
302  }
303
304  //returns digits of a number in base b
305  std::vector<int> base_digits(int x, int b = 10) {
306    std::vector<int> baseb;
307    while (x != 0) {
308      baseb.push_back(x % b);
309      x /= b;
310    }
311    std::reverse(baseb.begin(), baseb.end());
312    return baseb;
313  }
314
315  /*
316
317  Integer to Roman Numerals Conversion
318
319  Given an integer x, this function returns the Roman numeral representation
320  of x as a C++ string. More 'M's are appended to the front of the resulting
321  string if x is greater than 1000. e.g. to_roman(1234) returns "MCCXXXIV"
322  and to_roman(5678) returns "MMMMMDCLXXVIII".
323
324  */
325
326  std::string to_roman(unsigned int x) {
327    static std::string h[] = {"","C","CC","CCC","CD","D","DC","DCC","DCCC","CM"};
328    static std::string t[] = {"","X","XX","XXX","XL","L","LX","LXX","LXXX","XC"};
329    static std::string o[] = {"","I","II","III","IV","V","VI","VII","VIII","IX"};
330    std::string res(x / 1000, 'M');
331    x %= 1000;
332    return res + h[x / 100] + t[x / 10 % 10] + o[x % 10];
```

```
333  }
334
335  /*** Example Usage ***/
336
337  #include <algorithm>
338  #include <cassert>
339  #include <iostream>
340  using namespace std;
341
342  int main() {
343      cout << "PI:␣" << PI << "\n";
344      cout << "E:␣" << E << "\n";
345      cout << "sqrt(2):␣" << root2 << "\n";
346      cout << "Golden␣ratio:␣" << phi << "\n";
347
348      //some properties of posinf, neginf, and NaN:
349      double x = -1234.567890; //any normal value of x will work
350      assert((posinf > x) && (neginf < x) && (posinf == -neginf));
351      assert((posinf + x == posinf) && (posinf - x == posinf));
352      assert((neginf + x == neginf) && (neginf - x == neginf));
353      assert((posinf + posinf == posinf) && (neginf - posinf == neginf));
354      assert((NaN != x) && (NaN != NaN) && (NaN != posinf) && (NaN != neginf));
355      assert(!(NaN < x) && !(NaN > x) && !(NaN <= x) && !(NaN >= x));
356      assert(isnan(0.0*posinf) && isnan(0.0*neginf) && isnan(posinf/neginf));
357      assert(isnan(NaN) && isnan(-NaN) && isnan(NaN*x + x - x/-NaN));
358      assert(isnan(neginf-neginf) && isnan(posinf-posinf) && isnan(posinf+neginf));
359      assert(!signbit(NaN) && signbit(-NaN) && !signbit(posinf) && signbit(neginf));
360
361      assert(copysign(1.0, +2.0) == +1.0 && copysign(posinf, -2.0) == neginf);
362      assert(copysign(1.0, -2.0) == -1.0 && signbit(copysign(NaN, -2.0)));
363      assert(sgn(-1.234) == -1 && sgn(0.0) == 0 && sgn(5678) == 1);
364
365      assert(EQ(floor0(1.5), 1.0) && EQ(floor0(-1.5), -1.0));
366      assert(EQ(ceil0(1.5), 2.0) && EQ(ceil0(-1.5), -2.0));
367      assert(EQ(roundhalfup(1.5), 2.0) && EQ(roundhalfup(-1.5), -1.0));
368      assert(EQ(roundhalfdown(1.5), 1.0) && EQ(roundhalfdown(-1.5), -2.0));
369      assert(EQ(roundhalfup0(1.5), 2.0) && EQ(roundhalfup0(-1.5), -2.0));
370      assert(EQ(roundhalfdown0(1.5), 1.0) && EQ(roundhalfdown0(-1.5), -1.0));
371      assert(EQ(roundhalfeven(1.5), 2.0) && EQ(roundhalfeven(-1.5), -2.0));
372      assert(NE(roundalternate(1.5), roundalternate(1.5)));
373      assert(EQ(roundplaces(-1.23456, 3, roundhalfdown0<double>), -1.235));
374
375      assert(EQ(erf(1.0), 0.8427007929) && EQ(erf(-1.0), -0.8427007929));
376      assert(EQ(tgamma(0.5), 1.7724538509) && EQ(tgamma(1.0), 1.0));
377      assert(EQ(lgamma(0.5), 0.5723649429) && EQ(lgamma(1.0), 0.0));
378
379      int base10digs[] = {1, 2, 3, 4, 5, 6}, a = 20, b = 10;
380      vector<int> basea = base_digits(123456, a);
381      vector<int> baseb = convert_base(basea, a, b);
382      assert(equal(baseb.begin(), baseb.end(), base10digs));
383
384      assert(to_roman(1234) == "MCCXXXIV");
385      assert(to_roman(5678) == "MMMMMDCLXXVIII");
386      return 0;
387  }
```

4.2   Combinatorics

4.2.1   Combinatorial Calculations

```
1   /*
2
3   4.2.1 - Combinatorial Calculations
4
5   The meanings of the following functions can respectively be
6   found with quick searches online. All of them computes the
7   answer modulo m, since contest problems typically ask us for
8   this due to the actual answer being potentially very large.
9   All functions using tables to generate every answer below
10  n and k can be optimized using recursion and memoization.
11
12  Note: The following are only defined for nonnegative inputs.
13
14  */
15
16  #include <vector>
17
18  typedef std::vector<std::vector<long long> > table;
19
20  //n! mod m in O(n)
21  long long factorial(int n, int m = 1000000007) {
22    long long res = 1;
23    for (int i = 2; i <= n; i++) res = (res * i) % m;
24    return res % m;
25  }
26
27  //n! mod p, where p is a prime number, in O(p log n)
28  long long factorialp(long long n, long long p = 1000000007) {
29    long long res = 1, h;
30    while (n > 1) {
31      res = (res * ((n / p) % 2 == 1 ? p - 1 : 1)) % p;
32      h = n % p;
33      for (int i = 2; i <= h; i++) res = (res * i) % p;
34      n /= p;
35    }
36    return res % p;
37  }
38
39  //first n rows of pascal's triangle (mod m) in O(n^2)
40  table binomial_table(int n, long long m = 1000000007) {
41    table t(n + 1);
42    for (int i = 0; i <= n; i++)
43      for (int j = 0; j <= i; j++)
44        if (i < 2 || j == 0 || i == j)
45          t[i].push_back(1);
46        else
47          t[i].push_back((t[i - 1][j - 1] + t[i - 1][j]) % m);
48    return t;
49  }
50
51  //if the product of two 64-bit ints (a*a, a*b, or b*b) can
52  //overflow, you must use mulmod (multiplication by adding)
53  long long powmod(long long a, long long b, long long m) {
```

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

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

```
172  #include <iostream>
173  using namespace std;
174
175  int main() {
176    table t = binomial_table(10);
177    for (int i = 0; i < (int)t.size(); i++) {
178      for (int j = 0; j < (int)t[i].size(); j++)
179        cout << t[i][j] << "␣";
180      cout << "\n";
181    }
182    assert(factorial(10)      == 3628800);
183    assert(factorialp(123456) == 639390503);
184    assert(choose(20, 7)      == 77520);
185    assert(multichoose(20, 7) == 657800);
186    assert(permute(10, 4)     == 5040);
187    assert(partitions(4)      == 5);
188    assert(partitions(100, 5) == 38225);
189    assert(stirling1(4, 2)    == 11);
190    assert(stirling2(4, 3)    == 6);
191    assert(eulerian1(9, 5)    == 88234);
192    assert(eulerian2(8, 3)    == 195800);
193    assert(catalan(10)        == 16796);
194    return 0;
195  }
```

4.2.2   Enumerating Arrangements

```
1   /*
2
3   4.2.2 - Enumerating Arrangements
4
5   We shall consider an arrangement to be a permutation of
6   all the integers from 0 to n - 1. For our purposes, the
7   difference between an arrangement and a permutation is
8   simply that a permutation can pertain to a set of any
9   given values, not just distinct integers from 0 to n-1.
10
11  */
12
13  #include <algorithm> /* std::copy(), std::fill() */
14  #include <vector>
15
16  /*
17
18  Changes a[] to the next lexicographically greater
19  permutation of any k distinct integers in range [0, n).
20  The values of a[] that's passed should be k distinct
21  integers, each in range [0, n).
22
23  returns: whether the function could rearrange a[] to
24  a lexicographically greater arrangement.
25
26  examples:
27  next_arrangement(4, 3, {0, 1, 2}) => 1,  a[] = {0, 1, 3}
28  next_arrangement(4, 3, {0, 1, 3}) => 1,  a[] = {0, 2, 1}
29  next_arrangement(4, 3, {3, 2, 1}) => 0,  a[] unchanged
30
```

```
31   */
32
33   bool next_arrangement(int n, int k, int a[]) {
34     std::vector<bool> used(n);
35     for (int i = 0; i < k; i++) used[a[i]] = true;
36     for (int i = k - 1; i >= 0; i--) {
37       used[a[i]] = false;
38       for (int j = a[i] + 1; j < n; j++) {
39         if (!used[j]) {
40           a[i++] = j;
41           used[j] = true;
42           for (int x = 0; i < k; x++)
43             if (!used[x]) a[i++] = x;
44           return true;
45         }
46       }
47     }
48     return false;
49   }
50
51   /*
52
53   Computes n permute k using formula: nPk = n!/(n - k)!
54   Complexity: O(k).  E.g. n_permute_k(10, 7) = 604800
55
56   */
57
58   long long n_permute_k(int n, int k) {
59     long long res = 1;
60     for (int i = 0; i < k; i++) res *= n - i;
61     return res;
62   }
63
64   /*
65
66   Given an integer rank x in range [0, n permute k), returns
67   a vector of integers representing the x-th lexicographically
68   smallest permutation of any k distinct integers in [0, n).
69
70   examples: arrangement_by_rank(4, 3, 0) => {0, 1, 2}
71             arrangement_by_rank(4, 3, 5) => {0, 3, 2}
72
73   */
74
75   std::vector<int> arrangement_by_rank(int n, int k, long long x) {
76     std::vector<int> free(n), res(k);
77     for (int i = 0; i < n; i++) free[i] = i;
78     for (int i = 0; i < k; i++) {
79       long long cnt = n_permute_k(n - 1 - i, k - 1 - i);
80       int pos = (int)(x / cnt);
81       res[i] = free[pos];
82       std::copy(free.begin() + pos + 1, free.end(),
83                 free.begin() + pos);
84       x %= cnt;
85     }
86     return res;
87   }
88
89   /*
```

```
90
91   Given an array a[] of k integers each in range [0, n), returns
92   the (0-based) lexicographical rank (counting from least to
93   greatest) of the arrangement specified by a[] in all possible
94   permutations of k distinct integers in range [0, n).
95
96   examples: rank_by_arrangement(4, 3, {0, 1, 2}) => 0
97             rank_by_arrangement(4, 3, {0, 3, 2}) => 5
98
99   */
100
101  long long rank_by_arrangement(int n, int k, int a[]) {
102    long long res = 0;
103    std::vector<bool> used(n);
104    for (int i = 0; i < k; i++) {
105      int cnt = 0;
106      for (int j = 0; j < a[i]; j++)
107        if (!used[j]) cnt++;
108      res += n_permute_k(n - i - 1, k - i - 1) * cnt;
109      used[a[i]] = true;
110    }
111    return res;
112  }
113
114  /*
115
116  Changes a[] to the next lexicographically greater
117  permutation of k (not-necessarily distinct) integers in
118  range [0, n). The values of a[] should be in range [0, n).
119  If a[] was interpreted as a base-n integer that is k digits
120  long, this function would be equivalent to incrementing a.
121  Ergo, there are n^k arrangements if repeats are allowed.
122
123  returns: whether the function could rearrange a[] to a
124  lexicographically greater arrangement with repeats.
125
126  examples:
127  n_a_w_r(4, 3, {0, 0, 0}) => 1,   a[] = {0, 0, 1}
128  n_a_w_r(4, 3, {0, 1, 3}) => 1,   a[] = {0, 2, 0}
129  n_a_w_r(4, 3, {3, 3, 3}) => 0,   a[] unchanged
130
131  */
132
133  bool next_arrangement_with_repeats(int n, int k, int a[]) {
134    for (int i = k - 1; i >= 0; i--) {
135      if (a[i] < n - 1) {
136        a[i]++;
137        std::fill(a + i + 1, a + k, 0);
138        return true;
139      }
140    }
141    return false;
142  }
143
144  /*** Example Usage ***/
145
146  #include <cassert>
147  #include <iostream>
148  using namespace std;
```

```
149
150   template<class it> void print(it lo, it hi) {
151     for (; lo != hi; ++lo) cout << *lo << "␣";
152     cout << "\n";
153   }
154
155   int main() {
156     {
157       int n = 4, k = 3, a[] = {0, 1, 2};
158       cout << n << "␣permute␣" << k << "␣arrangements:\n";
159       int cnt = 0;
160       do {
161         print(a, a + k);
162         vector<int> b = arrangement_by_rank(n, k, cnt);
163         assert(equal(a, a + k, b.begin()));
164         assert(rank_by_arrangement(n, k, a) == cnt);
165         cnt++;
166       } while (next_arrangement(n, k, a));
167       cout << "\n";
168     }
169
170     {
171       int n = 4, k = 2, a[] = {0, 0};
172       cout << n << "^" << k << "␣arrangements␣with␣repeats:\n";
173       do {
174         print(a, a + k);
175       } while (next_arrangement_with_repeats(n, k, a));
176     }
177     return 0;
178   }
```

4.2.3   Enumerating Permutations

```
1    /*
2
3    4.2.3 - Enumerating Permutations
4
5    We shall consider a permutation of n objects to be an
6    ordered list of size n that contains all n elements,
7    where order is important. E.g. 1 1 2 0 and 0 1 2 1
8    are considered two different permutations of 0 1 1 2.
9    Compared to our prior definition of an arrangement, a
10   permutable range of size n may contain repeated values
11   of any type, not just the integers from 0 to n - 1.
12
13   */
14
15   #include <algorithm> /* copy, iter_swap, reverse, swap */
16   #include <vector>
17
18   //identical to std::next_permutation()
19   template<class It> bool _next_permutation(It lo, It hi) {
20     if (lo == hi) return false;
21     It i = lo;
22     if (++i == hi) return false;
23     i = hi; --i;
24     for (;;) {
```

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

```
84   }
85
86   /*
87
88   Finds the next lexicographically greater permutation of
89   the binary digits of x. In other words, next_permutation()
90   simply returns the smallest integer greater than x which
91   has the same number of 1 bits (i.e. same popcount) as x.
92
93   examples: next_permutation(10101 base 2) =  10110
94             next_permutation(11100 base 2) = 100011
95
96   This can also be used to generate combinations as follows:
97   If we let k = popcount(x), then we can use this to generate
98   all possible masks to tell us which k items to take out of
99   n total items (represented by the first n bits of x).
100
101  */
102
103  long long next_permutation(long long x) {
104    long long s = x & -x, r = x + s;
105    return r | (((x ^ r) >> 2) / s);
106  }
107
108  /*
109
110  Given an integer rank x in range [0, n!), returns a vector
111  of integers representing the x-th lexicographically smallest
112  permutation of the integers in [0, n).
113
114  examples: permutation_by_rank(4, 0) => {0, 1, 2, 3}
115            permutation_by_rank(4, 5) => {0, 3, 2, 1}
116
117  */
118
119  std::vector<int> permutation_by_rank(int n, long long x) {
120    long long fact[n];
121    fact[0] = 1;
122    for (int i = 1; i < n; i++)
123      fact[i] = i * fact[i - 1];
124    std::vector<int> free(n), res(n);
125    for (int i = 0; i < n; i++) free[i] = i;
126    for (int i = 0; i < n; i++) {
127      int pos = x / fact[n - 1 - i];
128      res[i] = free[pos];
129      std::copy(free.begin() + pos + 1, free.end(),
130                free.begin() + pos);
131      x %= fact[n - 1 - i];
132    }
133    return res;
134  }
135
136  /*
137
138  Given an array a[] of n integers each in range [0, n), returns
139  the (0-based) lexicographical rank (counting from least to
140  greatest) of the arrangement specified by a[] in all possible
141  permutations of the integers from 0 to n - 1.
142
```

```
143  examples: rank_by_permutation(3, {0, 1, 2}) => 0
144           rank_by_permutation(3, {2, 1, 0}) => 5
145
146  */
147
148  template<class T> long long rank_by_permutation(int n, T a[]) {
149    long long fact[n];
150    fact[0] = 1;
151    for (int i = 1; i < n; i++)
152      fact[i] = i * fact[i - 1];
153    long long res = 0;
154    for (int i = 0; i < n; i++) {
155      int v = a[i];
156      for (int j = 0; j < i; j++)
157        if (a[j] < a[i]) v--;
158      res += v * fact[n - 1 - i];
159    }
160    return res;
161  }
162
163  /*
164
165  Given a permutation a[] of the integers from 0 to n - 1,
166  returns a decomposition of the permutation into cycles.
167  A permutation cycle is a subset of a permutation whose
168  elements trade places with one another. For example, the
169  permutation {0, 2, 1, 3} decomposes to {0, 3, 2} and {1}.
170  Here, the notation {0, 3, 2} means that starting from the
171  original ordering {0, 1, 2, 3}, the 0th value is replaced
172  by the 3rd, the 3rd by the 2nd, and the 2nd by the first,
173  See: http://mathworld.wolfram.com/PermutationCycle.html
174
175  */
176
177  typedef std::vector<std::vector<int> > cycles;
178
179  cycles decompose_into_cycles(int n, int a[]) {
180    std::vector<bool> vis(n);
181    cycles res;
182    for (int i = 0; i < n; i++) {
183      if (vis[i]) continue;
184      int j = i;
185      std::vector<int> cur;
186      do {
187        cur.push_back(j);
188        vis[j] = true;
189        j = a[j];
190      } while (j != i);
191      res.push_back(cur);
192    }
193    return res;
194  }
195
196  /*** Example Usage ***/
197
198  #include <bitset>
199  #include <cassert>
200  #include <iostream>
201  using namespace std;
```

```
202
203  void printperm(const vector<int> & perm) {
204    for (int i = 0; i < (int)perm.size(); i++)
205      cout << perm[i] << "␣";
206    cout << "\n";
207  }
208
209  template<class it> void print(it lo, it hi) {
210    for (; lo != hi; ++lo) cout << *lo << "␣";
211    cout << "\n";
212  }
213
214  int main() {
215    { //method 1: ordered
216      int n = 4, a[] = {0, 1, 2, 3};
217      int b[n], c[n];
218      for (int i = 0; i < n; i++) b[i] = c[i] = a[i];
219      cout << "Ordered␣permutations␣of␣0␣to␣" << n-1 << ":\n";
220      int cnt = 0;
221      do {
222        print(a, a + n);
223        assert(equal(b, b + n, a));
224        assert(equal(c, c + n, a));
225        vector<int> d = permutation_by_rank(n, cnt);
226        assert(equal(d.begin(), d.end(), a));
227        assert(rank_by_permutation(n, a) == cnt);
228        cnt++;
229        std::next_permutation(b, b + n);
230        _next_permutation(c, c + n);
231      } while (next_permutation(n, a));
232      cout << "\n";
233    }
234
235    { //method 2: unordered
236      int n = 3;
237      cout << "Unordered␣permutations␣of␣0␣to␣" << n-1 << ":\n";
238      gen_permutations(n, printperm);
239      cout << "\n";
240    }
241
242    { //permuting binary digits
243      const int n = 5;
244      cout << "Permutations␣of␣2␣zeros␣and␣3␣ones:\n";
245      long long lo = 7;  // 00111 in base 2
246      long long hi = 35; //100011 in base 2
247      do {
248        cout << bitset<n>(lo).to_string() << "\n";
249      } while ((lo = next_permutation(lo)) != hi);
250      cout << "\n";
251    }
252
253    { //permutation cycles
254      int n = 4, a[] = {3, 1, 0, 2};
255      cout << "Decompose␣0␣2␣1␣3␣into␣cycles:\n";
256      cycles c = decompose_into_cycles(n, a);
257      for (int i = 0; i < (int)c.size(); i++) {
258        cout << "Cycle␣" << i + 1 << ":";
259        for (int j = 0; j < (int)c[i].size(); j++)
260          cout << "␣" << c[i][j];
```

```
261        cout << "\n";
262      }
263    }
264    return 0;
265 }
```

4.2.4   Enumerating Combinations

```
1  /*
2
3  4.2.4 - Enumerating Combinations
4
5  We shall consider a combination n choose k to be an
6  set of k elements chosen from a total of n elements.
7  Unlike n permute k, the order here doesn't matter.
8  That is, 0 1 2 is considered the same as 0 2 1, so
9  we will consider the sorted representation of each
10 combination for purposes of the functions below.
11
12 */
13
14 #include <algorithm> /* iter_swap, rotate, swap, swap_ranges */
15 #include <iterator>  /* std::iterator_traits */
16 #include <vector>
17
18 /*
19
20 Rearranges the values in the range [lo, hi) such that
21 elements in the range [lo, mid) becomes the next
22 lexicographically greater combination of the values from
23 [lo, hi) than it currently is, and returns whether the
24 function could rearrange [lo, hi) to a lexicographically
25 greater combination. If the range [lo, hi) contains n
26 elements and the range [lo, mid) contains k elements,
27 then starting off with a sorted range [lo, hi) and
28 calling next_combination() repeatedly will return true
29 for n choose k iterations before returning false.
30
31 */
32
33 template<class It>
34 bool next_combination(It lo, It mid, It hi) {
35   if (lo == mid || mid == hi) return false;
36   It l(mid - 1), h(hi - 1);
37   int sz1 = 1, sz2 = 1;
38   while (l != lo && !(*l < *h)) --l, ++sz1;
39   if (l == lo && !(*l < *h)) {
40     std::rotate(lo, mid, hi);
41     return false;
42   }
43   for (; mid < h; ++sz2) if (!(*l < *--h)) { ++h; break; }
44   if (sz1 == 1 || sz2 == 1) {
45     std::iter_swap(l, h);
46   } else if (sz1 == sz2) {
47     std::swap_ranges(l, mid, h);
48   } else {
49     std::iter_swap(l, h);
```

```
50        ++l; ++h; --sz1; --sz2;
51        int total = sz1 + sz2, gcd = total;
52        for (int i = sz1; i != 0; ) std::swap(gcd %= i, i);
53        int skip = total / gcd - 1;
54        for (int i = 0; i < gcd; i++) {
55          It curr(i < sz1 ? l + i : h + (i - sz1));
56          int k = i;
57          typename std::iterator_traits<It>::value_type v(*curr);
58          for (int j = 0; j < skip; j++) {
59            k = (k + sz1) % total;
60            It next(k < sz1 ? l + k : h + (k - sz1));
61            *curr = *next;
62            curr = next;
63          }
64          *curr = v;
65        }
66      }
67      return true;
68    }
69
70    /*
71
72    Changes a[] to the next lexicographically greater
73    combination of any k distinct integers in range [0, n).
74    The values of a[] that's passed should be k distinct
75    integers, each in range [0, n).
76
77    */
78
79    bool next_combination(int n, int k, int a[]) {
80      for (int i = k - 1; i >= 0; i--) {
81        if (a[i] < n - k + i) {
82          for (++a[i]; ++i < k; ) a[i] = a[i - 1] + 1;
83          return true;
84        }
85      }
86      return false;
87    }
88
89    /*
90
91    Finds the "mask" of the next combination of x. This is
92    equivalent to the next lexicographically greater permutation
93    of the binary digits of x. In other words, the function
94    simply returns the smallest integer greater than x which
95    has the same number of 1 bits (i.e. same popcount) as x.
96
97    examples: next_combination_mask(10101 base 2) =  10110
98              next_combination_mask(11100 base 2) = 100011
99
100   If we arbitrarily number the n items of our collection from
101   0 to n-1, then generating all combinations n choose k can
102   be done as follows: initialize x such that popcount(x) = k
103   and the first (least-significant) k bits are all set to 1
104   (e.g. to do 5 choose 3, start at x = 00111 (base 2) = 7).
105   Then, we repeatedly call x = next_combination_mask(x) until
106   we reach 11100 (the lexicographically greatest mask for 5
107   choose 3), after which we stop. At any point in the process,
108   we can say that the i-th item is being "taken" (0 <= i < n)
```

```
109   iff the i-th bit of x is set.
110
111   Note: this does not produce combinations in the same order
112   as next_combination, nor does it work if your n items have
113   repeated values (in that case, repeated combos will be
114   generated).
115
116   */
117
118   long long next_combination_mask(long long x) {
119     long long s = x & -x, r = x + s;
120     return r | (((x ^ r) >> 2) / s);
121   }
122
123   //n choose k in O(min(k, n - k))
124   long long n_choose_k(long long n, long long k) {
125     if (k > n - k) k = n - k;
126     long long res = 1;
127     for (int i = 0; i < k; i++)
128       res = res * (n - i) / (i + 1);
129     return res;
130   }
131
132   /*
133
134   Given an integer rank x in range [0, n choose k), returns
135   a vector of integers representing the x-th lexicographically
136   smallest combination k distinct integers in [0, n).
137
138   examples: combination_by_rank(4, 3, 0) => {0, 1, 2}
139             combination_by_rank(4, 3, 2) => {0, 2, 3}
140
141   */
142
143   std::vector<int> combination_by_rank(int n, int k, long long x) {
144     std::vector<int> res(k);
145     int cnt = n;
146     for (int i = 0; i < k; i++) {
147       int j = 1;
148       for (;; j++) {
149         long long am = n_choose_k(cnt - j, k - 1 - i);
150         if (x < am) break;
151         x -= am;
152       }
153       res[i] = i > 0 ? (res[i - 1] + j) : (j - 1);
154       cnt -= j;
155     }
156     return res;
157   }
158
159   /*
160
161   Given an array a[] of k integers each in range [0, n), returns
162   the (0-based) lexicographical rank (counting from least to
163   greatest) of the combination specified by a[] in all possible
164   combination of k distinct integers in range [0, n).
165
166   examples: rank_by_combination(4, 3, {0, 1, 2}) => 0
167             rank_by_combination(4, 3, {0, 2, 3}) => 2
```

```
168
169   */
170
171   long long rank_by_combination(int n, int k, int a[]) {
172     long long res = 0;
173     int prev = -1;
174     for (int i = 0; i < k; i++) {
175       for (int j = prev + 1; j < a[i]; j++)
176         res += n_choose_k(n - 1 - j, k - 1 - i);
177       prev = a[i];
178     }
179     return res;
180   }
181
182   /*
183
184   Changes a[] to the next lexicographically greater
185   combination of any k (not necessarily distinct) integers
186   in range [0, n). The values of a[] that's passed should
187   be k integers, each in range [0, n). Note that there are
188   a total of n multichoose k combinations with repetition,
189   where n multichoose k = (n + k - 1) choose k
190
191   */
192
193   bool next_combination_with_repeats(int n, int k, int a[]) {
194     for (int i = k - 1; i >= 0; i--) {
195       if (a[i] < n - 1) {
196         for (++a[i]; ++i < k; ) a[i] = a[i - 1];
197         return true;
198       }
199     }
200     return false;
201   }
202
203   /*** Example Usage ***/
204
205   #include <cassert>
206   #include <iostream>
207   using namespace std;
208
209   template<class it> void print(it lo, it hi) {
210     for (; lo != hi; ++lo) cout << *lo << "␣";
211     cout << "\n";
212   }
213
214   int main() {
215     { //like std::next_permutation(), repeats in the range allowed
216       int k = 3;
217       string s = "11234";
218       cout << s << "␣choose␣" << k << ":\n";
219       do {
220         cout << s.substr(0, k) << "\n";
221       } while (next_combination(s.begin(), s.begin() + k, s.end()));
222       cout << "\n";
223     }
224
225     { //unordered combinations with masks
226       int n = 5, k = 3;
```

```
227        string s = "abcde"; //must be distinct values
228        cout << s << "⊔choose⊔" << k << "⊔with⊔masks:\n";
229        long long mask = 0, dest = 0;
230        for (int i = 0; i < k; i++) mask |= 1 << i;
231        for (int i = k - 1; i < n; i++) dest |= 1 << i;
232        do {
233          for (int i = 0; i < n; i++)
234            if ((mask >> i) & 1) cout << s[i];
235          cout << "\n";
236          mask = next_combination_mask(mask);
237        } while (mask != dest);
238        cout << "\n";
239      }
240
241      { //only combinations of distinct integers from 0 to n - 1
242        int n = 5, k = 3, a[] = {0, 1, 2};
243        cout << n << "⊔choose⊔" << k << ":\n";
244        int cnt = 0;
245        do {
246          print(a, a + k);
247          vector<int> b = combination_by_rank(n, k, cnt);
248          assert(equal(a, a + k, b.begin()));
249          assert(rank_by_combination(n, k, a) == cnt);
250          cnt++;
251        } while (next_combination(n, k, a));
252        cout << "\n";
253      }
254
255      { //combinations with repetition
256        int n = 3, k = 2, a[] = {0, 0};
257        cout << n << "⊔multichoose⊔" << k << ":\n";
258        do {
259          print(a, a + k);
260        } while (next_combination_with_repeats(n, k, a));
261      }
262      return 0;
263    }
```

4.2.5   Enumerating Partitions

```
1   /*
2
3   4.2.5 - Enumerating Partitions
4
5   We shall consider a partition of an integer n to be an
6   unordered multiset of positive integers that has a total
7   sum equal to n. Since both 2 1 1 and 1 2 1 represent the
8   same partition of 4, we shall consider only descending
9   sorted lists as "valid" partitions for functions below.
10
11  */
12
13  #include <vector>
14
15  /*
16
17  Given a vector representing a partition of some
```

```
18   integer n (the sum of all values in the vector),
19   changes p to the next lexicographically greater
20   partition of n and returns whether the change was
21   successful (whether a lexicographically greater
22   partition existed). Note that the "initial" value
23   of p must be a vector of size n, all initialized 1.
24
25   e.g. next_partition({2, 1, 1}) => 1, p becomes {2, 2}
26        next_partition({2, 2})    => 1, p becomes {3, 1}
27        next_partition({4})       => 0, p is unchanged
28
29   */
30
31   bool next_partition(std::vector<int> & p) {
32     int n = p.size();
33     if (n <= 1) return false;
34     int s = p[n - 1] - 1, i = n - 2;
35     p.pop_back();
36     for (; i > 0 && p[i] == p[i - 1]; i--) {
37       s += p[i];
38       p.pop_back();
39     }
40     for (p[i]++; s-- > 0; ) p.push_back(1);
41     return true;
42   }
43
44   /* Returns the number of partitions of n. */
45
46   long long count_partitions(int n) {
47     std::vector<long long> p(n + 1, 0);
48     p[0] = 1;
49     for (int i = 1; i <= n; i++)
50       for (int j = i; j <= n; j++)
51         p[j] += p[j - i];
52     return p[n];
53   }
54
55   /* Helper function for partitioning by rank */
56
57   std::vector< std::vector<long long> >
58     p(1, std::vector<long long>(1, 1)); //memoization
59
60   long long partition_function(int a, int b) {
61     if (a >= (int)p.size()) {
62       int old = p.size();
63       p.resize(a + 1);
64       p[0].resize(a + 1);
65       for (int i = 1; i <= a; i++) {
66         p[i].resize(a + 1);
67         for (int j = old; j <= i; j++)
68           p[i][j] = p[i - 1][j - 1] + p[i - j][j];
69       }
70     }
71     return p[a][b];
72   }
73
74   /*
75
76   Given an integer n to partition and a 0-based rank x,
```

```
77   returns a vector of integers representing the x-th
78   lexicographically smallest partition of n (if values
79   in each partition were sorted in decreasing order).
80
81   examples: partition_by_rank(4, 0) => {1, 1, 1, 1}
82             partition_by_rank(4, 3) => {3, 1}
83
84   */
85
86   std::vector<int> partition_by_rank(int n, long long x) {
87     std::vector<int> res;
88     for (int i = n; i > 0; ) {
89       int j = 1;
90       for (;; j++) {
91         long long cnt = partition_function(i, j);
92         if (x < cnt) break;
93         x -= cnt;
94       }
95       res.push_back(j);
96       i -= j;
97     }
98     return res;
99   }
100
101  /*
102
103  Given a partition of an integer n (sum of all values
104  in vector p), returns a 0-based rank x of the partition
105  represented by p, considering partitions from least to
106  greatest in lexicographical order (if each partition
107  had values sorted in descending order).
108
109  examples: rank_by_partition({1, 1, 1, 1}) => 0
110            rank_by_partition({3, 1})       => 3
111
112  */
113
114  long long rank_by_partition(const std::vector<int> & p) {
115    long long res = 0;
116    int sum = 0;
117    for (int i = 0; i < (int)p.size(); i++) sum += p[i];
118    for (int i = 0; i < (int)p.size(); i++) {
119      for (int j = 0; j < p[i]; j++)
120        res += partition_function(sum, j);
121      sum -= p[i];
122    }
123    return res;
124  }
125
126  /*
127
128  Calls the custom function f(vector) on all partitions
129  which consist of strictly *increasing* integers.
130  This will exclude partitions such as {1, 1, 1, 1}.
131
132  */
133
134  template<class ReportFunction>
135  void gen_increasing_partitons(int left, int prev, int i,
```

```
136                      ReportFunction f, std::vector<int> & p) {
137    if (left == 0) {
138      //warning: slow constructor - modify accordingly
139      f(std::vector<int>(p.begin(), p.begin() + i));
140      return;
141    }
142    for (p[i] = prev + 1; p[i] <= left; p[i]++)
143      gen_increasing_partitons(left - p[i], p[i], i + 1, f, p);
144  }
145
146  template<class ReportFunction>
147  void gen_increasing_partitons(int n, ReportFunction f) {
148    std::vector<int> partitions(n, 0);
149    gen_increasing_partitons(n, 0, 0, f, partitions);
150  }
151
152  /*** Example Usage ***/
153
154  #include <cassert>
155  #include <iostream>
156  using namespace std;
157
158  void print(const vector<int> & v) {
159    for (int i = 0; i < (int)v.size(); i++)
160      cout << v[i] << " ";
161    cout << "\n";
162  }
163
164  int main() {
165    assert(count_partitions(5) == 7);
166    assert(count_partitions(20) == 627);
167    assert(count_partitions(30) == 5604);
168    assert(count_partitions(50) == 204226);
169    assert(count_partitions(100) == 190569292);
170
171    {
172      int n = 4;
173      vector<int> a(n, 1);
174      cout << "Partitions of " << n << ":\n";
175      int cnt = 0;
176      do {
177        print(a);
178        vector<int> b = partition_by_rank(n, cnt);
179        assert(equal(a.begin(), a.end(), b.begin()));
180        assert(rank_by_partition(a) == cnt);
181        cnt++;
182      } while (next_partition(a));
183      cout << "\n";
184    }
185
186    {
187      int n = 8;
188      cout << "Increasing partitions of " << n << ":\n";
189      gen_increasing_partitons(n, print);
190    }
191    return 0;
192  }
```

4.2.6 Enumerating Generic Combinatorial Sequences

```
1   /*
2
3   4.2.6 - Enumerating Generic Combinatorial Sequences
4
5   The follow provides a universal method for enumerating
6   abstract combinatorial sequences in O(n^2) time.
7
8   */
9
10  #include <vector>
11
12  class abstract_enumeration {
13   protected:
14    int range, length;
15
16    abstract_enumeration(int r, int l): range(r), length(l) {}
17
18    virtual long long count(const std::vector<int> & pre) {
19      return 0;
20    }
21
22    std::vector<int> next(std::vector<int> & seq) {
23      return from_number(to_number(seq) + 1);
24    }
25
26    long long total_count() {
27      return count(std::vector<int>(0));
28    }
29
30  public:
31    long long to_number(const std::vector<int> & seq) {
32      long long res = 0;
33      for (int i = 0; i < (int)seq.size(); i++) {
34        std::vector<int> pre(seq.begin(), seq.end());
35        pre.resize(i + 1);
36        for (pre[i] = 0; pre[i] < seq[i]; ++pre[i])
37          res += count(pre);
38      }
39      return res;
40    }
41
42    std::vector<int> from_number(long long x) {
43      std::vector<int> seq(length);
44      for (int i = 0; i < (int)seq.size(); i++) {
45        std::vector<int> pre(seq.begin(), seq.end());
46        pre.resize(i + 1);
47        for (pre[i] = 0; pre[i] < range; ++pre[i]) {
48          long long cur = count(pre);
49          if (x < cur) break;
50          x -= cur;
51        }
52        seq[i] = pre[i];
53      }
54      return seq;
55    }
56
```

```
57      template<class ReportFunction>
58      void enumerate(ReportFunction report) {
59        long long total = total_count();
60        for (long long i = 0; i < total; i++) {
61          //assert(i == to_number(from_number(i)));
62          report(from_number(i));
63        }
64      }
65    };
66
67    class arrangements: public abstract_enumeration {
68     public:
69      arrangements(int n, int k) : abstract_enumeration(n, k) {}
70
71      long long count(const std::vector<int> & pre) {
72        int sz = pre.size();
73        for (int i = 0; i < sz - 1; i++)
74          if (pre[i] == pre[sz - 1]) return 0;
75        long long res = 1;
76        for (int i = 0; i < length - sz; i++)
77          res *= range - sz - i;
78        return res;
79      }
80    };
81
82    class permutations: public arrangements {
83     public:
84      permutations(int n) : arrangements(n, n) {}
85    };
86
87    class combinations: public abstract_enumeration {
88      std::vector<std::vector<long long> > binomial;
89
90     public:
91      combinations(int n, int k) : abstract_enumeration(n, k),
92        binomial(n + 1, std::vector<long long>(n + 1, 0)) {
93        for (int i = 0; i <= n; i++)
94          for (int j = 0; j <= i; j++)
95            binomial[i][j] = (j == 0) ? 1 :
96                    binomial[i - 1][j - 1] + binomial[i - 1][j];
97      }
98
99      long long count(const std::vector<int> & pre) {
100        int sz = pre.size();
101        if (sz >= 2 && pre[sz - 1] <= pre[sz - 2]) return 0;
102        int last = sz > 0 ? pre[sz - 1] : -1;
103        return binomial[range - 1 - last][length - sz];
104      }
105    };
106
107    class partitions: public abstract_enumeration {
108      std::vector<std::vector<long long> > p;
109
110     public:
111      partitions(int n) : abstract_enumeration(n + 1, n),
112        p(n + 1, std::vector<long long>(n + 1, 0)) {
113        std::vector<std::vector<long long> > pp(p);
114        pp[0][0] = 1;
115        for (int i = 1; i <= n; i++)
```

```
116        for (int j = 1; j <= i; j++)
117          pp[i][j] = pp[i - 1][j - 1] + pp[i - j][j];
118      for (int i = 1; i <= n; i++)
119        for (int j = 1; j <= n; j++)
120          p[i][j] = pp[i][j] + p[i][j - 1];
121    }
122
123    long long count(const std::vector<int> & pre) {
124      int size = pre.size(), sum = 0;
125      for (int i = 0; i < (int)pre.size(); i++) sum += pre[i];
126      if (sum == range - 1) return 1;
127      if (sum > range - 1 || (size > 0 && pre[size - 1] == 0) ||
128          (size >= 2 && pre[size - 1] > pre[size - 2])) return 0;
129      int last = size > 0 ? pre[size - 1] : range - 1;
130      return p[range - 1 - sum][last];
131    }
132  };
133
134  /*** Example Usage ***/
135
136  #include <iostream>
137  using namespace std;
138
139  void print(const std::vector<int> & v) {
140    for (int i = 0; i < (int)v.size(); i++)
141      cout << v[i] << "␣";
142    cout << "\n";
143  }
144
145  int main() {
146    cout << "Arrangement(3,␣2):\n";
147    arrangements arrg(3, 2);
148    arrg.enumerate(print);
149
150    cout << "Permutation(3):\n";
151    permutations perm(3);
152    perm.enumerate(print);
153
154    cout << "Combination(4,␣3):\n";
155    combinations comb(4, 3);
156    comb.enumerate(print);
157
158    cout << "Partition(4):\n";
159    partitions part(4);
160    part.enumerate(print);
161    return 0;
162  }
```

4.3   Number Theory

4.3.1   GCD, LCM, Mod Inverse, Chinese Remainder

```
1  /*
2
3  4.3.1 - 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 = a (mod r) and N = b (mod 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 = a[i] (mod p[i]), for i = 1..n
120
121 the solution of the set of congruences is:
122
```

```
123    x = a[1] * b[1] * (M/p[1]) + ... + a[n] * b[n] * (M/p[n]) (mod M)
124
125   where M = p[1] * p[2] ... * p[n] and the b[i] are determined for
126
127    b[i] * (M/p[i]) = 1 (mod 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  4.3.2 - Generating Primes
4
5  The following are three methods to generate primes.
6  Although the latter two functions are theoretically
7  linear, the former function with the sieve of
8  Eratosthenes is still significantly the fastest even
9  for n under 1 billion, since its constant factor is
```

```
10  so much better because of its minimal arithmetic
11  operations. For this reason, it should be favored
12  over the other two algorithms in most contest
13  applications. For the computation of larger primes,
14  you should replace int with long long or an arbitrary
15  precision class.
16
17  */
18
19  #include <cmath> /* ceil(), sqrt() */
20  #include <vector>
21
22  //Sieve of Eratosthenes in ~ O(n log log n)
23  //returns: a vector of all primes under n
24  std::vector<int> gen_primes(int n) {
25    std::vector<bool> prime(n + 1, true);
26    int sqrtn = (int)ceil(sqrt(n));
27    for (int i = 2; i <= sqrtn; i++) {
28      if (prime[i])
29        for (int j = i * i; j <= n; j += i)
30          prime[j] = false;
31    }
32    std::vector<int> res;
33    for (int i = 2; i <= n; i++)
34      if (prime[i]) res.push_back(i);
35    return res;
36  }
37
38  //Technically O(n), but on -O2, this is about
39  //as fast as the above sieve for n = 100 million
40  std::vector<int> gen_primes_linear(int n) {
41    std::vector<int> lp(n + 1), res;
42    for (int i = 2; i <= n; i++) {
43      if (lp[i] == 0) {
44        lp[i] = i;
45        res.push_back(i);
46      }
47      for (int j = 0; j < (int)res.size(); j++) {
48        if (res[j] > lp[i] || i * res[j] > n)
49          break;
50        lp[i * res[j]] = res[j];
51      }
52    }
53    return res;
54  }
55
56  //Sieve of Atkins in O(n), somewhat slow due to
57  //its heavier arithmetic compared to the above
58  std::vector<int> gen_primes_atkins(int n) {
59    std::vector<bool> prime(n + 1, false);
60    std::vector<int> res;
61    prime[2] = true;
62    prime[3] = true;
63    int num, lim = ceil(sqrt(n));
64    for (int x = 1; x <= lim; x++) {
65      for (int y = 1; y <= lim; y++) {
66        num = 4 * x * x + y * y;
67        if (num <= n && (num % 12 == 1 || num % 12 == 5))
68          prime[num] = true;
```

```
69          num = 3 * x * x + y * y;
70          if (num <= n && (num % 12 == 7))
71            prime[num] = true;
72          if (x > y) {
73            num = (3 * x * x - y * y);
74            if (num <= n && num % 12 == 11)
75              prime[num] = true;
76          }
77        }
78      }
79      for (int i = 5; i <= lim; i++) {
80        if (prime[i])
81          for (int j = i * i; j <= n; j += i)
82            prime[j] = false;
83      }
84      for (int i = 2; i <= n; i++)
85        if (prime[i]) res.push_back(i);
86      return res;
87    }
88
89    //Double sieve to find primes in [l, h]
90    //Approximately O(sqrt(h) * log log(h - l))
91    std::vector<int> gen_primes(int l, int h) {
92      int sqrth = (int)ceil(sqrt(h));
93      int sqrtsqrth = (int)ceil(sqrt(sqrth));
94      std::vector<bool> prime1(sqrth + 1, true);
95      std::vector<bool> prime2(h - l + 1, true);
96      for (int i = 2; i <= sqrtsqrth; i++) {
97        if (prime1[i])
98          for (int j = i * i; j <= sqrth; j += i)
99            prime1[j] = false;
100     }
101     for (int i = 2, n = h - l; i <= sqrth; i++) {
102       if (prime1[i])
103         for (int j = l / i * i - l; j <= n; j += i)
104           if (j >= 0 && j + l != i)
105             prime2[j] = false;
106     }
107     std::vector<int> res;
108     for (int i = l > 1 ? l : 2; i <= h; i++)
109       if (prime2[i - l]) res.push_back(i);
110     return res;
111   }
112
113   /*** Example Usage ***/
114
115   #include <cassert>
116   #include <ctime>
117   #include <iostream>
118   using namespace std;
119
120   template<class It> void print(It lo, It hi) {
121     while (lo != hi) cout << *(lo++) << "␣";
122     cout << "\n";
123   }
124
125   int main() {
126     int pmax = 10000000;
127     vector<int> p;
```

```
128     time_t start;
129     double delta;
130
131     cout << "Generating␣primes␣up␣to␣" << pmax << "...\n";
132     start = clock();
133     p = gen_primes(pmax);
134     delta = (double)(clock() - start)/CLOCKS_PER_SEC;
135     cout << "gen_primes()␣took␣" << delta << "s.\n";
136
137     start = clock();
138     p = gen_primes_linear(pmax);
139     delta = (double)(clock() - start)/CLOCKS_PER_SEC;
140     cout << "gen_primes_linear()␣took␣" << delta << "s.\n";
141
142     start = clock();
143     p = gen_primes_atkins(pmax);
144     delta = (double)(clock() - start)/CLOCKS_PER_SEC;
145     cout << "gen_primes_atkins()␣took␣" << delta << "s.\n";
146
147     cout << "Generated␣" << p.size() << "␣primes.\n";
148     //print(p.begin(), p.end());
149
150     for (int i = 0; i <= 1000; i++) {
151       assert(gen_primes(i) == gen_primes_linear(i));
152       assert(gen_primes(i) == gen_primes_atkins(i));
153     }
154
155     int l = 1000000000, h = 1000000500;
156     cout << "Generating␣primes␣in␣[" << l << ",␣" << h << "]...\n";
157     start = clock();
158     p = gen_primes(l, h);
159     delta = (double)(clock() - start)/CLOCKS_PER_SEC;
160     cout << "Generated␣" << p.size() << "␣primes␣in␣" << delta << "s.\n";
161     print(p.begin(), p.end());
162     return 0;
163   }
```

4.3.3   Primality Testing

```
1   /* 4.3.3 - Primality Testing */
2
3   #include <cstdlib>  /* rand(), srand() */
4   #include <ctime>    /* time() */
5   #include <stdint.h> /* uint64_t */
6
7   /*
8
9   Trial division in O(sqrt(n)) to return whether n is prime
10  Applies an optimization based on the fact that all
11  primes greater than 3 take the form 6n + 1 or 6n - 1.
12
13  */
14
15  template<class Int> bool is_prime(Int n) {
16    if (n == 2 || n == 3) return true;
17    if (n < 2 || !(n % 2) || !(n % 3)) return false;
18    for (Int i = 5, w = 4; i * i <= n; i += (w = 6 - w))
```

```
19        if (n % i == 0) return false;
20      return true;
21    }
22
23    /*
24
25    Miller-Rabin Primality Test (Probabilistic)
26
27    Checks whether a number n is probably prime. If n is prime,
28    the function is guaranteed to return 1. If n is composite,
29    the function returns 1 with a probability of (1/4)^k,
30    where k is the number of iterations. With k = 1, the
31    probability of a composite being falsely predicted to be a
32    prime is 25%. If k = 5, the probability for this error is
33    just less than 0.1%. Thus, k = 18 to 20 is accurate enough
34    for most applications. All values of n < 2^63 is supported.
35
36    Complexity: O(k log^3(n)). In comparison to trial division,
37    the Miller-Rabin algorithm on 32-bit ints take ~45
38    operations for k = 10 iterations (~0.0001% error), while the
39    former takes ~10,000.
40
41    Warning: Due to the overflow of modular exponentiation,
42             this will only work on inputs less than 2^63.
43
44    */
45
46    uint64_t mulmod(uint64_t a, uint64_t b, uint64_t m) {
47      uint64_t x = 0, y = a % m;
48      for (; b > 0; b >>= 1) {
49        if (b & 1) x = (x + y) % m;
50        y = (y << 1) % m;
51      }
52      return x % m;
53    }
54
55    uint64_t powmod(uint64_t a, uint64_t b, uint64_t m) {
56      uint64_t x = 1, y = a;
57      for (; b > 0; b >>= 1) {
58        if (b & 1) x = mulmod(x, y, m);
59        y = mulmod(y, y, m);
60      }
61      return x % m;
62    }
63
64    //5 calls to rand() is unnecessary if RAND_MAX is 2^31-1
65    uint64_t rand64u() {
66      return ((uint64_t)(rand() & 0xf) << 60) |
67             ((uint64_t)(rand() & 0x7fff) << 45) |
68             ((uint64_t)(rand() & 0x7fff) << 30) |
69             ((uint64_t)(rand() & 0x7fff) << 15) |
70             ((uint64_t)(rand() & 0x7fff));
71    }
72
73    bool is_probable_prime(long long n, int k = 20) {
74      if (n < 2 || (n != 2 && !(n & 1))) return false;
75      uint64_t s = n - 1, p = n - 1, x, r;
76      while (!(s & 1)) s >>= 1;
77      for (int i = 0; i < k; i++) {
```

```
78       r = powmod(rand64u() % p + 1, s, n);
79       for (x = s; x != p && r != 1 && r != p; x <<= 1)
80         r = mulmod(r, r, n);
81       if (r != p && !(x & 1)) return false;
82     }
83     return true;
84   }
85
86   /*
87
88   Miller-Rabin - Deterministic for all unsigned long long
89
90   Although Miller-Rabin is generally probabilistic, the seven
91   bases 2, 325, 9375, 28178, 450775, 9780504, 1795265022 have
92   been proven to deterministically test the primality of all
93   numbers under 2^64. See: http://miller-rabin.appspot.com/
94
95   Complexity: O(log^3(n)).
96   Warning: Due to the overflow of modular exponentiation,
97            this will only work on inputs less than 2^63.
98
99   */
100
101  bool is_prime_fast(long long n) {
102    static const uint64_t witnesses[] =
103      {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
104    if (n <= 1) return false;
105    if (n <= 3) return true;
106    if ((n & 1) == 0) return false;
107    uint64_t d = n - 1;
108    int s = 0;
109    for (; ~d & 1; s++) d >>= 1;
110    for (int i = 0; i < 7; i++) {
111      if (witnesses[i] > (uint64_t)n - 2) break;
112      uint64_t x = powmod(witnesses[i], d, n);
113      if (x == 1 || x == (uint64_t)n - 1) continue;
114      bool flag = false;
115      for (int j = 0; j < s; j++) {
116        x = powmod(x, 2, n);
117        if (x == 1) return false;
118        if (x == (uint64_t)n - 1) {
119          flag = true;
120          break;
121        }
122      }
123      if (!flag) return false;
124    }
125    return true;
126  }
127
128  /*** Example Usage ***/
129
130  #include <cassert>
131
132  int main() {
133    int len = 20;
134    unsigned long long v[] = {
135      0, 1, 2, 3, 4, 5, 11,
136      1000000ull,
```

```
137        772023803ull,
138        792904103ull,
139        813815117ull,
140        834753187ull,
141        855718739ull,
142        876717799ull,
143        897746119ull,
144        2147483647ull,
145        5705234089ull,
146        5914686649ull,
147        6114145249ull,
148        6339503641ull,
149        6548531929ull
150      };
151      for (int i = 0; i < len; i++) {
152        bool p = is_prime(v[i]);
153        assert(p == is_prime_fast(v[i]));
154        assert(p == is_probable_prime(v[i]));
155      }
156      return 0;
157    }
```

4.3.4   Integer Factorization

```
1   /* 4.3.4 - Integer Factorization */
2
3   #include <algorithm> /* std::sort() */
4   #include <cmath>      /* sqrt() */
5   #include <cstdlib>    /* rand(), srand() */
6   #include <stdint.h>  /* uint64_t */
7   #include <vector>
8
9   /*
10
11  Trial division in O(sqrt(n))
12
13  Returns a vector of pair<prime divisor, exponent>
14  e.g. prime_factorize(15435) => {(3,2),(5,1),(7,3)}
15  because 3^2 * 5^1 * 7^3 = 15435
16
17  */
18
19  template<class Int>
20  std::vector<std::pair<Int, int> > prime_factorize(Int n) {
21    std::vector<std::pair<Int, int> > res;
22    for (Int d = 2; ; d++) {
23      int power = 0, quot = n / d, rem = n - quot * d;
24      if (d > quot || (d == quot && rem > 0)) break;
25      for (; rem == 0; rem = n - quot * d) {
26        power++;
27        n = quot;
28        quot = n / d;
29      }
30      if (power > 0) res.push_back(std::make_pair(d, power));
31    }
32    if (n > 1) res.push_back(std::make_pair(n, 1));
33    return res;
```

```
34  }
35
36  /*
37
38  Trial division in O(sqrt(n))
39
40  Returns a sorted vector of all divisors of n.
41  e.g. get_all_divisors(28) => {1, 2, 4, 7, 14, 28}
42
43  */
44
45  template<class Int>
46  std::vector<Int> get_all_divisors(Int n) {
47    std::vector<Int> res;
48    for (int d = 1; d * d <= n; d++) {
49      if (n % d == 0) {
50        res.push_back(d);
51        if (d * d != n)
52          res.push_back(n / d);
53      }
54    }
55    std::sort(res.begin(), res.end());
56    return res;
57  }
58
59  /*
60
61  Fermat's Method ~ O(sqrt(N))
62
63  Given a number n, returns one factor of n that is
64  not necessary prime. Fermat's algorithm is pretty
65  good when the number you wish to factor has two
66  factors very near to sqrt(n). Otherwise, it is just
67  as slow as the basic trial division algorithm.
68
69  e.g. 14917627 => 1 (it's a prime), or
70       1234567 => 127 (because 127*9721 = 1234567)
71
72  */
73
74  long long fermat(long long n) {
75    if (n % 2 == 0) return 2;
76    long long x = sqrt(n), y = 0;
77    long long r = x * x - y * y - n;
78    while (r != 0) {
79      if (r < 0) {
80        r += x + x + 1;
81        x++;
82      } else {
83        r -= y + y + 1;
84        y++;
85      }
86    }
87    return x != y ? x - y : x + y;
88  }
89
90  /*
91
92  Pollard's rho Algorithm with Brent's Optimization
```

```
93
94   Brent's algorithm is a much faster variant of Pollard's
95   rho algorithm using Brent's cycle-finding method. The
96   following function returns a (not necessarily prime) factor
97   of n, or n if n is prime. Note that this is not necessarily
98   guaranteed to always work perfectly. brent(9) may return 9
99   instead of 3. However, it works well when coupled with trial
100  division in the function prime_factorize_big() below.
101
102  */
103
104  uint64_t mulmod(uint64_t a, uint64_t b, uint64_t m) {
105    uint64_t x = 0, y = a % m;
106    for (; b > 0; b >>= 1) {
107      if (b & 1) x = (x + y) % m;
108      y = (y << 1) % m;
109    }
110    return x % m;
111  }
112
113  //5 calls to rand() is unnecessary if RAND_MAX is 2^31-1
114  uint64_t rand64u() {
115    return ((uint64_t)(rand() & 0xf) << 60) |
116           ((uint64_t)(rand() & 0x7fff) << 45) |
117           ((uint64_t)(rand() & 0x7fff) << 30) |
118           ((uint64_t)(rand() & 0x7fff) << 15) |
119           ((uint64_t)(rand() & 0x7fff));
120  }
121
122  uint64_t gcd(uint64_t a, uint64_t b) {
123    return b == 0 ? a : gcd(b, a % b);
124  }
125
126  long long brent(long long n) {
127    if (n % 2 == 0) return 2;
128    long long y = rand64u() % (n - 1) + 1;
129    long long c = rand64u() % (n - 1) + 1;
130    long long m = rand64u() % (n - 1) + 1;
131    long long g = 1, r = 1, q = 1, ys = 0, hi = 0, x = 0;
132    while (g == 1) {
133      x = y;
134      for (int i = 0; i < r; i++)
135        y = (mulmod(y, y, n) + c) % n;
136      for (long long k = 0; k < r && g == 1; k += m) {
137        ys = y;
138        hi = std::min(m, r - k);
139        for (int j = 0; j < hi; j++) {
140          y = (mulmod(y, y, n) + c) % n;
141          q = mulmod(q, x > y ? x - y : y - x, n);
142        }
143        g = gcd(q, n);
144      }
145      r *= 2;
146    }
147    if (g == n) do {
148      ys = (mulmod(ys, ys, n) + c) % n;
149      g = gcd(x > ys ? x - ys : ys - x, n);
150    } while (g <= 1);
151    return g;
```

```
152  }
153
154  /*
155
156  Combines Brent's method with trial division to efficiently
157  generate the prime factorization of large integers.
158
159  Returns a vector of prime divisors that multiply to n.
160  e.g. prime_factorize(15435) => {3, 3, 5, 7, 7, 7}
161        because 3^2 * 5^1 * 7^3 = 15435
162
163  */
164
165  std::vector<long long> prime_factorize_big(long long n) {
166    if (n <= 0) return std::vector<long long>(0);
167    if (n == 1) return std::vector<long long>(1, 1);
168    std::vector<long long> res;
169    for (; n % 2 == 0; n /= 2) res.push_back(2);
170    for (; n % 3 == 0; n /= 3) res.push_back(3);
171    int mx = 1000000; //trial division for factors <= 1M
172    for (int i = 5, w = 2; i <= mx; i += w, w = 6 - w) {
173      for (; n % i == 0; n /= i) res.push_back(i);
174    }
175    for (long long p = 0, p1; n > mx; n /= p1) { //brent
176      for (p1 = n; p1 != p; p1 = brent(p)) p = p1;
177      res.push_back(p1);
178    }
179    if (n != 1) res.push_back(n);
180    sort(res.begin(), res.end());
181    return res;
182  }
183
184  /*** Example Usage ***/
185
186  #include <cassert>
187  #include <iostream>
188  #include <ctime>
189  using namespace std;
190
191  template<class It> void print(It lo, It hi) {
192    while (lo != hi) cout << *(lo++) << "␣";
193    cout << "\n";
194  }
195
196  template<class It> void printp(It lo, It hi) {
197    for (; lo != hi; ++lo)
198      cout << "(" << lo->first << "," << lo->second << ")␣";
199    cout << "\n";
200  }
201
202  int main() {
203    srand(time(0));
204
205    vector< pair<int, int> > v1 = prime_factorize(15435);
206    printp(v1.begin(), v1.end());
207
208    vector<int> v2 = get_all_divisors(28);
209    print(v2.begin(), v2.end());
210
```

```
211    long long n = 1000003ll*10000003711;
212    assert(fermat(n) == 1000003ll);
213
214    vector<long long> v3 = prime_factorize_big(n);
215    print(v3.begin(), v3.end());
216
217    return 0;
218  }
```

4.3.5   Euler's Totient Function

```
1   /*
2
3   4.3.5 - Euler's Totient Function
4
5   Euler's totient function (or Euler's phi function) counts
6   the positive integers less than or equal to n that are
7   relatively prime to n. (These integers are sometimes
8   referred to as totatives of n.) Thus, phi(n) is the number
9   of integers k in the range [1, n] for which gcd(n, k) = 1.
10
11  E.g. if n = 9. Then gcd(9, 3) = gcd(9, 6) = 3 and gcd(9, 9)
12  = 9. The other six numbers in the range [1, 9], i.e. 1, 2,
13  4, 5, 7 and 8 are relatively prime to 9. Thus, phi(9) = 6.
14
15  */
16
17  #include <vector>
18
19  int phi(int n) {
20    int res = n;
21    for (int i = 2; i * i <= n; i++)
22      if (n % i == 0) {
23        while (n % i == 0) n /= i;
24        res -= res / i;
25      }
26    if (n > 1) res -= res / n;
27    return res;
28  }
29
30  std::vector<int> phi_table(int n) {
31    std::vector<int> res(n + 1);
32    for (int i = 1; i <= n; i++)
33      res[i] = i;
34    for (int i = 1; i <= n; i++)
35      for (int j = i + i; j <= n; j += i)
36        res[j] -= res[i];
37    return res;
38  }
39
40  /*** Example Usage ***/
41
42  #include <cassert>
43  #include <iostream>
44  using namespace std;
45
46  int main() {
```

```
47    cout << phi(1) << "\n";        //1
48    cout << phi(9) << "\n";        //6
49    cout << phi(1234567) << "\n"; //1224720
50
51    int n = 1000;
52    vector<int> v = phi_table(n);
53    for (int i = 0; i <= n; i++)
54      assert(v[i] == phi(i));
55    return 0;
56  }
```

4.4   Arbitrary Precision Arithmetic

4.4.1   Big Integers (Simple)

```
1   /*
2
3   4.4.1 - Big Integers (Simple)
4
5   Description: Integer arbitrary precision functions.
6   To use, pass bigints to the functions by addresses.
7   e.g. add(&a, &b, &c) stores the sum of a and b into c.
8
9   Complexity: comp(), to_string(), digit_shift(), add(),
10  and sub() are O(N) on the number of digits. mul() and
11  div() are O(N^2). zero_justify() is amortized constant.
12
13  */
14
15  #include <string>
16
17  struct bigint {
18    static const int maxdigits = 1000;
19
20    char dig[maxdigits], sign;
21    int last;
22
23    bigint(long long x = 0): sign(x < 0 ? -1 : 1) {
24      for (int i = 0; i < maxdigits; i++) dig[i] = 0;
25      if (x == 0) { last = 0; return; }
26      if (x < 0) x = -x;
27      for (last = -1; x > 0; x /= 10) dig[++last] = x % 10;
28    }
29
30    bigint(const std::string & s): sign(s[0] == '-' ? -1 : 1) {
31      for (int i = 0; i < maxdigits; i++) dig[i] = 0;
32      last = -1;
33      for (int i = s.size() - 1; i >= 0; i--)
34        dig[++last] = (s[i] - '0');
35      if (dig[last] + '0' == '-') dig[last--] = 0;
36    }
37  };
38
39  void zero_justify(bigint * x) {
40    while (x->last > 0 && !x->dig[x->last]) x->last--;
41    if (x->last == 0 && x->dig[0] == 0) x->sign = 1;
```

```
42  }
43
44  void add(bigint * a, bigint * b, bigint * c);
45  void sub(bigint * a, bigint * b, bigint * c);
46
47  //returns: -1 if a < b, 0 if a == b, or 1 if a > b
48  int comp(bigint * a, bigint * b) {
49    if (a->sign != b->sign) return b->sign;
50    if (b->last > a->last) return a->sign;
51    if (a->last > b->last) return -a->sign;
52    for (int i = a->last; i >= 0; i--) {
53      if (a->dig[i] > b->dig[i]) return -a->sign;
54      if (b->dig[i] > a->dig[i]) return  a->sign;
55    }
56    return 0;
57  }
58
59  void add(bigint * a, bigint * b, bigint * c) {
60    if (a->sign != b->sign) {
61      if (a->sign == -1)
62        a->sign = 1, sub(b, a, c), a->sign = -1;
63      else
64        b->sign = 1, sub(a, b, c), b->sign = -1;
65      return;
66    }
67    c->sign = a->sign;
68    c->last = (a->last > b->last ? a->last : b->last) + 1;
69    for (int i = 0, carry = 0; i <= c->last; i++) {
70      c->dig[i] = (carry + a->dig[i] + b->dig[i]) % 10;
71      carry = (carry + a->dig[i] + b->dig[i]) / 10;
72    }
73    zero_justify(c);
74  }
75
76  void sub(bigint * a, bigint * b, bigint * c) {
77    if (a->sign == -1 || b->sign == -1) {
78      b->sign *= -1, add(a, b, c), b->sign *= -1;
79      return;
80    }
81    if (comp(a, b) == 1) {
82      sub(b, a, c), c->sign = -1;
83      return;
84    }
85    c->last = (a->last > b->last) ? a->last : b->last;
86    for (int i = 0, borrow = 0, v; i <= c->last; i++) {
87      v = a->dig[i] - borrow;
88      if (i <= b->last) v -= b->dig[i];
89      if (a->dig[i] > 0) borrow = 0;
90      if (v < 0) v += 10, borrow = 1;
91      c->dig[i] = v % 10;
92    }
93    zero_justify(c);
94  }
95
96  void digit_shift(bigint * x, int n) {
97    if (!x->last && !x->dig[0]) return;
98    for (int i = x->last; i >= 0; i--)
99      x->dig[i + n] = x->dig[i];
100   for (int i = 0; i < n; i++) x->dig[i] = 0;
```

```
101     x->last += n;
102   }
103
104   void mul(bigint * a, bigint * b, bigint * c) {
105     bigint row = *a, tmp;
106     for (int i = 0; i <= b->last; i++) {
107       for (int j = 1; j <= b->dig[i]; j++) {
108         add(c, &row, &tmp);
109         *c = tmp;
110       }
111       digit_shift(&row, 1);
112     }
113     c->sign = a->sign * b->sign;
114     zero_justify(c);
115   }
116
117   void div(bigint * a, bigint * b, bigint * c) {
118     bigint row, tmp;
119     int asign = a->sign, bsign = b->sign;
120     a->sign = b->sign = 1;
121     c->last = a->last;
122     for (int i = a->last; i >= 0; i--) {
123       digit_shift(&row, 1);
124       row.dig[0] = a->dig[i];
125       c->dig[i] = 0;
126       for (; comp(&row, b) != 1; row = tmp) {
127         c->dig[i]++;
128         sub(&row, b, &tmp);
129       }
130     }
131     c->sign = (a->sign = asign) * (b->sign = bsign);
132     zero_justify(c);
133   }
134
135   std::string to_string(bigint * x) {
136     std::string s(x->sign == -1 ? "-" : "");
137     for (int i = x->last; i >= 0; i--)
138       s += (char)('0' + x->dig[i]);
139     return s;
140   }
141
142   /*** Example Usage ***/
143
144   #include <cassert>
145
146   int main() {
147     bigint a("-9899819294989142124"), b("12398124981294214");
148     bigint sum; add(&a, &b, &sum);
149     bigint dif; sub(&a, &b, &dif);
150     bigint prd; mul(&a, &b, &prd);
151     bigint quo; div(&a, &b, &quo);
152     assert(to_string(&sum) == "-9887421170007847910");
153     assert(to_string(&dif) == "-9912217419970436338");
154     assert(to_string(&prd) == "-122739196911503356525379735104870536");
155     assert(to_string(&quo) == "-798");
156     return 0;
157   }
```

4.4.2 Big Integer and Rational Class

```
1   /*
2
3   4.4.2 - Big Integer and Rational Class
4
5   The following bigint class is implemented by storing "chunks"
6   of the big integer in a large base that is a power of 10 so
7   it can be efficiently stored, operated on, and printed.
8
9   It has extensive features including karatsuba multiplication,
10  exponentiation by squaring, and n-th root using binary search.
11  The class is thoroughly templatized, so you can use it as
12  easily as you do for normal ints. For example, you may use
13  operators with a bigint and a string (e.g. bigint(1234)+"-567"
14  and the result will be correctly promoted to a bigint that has
15  a value of 667). I/O is done using <iostream>. For example:
16    bigint a, b; cin >> a >> b; cout << a + b << "\n";
17  adds two integers together and prints the result, just as you
18  would expect for a normal int, except with arbitrary precision.
19  The class also supports other streams such as fstream.
20
21  After the bigint class, a class for rational numbers is
22  implemented, using two bigints to store its numerators and
23  denominators. It is useful for when exact results of division
24  operations are needed.
25
26  */
27
28  #include <algorithm> /* std::max(), std::swap() */
29  #include <cmath>      /* sqrt() */
30  #include <cstdlib>    /* rand() */
31  #include <iomanip>    /* std::setw(), std::setfill() */
32  #include <istream>
33  #include <ostream>
34  #include <sstream>
35  #include <stdexcept> /* std::runtime_error() */
36  #include <string>
37  #include <utility>    /* std::pair */
38  #include <vector>
39
40  struct bigint {
41    //base should be a power of 10 for I/O to work
42    //base and base_digits should be consistent
43    static const int base = 1000000000, base_digits = 9;
44
45    typedef std::vector<int> vint;
46    typedef std::vector<long long> vll;
47
48    vint a; //a[0] stores right-most (least significant) base-digit
49    int sign;
50
51    bigint() : sign(1) {}
52    bigint(int v) { *this = (long long)v; }
53    bigint(long long v) { *this = v; }
54    bigint(const std::string & s) { read(s); }
55    bigint(const char * s) { read(std::string(s)); }
56
```

```
57      void trim() {
58        while (!a.empty() && a.back() == 0) a.pop_back();
59        if (a.empty()) sign = 1;
60      }
61
62      void read(const std::string & s) {
63        sign = 1;
64        a.clear();
65        int pos = 0;
66        while (pos < (int)s.size() && (s[pos] == '-' || s[pos] == '+')) {
67          if (s[pos] == '-') sign = -sign;
68          pos++;
69        }
70        for (int i = s.size() - 1; i >= pos; i -= base_digits) {
71          int x = 0;
72          for (int j = std::max(pos, i - base_digits + 1); j <= i; j++)
73            x = x * 10 + s[j] - '0';
74          a.push_back(x);
75        }
76        trim();
77      }
78
79      void operator = (const bigint & v) {
80        sign = v.sign;
81        a = v.a;
82      }
83
84      void operator = (long long v) {
85        sign = 1;
86        if (v < 0) sign = -1, v = -v;
87        a.clear();
88        for (; v > 0; v /= base) a.push_back(v % base);
89      }
90
91      bigint operator + (const bigint & v) const {
92        if (sign == v.sign) {
93          bigint res = v;
94          int carry = 0;
95          for (int i = 0; i < (int)std::max(a.size(), v.a.size()) || carry; i++) {
96            if (i == (int)res.a.size()) res.a.push_back(0);
97            res.a[i] += carry + (i < (int)a.size() ? a[i] : 0);
98            carry = res.a[i] >= base;
99            if (carry) res.a[i] -= base;
100         }
101         return res;
102       }
103       return *this - (-v);
104     }
105
106     bigint operator - (const bigint & v) const {
107       if (sign == v.sign) {
108         if (abs() >= v.abs()) {
109           bigint res(*this);
110           for (int i = 0, carry = 0; i < (int)v.a.size() || carry; i++) {
111             res.a[i] -= carry + (i < (int)v.a.size() ? v.a[i] : 0);
112             carry = res.a[i] < 0;
113             if (carry) res.a[i] += base;
114           }
115           res.trim();
```

```
116            return res;
117          }
118          return -(v - *this);
119        }
120        return *this + (-v);
121      }
122
123      void operator *= (int v) {
124        if (v < 0) sign = -sign, v = -v;
125        for (int i = 0, carry = 0; i < (int)a.size() || carry; i++) {
126          if (i == (int)a.size()) a.push_back(0);
127          long long cur = a[i] * (long long)v + carry;
128          carry = (int)(cur / base);
129          a[i] = (int)(cur % base);
130          //asm("divl %%ecx" : "=a"(carry), "=d"(a[i]) : "A"(cur), "c"(base));
131        }
132        trim();
133      }
134
135      bigint operator * (int v) const {
136        bigint res(*this);
137        res *= v;
138        return res;
139      }
140
141      static vint convert_base(const vint & a, int l1, int l2) {
142        vll p(std::max(l1, l2) + 1);
143        p[0] = 1;
144        for (int i = 1; i < (int)p.size(); i++) p[i] = p[i - 1] * 10;
145        vint res;
146        long long cur = 0;
147        for (int i = 0, cur_digits = 0; i < (int)a.size(); i++) {
148          cur += a[i] * p[cur_digits];
149          cur_digits += l1;
150          while (cur_digits >= l2) {
151            res.push_back((int)(cur % p[l2]));
152            cur /= p[l2];
153            cur_digits -= l2;
154          }
155        }
156        res.push_back((int)cur);
157        while (!res.empty() && res.back() == 0) res.pop_back();
158        return res;
159      }
160
161      //complexity: O(3N^log2(3)) ~ O(3N^1.585)
162      static vll karatsuba_multiply(const vll & a, const vll & b) {
163        int n = a.size();
164        vll res(n + n);
165        if (n <= 32) {
166          for (int i = 0; i < n; i++)
167            for (int j = 0; j < n; j++)
168              res[i + j] += a[i] * b[j];
169          return res;
170        }
171        int k = n >> 1;
172        vll a1(a.begin(), a.begin() + k), a2(a.begin() + k, a.end());
173        vll b1(b.begin(), b.begin() + k), b2(b.begin() + k, b.end());
174        vll a1b1 = karatsuba_multiply(a1, b1);
```

```cpp
175        vll a2b2 = karatsuba_multiply(a2, b2);
176        for (int i = 0; i < k; i++) a2[i] += a1[i];
177        for (int i = 0; i < k; i++) b2[i] += b1[i];
178        vll r = karatsuba_multiply(a2, b2);
179        for (int i = 0; i < (int)a1b1.size(); i++) r[i] -= a1b1[i];
180        for (int i = 0; i < (int)a2b2.size(); i++) r[i] -= a2b2[i];
181        for (int i = 0; i < (int)r.size(); i++) res[i + k] += r[i];
182        for (int i = 0; i < (int)a1b1.size(); i++) res[i] += a1b1[i];
183        for (int i = 0; i < (int)a2b2.size(); i++) res[i + n] += a2b2[i];
184        return res;
185      }
186
187      bigint operator * (const bigint & v) const {
188        //if really big values cause overflow, use smaller _base
189        static const int _base = 10000, _base_digits = 4;
190        vint _a = convert_base(this->a, base_digits, _base_digits);
191        vint _b = convert_base(v.a, base_digits, _base_digits);
192        vll a(_a.begin(), _a.end());
193        vll b(_b.begin(), _b.end());
194        while (a.size() < b.size()) a.push_back(0);
195        while (b.size() < a.size()) b.push_back(0);
196        while (a.size() & (a.size() - 1)) {
197          a.push_back(0);
198          b.push_back(0);
199        }
200        vll c = karatsuba_multiply(a, b);
201        bigint res;
202        res.sign = sign * v.sign;
203        for (int i = 0, carry = 0; i < (int)c.size(); i++) {
204          long long cur = c[i] + carry;
205          res.a.push_back((int)(cur % _base));
206          carry = (int)(cur / _base);
207        }
208        res.a = convert_base(res.a, _base_digits, base_digits);
209        res.trim();
210        return res;
211      }
212
213      bigint operator ^ (const bigint & v) const {
214        if (v.sign == -1) return bigint(0);
215        bigint x(*this), n(v), res(1);
216        while (!n.is_zero()) {
217          if (n.a[0] % 2 == 1) res *= x;
218          x *= x;
219          n /= 2;
220        }
221        return res;
222      }
223
224      friend std::pair<bigint, bigint> divmod(const bigint & a1, const bigint & b1) {
225        int norm = base / (b1.a.back() + 1);
226        bigint a = a1.abs() * norm;
227        bigint b = b1.abs() * norm;
228        bigint q, r;
229        q.a.resize(a.a.size());
230        for (int i = a.a.size() - 1; i >= 0; i--) {
231          r *= base;
232          r += a.a[i];
233          int s1 = r.a.size() <= b.a.size() ? 0 : r.a[b.a.size()];
```

```
234        int s2 = r.a.size() <= b.a.size() - 1 ? 0 : r.a[b.a.size() - 1];
235        int d = ((long long)base * s1 + s2) / b.a.back();
236        for (r -= b * d; r < 0; r += b) d--;
237        q.a[i] = d;
238      }
239      q.sign = a1.sign * b1.sign;
240      r.sign = a1.sign;
241      q.trim();
242      r.trim();
243      return std::make_pair(q, r / norm);
244    }
245
246    bigint operator / (const bigint & v) const { return divmod(*this, v).first; }
247    bigint operator % (const bigint & v) const { return divmod(*this, v).second; }
248
249    bigint & operator /= (int v) {
250      if (v < 0) sign = -sign, v = -v;
251      for (int i = a.size() - 1, rem = 0; i >= 0; i--) {
252        long long cur = a[i] + rem * (long long)base;
253        a[i] = (int)(cur / v);
254        rem = (int)(cur % v);
255      }
256      trim();
257      return *this;
258    }
259
260    bigint operator / (int v) const {
261      bigint res(*this);
262      res /= v;
263      return res;
264    }
265
266    int operator % (int v) const {
267      if (v < 0) v = -v;
268      int m = 0;
269      for (int i = a.size() - 1; i >= 0; i--)
270        m = (a[i] + m * (long long)base) % v;
271      return m * sign;
272    }
273
274    bigint operator ++(int) { bigint t(*this); operator++(); return t; }
275    bigint operator --(int) { bigint t(*this); operator--(); return t; }
276    bigint & operator ++() { *this = *this + bigint(1); return *this; }
277    bigint & operator --() { *this = *this - bigint(1); 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    bigint & operator %= (const bigint & v) { *this = *this % v; return *this; }
283    bigint & operator ^= (const bigint & v) { *this = *this ^ v; return *this; }
284
285    bool operator < (const bigint & v) const {
286      if (sign != v.sign) return sign < v.sign;
287      if (a.size() != v.a.size())
288        return a.size() * sign < v.a.size() * v.sign;
289      for (int i = a.size() - 1; i >= 0; i--)
290        if (a[i] != v.a[i])
291          return a[i] * sign < v.a[i] * sign;
292      return false;
```

```
293      }
294
295      bool operator >  (const bigint & v) const { return v < *this; }
296      bool operator <= (const bigint & v) const { return !(v < *this); }
297      bool operator >= (const bigint & v) const { return !(*this < v); }
298      bool operator == (const bigint & v) const { return !(*this < v) && !(v < *this); }
299      bool operator != (const bigint & v) const { return *this < v || v < *this; }
300
301      int size() const {
302        if (a.empty()) return 1;
303        std::ostringstream oss;
304        oss << a.back();
305        return oss.str().length() + base_digits*(a.size() - 1);
306      }
307
308      bool is_zero() const {
309        return a.empty() || (a.size() == 1 && !a[0]);
310      }
311
312      bigint operator - () const {
313        bigint res(*this);
314        res.sign = -sign;
315        return res;
316      }
317
318      bigint abs() const {
319        bigint res(*this);
320        res.sign *= res.sign;
321        return res;
322      }
323
324      friend bigint abs(const bigint & a) {
325        return a.abs();
326      }
327
328      friend bigint gcd(const bigint & a, const bigint & b) {
329        return b.is_zero() ? a : gcd(b, a % b);
330      }
331
332      friend bigint lcm(const bigint & a, const bigint & b) {
333        return a / gcd(a, b) * b;
334      }
335
336      friend bigint sqrt(const bigint & x) {
337        bigint a = x;
338        while (a.a.empty() || a.a.size() % 2 == 1) a.a.push_back(0);
339        int n = a.a.size();
340        int firstdig = sqrt((double)a.a[n - 1] * base + a.a[n - 2]);
341        int norm = base / (firstdig + 1);
342        a *= norm;
343        a *= norm;
344        while (a.a.empty() || a.a.size() % 2 == 1) a.a.push_back(0);
345        bigint r = (long long)a.a[n - 1] * base + a.a[n - 2];
346        firstdig = sqrt((double)a.a[n - 1] * base + a.a[n - 2]);
347        int q = firstdig;
348        bigint res;
349        for (int j = n / 2 - 1; j >= 0; j--) {
350          for (;; q--) {
351            bigint r1 = (r - (res * 2 * base + q) * q) * base * base + (j > 0 ?
```

```
352                            (long long)a.a[2 * j - 1] * base + a.a[2 * j - 2] : 0);
353            if (r1 >= 0) {
354              r = r1;
355                break;
356            }
357          }
358          res = (res * base) + q;
359          if (j > 0) {
360            int d1 = res.a.size() + 2 < r.a.size() ? r.a[res.a.size() + 2] : 0;
361            int d2 = res.a.size() + 1 < r.a.size() ? r.a[res.a.size() + 1] : 0;
362            int d3 = res.a.size() < r.a.size() ? r.a[res.a.size()] : 0;
363            q = ((long long)d1*base*base + (long long)d2*base + d3)/(firstdig * 2);
364          }
365        }
366        res.trim();
367        return res / norm;
368      }
369
370      friend bigint nthroot(const bigint & x, const bigint & n) {
371        bigint hi = 1;
372        while ((hi ^ n) <= x) hi *= 2;
373        bigint lo = hi / 2, mid, midn;
374        while (lo < hi) {
375          mid = (lo + hi) / 2;
376          midn = mid ^ n;
377          if (lo < mid && midn < x) {
378            lo = mid;
379          } else if (mid < hi && x < midn) {
380            hi = mid;
381          } else {
382            return mid;
383          }
384        }
385        return mid + 1;
386      }
387
388      friend std::istream & operator >> (std::istream & in, bigint & v) {
389        std::string s;
390        in >> s;
391        v.read(s);
392        return in;
393      }
394
395      friend std::ostream & operator << (std::ostream & out, const bigint & v) {
396        if (v.sign == -1) out << '-';
397        out << (v.a.empty() ? 0 : v.a.back());
398        for (int i = v.a.size() - 2; i >= 0; i--)
399          out << std::setw(base_digits) << std::setfill('0') << v.a[i];
400        return out;
401      }
402
403      std::string to_string() const {
404        std::ostringstream oss;
405        if (sign == -1) oss << '-';
406        oss << (a.empty() ? 0 : a.back());
407        for (int i = a.size() - 2; i >= 0; i--)
408          oss << std::setw(base_digits) << std::setfill('0') << a[i];
409        return oss.str();
410      }
```

```
411
412    long long to_llong() const {
413      long long res = 0;
414      for (int i = a.size() - 1; i >= 0; i--)
415        res = res * base + a[i];
416      return res * sign;
417    }
418
419    double to_double() const {
420      std::stringstream ss(to_string());
421      double res;
422      ss >> res;
423      return res;
424    }
425
426    long double to_ldouble() const {
427      std::stringstream ss(to_string());
428      long double res;
429      ss >> res;
430      return res;
431    }
432
433    static bigint rand(int len) {
434      if (len == 0) return bigint(0);
435      std::string s(1, '1' + (::rand() % 9));
436      for (int i = 1; i < len; i++) s += '0' + (::rand() % 10);
437      return bigint(s);
438    }
439  };
440
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> bool operator == (const T & a, const bigint & b) { return bigint(a) == b; }
446  template<class T> bool 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  template<class T> bigint operator - (const T & a, const bigint & b) { return bigint(a) - b;  }
449  template<class T> bigint operator ^ (const T & a, const bigint & b) { return bigint(a) ^ b;  }
450
451  /*
452
453  Exclude *, /, and % to force a user decision between int and bigint algorithms
454
455  bigint operator * (bigint a, bigint b) vs. bigint operator * (bigint a, int b)
456  bigint operator / (bigint a, bigint b) vs. bigint operator / (bigint a, int b)
457  bigint operator % (bigint a, bigint b) vs. int operator % (bigint a, int b)
458
459  */
460
461  struct rational {
462    bigint num, den;
463
464    rational(): num(0), den(1) {}
465    rational(long long n): num(n), den(1) {}
466    rational(const bigint & n) : num(n), den(1) {}
467
468    template<class T1, class T2>
469    rational(const T1 & n, const T2 & d): num(n), den(d) {
```

```
470      if (den == 0)
471        throw std::runtime_error("Rational division by zero.");
472      if (den < 0) {
473        num = -num;
474        den = -den;
475      }
476      bigint a(num < 0 ? -num : num), b(den), tmp;
477      while (a != 0 && b != 0) {
478        tmp = a % b;
479        a = b;
480        b = tmp;
481      }
482      bigint gcd = (b == 0) ? a : b;
483      num /= gcd;
484      den /= gcd;
485    }
486
487    bool operator < (const rational & r) const {
488      return num * r.den < r.num * den;
489    }
490
491    bool operator > (const rational & r) const {
492      return r.num * den < num * r.den;
493    }
494
495    bool operator <= (const rational & r) const {
496      return !(r < *this);
497    }
498
499    bool operator >= (const rational & r) const {
500      return !(*this < r);
501    }
502
503    bool operator == (const rational & r) const {
504      return num == r.num && den == r.den;
505    }
506
507    bool operator != (const rational & r) const {
508      return num != r.num || den != r.den;
509    }
510
511    rational operator + (const rational & r) const {
512      return rational(num * r.den + r.num * den, den * r.den);
513    }
514
515    rational operator - (const rational & r) const {
516      return rational(num * r.den - r.num * den, r.den * den);
517    }
518
519    rational operator * (const rational & r) const {
520      return rational(num * r.num, r.den * den);
521    }
522
523    rational operator / (const rational & r) const {
524      return rational(num * r.den, den * r.num);
525    }
526
527    rational operator % (const rational & r) const {
528      return *this - r * rational(num * r.den / (r.num * den), 1);
```

```
529      }
530
531      rational operator ^ (const bigint & p) const {
532        return rational(num ^ p, den ^ p);
533      }
534
535      rational operator ++(int) { rational t(*this); operator++(); return t; }
536      rational operator --(int) { rational t(*this); operator--(); return t; }
537      rational & operator ++() { *this = *this + 1; return *this; }
538      rational & operator --() { *this = *this - 1; 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 rational & r) { *this = *this / r; return *this; }
543      rational & operator %= (const rational & r) { *this = *this % r; return *this; }
544      rational & operator ^= (const bigint & r) { *this = *this ^ r; return *this; }
545
546      rational operator - () const {
547        return rational(-num, den);
548      }
549
550      rational abs() const {
551        return rational(num.abs(), den);
552      }
553
554      long long to_llong() const {
555        return num.to_llong() / den.to_llong();
556      }
557
558      double to_double() const {
559        return num.to_double() / den.to_double();
560      }
561
562      friend rational abs(const rational & r) {
563        return rational(r.num.abs(), r.den);
564      }
565
566      friend std::istream & operator >> (std::istream & in, rational & r) {
567        std::string s;
568        in >> r.num;
569        r.den = 1;
570        return in;
571      }
572
573      friend std::ostream & operator << (std::ostream & out, const rational & r) {
574        out << r.num << "/" << r.den;
575        return out;
576      }
577
578      //rational in range [0, 1] with precision no greater than prec
579      static rational rand(int prec) {
580        rational r(bigint::rand(prec), bigint::rand(prec));
581        if (r.num > r.den) std::swap(r.num, r.den);
582        return r;
583      }
584    };
585
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> bool operator == (const T & a, const rational & b) { return rational(a) == b; }
591   template<class T> bool 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   template<class T> rational operator % (const T & a, const rational & b) { return rational(a) % b;  }
597   template<class T> rational operator ^ (const T & a, const rational & b) { return rational(a) ^ b;  }
598
599   /*** Example Usage ***/
600
601   #include <cassert>
602   #include <cstdio>
603   #include <ctime>
604   #include <iostream>
605   using namespace std;
606
607   int main() {
608     for (int i = 0; i < 20; i++) {
609       int n = rand() % 100 + 1;
610       bigint a = bigint::rand(n);
611       bigint res = sqrt(a);
612       bigint xx(res * res);
613       bigint yy(res + 1);
614       yy *= yy;
615       assert(xx <= a && yy > a);
616       int m = rand() % n + 1;
617       bigint b = bigint::rand(m) + 1;
618       res = a / b;
619       xx = res * b;
620       yy = b * (res + 1);
621       assert(a >= xx && a < yy);
622     }
623
624     assert("995291497" ==
625       nthroot(bigint("9812985918924981892491829984298981 24"), 4));
626
627     bigint x(5);
628     x = -6;
629     assert(x.to_llong() == -6ll);
630     assert(x.to_string() == "-6");
631
632     clock_t start;
633
634     start = clock();
635     bigint c = bigint::rand(10000) / bigint::rand(2000);
636     cout << "Div␣took␣" << (float)(clock() - start)/CLOCKS_PER_SEC << "s\n";
637
638     start = clock();
639     assert((20^bigint(12345)).size() == 16062);
640     cout << "Pow␣took␣" << (float)(clock() - start)/CLOCKS_PER_SEC << "s\n";
641
642     int nn = -21, dd = 2;
643     rational n(nn, 1), d(dd);
644     cout << (nn % dd) << "\n";
645     cout << (n % d) << "\n";
646     cout << fmod(-5.3, -1.7) << "\n";
```

```
647      cout << rational(-53, 10) % rational(-17, 10) << "\n";
648      cout << rational(-53, 10).abs() << "\n";
649      cout << (rational(-53, 10) ^ 20) << "\n";
650      cout << rational::rand(20) << "\n";
651      return 0;
652    }
```

4.4.3   FFT and Multiplication

```
1   /*
2
3   4.4.3 - Fast Fourier Transform and Multiplication
4
5   A discrete Fourier transform (DFT) converts a list of equally
6   spaced samples of a function into the list of coefficients of
7   a finite combination of complex sinusoids, ordered by their
8   frequencies, that has those same sample values. A Fast Fourier
9   Transform (FFT) rapidly computes the DFT by factorizing the
10  DFT matrix into a product of sparse (mostly zero) factors.
11  The FFT can be used to solve problems such as efficiently
12  multiplying big integers or polynomials
13
14  The fft() function below is a generic function that will
15  work well in many applications beyond just multiplying
16  big integers. While Karatsuba multiplication is ~ O(n^1.58),
17  the complexity of the fft multiplication is only O(n log n).
18
19  Note that mul(string, string) in the following implementation
20  only works for strings of strictly digits from '0' to '9'.
21  It is also easy to adapt this for the bigint class in the
22  previous section. Simply replace the old bigint operator *
23  definition with the following modified version of mul():
24
25    bigint operator * (const bigint & v) const {
26      static const int _base = 10000, _base_digits = 4;
27      vint _a = convert_base(this->a, base_digits, _base_digits);
28      vint _b = convert_base(v.a, base_digits, _base_digits);
29      int len = 32 - __builtin_clz(std::max(_a.size(), _b.size()) - 1);
30      len = 1 << (len + 1);
31      vcd a(len), b(len);
32      for (int i = 0; i < _a.size(); i++) a[i] = cd(_a[i], 0);
33      for (int i = 0; i < _b.size(); i++) b[i] = cd(_b[i], 0);
34      a = fft(a);
35      b = fft(b);
36      for (int i = 0; i < len; i++) {
37        double real = a[i].real() * b[i].real() - a[i].imag() * b[i].imag();
38        a[i].imag() = a[i].imag() * b[i].real() + b[i].imag() * a[i].real();
39        a[i].real() = real;
40      }
41      a = fft(a, true);
42      vll c(len);
43      for (int i = 0; i < len; i++) c[i] = (long long)(a[i].real() + 0.5);
44      bigint res;
45      res.sign = sign * v.sign;
46      for (int i = 0, carry = 0; i < c.size(); i++) {
47        long long cur = c[i] + carry;
48        res.a.push_back((int)(cur % _base));
```

```
49        carry = (int)(cur / _base);
50      }
51      res.a = convert_base(res.a, _base_digits, base_digits);
52      res.trim();
53      return res;
54    }

56  */

58  #include <algorithm> /* std::max(), std::reverse() */
59  #include <cmath>      /* M_PI, cos(), sin() */
60  #include <complex>
61  #include <iomanip>    /* std::setw(), std::setfill() */
62  #include <sstream>
63  #include <string>
64  #include <vector>

66  typedef std::complex<double> cd;
67  typedef std::vector<cd> vcd;

69  vcd fft(const vcd & v, bool inverse = false) {
70    static const double PI = acos(-1.0);
71    int n = v.size(), k = 0, high1 = -1;
72    while ((1 << k) < n) k++;
73    std::vector<int> rev(n);
74    rev[0] = 0;
75    for (int i = 1; i < n; i++) {
76      if ((i & (i - 1)) == 0) high1++;
77      rev[i] = rev[i ^ (1 << high1)];
78      rev[i] |= (1 << (k - high1 - 1));
79    }
80    vcd roots(n), res(n);
81    for (int i = 0; i < n; i++) {
82      double alpha = 2 * PI * i / n;
83      roots[i] = cd(cos(alpha), sin(alpha));
84    }
85    for (int i = 0; i < n; i++) res[i] = v[rev[i]];
86    for (int len = 1; len < n; len <<= 1) {
87      vcd tmp(n);
88      int rstep = roots.size() / (len * 2);
89      for (int pdest = 0; pdest < n;) {
90        int p1 = pdest;
91        for (int i = 0; i < len; i++) {
92          cd val = roots[i * rstep] * res[p1 + len];
93          tmp[pdest] = res[p1] + val;
94          tmp[pdest + len] = res[p1] - val;
95          pdest++, p1++;
96        }
97        pdest += len;
98      }
99      res.swap(tmp);
100   }
101   if (inverse) {
102     for (int i = 0; i < (int)res.size(); i++) res[i] /= v.size();
103     std::reverse(res.begin() + 1, res.end());
104   }
105   return res;
106 }
107
```

```cpp
108  typedef std::vector<long long> vll;
109
110  vll mul(const vll & va, const vll & vb) {
111    int len = 32 - __builtin_clz(std::max(va.size(), vb.size()) - 1);
112    len = 1 << (len + 1);
113    vcd a(len), b(len);
114    for (int i = 0; i < (int)va.size(); i++) a[i] = cd(va[i], 0);
115    for (int i = 0; i < (int)vb.size(); i++) b[i] = cd(vb[i], 0);
116    a = fft(a);
117    b = fft(b);
118    for (int i = 0; i < len; i++) {
119      double real = a[i].real() * b[i].real() - a[i].imag() * b[i].imag();
120      a[i].imag() = a[i].imag() * b[i].real() + b[i].imag() * a[i].real();
121      a[i].real() = real;
122    }
123    a = fft(a, true);
124    vll res(len);
125    for (int i = 0; i < len; i++) res[i] = (long long)(a[i].real() + 0.5);
126    return res;
127  }
128
129  const int base = 10000, base_digits = 4;
130
131  std::string mul(const std::string & as, const std::string & bs) {
132    vll a, b;
133    for (int i = as.size() - 1; i >= 0; i -= base_digits) {
134      int x = 0;
135      for (int j = std::max(0, i - base_digits + 1); j <= i; j++)
136        x = x * 10 + as[j] - '0';
137      a.push_back(x);
138    }
139    for (int i = bs.size() - 1; i >= 0; i -= base_digits) {
140      int x = 0;
141      for (int j = std::max(0, i - base_digits + 1); j <= i; j++)
142        x = x * 10 + bs[j] - '0';
143      b.push_back(x);
144    }
145    vll c = mul(a, b);
146    long long carry = 0;
147    for (int i = 0; i < (int)c.size(); i++) {
148      c[i] += carry;
149      carry = c[i] / base;
150      c[i] %= base;
151    }
152    while (c.back() == 0) c.pop_back();
153    if (c.empty()) c.push_back(0);
154    std::ostringstream oss;
155    oss << (c.empty() ? 0 : c.back());
156    for (int i = c.size() - 2; i >= 0; i--)
157      oss << std::setw(base_digits) << std::setfill('0') << c[i];
158    return oss.str();
159  }
160
161  /*** Example Usage ***/
162
163  #include <cassert>
164
165  int main() {
166    assert(mul("98904189", "244212") == "24153589804068");
```

```
167    return 0;
168  }
```

4.5   Linear Algebra

4.5.1   Matrix Class

```
1    /*
2
3    4.5.1 - Matrix Class
4
5    Basic matrix class with support for arithmetic operations
6    as well as matrix multiplication and exponentiation. You
7    can access/modify indices using m(r, c) or m[r][c]. You
8    can also treat it as a 2d vector, since the cast operator
9    to a reference to its internal 2d vector is defined. This
10   makes it compatible with the 2d vector functions such as
11   det() and lu_decompose() in later sections.
12
13   */
14
15   #include <ostream>
16   #include <stdexcept> /* std::runtime_error() */
17   #include <vector>
18
19   template<class val_t> class matrix {
20     int r, c;
21     std::vector<std::vector<val_t> > mat;
22
23    public:
24     matrix(int rows, int cols, val_t init = val_t()) {
25       r = rows;
26       c = cols;
27       mat.resize(r, std::vector<val_t>(c, init));
28     }
29
30     matrix(const std::vector<std::vector<val_t> > & m) {
31       r = m.size();
32       c = m[0].size();
33       mat = m;
34       mat.resize(r, std::vector<val_t>(c));
35     }
36
37     template<size_t rows, size_t cols>
38     matrix(val_t (&init)[rows][cols]) {
39       r = rows;
40       c = cols;
41       mat.resize(r, std::vector<val_t>(c));
42       for (int i = 0; i < r; i++)
43         for (int j = 0; j < c; j++)
44           mat[i][j] = init[i][j];
45     }
46
47     operator std::vector<std::vector<val_t> > &() { return mat; }
48     val_t & operator() (int r, int c) { return mat[r][c]; }
49     std::vector<val_t> & operator[] (int r) { return mat[r]; }
```

```
50    val_t at(int r, int c) const { return mat[r][c]; }
51    int rows() const { return r; }
52    int cols() const { return c; }
53
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    friend bool operator == (const matrix & a, const matrix & b) { return a.mat == b.mat; }
59    friend bool operator != (const matrix & a, const matrix & b) { return a.mat != b.mat; }
60
61    friend matrix operator + (const matrix & a, const matrix & b) {
62      if (a.r != b.r || a.c != b.c)
63        throw std::runtime_error("Matrix dimensions don't match.");
64      matrix res(a);
65      for (int i = 0; i < res.r; i++)
66        for (int j = 0; j < res.c; j++)
67          res.mat[i][j] += b.mat[i][j];
68      return res;
69    }
70
71    friend matrix operator - (const matrix & a, const matrix & b) {
72      if (a.r != b.r || a.c != b.c)
73        throw std::runtime_error("Matrix dimensions don't match.");
74      matrix res(a);
75      for (int i = 0; i < a.r; i++)
76        for (int j = 0; j < a.c; j++)
77          res.mat[i][j] -= b.mat[i][j];
78      return res;
79    }
80
81    friend matrix operator * (const matrix & a, const matrix & b) {
82      if (a.c != b.r)
83        throw std::runtime_error("# of a cols must equal # of b rows.");
84      matrix res(a.r, b.c, 0);
85      for (int i = 0; i < a.r; i++)
86        for (int j = 0; j < b.c; j++)
87          for (int k = 0; k < a.c; k++)
88            res.mat[i][j] += a.mat[i][k] * b.mat[k][j];
89      return res;
90    }
91
92    friend matrix operator + (const matrix & a, const val_t & v) {
93      matrix res(a);
94      for (int i = 0; i < a.r; i++)
95        for (int j = 0; j < a.c; j++) res.mat[i][j] += v;
96      return res;
97    }
98
99    friend matrix operator - (const matrix & a, const val_t & v) {
100     matrix res(a);
101     for (int i = 0; i < a.r; i++)
102       for (int j = 0; j < a.c; j++) res.mat[i][j] -= v;
103     return res;
104   }
105
106   friend matrix operator * (const matrix & a, const val_t & v) {
107     matrix res(a);
108     for (int i = 0; i < a.r; i++)
```

```
109            for (int j = 0; j < a.c; j++) res.mat[i][j] *= v;
110          return res;
111        }
112
113        friend matrix operator / (const matrix & a, const val_t & v) {
114          matrix res(a);
115          for (int i = 0; i < a.r; i++)
116            for (int j = 0; j < a.c; j++)
117              res.mat[i][j] /= v;
118          return res;
119        }
120
121        //raise matrix to the n-th power. precondition: a must be a square matrix
122        friend matrix operator ^ (const matrix & a, unsigned int n) {
123          if (a.r != a.c)
124            throw std::runtime_error("Matrix must be square for exponentiation.");
125          if (n == 0) return identity_matrix(a.r);
126          if (n % 2 == 0) return (a * a) ^ (n / 2);
127          return a * (a ^ (n - 1));
128        }
129
130        //returns a^1 + a^2 + ... + a^n
131        friend matrix powsum(const matrix & a, unsigned int n) {
132          if (n == 0) return matrix(a.r, a.r);
133          if (n % 2 == 0)
134            return powsum(a, n / 2) * (identity_matrix(a.r) + (a ^ (n / 2)));
135          return a + a * powsum(a, n - 1);
136        }
137
138        matrix & operator += (const matrix & m) { *this = *this + m; return *this; }
139        matrix & operator -= (const matrix & m) { *this = *this - m; return *this; }
140        matrix & operator *= (const matrix & m) { *this = *this * m; 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 *= (const val_t & v)  { *this = *this * v; return *this; }
144        matrix & operator /= (const val_t & v)  { *this = *this / v; return *this; }
145        matrix & operator ^= (unsigned int n)   { *this = *this ^ n; return *this; }
146
147        static matrix identity_matrix(int n) {
148          matrix res(n, n);
149          for (int i = 0; i < n; i++) res[i][i] = 1;
150          return res;
151        }
152
153        friend std::ostream & operator << (std::ostream & out, const matrix & m) {
154          out << "[";
155          for (int i = 0; i < m.r; i++) {
156            out << (i > 0 ? ",[" : "[");
157            for (int j = 0; j < m.c; j++)
158              out << (j > 0 ? "," : "") << m.mat[i][j];
159            out << "]";
160          }
161          out << "]";
162          return out;
163        }
164  };
165
166  /*** Example Usage ***/
167
```

```
168   #include <cassert>
169   #include <iostream>
170   using namespace std;
171
172   int main() {
173     int a[2][2] = {{1,8}, {5,9}};
174     matrix<int> m(5, 5, 10), m2(a);
175     m += 10;
176     m[0][0] += 10;
177     assert(m[0][0] == 30 && m[1][1] == 20);
178     assert(powsum(m2, 3) == m2 + m2*m2 + (m2^3));
179     return 0;
180   }
```

4.5.2   Determinant (Gauss)

```
1    /*
2
3    4.5.2 - Determinant (Gauss's Method)
4
5    The following are ways to compute the determinant of a
6    matrix directly using Gaussian elimination. See the
7    following section for a generalized solution using LU
8    decompositions. Since the determinant can get very large,
9    look out for overflows and floating-point inaccuracies.
10   Bignums are recommended for maximal correctness.
11
12   Complexity: O(N^3), except for the adjustment for
13   overflow in the integer det() function.
14
15   Precondition: All input matrices must be square.
16
17   */
18
19   #include <algorithm> /* std::swap() */
20   #include <cassert>
21   #include <cmath>      /* fabs() */
22   #include <map>
23   #include <vector>
24
25   static const double eps = 1e-10;
26   typedef std::vector<std::vector<int> > vvi;
27   typedef std::vector<std::vector<double> > vvd;
28
29   double det(vvd a) {
30     int n = a.size();
31     assert(!a.empty() && n == (int)a[0].size());
32     double res = 1;
33     std::vector<bool> used(n, false);
34     for (int i = 0; i < n; i++) {
35       int p;
36       for (p = 0; p < n; p++)
37         if (!used[p] && fabs(a[p][i]) > eps)
38           break;
39       if (p >= n) return 0;
40       res *= a[p][i];
41       used[p] = true;
```

```
42        double z = 1 / a[p][i];
43        for (int j = 0; j < n; j++) a[p][j] *= z;
44        for (int j = 0; j < n; j++) {
45          if (j == p) continue;
46          z = a[j][i];
47          for (int k = 0; k < n; k++)
48            a[j][k] -= z * a[p][k];
49        }
50      }
51      return res;
52    }
53
54    /*
55
56    Determinant of Integer Matrix
57
58    This is prone to overflow, so it is recommended you use your
59    own bigint class instead of long long. At the end of this
60    function, the final answer is found as a product of powers.
61    You have two choices: change the "#if 0" to "#if 1" and use
62    the naive method to compute this product and risk overflow,
63    or keep it as "#if 0" and try to make the situation better
64    through prime factorization (less efficient). Note that
65    even in the prime factorization method, overflow may happen
66    if the final answer is too big for a long long.
67
68    */
69
70    //C++98 doesn't have an abs() for long long
71    template<class T> inline T _abs(const T & x) {
72      return x < 0 ? -x : x;
73    }
74
75    long long det(const vvi & a) {
76      int n = a.size();
77      assert(!a.empty() && n == (int)a[0].size());
78      long long b[n][n], det = 1;
79      for (int i = 0; i < n; i++)
80        for (int j = 0; j < n; j++) b[i][j] = a[i][j];
81      int sign = 1, exponent[n];
82      for (int i = 0; i < n; i++) {
83        exponent[i] = 0;
84        int k = i;
85        for (int j = i + 1; j < n; j++) {
86          if (b[k][i] == 0 || (b[j][i] != 0 && _abs(b[k][i]) > _abs(b[j][i])))
87            k = j;
88        }
89        if (b[k][i] == 0) return 0;
90        if (i != k) {
91          sign = -sign;
92          for (int j = 0; j < n; j++)
93            std::swap(b[i][j], b[k][j]);
94        }
95        exponent[i]++;
96        for (int j = i + 1; j < n; j++)
97          if (b[j][i] != 0) {
98            for (int p = i + 1; p < n; ++p)
99              b[j][p] = b[j][p] * b[i][i] - b[i][p] * b[j][i];
100             exponent[i]--;
```

```
101        }
102      }
103
104  #if 0
105    for (int i = 0; i < n; i++)
106      for (; exponent[i] > 0; exponent[i]--)
107        det *= b[i][i];
108    for (int i = 0; i < n; i++)
109      for (; exponent[i] < 0; exponent[i]++)
110        det /= b[i][i];
111  #else
112    std::map<long long, int> m;
113    for (int i = 0; i < n; i++) {
114      long long x = b[i][i];
115      for (long long d = 2; ; d++) {
116        long long power = 0, quo = x / d, rem = x - quo * d;
117        if (d > quo || (d == quo && rem > 0)) break;
118        for (; rem == 0; rem = x - quo * d) {
119          power++;
120          x = quo;
121          quo = x / d;
122        }
123        if (power > 0) m[d] += power * exponent[i];
124      }
125      if (x > 1) m[x] += exponent[i];
126    }
127    std::map<long long, int>::iterator it;
128    for (it = m.begin(); it != m.end(); ++it)
129      for (int i = 0; i < it->second; i++)
130        det *= it->first;
131  #endif
132
133    return sign < 0 ? -det : det;
134  }
135
136  /*** Example Usage ***/
137
138  #include <iostream>
139  using namespace std;
140
141  int main() {
142    const int n = 3;
143    int a[n][n] = {{6,1,1},{4,-2,5},{2,8,7}};
144    vvi v1(n);
145    vvd v2(n);
146    for (int i = 0; i < n; i++) {
147      v1[i] = vector<int>(a[i], a[i] + n);
148      v2[i] = vector<double>(a[i], a[i] + n);
149    }
150    int d1 = det(v1);
151    int d2 = (int)det(v2);
152    assert(d1 == d2 && d2 == -306);
153    return 0;
154  }
```

4.5.3   Gaussian Elimination

```
1   /*
2
3   4.5.3 - System Solver (Gaussian Elimination)
4
5   Given a system of m linear equations with n unknowns:
6
7   A(1,1)*x(1) + A(1,2)*x(2) + ... + A(1,n)*x(n) = B(1)
8   A(2,1)*x(1) + A(2,2)*x(2) + ... + A(2,n)*x(n) = B(2)
9                         ...
10  A(m,1)*x(1) + A(m,2)*x(2) + ... + A(m,n)*x(n) = B(m)
11
12  For any system of linear equations, there will either
13  be no solution (in 2d, lines are parallel), a single
14  solution (in 2d, the lines intersect at a point), or
15  or infinite solutions (in 2d, lines are the same).
16
17  Using Gaussian elimination in O(n^3), this program
18  solves for the values of x(1) ... x(n) or determines
19  that no unique solution of x() exists. Note that
20  the implementation below uses 0-based indices.
21
22  */
23
24  #include <algorithm> /* std::swap() */
25  #include <cmath>      /* fabs() */
26  #include <vector>
27
28  const double eps = 1e-9;
29  typedef std::vector<double> vd;
30  typedef std::vector<vd> vvd;
31
32  //note: A[i][n] stores B[i]
33  //if no unique solution found, returns empty vector
34  vd solve_system(vvd A) {
35    int m = A.size(), n = A[0].size() - 1;
36    vd x(n);
37    if (n > m) goto fail;
38    for (int k = 0; k < n; k++) {
39      double mv = 0;
40      int mi = -1;
41      for (int i = k; i < m; i++)
42        if (mv < fabs(A[i][k])) {
43          mv = fabs(A[i][k]);
44          mi = i;
45        }
46      if (mv < eps) goto fail;
47      for (int i = 0; i <= n; i++)
48        std::swap(A[mi][i], A[k][i]);
49      for (int i = k + 1; i < m; i++) {
50        double v = A[i][k] / A[k][k];
51        for (int j = k; j <= n; j++)
52          A[i][j] -= v * A[k][j];
53        A[i][k] = 0;
54      }
55    }
56    for (int i = n; i < m; i++)
57      if (fabs(A[i][n]) > eps) goto fail;
58    for (int i = n - 1; i >= 0; i--) {
59      if (fabs(A[i][i]) < eps) goto fail;
```

```
60      double v = 0;
61      for (int j = i + 1; j < n; j++)
62        v += A[i][j] * x[j];
63      v = A[i][n] - v;
64      x[i] = v / A[i][i];
65    }
66    return x;
67  fail:
68    return vd();
69  }
70
71  /*** Example Usage (wcipeg.com/problem/syssolve) ***/
72
73  #include <iostream>
74  using namespace std;
75
76  int main() {
77    int n, m;
78    cin >> n >> m;
79    vvd a(m, vd(n + 1));
80    for (int i = 0; i < m; i++)
81      for (int j = 0; j <= n; j++)
82        cin >> a[i][j];
83    vd x = solve_system(a);
84    if (x.empty()) {
85      cout << "NO UNIQUE SOLUTION\n";
86    } else {
87      cout.precision(6);
88      for (int i = 0; i < n; i++)
89        cout << fixed << x[i] << "\n";
90    }
91    return 0;
92  }
```

4.5.4   LU Decomposition

```
1  /*
2
3  4.5.4 - LU Decomposition
4
5  The LU (lower upper) decomposition of a matrix is a factorization
6  of a matrix as the product of a lower triangular matrix and an
7  upper triangular matrix. With the LU decomposition, we can solve
8  many problems, including the determinant of the matrix, a systems
9  of linear equations, and the inverse of a matrix.
10
11  Note: in the following implementation, each call to det(),
12  solve_system(), and inverse() recomputes the lu decomposition.
13  For the same matrix, you should precompute the lu decomposition
14  and reuse it for several of these operations afterwards.
15
16  Complexity: O(n^3) for lu_decompose(). det() uses the running time
17  of lu_decompose(), plus an addition O(n) term. solve_system() and
18  inverse() both have the running time of lu_decompose(), plus an
19  additional O(n^3) term.
20
21  */
```

```
22
23   #include <algorithm> /* std::swap() */
24   #include <cassert>
25   #include <cmath>       /* fabs() */
26   #include <vector>
27
28   static const double eps = 1e-10;
29   typedef std::vector<double> vd;
30   typedef std::vector<vd> vvd;
31
32   /*
33
34   LU decomposition with Gauss-Jordan elimination. This is generalized
35   for rectangular matrices. Since the resulting L and U matrices have
36   all mutually exclusive 0's (except when i == j), we can merge them
37   into a single LU matrix to save memory. Note: l[i][i] = 1 for all i.
38
39   Optionally determine the permutation vector p. If an array p is
40   passed, p[i] will be populated such that p[i] is the only column of
41   the i-th row of the permutation matrix that is equal to 1.
42
43   Returns: a matrix m, the merged lower/upper triangular matrix:
44           m[i][j] = l[i][j] (for i > j) or u[i][j] (for i <= j)
45
46   */
47
48   vvd lu_decompose(vvd a, int * detsign = 0, int * p = 0) {
49     int n = a.size(), m = a[0].size();
50     int sign = 1;
51     if (p != 0)
52       for (int i = 0; i < n; i++) p[i] = i;
53     for (int r = 0, c = 0; r < n && c < m; r++, c++) {
54       int pr = r;
55       for (int i = r + 1; i < n; i++)
56         if (fabs(a[i][c]) > fabs(a[pr][c]))
57           pr = i;
58       if (fabs(a[pr][c]) <= eps) {
59         r--;
60         continue;
61       }
62       if (pr != r) {
63         if (p != 0) std::swap(p[r], p[pr]);
64         sign = -sign;
65         for (int i = 0; i < m; i++)
66           std::swap(a[r][i], a[pr][i]);
67       }
68       for (int s = r + 1; s < n; s++) {
69         a[s][c] /= a[r][c];
70         for (int d = c + 1; d < m; d++)
71           a[s][d] -= a[s][c] * a[r][d];
72       }
73     }
74     if (detsign != 0) *detsign = sign;
75     return a;
76   }
77
78   double getl(const vvd & lu, int i, int j) {
79     if (i > j) return lu[i][j];
80     return i < j ? 0.0 : 1.0;
```

```
81   }
82
83   double getu(const vvd & lu, int i, int j) {
84     return i <= j ? lu[i][j] : 0.0;
85   }
86
87   //Precondition: A is square matrix.
88   double det(const vvd & a) {
89     int n = a.size(), detsign;
90     assert(!a.empty() && n == (int)a[0].size());
91     vvd lu = lu_decompose(a, &detsign);
92     double det = 1;
93     for (int i = 0; i < n; i++)
94       det *= lu[i][i];
95     return detsign < 0 ? -det : det;
96   }
97
98   /*
99
100  Solves system of linear equations with forward/backwards
101  substitution. Precondition: A must be n*n and B must be n*m.
102  Returns: an n by m matrix X such that A*X = B.
103
104  */
105
106  vvd solve_system(const vvd & a, const vvd & b) {
107    int n = b.size(), m = b[0].size();
108    assert(!a.empty() && n == (int)a.size() && n == (int)a[0].size());
109    int detsign, p[a.size()];
110    vvd lu = lu_decompose(a, &detsign, p);
111    //forward substitute for Y in L*Y = B
112    vvd y(n, vd(m));
113    for (int j = 0; j < m; j++) {
114      y[0][j] = b[p[0]][j] / getl(lu, 0, 0);
115      for (int i = 1; i < n; i++) {
116        double s = 0;
117        for (int k = 0; k < i; k++)
118          s += getl(lu, i, k) * y[k][j];
119        y[i][j] = (b[p[i]][j] - s) / getl(lu, i, i);
120      }
121    }
122    //backward substitute for X in U*X = Y
123    vvd x(n, vd(m));
124    for (int j = 0; j < m; j++) {
125      x[n - 1][j] = y[n - 1][j] / getu(lu, n-1, n-1);
126      for (int i = n - 2; i >= 0; i--) {
127        double s = 0;
128        for (int k = i + 1; k < n; k++)
129          s += getu(lu, i, k) * x[k][j];
130        x[i][j] = (y[i][j] - s) / getu(lu, i, i);
131      }
132    }
133    return x;
134  }
135
136  /*
137
138  Find the inverse A^-1 of a matrix A. The inverse of a matrix
139  satisfies A * A^-1 = I, where I is the identity matrix (for
```

```
140    all pairs (i, j), I[i][j] = 1 iff i = j, else I[i][j] = 0).
141    The inverse of a matrix exists if and only if det(a) is not 0.
142    We're lazy, so we just generate I and call solve_system().
143
144    Precondition: A is a square and det(A) != 0.
145
146    */
147
148    vvd inverse(const vvd & a) {
149      int n = a.size();
150      assert(!a.empty() && n == (int)a[0].size());
151      vvd I(n, vd(n));
152      for (int i = 0; i < n; i++) I[i][i] = 1;
153      return solve_system(a, I);
154    }
155
156    /*** Example Usage ***/
157
158    #include <cstdio>
159    #include <iostream>
160    using namespace std;
161
162    void print(const vvd & m) {
163      cout << "[";
164      for (int i = 0; i < (int)m.size(); i++) {
165        cout << (i > 0 ? ",[" : "[");
166        for (int j = 0; j < (int)m[0].size(); j++)
167          cout << (j > 0 ? "," : "") << m[i][j];
168        cout << "]";
169      }
170      cout << "]\n";
171    }
172
173    void printlu(const vvd & lu) {
174      printf("L:\n");
175      for (int i = 0; i < (int)lu.size(); i++) {
176        for (int j = 0; j < (int)lu[0].size(); j++)
177          printf("%10.5f ", getl(lu, i, j));
178        printf("\n");
179      }
180      printf("U:\n");
181      for (int i = 0; i < (int)lu.size(); i++) {
182        for (int j = 0; j < (int)lu[0].size(); j++)
183          printf("%10.5f ", getu(lu, i, j));
184        printf("\n");
185      }
186    }
187
188    int main() {
189      { //determinant of 3x3
190        const int n = 3;
191        double a[n][n] = {{1,3,5},{2,4,7},{1,1,0}};
192        vvd v(n);
193        for (int i = 0; i < n; i++)
194          v[i] = vector<double>(a[i], a[i] + n);
195        printlu(lu_decompose(v));
196        cout << "determinant: " << det(v) << "\n"; //4
197      }
198
```

```
199    { //determinant of 4x4
200      const int n = 4;
201      double a[n][n] = {{11,9,24,2},{1,5,2,6},{3,17,18,1},{2,5,7,1}};
202      vvd v(n);
203      for (int i = 0; i < n; i++)
204        v[i] = vector<double>(a[i], a[i] + n);
205      printlu(lu_decompose(v));
206      cout << "determinant:␣" << det(v) << "\n"; //284
207    }
208
209    { //solve for [x, y] in x + 3y = 4 && 2x + 3y = 6
210      const int n = 2;
211      double a[n][n] = {{1,3},{2,3}};
212      double b[n] = {4, 6};
213      vvd va(n), vb(n);
214      for (int i = 0; i < n; i++) {
215        va[i] = vector<double>(a[i], a[i] + n);
216        vb[i] = vector<double>(1, b[i]);
217      }
218      vvd x = solve_system(va, vb);
219      for (int i = 0; i < n; i++) {
220        assert(fabs(a[i][0]*x[0][0] + a[i][1]*x[1][0] - b[i]) < eps);
221      }
222    }
223
224    { //find inverse by solving a system
225      const int n = 2;
226      double a[n][n] = {{2,3},{1,2}};
227      vvd v(n);
228      for (int i = 0; i < n; i++)
229        v[i] = vector<double>(a[i], a[i] + n);
230      print(inverse(v)); //[[2,-3],[-1,2]]
231    }
232    return 0;
233  }
```

4.5.5   Simplex Algorithm

```
1  /*
2
3  4.5.5 - Linear Programming using Simplex Algorithm
4
5  Description: The canonical form of a linear programming
6  problem is to maximize c^T*x, subject to Ax <= b, and x >= 0.
7  where x is the vector of variables (to be solved), c and b
8  are vectors of (known) coefficients, A is a (known) matrix of
9  coefficients, and (.)^T is the matrix transpose. The following
10 implementation solves n variables in a system of m constraints.
11
12 Precondition: ab has dimensions m by n+1 and c has length n+1.
13
14 Complexity: The simplex method is remarkably efficient in
15 practice, usually taking 2m or 3m iterations, converging in
16 expected polynomial time for certain distributions of random
17 inputs. However, its worst-case complexity is exponential,
18 and can be demonstrated with carefully constructed examples.
19
```

```
20   */
21
22   #include <algorithm> /* std::swap() */
23   #include <cfloat>    /* DBL_MAX */
24   #include <cmath>     /* fabs() */
25   #include <vector>
26
27   typedef std::vector<double> vd;
28   typedef std::vector<vd> vvd;
29
30   //ab[i][0..n-1] stores A and ab[i][n] stores B
31   vd simplex(const vvd & ab, const vd & c, bool max = true) {
32     const double eps = 1e-10;
33     int n = c.size() - 1, m = ab.size();
34     vvd ts(m + 2, vd(n + 2));
35     ts[1][1] = max ? c[n] : -c[n];
36     for (int j = 1; j <= n; j++)
37       ts[1][j + 1] = max ? c[j - 1] : -c[j - 1];
38     for (int i = 1; i <= m; i++) {
39       for (int j = 1; j <= n; j++)
40         ts[i + 1][j + 1] = -ab[i - 1][j - 1];
41       ts[i + 1][1] = ab[i - 1][n];
42     }
43     for (int j = 1; j <= n; j++)
44       ts[0][j + 1] = j;
45     for (int i = n + 1; i <= n + m; i++)
46       ts[i - n + 1][0] = i;
47     double p1 = 0.0, p2 = 0.0;
48     bool done = true;
49     do {
50       double mn = DBL_MAX, xmax = 0.0, v;
51       for (int j = 2; j <= n + 1; j++)
52         if (ts[1][j] > 0.0 && ts[1][j] > xmax) {
53           p2 = j;
54           xmax = ts[1][j];
55         }
56       for (int i = 2; i <= m + 1; i++) {
57         v = fabs(ts[i][1] / ts[i][p2]);
58         if (ts[i][p2] < 0.0 && mn > v) {
59           mn = v;
60           p1 = i;
61         }
62       }
63       std::swap(ts[p1][0], ts[0][p2]);
64       for (int i = 1; i <= m + 1; i++) {
65         if (i == p1) continue;
66         for (int j = 1; j <= n + 1; j++)
67           if (j != p2)
68             ts[i][j] -= ts[p1][j] * ts[i][p2] / ts[p1][p2];
69       }
70       ts[p1][p2] = 1.0 / ts[p1][p2];
71       for (int j = 1; j <= n + 1; j++) {
72         if (j != p2)
73           ts[p1][j] *= fabs(ts[p1][p2]);
74       }
75       for (int i = 1; i <= m + 1; i++) {
76         if (i != p1)
77           ts[i][p2] *= ts[p1][p2];
78       }
```

```
79        for (int i = 2; i <= m + 1; i++)
80           if (ts[i][1] < 0.0) return vd(); //no solution
81        done = true;
82        for (int j = 2; j <= n + 1; j++)
83           if (ts[1][j] > 0) done = false;
84      } while (!done);
85      vd res;
86      for (int i = 1; i <= n; i++)
87        for (int j = 2; j <= m + 1; j++)
88           if (fabs(ts[j][0] - i) <= eps)
89             res.push_back(ts[j][1]);
90      //the solution is stored in ts[1][1]
91      return res;
92    }
93
94    /*** Example Usage ***/
95
96    #include <iostream>
97    using namespace std;
98
99    /*
100     Maximize 3x + 4y + 5, subject to x, y >= 0 and:
101         -2x +    1y <=   0
102          1x + 0.85y <=   9
103          1x +    2y <= 14
104
105     Note: The solution is 38.3043 at (5.30435, 4.34783).
106    */
107
108    int main() {
109      const int n = 2, m = 3;
110      double ab[m][n + 1] = {{-2, 1, 0}, {1, 0.85, 9}, {1, 2, 14}};
111      double c[n + 1] = {3, 4, 5};
112      vvd vab(m, vd(n + 1));
113      vd vc(c, c + n + 1);
114      for (int i = 0; i < m; i++) {
115        for (int j = 0; j <= n; j++)
116          vab[i][j] = ab[i][j];
117      }
118      vd x = simplex(vab, vc);
119      if (x.empty()) {
120        cout << "No␣solution.\n";
121      } else {
122        double solval = c[n];
123        for (int i = 0; i < (int)x.size(); i++)
124          solval += c[i] * x[i];
125        cout << "Solution␣=␣" << solval;
126        cout << "␣at␣(" << x[0];
127        for (int i = 1; i < (int)x.size(); i++)
128          cout << ",␣" << x[i];
129        cout << ").\n";
130      }
131      return 0;
132    }
```

4.6 Root-Finding

4.6.1 Real Root Finding (Differentiation)

```
1   /*
2
3   4.6.1 - Real Root Finding (Differentiation)
4
5   Real roots can be found via binary searching, a.k.a the bisection
6   method. If two x-coordinates evaluate to y-coordinates that have
7   opposite signs, a root must exist between them. For a polynomial
8   function, at most 1 root lies between adjacent local extrema.
9   Since local extrema exist where the derivative equals 0, we can
10  break root-finding into the subproblem of finding the roots of
11  the derivative. Recursively solve for local extrema until we get
12  to a base case of degree 0. For each set of local extrema found,
13  binary search between pairs of extrema for a root. This method is
14  easy, robust, and allows us to find the root to an arbitrary level
15  of accuracy. We're limited only by the precision of the arithmetic.
16
17  Complexity: For a degree N polynomial, repeatedly differentiating
18  it will take N + (N-1) + ... + 1 = O(N^2) operations. At each step
19  we binary search the number of times equal to the current degree.
20  If we want to make roots precise to eps=10^-P, each binary search
21  will take O(log P). Thus the overall complexity is O(N^2 log P).
22
23  */
24
25  #include <cmath>   /* fabsl(), powl() */
26  #include <limits>  /* std::numeric_limits<>::quiet_NaN() */
27  #include <utility> /* std::pair<> */
28  #include <vector>
29
30  typedef long double Double;
31  typedef std::vector<std::pair<Double, int> > poly;
32
33  const Double epsa = 1e-11; //required precision of roots in absolute error
34  const Double epsr = 1e-15; //required precision of roots in relative error
35  const Double eps0 = 1e-17; //x is considered a root if fabs(eval(x))<=eps0
36  const Double inf = 1e20;   //[-inf, inf] is the range of roots to consider
37  const Double NaN = std::numeric_limits<Double>::quiet_NaN();
38
39  Double eval(const poly & p, Double x) {
40    Double res = 0;
41    for (int i = 0; i < (int)p.size(); i++)
42      res += p[i].first * powl(x, p[i].second);
43    return res;
44  }
45
46  Double find_root(const poly & p, Double x1, Double x2) {
47    Double y1 = eval(p, x1), y2 = eval(p, x2);
48    if (fabsl(y1) <= eps0) return x1;
49    bool neg1 = (y1 < 0), neg2 = (y2 < 0);
50    if (fabsl(y2) <= eps0 || neg1 == neg2) return NaN;
51    while (x2 - x1 > epsa && x1 * (1 + epsr) < x2 && x2 * (1 + epsr) > x1) {
52      Double x = (x1 + x2) / 2;
53      ((eval(p, x) < 0) == neg1 ? x1 : x2) = x;
```

```
54      }
55      return x1;
56   }
57
58   std::vector<Double> find_all_roots(const poly & p) {
59      poly dif;
60      for (int i = 0; i < (int)p.size(); i++)
61        if (p[i].second > 0)
62          dif.push_back(std::make_pair(p[i].first * p[i].second, p[i].second - 1));
63      if (dif.empty()) return std::vector<Double>();
64      std::vector<Double> res, r = find_all_roots(dif);
65      r.insert(r.begin(), -inf);
66      r.push_back(inf);
67      for (int i = 0; i < (int)r.size() - 1; i++) {
68        Double root = find_root(p, r[i], r[i + 1]);
69        if (root != root) continue; //NaN, not found
70        if (res.empty() || root != res.back())
71          res.push_back(root);
72      }
73      return res;
74   }
75
76   /*** Example Usage (http://wcipeg.com/problem/rootsolve) ***/
77
78   #include <iostream>
79   using namespace std;
80
81   int main() {
82      int n, d;
83      Double c;
84      poly p;
85      cin >> n;
86      for (int i = 0; i < n; i++) {
87        cin >> c >> d;
88        p.push_back(make_pair(c, d));
89      }
90      vector<Double> sol = find_all_roots(p);
91      if (sol.empty()) {
92        cout << "NO REAL ROOTS\n";
93      } else {
94        cout.precision(9);
95        for (int i = 0; i < (int)sol.size(); i++)
96          cout << fixed << sol[i] << "\n";
97      }
98      return 0;
99   }
```

4.6.2   Complex Root Finding (Laguerre's)

```
1   /*
2
3   4.6.2 - Complex Root Finding (Laguerre's Method)
4
5   Laguerre's method can be used to not only find complex roots of
6   a polynomial, the polynomial may also have complex coefficients.
7   From extensive empirical study, Laguerre's method is observed to
8   be very close to being a "sure-fire" method, as it is almost
```

```
 9   guaranteed to always converge to some root of the polynomial
10   regardless of what initial guess is chosen.
11
12   */
13
14   #include <complex>
15   #include <cstdlib> /* rand(), RAND_MAX */
16   #include <vector>
17
18   typedef long double Double;
19   typedef std::complex<Double> cdouble;
20   typedef std::vector<cdouble> poly;
21
22   const Double eps = 1e-12;
23
24   std::pair<poly, cdouble> horner(const poly & a, const cdouble & x) {
25     int n = a.size();
26     poly b = poly(std::max(1, n - 1));
27     for (int i = n - 1; i > 0; i--)
28       b[i - 1] = a[i] + (i < n - 1 ? b[i] * x : 0);
29     return std::make_pair(b, a[0] + b[0] * x);
30   }
31
32   cdouble eval(const poly & p, const cdouble & x) {
33     return horner(p, x).second;
34   }
35
36   poly derivative(const poly & p) {
37     int n = p.size();
38     poly r(std::max(1, n - 1));
39     for(int i = 1; i < n; i++)
40       r[i - 1] = p[i] * cdouble(i);
41     return r;
42   }
43
44   int comp(const cdouble & x, const cdouble & y) {
45     Double diff = std::abs(x) - std::abs(y);
46     return diff < -eps ? -1 : (diff > eps ? 1 : 0);
47   }
48
49   cdouble find_one_root(const poly & p, cdouble x) {
50     int n = p.size() - 1;
51     poly p1 = derivative(p), p2 = derivative(p1);
52     for (int step = 0; step < 10000; step++) {
53       cdouble y0 = eval(p, x);
54       if (comp(y0, 0) == 0) break;
55       cdouble G = eval(p1, x) / y0;
56       cdouble H = G * G - eval(p2, x) / y0;
57       cdouble R = std::sqrt(cdouble(n - 1) * (H * cdouble(n) - G * G));
58       cdouble D1 = G + R, D2 = G - R;
59       cdouble a = cdouble(n) / (comp(D1, D2) > 0 ? D1 : D2);
60       x -= a;
61       if (comp(a, 0) == 0) break;
62     }
63     return x;
64   }
65
66   std::vector<cdouble> find_all_roots(const poly & p) {
67     std::vector<cdouble> res;
```

```
68      poly q = p;
69      while (q.size() > 2) {
70        cdouble z(rand()/Double(RAND_MAX), rand()/Double(RAND_MAX));
71        z = find_one_root(q, z);
72        z = find_one_root(p, z);
73        q = horner(q, z).first;
74        res.push_back(z);
75      }
76      res.push_back(-q[0] / q[1]);
77      return res;
78    }
79
80    /*** Example Usage ***/
81
82    #include <cstdio>
83    #include <iostream>
84    using namespace std;
85
86    void print_roots(vector<cdouble> roots) {
87      for (int i = 0; i < (int)roots.size(); i++) {
88        printf("(%9.5f, ", (double)roots[i].real());
89        printf("%9.5f)\n", (double)roots[i].imag());
90      }
91    }
92
93    int main() {
94      { // x^3 - 8x^2 - 13x + 140 = (x + 4)(x - 5)(x - 7)
95        printf("Roots of x^3 - 8x^2 - 13x + 140:\n");
96        poly p;
97        p.push_back(140);
98        p.push_back(-13);
99        p.push_back(-8);
100       p.push_back(1);
101       vector<cdouble> roots = find_all_roots(p);
102       print_roots(roots);
103     }
104
105     { //(-6+4i)x^4 + (-26+12i)x^3 + (-30+40i)x^2 + (-26+12i)x + (-24+36i)
106       // = ((2+3i)x + 6)*(x + i)*(2x + (6+4i))*(x*i + 1)
107       printf("Roots of ((2+3i)x + 6)(x + i)(2x + (6+4i))(x*i + 1):\n");
108       poly p;
109       p.push_back(cdouble(-24, 36));
110       p.push_back(cdouble(-26, 12));
111       p.push_back(cdouble(-30, 40));
112       p.push_back(cdouble(-26, 12));
113       p.push_back(cdouble(-6, 4));
114       vector<cdouble> roots = find_all_roots(p);
115       print_roots(roots);
116     }
117     return 0;
118   }
```

4.6.3   Complex Root Finding (RPOLY)

```
1    /*
2
3    4.6.3 - Complex Root Finding (Jenkins-Traub Algorithm)
```

```
4
5    Determine the complex roots of a polynomial with real coefficients.
6    This is the variant of the Jenkins-Traub algorithm for polynomials
7    with real coefficient, known as RPOLY. RPOLY follows follows the
8    same pattern as the CPOLY algorithm, but computes two roots at a
9    time, either two real roots or a pair of conjugate complex roots.
10   See: https://en.wikipedia.org/wiki/Jenkins%E2%80%93Traub_algorithm
11
12   The following is a translation of TOMS493 (www.netlib.org/toms/)
13   from FORTRAN to C++, with a simple wrapper at the end for the C++
14   <complex> class. Although the code is not meant to be read, it is
15   extremely efficient and robust, capable of achieving an accuracy
16   of at least 5 decimal places for even the most strenuous inputs.
17
18   */
19
20   #include <cfloat> /* LDBL_EPSILON, LDBL_MAX, LDBL_MIN */
21   #include <cmath>  /* cosl, expl, fabsl, logl, powl, sinl, sqrtl */
22
23   typedef long double LD;
24
25   void divide_quadratic(int n, LD u, LD v, LD p[], LD q[], LD * a, LD * b) {
26     q[0] = *b = p[0];
27     q[1] = *a = -((*b) * u) + p[1];
28     for (int i = 2; i < n; i++) {
29       q[i] = -((*a) * u + (*b) * v) + p[i];
30       *b = *a;
31       *a = q[i];
32     }
33   }
34
35   int get_flag(int n, LD a, LD b, LD * a1, LD * a3, LD * a7,
36               LD * c, LD * d, LD * e, LD * f, LD * g, LD * h,
37               LD k[], LD u, LD v, LD qk[]) {
38     divide_quadratic(n, u, v, k, qk, c, d);
39     if (fabsl(*c) <= 100.0 * LDBL_EPSILON * fabsl(k[n - 1]) &&
40         fabsl(*d) <= 100.0 * LDBL_EPSILON * fabsl(k[n - 2])) return 3;
41     *h = v * b;
42     if (fabsl(*d) >= fabsl(*c)) {
43       *e = a / (*d);
44       *f = (*c) / (*d);
45       *g = u * b;
46       *a1 = (*f) * b - a;
47       *a3 = (*e) * ((*g) + a) + (*h) * (b / (*d));
48       *a7 = (*h) + ((*f) + u) * a;
49       return 2;
50     }
51     *e = a / (*c);
52     *f = (*d) / (*c);
53     *g = (*e) * u;
54     *a1 = -(a * ((*d) / (*c))) + b;
55     *a3 = (*e) * a + ((*g) + (*h) / (*c)) * b;
56     *a7 = (*g) * (*d) + (*h) * (*f) + a;
57     return 1;
58   }
59
60   void find_polynomials(int n, int flag, LD a, LD b, LD a1, LD * a3,
61                         LD * a7, LD k[], LD qk[], LD qp[]) {
62     if (flag == 3) {
```

```
63       k[1] = k[0] = 0.0;
64       for (int i = 2; i < n; i++) k[i] = qk[i - 2];
65       return;
66     }
67     if (fabsl(a1) > 10.0 * LDBL_EPSILON * fabsl(flag == 1 ? b : a)) {
68       *a7 /= a1;
69       *a3 /= a1;
70       k[0] = qp[0];
71       k[1] = qp[1] - (*a7) * qp[0];
72       for (int i = 2; i < n; i++)
73         k[i] = qp[i] - ((*a7) * qp[i - 1]) + (*a3) * qk[i - 2];
74     } else {
75       k[0] = 0.0;
76       k[1] = -(*a7) * qp[0];
77       for (int i = 2; i < n; i++)
78         k[i] = (*a3) * qk[i - 2] - (*a7) * qp[i - 1];
79     }
80   }
81
82   void estimate_coeff(int flag, LD * uu, LD * vv, LD a, LD a1, LD a3, LD a7,
83                       LD b, LD c, LD d, LD f, LD g, LD h, LD u, LD v, LD k[],
84                       int n, LD p[]) {
85     LD a4, a5, b1, b2, c1, c2, c3, c4, temp;
86     *vv = *uu = 0.0;
87     if (flag == 3) return;
88     if (flag != 2) {
89       a4 = a + u * b + h * f;
90       a5 = c + (u + v * f) * d;
91     } else {
92       a4 = (a + g) * f + h;
93       a5 = (f + u) * c + v * d;
94     }
95     b1 = -k[n - 1] / p[n];
96     b2 = -(k[n - 2] + b1 * p[n - 1]) / p[n];
97     c1 = v * b2 * a1;
98     c2 = b1 * a7;
99     c3 = b1 * b1 * a3;
100    c4 = c1 - c2 - c3;
101    temp = b1 * a4 - c4 + a5;
102    if (temp != 0.0) {
103      *uu= u - (u * (c3 + c2) + v * (b1 * a1 + b2 * a7)) / temp;
104      *vv = v * (1.0 + c4 / temp);
105    }
106  }
107
108  void solve_quadratic(LD a, LD b1, LD c, LD * sr, LD * si, LD * lr, LD * li) {
109    LD b, d, e;
110    *sr = *si = *lr = *li = 0.0;
111    if (a == 0) {
112      *sr = (b1 != 0) ? -c / b1 : *sr;
113      return;
114    }
115    if (c == 0) {
116      *lr = -b1 / a;
117      return;
118    }
119    b = b1 / 2.0;
120    if (fabsl(b) < fabsl(c)) {
121      e = (c >= 0) ? a : -a;
```

```
122        e = b * (b / fabsl(c)) - e;
123        d = sqrtl(fabsl(e)) * sqrtl(fabsl(c));
124      } else {
125        e = 1.0 - (a / b) * (c / b);
126        d = sqrtl(fabsl(e)) * fabsl(b);
127      }
128      if (e >= 0) {
129        d = (b >= 0) ? -d : d;
130        *lr = (d - b) / a;
131        *sr = (*lr != 0) ? (c / *lr / a) : *sr;
132      } else {
133        *lr = *sr = -b / a;
134        *si = fabsl(d / a);
135        *li = -(*si);
136      }
137    }
138
139    void quadratic_iterate(int N, int * NZ, LD uu, LD vv,
140                           LD * szr, LD * szi, LD * lzr, LD * lzi, LD qp[],
141                           int n, LD * a, LD * b, LD p[], LD qk[],
142                           LD * a1, LD * a3, LD * a7, LD * c, LD * d, LD * e,
143                           LD * f, LD * g, LD * h, LD k[]) {
144      int steps = 0, flag, tried_flag = 0;
145      LD ee, mp, omp = 0.0, relstp = 0.0, t, u, ui, v, vi, zm;
146      *NZ = 0;
147      u = uu;
148      v = vv;
149      do {
150        solve_quadratic(1.0, u, v, szr, szi, lzr, lzi);
151        if (fabsl(fabsl(*szr) - fabsl(*lzr)) > 0.01 * fabsl(*lzr)) break;
152        divide_quadratic(n, u, v, p, qp, a, b);
153        mp = fabsl(-((*szr) * (*b)) + *a) + fabsl((*szi) * (*b));
154        zm = sqrtl(fabsl(v));
155        ee = 2.0 * fabsl(qp[0]);
156        t = -(*szr) * (*b);
157        for (int i = 1; i < N; i++) ee = ee * zm + fabsl(qp[i]);
158        ee = ee * zm + fabsl(*a + t);
159        ee = ee * 9.0 + 2.0 * fabsl(t) - 7.0 * (fabsl(*a + t) + zm * fabsl(*b));
160        ee *= LDBL_EPSILON;
161        if (mp <= 20.0 * ee) {
162          *NZ = 2;
163          break;
164        }
165        if (++steps > 20) break;
166        if (steps >= 2 && relstp <= 0.01 && mp >= omp && !tried_flag) {
167          relstp = (relstp < LDBL_EPSILON) ? sqrtl(LDBL_EPSILON) : sqrtl(relstp);
168          u -= u * relstp;
169          v += v * relstp;
170          divide_quadratic(n, u, v, p, qp, a, b);
171          for (int i = 0; i < 5; i++) {
172            flag = get_flag(N, *a, *b, a1, a3, a7, c, d, e, f, g, h, k, u, v, qk);
173            find_polynomials(N, flag, *a, *b, *a1, a3, a7, k, qk, qp);
174          }
175          tried_flag = 1;
176          steps = 0;
177        }
178        omp = mp;
179        flag = get_flag(N, *a, *b, a1, a3, a7, c, d, e, f, g, h, k, u, v, qk);
180        find_polynomials(N, flag, *a, *b, *a1, a3, a7, k, qk, qp);
```

```
181      flag = get_flag(N, *a, *b, a1, a3, a7, c, d, e, f, g, h, k, u, v, qk);
182      estimate_coeff(flag, &ui, &vi, *a, *a1, *a3, *a7, *b, *c, *d, *f, *g, *h,
183                     u, v, k, N, p);
184      if (vi != 0) {
185        relstp = fabsl((-v + vi) / vi);
186        u = ui;
187        v = vi;
188      }
189    } while (vi != 0);
190  }
191
192  void real_iterate(int * flag, int * nz, LD * sss, int n, LD p[],
193                    int nn, LD qp[], LD * szr, LD * szi, LD k[], LD qk[]) {
194    int steps = 0;
195    LD ee, kv, mp, ms, omp = 0.0, pv, s, t = 0.0;
196    *flag = *nz = 0;
197    for (s = *sss; ; s += t) {
198      pv = p[0];
199      qp[0] = pv;
200      for (int i = 1; i < nn; i++) qp[i] = pv = pv * s + p[i];
201      mp = fabsl(pv);
202      ms = fabsl(s);
203      ee = 0.5 * fabsl(qp[0]);
204      for (int i = 1; i < nn; i++) ee = ee * ms + fabsl(qp[i]);
205      if (mp <= 20.0 * LDBL_EPSILON * (2.0 * ee - mp)) {
206        *nz = 1;
207        *szr = s;
208        *szi = 0.0;
209        break;
210      }
211      if (++steps > 10) break;
212      if (steps >= 2 && fabsl(t) <= 0.001 * fabsl(s - t) && mp > omp) {
213        *flag = 1;
214        *sss = s;
215        break;
216      }
217      omp = mp;
218      qk[0] = kv = k[0];
219      for (int i = 1; i < n; i++) qk[i] = kv = kv * s + k[i];
220      if (fabsl(kv) > fabsl(k[n - 1]) * 10.0 * LDBL_EPSILON) {
221        t = -pv / kv;
222        k[0] = qp[0];
223        for (int i = 1; i < n; i++)
224          k[i] = t * qk[i - 1] + qp[i];
225      } else {
226        k[0] = 0.0;
227        for (int i = 1; i < n; i++)
228          k[i] = qk[i - 1];
229      }
230      kv = k[0];
231      for (int i = 1; i < n; i++) kv = kv * s + k[i];
232      t = fabsl(kv) > (fabsl(k[n - 1]) * 10.0 * LDBL_EPSILON) ? -pv / kv : 0.0;
233    }
234  }
235
236  void solve_fixedshift(int l2, int * nz, LD sr, LD v, LD k[], int n,
237                        LD p[], int nn, LD qp[], LD u, LD qk[], LD svk[],
238                        LD * lzi, LD * lzr, LD * szi, LD * szr) {
239    int flag, _flag, __flag = 1, spass, stry, vpass, vtry;
```

```
240    LD a, a1, a3, a7, b, betas, betav, c, d, e, f, g, h;
241    LD oss, ots = 0.0, otv = 0.0, ovv, s, ss, ts, tss, tv, tvv, ui, vi, vv;
242    *nz = 0;
243    betav = betas = 0.25;
244    oss = sr;
245    ovv = v;
246    divide_quadratic(nn, u, v, p, qp, &a, &b);
247    flag = get_flag(n, a, b, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h,
248                    k, u, v, qk);
249    for (int j = 0; j < l2; j++) {
250      _flag = 1;
251      find_polynomials(n, flag, a, b, a1, &a3, &a7, k, qk, qp);
252      flag = get_flag(n, a, b, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h,
253                      k, u, v, qk);
254      estimate_coeff(flag, &ui, &vi, a, a1, a3, a7, b, c, d, f, g, h,
255                     u, v, k, n, p);
256      vv = vi;
257      ss = k[n - 1] != 0.0 ? -p[n] / k[n - 1] : 0.0;
258      ts = tv = 1.0;
259      if (j != 0 && flag != 3) {
260        tv = (vv != 0.0) ? fabsl((vv - ovv) / vv) : tv;
261        ts = (ss != 0.0) ? fabsl((ss - oss) / ss) : ts;
262        tvv = (tv < otv) ? tv * otv : 1.0;
263        tss = (ts < ots) ? ts * ots : 1.0;
264        vpass = (tvv < betav) ? 1 : 0;
265        spass = (tss < betas) ? 1 : 0;
266        if (spass || vpass) {
267          for (int i = 0; i < n; i++) svk[i] = k[i];
268          s = ss; stry = vtry = 0;
269          for (;;) {
270            if (!(_flag && spass && (!vpass || tss < tvv))) {
271              quadratic_iterate(n, nz, ui, vi, szr, szi, lzr, lzi, qp, nn,
272                      &a, &b, p, qk, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h, k);
273              if (*nz > 0) return;
274              __flag = vtry = 1;
275              betav *= 0.25;
276              if (stry || !spass) {
277                __flag = 0;
278              } else {
279                for (int i = 0; i < n; i++) k[i] = svk[i];
280              }
281            }
282            _flag = 0;
283            if (__flag != 0) {
284              real_iterate(&__flag, nz, &s, n, p, nn, qp, szr, szi, k, qk);
285              if (*nz > 0) return;
286              stry = 1;
287              betas *= 0.25;
288              if (__flag != 0) {
289                ui = -(s + s);
290                vi = s * s;
291                continue;
292              }
293            }
294            for (int i = 0; i < n; i++) k[i] = svk[i];
295            if (!vpass || vtry) break;
296          }
297          divide_quadratic(nn, u, v, p, qp, &a, &b);
298          flag = get_flag(n, a, b, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h,
```

```
299                              k, u, v, qk);
300            }
301          }
302        ovv = vv;
303        oss = ss;
304        otv = tv;
305        ots = ts;
306      }
307  }
308
309  void find_roots(int degree, LD co[], LD re[], LD im[]) {
310    int j, jj, n, nm1, nn, nz, zero, SZ = degree + 1;
311    LD k[SZ], p[SZ], pt[SZ], qp[SZ], temp[SZ], qk[SZ], svk[SZ];
312    LD bnd, df, dx, factor, ff, moduli_max, moduli_min, sc, x, xm;
313    LD aa, bb, cc, lzi, lzr, sr, szi, szr, t, u, xx, xxx, yy;
314    n = degree;
315    xx = sqrtl(0.5);
316    yy = -xx;
317    for (j = 0; co[n] == 0; n--, j++) re[j] = im[j] = 0.0;
318    nn = n + 1;
319    for (int i = 0; i < nn; i++) p[i] = co[i];
320    while (n >= 1) {
321      if (n <= 2) {
322        if (n < 2) {
323          re[degree - 1] = -p[1] / p[0];
324          im[degree - 1] = 0.0;
325        } else {
326          solve_quadratic(p[0], p[1], p[2], &re[degree - 2], &im[degree - 2],
327                                           &re[degree - 1], &im[degree - 1]);
328        }
329        break;
330      }
331      moduli_max = 0.0;
332      moduli_min = LDBL_MAX;
333      for (int i = 0; i < nn; i++) {
334        x = fabsl(p[i]);
335        if (x > moduli_max) moduli_max = x;
336        if (x != 0 && x < moduli_min) moduli_min = x;
337      }
338      sc = LDBL_MIN / LDBL_EPSILON / moduli_min;
339      if ((sc <= 1.0 && moduli_max >= 10) ||
340          (sc > 1.0 && LDBL_MAX / sc >= moduli_max)) {
341        sc = (sc == 0) ? LDBL_MIN : sc;
342        factor = powl(2.0, logl(sc) / logl(2.0));
343        if (factor != 1.0)
344          for (int i = 0; i < nn; i++) p[i] *= factor;
345      }
346      for (int i = 0; i < nn; i++) pt[i] = fabsl(p[i]);
347      pt[n] = -pt[n];
348      nm1 = n - 1;
349      x = expl((logl(-pt[n]) - logl(pt[0])) / (LD)n);
350      if (pt[nm1] != 0) {
351        xm = -pt[n] / pt[nm1];
352        if (xm < x) x = xm;
353      }
354      xm = x;
355      do {
356        x = xm;
357        xm = 0.1 * x;
```

```
358          ff = pt[0];
359          for (int i = 1; i < nn; i++) ff = ff * xm + pt[i];
360        } while (ff > 0);
361        dx = x;
362        do {
363          df = ff = pt[0];
364          for (int i = 1; i < n; i++) {
365            ff = x * ff + pt[i];
366            df = x * df + ff;
367          }
368          ff = x * ff + pt[n];
369          dx = ff / df;
370          x -= dx;
371        } while (fabsl(dx / x) > 0.005);
372        bnd = x;
373        for (int i = 1; i < n; i++)
374          k[i] = (LD)(n - i) * p[i] / (LD)n;
375        k[0] = p[0];
376        aa = p[n];
377        bb = p[nm1];
378        zero = (k[nm1] == 0) ? 1 : 0;
379        for (jj = 0; jj < 5; jj++) {
380          cc = k[nm1];
381          if (zero) {
382            for (int i = 0; i < nm1; i++) {
383              j = nm1 - i;
384              k[j] = k[j - 1];
385            }
386            k[0] = 0;
387            zero = (k[nm1] == 0) ? 1 : 0;
388          } else {
389            t = -aa / cc;
390            for (int i = 0; i < nm1; i++) {
391              j = nm1 - i;
392              k[j] = t * k[j - 1] + p[j];
393            }
394            k[0] = p[0];
395            zero = (fabsl(k[nm1]) <= fabsl(bb) * LDBL_EPSILON * 10.0) ? 1 : 0;
396          }
397        }
398        for (int i = 0; i < n; i++) temp[i] = k[i];
399        static const LD DEG = 0.01745329251994329576923690768489L;
400        for (jj = 1; jj <= 20; jj++) {
401          xxx = -sinl(94.0 * DEG) * yy + cosl(94.0 * DEG) * xx;
402          yy = sinl(94.0 * DEG) * xx + cosl(94.0 * DEG) * yy;
403          xx = xxx;
404          sr = bnd * xx;
405          u = -2.0 * sr;
406          for (int i = 0; i < nn; i++) qk[i] = svk[i] = 0.0;
407          solve_fixedshift(20 * jj, &nz, sr, bnd, k, n, p, nn, qp, u,
408                           qk, svk, &lzi, &lzr, &szi, &szr);
409          if (nz != 0) {
410            j = degree - n;
411            re[j] = szr;
412            im[j] = szi;
413            nn = nn - nz;
414            n = nn - 1;
415            for (int i = 0; i < nn; i++) p[i] = qp[i];
416            if (nz != 1) {
```

```
417             re[j + 1] = lzr;
418             im[j + 1] = lzi;
419           }
420           break;
421         } else {
422           for (int i = 0; i < n; i++) k[i] = temp[i];
423         }
424       }
425       if (jj > 20) break;
426     }
427 }
428
429 /*** Wrapper ***/
430
431 #include <algorithm> /* std::reverse(), std::sort() */
432 #include <complex>
433 #include <vector>
434
435 typedef std::complex<LD> root;
436
437 bool comp(const root & a, const root & b) {
438   if (real(a) != real(b)) return real(a) < real(b);
439   return imag(a) < imag(b);
440 }
441
442 std::vector<root> find_roots(int degree, LD coefficients[]) {
443   std::reverse(coefficients, coefficients + degree + 1);
444   LD re[degree], im[degree];
445   find_roots(degree, coefficients, re, im);
446   std::vector<root> res;
447   for (int i = 0; i < degree; i++)
448     res.push_back(root(re[i], im[i]));
449   std::sort(res.begin(), res.end(), comp);
450   return res;
451 }
452
453 /*** Example Usage (http://wcipeg.com/problem/rootsolve) ***/
454
455 #include <iostream>
456 using namespace std;
457
458 int T, degree, p;
459 LD c, coeff[101];
460
461 int main() {
462   degree = 0;
463   cin >> T;
464   for (int i = 0; i < T; i++) {
465     cin >> c >> p;
466     if (p > degree) degree = p;
467     coeff[p] = c;
468   }
469   std::vector<root> roots = find_roots(degree, coeff);
470   bool printed = false;
471   cout.precision(6);
472   for (int i = 0; i < (int)roots.size(); i++) {
473     if (fabsl(roots[i].imag()) < LDBL_EPSILON) {
474       cout << fixed << roots[i].real() << "\n";
475       printed = true;
```

```
476        }
477      }
478      if (!printed) cout << "NO_REAL_ROOTS\n";
479      return 0;
480    }
```

4.7   Integration

4.7.1   Simpson's Rule

```
1    /*
2
3    4.7.1 - Integration (Simpson's Rule)
4
5    Simpson's rule is a method for numerical integration, the
6    numerical approximation of definite integrals. The rule is:
7
8    Integral of f(x) dx from a to b ~=
9      [f(a) + 4*f((a + b)/2) + f(b)] * (b - a)/6
10
11   */
12
13   #include <cmath> /* fabs() */
14
15   template<class DoubleFunction>
16   double simpsons(DoubleFunction f, double a, double b) {
17     return (f(a) + 4 * f((a + b)/2) + f(b)) * (b - a)/6;
18   }
19
20   template<class DoubleFunction>
21   double integrate(DoubleFunction f, double a, double b) {
22     static const double eps = 1e-10;
23     double m = (a + b) / 2;
24     double am = simpsons(f, a, m);
25     double mb = simpsons(f, m, b);
26     double ab = simpsons(f, a, b);
27     if (fabs(am + mb - ab) < eps) return ab;
28     return integrate(f, a, m) + integrate(f, m, b);
29   }
30
31   /*** Example Usage ***/
32
33   #include <iostream>
34   using namespace std;
35
36   double f(double x) { return sin(x); }
37
38   int main () {
39     double PI = acos(-1.0);
40     cout << integrate(f, 0.0, PI/2) << "\n"; //1
41     return 0;
42   }
```

Chapter 5

Geometry

5.1 Geometric Classes

5.1.1 Point

```
1   /*
2
3   5.1.1 - 2D Point Class
4
5   This class is very similar to std::complex, except it uses epsilon
6   comparisons and also supports other operations such as reflection
7   and rotation. In addition, this class supports many arithmetic
8   operations (e.g. overloaded operators for vector addition, subtraction,
9   multiplication, and division; dot/cross products, etc.) pertaining to
10  2D cartesian vectors.
11
12  All operations are O(1) in time and space.
13
14  */
15
16  #include <cmath>   /* atan(), fabs(), sqrt() */
17  #include <ostream>
18  #include <utility> /* std::pair */
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
25  struct point {
26
27    double x, y;
28
29    point() : x(0), y(0) {}
30    point(const point & p) : x(p.x), y(p.y) {}
31    point(const std::pair<double, double> & p) : x(p.first), y(p.second) {}
32    point(const double & a, const double & b) : x(a), y(b) {}
33
34    bool operator < (const point & p) const {
35      return EQ(x, p.x) ? LT(y, p.y) : LT(x, p.x);
```

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

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

```
153      return 0;
154    }
```

5.1.2 Line

```
1    /*
2
3    5.1.2 - 2D Line Class
4
5    A 2D line is expressed in the form Ax + By + C = 0. All lines can be
6    "normalized" to a canonical form by insisting that the y-coefficient
7    equal 1 if it is non-zero. Otherwise, we set the x-coefficient to 1.
8    If B is non-zero, then we have the common case where the slope = -A
9    after normalization.
10
11    All operations are O(1) in time and space.
12
13    */
14
15    #include <cmath>    /* fabs() */
16    #include <limits>   /* std::numeric_limits */
17    #include <ostream>
18    #include <utility>  /* std::pair */
19
20    const double eps = 1e-9, NaN = std::numeric_limits<double>::quiet_NaN();
21
22    #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
23    #define LT(a, b) ((a) < (b) - eps)        /* less than */
24
25    typedef std::pair<double, double> point;
26    #define x first
27    #define y second
28
29    struct line {
30
31      double a, b, c;
32
33      line(): a(0), b(0), c(0) {} //invalid or uninitialized line
34
35      line(const double & A, const double & B, const double & C) {
36        a = A;
37        b = B;
38        c = C;
39        if (!EQ(b, 0)) {
40          a /= b; c /= b; b = 1;
41        } else {
42          c /= a; a = 1; b = 0;
43        }
44      }
45
46      line(const double & slope, const point & p) {
47        a = -slope;
48        b = 1;
49        c = slope * p.x - p.y;
50      }
51
52      line(const point & p, const point & q): a(0), b(0), c(0) {
```

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

```
112    //returns whether the line is perpendicular to l
113    bool perpendicular(const line & l) const {
114      return EQ(-a * l.a, b * l.b);
115    }
116
117    //return the parallel line passing through point p
118    line parallel(const point & p) const {
119      return line(a, b, -a * p.x - b * p.y);
120    }
121
122    //return the perpendicular line passing through point p
123    line perpendicular(const point & p) const {
124      return line(-b, a, b * p.x - a * p.y);
125    }
126
127    friend std::ostream & operator << (std::ostream & out, const line & l) {
128      out << (fabs(l.a) < eps ? 0 : l.a) << "x" << std::showpos;
129      out << (fabs(l.b) < eps ? 0 : l.b) << "y";
130      out << (fabs(l.c) < eps ? 0 : l.c) << "=0" << std::noshowpos;
131      return out;
132    }
133  };
134
135  /*** Example Usage ***/
136
137  #include <cassert>
138
139  int main() {
140    line l(2, -5, -8);
141    line para = line(2, -5, -8).parallel(point(-6, -2));
142    line perp = line(2, -5, -8).perpendicular(point(-6, -2));
143    assert(l.parallel(para) && l.perpendicular(perp));
144    assert(l.slope() == 0.4);
145    assert(para == line(-0.4, 1, -0.4)); //-0.4x+1y-0.4=0
146    assert(perp == line(2.5, 1, 17));    //2.5x+1y+17=0
147    return 0;
148  }
```

5.1.3   Circle

```
1   /*
2
3   5.1.3 - 2D Circle Class
4
5   A 2D circle with center at (h, k) and a radius of r can be expressed by
6   the relation (x - h)^2 + (y - k)^2 = r^2. In the following definition,
7   the radius used to construct it is forced to be a positive number.
8
9   All operations are O(1) in time and space.
10
11  */
12
13  #include <cmath>     /* fabs(), sqrt() */
14  #include <ostream>
15  #include <stdexcept> /* std::runtime_error() */
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 GT(a, b) ((a) > (b) + eps)        /* greater 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   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 & R): h(0), k(0), r(fabs(R)) {}
37     circle(const point & o, const double & R): h(o.x), k(o.y), r(fabs(R)) {}
38     circle(const double & H, const double & K, const double & R):
39       h(H), k(K), r(fabs(R)) {}
40
41     //circumcircle with the diameter equal to the distance from a to b
42     circle(const point & a, const point & b) {
43       h = (a.x + b.x) / 2.0;
44       k = (a.y + b.y) / 2.0;
45       r = abs(point(a.x - h, a.y - k));
46     }
47
48     //circumcircle of 3 points - throws exception if abc are collinear/equal
49     circle(const point & a, const point & b, const point & c) {
50       double an = norm(point(b.x - c.x, b.y - c.y));
51       double bn = norm(point(a.x - c.x, a.y - c.y));
52       double cn = norm(point(a.x - b.x, a.y - b.y));
53       double wa = an * (bn + cn - an);
54       double wb = bn * (an + cn - bn);
55       double wc = cn * (an + bn - cn);
56       double w = wa + wb + wc;
57       if (fabs(w) < eps)
58         throw std::runtime_error("No circle from collinear points.");
59       h = (wa * a.x + wb * b.x + wc * c.x) / w;
60       k = (wa * a.y + wb * b.y + wc * c.y) / w;
61       r = abs(point(a.x - h, a.y - k));
62     }
63
64     //circle from 2 points and a radius - many possible edge cases!
65     //in the "normal" case, there will be 2 possible circles, one
66     //centered at (h1, k1) and the other (h2, k2). Only one is used.
67     //note that (h1, k1) equals (h2, k2) if dist(a, b) = 2 * r = d
68     circle(const point & a, const point & b, const double & R) {
69       r = fabs(R);
70       if (LE(r, 0) && a == b) { //circle is a point
71         h = a.x;
72         k = a.y;
73         return;
74       }
75       double d = abs(point(b.x - a.x, b.y - a.y));
76       if (EQ(d, 0))
```

```
77            throw std::runtime_error("Identical␣points,␣infinite␣circles.");
78         if (GT(d, r * 2.0))
79            throw std::runtime_error("Points␣too␣far␣away␣to␣make␣circle.");
80         double v = sqrt(r * r - d * d / 4.0) / d;
81         point m((a.x + b.x) / 2.0, (a.y + b.y) / 2.0);
82         h = m.x + (a.y - b.y) * v;
83         k = m.y + (b.x - a.x) * v;
84         //other answer is (h, k) = (m.x-(a.y-b.y)*v, m.y-(b.x-a.x)*v)
85      }
86
87      bool operator == (const circle & c) const {
88         return EQ(h, c.h) && EQ(k, c.k) && EQ(r, c.r);
89      }
90
91      bool operator != (const circle & c) const {
92         return !(*this == c);
93      }
94
95      bool contains(const point & p) const {
96         return LE(norm(point(p.x - h, p.y - k)), r * r);
97      }
98
99      bool on_edge(const point & p) const {
100        return EQ(norm(point(p.x - h, p.y - k)), r * r);
101     }
102
103     point center() const {
104        return point(h, k);
105     }
106
107     friend std::ostream & operator << (std::ostream & out, const circle & c) {
108        out << std::showpos;
109        out << "(x" << -(fabs(c.h) < eps ? 0 : c.h) << ")^2+";
110        out << "(y" << -(fabs(c.k) < eps ? 0 : c.k) << ")^2";
111        out << std::noshowpos;
112        out << "=" << (fabs(c.r) < eps ? 0 : c.r * c.r);
113        return out;
114     }
115  };
116
117  //circle inscribed within points a, b, and c
118  circle incircle(const point & a, const point & b, const point & c) {
119     double al = abs(point(b.x - c.x, b.y - c.y));
120     double bl = abs(point(a.x - c.x, a.y - c.y));
121     double cl = abs(point(a.x - b.x, a.y - b.y));
122     double p = al + bl + cl;
123     if (EQ(p, 0)) return circle(a.x, a.y, 0);
124     circle res;
125     res.h = (al * a.x + bl * b.x + cl * c.x) / p;
126     res.k = (al * a.y + bl * b.y + cl * c.y) / p;
127     res.r = fabs((a.x - c.x) * (b.y - c.y) - (a.y - c.y) * (b.x - c.x)) / p;
128     return res;
129  }
130
131  /*** Example Usage ***/
132
133  #include <cassert>
134
135  int main() {
```

```
136    circle c(-2, 5, sqrt(10)); //(x+2)^2+(y-5)^2=10
137    assert(c == circle(point(-2, 5), sqrt(10)));
138    assert(c == circle(point(1, 6), point(-5, 4)));
139    assert(c == circle(point(-3, 2), point(-3, 8), point(-1, 8)));
140    assert(c == incircle(point(-12, 5), point(3, 0), point(0, 9)));
141    assert(c.contains(point(-2, 8)) && !c.contains(point(-2, 9)));
142    assert(c.on_edge(point(-1, 2)) && !c.on_edge(point(-1.01, 2)));
143    return 0;
144  }
```

5.2   Geometric Calculations

5.2.1   Angles

```
1   /*
2
3   5.2.1 - Angles (2D)
4
5   Angle calculations in 2 dimensions. All returned angles are in radians,
6   except for reduce_deg(). If x is an angle in radians, then you may use
7   x * DEG to convert x to degrees, and vice versa to radians with x * RAD.
8
9   All operations are O(1) in time and space.
10
11  */
12
13  #include <cmath>      /* acos(), fabs(), sqrt(), atan2() */
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 PI = acos(-1.0), RAD = 180 / PI, DEG = PI / 180;
21
22  double abs(const point & a) { return sqrt(a.x * a.x + a.y * a.y); }
23
24  //reduce angles to the range [0, 360) degrees. e.g. reduce_deg(-630) = 90
25  double reduce_deg(const double & t) {
26    if (t < -360) return reduce_deg(fmod(t, 360));
27    if (t < 0) return t + 360;
28    return t >= 360 ? fmod(t, 360) : t;
29  }
30
31  //reduce angles to the range [0, 2*pi) radians. e.g. reduce_rad(720.5) = 0.5
32  double reduce_rad(const double & t) {
33    if (t < -2 * PI) return reduce_rad(fmod(t, 2 * PI));
34    if (t < 0) return t + 2 * PI;
35    return t >= 2 * PI ? fmod(t, 2 * PI) : t;
36  }
37
38  //like std::polar(), but returns a point instead of an std::complex
39  point polar_point(const double & r, const double & theta) {
40    return point(r * cos(theta), r * sin(theta));
41  }
42
```

```
43  //angle of segment (0, 0) to p, relative (CCW) to the +'ve x-axis in radians
44  double polar_angle(const point & p) {
45    double t = atan2(p.y, p.x);
46    return t < 0 ? t + 2 * PI : t;
47  }
48
49  //smallest angle formed by points aob (angle is at point o) in radians
50  double angle(const point & a, const point & o, const point & b) {
51    point u(o.x - a.x, o.y - a.y), v(o.x - b.x, o.y - b.y);
52    return acos((u.x * v.x + u.y * v.y) / (abs(u) * abs(v)));
53  }
54
55  //angle of line segment ab relative (CCW) to the +'ve x-axis in radians
56  double angle_between(const point & a, const point & b) {
57    double t = atan2(a.x * b.y - a.y * b.x, a.x * b.x + a.y * b.y);
58    return t < 0 ? t + 2 * PI : t;
59  }
60
61  //Given the A, B values of two lines in Ax + By + C = 0 form, finds the
62  //minimum angle in radians between the two lines in the range [0, PI/2]
63  double angle_between(const double & a1, const double & b1,
64                       const double & a2, const double & b2) {
65    double t = atan2(a1 * b2 - a2 * b1, a1 * a2 + b1 * b2);
66    if (t < 0) t += PI; //force angle to be positive
67    if (t > PI / 2) t = PI - t; //force angle to be <= 90 degrees
68    return t;
69  }
70
71  //magnitude of the 3D cross product with Z component implicitly equal to 0
72  //the answer assumes the origin (0, 0) is instead shifted to point o.
73  //this is equal to 2x the signed area of the triangle from these 3 points.
74  double cross(const point & o, const point & a, const point & b) {
75    return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
76  }
77
78  //does the path a->o->b form:
79  // -1 ==> a left turn on the plane?
80  //  0 ==> a single straight line segment? (i.e. are a,o,b collinear?) or
81  // +1 ==> a right turn on the plane?
82  //warning: the order of parameters is a,o,b, and NOT o,a,b as in cross()
83  int turn(const point & a, const point & o, const point & b) {
84    double c = cross(o, a, b);
85    return c < 0 ? -1 : (c > 0 ? 1 : 0);
86  }
87
88  /*** Example Usage ***/
89
90  #include <cassert>
91  #define pt point
92  #define EQ(a, b) (fabs((a) - (b)) <= 1e-9)
93
94  int main() {
95    assert(EQ(123,    reduce_deg(-(8 * 360) + 123)));
96    assert(EQ(1.2345, reduce_rad(2 * PI * 8 + 1.2345)));
97    point p = polar_point(4, PI), q = polar_point(4, -PI / 2);
98    assert(EQ(p.x, -4) && EQ(p.y, 0));
99    assert(EQ(q.x, 0) && EQ(q.y, -4));
100   assert(EQ(45,  polar_angle(pt(5, 5)) * RAD));
101   assert(EQ(135, polar_angle(pt(-4, 4)) * RAD));
```

```
102    assert(EQ(90,  angle(pt(5, 0), pt(0, 5), pt(-5, 0)) * RAD));
103    assert(EQ(225, angle_between(pt(0, 5), pt(5, -5)) * RAD));
104    assert(EQ(90,  angle_between(-1, 1, -1, -1) * RAD)); //y=x and y=-x
105    assert(-1 == cross(pt(0, 0), pt(0, 1), pt(1, 0)));
106    assert(+1 == turn(pt(0, 1), pt(0, 0), pt(-5, -5)));
107    return 0;
108  }
```

5.2.2   Distances

```
1    /*
2
3    5.2.2 - Distances (2D)
4
5    Distance calculations in 2 dimensions between points, lines, and segments.
6    All operations are O(1) in time and space.
7
8    */
9
10   #include <algorithm> /* std::max(), std::min() */
11   #include <cmath>      /* 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 LE(a, b) ((a) <= (b) + eps)        /* less than or equal to */
22   #define GE(a, b) ((a) >= (b) - eps)        /* greater than or equal to */
23
24   double norm(const point & a) { return a.x * a.x + a.y * a.y; }
25   double abs(const point & a) { return sqrt(norm(a)); }
26
27   //distance from point a to point b
28   double dist(const point & a, const point & b) {
29     return abs(point(b.x - a.x, b.y - a.y));
30   }
31
32   //squared distance from point a to point b
33   double dist2(const point & a, const point & b) {
34     return norm(point(b.x - a.x, b.y - a.y));
35   }
36
37   //minimum distance from point p to line l denoted by ax + by + c = 0
38   //if a = b = 0, then -inf, nan, or +inf is returned depending on sgn(c)
39   double dist_line(const point & p,
40                    const double & a, const double & b, const double & c) {
41     return fabs(a * p.x + b * p.y + c) / sqrt(a * a + b * b);
42   }
43
44   //minimum distance from point p to the infinite line containing a and b
45   //if a = b, then the point distance from p to the single point is returned
46   double dist_line(const point & p, const point & a, const point & b) {
47     double ab2 = dist2(a, b);
```

```
48       if (EQ(ab2, 0)) return dist(p, a);
49       double u = ((p.x - a.x) * (b.x - a.x) + (p.y - a.y) * (b.y - a.y)) / ab2;
50       return abs(point(a.x + u * (b.x - a.x) - p.x, a.y + u * (b.y - a.y) - p.y));
51     }
52
53     //distance between two lines each denoted by the form ax + by + c = 0
54     //if the lines are nonparallel, then the distance is 0, otherwise
55     //it is the perpendicular distance from a point on one line to the other
56     double dist_lines(const double & a1, const double & b1, const double & c1,
57                       const double & a2, const double & b2, const double & c2) {
58       if (EQ(a1 * b2, a2 * b1)) {
59         double factor = EQ(b1, 0) ? (a1 / a2) : (b1 / b2);
60         if (EQ(c1, c2 * factor)) return 0;
61         return fabs(c2 * factor - c1) / sqrt(a1 * a1 + b1 * b1);
62       }
63       return 0;
64     }
65
66     //distance between two infinite lines respectively containing ab and cd
67     //same results as above, except we solve for the lines here first.
68     double dist_lines(const point & a, const point & b,
69                       const point & c, const point & d) {
70       double A1 = a.y - b.y, B1 = b.x - a.x;
71       double A2 = c.y - d.y, B2 = d.x - c.x;
72       double C1 = -A1 * a.x - B1 * a.y, C2 = -A2 * c.x - B2 * c.y;
73       return dist_lines(A1, B1, C1, A2, B2, C2);
74     }
75
76     //minimum distance from point p to any point on segment ab
77     double dist_seg(const point & p, const point & a, const point & b) {
78       if (a == b) return dist(p, a);
79       point ab(b.x - a.x, b.y - a.y), ap(p.x - a.x, p.y - a.y);
80       double n = norm(ab), d = ab.x * ap.x + ab.y * ap.y;
81       if (LE(d, 0) || EQ(n, 0)) return abs(ap);
82       if (GE(d, n)) return abs(point(ap.x - ab.x, ap.y - ab.y));
83       return abs(point(ap.x - ab.x * (d / n), ap.y - ab.y * (d / n)));
84     }
85
86     double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
87     double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }
88
89     //minimum distance from any point on segment ab to any point on segment cd
90     double dist_segs(const point & a, const point & b,
91                      const point & c, const point & d) {
92       //check if segments are touching or intersecting - if so, distance is 0
93       point ab(b.x - a.x, b.y - a.y);
94       point ac(c.x - a.x, c.y - a.y);
95       point cd(d.x - c.x, d.y - c.y);
96       double c1 = cross(ab, cd), c2 = cross(ac, ab);
97       if (EQ(c1, 0) && EQ(c2, 0)) {
98         double t0 = dot(ac, ab) / norm(ab);
99         double t1 = t0 + dot(cd, ab) / norm(ab);
100        if (LE(std::min(t0, t1), 1) && LE(0, std::max(t0, t1)))
101          return 0;
102      } else {
103        double t = cross(ac, cd) / c1, u = c2 / c1;
104        if (!EQ(c1, 0) && LE(0, t) && LE(t, 1) && LE(0, u) && LE(u, 1))
105          return 0;
106      }
```

```
107    //find min distances across each endpoint to opposing segment
108    return std::min(std::min(dist_seg(a, c, d), dist_seg(b, c, d)),
109                    std::min(dist_seg(c, a, b), dist_seg(d, a, b)));
110  }
111
112  /*** Example Usage ***/
113
114  #include <cassert>
115  #define pt point
116
117  int main() {
118    assert(EQ(5, dist(pt(-1, -1), pt(2, 3))));
119    assert(EQ(25, dist2(pt(-1, -1), pt(2, 3))));
120    assert(EQ(1.2, dist_line(pt(2, 1), -4, 3, -1)));
121    assert(EQ(0.8, dist_line(pt(3, 3), pt(-1, -1), pt(2, 3))));
122    assert(EQ(1.2, dist_line(pt(2, 1), pt(-1, -1), pt(2, 3))));
123    assert(EQ(0.0, dist_lines(-4, 3, -1, 8, 6, 2)));
124    assert(EQ(0.8, dist_lines(-4, 3, -1, -8, 6, -10)));
125    assert(EQ(1.0, dist_seg(pt(3, 3), pt(-1, -1), pt(2, 3))));
126    assert(EQ(1.2, dist_seg(pt(2, 1), pt(-1, -1), pt(2, 3))));
127    assert(EQ(0.0, dist_segs(pt(0, 2), pt(3, 3), pt(-1, -1), pt(2, 3))));
128    assert(EQ(0.6, dist_segs(pt(-1, 0), pt(-2, 2), pt(-1, -1), pt(2, 3))));
129    return 0;
130  }
```

5.2.3   Line Intersections

```
1    /*
2
3    5.2.3 - Line Intersections (2D)
4
5    Intersections between straight lines, as well as between line segments
6    in 2 dimensions. Also included are functions to determine the closest
7    point to a line, which is done by finding the intersection through the
8    perpendicular. Note that you should modify the TOUCH_IS_INTERSECT flag
9    used for line segment intersection, depending on whether you wish for
10   the algorithm to consider barely touching segments to intersect.
11
12   All operations are O(1) in time and space.
13
14   */
15
16   #include <algorithm> /* std::min(), std::max() */
17   #include <cmath>      /* fabs(), sqrt() */
18   #include <utility>    /* std::pair */
19
20   typedef std::pair<double, double> point;
21   #define x first
22   #define y second
23
24   const double eps = 1e-9;
25
26   #define EQ(a, b) (fabs((a) - (b)) <= eps)  /* equal to */
27   #define LT(a, b) ((a) < (b) - eps)         /* less than */
28   #define LE(a, b) ((a) <= (b) + eps)        /* less than or equal to */
29
30   //intersection of line l1 and line l2, each in ax + by + c = 0 form
```

```
31  //returns: -1, if lines do not intersect,
32  //          0, if there is exactly one intersection point, or
33  //         +1, if there are infinite intersection
34  //in the 2nd case, the intersection point is optionally stored into p
35  int line_intersection(const double & a1, const double & b1, const double & c1,
36                        const double & a2, const double & b2, const double & c2,
37                        point * p = 0) {
38    if (EQ(a1 * b2, a2 * b1))
39      return (EQ(a1 * c2, a2 * c1) || EQ(b1 * c2, b2 * c1)) ? 1 : -1;
40    if (p != 0) {
41      p->x = (b1 * c1 - b1 * c2) / (a2 * b1 - a1 * b2);
42      if (!EQ(b1, 0)) p->y = -(a1 * p->x + c1) / b1;
43      else p->y = -(a2 * p->x + c2) / b2;
44    }
45    return 0;
46  }
47
48  //intersection of line through p1, p2, and line through p2, p3
49  //returns: -1, if lines do not intersect,
50  //          0, if there is exactly one intersection point, or
51  //         +1, if there are infinite intersections
52  //in the 2nd case, the intersection point is optionally stored into p
53  int line_intersection(const point & p1, const point & p2,
54                        const point & p3, const point & p4, point * p = 0) {
55    double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
56    double c1 = -(p1.x * p2.y - p2.x * p1.y);
57    double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
58    double c2 = -(p3.x * p4.y - p4.x * p3.y);
59    double x = -(c1 * b2 - c2 * b1), y = -(a1 * c2 - a2 * c1);
60    double det = a1 * b2 - a2 * b1;
61    if (EQ(det, 0))
62      return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
63    if (p != 0) *p = point(x / det, y / det);
64    return 0;
65  }
66
67  //Line Segment Intersection (http://stackoverflow.com/a/565282)
68
69  double norm(const point & a) { return a.x * a.x + a.y * a.y; }
70  double abs(const point & a) { return sqrt(norm(a)); }
71  double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
72  double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }
73
74  //should we consider barely touching segments an intersection?
75  const bool TOUCH_IS_INTERSECT = true;
76
77  //does [l, h] contain m?
78  //precondition: l <= h
79  bool contain(const double & l, const double & m, const double & h) {
80    if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
81    return LT(l, m) && LT(m, h);
82  }
83
84  //does [l1, h1] overlap with [l2, h2]?
85  //precondition: l1 <= h1 and l2 <= h2
86  bool overlap(const double & l1, const double & h1,
87               const double & l2, const double & h2) {
88    if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
89    return LT(l1, h2) && LT(l2, h1);
```

```
 90  }
 91
 92  //intersection of line segment ab with line segment cd
 93  //returns: -1, if segments do not intersect,
 94  //           0, if there is exactly one intersection point
 95  //          +1, if the intersection is another line segment
 96  //In case 2, the intersection point is stored into p
 97  //In case 3, the intersection segment is stored into p and q
 98  int seg_intersection(const point & a, const point & b,
 99                       const point & c, const point & d,
100                       point * p = 0, point * q = 0) {
101    point ab(b.x - a.x, b.y - a.y);
102    point ac(c.x - a.x, c.y - a.y);
103    point cd(d.x - c.x, d.y - c.y);
104    double c1 = cross(ab, cd), c2 = cross(ac, ab);
105    if (EQ(c1, 0) && EQ(c2, 0)) { //collinear
106      double t0 = dot(ac, ab) / norm(ab);
107      double t1 = t0 + dot(cd, ab) / norm(ab);
108      if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
109        point res1 = std::max(std::min(a, b), std::min(c, d));
110        point res2 = std::min(std::max(a, b), std::max(c, d));
111        if (res1 == res2) {
112          if (p != 0) *p = res1;
113          return 0; //collinear, meeting at an endpoint
114        }
115        if (p != 0 && q != 0) *p = res1, *q = res2;
116        return 1; //collinear and overlapping
117      } else {
118        return -1; //collinear and disjoint
119      }
120    }
121    if (EQ(c1, 0)) return -1; //parallel and disjoint
122    double t = cross(ac, cd) / c1, u = c2 / c1;
123    if (contain(0, t, 1) && contain(0, u, 1)) {
124      if (p != 0) *p = point(a.x + t * ab.x, a.y + t * ab.y);
125      return 0; //non-parallel with one intersection
126    }
127    return -1; //non-parallel with no intersections
128  }
129
130  //determines the point on line ax + by + c = 0 that is closest to point p
131  //this always lies on the line through p perpendicular to l.
132  point closest_point(const double & a, const double & b, const double & c,
133                      const point & p) {
134    if (EQ(a, 0)) return point(p.x, -c); //horizontal line
135    if (EQ(b, 0)) return point(-c, p.y); //vertical line
136    point res;
137    line_intersection(a, b, c, -b, a, b * p.x - a * p.y, &res);
138    return res;
139  }
140
141  //determines the point on segment ab closest to point p
142  point closest_point(const point & a, const point & b, const point & p) {
143    if (a == b) return a;
144    point ap(p.x - a.x, p.y - a.y), ab(b.x - a.x, b.y - a.y);
145    double t = dot(ap, ab) / norm(ab);
146    if (t <= 0) return a;
147    if (t >= 1) return b;
148    return point(a.x + t * ab.x, a.y + t * ab.y);
```

```
149  }
150
151  /*** Example Usage ***/
152
153  #include <cassert>
154  #define pt point
155
156  int main() {
157    point p;
158    assert(line_intersection(-1, 1, 0, 1, 1, -3, &p) == 0);
159    assert(p == pt(1.5, 1.5));
160    assert(line_intersection(pt(0, 0), pt(1, 1), pt(0, 4), pt(4, 0), &p) == 0);
161    assert(p == pt(2, 2));
162
163    //tests for segment intersection (examples in order from link below)
164    //http://martin-thoma.com/how-to-check-if-two-line-segments-intersect/
165    {
166  #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)
167      pt p, q;
168      //intersection is a point
169      assert(0 == test(-4, 0, 4, 0, 0, -4, 0, 4));    assert(p == pt(0, 0));
170      assert(0 == test(0, 0, 10, 10, 2, 2, 16, 4));   assert(p == pt(2, 2));
171      assert(0 == test(-2, 2, -2, -2, -2, 0, 0, 0));  assert(p == pt(-2, 0));
172      assert(0 == test(0, 4, 4, 4, 4, 0, 4, 8));      assert(p == pt(4, 4));
173
174      //intersection is a segment
175      assert(1 == test(10, 10, 0, 0, 2, 2, 6, 6));
176      assert(p == pt(2, 2) && q == pt(6, 6));
177      assert(1 == test(6, 8, 14, -2, 14, -2, 6, 8));
178      assert(p == pt(6, 8) && q == pt(14, -2));
179
180      //no intersection
181      assert(-1 == test(6, 8, 8, 10, 12, 12, 4, 4));
182      assert(-1 == test(-4, 2, -8, 8, 0, 0, -4, 6));
183      assert(-1 == test(4, 4, 4, 6, 0, 2, 0, 0));
184      assert(-1 == test(4, 4, 6, 4, 0, 2, 0, 0));
185      assert(-1 == test(-2, -2, 4, 4, 10, 10, 6, 6));
186      assert(-1 == test(0, 0, 2, 2, 4, 0, 1, 4));
187      assert(-1 == test(2, 2, 2, 8, 4, 4, 6, 4));
188      assert(-1 == test(4, 2, 4, 4, 0, 8, 10, 0));
189    }
190    assert(pt(2.5, 2.5) == closest_point(-1, -1, 5, pt(0, 0)));
191    assert(pt(3, 0)    == closest_point(1, 0, -3, pt(0, 0)));
192    assert(pt(0, 3)    == closest_point(0, 1, -3, pt(0, 0)));
193
194    assert(pt(3, 0)  == closest_point(pt(3, 0), pt(3, 3), pt(0, 0)));
195    assert(pt(2, -1) == closest_point(pt(2, -1), pt(4, -1), pt(0, 0)));
196    assert(pt(4, -1) == closest_point(pt(2, -1), pt(4, -1), pt(5, 0)));
197    return 0;
198  }
```

5.2.4   Circle Intersections

```
1  /*
2
3  5.2.4 - Circle Intersection (2D)
4
```

```
5   Tangent lines to circles, circle-line intersections, and circle-circle
6   intersections (intersection point(s) as well as area) in 2 dimensions.
7
8   All operations are O(1) in time and space.
9
10  */
11
12  #include <algorithm> /* std::min(), std::max() */
13  #include <cmath>      /* acos(), fabs(), sqrt() */
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 NE(a, b) (fabs((a) - (b)) > eps)  /* not equal to */
24  #define LT(a, b) ((a) < (b) - eps)        /* less than */
25  #define GT(a, b) ((a) > (b) + eps)        /* greater than */
26  #define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */
27  #define GE(a, b) ((a) >= (b) - eps)       /* greater than or equal to */
28
29  struct circle {
30    double h, k, r;
31
32    circle(const double & h, const double & k, const double & r) {
33      this->h = h;
34      this->k = k;
35      this->r = r;
36    }
37  };
38
39  //note: this is a simplified version of line that is not canonicalized.
40  // e.g. comparing lines with == signs will not work as intended. For a
41  //      fully featured line class, see the whole geometry library.
42  struct line {
43    double a, b, c;
44
45    line() { a = b = c = 0; }
46
47    line(const double & a, const double & b, const double & c) {
48      this->a = a;
49      this->b = b;
50      this->c = c;
51    }
52
53    line(const point & p, const point & q) {
54      a = p.y - q.y,
55      b = q.x - p.x;
56      c = -a * p.x - b * p.y;
57    }
58  };
59
60  double norm(const point & a) { return a.x * a.x + a.y * a.y; }
61  double abs(const point & a) { return sqrt(norm(a)); }
62  double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
63
```

```
64   //tangent line(s) to circle c passing through p. there are 3 cases:
65   //returns: 0, if there are no lines (p is strictly inside c)
66   //           1, if there is 1 tangent line (p is on the edge)
67   //           2, if there are 2 tangent lines (p is strictly outside)
68   //If there is only 1 tangent, then the line will be stored in l1.
69   //If there are 2, then they will be stored in l1 and l2 respectively.
70   int tangents(const circle & c, const point & p, line * l1 = 0, line * l2 = 0) {
71     point vop(p.x - c.h, p.y - c.k);
72     if (EQ(norm(vop), c.r * c.r)) { //on an edge, get perpendicular through p
73       if (l1 != 0) {
74         *l1 = line(point(c.h, c.k), p);
75         *l1 = line(-l1->b, l1->a, l1->b * p.x - l1->a * p.y);
76       }
77       return 1;
78     }
79     if (LE(norm(vop), c.r * c.r)) return 0; //inside circle
80     point q(vop.x / c.r, vop.y / c.r);
81     double n = norm(q), d = q.y * sqrt(norm(q) - 1.0);
82     point t1((q.x - d) / n, c.k), t2((q.x + d) / n, c.k);
83     if (NE(q.y, 0)) { //common case
84       t1.y += c.r * (1.0 - t1.x * q.x) / q.y;
85       t2.y += c.r * (1.0 - t2.x * q.x) / q.y;
86     } else { //point at center horizontal, y = 0
87       d = c.r * sqrt(1.0 - t1.x * t1.x);
88       t1.y += d;
89       t2.y -= d;
90     }
91     t1.x = t1.x * c.r + c.h;
92     t2.x = t2.x * c.r + c.h;
93     //note: here, t1 and t2 are the two points of tangencies
94     if (l1 != 0) *l1 = line(p, t1);
95     if (l2 != 0) *l2 = line(p, t2);
96     return 2;
97   }
98
99   //determines the intersection(s) between a circle c and line l
100  //returns: 0, if the line does not intersect with the circle
101  //           1, if the line is tangent (one intersection)
102  //           2, if the line crosses through the circle
103  //If there is 1 intersection point, it will be stored in p
104  //If there are 2, they will be stored in p and q respectively
105  int intersection(const circle & c, const line & l,
106                   point * p = 0, point * q = 0) {
107    double v = c.h * l.a + c.k * l.b + l.c;
108    double aabb = l.a * l.a + l.b * l.b;
109    double disc = v * v / aabb - c.r * c.r;
110    if (disc > eps) return 0;
111    double x0 = -l.a * l.c / aabb, y0 = -l.b * v / aabb;
112    if (disc > -eps) {
113      if (p != 0) *p = point(x0 + c.h, y0 + c.k);
114      return 1;
115    }
116    double k = sqrt((disc /= -aabb) < 0 ? 0 : disc);
117    if (p != 0) *p = point(x0 + k * l.b + c.h, y0 - k * l.a + c.k);
118    if (q != 0) *q = point(x0 - k * l.b + c.h, y0 + k * l.a + c.k);
119    return 2;
120  }
121
122  //determines the intersection points between two circles c1 and c2
```

```
123   //returns: -2, if circle c2 completely encloses circle c1
124   //          -1, if circle c1 completely encloses circle c2
125   //           0, if the circles are completely disjoint
126   //           1, if the circles are tangent (one intersection point)
127   //           2, if the circles intersect at two points
128   //           3, if the circles intersect at infinite points (c1 = c2)
129   //If one intersection, the intersection point is stored in p
130   //If two, the intersection points are stored in p and q respectively
131   int intersection(const circle & c1, const circle & c2,
132                    point * p = 0, point * q = 0) {
133     if (EQ(c1.h, c2.h) && EQ(c1.k, c2.k))
134       return EQ(c1.r, c2.r) ? 3 : (c1.r > c2.r ? -1 : -2);
135     point d12(point(c2.h - c1.h, c2.k - c1.k));
136     double d = abs(d12);
137     if (GT(d, c1.r + c2.r)) return 0;
138     if (LT(d, fabs(c1.r - c2.r))) return c1.r > c2.r ? -1 : -2;
139     double a = (c1.r * c1.r - c2.r * c2.r + d * d) / (2 * d);
140     double x0 = c1.h + (d12.x * a / d);
141     double y0 = c1.k + (d12.y * a / d);
142     double s = sqrt(c1.r * c1.r - a * a);
143     double rx = -d12.y * s / d, ry = d12.x * s / d;
144     if (EQ(rx, 0) && EQ(ry, 0)) {
145       if (p != 0) *p = point(x0, y0);
146       return 1;
147     }
148     if (p != 0) *p = point(x0 - rx, y0 - ry);
149     if (q != 0) *q = point(x0 + rx, y0 + ry);
150     return 2;
151   }
152
153   const double PI = acos(-1.0);
154
155   //intersection area of circles c1 and c2
156   double intersection_area(const circle & c1, const circle c2) {
157     double r = std::min(c1.r, c2.r), R = std::max(c1.r, c2.r);
158     double d = abs(point(c2.h - c1.h, c2.k - c1.k));
159     if (LE(d, R - r)) return PI * r * r;
160     if (GE(d, R + r)) return 0;
161     return r * r * acos((d * d + r * r - R * R) / 2 / d / r) +
162            R * R * acos((d * d + R * R - r * r) / 2 / d / R) -
163            0.5 * sqrt((-d + r + R) * (d + r - R) * (d - r + R) * (d + r + R));
164   }
165
166   /*** Example Usage ***/
167
168   #include <cassert>
169   #include <iostream>
170   using namespace std;
171   #define pt point
172
173   int main() {
174     line l1, l2;
175     assert(0 == tangents(circle(0, 0, 4), pt(1, 1), &l1, &l2));
176     assert(1 == tangents(circle(0, 0, sqrt(2)), pt(1, 1), &l1, &l2));
177     cout << l1.a << " " << l1.b << " " << l1.c << "\n"; // -x - y + 2 = 0
178     assert(2 == tangents(circle(0, 0, 2), pt(2, 2), &l1, &l2));
179     cout << l1.a << " " << l1.b << " " << l1.c << "\n"; //    -2y + 4 = 0
180     cout << l2.a << " " << l2.b << " " << l2.c << "\n"; // 2x      - 4 = 0
181
```

```
182    pt p, q;
183    assert(0 == intersection(circle(1, 1, 3), line(5, 3, -30), &p, &q));
184    assert(1 == intersection(circle(1, 1, 3), line(0, 1, -4), &p, &q));
185    assert(p == pt(1, 4));
186    assert(2 == intersection(circle(1, 1, 3), line(0, 1, -1), &p, &q));
187    assert(p == pt(4, 1));
188    assert(q == pt(-2, 1));
189
190    assert(-2 == intersection(circle(1, 1, 1), circle(0, 0, 3), &p, &q));
191    assert(-1 == intersection(circle(0, 0, 3), circle(1, 1, 1), &p, &q));
192    assert(0 == intersection(circle(5, 0, 4), circle(-5, 0, 4), &p, &q));
193    assert(1 == intersection(circle(-5, 0, 5), circle(5, 0, 5), &p, &q));
194    assert(p == pt(0, 0));
195    assert(2 == intersection(circle(-0.5, 0, 1), circle(0.5, 0, 1), &p, &q));
196    assert(p == pt(0, -sqrt(3) / 2));
197    assert(q == pt(0, sqrt(3) / 2));
198
199    //example where each circle passes through the other circle's center
200    //http://math.stackexchange.com/a/402891
201    double r = 3;
202    double a = intersection_area(circle(-r / 2, 0, r), circle(r / 2, 0, r));
203    assert(EQ(a, r * r * (2 * PI / 3 - sqrt(3) / 2)));
204    return 0;
205 }
```

5.3   Common Geometric Computations

5.3.1   Polygon Sorting and Area

```
1  /*
2
3  5.3.1 - Polygon Sorting and Area
4
5  centroid() - Simply returns the geometric average point of all the
6  points given. This could be used to find the reference center point
7  for the following function. An empty range will result in (0, 0).
8  Complexity: O(n) on the number of points in the given range.
9
10 cw_comp() - Given a set of points, these points could possibly form
11 many different polygons. The following sorting comparators, when
12 used in conjunction with std::sort, will produce one such ordering
13 of points which is sorted in clockwise order relative to a custom-
14 defined center point that must be set beforehand. This could very
15 well be the result of mean_point(). ccw_comp() is the opposite
16 function, which produces the points in counterclockwise order.
17 Complexity: O(1) per call.
18
19 polygon_area() - A given range of points is interpreted as a polygon
20 based on the ordering they're given in. The shoelace formula is used
21 to determine its area. The polygon does not necessarily have to be
22 sorted using one of the functions above, but may be any ordering that
23 produces a valid polygon. You may optionally pass the last point in
24 the range equal to the first point and still expect the correct result.
25 Complexity: O(n) on the number of points in the range, assuming that
26 the points are already sorted in the order that specifies the polygon.
27
```

```
28  */
29
30  #include <algorithm> /* std::sort() */
31  #include <cmath>     /* fabs() */
32  #include <utility>   /* std::pair */
33
34  typedef std::pair<double, double> point;
35  #define x first
36  #define y second
37
38  const double eps = 1e-9;
39
40  #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
41  #define LT(a, b) ((a) < (b) - eps)        /* less than */
42  #define GE(a, b) ((a) >= (b) - eps)       /* greater than or equal to */
43
44  //magnitude of the 3D cross product with Z component implicitly equal to 0
45  //the answer assumes the origin (0, 0) is instead shifted to point o.
46  //this is equal to 2x the signed area of the triangle from these 3 points.
47  double cross(const point & o, const point & a, const point & b) {
48    return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
49  }
50
51  point ctr;
52
53  template<class It> point centroid(It lo, It hi) {
54    if (lo == hi) return point(0, 0);
55    double xtot = 0, ytot = 0, points = hi - lo;
56    for (; lo != hi; ++lo) {
57      xtot += lo->x;
58      ytot += lo->y;
59    }
60    return point(xtot / points, ytot / points);
61  }
62
63  //ctr must be defined beforehand
64  bool cw_comp(const point & a, const point & b) {
65    if (GE(a.x - ctr.x, 0) && LT(b.x - ctr.x, 0)) return true;
66    if (LT(a.x - ctr.x, 0) && GE(b.x - ctr.x, 0)) return false;
67    if (EQ(a.x - ctr.x, 0) && EQ(b.x - ctr.x, 0)) {
68      if (GE(a.y - ctr.y, 0) || GE(b.y - ctr.y, 0))
69        return a.y > b.y;
70      return b.y > a.y;
71    }
72    double det = cross(ctr, a, b);
73    if (EQ(det, 0))
74      return (a.x - ctr.x) * (a.x - ctr.x) + (a.y - ctr.y) * (a.y - ctr.y) >
75             (b.x - ctr.x) * (b.x - ctr.x) + (b.y - ctr.y) * (b.y - ctr.y);
76    return det < 0;
77  }
78
79  bool ccw_comp(const point & a, const point & b) {
80    return cw_comp(b, a);
81  }
82
83  //area of a polygon specified by range [lo, hi) - shoelace formula in O(n)
84  //[lo, hi) must point to the polygon vertices, sorted in CW or CCW order
85  template<class It> double polygon_area(It lo, It hi) {
86    if (lo == hi) return 0;
```

```
87    double area = 0;
88    if (*lo != *--hi)
89      area += (lo->x - hi->x) * (lo->y + hi->y);
90    for (It i = hi, j = hi - 1; i != lo; --i, --j)
91      area += (i->x - j->x) * (i->y + j->y);
92    return fabs(area / 2.0);
93  }
94
95  /*** Example Usage ***/
96
97  #include <cassert>
98  #include <vector>
99  using namespace std;
100 #define pt point
101
102 int main() {
103   //irregular pentagon with only (1, 2) not on the convex hull
104   //the ordering here is already sorted in ccw order around their centroid
105   //we will scramble them and see if our comparator works
106   pt pts[] = {pt(1, 3), pt(1, 2), pt(2, 1), pt(0, 0), pt(-1, 3)};
107   vector<pt> v(pts, pts + 5);
108   std::random_shuffle(v.begin(), v.end());
109   ctr = centroid(v.begin(), v.end()); //note: ctr is a global variable
110   assert(EQ(ctr.x, 0.6) && EQ(ctr.y, 1.8));
111   sort(v.begin(), v.end(), cw_comp);
112   for (int i = 0; i < (int)v.size(); i++) assert(v[i] == pts[i]);
113   assert(EQ(polygon_area(v.begin(), v.end()), 5));
114   return 0;
115 }
```

5.3.2   Point in Polygon Query

```
1   /*
2
3   5.3.2 - Point in Polygon Query
4
5   Given a single point p and another range of points specifying a
6   polygon, determine whether p lies within the polygon. Note that
7   you should modify the EDGE_IS_INSIDE flag, depending on whether
8   you wish for the algorithm to consider points lying on an edge of
9   the polygon to be inside it.
10
11  Complexity: O(n) on the number of vertices in the polygon.
12
13  */
14
15  #include <algorithm> /* std::sort() */
16  #include <cmath>     /* fabs() */
17  #include <utility>   /* std::pair */
18
19  typedef std::pair<double, double> point;
20  #define x first
21  #define y second
22
23  const double eps = 1e-9;
24
25  #define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
```

```
26  #define GT(a, b) ((a) > (b) + eps)        /* greater than */
27  #define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */
28
29  //should we consider points lying on an edge to be inside the polygon?
30  const bool EDGE_IS_INSIDE = true;
31
32  //magnitude of the 3D cross product with Z component implicitly equal to 0
33  //the answer assumes the origin (0, 0) is instead shifted to point o.
34  //this is equal to 2x the signed area of the triangle from these 3 points.
35  double cross(const point & o, const point & a, const point & b) {
36    return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
37  }
38
39  //return whether point p is in polygon specified by range [lo, hi) in O(n)
40  //[lo, hi) must point to the polygon vertices, sorted in CW or CCW order
41  template<class It> bool point_in_polygon(const point & p, It lo, It hi) {
42    int cnt = 0;
43    for (It i = lo, j = hi - 1; i != hi; j = i++) {
44      if (EQ(i->y, p.y) && (EQ(i->x, p.x) ||
45                            (EQ(j->y, p.y) && (LE(i->x, p.x) || LE(j->x, p.x)))))
46        return EDGE_IS_INSIDE; //on an edge
47      if (GT(i->y, p.y) != GT(j->y, p.y)) {
48        double det = cross(p, *i, *j);
49        if (EQ(det, 0)) return EDGE_IS_INSIDE; //on an edge
50        if (GT(det, 0) != GT(j->y, i->y)) cnt++;
51      }
52    }
53    return cnt % 2 == 1;
54  }
55
56  /*** Example Usage ***/
57
58  #include <cassert>
59  using namespace std;
60  #define pt point
61
62  int main() {
63    //irregular trapezoid
64    pt p[] = {pt(-1, 3), pt(1, 3), pt(2, 1), pt(0, 0)};
65    assert(point_in_polygon(pt(1, 2), p, p + 4));
66    assert(point_in_polygon(pt(0, 3), p, p + 4));
67    assert(!point_in_polygon(pt(0, 3.01), p, p + 4));
68    assert(!point_in_polygon(pt(2, 2), p, p + 4));
69    return 0;
70  }
```

5.3.3   Convex Hull

```
1  /*
2
3  5.3.3 - 2D Convex Hull
4
5  Determines the convex hull from a range of points, that is, the
6  smallest convex polygon (a polygon such that every line which
7  crosses through it will only cross through it once) that contains
8  all of the points. This function uses the monotone chain algorithm
9  to compute the upper and lower hulls separately.
```

```
10
11   Returns: a vector of the convex hull points in clockwise order.
12   Complexity: O(n log n) on the number of points given
13
14   Notes: To yield the hull points in counterclockwise order,
15          replace every usage of GE() in the function with LE().
16          To have the first point on the hull repeated as the last,
17          replace the last line of the function to res.resize(k);
18
19   */
20
21   #include <algorithm> /* std::sort() */
22   #include <cmath>      /* fabs() */
23   #include <utility>    /* std::pair */
24   #include <vector>
25
26   typedef std::pair<double, double> point;
27   #define x first
28   #define y second
29
30   //change < 0 comparisons to > 0 to produce hull points in CCW order
31   double cw(const point & o, const point & a, const point & b) {
32     return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x) < 0;
33   }
34
35   //convex hull from a range [lo, hi) of points
36   //monotone chain in O(n log n) to find hull points in CW order
37   //notes: the range of input points will be sorted lexicographically
38   template<class It> std::vector<point> convex_hull(It lo, It hi) {
39     int k = 0;
40     if (hi - lo <= 1) return std::vector<point>(lo, hi);
41     std::vector<point> res(2 * (int)(hi - lo));
42     std::sort(lo, hi); //compare by x, then by y if x-values are equal
43     for (It it = lo; it != hi; ++it) {
44       while (k >= 2 && !cw(res[k - 2], res[k - 1], *it)) k--;
45       res[k++] = *it;
46     }
47     int t = k + 1;
48     for (It it = hi - 2; it != lo - 1; --it) {
49       while (k >= t && !cw(res[k - 2], res[k - 1], *it)) k--;
50       res[k++] = *it;
51     }
52     res.resize(k - 1);
53     return res;
54   }
55
56   /*** Example Usage ***/
57
58   #include <iostream>
59   using namespace std;
60
61   int main() {
62     //irregular pentagon with only (1, 2) not on the convex hull
63     vector<point> v;
64     v.push_back(point(1, 3));
65     v.push_back(point(1, 2));
66     v.push_back(point(2, 1));
67     v.push_back(point(0, 0));
68     v.push_back(point(-1, 3));
```

```
69    std::random_shuffle(v.begin(), v.end());
70    vector<point> h = convex_hull(v.begin(), v.end());
71    cout << "hull␣points:";
72    for (int i = 0; i < (int)h.size(); i++)
73      cout << "␣(" << h[i].x << "," << h[i].y << ")";
74    cout << "\n";
75    return 0;
76  }
```

5.3.4   Minimum Enclosing Circle

```
1   /*
2
3   5.3.4 - Minimum Enclosing Circle (2D)
4
5   Given a range of points on the 2D cartesian plane, determine
6   the equation of the circle with smallest possible area which
7   encloses all of the points. Note: in an attempt to avoid the
8   worst case, the circles are randomly shuffled before the
9   algorithm is performed. This is not necessary to obtain the
10  correct answer, and may be removed if the input order must
11  be preserved.
12
13  Time Complexity: O(n) average on the number of points given.
14
15  */
16
17  #include <algorithm>
18  #include <cmath>
19  #include <stdexcept>
20  #include <utility>
21
22  const double eps = 1e-9;
23
24  #define LE(a, b) ((a) <= (b) + eps)        /* less than or equal to */
25
26  typedef std::pair<double, double> point;
27  #define x first
28  #define y second
29
30  double norm(const point & a) { return a.x * a.x + a.y * a.y; }
31  double abs(const point & a) { return sqrt(norm(a)); }
32
33  struct circle {
34
35    double h, k, r;
36
37    circle(): h(0), k(0), r(0) {}
38    circle(const double & H, const double & K, const double & R):
39      h(H), k(K), r(fabs(R)) {}
40
41    //circumcircle with the diameter equal to the distance from a to b
42    circle(const point & a, const point & b) {
43      h = (a.x + b.x) / 2.0;
44      k = (a.y + b.y) / 2.0;
45      r = abs(point(a.x - h, a.y - k));
46    }
```

```
47
48    //circumcircle of 3 points - throws exception if abc are collinear/equal
49    circle(const point & a, const point & b, const point & c) {
50      double an = norm(point(b.x - c.x, b.y - c.y));
51      double bn = norm(point(a.x - c.x, a.y - c.y));
52      double cn = norm(point(a.x - b.x, a.y - b.y));
53      double wa = an * (bn + cn - an);
54      double wb = bn * (an + cn - bn);
55      double wc = cn * (an + bn - cn);
56      double w = wa + wb + wc;
57      if (fabs(w) < eps)
58        throw std::runtime_error("No circle from collinear points.");
59      h = (wa * a.x + wb * b.x + wc * c.x) / w;
60      k = (wa * a.y + wb * b.y + wc * c.y) / w;
61      r = abs(point(a.x - h, a.y - k));
62    }
63
64    bool contains(const point & p) const {
65      return LE(norm(point(p.x - h, p.y - k)), r * r);
66    }
67
68  };
69
70  template<class It> circle smallest_circle(It lo, It hi) {
71    if (lo == hi) return circle(0, 0, 0);
72    if (lo + 1 == hi) return circle(lo->x, lo->y, 0);
73    std::random_shuffle(lo, hi);
74    circle res(*lo, *(lo + 1));
75    for (It i = lo + 2; i != hi; ++i) {
76      if (res.contains(*i)) continue;
77      res = circle(*lo, *i);
78      for (It j = lo + 1; j != i; ++j) {
79        if (res.contains(*j)) continue;
80        res = circle(*i, *j);
81        for (It k = lo; k != j; ++k)
82          if (!res.contains(*k)) res = circle(*i, *j, *k);
83      }
84    }
85    return res;
86  }
87
88  /*** Example Usage ***/
89
90  #include <iostream>
91  #include <vector>
92  using namespace std;
93
94  int main() {
95    vector<point> v;
96    v.push_back(point(0, 0));
97    v.push_back(point(0, 1));
98    v.push_back(point(1, 0));
99    v.push_back(point(1, 1));
100   circle res = smallest_circle(v.begin(), v.end());
101   cout << "center: (" << res.h << ", " << res.k << ")\n";
102   cout << "radius: " << res.r << "\n";
103   return 0;
104 }
```

5.3.5   Diameter of Point Set

```
1   /*
2
3   5.3.5 - Diameter of Point Set (2D)
4
5   Determines the diametral pair of a range of points. The diamter
6   of a set of points is the largest distance between any two
7   points in the set. A diametral pair is a pair of points in the
8   set whose distance is equal to the set's diameter. The following
9   program uses rotating calipers method to find a solution.
10
11  Time Complexity: O(n log n) on the number of points in the set.
12
13  */
14
15  #include <algorithm> /* std::sort() */
16  #include <cmath>      /* fabs(), sqrt() */
17  #include <utility>   /* std::pair */
18  #include <vector>
19
20  typedef std::pair<double, double> point;
21  #define x first
22  #define y second
23
24  double sqdist(const point & a, const point & b) {
25    double dx = a.x - b.x, dy = a.y - b.y;
26    return sqrt(dx * dx + dy * dy);
27  }
28
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  bool cw(const point & o, const point & a, const point & b) {
34    return cross(o, a, b) < 0;
35  }
36
37  double area(const point & o, const point & a, const point & b) {
38    return fabs(cross(o, a, b));
39  }
40
41  template<class It> std::vector<point> convex_hull(It lo, It hi) {
42    int k = 0;
43    if (hi - lo <= 1) return std::vector<point>(lo, hi);
44    std::vector<point> res(2 * (int)(hi - lo));
45    std::sort(lo, hi); //compare by x, then by y if x-values are equal
46    for (It it = lo; it != hi; ++it) {
47      while (k >= 2 && !cw(res[k - 2], res[k - 1], *it)) k--;
48      res[k++] = *it;
49    }
50    int t = k + 1;
51    for (It it = hi - 2; it != lo - 1; --it) {
52      while (k >= t && !cw(res[k - 2], res[k - 1], *it)) k--;
53      res[k++] = *it;
54    }
55    res.resize(k - 1);
56    return res;
```

```
57  }
58
59  template<class It> std::pair<point, point> diametral_pair(It lo, It hi) {
60    std::vector<point> h = convex_hull(lo, hi);
61    int m = h.size();
62    if (m == 1) return std::make_pair(h[0], h[0]);
63    if (m == 2) return std::make_pair(h[0], h[1]);
64    int k = 1;
65    while (area(h[m - 1], h[0], h[(k + 1) % m]) > area(h[m - 1], h[0], h[k]))
66      k++;
67    double maxdist = 0, d;
68    std::pair<point, point> res;
69    for (int i = 0, j = k; i <= k && j < m; i++) {
70      d = sqdist(h[i], h[j]);
71      if (d > maxdist) {
72        maxdist = d;
73        res = std::make_pair(h[i], h[j]);
74      }
75      while (j < m && area(h[i], h[(i + 1) % m], h[(j + 1) % m]) >
76                     area(h[i], h[(i + 1) % m], h[j])) {
77        d = sqdist(h[i], h[(j + 1) % m]);
78        if (d > maxdist) {
79          maxdist = d;
80          res = std::make_pair(h[i], h[(j + 1) % m]);
81        }
82        j++;
83      }
84    }
85    return res;
86  }
87
88  /*** Example Usage ***/
89
90  #include <iostream>
91  using namespace std;
92
93  int main() {
94    vector<point> v;
95    v.push_back(point(0, 0));
96    v.push_back(point(3, 0));
97    v.push_back(point(0, 3));
98    v.push_back(point(1, 1));
99    v.push_back(point(4, 4));
100   pair<point, point> res = diametral_pair(v.begin(), v.end());
101   cout << "diametral pair: (" << res.first.x << "," << res.first.y << ") ";
102   cout << "(" << res.second.x << "," << res.second.y << ")\n";
103   cout << "diameter: " << sqrt(sqdist(res.first, res.second)) << "\n";
104   return 0;
105 }
```

5.3.6   Closest Point Pair

```
1  /*
2
3  5.3.6 - Closest Point Pair (2D)
4
5  Given a range containing distinct points on the Cartesian plane,
```

```
6   determine two points which have the closest possible distance.
7   A divide and conquer algorithm is used. Note that the ordering
8   of points in the input range may be changed by the function.
9
10  Time Complexity: O(n log^2 n) where n is the number of points.
11
12  */
13
14  #include <algorithm> /* std::min, std::sort */
15  #include <cfloat>     /* DBL_MAX */
16  #include <cmath>      /* fabs */
17  #include <utility>    /* std::pair */
18
19  typedef std::pair<double, double> point;
20  #define x first
21  #define y second
22
23  double sqdist(const point & a, const point & b) {
24    double dx = a.x - b.x, dy = a.y - b.y;
25    return dx * dx + dy * dy;
26  }
27
28  bool cmp_x(const point & a, const point & b) { return a.x < b.x; }
29  bool cmp_y(const point & a, const point & b) { return a.y < b.y; }
30
31  template<class It>
32  double rec(It lo, It hi, std::pair<point, point> & res, double mindist) {
33    if (lo == hi) return DBL_MAX;
34    It mid = lo + (hi - lo) / 2;
35    double midx = mid->x;
36    double d1 = rec(lo, mid, res, mindist);
37    mindist = std::min(mindist, d1);
38    double d2 = rec(mid + 1, hi, res, mindist);
39    mindist = std::min(mindist, d2);
40    std::sort(lo, hi, cmp_y);
41    int size = 0;
42    It t[hi - lo];
43    for (It it = lo; it != hi; ++it)
44      if (fabs(it->x - midx) < mindist)
45        t[size++] = it;
46    for (int i = 0; i < size; i++) {
47      for (int j = i + 1; j < size; j++) {
48        point a = *t[i], b = *t[j];
49        if (b.y - a.y >= mindist) break;
50        double dist = sqdist(a, b);
51        if (mindist > dist) {
52          mindist = dist;
53          res = std::make_pair(a, b);
54        }
55      }
56    }
57    return mindist;
58  }
59
60  template<class It> std::pair<point, point> closest_pair(It lo, It hi) {
61    std::pair<point, point> res;
62    std::sort(lo, hi, cmp_x);
63    rec(lo, hi, res, DBL_MAX);
64    return res;
```

```
65  }
66
67  /*** Example Usage ***/
68
69  #include <iostream>
70  #include <vector>
71  using namespace std;
72
73  int main() {
74    vector<point> v;
75    v.push_back(point(2, 3));
76    v.push_back(point(12, 30));
77    v.push_back(point(40, 50));
78    v.push_back(point(5, 1));
79    v.push_back(point(12, 10));
80    v.push_back(point(3, 4));
81    pair<point, point> res = closest_pair(v.begin(), v.end());
82    cout << "closest␣pair:␣(" << res.first.x << "," << res.first.y << ")␣";
83    cout << "(" << res.second.x << "," << res.second.y << ")\n";
84    cout << "dist:␣" << sqrt(sqdist(res.first, res.second)) << "\n"; //1.41421
85    return 0;
86  }
```

5.3.7   Segment Intersection Finding

```
1   /*
2
3   5.3.7 - Segment Intersection Finding
4
5   Given a range of segments on the Cartesian plane, identify one
6   pair of segments which intersect each other. This is done using
7   a sweep line algorithm.
8
9   Time Complexity: O(n log n) where n is the number of segments.
10
11  */
12
13  #include <algorithm> /* std::min(), std::max(), std::sort() */
14  #include <cmath>     /* fabs() */
15  #include <set>
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  double norm(const point & a) { return a.x * a.x + a.y * a.y; }
29  double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
30  double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }
31  double cross(const point & o, const point & a, const point & b) {
32    return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
```

```
33   }
34
35   const bool TOUCH_IS_INTERSECT = true;
36
37   bool contain(const double & l, const double & m, const double & h) {
38     if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
39     return LT(l, m) && LT(m, h);
40   }
41
42   bool overlap(const double & l1, const double & h1,
43               const double & l2, const double & h2) {
44     if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
45     return LT(l1, h2) && LT(l2, h1);
46   }
47
48   int seg_intersection(const point & a, const point & b,
49                        const point & c, const point & d) {
50     point ab(b.x - a.x, b.y - a.y);
51     point ac(c.x - a.x, c.y - a.y);
52     point cd(d.x - c.x, d.y - c.y);
53     double c1 = cross(ab, cd), c2 = cross(ac, ab);
54     if (EQ(c1, 0) && EQ(c2, 0)) {
55       double t0 = dot(ac, ab) / norm(ab);
56       double t1 = t0 + dot(cd, ab) / norm(ab);
57       if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
58         point res1 = std::max(std::min(a, b), std::min(c, d));
59         point res2 = std::min(std::max(a, b), std::max(c, d));
60         return (res1 == res2) ? 0 : 1;
61       }
62       return -1;
63     }
64     if (EQ(c1, 0)) return -1;
65     double t = cross(ac, cd) / c1, u = c2 / c1;
66     if (contain(0, t, 1) && contain(0, u, 1)) return 0;
67     return -1;
68   }
69
70   struct segment {
71     point p, q;
72
73     segment() {}
74     segment(const point & p, const point & q) {
75       if (p < q) {
76         this->p = p;
77         this->q = q;
78       } else {
79         this->p = q;
80         this->q = p;
81       }
82     }
83
84     bool operator < (const segment & rhs) const {
85       if (p.x < rhs.p.x) {
86         double c = cross(p, q, rhs.p);
87         if (c != 0) return c > 0;
88       } else if (p.x > rhs.p.x) {
89         double c = cross(rhs.p, rhs.q, q);
90         if (c != 0) return c < 0;
91       }
```

```
92       return p.y < rhs.p.y;
93     }
94   };
95
96   template<class SegIt> struct event {
97     point p;
98     int type;
99     SegIt seg;
100
101    event() {}
102    event(const point & p, const int type, SegIt seg) {
103      this->p = p;
104      this->type = type;
105      this->seg = seg;
106    }
107
108    bool operator < (const event & rhs) const {
109      if (p.x != rhs.p.x) return p.x < rhs.p.x;
110      if (type != rhs.type) return type > rhs.type;
111      return p.y < rhs.p.y;
112    }
113  };
114
115  bool intersect(const segment & s1, const segment & s2) {
116    return seg_intersection(s1.p, s1.q, s2.p, s2.q) >= 0;
117  }
118
119  //returns whether any pair of segments in the range [lo, hi) intersect
120  //if the result is true, one such intersection pair will be stored
121  //into values pointed to by res1 and res2.
122  template<class It>
123  bool find_intersection(It lo, It hi, segment * res1, segment * res2) {
124    int cnt = 0;
125    event<It> e[2 * (hi - lo)];
126    for (It it = lo; it != hi; ++it) {
127      if (it->p > it->q) std::swap(it->p, it->q);
128      e[cnt++] = event<It>(it->p, 1, it);
129      e[cnt++] = event<It>(it->q, -1, it);
130    }
131    std::sort(e, e + cnt);
132    std::set<segment> s;
133    std::set<segment>::iterator it, next, prev;
134    for (int i = 0; i < cnt; i++) {
135      It seg = e[i].seg;
136      if (e[i].type == 1) {
137        it = s.lower_bound(*seg);
138        if (it != s.end() && intersect(*it, *seg)) {
139          *res1 = *it; *res2 = *seg;
140          return true;
141        }
142        if (it != s.begin() && intersect(*--it, *seg)) {
143          *res1 = *it; *res2 = *seg;
144          return true;
145        }
146        s.insert(*seg);
147      } else {
148        it = s.lower_bound(*seg);
149        next = prev = it;
150        prev = it;
```

```
151        if (it != s.begin() && it != --s.end()) {
152          ++next;
153          --prev;
154          if (intersect(*next, *prev)) {
155            *res1 = *next; *res2 = *prev;
156            return true;
157          }
158        }
159        s.erase(it);
160      }
161    }
162    return false;
163  }
164
165  /*** Example Usage ***/
166
167  #include <iostream>
168  #include <vector>
169  using namespace std;
170
171  void print(const segment & s) {
172    cout << "(" << s.p.x << "," << s.p.y << ")<->";
173    cout << "(" << s.q.x << "," << s.q.y << ")\n";
174  }
175
176  int main() {
177    vector<segment> v;
178    v.push_back(segment(point(0, 0), point(2, 2)));
179    v.push_back(segment(point(3, 0), point(0, -1)));
180    v.push_back(segment(point(0, 2), point(2, -2)));
181    v.push_back(segment(point(0, 3), point(9, 0)));
182    segment res1, res2;
183    bool res = find_intersection(v.begin(), v.end(), &res1, &res2);
184    if (res) {
185      print(res1);
186      print(res2);
187    } else {
188      cout << "No intersections.\n";
189    }
190    return 0;
191  }
```

5.4   Advanced Geometric Computations

5.4.1   Convex Polygon Cut

```
1   /*
2
3   5.4.1 - Convex Polygon Cut
4
5   Given a range of points specifying a polygon on the Cartesian
6   plane, as well as two points specifying an infinite line, "cut"
7   off the right part of the polygon with the line and return the
8   resulting polygon that is the left part.
9
10  Time Complexity: O(n) on the number of points in the poylgon.
```

```
11
12   */
13
14   #include <cmath>    /* fabs() */
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   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 GT(a, b) ((a) > (b) + eps)         /* greater than */
27
28   double cross(const point & o, const point & a, const point & b) {
29     return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
30   }
31
32   int orientation(const point & o, const point & a, const point & b) {
33     double c = cross(o, a, b);
34     return LT(c, 0) ? -1 : (GT(c, 0) ? 1 : 0);
35   }
36
37   int line_intersection(const point & p1, const point & p2,
38                         const point & p3, const point & p4, point * p = 0) {
39     double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
40     double c1 = -(p1.x * p2.y - p2.x * p1.y);
41     double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
42     double c2 = -(p3.x * p4.y - p4.x * p3.y);
43     double x = -(c1 * b2 - c2 * b1), y = -(a1 * c2 - a2 * c1);
44     double det = a1 * b2 - a2 * b1;
45     if (EQ(det, 0))
46       return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
47     if (p != 0) *p = point(x / det, y / det);
48     return 0;
49   }
50
51   template<class It>
52   std::vector<point> convex_cut(It lo, It hi, const point & p, const point & q) {
53     std::vector<point> res;
54     for (It i = lo, j = hi - 1; i != hi; j = i++) {
55       int d1 = orientation(p, q, *j), d2 = orientation(p, q, *i);
56       if (d1 >= 0) res.push_back(*j);
57       if (d1 * d2 < 0) {
58         point r;
59         line_intersection(p, q, *j, *i, &r);
60         res.push_back(r);
61       }
62     }
63     return res;
64   }
65
66   /*** Example Usage ***/
67
68   #include <iostream>
69   using namespace std;
```

```
70
71  int main() {
72    //irregular pentagon with only (1, 2) not on the convex hull
73    vector<point> v;
74    v.push_back(point(1, 3));
75    v.push_back(point(1, 2));
76    v.push_back(point(2, 1));
77    v.push_back(point(0, 0));
78    v.push_back(point(-1, 3));
79    //cut using the vertical line through (0, 0)
80    vector<point> res = convex_cut(v.begin(), v.end(), point(0, 0), point(0, 1));
81    cout << "left␣cut:\n";
82    for (int i = 0; i < (int)res.size(); i++)
83      cout << "(" << res[i].x << "," << res[i].y << ")\n";
84    return 0;
85  }
```

5.4.2   Polygon Union and Intersection

```
1   /*
2
3   5.4.2 - Polygon Union and Intersection Area
4
5   Given two ranges of points respectively denoting the vertices of
6   two polygons, determine the intersection area of those polygons.
7   Using this, we can easily calculate their union with the forumla:
8     union_area(A, B) = area(A) + area(B) - intersection_area(A, B)
9
10  Time Complexity: O(n^2 log n), where n is the total number of vertices.
11
12  */
13
14  #include <algorithm> /* std::sort() */
15  #include <cmath>      /* fabs(), sqrt() */
16  #include <set>
17  #include <utility>   /* std::pair */
18  #include <vector>
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  typedef std::pair<double, double> point;
27  #define x first
28  #define y second
29
30  inline int sgn(const double & x) {
31    return (0.0 < x) - (x < 0.0);
32  }
33
34  //Line and line segment intersection (see their own sections)
35
36  int line_intersection(const point & p1, const point & p2,
37                        const point & p3, const point & p4, point * p = 0) {
38    double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
```

```
39      double c1 = -(p1.x * p2.y - p2.x * p1.y);
40      double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
41      double c2 = -(p3.x * p4.y - p4.x * p3.y);
42      double x = -(c1 * b2 - c2 * b1), y = -(a1 * c2 - a2 * c1);
43      double det = a1 * b2 - a2 * b1;
44      if (EQ(det, 0))
45        return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
46      if (p != 0) *p = point(x / det, y / det);
47      return 0;
48    }
49
50    double norm(const point & a) { return a.x * a.x + a.y * a.y; }
51    double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
52    double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }
53
54    const bool TOUCH_IS_INTERSECT = true;
55
56    bool contain(const double & l, const double & m, const double & h) {
57      if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
58      return LT(l, m) && LT(m, h);
59    }
60
61    bool overlap(const double & l1, const double & h1,
62                 const double & l2, const double & h2) {
63      if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
64      return LT(l1, h2) && LT(l2, h1);
65    }
66
67    int seg_intersection(const point & a, const point & b,
68                         const point & c, const point & d,
69                         point * p = 0, point * q = 0) {
70      point ab(b.x - a.x, b.y - a.y);
71      point ac(c.x - a.x, c.y - a.y);
72      point cd(d.x - c.x, d.y - c.y);
73      double c1 = cross(ab, cd), c2 = cross(ac, ab);
74      if (EQ(c1, 0) && EQ(c2, 0)) { //collinear
75        double t0 = dot(ac, ab) / norm(ab);
76        double t1 = t0 + dot(cd, ab) / norm(ab);
77        if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
78          point res1 = std::max(std::min(a, b), std::min(c, d));
79          point res2 = std::min(std::max(a, b), std::max(c, d));
80          if (res1 == res2) {
81            if (p != 0) *p = res1;
82            return 0; //collinear, meeting at an endpoint
83          }
84          if (p != 0 && q != 0) *p = res1, *q = res2;
85          return 1; //collinear and overlapping
86        } else {
87          return -1; //collinear and disjoint
88        }
89      }
90      if (EQ(c1, 0)) return -1; //parallel and disjoint
91      double t = cross(ac, cd) / c1, u = c2 / c1;
92      if (contain(0, t, 1) && contain(0, u, 1)) {
93        if (p != 0) *p = point(a.x + t * ab.x, a.y + t * ab.y);
94        return 0; //non-parallel with one intersection
95      }
96      return -1; //non-parallel with no intersections
97    }
```

```
98
99   struct event {
100    double y;
101    int mask_delta;
102
103    event(double y = 0, int mask_delta = 0) {
104      this->y = y;
105      this->mask_delta = mask_delta;
106    }
107
108    bool operator < (const event & e) const {
109      if (y != e.y) return y < e.y;
110      return mask_delta < e.mask_delta;
111    }
112  };
113
114  template<class It>
115  double intersection_area(It lo1, It hi1, It lo2, It hi2) {
116    It plo[2] = {lo1, lo2}, phi[] = {hi1, hi2};
117    std::set<double> xs;
118    for (It i1 = lo1; i1 != hi1; ++i1) xs.insert(i1->x);
119    for (It i2 = lo2; i2 != hi2; ++i2) xs.insert(i2->x);
120    for (It i1 = lo1, j1 = hi1 - 1; i1 != hi1; j1 = i1++) {
121      for (It i2 = lo2, j2 = hi2 - 1; i2 != hi2; j2 = i2++) {
122        point p;
123        if (seg_intersection(*i1, *j1, *i2, *j2, &p) == 0)
124          xs.insert(p.x);
125      }
126    }
127    std::vector<double> xsa(xs.begin(), xs.end());
128    double res = 0;
129    for (int k = 0; k < (int)xsa.size() - 1; k++) {
130      double x = (xsa[k] + xsa[k + 1]) / 2;
131      point sweep0(x, 0), sweep1(x, 1);
132      std::vector<event> events;
133      for (int poly = 0; poly < 2; poly++) {
134        It lo = plo[poly], hi = phi[poly];
135        double area = 0;
136        for (It i = lo, j = hi - 1; i != hi; j = i++)
137          area += (j->x - i->x) * (j->y + i->y);
138        for (It j = lo, i = hi - 1; j != hi; i = j++) {
139          point p;
140          if (line_intersection(*j, *i, sweep0, sweep1, &p) == 0) {
141            double y = p.y, x0 = i->x, x1 = j->x;
142            if (x0 < x && x1 > x) {
143              events.push_back(event(y,  sgn(area) * (1 << poly)));
144            } else if (x0 > x && x1 < x) {
145              events.push_back(event(y, -sgn(area) * (1 << poly)));
146            }
147          }
148        }
149      }
150      std::sort(events.begin(), events.end());
151      double a = 0.0;
152      int mask = 0;
153      for (int j = 0; j < (int)events.size(); j++) {
154        if (mask == 3)
155          a += events[j].y - events[j - 1].y;
156        mask += events[j].mask_delta;
```

```
157       }
158       res += a * (xsa[k + 1] - xsa[k]);
159     }
160     return res;
161   }
162
163   template<class It> double polygon_area(It lo, It hi) {
164     if (lo == hi) return 0;
165     double area = 0;
166     if (*lo != *--hi)
167       area += (lo->x - hi->x) * (lo->y + hi->y);
168     for (It i = hi, j = hi - 1; i != lo; --i, --j)
169       area += (i->x - j->x) * (i->y + j->y);
170     return fabs(area / 2.0);
171   }
172
173   template<class It>
174   double union_area(It lo1, It hi1, It lo2, It hi2) {
175     return polygon_area(lo1, hi1) + polygon_area(lo2, hi2) -
176            intersection_area(lo1, hi1, lo2, hi2);
177   }
178
179   /*** Example Usage ***/
180
181   #include <cassert>
182   using namespace std;
183
184   int main() {
185     vector<point> p1, p2;
186
187     //irregular pentagon with area 1.5 triangle in quadrant 2
188     p1.push_back(point(1, 3));
189     p1.push_back(point(1, 2));
190     p1.push_back(point(2, 1));
191     p1.push_back(point(0, 0));
192     p1.push_back(point(-1, 3));
193     //a big square in quadrant 2
194     p2.push_back(point(0, 0));
195     p2.push_back(point(0, 3));
196     p2.push_back(point(-3, 3));
197     p2.push_back(point(-3, 0));
198
199     assert(EQ(1.5, intersection_area(p1.begin(), p1.end(),
200                                      p2.begin(), p2.end())));
201     assert(EQ(12.5, union_area(p1.begin(), p1.end(),
202                                p2.begin(), p2.end())));
203     return 0;
204   }
```

5.4.3   Delaunay Triangulation (Simple)

```
1   /*
2
3   5.4.3 - Delaunay Triangulation (Simple)
4
5   Given a range of points P on the Cartesian plane, the Delaunay
6   Triangulation of said points is a set of non-overlapping triangles
```

```
 7   covering the entire convex hull of P, such that no point in P lies
 8   within the circumcircle of any of the resulting triangles. The
 9   triangulation maximizes the minimum angle of all the angles of the
10   triangles in the triangulation. In addition, for any point p in the
11   convex hull (not necessarily in P), the nearest point is guaranteed
12   to be a vertex of the enclosing triangle from the triangulation.
13   See: https://en.wikipedia.org/wiki/Delaunay_triangulation
14
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 triangulation may not exist (e.g. for a set of collinear points)
18   or it may not be unique (multiple possible triangulations may exist).
19   The following program assumes that a triangulation exists, and
20   produces one such valid result using one of the simplest algorithms
21   to solve this problem. It involves encasing the simplex in a circle
22   and rejecting the simplex if another point in the tessellation is
23   within the generalized circle.
24
25   Time Complexity: O(n^4) on the number of input points.
26
27   */
28
29   #include <algorithm> /* std::sort() */
30   #include <cmath>      /* fabs(), sqrt() */
31   #include <utility>    /* std::pair */
32   #include <vector>
33
34   const double eps = 1e-9;
35
36   #define EQ(a, b) (fabs((a) - (b)) <= eps) /* 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   typedef std::pair<double, double> point;
43   #define x first
44   #define y second
45
46   double norm(const point & a) { return a.x * a.x + a.y * a.y; }
47   double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
48   double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }
49
50   const bool TOUCH_IS_INTERSECT = false;
51
52   bool contain(const double & l, const double & m, const double & h) {
53     if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
54     return LT(l, m) && LT(m, h);
55   }
56
57   bool overlap(const double & l1, const double & h1,
58                const double & l2, const double & h2) {
59     if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
60     return LT(l1, h2) && LT(l2, h1);
61   }
62
63   int seg_intersection(const point & a, const point & b,
64                        const point & c, const point & d) {
65     point ab(b.x - a.x, b.y - a.y);
```

```
66     point ac(c.x - a.x, c.y - a.y);
67     point cd(d.x - c.x, d.y - c.y);
68     double c1 = cross(ab, cd), c2 = cross(ac, ab);
69     if (EQ(c1, 0) && EQ(c2, 0)) {
70       double t0 = dot(ac, ab) / norm(ab);
71       double t1 = t0 + dot(cd, ab) / norm(ab);
72       if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
73         point res1 = std::max(std::min(a, b), std::min(c, d));
74         point res2 = std::min(std::max(a, b), std::max(c, d));
75         return (res1 == res2) ? 0 : 1;
76       }
77       return -1;
78     }
79     if (EQ(c1, 0)) return -1;
80     double t = cross(ac, cd) / c1, u = c2 / c1;
81     if (contain(0, t, 1) && contain(0, u, 1)) return 0;
82     return -1;
83   }
84
85   struct triangle { point a, b, c; };
86
87   template<class It>
88   std::vector<triangle> delaunay_triangulation(It lo, It hi) {
89     int n = hi - lo;
90     std::vector<double> x, y, z;
91     for (It it = lo; it != hi; ++it) {
92       x.push_back(it->x);
93       y.push_back(it->y);
94       z.push_back((it->x) * (it->x) + (it->y) * (it->y));
95     }
96     std::vector<triangle> res;
97     for (int i = 0; i < n - 2; i++) {
98       for (int j = i + 1; j < n; j++) {
99         for (int k = i + 1; k < n; k++) {
100           if (j == k) continue;
101           double nx = (y[j] - y[i]) * (z[k] - z[i]) - (y[k] - y[i]) * (z[j] - z[i]);
102           double ny = (x[k] - x[i]) * (z[j] - z[i]) - (x[j] - x[i]) * (z[k] - z[i]);
103           double nz = (x[j] - x[i]) * (y[k] - y[i]) - (x[k] - x[i]) * (y[j] - y[i]);
104           if (GE(nz, 0)) continue;
105           bool done = false;
106           for (int m = 0; m < n; m++)
107             if (x[m] - x[i]) * nx + (y[m] - y[i]) * ny + (z[m] - z[i]) * nz > 0) {
108               done = true;
109               break;
110             }
111           if (!done) { //handle 4 points on a circle
112             point s1[] = { *(lo + i), *(lo + j), *(lo + k), *(lo + i) };
113             for (int t = 0; t < (int)res.size(); t++) {
114               point s2[] = { res[t].a, res[t].b, res[t].c, res[t].a };
115               for (int u = 0; u < 3; u++)
116                 for (int v = 0; v < 3; v++)
117                   if (seg_intersection(s1[u], s1[u + 1], s2[v], s2[v + 1]) == 0)
118                     goto skip;
119             }
120             res.push_back((triangle){*(lo + i), *(lo + j), *(lo + k)});
121           }
122   skip:;
123         }
124       }
```

```
125     }
126       return res;
127   }
128
129   /*** Example Usage ***/
130
131   #include <iostream>
132   using namespace std;
133
134   int main() {
135     vector<point> v;
136     v.push_back(point(1, 3));
137     v.push_back(point(1, 2));
138     v.push_back(point(2, 1));
139     v.push_back(point(0, 0));
140     v.push_back(point(-1, 3));
141     vector<triangle> dt = delaunay_triangulation(v.begin(), v.end());
142     for (int i = 0; i < (int)dt.size(); i++) {
143       cout << "Triangle:␣";
144       cout << "(" << dt[i].a.x << "," << dt[i].a.y << ")␣";
145       cout << "(" << dt[i].b.x << "," << dt[i].b.y << ")␣";
146       cout << "(" << dt[i].c.x << "," << dt[i].c.y << ")\n";
147     }
148     return 0;
149   }
```

5.4.4   Delaunay Triangulation (Fast)

```
1    /*
2
3    5.4.3 - Delaunay Triangulation (Fast)
4
5    Given a range of points P on the Cartesian plane, the Delaunay
6    Triangulation of said points is a set of non-overlapping triangles
7    covering the entire convex hull of P, such that no point in P lies
8    within the circumcircle of any of the resulting triangles. The
9    triangulation maximizes the minimum angle of all the angles of the
10   triangles in the triangulation. In addition, for any point p in the
11   convex hull (not necessarily in P), the nearest point is guaranteed
12   to be a vertex of the enclosing triangle from the triangulation.
13   See: https://en.wikipedia.org/wiki/Delaunay_triangulation
14
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. The following is a C++ adaptation of
19   a FORTRAN90 program, which applies a divide and conquer algorithm
20   with complex linear-time merging. The original program can be found
21   via the following link. It contains more thorough documentation,
22   comments, and debugging messages associated with the current asserts().
23   http://people.sc.fsu.edu/~burkardt/f_src/table_delaunay/table_delaunay.html
24
25   Time Complexity: O(n log n) on the number of input points.
26
27   */
28
29   #include <algorithm> /* std::min(), std::max() */
```

```
30  #include <cassert>
31  #include <cmath>     /* fabs(), sqrt() */
32  #include <utility>  /* std::pair */
33  #include <vector>
34
35  int wrap(int ival, int ilo, int ihi) {
36    int jlo = std::min(ilo, ihi), jhi = std::max(ilo, ihi);
37    int wide = jhi + 1 - jlo, res = jlo;
38    if (wide != 1)  {
39      assert(wide != 0);
40      int tmp = (ival - jlo) % wide;
41      if (tmp < 0) res += abs(wide);
42      res += tmp;
43    }
44    return res;
45  }
46
47  double epsilon() {
48    double r = 1;
49    while (1 < (double)(r + 1)) r /= 2;
50    return 2 * r;
51  }
52
53  void permute(int n, double a[][2], int p[]) {
54    for (int istart = 1; istart <= n; istart++) {
55      if (p[istart - 1] < 0) continue;
56      if (p[istart - 1] == istart) {
57        p[istart - 1] = -p[istart - 1];
58        continue;
59      }
60      double tmp0 = a[istart - 1][0];
61      double tmp1 = a[istart - 1][1];
62      int iget = istart;
63      for (;;) {
64        int iput = iget;
65        iget = p[iget - 1];
66        p[iput - 1] = -p[iput - 1];
67        assert(!(iget < 1 || n < iget));
68        if (iget == istart) {
69          a[iput - 1][0] = tmp0;
70          a[iput - 1][1] = tmp1;
71          break;
72        }
73        a[iput - 1][0] = a[iget - 1][0];
74        a[iput - 1][1] = a[iget - 1][1];
75      }
76    }
77    for (int i = 0; i < n; i++) p[i] = -p[i];
78    return;
79  }
80
81  int * sort_heap(int n, double a[][2]) {
82    double aval[2];
83    int i, ir, j, l, idxt;
84    int *idx;
85    if (n < 1) return NULL;
86    if (n == 1) {
87      idx = new int[1];
88      idx[0] = 1;
```

```
 89        return idx;
 90      }
 91      idx = new int[n];
 92      for (int i = 0; i < n; i++) idx[i] = i + 1;
 93      l = n / 2 + 1;
 94      ir = n;
 95      for (;;) {
 96        if (1 < l) {
 97          l--;
 98          idxt = idx[l - 1];
 99          aval[0] = a[idxt - 1][0];
100          aval[1] = a[idxt - 1][1];
101        } else {
102          idxt = idx[ir - 1];
103          aval[0] = a[idxt - 1][0];
104          aval[1] = a[idxt - 1][1];
105          idx[ir - 1] = idx[0];
106          if (--ir == 1) {
107            idx[0] = idxt;
108            break;
109          }
110        }
111        i = l;
112        j = 2 * l;
113        while (j <= ir) {
114          if (j < ir && (a[idx[j - 1] - 1][0] <  a[idx[j] - 1][0] ||
115                         (a[idx[j - 1] - 1][0] == a[idx[j] - 1][0] &&
116                          a[idx[j - 1] - 1][1] <  a[idx[j] - 1][1]))) {
117            j++;
118          }
119          if ( aval[0]  < a[idx[j - 1] - 1][0] ||
120              (aval[0] == a[idx[j - 1] - 1][0] &&
121               aval[1]  < a[idx[j - 1] - 1][1])) {
122            idx[i - 1] = idx[j - 1];
123            i = j;
124            j *= 2;
125          } else {
126            j = ir + 1;
127          }
128        }
129        idx[i - 1] = idxt;
130      }
131      return idx;
132    }
133
134    int lrline(double xu, double yu, double xv1, double yv1,
135               double xv2, double yv2, double dv) {
136      double tol = 1e-7;
137      double dx = xv2 - xv1, dy = yv2 - yv1;
138      double dxu = xu - xv1, dyu = yu - yv1;
139      double t = dy * dxu - dx * dyu + dv * sqrt(dx * dx + dy * dy);
140      double tolabs = tol * std::max(std::max(fabs(dx), fabs(dy)),
141                            std::max(fabs(dxu), std::max(fabs(dyu), fabs(dv))));
142      if (tolabs < t) return 1;
143      if (-tolabs <= t) return 0;
144      return -1;
145    }
146
147    void vbedg(double x, double y, int point_num, double point_xy[][2],
```

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

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

```
266        f = 2;
267        c = tri_nodes[u - 1][0];
268      } else {
269        f = 3;
270        c = tri_nodes[u - 1][1];
271      }
272      swap = diaedg(x, y, point_xy[a - 1][0], point_xy[a - 1][1],
273                          point_xy[c - 1][0], point_xy[c - 1][1],
274                          point_xy[b - 1][0], point_xy[b - 1][1]);
275      if (swap == 1) {
276        em1 = wrap(e - 1, 1, 3);
277        ep1 = wrap(e + 1, 1, 3);
278        fm1 = wrap(f - 1, 1, 3);
279        fp1 = wrap(f + 1, 1, 3);
280        tri_nodes[t - 1][ep1 - 1] = c;
281        tri_nodes[u - 1][fp1 - 1] = i;
282        r = tri_neigh[t - 1][ep1 - 1];
283        s = tri_neigh[u - 1][fp1 - 1];
284        tri_neigh[t - 1][ep1 - 1] = u;
285        tri_neigh[u - 1][fp1 - 1] = t;
286        tri_neigh[t - 1][e - 1] = s;
287        tri_neigh[u - 1][f - 1] = r;
288        if (0 < tri_neigh[u - 1][fm1 - 1]) {
289          *top = *top + 1;
290          stack[*top - 1] = u;
291        }
292        if (0 < s) {
293          if (tri_neigh[s - 1][0] == u) {
294            tri_neigh[s - 1][0] = t;
295          } else if (tri_neigh[s - 1][1] == u) {
296            tri_neigh[s - 1][1] = t;
297          } else {
298            tri_neigh[s - 1][2] = t;
299          }
300          *top = *top + 1;
301          if (point_num < *top) return 8;
302          stack[*top - 1] = t;
303        } else {
304          if (u == *btri && fp1 == *bedg) {
305            *btri = t;
306            *bedg = e;
307          }
308          l = - (3 * t + e - 1);
309          tt = t;
310          ee = em1;
311          while (0 < tri_neigh[tt - 1][ee - 1]) {
312            tt = tri_neigh[tt - 1][ee - 1];
313            if (tri_nodes[tt - 1][0] == a) {
314              ee = 3;
315            } else if (tri_nodes[tt - 1][1] == a) {
316              ee = 1;
317            } else {
318              ee = 2;
319            }
320          }
321          tri_neigh[tt - 1][ee - 1] = l;
322        }
323        if (0 < r) {
324          if (tri_neigh[r - 1][0] == t) {
```

```
325              tri_neigh[r - 1][0] = u;
326            } else if (tri_neigh[r - 1][1] == t) {
327              tri_neigh[r - 1][1] = u;
328            } else {
329              tri_neigh[r - 1][2] = u;
330            }
331          } else {
332            if (t == *btri && ep1 == *bedg) {
333              *btri = u;
334              *bedg = f;
335            }
336            l = -(3 * u + f - 1);
337            tt = u;
338            ee = fm1;
339            while (0 < tri_neigh[tt - 1][ee - 1]) {
340              tt = tri_neigh[tt - 1][ee - 1];
341              if (tri_nodes[tt - 1][0] == b) {
342                ee = 3;
343              } else if (tri_nodes[tt - 1][1] == b) {
344                ee = 1;
345              } else {
346                ee = 2;
347              }
348            }
349            tri_neigh[tt - 1][ee - 1] = l;
350          }
351        }
352      }
353      return 0;
354    }
355
356    void perm_inv(int n, int p[]) {
357      int i, i0, i1, i2;
358      assert(n > 0);
359      for (i = 1; i <= n; i++) {
360        i1 = p[i - 1];
361        while (i < i1) {
362          i2 = p[i1 - 1];
363          p[i1 - 1] = -i2;
364          i1 = i2;
365        }
366        p[i - 1] = -p[i - 1];
367      }
368      for (i = 1; i <= n; i++) {
369        i1 = -p[i - 1];
370        if (0 <= i1) {
371          i0 = i;
372          for (;;) {
373            i2 = p[i1 - 1];
374            p[i1 - 1] = i0;
375            if (i2 < 0) break;
376            i0 = i1;
377            i1 = i2;
378          }
379        }
380      }
381      return;
382    }
383
```

```
384  int dtris2(int point_num, double point_xy[][2],
385              int tri_nodes[][3], int tri_neigh[][3]) {
386    double cmax;
387    int e, error;
388    int i, j, k, l, m, m1, m2, n;
389    int ledg, lr, ltri, redg, rtri, t, top;
390    double tol;
391    int *stack = new int[point_num];
392    tol = 100.0 * epsilon();
393    int *idx = sort_heap(point_num, point_xy);
394    permute(point_num, point_xy, idx);
395    m1 = 0;
396    for (i = 1; i < point_num; i++) {
397      m = m1;
398      m1 = i;
399      k = -1;
400      for (j = 0; j <= 1; j++) {
401        cmax = std::max(fabs(point_xy[m][j]), fabs(point_xy[m1][j]));
402        if (tol * (cmax + 1.0) < fabs(point_xy[m][j] - point_xy[m1][j])) {
403          k = j;
404          break;
405        }
406      }
407      assert(k != -1);
408    }
409    m1 = 1;
410    m2 = 2;
411    j = 3;
412    for (;;) {
413      assert(point_num >= j);
414      m = j;
415      lr = lrline(point_xy[m - 1][0], point_xy[m - 1][1],
416                  point_xy[m1 - 1][0], point_xy[m1 - 1][1],
417                  point_xy[m2 - 1][0], point_xy[m2 - 1][1], 0.0);
418      if (lr != 0) break;
419      j++;
420    }
421    int tri_num = j - 2;
422    if (lr == -1) {
423      tri_nodes[0][0] = m1;
424      tri_nodes[0][1] = m2;
425      tri_nodes[0][2] = m;
426      tri_neigh[0][2] = -3;
427      for (i = 2; i <= tri_num; i++) {
428        m1 = m2;
429        m2 = i + 1;
430        tri_nodes[i - 1][0] = m1;
431        tri_nodes[i - 1][1] = m2;
432        tri_nodes[i - 1][2] = m;
433        tri_neigh[i - 1][0] = -3 * i;
434        tri_neigh[i - 1][1] = i;
435        tri_neigh[i - 1][2] = i - 1;
436      }
437      tri_neigh[tri_num - 1][0] = -3 * tri_num - 1;
438      tri_neigh[tri_num - 1][1] = -5;
439      ledg = 2;
440      ltri = tri_num;
441    } else {
442      tri_nodes[0][0] = m2;
```

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

```
502          tri_neigh[tri_num - 1][0] = t;
503          tri_neigh[tri_num - 1][1] = tri_num - 1;
504          tri_neigh[tri_num - 1][2] = tri_num + 1;
505          top++;
506          assert(point_num >= top);
507          stack[top - 1] = tri_num;
508          if (t == rtri && e == redg) break;
509        }
510        tri_neigh[ltri - 1][ledg - 1] = -3 * n - 1;
511        tri_neigh[n - 1][1] = -3 * tri_num - 2;
512        tri_neigh[tri_num - 1][2] = -l;
513        ltri = n;
514        ledg = 2;
515        error = swapec(m, &top, &ltri, &ledg, point_num, point_xy,
516                       tri_num, tri_nodes, tri_neigh, stack);
517        assert(error == 0);
518      }
519      for (i = 0; i < 3; i++)
520        for (j = 0; j < tri_num; j++)
521          tri_nodes[j][i] = idx[tri_nodes[j][i] - 1];
522      perm_inv(point_num, idx);
523      permute(point_num, point_xy, idx);
524      delete[] idx;
525      delete[] stack;
526      return tri_num;
527  }
528
529  /*** C++ Wrapper ***/
530
531  typedef std::pair<double, double> point;
532  #define x first
533  #define y second
534
535  struct triangle { point a, b, c; };
536
537  template<class It>
538  std::vector<triangle> delaunay_triangulation(It lo, It hi) {
539    int n = hi - lo;
540    double points[n][2];
541    int tri_nodes[3 * n][3], tri_neigh[3 * n][3];
542    int curr = 0;
543    for (It it = lo; it != hi; ++curr, ++it) {
544      points[curr][0] = it->x;
545      points[curr][1] = it->y;
546    }
547    int m = dtris2(n, points, tri_nodes, tri_neigh);
548    std::vector<triangle> res;
549    for (int i = 0; i < m; i++)
550      res.push_back((triangle){*(lo + (tri_nodes[i][0] - 1)),
551                               *(lo + (tri_nodes[i][1] - 1)),
552                               *(lo + (tri_nodes[i][2] - 1))});
553    return res;
554  }
555
556  /*** Example Usage ***/
557
558  #include <iostream>
559  using namespace std;
560
```

```
561  int main() {
562    vector<point> v;
563    v.push_back(point(1, 3));
564    v.push_back(point(1, 2));
565    v.push_back(point(2, 1));
566    v.push_back(point(0, 0));
567    v.push_back(point(-1, 3));
568    vector<triangle> dt = delaunay_triangulation(v.begin(), v.end());
569    for (int i = 0; i < (int)dt.size(); i++) {
570      cout << "Triangle:␣";
571      cout << "(" << dt[i].a.x << "," << dt[i].a.y << ")␣";
572      cout << "(" << dt[i].b.x << "," << dt[i].b.y << ")␣";
573      cout << "(" << dt[i].c.x << "," << dt[i].c.y << ")\n";
574    }
575    return 0;
576  }
```

Chapter 6

Strings

6.1 Strings Toolbox

```
1  /*
2
3  6.1 - Strings Toolbox
4
5  Useful or trivial string operations. These functions are not particularly
6  algorithmic. They are typically naive implementations using C++ features.
7  They depend on many features of the C++ <string> library, which tend to
8  have an unspecified complexity. They may not be optimally efficient.
9
10 */
11
12 #include <cstdlib>
13 #include <sstream>
14 #include <string>
15 #include <vector>
16
17 //integer to string conversion and vice versa using C++ features
18
19 //note that a similar std::to_string is introduced in C++0x
20 template<class Int>
21 std::string to_string(const Int & i) {
22   std::ostringstream oss;
23   oss << i;
24   return oss.str();
25 }
26
27 //like atoi, except during special cases like overflows
28 int to_int(const std::string & s) {
29   std::istringstream iss(s);
30   int res;
31   if (!(iss >> res)) /* complain */;
32   return res;
33 }
34
35 /*
36
37 itoa implementation (fast)
```

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

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

```
156
157  Like the explode() function in PHP, the string s is tokenized based
158  on delim, which is considered as a whole boundary string, not just a
159  sequence of possible boundary characters like the split() function above.
160  This will not skip empty tokens.
161    e.g. explode("a::b", ":") yields {"a", "", "b"}, not {"a", "b"}.
162
163  Time Complexity: O(s.length() * delim.length())
164
165  */
166
167  std::vector<std::string> explode(const std::string & s,
168                                   const std::string & delim) {
169    std::vector<std::string> res;
170    unsigned int last = 0, next = 0;
171    while ((next = s.find(delim, last)) != std::string::npos) {
172      res.push_back(s.substr(last, next - last));
173      last = next + delim.size();
174    }
175    res.push_back(s.substr(last));
176    return res;
177  }
178
179  /*** Example Usage ***/
180
181  #include <cassert>
182  #include <cstdio>
183  #include <iostream>
184  using namespace std;
185
186  void print(const vector<string> & v) {
187    cout << "[";
188    for (int i = 0; i < (int)v.size(); i++)
189      cout << (i ? "\",␣\"" : "\"") << v[i];
190    cout << "\"]\n";
191  }
192
193  int main() {
194    assert(to_string(123) + "4" == "1234");
195    assert(to_int("1234") == 1234);
196    char buffer[50];
197    assert(string(itoa(1750, buffer, 10)) == "1750");
198    assert(string(itoa(1750, buffer, 16)) == "6d6");
199    assert(string(itoa(1750, buffer, 2)) == "11011010110");
200
201    string s("␣␣␣abc␣\n");
202    string t = s;
203    assert(ltrim(s) == "abc␣\n");
204    assert(rtrim(s) == trim(t));
205    assert(replace("abcdabba", "ab", "00") == "00cd00ba");
206
207    vector<string> tokens;
208
209    tokens = split("a\nb\ncde\nf", '\n');
210    cout << "split␣v1:␣";
211    print(tokens); //["a", "b", "cde", "f"]
212
213    tokens = split("a::b,cde:,f", ":,");
214    cout << "split␣v2:␣";
```

```
215    print(tokens); //["a", "b", "cde", "f"]
216
217    tokens = explode("a..b.cde....f", "..");
218    cout << "explode:␣";
219    print(tokens); //["a", ".b.cde", "", ".f"]
220    return 0;
221  }
```

6.2   Expression Parsing

6.2.1   Recursive Descent

```
1   /*
2
3   6.2.1 Recursive Descent Parser
4
5   Evaluate a mathematica expression in accordance to the order
6   of operations (parentheses, exponents, multiplication, division,
7   addition, subtraction). Does not handle unary operators like '-'.
8
9   */
10
11  /*** Example Usage ***/
12
13  #include <cctype>
14  #include <cmath>
15  #include <sstream>
16  #include <stdexcept>
17  #include <string>
18
19  class parser {
20    int pos;
21    double tokval;
22    std::string s;
23
24    bool is_dig_or_dot(char c) {
25      return isdigit(c) || c == '.';
26    }
27
28    double to_double(const std::string & s) {
29      std::stringstream ss(s);
30      double res;
31      ss >> res;
32      return res;
33    }
34
35  public:
36    char token;
37
38    parser(const std::string & s) {
39      this->s = s;
40      pos = 0;
41    }
42
43    int next() {
44      for (;;) {
```

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

```
103        }
104      }
105
106      // expression ::= term | expression '+' term | expression '-' term
107      double expression() {
108        double v = term();
109        for (;;) {
110          if (token == '+') {
111            skip('+');
112            v += term();
113          } else if (token == '-') {
114            skip('-');
115            v -= term();
116          } else {
117            return v;
118          }
119        }
120      }
121    };
122
123    #include <iostream>
124    using namespace std;
125
126    int main() {
127      parser p("1+2*3*4+3*(2+2)-100\n");
128      p.next();
129      while (p.token != -1) {
130        if (p.token == '\n') {
131          p.skip('\n');
132          continue;
133        }
134        cout << p.expression() << "\n";
135      }
136      return 0;
137    }
```

6.2.2 Recursive Descent (Simple)

```
 1   /*
 2
 3   6.2.2 Recursive Descent Parser (Simple)
 4
 5   Evaluate a mathematica expression in accordance to the order
 6   of operations (parentheses, exponents, multiplication, division,
 7   addition, subtraction). This handles unary operators like '-'.
 8
 9   */
10
11   #include <string>
12
13   template<class It> int eval(It & it, int prec) {
14     if (prec == 0) {
15       int sign = 1, ret = 0;
16       for (; *it == '-'; it++) sign *= -1;
17       if (*it == '(') {
18         ret = eval(++it, 2);
19         it++;
```

```
20      } else while (*it >= '0' && *it <= '9') {
21        ret = 10 * ret + (*(it++) - '0');
22      }
23      return sign * ret;
24    }
25    int num = eval(it, prec - 1);
26    while (!((prec == 2 && *it != '+' && *it != '-') ||
27            (prec == 1 && *it != '*' && *it != '/'))) {
28      switch (*(it++)) {
29        case '+': num += eval(it, prec - 1); break;
30        case '-': num -= eval(it, prec - 1); break;
31        case '*': num *= eval(it, prec - 1); break;
32        case '/': num /= eval(it, prec - 1); break;
33      }
34    }
35    return num;
36  }
37
38  /*** Wrapper Function ***/
39
40  int eval(const std::string & s) {
41    std::string::iterator it = std::string(s).begin();
42    return eval(it, 2);
43  }
44
45  /*** Example Usage ***/
46
47  #include <iostream>
48  using namespace std;
49
50  int main() {
51    cout << eval("1+2*3*4+3*(2+2)-100") << "\n";
52    return 0;
53  }
```

6.2.3   Shunting Yard Algorithm

```
1   /*
2
3   6.2.3 - Shunting Yard Expression Parser
4
5   Evaluate a mathematical expression in accordance to the order
6   of operations (parentheses, exponents, multiplication, division,
7   addition, subtraction). This also handles unary operators like '-'.
8   We use strings for operators so we can even define things like "sqrt"
9   and "mod" as unary operators by changing prec() and split_expr()
10  accordingly.
11
12  Time Complexity: O(n) on the total number of operators and operands.
13
14  */
15
16  #include <cstdlib>   /* strtol() */
17  #include <stack>
18  #include <stdexcept> /* std::runtime_error */
19  #include <string>
20  #include <vector>
```

```
21
22   // Classify the precedences of operators here.
23   inline int prec(const std::string & op, bool unary) {
24     if (unary) {
25       if (op == "+" || op == "-") return 3;
26       return 0; // not a unary operator
27     }
28     if (op == "*" || op == "/") return 2;
29     if (op == "+" || op == "-") return 1;
30     return 0;  // not a binary operator
31   }
32
33   inline int calc1(const std::string & op, int val) {
34     if (op == "+") return +val;
35     if (op == "-") return -val;
36     throw std::runtime_error("Invalid unary operator: " + op);
37   }
38
39   inline int calc2(const std::string & op, int L, int R) {
40     if (op == "+") return L + R;
41     if (op == "-") return L - R;
42     if (op == "*") return L * R;
43     if (op == "/") return L / R;
44     throw std::runtime_error("Invalid binary operator: " + op);
45   }
46
47   inline bool is_operand(const std::string & s) {
48     return s != "(" && s != ")" && !prec(s, 0) && !prec(s, 1);
49   }
50
51   int eval(std::vector<std::string> E) { // E stores the tokens
52     E.insert(E.begin(), "(");
53     E.push_back(")");
54     std::stack<std::pair<std::string, bool> > ops;
55     std::stack<int> vals;
56     for (int i = 0; i < (int)E.size(); i++) {
57       if (is_operand(E[i])) {
58         vals.push(strtol(E[i].c_str(), 0, 10)); // convert to int
59         continue;
60       }
61       if (E[i] == "(") {
62         ops.push(std::make_pair("(", 0));
63         continue;
64       }
65       if (prec(E[i], 1) && (i == 0 || E[i - 1] == "(" || prec(E[i - 1], 0))) {
66         ops.push(std::make_pair(E[i], 1));
67         continue;
68       }
69       while (prec(ops.top().first, ops.top().second) >= prec(E[i], 0)) {
70         std::string op = ops.top().first;
71         bool is_unary = ops.top().second;
72         ops.pop();
73         if (op == "(") break;
74         int y = vals.top(); vals.pop();
75         if (is_unary) {
76           vals.push(calc1(op, y));
77         } else {
78           int x = vals.top(); vals.pop();
79           vals.push(calc2(op, x, y));
```

```
80          }
81        }
82        if (E[i] != ")") ops.push(std::make_pair(E[i], 0));
83      }
84      return vals.top();
85    }
86
87    /*
88
89    Split a string expression to tokens, ignoring whitespace delimiters.
90    A vector of tokens is a more flexible format since you can decide to
91    parse the expression however you wish just by modifying this function.
92    e.g. "1+(51 * -100)" converts to {"1","+","(","51","*","-","100",")"}
93
94    */
95
96    std::vector<std::string> split_expr(const std::string &s,
97                      const std::string &delim = "␣\n\t\v\f\r") {
98      std::vector<std::string> ret;
99      std::string acc = "";
100     for (int i = 0; i < (int)s.size(); i++)
101       if (s[i] >= '0' && s[i] <= '9') {
102         acc += s[i];
103       } else {
104         if (i > 0 && s[i - 1] >= '0' && s[i - 1] <= '9')
105           ret.push_back(acc);
106         acc = "";
107         if (delim.find(s[i]) != std::string::npos) continue;
108         ret.push_back(std::string("") + s[i]);
109       }
110     if (s[s.size() - 1] >= '0' && s[s.size() - 1] <= '9')
111       ret.push_back(acc);
112     return ret;
113   }
114
115   int eval(const std::string & s) {
116     return eval(split_expr(s));
117   }
118
119   /*** Example Usage ***/
120
121   #include <iostream>
122   using namespace std;
123
124   int main() {
125     cout << eval("1+2*3*4+3*(2+2)-100") << endl;
126     return 0;
127   }
```

6.3   String Searching

6.3.1   Knuth-Morris-Pratt Algorithm

```
1    /*
2
3    6.3.1 - String Searching (Knuth-Morris-Pratt)
```

```
 4
 5   Given a text and a pattern to be searched for within the text,
 6   determine the first position in which the pattern occurs in
 7   the text. The KMP algorithm is much faster than the naive,
 8   quadratic time, string searching algorithm that is found in
 9   string.find() in the C++ standard library.
10
11   KMP generates a table using a prefix function of the pattern.
12   Then, the precomputed table of the pattern can be used indefinitely
13   for any number of texts.
14
15   Time Complexity: O(n + m) where n is the length of the text
16   and m is the length of the pattern.
17
18   Space Complexity: O(m) auxiliary on the length of the pattern.
19
20   */
21
22   #include <string>
23   #include <vector>
24
25   int find(const std::string & text, const std::string & pattern) {
26     if (pattern.empty()) return 0;
27     //generate table using pattern
28     std::vector<int> p(pattern.size());
29     for (int i = 0, j = p[0] = -1; i < (int)pattern.size(); ) {
30       while (j >= 0 && pattern[i] != pattern[j])
31         j = p[j];
32       i++;
33       j++;
34       p[i] = (pattern[i] == pattern[j]) ? p[j] : j;
35     }
36     //use the precomputed table to search within text
37     //the following can be repeated on many different texts
38     for (int i = 0, j = 0; j < (int)text.size(); ) {
39       while (i >= 0 && pattern[i] != text[j])
40         i = p[i];
41       i++;
42       j++;
43       if (i >= (int)pattern.size())
44         return j - i;
45     }
46     return std::string::npos;
47   }
48
49   /*** Example Usage ***/
50
51   #include <cassert>
52
53   int main() {
54     assert(15 == find("ABC␣ABCDAB␣ABCDABCDABDE", "ABCDABD"));
55     return 0;
56   }
```

6.3.2   Aho Corasick Algorithm

6.3.3   Aho Corasick Algorithm (Simple)

6.3.4   Z-Function

6.4   Dynamic Programming

6.4.1   Longest Common Substring

6.4.2   Longest Common Subsequence

6.4.3   Edit Distance

6.5   Suffix Algorithms

6.5.1   Suffix and LCP Array

6.5.2   Simple Trie

6.5.3   Radix Trie

6.5.4   Suffix Trie