Alex's Anthology of Algorithms

Common Code for Contests in Concise C++

(Draft, December 2015)

Compiled, Edited, and Written by

Alex Li

University of Toronto

Email: alex@alexli.ca

This page is intentionally left blank.

Preface

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

0.1 Introduction

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

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

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

0.2 Portability

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

That means the following assumptions are made:

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

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

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

0.3 Usage Notes

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

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

Cheers and hope you enjoy!

— Alex Li

December, 2015

Contents

Chapter 1

Elementary Algorithms

1.1 Array Transformations

1.1.1 Sorting Algorithms

```
/*

The following functions are to be used like std::sort(), taking two
RandomAccessIterators as the range to be sorted, and optionally a
comparison function object to replace the default < operator.

They are not intended to compete with the standard library sorting
functions in terms of speed, but are merely demonstrations of how to
implement common sorting algorithms concisely in C++.

*/

#include <algorithm>  /* std::copy(), std::swap() */
#include <functional> /* std::less */
#include <iterator>   /* std::iterator_traits */

/*

Quicksort

Quicksort repeatedly selects a pivot and "partitions" the range so that
all values comparing less than the pivot come before it, and all values
comparing greater comes after it. Divide and conquer is then applied to
both sides of the pivot until the original range is sorted. Despite
having a worst case of O(n^2), quicksort is faster in practice than
merge sort and heapsort, which each has a worst case of O(n log n).

The pivot chosen in this implementation is always the middle element
of the range to be sorted. To reduce the likelihood of encountering the
worst case, the algorithm should be modified to select a random pivot,
or use the "median of three" method.

Time Complexity (Average): O(n log n)
Time Complexity (Worst): O(n^2)
Space Complexity: O(log n) auxiliary.
```

```
Stable?: No

*/

template<class It, class Compare>
void quicksort(It lo, It hi, Compare comp) {
  if (hi - lo < 2) return;
  typedef typename std::iterator_traits<It>::value_type T;
  T pivot = *(lo + (hi - lo) / 2);
  It i, j;
  for (i = lo, j = hi - 1; ; ++i, --j) {
    while (comp(*i, pivot))
      ++i;
    while (comp(pivot, *j))
      --j;
    if (i >= j)
      break;
    std::swap(*i, *j);
  }
  quicksort(lo, i, comp);
  quicksort(i, hi, comp);
}

template<class It> void quicksort(It lo, It hi) {
  typedef typename std::iterator_traits<It>::value_type T;
  quicksort(lo, hi, std::less<T>());
}

/*

Merge Sort

Merge sort works by first dividing a list into n sublists, each with
one element, then recursively merging sublists to produce new sorted
sublists until only a single sorted sublist remains. Merge sort has a
better worse case than quicksort, and is also stable, meaning that it
will preserve the relative ordering of elements considered equal by
the < operator or comparator (a < b and b < a both return false).

While std::stable_sort() is a corresponding function in the standard
library, the implementation below differs in that it will simply fail
if extra memory is not available. Meanwhile, std::stable_sort() will
not fail, but instead fall back to a time complexity of O(n log^2 n).

Time Complexity (Average): O(n log n)
Time Complexity (Worst): O(n log n)
Space Complexity: O(n) auxiliary.
Stable?: Yes

*/

template<class It, class Compare>
void mergesort(It lo, It hi, Compare comp) {
  if (hi - lo < 2) return;
  It mid = lo + (hi - lo - 1) / 2, a = lo, c = mid + 1;
  mergesort(lo, mid + 1, comp);
  mergesort(mid + 1, hi, comp);
  typedef typename std::iterator_traits<It>::value_type T;
  T *buf = new T[hi - lo], *b = buf;
  while (a <= mid && c < hi)
    *(b++) = comp(*c, *a) ? *(c++) : *(a++);
  if (a > mid) {
    for (It k = c; k < hi; k++)
      *(b++) = *k;
  } else {
    for (It k = a; k <= mid; k++)
      *(b++) = *k;
  }
  for (int i = hi - lo - 1; i >= 0; i--)
    *(lo + i) = buf[i];
  delete[] buf;
}

template<class It> void mergesort(It lo, It hi) {
  typedef typename std::iterator_traits<It>::value_type T;
  mergesort(lo, hi, std::less<T>());
}

/*

Heapsort

Heapsort first rearranges an array to satisfy the heap property, and
then the max element of the heap is repeated removed and added to the
end of the resulting sorted list. A heapified array has the root node
at index 0. The two children of the node at index n are respectively
located at indices 2n + 1 and 2n + 2. Each node is greater than both
of its children. This leads to a structure that takes O(log n) to
insert any element or remove the max element. Heapsort has a better
worst case complexity than quicksort, but a better space complexity
complexity than merge sort.

The standard library equivalent is calling std::make_heap(), followed
by std::sort_heap() on the input range.

Time Complexity (Average): O(n log n)
Time Complexity (Worst): O(n log n)
Space Complexity: O(1) auxiliary.
Stable?: No

*/

template<class It, class Compare>
void heapsort(It lo, It hi, Compare comp) {
  typename std::iterator_traits<It>::value_type t;
  It i = lo + (hi - lo) / 2, j = hi, parent, child;
  for (;;) {
    if (i <= lo) {
      if (--j == lo)
        return;
      t = *j;
      *j = *lo;
    } else {
      t = *(--i);
    }
    parent = i;
    child = lo + 2 * (i - lo) + 1;
    while (child < j) {
      if (child + 1 < j && comp(*child, *(child + 1)))
        child++;
      if (!comp(t, *child))
        break;
      *parent = *child;
      parent = child;
      child = lo + 2 * (parent - lo) + 1;
    }
    *(lo + (parent - lo)) = t;
  }
}

template<class It> void heapsort(It lo, It hi) {
  typedef typename std::iterator_traits<It>::value_type T;
  heapsort(lo, hi, std::less<T>());
}

/*

Comb Sort

Comb sort is an improved bubble sort. While bubble sort increases the
gap between swapped elements for every inner loop iteration, comb sort
uses a fixed gap for the inner loop and decreases the gap size by a
shrink factor for every iteration of the outer loop.

Even though the average time complexity is theoretically O(n^2), if the
increments (gap sizes) are relatively prime and the shrink factor is
sensible (1.3 is empirically determined to be the best), then it will
require astronomically large n to make the algorithm exceed O(n log n)
steps. In practice, comb sort is only 2-3 times slower than merge sort.

Time Complexity (Average): O(n^2 / 2^p) for p increments.
Time Complexity (Worst): O(n^2)
Space Complexity: O(1) auxiliary.
Stable?: No

*/

template<class It, class Compare>
void combsort(It lo, It hi, Compare comp) {
  int gap = hi - lo;
  bool swapped = true;
  while (gap > 1 || swapped) {
    if (gap > 1)
      gap = (int)((float)gap / 1.3f);
    swapped = false;
    for (It i = lo; i + gap < hi; i++)
      if (comp(*(i + gap), *i)) {
        std::swap(*i, *(i + gap));
        swapped = true;
      }
  }
}

template<class It> void combsort(It lo, It hi) {
  typedef typename std::iterator_traits<It>::value_type T;
  combsort(lo, hi, std::less<T>());
}
```

```

/*

Radix Sort

Radix sort can be used to sort integer keys with a constant number of
bits in linear time. The keys are grouped by the individual digits of
a particular base which share the same significant position and value.

The implementation below only works on ranges pointing to unsigned
integer primitives (but can be modified to also work on signed values).
Note that the input range need not strictly be "unsigned" types, as
long as the values are all technically non-negative. A power of two is
chosen to be the base of the sort since bitwise operations may be used
to extract digits (instead of modulos and powers, which are much less
efficient). In practice, it's been demonstrated that 2^8 is the best
choice for sorting 32-bit integers (roughly 5 times faster than using
std::sort and 2 to 4 times faster than any other chosen power of two).

This implementation was adapted from: http://qr.ae/RbdDTa
Explanation of base 2^8 choice: http://qr.ae/RbdDcG

Time Complexity: O(n * w) for n integers of w bits.
Space Complexity: O(n + w) auxiliary.
Stable?: Yes

*/

template<class UnsignedIt>
void radix_sort(UnsignedIt lo, UnsignedIt hi) {
  if (hi - lo < 2)
    return;
  const int radix_bits = 8;
  const int radix_base = 1 << radix_bits; //e.g. 2^8 = 256
  const int radix_mask = radix_base - 1;  //e.g. 2^8 - 1 = 0xFF
  int num_bits = 8 * sizeof(*lo); //8 bits per byte
  typedef typename std::iterator_traits<UnsignedIt>::value_type T;
  T *l = new T[hi - lo];
  for (int pos = 0; pos < num_bits; pos += radix_bits) {
    int count[radix_base] = {0};
    for (UnsignedIt it = lo; it != hi; it++)
      count[(*it >> pos) & radix_mask]++;
    T *bucket[radix_base], *curr = l;
    for (int i = 0; i < radix_base; curr += count[i++])
      bucket[i] = curr;
    for (UnsignedIt it = lo; it != hi; it++)
      *bucket[(*it >> pos) & radix_mask]++ = *it;
    std::copy(l, l + (hi - lo), lo);
  }
  delete[] l;
}

/*** Example Usage

Sample Output:

mergesort() with default comparisons: 1.32 1.41 1.62 1.73 2.58 2.72 3.14 4.67
mergesort() with 'compare_as_ints()': 1.41 1.73 1.32 1.62 2.72 2.58 3.14 4.67
------
```

```
Sorting five million integers...
std::sort():  0.429s
quicksort():  0.498s
mergesort():  1.437s
heapsort():   1.179s
combsort():   1.023s
radix_sort(): 0.078s

***/

#include <cassert>
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <iostream>
#include <vector>
using namespace std;

template<class It> void print_range(It lo, It hi) {
  while (lo != hi)
    cout << *(lo++) << "␣";
  cout << endl;
}

template<class It> bool is_sorted(It lo, It hi) {
  while (++lo != hi)
    if (*(lo - 1) > *lo)
      return false;
  return true;
}

bool compare_as_ints(double i, double j) {
  return (int)i < (int)j;
}

int main () {
  { //can be used to sort arrays like std::sort()
    int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
    quicksort(a, a + 8);
    assert(is_sorted(a, a + 8));
  }
  { //STL containers work too
    int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
    vector<int> v(a, a + 8);
    quicksort(v.begin(), v.end());
    assert(is_sorted(v.begin(), v.end()));
  }
  { //reverse iterators work as expected
    int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
    vector<int> v(a, a + 8);
    heapsort(v.rbegin(), v.rend());
    assert(is_sorted(v.rbegin(), v.rend()));
  }
  { //doubles are also fine
    double a[] = {1.1, -5.0, 6.23, 4.123, 155.2};
    vector<double> v(a, a + 5);
    combsort(v.begin(), v.end());
    assert(is_sorted(v.begin(), v.end()));
  }
  { //only unsigned ints work for radix_sort (but reverse works!)
    int a[] = {32, 71, 12, 45, 26, 80, 53, 33};
    vector<int> v(a, a + 8);
    radix_sort(v.rbegin(), v.rend());
    assert(is_sorted(v.rbegin(), v.rend()));
  }

  //example from http://www.cplusplus.com/reference/algorithm/stable_sort
  double a[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};
  {
    vector<double> v(a, a + 8);
    cout << "mergesort()␣with␣default␣comparisons:␣";
    mergesort(v.begin(), v.end());
    print_range(v.begin(), v.end());
  }
  {
    vector<double> v(a, a + 8);
    cout << "mergesort()␣with␣'compare_as_ints()':␣";
    mergesort(v.begin(), v.end(), compare_as_ints);
    print_range(v.begin(), v.end());
  }
  cout << "------" << endl;

  vector<int> v, v2;
  for (int i = 0; i < 5000000; i++)
    v.push_back((rand() & 0x7fff) | ((rand() & 0x7fff) << 15));
  v2 = v;
  cout << "Sorting␣five␣million␣integers..." << endl;
  cout.precision(3);

#define test(sortfunc) {                                  \
  clock_t start = clock();                                \
  sortfunc(v.begin(), v.end());                           \
  double t = (double)(clock() - start) / CLOCKS_PER_SEC; \
  cout << setw(14) << left << #sortfunc "():␣";           \
  cout << fixed << t << "s" << endl;                      \
  assert(is_sorted(v.begin(), v.end()));                  \
  v = v2;                                                 \
}
  test(std::sort);
  test(quicksort);
  test(mergesort);
  test(heapsort);
  test(combsort);
  test(radix_sort);

  return 0;
}
```

1.1.2 Array Rotation

```
/*

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

```

All three versions achieve the same result using no temporary arrays.
Version 1 uses a straightforward swapping algorithm listed on many C++
reference sites, requiring only forward iterators. Version 2 requires
bidirectional iterators, employing the well-known technique of three
simple reversals. Version 3 applies a "juggling" algorithm which first
divides the range into gcd(n, k) sets (n = hi - lo and k = mid - lo)
and then rotates the corresponding elements in each set. This version
requires random access iterators.

Time Complexity: O(n) on the distance between lo and hi.
Space Complexity: O(1) auxiliary.

*/

#include <algorithm> /* std::reverse(), std::rotate(), std::swap() */

template<class It> void rotate1(It lo, It mid, It hi) {
  It next = mid;
  while (lo != next) {
    std::swap(*lo++, *next++);
    if (next == hi)
      next = mid;
    else if (lo == mid)
      mid = next;
  }
}

template<class It> void rotate2(It lo, It mid, It hi) {
  std::reverse(lo, mid);
  std::reverse(mid, hi);
  std::reverse(lo, hi);
}

int gcd(int a, int b) {
  return b == 0 ? a : gcd(b, a % b);
}

template<class It> void rotate3(It lo, It mid, It hi) {
  int n = hi - lo, jump = mid - lo;
  int g = gcd(jump, n), cycle = n / g;
  for (int i = 0; i < g; i++) {
    int curr = i, next;
    for (int j = 0; j < cycle - 1; j++) {
      next = curr + jump;
      if (next >= n)
        next -= n;
      std::swap(*(lo + curr), *(lo + next));
      curr = next;
    }
  }
}

/*** Example Usage

Sample Output:

before sort:  2 4 2 0 5 10 7 3 7 1
after sort:   0 1 2 2 3 4 5 7 7 10
```

```
rotate left:  1 2 2 3 4 5 7 7 10 0
rotate right: 0 1 2 2 3 4 5 7 7 10

***/

#include <algorithm>
#include <cassert>
#include <iostream>
#include <vector>
using namespace std;

int main() {
  std::vector<int> v0, v1, v2, v3;
  for (int i = 0; i < 10000; i++)
    v0.push_back(i);
  v1 = v2 = v3 = v0;
  int mid = 5678;
  std::rotate(v0.begin(), v0.begin() + mid, v0.end());
  rotate1(v1.begin(), v1.begin() + mid, v1.end());
  rotate2(v2.begin(), v2.begin() + mid, v2.end());
  rotate3(v3.begin(), v3.begin() + mid, v3.end());
  assert(v0 == v1 && v0 == v2 && v0 == v3);

  //example from: http://en.cppreference.com/w/cpp/algorithm/rotate
  int a[] = {2, 4, 2, 0, 5, 10, 7, 3, 7, 1};
  vector<int> v(a, a + 10);
  cout << "before␣sort:␣␣";
  for (int i = 0; i < (int)v.size(); i++)
    cout << v[i] << ’␣’;
  cout << endl;

  //insertion sort
  for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)
    rotate1(std::upper_bound(v.begin(), i, *i), i, i + 1);
  cout << "after␣sort:␣␣␣";
  for (int i = 0; i < (int)v.size(); i++)
    cout << v[i] << ’␣’;
  cout << endl;

  //simple rotation to the left
  rotate2(v.begin(), v.begin() + 1, v.end());
  cout << "rotate␣left:␣␣";
  for (int i = 0; i < (int)v.size(); i++)
    cout << v[i] << ’␣’;
  cout << endl;

  //simple rotation to the right
  rotate3(v.rbegin(), v.rbegin() + 1, v.rend());
  cout << "rotate␣right:␣";
  for (int i = 0; i < (int)v.size(); i++)
    cout << v[i] << ’␣’;
  cout << endl;
  return 0;
}
```

1.1.3 Counting Inversions

```
/*

The number of inversions in an array a[] is the number of ordered pairs
(i, j) such that i < j and a[i] > a[j]. This is roughly how "close" an
array is to being sorted, but is *not* the same as the minimum number
of swaps required to sort the array. If the array is sorted then the
inversion count is 0. If the array is sorted in decreasing order, then
the inversion count is maximal. The following are two methods of
efficiently counting the number of inversions.

*/

#include <algorithm> /* std::fill(), std::max() */
#include <iterator>  /* std::iterator_traits */

/*

Version 1: Merge sort

The input range [lo, hi) will become sorted after the function call,
and then the number of inversions will be returned. The iterator’s
value type must have the less than < operator defined appropriately.

Explanation: http://www.geeksforgeeks.org/counting-inversions

Time Complexity: O(n log n) on the distance between lo and hi.
Space Complexty: O(n) auxiliary.

*/

template<class It> long long inversions(It lo, It hi) {
  if (hi - lo < 2) return 0;
  It mid = lo + (hi - lo - 1) / 2, a = lo, c = mid + 1;
  long long res = 0;
  res += inversions(lo, mid + 1);
  res += inversions(mid + 1, hi);
  typedef typename std::iterator_traits<It>::value_type T;
  T *buf = new T[hi - lo], *ptr = buf;
  while (a <= mid && c < hi) {
    if (*c < *a) {
      *(ptr++) = *(c++);
      res += (mid - a) + 1;
    } else {
      *(ptr++) = *(a++);
    }
  }
  if (a > mid) {
    for (It k = c; k < hi; k++)
      *(ptr++) = *k;
  } else {
    for (It k = a; k <= mid; k++)
      *(ptr++) = *k;
  }
  for (int i = hi - lo - 1; i >= 0; i--)
    *(lo + i) = buf[i];
  delete[] buf;
  return res;
}
```

```
/*

Version 2: Magic

The following magic is courtesy of misof, and works for any array of
nonnegative integers.

Explanation: http://codeforces.com/blog/entry/17881?#comment-232099

The complexity depends on the magnitude of the maximum value in a[].
Coordinate compression should be applied on the values of a[] so that
they are strictly integers with magnitudes up to n for best results.
Note that after calling the function, a[] will be entirely set to 0.

Time Complexity: O(m log m), where m is maximum value in the array.
Space Complexity: O(m) auxiliary.

*/

long long inversions(int n, int a[]) {
  int mx = 0;
  for (int i = 0; i < n; i++)
    mx = std::max(mx, a[i]);
  int *cnt = new int[mx];
  long long res = 0;
  while (mx > 0) {
    std::fill(cnt, cnt + mx, 0);
    for (int i = 0; i < n; i++) {
      if (a[i] % 2 == 0)
        res += cnt[a[i] / 2];
      else
        cnt[a[i] / 2]++;
    }
    mx = 0;
    for (int i = 0; i < n; i++)
      mx = std::max(mx, a[i] /= 2);
  }
  delete[] cnt;
  return res;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  {
    int a[] = {6, 9, 1, 14, 8, 12, 3, 2};
    assert(inversions(a, a + 8) == 16);
  }
  {
    int a[] = {6, 9, 1, 14, 8, 12, 3, 2};
    assert(inversions(8, a) == 16);
  }
  return 0;
}
```

1.1.4 Coordinate Compression

```cpp
/*

Given an array a[] of size n, reassign integers to each value of a[]
such that the magnitude of each new value is no more than n, while the
relative order of each value as they were in the original array is
preserved. That is, if a[] is the original array and b[] is the result
array, then for every pair (i, j), the result of comparing a[i] < a[j]
will be exactly the same as the result of b[i] < b[j]. Furthermore,
no value of b[] will exceed the *number* of distinct values in a[].

In the following implementations, values in the range [lo, hi) will be
converted to integers in the range [0, d), where d is the number of
distinct values in the original range. lo and hi must be random access
iterators pointing to a numerical type that int can be assigned to.

Time Complexity: O(n log n) on the distance between lo and hi.
Space Complexity: O(n) auxiliary.

*/

#include <algorithm> /* std::lower_bound(), std::sort(), std::unique() */
#include <iterator>  /* std::iterator_traits */
#include <map>

//version 1 - using std::sort(), std::unique() and std::lower_bound()
template<class It> void compress1(It lo, It hi) {
  typedef typename std::iterator_traits<It>::value_type T;
  T *a = new T[hi - lo];
  int n = 0;
  for (It it = lo; it != hi; ++it)
    a[n++] = *it;
  std::sort(a, a + n);
  int n2 = std::unique(a, a + n) - a;
  for (It it = lo; it != hi; ++it)
    *it = (int)(std::lower_bound(a, a + n2, *it) - a);
  delete[] a;
}

//version 2 - using std::map
template<class It> void compress2(It lo, It hi) {
  typedef typename std::iterator_traits<It>::value_type T;
  std::map<T, int> m;
  for (It it = lo; it != hi; ++it)
    m[*it] = 0;
  typename std::map<T, int>::iterator x = m.begin();
  for (int i = 0; x != m.end(); x++)
    x->second = i++;
  for (It it = lo; it != hi; ++it)
    *it = m[*it];
}

/*** Example Usage

Sample Output:

0 4 4 1 3 2 5 5
```

```
0 4 4 1 3 2 5 5
1 0 2 0 3 1

***/

#include <iostream>
using namespace std;

template<class It> void print_range(It lo, It hi) {
  while (lo != hi)
    cout << *(lo++) << "␣";
  cout << endl;
}

int main() {
  {
    int a[] = {1, 30, 30, 7, 9, 8, 99, 99};
    compress1(a, a + 8);
    print_range(a, a + 8);
  }
  {
    int a[] = {1, 30, 30, 7, 9, 8, 99, 99};
    compress2(a, a + 8);
    print_range(a, a + 8);
  }
  { //works on doubles too
    double a[] = {0.5, -1.0, 3, -1.0, 20, 0.5};
    compress1(a, a + 6);
    print_range(a, a + 6);
  }
  return 0;
}
```

1.1.5 Selection (Quickselect)

```
/*

Quickselect (also known as Hoare's algorithm) is a selection algorithm
which rearranges the elements in a sequence such that the element at
the nth position is the element that would be there if the sequence
were sorted. The other elements in the sequence are partioned around
the nth element. That is, they are left in no particular order, except
that no element before the nth element is is greater than it, and no
element after it is less.

The following implementation is equivalent to std::nth_element(),
taking in two random access iterators as the range and performing the
described operation in expected linear time.

Time Complexity (Average): O(n) on the distance between lo and hi.
Time Complexity (Worst): O(n^2), although this *almost never* occurs.
Space Complexity: O(1) auxiliary.

*/

#include <algorithm> /* std::swap() */
#include <cstdlib>   /* rand() */
```

```
#include <iterator>  /* std::iterator_traits */

int rand32() {
  return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
}

template<class It> It rand_partition(It lo, It hi) {
  std::swap(*(lo + rand32() % (hi - lo)), *(hi - 1));
  typename std::iterator_traits<It>::value_type mid = *(hi - 1);
  It i = lo - 1;
  for (It j = lo; j != hi; ++j)
    if (*j <= mid)
      std::swap(*(++i), *j);
  return i;
}

template<class It> void nth_element2(It lo, It n, It hi) {
  for (;;) {
    It k = rand_partition(lo, hi);
    if (n < k)
      hi = k;
    else if (n > k)
      lo = k + 1;
    else
      return;
  }
}

/*** Example Usage

Sample Output:
2 3 1 5 4 6 8 7 9

***/

#include <iostream>
using namespace std;

template<class It> void print_range(It lo, It hi) {
  while (lo != hi)
    cout << *(lo++) << "␣";
  cout << endl;
}

int main () {
  int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  random_shuffle(a, a + 9);
  nth_element2(a, a + 5, a + 9);
  print_range(a, a + 9);
  return 0;
}
```

1.2 Array Queries

1.2.1 Longest Increasing Subsequence

```
/*

Given an array a[] of size n, determine a longest subsequence of a[]
such that all of its elements are in ascending order. This subsequence
is not necessarily contiguous or unique, so only one such answer will
be found. The problem is efficiently solved using dynamic programming
and binary searching, since it has the following optimal substructure
with respect to the i-th position in the array:

  LIS[i] = 1 + max(LIS[j] for all j < i and a[j] < a[i])
  Otherwise if such a j does not exist, then LIS[i] = 1.

Explanation: https://en.wikipedia.org/wiki/Longest_increasing_subsequence

Time Complexity: O(n log n) on the size of the array.
Space Complexity: O(n) auxiliary.

*/

#include <vector>

std::vector<int> tail, prev;

template<class T> int lower_bound(int len, T a[], int key) {
  int lo = -1, hi = len;
  while (hi - lo > 1) {
    int mid = (lo + hi) / 2;
    if (a[tail[mid]] < key)
      lo = mid;
    else
      hi = mid;
  }
  return hi;
}

template<class T> std::vector<T> lis(int n, T a[]) {
  tail.resize(n);
  prev.resize(n);
  int len = 0;
  for (int i = 0; i < n; i++) {
    int pos = lower_bound(len, a, a[i]);
    if (len < pos + 1)
      len = pos + 1;
    prev[i] = pos > 0 ? tail[pos - 1] : -1;
    tail[pos] = i;
  }
  std::vector<T> res(len);
  for (int i = tail[len - 1]; i != -1; i = prev[i])
    res[--len] = a[i];
  return res;
}

/*** Example Usage

Sample Output:
-5 1 9 10 11 13

***/

```

```
#include <iostream>
using namespace std;

template<class It> void print_range(It lo, It hi) {
  while (lo != hi)
    cout << *(lo++) << "␣";
  cout << endl;
}

int main () {
  int a[] = {-2, -5, 1, 9, 10, 8, 11, 10, 13, 11};
  vector<int> res = lis(10, a);
  print_range(res.begin(), res.end());
  return 0;
}
```

1.2.2 Maximal Subarray Sum (Kadane's)

```
/*

Given a sequence of numbers (with at least one positive number), find
the maximum possible sum of any contiguous subarray. Kadane's algorithm
scans through the array, computing at each index the maximum (positive
sum) subarray ending at that position. This subarray is either empty
(in which case its sum is zero) or consists of one more element than
the maximum subarray ending at the previous position.

*/

#include <algorithm> /* std::fill() */
#include <iterator>  /* std::iterator_traits */
#include <limits>    /* std::numeric_limits */
#include <vector>

/*

The following implementation takes two random access iterators as the
range of values to be considered. Optionally, two pointers to integers
may be passed to have the positions of the begin and end indices of
the maximal sum subarray stored. begin_idx will be inclusive while
end_idx will be exclusive (i.e. (lo + begin_idx) will reference the
first element of the max sum subarray and (lo + end_idx) will reference
the index just past the last element of the subarray. Note that the
following version does not allow empty subarrays to be returned, so the
the max element will simply be returned if the array is all negative.

Time Complexity: O(n) on the distance between lo and hi.
Space Complexty: O(1) auxiliary.

*/

template<class It> typename std::iterator_traits<It>::value_type
max_subarray_sum(It lo, It hi, int *begin_idx = 0, int *end_idx = 0) {
  typedef typename std::iterator_traits<It>::value_type T;
  int curr_begin = 0, begin = 0, end = -1;
  T sum = 0, max_sum = std::numeric_limits<T>::min();
  for (It it = lo; it != hi; ++it) {
    sum += *it;
    if (sum < 0) {
      sum = 0;
      curr_begin = (it - lo) + 1;
    } else if (max_sum < sum) {
      max_sum = sum;
      begin = curr_begin;
      end = (it - lo) + 1;
    }
  }
  if (end == -1) { //all negative, just return the max value
    for (It it = lo; it != hi; ++it) {
      if (max_sum < *it) {
        max_sum = *it;
        begin = it - lo;
        end = begin + 1;
      }
    }
  }
  if (begin_idx != 0 && end_idx != 0) {
    *begin_idx = begin;
    *end_idx = end;
  }
  return max_sum;
}

/*

Maximal Submatrix Sum

In the 2-dimensional version of the problem, the largest sum of any
rectangular submatrix must be found for a matrix n rows by m columns.
Kadane's algorithm is applied to each interval [lcol, hcol] of columns
in the matrix, for an overall cubic time solution. The input must be a
two dimensional vector, where the outer vector must contain n vectors
each with m elements. Optionally, four int pointers begin_row, end_row,
begin_col, and end_col may be passed. If so, then their dereferenced
values will be set to the boundary indices of the max sum submatrix.
Note that begin_row and begin_col are inclusive indices, while end_row
and end_col are exclusive (referring to the index just past the end).

Time Complexity: O(m^2 * n) for a matrix with m columns and n rows.
Space Complexity: O(n) auxiliary.

*/

template<class T>
T max_submatrix_sum(const std::vector< std::vector<T> > & mat,
                    int *begin_row = 0, int *end_row = 0,
                    int *begin_col = 0, int *end_col = 0) {
  int n = mat.size(), m = mat[0].size();
  std::vector<T> sums(n);
  T sum, max_sum = std::numeric_limits<T>::min();
  for (int lcol = 0; lcol < m; lcol++) {
    std::fill(sums.begin(), sums.end(), 0);
    for (int hcol = lcol; hcol < m; hcol++) {
      for (int i = 0; i < n; i++)
        sums[i] += mat[i][hcol];
      int begin, end;
      sum = max_subarray_sum(sums.begin(), sums.end(), &begin, &end);
      if (sum > max_sum) {
        max_sum = sum;
        if (begin_row != 0) {
          *begin_row = begin;
          *end_row = end;
          *begin_col = lcol;
          *end_col = hcol + 1;
        }
      }
    }
  }
  return max_sum;
}

/*** Example Usage

Sample Output:
1D example - the max sum subarray is
4 -1 2 1
2D example - the max sum submatrix is
9 2
-4 1
-1 8

***/

#include <cassert>
#include <iostream>
using namespace std;

int main() {
  {
    int a[] = {-2, -1, -3, 4, -1, 2, 1, -5, 4};
    int begin, end;
    assert(max_subarray_sum(a, a + 3) == -1);
    assert(max_subarray_sum(a, a + 9, &begin, &end) == 6);
    cout << "1D␣example␣-␣the␣max␣sum␣subarray␣is" << endl;
    for (int i = begin; i < end; i++)
      cout << a[i] << "␣";
    cout << endl;
  }
  {
    const int n = 4, m = 5;
    int a[n][m] = {{ 0, -2, -7,  0,  5},
                   { 9,  2, -6,  2, -4},
                   {-4,  1, -4,  1,  0},
                   {-1,  8,  0, -2,  3}};
    vector< vector<int> > mat(n);
    for (int i = 0; i < n; i++)
      mat[i] = vector<int>(a[i], a[i] + m);
    int lrow, hrow, lcol, hcol;
    assert(max_submatrix_sum(mat, &lrow, &hrow, &lcol, &hcol) == 15);
    cout << "2D␣example␣-␣The␣max␣sum␣submatrix␣is" << endl;
    for (int i = lrow; i < hrow; i++) {
      for (int j = lcol; j < hcol; j++)
        cout << mat[i][j] << "␣";
      cout << endl;
    }
  }
  return 0;
}
```

1.2.3 Majority Element (Boyer-Moore)

```
/*

Given a sequence of n elements, the majority vote problem asks to find
an element that occurs more frequently than all others, or determine
that no such element exists. Formally, a value must occur strictly
greater than floor(n/2) times to be considered the majority element.
Boyer-Moore majority vote algorithm scans through the sequence and
keeps track of a running counter for the most likely candidate so far.
Whenever a value is equal to the current candidate, the counter is
incremented, otherwise the counter is decremented. When the counter is
zero, the candidate is eliminated and a new candidate is considered.

The following implementation takes two random access iterators as the
sequence [lo, hi) of elements and returns an iterator pointing to one
instance of the majority element if it exists, or the iterator hi if
there is no majority.

Time Complexity: O(n) on the size of the array.
Space Complexty: O(1) auxiliary.

*/

template<class It> It majority(It lo, It hi) {
  int cnt = 0;
  It candidate = lo;
  for (It it = lo; it != hi; ++it) {
    if (cnt == 0) {
      candidate = it;
      cnt = 1;
    } else if (*it == *candidate) {
      cnt++;
    } else {
      cnt--;
    }
  }
  cnt = 0;
  for (It it = lo; it != hi; ++it) {
    if (*it == *candidate)
      cnt++;
  }
  if (cnt <= (hi - lo) / 2)
    return hi;
  return candidate;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  int a[] = {3, 2, 3, 1, 3};
```

```cpp
  assert(*majority(a, a + 5) == 3);
  int b[] = {2, 3, 3, 3, 2, 1};
  assert(majority(b, b + 6) == b + 6);
  return 0;
}
```

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

```cpp
/*

Given a sequence of n (not necessarily unique) integers and a number v,
determine the minimum possible sum of any subset of the given sequence
that is not less than v. This is a generalization of a more well-known
version of the subset sum problem which asks whether a subset summing
to 0 exists (equivalent here to seeing if v = 0 yields an answer of 0).
Both problems are NP-complete. A meet-in-the-middle algorithm divides
the array in two equal parts. All possible sums of the lower and higher
parts are precomputed and sorted in a table. Finally, the table is
searched to find the lower bound.

The following implementation accepts two random access iterators as the
sequence [lo, hi) of integers, and the number v. Note that since the
sums can get large, 64-bit integers are necessary to avoid overflow.

Time Complexity: O(n * 2^(n/2)) on the distance between lo and hi.
Space Complexity: O(n) auxiliary.

*/

#include <algorithm> /* std::max(), std::sort() */
#include <limits>    /* std::numeric_limits */

template<class It>
long long sum_lower_bound(It lo, It hi, long long v) {
  int n = hi - lo;
  int llen = 1 << (n / 2);
  int hlen = 1 << (n - n / 2);
  long long *lsum = new long long[llen];
  long long *hsum = new long long[hlen];
  std::fill(lsum, lsum + llen, 0);
  std::fill(hsum, hsum + hlen, 0);
  for (int mask = 0; mask < llen; mask++) {
    for (int i = 0; i < n / 2; i++) {
      if ((mask >> i) & 1)
        lsum[mask] += *(lo + i);
    }
  }
  for (int mask = 0; mask < hlen; mask++) {
    for (int i = 0; i < n - n / 2; i++) {
      if ((mask >> i) & 1)
        hsum[mask] += *(lo + i + n / 2);
    }
  }
  std::sort(lsum, lsum + llen);
  std::sort(hsum, hsum + llen);
  int l = 0, r = hlen - 1;
  long long curr = std::numeric_limits<long long>::min();
  while (l < llen && r >= 0) {
    if (lsum[l] + hsum[r] <= v) {
      curr = std::max(curr, lsum[l] + hsum[r]);
      l++;
    } else {
      r--;
    }
  }
  delete[] lsum;
  delete[] hsum;
  return curr;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  int a[] = {9, 1, 5, 0, 1, 11, 5};
  assert(sum_lower_bound(a, a + 7, 8) == 7);
  int b[] = {-7, -3, -2, 5, 8};
  assert(sum_lower_bound(b, b + 5, 0) == 0);
  return 0;
}
```

1.2.5 Maximal Zero Submatrix

```
/*

Given an n by m rectangular matrix of 0's and 1's, determine the area
of the largest rectangular submatrix which contains only 0's. This can
be reduced the problem of finding the maximum rectangular area under a
histogram, which can be efficiently solved using a stack. The following
implementation accepts a 2-dimensional vector of bools and returns the
area of the maximum zero submatrix.

Explanation: http://stackoverflow.com/a/13657337

Time Complexity: O(n * m) for a matrix n rows by m columns.
Space Complexity: O(m) auxiliary.

*/

#include <algorithm> /* std::max() */
#include <vector>

int max_zero_submatrix(const std::vector< std::vector<bool> > & mat) {
  int n = mat.size(), m = mat[0].size(), res = 0;
  std::vector<int> d(m, -1), d1(m), d2(m), stack;
  for (int r = 0; r < n; r++) {
    for (int c = 0; c < m; c++) {
      if (mat[r][c])
        d[c] = r;
    }
    stack.clear();
    for (int c = 0; c < m; c++) {
      while (!stack.empty() && d[stack.back()] <= d[c])
```

```cpp
        stack.pop_back();
      d1[c] = stack.empty() ? -1 : stack.back();
      stack.push_back(c);
    }
    stack.clear();
    for (int c = m - 1; c >= 0; c--) {
      while (!stack.empty() && d[stack.back()] <= d[c])
        stack.pop_back();
      d2[c] = stack.empty() ? m : stack.back();
      stack.push_back(c);
    }
    for (int j = 0; j < m; j++)
      res = std::max(res, (r - d[j]) * (d2[j] - d1[j] - 1));
  }
  return res;
}

/*** Example Usage ***/

#include <cassert>
using namespace std;

int main() {
  const int n = 5, m = 6;
  bool a[n][m] = {{1, 0, 1, 1, 0, 0},
                  {1, 0, 0, 1, 0, 0},
                  {0, 0, 0, 0, 0, 1},
                  {1, 0, 0, 1, 0, 0},
                  {1, 0, 1, 0, 0, 1}};
  std::vector< std::vector<bool> > mat(n);
  for (int i = 0; i < n; i++)
    mat[i] = vector<bool>(a[i], a[i] + m);
  assert(max_zero_submatrix(mat) == 6);
  return 0;
}
```

1.3 Searching

1.3.1 Discrete Binary Search

```cpp
/*

Not only can binary search be used to find the position of a given
element in a sorted array, it can also be used to find the input value
corresponding to any output value of a monotonic (either strictly
non-increasing or strictly non-decreasing) function in O(log n) running
time with respect to the domain. This is a special case of finding
the exact point at which any given monotonic Boolean function changes
from true to false (or vice versa). Unlike searching through an array,
discrete binary search is not restricted by available memory, which is
especially important while handling infinitely large search spaces such
as the real numbers.

binary_search_first_true() takes two integers lo and hi as boundaries
for the search space [lo, hi) (i.e. including lo, but excluding hi),
and returns the least integer k (lo <= k < hi) for which the Boolean
```

```
predicate pred(k) tests true. This function is correct if and only if
there exists a constant k where the return value of pred(x) is false
for all x < k and true for all x >= k.

binary_search_last_true() takes two integers lo and hi as boundaries
for the search space [lo, hi) (i.e. including lo, but excluding hi),
and returns the greatest integer k (lo <= k < hi) for which the Boolean
predicate pred(k) tests true. This function is correct if and only if
there exists a constant k where the return value of pred(x) is true
for all x <= k and false for all x > k.

Time Complexity: At most O(log n) calls to pred(), where n is the
distance between lo and hi.

Space Complexity: O(1) auxiliary.

*/

//000[1]11
template<class Int, class IntPredicate>
Int binary_search_first_true(Int lo, Int hi, IntPredicate pred) {
  Int mid, _hi = hi;
  while (lo < hi) {
    mid = lo + (hi - lo) / 2;
    if (pred(mid))
      hi = mid;
    else
      lo = mid + 1;
  }
  if (!pred(lo)) return _hi; //all false
  return lo;
}

//11[1]000
template<class Int, class IntPredicate>
Int binary_search_last_true(Int lo, Int hi, IntPredicate pred) {
  Int mid, _hi = hi;
  while (lo < hi) {
    mid = lo + (hi - lo + 1) / 2;
    if (pred(mid))
      lo = mid;
    else
      hi = mid - 1;
  }
  if (!pred(lo)) return _hi; //all true
  return lo;
}

/*

fbinary_search() is the equivalent of binary_search_first_true() on
floating point predicates. Since any given range of reals numbers is
dense, it is clear that the exact target cannot be found. Instead, the
function will return a value that is very close to the border between
false and true. The precision of the answer depends on the number of
repetitions the function uses. Since each repetition bisects the search
space, for r repetitions, the absolute error of the answer will be
1/(2^r) times the distance between lo and hi. Although it's possible to
control the error by looping while hi - lo is greater than an arbitrary
```

```
epsilon, it is much simpler to let the loop run for a sizable number of
iterations until floating point arithmetic breaks down. 100 iterations
is typically sufficient, reducing the search space to 2^-100 ~ 10^-30
times its original size.

Note that the function can be modified to find the "last true" point
in the range by interchanging lo and hi in the if-else statement.

Time Complexity: At most O(log n) calls to pred(), where n is the
distance between lo and hi divided by the desired absolute error.

Space Complexity: O(1) auxiliary.

*/

//000[1]11
template<class DoublePredicate>
double fbinary_search(double lo, double hi, DoublePredicate pred) {
  double mid;
  for (int reps = 0; reps < 100; reps++) {
    mid = (lo + hi) / 2.0;
    if (pred(mid))
      hi = mid;
    else
      lo = mid;
  }
  return lo;
}

/*** Example Usage ***/

#include <cassert>
#include <cmath>

//Simple predicate examples:
bool pred1(int x) { return x >= 3; }
bool pred2(int x) { return false; }
bool pred3(int x) { return x <= 5; }
bool pred4(int x) { return true; }
bool pred5(double x) { return x >= 1.2345; }

int main() {
  assert(binary_search_first_true(0, 7, pred1) == 3);
  assert(binary_search_first_true(0, 7, pred2) == 7);
  assert(binary_search_last_true(0, 7, pred3)  == 5);
  assert(binary_search_last_true(0, 7, pred4)  == 7);
  assert(fabs(fbinary_search(-10.0, 10.0, pred5) - 1.2345) < 1e-15);
  return 0;
}
```

1.3.2 Ternary Search

```
/*

Given a unimodal function f(x), find its maximum or minimum point to a
an arbitrarily specified absolute error.

```

```
ternary_search_min() takes the domain [lo, hi] of a continuous function
f(x) and returns a number x such that f is strictly decreasing on the
interval [lo, x] and strictly increasing on the interval [x, hi]. For
ternary search to work, this x must exist and be unique.

ternary_search_max() takes the domain [lo, hi] of a continuous function
f(x) and returns a number x such that f is strictly increasing on the
interval [lo, x] and strictly decreasing on the interval [x, hi]. For
ternary search to work, this x must exist and be unique.

Time Complexity: At most O(log n) calls to f, where n is the distance
between lo and hi divided by the desired absolute error (epsilon).

Space Complexity: O(1) auxiliary.

*/

template<class UnimodalFunction>
double ternary_search_min(double lo, double hi, UnimodalFunction f) {
  static const double EPS = 1e-9;
  double lthird, hthird;
  while (hi - lo > EPS) {
    lthird = lo + (hi - lo) / 3;
    hthird = hi - (hi - lo) / 3;
    if (f(lthird) < f(hthird))
      hi = hthird;
    else
      lo = lthird;
  }
  return lo;
}

template<class UnimodalFunction>
double ternary_search_max(double lo, double hi, UnimodalFunction f) {
  static const double EPS = 1e-9;
  double lthird, hthird;
  while (hi - lo > EPS) {
    lthird = lo + (hi - lo) / 3;
    hthird = hi - (hi - lo) / 3;
    if (f(lthird) < f(hthird))
      lo = lthird;
    else
      hi = hthird;
  }
  return hi;
}

/*** Example Usage ***/

#include <cmath>
#include <cassert>

bool eq(double a, double b) {
  return fabs(a - b) < 1e-9;
}

//parabola opening up with vertex at (-2, -24)
double f1(double x) {
  return 3*x*x + 12*x - 12;
}

//parabola opening down with vertex at (2/19, 8366/95)
double f2(double x) {
  return -5.7*x*x + 1.2*x + 88;
}

//absolute value function shifted to the right by 30 units
double f3(double x) {
  return fabs(x - 30);
}

int main() {
  assert(eq(ternary_search_min(-1000, 1000, f1), -2));
  assert(eq(ternary_search_max(-1000, 1000, f2), 2.0 / 19));
  assert(eq(ternary_search_min(-1000, 1000, f3), 30));
  return 0;
}
```

1.3.3 Hill Climbing

```
/*

Given a continuous function f on two real numbers, hill climbing is a
technique that can be used to find the local maximum or minimum point
based on some (possibly random) initial guess. Then, the algorithm
considers taking a single step in each of a fixed number of directions.
The direction with the best result is selected and steps are further
taken there until the answer no longer improves. When this happens, the
step size is reduced and the process repeats until a desired absolute
error is reached. The result is not necessarily the global extrema, and
the algorithm's success will heavily depend on the initial guess.

The following function find_min() takes the function f, any starting
guess (x0, y0), and optionally two pointers to double used for storing
the answer coordinates. find_min() returns a local minimum point near
the initial guess, and if the two pointers are given, then coordinates
will be stored into the variables pointed to by x_ans and y_ans.

Time Complexity: At most O(d log n) calls to f, where d is the number
of directions considered at each position and n is the search space,
roughly proportional to the largest possible step size divided by the
smallest possible step size.

*/

#include <cmath>
#include <iostream>
using namespace std;

template<class BinaryFunction>
double find_min(BinaryFunction f, double x0, double y0,
                double *x_ans = 0, double *y_ans = 0) {
  static const double PI = acos(-1.0);
  static const double STEP_MAX = 1000000;
  static const double STEP_MIN = 1e-9;
  static const int DIRECTIONS = 6;
  double x = x0, y = y0, res = f(x0, y0);
  for (double step = STEP_MAX; step > STEP_MIN; ) {
    double best = res, best_x = x, best_y = y;
    bool found = false;
    for (int i = 0; i < DIRECTIONS; i++) {
      double a = 2.0 * PI * i / DIRECTIONS;
      double x2 = x + step * cos(a);
      double y2 = y + step * sin(a);
      double val = f(x2, y2);
      if (best > val) {
        best_x = x2;
        best_y = y2;
        best = val;
        found = true;
      }
    }
    if (!found) {
      step /= 2.0;
    } else {
      x = best_x;
      y = best_y;
      res = best;
    }
  }
  if (x_ans != 0 && y_ans != 0) {
    *x_ans = x;
    *y_ans = y;
  }
  return res;
}

/*** Example Usage ***/

#include <cassert>
#include <cmath>

bool eq(double a, double b) {
  return fabs(a - b) < 1e-8;
}

//minimized at f(2, 3) = 0
double f(double x, double y) {
  return (x - 2)*(x - 2) + (y - 3)*(y - 3);
}

int main() {
  double x, y;
  assert(eq(find_min(f, 0, 0, &x, &y), 0));
  assert(eq(x, 2) && eq(y, 3));
  return 0;
}
```

1.3.4 Convex Hull Trick (Semi-Dynamic)

```
/*

Given a set of pairs (m, b) describing lines of the form y = mx + b,
process a set of x-coordinate queries each asking to find the minimum
y-value of any of the given lines when evaluated at the specified x.
The convex hull optimization technique first ignores all lines which
never take on the maximum at any x value, then sorts the rest in order
of descending slope. The intersection points of adjacent lines in this
sorted list form the upper envelope of a convex hull, and line segments
connecting these points always take on the minimum y-value. The result
can be split up into x-intervals each mapped to the line which takes on
the minimum in that interval. The intervals can be binary searched to
solve each query in O(log n) time on the number of lines.

Explanation: http://wcipeg.com/wiki/Convex_hull_trick

The following implementation is a concise, semi-dynamic version which
supports an an interlaced series of add line and query operations.
However, two key preconditions are that each call to add_line(m, b)
must have m as the minimum slope of all lines added so far, and each
call to get_min(x) must have x as the maximum x of all queries so far.
As a result, pre-sorting the lines and queries may be necessary (in
which case the running time will be that of the sorting algorithm).

Time Complexity: O(n) on the number of calls to add_line(). Since the
number of steps taken by add_line() and get_min() are both bounded by
the number of lines added so far, their running times are respectively
O(1) amortized.

Space Complexity: O(n) auxiliary on the number of calls to add_line().

*/

#include <vector>

std::vector<long long> M, B;
int ptr = 0;

void add_line(long long m, long long b) {
  int len = M.size();
  while (len > 1 && (B[len - 2] - B[len - 1]) * (m - M[len - 1]) >=
                    (B[len - 1] - b) * (M[len - 1] - M[len - 2])) {
    len--;
  }
  M.resize(len);
  B.resize(len);
  M.push_back(m);
  B.push_back(b);
}

long long get_min(long long x) {
  if (ptr >= (int)M.size())
    ptr = (int)M.size() - 1;
  while (ptr + 1 < (int)M.size() && M[ptr + 1] * x + B[ptr + 1] <=
                                    M[ptr] * x + B[ptr]) {
    ptr++;
  }
  return M[ptr] * x + B[ptr];
}

/*** Example Usage ***/

```

```cpp
#include <cassert>

int main() {
  add_line(3, 0);
  add_line(2, 1);
  add_line(1, 2);
  add_line(0, 6);
  assert(get_min(0) == 0);
  assert(get_min(1) == 3);
  assert(get_min(2) == 4);
  assert(get_min(3) == 5);
  return 0;
}
```

1.3.5 Convex Hull Trick (Fully Dynamic)

```cpp
/*

Given a set of pairs (m, b) describing lines of the form y = mx + b,
process a set of x-coordinate queries each asking to find the minimum
y-value of any of the given lines when evaluated at the specified x.
The convex hull optimization technique first ignores all lines which
never take on the maximum at any x value, then sorts the rest in order
of descending slope. The intersection points of adjacent lines in this
sorted list form the upper envelope of a convex hull, and line segments
connecting these points always take on the minimum y-value. The result
can be split up into x-intervals each mapped to the line which takes on
the minimum in that interval. The intervals can be binary search to
solve each query in O(log n) time on the number of lines.

Explanation: http://wcipeg.com/wiki/Convex_hull_trick

The following implementation is a fully dynamic version, using a
self-balancing binary search tree (std::set) to support calling line
addition and query operations in any desired order. In addition, one
may instead optimize for maximum y by setting QUERY_MAX to true.

Time Complexity: O(n log n) for n calls to add_line(), where each call
is O(log n) amortized on the number of lines added so far. Each call to
get_best() runs in O(log n) on the number of lines added so far.

Space Complexity: O(n) auxiliary on the number of calls to add_line().

*/

#include <set>

const bool QUERY_MAX = false;
const double INF = 1e30;

struct line {
  long long m, b, val;
  double xlo;
  bool is_query;

  line(long long m, long long b) {
    this->m = m;
    this->b = b;
    val = 0;
    xlo = -INF;
    is_query = false;
  }

  long long evaluate(long long x) const {
    return m * x + b;
  }

  bool parallel(const line & l) const {
    return m == l.m;
  }

  double intersect(const line & l) const {
    if (parallel(l))
      return INF;
    return (double)(l.b - b)/(m - l.m);
  }

  bool operator < (const line & l) const {
    if (l.is_query)
      return QUERY_MAX ? xlo < l.val : l.val < xlo;
    return m < l.m;
  }
};

std::set<line> hull;

typedef std::set<line>::iterator hulliter;

bool has_prev(hulliter it) {
  return it != hull.begin();
}

bool has_next(hulliter it) {
  return it != hull.end() && ++it != hull.end();
}

bool irrelevant(hulliter it) {
  if (!has_prev(it) || !has_next(it))
    return false;
  hulliter prev = it; --prev;
  hulliter next = it; ++next;
  return QUERY_MAX ?
          prev->intersect(*next) <= prev->intersect(*it) :
          next->intersect(*prev) <= next->intersect(*it);
}

hulliter update_left_border(hulliter it) {
  if ((QUERY_MAX && !has_prev(it)) || (!QUERY_MAX && !has_next(it)))
    return it;
  hulliter it2 = it;
  double val = it->intersect(QUERY_MAX ? *--it2 : *++it2);
  line buf(*it);
  buf.xlo = val;
  hull.erase(it++);
  return hull.insert(it, buf);
}
```

```

void add_line(long long m, long long b) {
  line l(m, b);
  hulliter it = hull.lower_bound(l);
  if (it != hull.end() && it->parallel(l)) {
    if ((QUERY_MAX && it->b < b) || (!QUERY_MAX && b < it->b))
      hull.erase(it++);
    else
      return;
  }
  it = hull.insert(it, l);
  if (irrelevant(it)) {
    hull.erase(it);
    return;
  }
  while (has_prev(it) && irrelevant(--it))
    hull.erase(it++);
  while (has_next(it) && irrelevant(++it))
    hull.erase(it--);
  it = update_left_border(it);
  if (has_prev(it))
    update_left_border(--it);
  if (has_next(++it))
    update_left_border(++it);
}

long long get_best(long long x) {
  line q(0, 0);
  q.val = x;
  q.is_query = true;
  hulliter it = hull.lower_bound(q);
  if (QUERY_MAX)
    --it;
  return it->evaluate(x);
}

/*** Example Usage ***/

#include <cassert>

int main() {
  add_line(3, 0);
  add_line(0, 6);
  add_line(1, 2);
  add_line(2, 1);
  assert(get_best(0) == 0);
  assert(get_best(1) == 3);
  assert(get_best(2) == 4);
  assert(get_best(3) == 5);
  return 0;
}
```

1.4 Cycle Detection

1.4.1 Floyd’s Algorithm

```cpp
/*

For a function f which maps a finite set S to itself and any initial
value x[0] in S, the same value must occur twice in the sequence below:

  x[0], x[1] = f(x[0]), x[2] = f(x[1]), ..., x[i] = f(x[i - 1])

That is, there must exist numbers i, j (i < j) such that x[i] = x[j].
Once this happens, the sequence will continue periodically by repeating
the same sequence of values from x[i] to x[j  1]. Cycle detection asks
to find i and j, given the function f and initial value x[0]. This is
also analogous to the problem of detecting a cycle in a linked list,
which will make it degenerate.

Floyd's cycle-finding algorithm, a.k.a. the "tortoise and the hare
algorithm", is a space-efficient algorithm that moves two pointers
through the sequence at different speeds. Each step in the algorithm
moves the "tortoise" one step forward and the "hare" two steps forward
in the sequence, comparing the sequence values at each step. The first
value which is simultaneously pointed to by both pointers is the start
of the sequence.

Time Complexity: O(mu + lambda), where mu is the smallest index of the
sequence on which a cycle starts, and lambda is the cycle's length.

Space Complexity: O(1) auxiliary.

*/

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

template<class IntFunction>
std::pair<int, int> find_cycle(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);
}

/*** Example Usage ***/

#include <cassert>
#include <set>
#include <iostream>
```

```
using namespace std;

const int x0 = 0;

int f(int x) {
  return (123 * x * x + 4567890) % 1337;
}

void verify(int x0, int start, int length) {
  set<int> s;
  int x = x0;
  for (int i = 0; i < start; i++) {
    assert(!s.count(x));
    s.insert(x);
    x = f(x);
  }
  int startx = x;
  s.clear();
  for (int i = 0; i < length; i++) {
    assert(!s.count(x));
    s.insert(x);
    x = f(x);
  }
  assert(startx == x);
}

int main () {
  pair<int, int> res = find_cycle(f, x0);
  assert(res == make_pair(4, 2));
  verify(x0, res.first, res.second);
  return 0;
}
```

1.4.2 Brent's Algorithm

```
/*

For a function f which maps a finite set S to itself and any initial
value x[0] in S, the same value must occur twice in the sequence below:

  x[0], x[1] = f(x[0]), x[2] = f(x[1]), ..., x[i] = f(x[i - 1])

That is, there must exist numbers i, j (i < j) such that x[i] = x[j].
Once this happens, the sequence will continue periodically by repeating
the same sequence of values from x[i] to x[j  1]. Cycle detection asks
to find i and j, given the function f and initial value x[0]. This is
also analogous to the problem of detecting a cycle in a linked list,
which will make it degenerate.

While Floyd's cycle-finding algorithm finds cycles by simultaneously
moving two pointers at different speeds, Brent's algorithm keeps the
tortoise pointer stationary and "teleports" it to the hare pointer
every power of two. The smallest power of two for which they meet is
the start of the first cycle. This improves upon the constant factor
of Floyd's algorithm by reducing the number of function calls.

Time Complexity: O(mu + lambda), where mu is the smallest index of the
```

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

Space Complexity: O(1) auxiliary.

*/


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

template<class IntFunction>
std::pair<int, int> find_cycle(IntFunction f, int x0) {
  int power = 1, length = 1;
  int tortoise = x0, hare = f(x0);
  while (tortoise != hare) {
    if (power == length) {
      tortoise = hare;
      power *= 2;
      length = 0;
    }
    hare = f(hare);
    length++;
  }
  hare = x0;
  for (int i = 0; i < length; i++)
    hare = f(hare);
  int start = 0;
  tortoise = x0;
  while (tortoise != hare) {
    tortoise = f(tortoise);
    hare = f(hare);
    start++;
  }
  return std::make_pair(start, length);
}

/*** Example Usage ***/

#include <cassert>
#include <set>
using namespace std;

const int x0 = 0;

int f(int x) {
  return (123 * x * x + 4567890) % 1337;
}

void verify(int x0, int start, int length) {
  set<int> s;
  int x = x0;
  for (int i = 0; i < start; i++) {
    assert(!s.count(x));
    s.insert(x);
    x = f(x);
  }
  int startx = x;
  s.clear();
  for (int i = 0; i < length; i++) {
    assert(!s.count(x));
    s.insert(x);
    x = f(x);
  }
  assert(startx == x);
}

int main () {
  pair<int, int> res = find_cycle(f, x0);
  assert(res == make_pair(4, 2));
  verify(x0, res.first, res.second);
  return 0;
}
```

1.5 Binary Exponentiation

```
/*

Given three positive, signed 64-bit integers, powmod() efficiently
computes the power of the first two integers, modulo the third integer.
Binary exponentiation, also known as "exponentiation by squaring,"
decomposes the computation with the observation that the exponent is
reduced by half whenever the base is squared. Odd-numbered exponents
can be dealt with by subtracting one and multiplying the overall
expression by the base of the power. This yields a logarithmic number
of multiplications while avoiding overflow. To further prevent overflow
in intermediate multiplications, multiplication can be done using the
similar principle of multiplication by adding. Despite using unsigned
64-bit integers for intermediate calculations and as parameter types,
each argument to powmod() must not exceed 2^63 - 1, the maximum value
of a signed 64-bit integer.

Time Complexity: O(log n) on the exponent of the power.
Space Complexity: O(1) auxiliary.

*/

typedef unsigned long long int64;

int64 mulmod(int64 a, int64 b, int64 m) {
  int64 x = 0, y = a % m;
  for (; b > 0; b >>= 1) {
    if (b & 1)
      x = (x + y) % m;
    y = (y << 1) % m;
  }
  return x % m;
}

int64 powmod(int64 a, int64 b, int64 m) {
  int64 x = 1, y = a;
  for (; b > 0; b >>= 1) {
    if (b & 1)
      x = mulmod(x, y, m);
    y = mulmod(y, y, m);
  }
  return x % m;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  assert(powmod(2, 10, 1000000007) == 1024);
  assert(powmod(2, 62, 1000000) == 387904);
  assert(powmod(10001, 10001, 100000) == 10001);
  return 0;
}
```

Chapter 2

Graph Theory

2.1 Depth-First Search

2.1.1 Graph Class and Depth-First Search

```
/*

A graph can be represented as a set of objects (a.k.a. vertices, or
nodes) and connections (a.k.a. edges) between pairs of objects. It can
also be stored as an adjacency matrix or adjacency list, the latter of
which is more space efficient but less time efficient for particular
operations such as checking whether a connection exists. A fundamental
task to perform on graphs is traversal, where all reachable vertices
are visited and actions are performed. Given any arbitrary starting
node, depth-first search (DFS) recursively explores each "branch" from
the current node as deep as possible before backtracking and following
other branches. Depth-first search has many applications, including
detecting cycles and solving generic puzzles.

The following implements a simple graph class using adjacency lists,
along with with depth-first search and a few applications. The nodes of
the graph are identified by integers indices numbered consecutively
starting from 0. The total number nodes will automatically increase
based upon the maximum argument ever passed to add_edge().

Time Complexity:
- add_edge() is O(1) amortized per call, or O(n) for n calls where each
  node index added is at most n.
- dfs(), has_cycle(), is_tree(), and is_dag() are each O(n) per call on
  the number of edges added so far.
- All other public member functions are O(1).

Space Complexity:
- O(n) to store a graph of n edges.
- dfs(), has_cycle(), is_tree(), and is_dag() each require O(n)
  auxiliary on the number of edges.
- All other public member functions require O(1) auxiliary.

*/

```

```cpp
#include <algorithm> /* std::max */
#include <cstddef>   /* size_t */
#include <vector>

class graph {
  std::vector< std::vector<int> > adj;
  bool _is_directed;

  template<class Action>
  void dfs(int n, std::vector<bool> & vis, Action act);

  bool has_cycle(int n, int prev, std::vector<bool> & vis,
                 std::vector<bool> & onstack);

 public:
  graph(bool is_directed = true) {
    this->_is_directed = is_directed;
  }

  bool is_directed() const {
    return _is_directed;
  }

  size_t nodes() const {
    return adj.size();
  }

  std::vector<int>& operator [] (int n) {
    return adj[n];
  }

  void add_edge(int u, int v);
  template<class Action> void dfs(int start, Action act);
  bool has_cycle();
  bool is_tree();
  bool is_dag();
};

void graph::add_edge(int u, int v) {
  if (u >= (int)adj.size() || v >= (int)adj.size())
    adj.resize(std::max(u, v) + 1);
  adj[u].push_back(v);
  if (!is_directed())
    adj[v].push_back(u);
}

template<class Action>
void graph::dfs(int n, std::vector<bool> & vis, Action act) {
  act(n);
  vis[n] = true;
  std::vector<int>::iterator it;
  for (it = adj[n].begin(); it != adj[n].end(); ++it) {
    if (!vis[*it])
      dfs(*it, vis, act);
  }
}

template<class Action> void graph::dfs(int start, Action act) {
  std::vector<bool> vis(nodes(), false);
  dfs(start, vis, act);
}

bool graph::has_cycle(int n, int prev, std::vector<bool> & vis,
                      std::vector<bool> & onstack) {
  vis[n] = true;
  onstack[n] = true;
  std::vector<int>::iterator it;
  for (it = adj[n].begin(); it != adj[n].end(); ++it) {
    if (is_directed() && onstack[*it])
      return true;
    if (!is_directed() && vis[*it] && *it != prev)
      return true;
    if (!vis[*it] && has_cycle(*it, n, vis, onstack))
      return true;
  }
  onstack[n] = false;
  return false;
}

bool graph::has_cycle() {
  std::vector<bool> vis(nodes(), false), onstack(nodes(), false);
  for (int i = 0; i < (int)adj.size(); i++)
    if (!vis[i] && has_cycle(i, -1, vis, onstack))
      return true;
  return false;
}

bool graph::is_tree() {
  return !is_directed() && !has_cycle();
}

bool graph::is_dag() {
  return is_directed() && !has_cycle();
}

/*** Example Usage

Sample Output:
DFS order: 0 1 2 3 4 5 6 7 8 9 10 11

***/

#include <cassert>
#include <iostream>
using namespace std;

void print_node(int n) {
  cout << n << "␣";
}

int main() {
  {
    graph g;
    g.add_edge(0, 1);
    g.add_edge(0, 6);
    g.add_edge(0, 7);
    g.add_edge(1, 2);
    g.add_edge(1, 5);
    g.add_edge(2, 3);
    g.add_edge(2, 4);
    g.add_edge(7, 8);
    g.add_edge(7, 11);
    g.add_edge(8, 9);
    g.add_edge(8, 10);
    cout << "DFS␣order:␣";
    g.dfs(0, print_node);
    cout << endl;
    assert(g[0].size() == 3);
    assert(!g.has_cycle());
  }
  {
    graph tree(false);
    tree.add_edge(0, 1);
    tree.add_edge(0, 2);
    tree.add_edge(1, 3);
    tree.add_edge(1, 4);
    assert(tree.is_tree());
    tree.add_edge(2, 3);
    assert(!tree.is_tree());
  }
  return 0;
}
```

2.1.2 Topological Sorting

```
/*

Description: Given a directed acyclic graph (DAG), order the nodes
such that for every edge from a to b, a precedes b in the ordering.
Usually, there is more than one possible valid ordering. The
following program uses DFS to produce one possible ordering.
This can also be used to detect whether the graph is a DAG.
Note that the DFS algorithm here produces a reversed topological
ordering, so the output must be printed backwards. The graph is
stored in an adjacency list.

Complexity: O(V+E) on the number of vertices and edges.

=~=~=~=~= Sample Input =~=~=~=~=
8 9
0 3
0 4
1 3
2 4
2 7
3 5
3 6
3 7
4 6

=~=~=~=~= Sample Output =~=~=~=~=
The topological order: 2 1 0 4 3 7 6 5

*/

```

```
#include <algorithm> /* std::fill(), std::reverse() */
#include <iostream>
#include <stdexcept> /* std::runtime_error() */
#include <vector>
using namespace std;

const int MAXN = 100;
vector<bool> vis(MAXN), done(MAXN);
vector<int> adj[MAXN], sorted;

void dfs(int u) {
  if (vis[u])
    throw std::runtime_error("Not␣a␣DAG.");
  if (done[u]) return;
  vis[u] = true;
  for (int j = 0; j < (int)adj[u].size(); j++)
    dfs(adj[u][j]);
  vis[u] = false;
  done[u] = true;
  sorted.push_back(u);
}

void toposort(int nodes) {
  fill(vis.begin(), vis.end(), false);
  fill(done.begin(), done.end(), false);
  sorted.clear();
  for (int i = 0; i < nodes; i++)
    if (!done[i]) dfs(i);
  reverse(sorted.begin(), sorted.end());
}

int main() {
  int nodes, edges, u, v;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v;
    adj[u].push_back(v);
  }
  toposort(nodes);
  cout << "The␣topological␣order:";
  for (int i = 0; i < (int)sorted.size(); i++)
    cout << "␣" << sorted[i];
  cout << "\n";
  return 0;
}
```

2.1.3 Eulerian Cycles

```
/*

Description: A Eulerian trail is a trail in a graph which
visits every edge exactly once. Similarly, an Eulerian circuit
or Eulerian cycle is an Eulerian trail which starts and ends
on the same vertex.

An undirected graph has an Eulerian cycle if and only if every
vertex has even degree, and all of its vertices with nonzero
```

```
degree belong to a single connected component.

A directed graph has an Eulerian cycle if and only if every
vertex has equal in degree and out degree, and all of its
vertices with nonzero degree belong to a single strongly
connected component.

Complexity: O(V+E) on the number of vertices and edges.

=~=~=~=~= Sample Input =~=~=~=~=
5 6
0 1
1 2
2 0
1 3
3 4
4 1

=~=~=~=~= Sample Output =~=~=~=~=
Eulerian cycle from 0 (directed): 0 1 3 4 1 2 0
Eulerian cycle from 2 (undirected): 2 1 3 4 1 0 2

*/

#include <algorithm> /* std::reverse() */
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 100;

vector<int> euler_cycle_directed(vector<int> adj[], int u) {
  vector<int> stack, res, cur_edge(MAXN);
  stack.push_back(u);
  while (!stack.empty()) {
    u = stack.back();
    stack.pop_back();
    while (cur_edge[u] < (int)adj[u].size()) {
      stack.push_back(u);
      u = adj[u][cur_edge[u]++];
    }
    res.push_back(u);
  }
  reverse(res.begin(), res.end());
  return res;
}

vector<int> euler_cycle_undirected(vector<int> adj[], int u) {
  vector<vector<bool> > used(MAXN, vector<bool>(MAXN, false));
  vector<int> stack, res, cur_edge(MAXN);
  stack.push_back(u);
  while (!stack.empty()) {
    u = stack.back();
    stack.pop_back();
    while (cur_edge[u] < (int)adj[u].size()) {
      int v = adj[u][cur_edge[u]++];
      if (!used[min(u, v)][max(u, v)]) {
        used[min(u, v)][max(u, v)] = 1;
        stack.push_back(u);
        u = v;
      }
    }
    res.push_back(u);
  }
  reverse(res.begin(), res.end());
  return res;
}

int main() {
  int nodes, edges, u, v;
  vector<int> g1[5], g2[5], cycle;

  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v;
    g1[u].push_back(v);
    g2[u].push_back(v);
    g2[v].push_back(u);
  }

  cycle = euler_cycle_directed(g1, 0);
  cout << "Eulerian␣cycle␣from␣0␣(directed):";
  for (int i = 0; i < (int)cycle.size(); i++)
    cout << "␣" << cycle[i];
  cout << "\n";

  cycle = euler_cycle_undirected(g2, 2);
  cout << "Eulerian␣cycle␣from␣2␣(undirected):";
  for (int i = 0; i < (int)cycle.size(); i++)
    cout << "␣" << cycle[i];
  cout << "\n";
  return 0;
}
```

2.1.4 Unweighted Tree Centers

```
/*

The following applies to unweighted, undirected trees only.

find_centers(): Returns 1 or 2 tree centers. The center
(or Jordan center) of a graph is the set of all vertices of
minimum eccentricity, that is, the set of all vertices A
where the max distance d(A,B) to other vertices B is minimal.

find_centroid(): Returns a vertex where all of its subtrees
have size <= N/2, where N is the number of nodes in the tree.

diameter(): The diameter of a tree is the greatest distance
d(A,B) between any two of the nodes in the tree.

Complexity: All three functions are O(V) on the number of
vertices in the tree.

=~=~=~=~= Sample Input =~=~=~=~=
6
```

```
0 1
1 2
1 4
3 4
4 5

=~=~=~=~= Sample Output =~=~=~=~=
Center(s): 1 4
Centroid: 4
Diameter: 3

*/

#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 100;
vector<int> adj[MAXN];

vector<int> find_centers(int n) {
  vector<int> leaves, degree(n);
  for (int i = 0; i < n; i++) {
    degree[i] = adj[i].size();
    if (degree[i] <= 1) leaves.push_back(i);
  }
  int removed = leaves.size();
  while (removed < n) {
    vector<int> nleaves;
    for (int i = 0; i < (int)leaves.size(); i++) {
      int u = leaves[i];
      for (int j = 0; j < (int)adj[u].size(); j++) {
        int v = adj[u][j];
        if (--degree[v] == 1)
          nleaves.push_back(v);
      }
    }
    leaves = nleaves;
    removed += leaves.size();
  }
  return leaves;
}

int find_centroid(int n, int u = 0, int p = -1) {
  int cnt = 1, v;
  bool good_center = true;
  for (int j = 0; j < (int)adj[u].size(); j++) {
    if ((v = adj[u][j]) == p) continue;
    int res = find_centroid(n, v, u);
    if (res >= 0) return res;
    int size = -res;
    good_center &= (size <= n / 2);
    cnt += size;
  }
  good_center &= (n - cnt <= n / 2);
  return good_center ? u : -cnt;
}

pair<int, int> dfs(int u, int p, int depth) {
  pair<int, int> res = make_pair(depth, u);
  for (int j = 0; j < (int)adj[u].size(); j++)
    if (adj[u][j] != p)
      res = max(res, dfs(adj[u][j], u, depth + 1));
  return res;
}

int diameter() {
  int furthest_vertex = dfs(0, -1, 0).second;
  return dfs(furthest_vertex, -1, 0).first;
}

int main() {
  int nodes, u, v;
  cin >> nodes;
  for (int i = 0; i < nodes - 1; i++) {
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
  vector<int> centers = find_centers(nodes);
  cout << "Center(s):";
  for (int i = 0; i < (int)centers.size(); i++)
    cout << "␣" << centers[i];
  cout << "\nCentroid:␣" << find_centroid(nodes);
  cout << "\nDiameter:␣" << diameter() << "\n";
  return 0;
}
```

2.2 Shortest Paths

2.2.1 Breadth First Search

```
/*

Description: Given an unweighted graph, traverse all reachable
nodes from a source node and determine the shortest path.

Complexity: O(V+E) on the number of vertices and edges.

Note: The line "for (q.push(start); !q.empty(); q.pop())"
is simply a mnemonic for looping a BFS with a FIFO queue.
This will not work as intended with a priority queue, such as in
Dijkstra's algorithm for solving weighted shortest paths

=~=~=~=~= Sample Input =~=~=~=~=
4 5
0 1
0 3
1 2
1 3
2 3
0 3

=~=~=~=~= Sample Output =~=~=~=~=
The shortest distance from 0 to 3 is 2.
```

```
Take the path: 0->1->3.

*/

#include <iostream>
#include <queue>
#include <vector>
using namespace std;

const int MAXN = 100, INF = 0x3f3f3f3f;
int dist[MAXN], pred[MAXN];
vector<int> adj[MAXN];

void bfs(int nodes, int start) {
  vector<bool> vis(nodes, false);
  for (int i = 0; i < nodes; i++) {
    dist[i] = INF;
    pred[i] = -1;
  }
  int u, v, d;
  queue<pair<int, int> > q;
  q.push(make_pair(start, 0));
  while (!q.empty()) {
    u = q.front().first;
    d = q.front().second;
    q.pop();
    vis[u] = true;
    for (int j = 0; j < (int)adj[u].size(); j++) {
      if (vis[v = adj[u][j]]) continue;
      dist[v] = d + 1;
      pred[v] = u;
      q.push(make_pair(v, d + 1));
    }
  }
}

//Use the precomputed pred[] array to print the path
void print_path(int dest) {
  int i = 0, j = dest, path[MAXN];
  while (pred[j] != -1) j = path[++i] = pred[j];
  cout << "Take the path: ";
  while (i > 0) cout << path[i--] << "->";
  cout << dest << ".\n";
}

int main() {
  int nodes, edges, u, v, start, dest;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v;
    adj[u].push_back(v);
  }
  cin >> start >> dest;
  bfs(nodes, start);
  cout << "The shortest distance from " << start;
  cout << " to " << dest << " is " << dist[dest] << ".\n";
  print_path(dest);
  return 0;
}
```

2.2.2 Dijkstra's Algorithm

```
/*

Description: Given a directed graph with positive weights only, find
the shortest distance to all nodes from a single starting node.

Implementation Notes: The graph is stored using an adjacency list.
This implementation negates distances before adding them to the
priority queue, since the container is a max-heap by default. This
method is suggested in contests because it is easier than defining
special comparators. An alternative would be declaring the queue
with template parameters (clearly, this way is very verbose and ugly):
  priority_queue< pair<int, int>, vector<pair<int, int> >,
                  greater<pair<int, int> > > pq;
If only the path between a single pair of nodes is needed, for speed,
we may break out of the loop as soon as the destination is reached
by inserting the line "if (a == dest) break;" after the line "pq.pop();"

Complexity: This version uses an adjacency list and priority queue
(internally a binary heap) and has a complexity of O((E+V) log V) =
O(E log V). The priority queue and adjacency list improves the
simplest O(V^2) version of the algorithm, which uses looping and
an adjacency matrix. If the priority queue is implemented as a more
sophisticated Fibonacci heap, the complexity becomes O(E + V log V).

Modification to Shortest Path Faster Algorithm: The code for Dijkstra's
algorithm here can be easily modified to become the Shortest Path Faster
Algorithm (SPFA) by simply commenting out "visit[a] = true;" and changing
the priority queue to a FIFO queue like in BFS. SPFA is a faster version
of the Bellman-Ford algorithm, working on negative path lengths (whereas
Dijkstra's cannot). Certain graphs can be constructed to make SPFA slow.

=~=~=~=~= Sample Input =~=~=~=~=
4 5
0 1 2
0 3 8
1 2 2
1 3 4
2 3 1
0 3

=~=~=~=~= Sample Output =~=~=~=~=
The shortest distance from 0 to 3 is 5.
Take the path: 0->1->2->3.

*/

#include <iostream>
#include <queue>
#include <vector>
using namespace std;

const int MAXN = 100, INF = 0x3f3f3f3f;
int dist[MAXN], pred[MAXN];
vector<pair<int, int> > adj[MAXN];

void dijkstra(int nodes, int start) {
  vector<bool> vis(nodes, false);
  for (int i = 0; i < nodes; i++) {
    dist[i] = INF;
    pred[i] = -1;
  }
  int u, v;
  dist[start] = 0;
  priority_queue<pair<int, int> > pq;
  pq.push(make_pair(0, start));
  while (!pq.empty()) {
    u = pq.top().second;
    pq.pop();
    vis[u] = true;
    for (int j = 0; j < (int)adj[u].size(); j++) {
      if (vis[v = adj[u][j].first]) continue;
      if (dist[v] > dist[u] + adj[u][j].second) {
        dist[v] = dist[u] + adj[u][j].second;
        pred[v] = u;
        pq.push(make_pair(-dist[v], v));
      }
    }
  }
}

//Use the precomputed pred[] array to print the path
void print_path(int dest) {
  int i = 0, j = dest, path[MAXN];
  while (pred[j] != -1) j = path[++i] = pred[j];
  cout << "Take␣the␣path:␣";
  while (i > 0) cout << path[i--] << "->";
  cout << dest << ".\n";
}

int main() {
  int nodes, edges, u, v, w, start, dest;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v >> w;
    adj[u].push_back(make_pair(v, w));
  }
  cin >> start >> dest;
  dijkstra(nodes, start);
  cout << "The␣shortest␣distance␣from␣" << start;
  cout << "␣to␣" << dest << "␣is␣" << dist[dest] << ".\n";
  print_path(dest);
  return 0;
}
```

2.2.3 Bellman-Ford Algorithm

```
/*

Description: Given a directed graph with positive or negative weights
but no negative cycles, find the shortest distance to all nodes from
a single starting node. The input graph is stored using an edge list.

Complexity: O(V*E) on the number of vertices and edges, respectively.
```

```

=~=~=~=~= Sample Input =~=~=~=~=
3 3
0 1 1
1 2 2
0 2 5
0 2

=~=~=~=~= Sample Output =~=~=~=~=
The shortest distance from 0 to 2 is 3.
Take the path: 0->1->2.

*/

#include <iostream>
#include <stdexcept>
#include <vector>
using namespace std;

struct edge { int u, v, w; };

const int MAXN = 100, INF = 0x3f3f3f3f;
int dist[MAXN], pred[MAXN];
vector<edge> e;

void bellman_ford(int nodes, int start) {
  for (int i = 0; i < nodes; i++) {
    dist[i] = INF;
    pred[i] = -1;
  }
  dist[start] = 0;
  for (int i = 0; i < nodes; i++) {
    for (int j = 0; j < (int)e.size(); j++) {
      if (dist[e[j].v] > dist[e[j].u] + e[j].w) {
        dist[e[j].v] = dist[e[j].u] + e[j].w;
        pred[e[j].v] = e[j].u;
      }
    }
  }
  //optional: report negative-weight cycles
  for (int i = 0; i < (int)e.size(); i++)
    if (dist[e[i].v] > dist[e[i].u] + e[i].w)
      throw std::runtime_error("Negative-weight␣found");
}

//Use the precomputed pred[] array to print the path
void print_path(int dest) {
  int i = 0, j = dest, path[MAXN];
  while (pred[j] != -1) j = path[++i] = pred[j];
  cout << "Take␣the␣path:␣";
  while (i > 0) cout << path[i--] << "->";
  cout << dest << ".\n";
}

int main() {
  int nodes, edges, u, v, w, start, dest;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v >> w;
    e.push_back((edge){u, v, w});
  }
  cin >> start >> dest;
  bellman_ford(nodes, start);
  cout << "The shortest distance from " << start;
  cout << " to " << dest << " is " << dist[dest] << ".\n";
  print_path(dest);
  return 0;
}
```

2.2.4 Floyd-Warshall Algorithm

```
/*

Description: Given a directed graph with positive or negative
weights but no negative cycles, find the shortest distance
between all pairs of nodes. The input graph is stored using
an adjacency matrix. Note that the input adjacency matrix
is converted to the distance matrix afterwards. If you still
need the adjacencies afterwards, back it up at the beginning.

Complexity: O(V^3) on the number of vertices.

=~=~=~=~= Sample Input =~=~=~=~=
3 3
0 1 1
1 2 2
0 2 5
0 2

=~=~=~=~= Sample Output =~=~=~=~=
The shortest distance from 0 to 2 is 3.
Take the path: 0->1->2.

*/

#include <iostream>
using namespace std;

const int MAXN = 100, INF = 0x3f3f3f3f;
int dist[MAXN][MAXN], next[MAXN][MAXN];

void initialize(int nodes) {
  for (int i = 0; i < nodes; i++)
    for (int j = 0; j < nodes; j++) {
      dist[i][j] = (i == j) ? 0 : INF;
      next[i][j] = -1;
    }
}

void floyd_warshall(int nodes) {
  for (int k = 0; k < nodes; k++)
   for (int i = 0; i < nodes; i++)
    for (int j = 0; j < nodes; j++)
      if (dist[i][j] > dist[i][k] + dist[k][j]) {
        dist[i][j] = dist[i][k] + dist[k][j];
        next[i][j] = k;
    }
}

void print_path(int u, int v) {
  if (next[u][v] != -1) {
    print_path(u, next[u][v]);
    cout << next[u][v];
    print_path(next[u][v], v);
  } else cout << "->";
}

int main() {
  int nodes, edges, u, v, w, start, dest;
  cin >> nodes >> edges;
  initialize(nodes);
  for (int i = 0; i < edges; i++) {
    cin >> u >> v >> w;
    dist[u][v] = w;
  }
  cin >> start >> dest;
  floyd_warshall(nodes);
  cout << "The␣shortest␣distance␣from␣" << start;
  cout << "␣to␣" << dest << "␣is␣";
  cout << dist[start][dest] << ".\n";

  //Use next[][] to recursively print the path
  cout << "Take␣the␣path␣" << start;
  print_path(start, dest);
  cout << dest << ".\n";
  return 0;
}
```

2.3 Connectivity

2.3.1 Strongly Connected Components (Kosaraju's Algorithm)

```
/*

Description: Determines the strongly connected components (SCC)
from a given directed graph. Given a directed graph, its SCCs
are its maximal strongly connected sub-graphs. A graph is
strongly connected if there is a path from each node to every
other node. Condensing the strongly connected components of a
graph into single nodes will result in a directed acyclic graph.
The input is stored in an adjacency list.

Complexity: O(V+E) on the number of vertices and edges.

Comparison with other SCC algorithms:
The strongly connected components of a graph can be efficiently
computed using Kosaraju's algorithm, Tarjan's algorithm, or the
path-based strong component algorithm. Tarjan's algorithm can
be seen as an improved version of Kosaraju's because it performs
a single DFS rather than two. Though they both have the same
complexity, Tarjan's algorithm is much more efficient in
practice. However, Kosaraju's algorithm is conceptually simpler.
```

```

=~=~=~=~= Sample Input =~=~=~=~=
8 14
0 1
1 2
1 4
1 5
2 3
2 6
3 2
3 7
4 0
4 5
5 6
6 5
7 3
7 6

=~=~=~=~= Sample Output =~=~=~=~=
Component: 1 4 0
Component: 7 3 2
Component: 5 6

*/

#include <algorithm> /* std::fill(), std::reverse() */
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 100;
vector<bool> vis(MAXN);
vector<int> adj[MAXN], order;
vector<vector<int> > scc;

void dfs(vector<int> graph[], vector<int> & res, int u) {
  vis[u] = true;
  for (int j = 0; j < (int)graph[u].size(); j++)
    if (!vis[graph[u][j]])
      dfs(graph, res, graph[u][j]);
  res.push_back(u);
}

void kosaraju(int nodes) {
  scc.clear();
  order.clear();
  vector<int> rev[nodes];
  fill(vis.begin(), vis.end(), false);
  for (int i = 0; i < nodes; i++)
    if (!vis[i]) dfs(adj, order, i);
  for (int i = 0; i < nodes; i++)
    for (int j = 0; j < (int)adj[i].size(); j++)
      rev[adj[i][j]].push_back(i);
  fill(vis.begin(), vis.end(), false);
  reverse(order.begin(), order.end());
  for (int i = 0; i < (int)order.size(); i++) {
    if (vis[order[i]]) continue;
    vector<int> component;
    dfs(rev, component, order[i]);
    scc.push_back(component);
  }
}

int main() {
  int nodes, edges, u, v;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v;
    adj[u].push_back(v);
  }
  kosaraju(nodes);
  for (int i = 0; i < (int)scc.size(); i++) {
    cout << "Component:";
    for (int j = 0; j < (int)scc[i].size(); j++)
      cout << "␣" << scc[i][j];
    cout << "\n";
  }
  return 0;
}
```

2.3.2 Strongly Connected Components (Tarjan's Algorithm)

```
/*

Description: Determines the strongly connected components (SCC)
from a given directed graph. Given a directed graph, its SCCs
are its maximal strongly connected sub-graphs. A graph is
strongly connected if there is a path from each node to every
other node. Condensing the strongly connected components of a
graph into single nodes will result in a directed acyclic graph.
The input is stored in an adjacency list.

In this implementation, a vector is used to emulate a stack
for the sake of simplicity. One useful property of Tarjans
algorithm is that, while there is nothing special about the
ordering of nodes within each component, the resulting DAG
is produced in reverse topological order.

Complexity: O(V+E) on the number of vertices and edges.

Comparison with other SCC algorithms:
The strongly connected components of a graph can be efficiently
computed using Kosaraju's algorithm, Tarjan's algorithm, or the
path-based strong component algorithm. Tarjan's algorithm can
be seen as an improved version of Kosaraju's because it performs
a single DFS rather than two. Though they both have the same
complexity, Tarjan's algorithm is much more efficient in
practice. However, Kosaraju's algorithm is conceptually simpler.

=~=~=~=~= Sample Input =~=~=~=~=
8 14
0 1
1 2
1 4
1 5
2 3
```

```
2 6
3 2
3 7
4 0
4 5
5 6
6 5
7 3
7 6

=~=~=~=~= Sample Output =~=~=~=~=
Component 1: 5 6
Component 2: 7 3 2
Component 3: 4 1 0

*/

#include <algorithm> /* std::fill() */
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 100, INF = 0x3f3f3f3f;
int timer, lowlink[MAXN];
vector<bool> vis(MAXN);
vector<int> adj[MAXN], stack;
vector<vector<int> > scc;

void dfs(int u) {
  lowlink[u] = timer++;
  vis[u] = true;
  stack.push_back(u);
  bool is_component_root = true;
  int v;
  for (int j = 0; j < (int)adj[u].size(); j++) {
    if (!vis[v = adj[u][j]]) dfs(v);
    if (lowlink[u] > lowlink[v]) {
      lowlink[u] = lowlink[v];
      is_component_root = false;
    }
  }
  if (!is_component_root) return;
  vector<int> component;
  do {
    vis[v = stack.back()] = true;
    stack.pop_back();
    lowlink[v] = INF;
    component.push_back(v);
  } while (u != v);
  scc.push_back(component);
}

void tarjan(int nodes) {
  scc.clear();
  stack.clear();
  fill(lowlink, lowlink + nodes, 0);
  fill(vis.begin(), vis.end(), false);
  timer = 0;
  for (int i = 0; i < nodes; i++)
```

```
    if (!vis[i]) dfs(i);
}

int main() {
  int nodes, edges, u, v;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v;
    adj[u].push_back(v);
  }
  tarjan(nodes);
  for (int i = 0; i < (int)scc.size(); i++) {
    cout << "Component:";
    for (int j = 0; j < (int)scc[i].size(); j++)
      cout << "␣" << scc[i][j];
    cout << "\n";
  }
  return 0;
}
```

2.3.3 Bridges, Cut-points, and Biconnectivity

```
/*

Description: The following operations apply to undirected graphs.

A bridge is an edge, when deleted, increases the number of
connected components. An edge is a bridge if and only ifit is not
contained in any cycle.

A cut-point (i.e. cut-vertex or articulation point) is any vertex
whose removal increases the number of connected components.

A biconnected component of a graph is a maximally biconnected
subgraph. A biconnected graph is a connected and "nonseparable"
graph, meaning that if any vertex were to be removed, the graph
will remain connected. Therefore, a biconnected graph has no
articulation vertices.

Any connected graph decomposes into a tree of biconnected
components called the "block tree" of the graph. An unconnected
graph will thus decompose into a "block forest."

See: http://en.wikipedia.org/wiki/Biconnected_component

Complexity: O(V+E) on the number of vertices and edges.

=~=~=~=~= Sample Input =~=~=~=~=
8 6
0 1
0 5
1 2
1 5
3 7
4 5

=~=~=~=~= Sample Output =~=~=~=~=
Cut Points: 5 1
Bridges:
1 2
5 4
3 7
Edge-Biconnected Components:
Component 1: 2
Component 2: 4
Component 3: 5 1 0
Component 4: 7
Component 5: 3
Component 6: 6
Adjacency List for Block Forest:
0 => 2
1 => 2
2 => 0 1
3 => 4
4 => 3
5 =>

*/

#include <algorithm> /* std::fill(), std::min() */
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 100;
int timer, lowlink[MAXN], tin[MAXN], comp[MAXN];
vector<bool> vis(MAXN);
vector<int> adj[MAXN], bcc_forest[MAXN];
vector<int> stack, cutpoints;
vector<vector<int> > bcc;
vector<pair<int, int> > bridges;

void dfs(int u, int p) {
  vis[u] = true;
  lowlink[u] = tin[u] = timer++;
  stack.push_back(u);
  int v, children = 0;
  bool cutpoint = false;
  for (int j = 0; j < (int)adj[u].size(); j++) {
    if ((v = adj[u][j]) == p) continue;
    if (vis[v]) {
      //lowlink[u] = min(lowlink[u], lowlink[v]);
      lowlink[u] = min(lowlink[u], tin[v]);
    } else {
      dfs(v, u);
      lowlink[u] = min(lowlink[u], lowlink[v]);
      cutpoint |= (lowlink[v] >= tin[u]);
      if (lowlink[v] > tin[u])
        bridges.push_back(make_pair(u, v));
      children++;
    }
  }
  if (p == -1) cutpoint = (children >= 2);
  if (cutpoint) cutpoints.push_back(u);
  if (lowlink[u] == tin[u]) {
    vector<int> component;
    do {
      v = stack.back();
      stack.pop_back();
      component.push_back(v);
    } while (u != v);
    bcc.push_back(component);
  }
}

void tarjan(int nodes) {
  bcc.clear();
  bridges.clear();
  cutpoints.clear();
  stack.clear();
  fill(lowlink, lowlink + nodes, 0);
  fill(tin, tin + nodes, 0);
  fill(vis.begin(), vis.end(), false);
  timer = 0;
  for (int i = 0; i < nodes; i++)
    if (!vis[i]) dfs(i, -1);
}

//condenses each bcc to a node and generates a tree
//global variables adj and bcc must be set beforehand
void get_block_tree(int nodes) {
  fill(comp, comp + nodes, 0);
  for (int i = 0; i < nodes; i++) bcc_forest[i].clear();
  for (int i = 0; i < (int)bcc.size(); i++)
    for (int j = 0; j < (int)bcc[i].size(); j++)
      comp[bcc[i][j]] = i;
  for (int i = 0; i < nodes; i++)
    for (int j = 0; j < (int)adj[i].size(); j++)
      if (comp[i] != comp[adj[i][j]])
        bcc_forest[comp[i]].push_back(comp[adj[i][j]]);
}

int main() {
  int nodes, edges, u, v;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
  tarjan(nodes);
  cout << "Cut-points:";
  for (int i = 0; i < (int)cutpoints.size(); i++)
    cout << "␣" << cutpoints[i];
  cout << "\nBridges:\n";
  for (int i = 0; i < (int)bridges.size(); i++)
    cout << bridges[i].first << "␣" << bridges[i].second << "\n";
  cout << "Edge-Biconnected␣Components:\n";
  for (int i = 0; i < (int)bcc.size(); i++) {
    cout << "Component:";
    for (int j = 0; j < (int)bcc[i].size(); j++)
      cout << "␣" << bcc[i][j];
    cout << "\n";
  }
  get_block_tree(nodes);
  cout << "Adjacency␣List␣for␣Block␣Forest:\n";
  for (int i = 0; i < (int)bcc.size(); i++) {
    cout << i << "␣=>";
    for (int j = 0; j < (int)bcc_forest[i].size(); j++)
      cout << "␣" << bcc_forest[i][j];
    cout << "\n";
  }
  return 0;
}
```

2.4 Minimal Spanning Trees

2.4.1 Prim's Algorithm

```
/*

Description: Given an undirected graph, its minimum spanning
tree (MST) is a tree connecting all nodes with a subset of its
edges such that their total weight is minimized. Prim's algorithm
greedily selects edges from a priority queue, and is similar to
Dijkstra's algorithm, where instead of processing nodes, we
process individual edges. If the graph is not connected, Prim's
algorithm will produce the minimum spanning forest. The input
graph is stored in an adjacency list.

Note that the concept of the minimum spanning tree makes Prim's
algorithm work with negative weights. In fact, a big positive
constant added to all of the edge weights of the graph will not
change the resulting spanning tree.

Implementation Notes: Similar to the implementation of Dijkstra's
algorithm in the previous section, weights are negated before they
are added to the priority queue (and negated once again when they
are retrieved). To find the maximum spanning tree, simply skip the
two negation steps and the max weighted edges will be prioritized.

Complexity: This version uses an adjacency list and priority queue
(internally a binary heap) and has a complexity of O((E+V) log V) =
O(E log V). The priority queue and adjacency list improves the
simplest O(V^2) version of the algorithm, which uses looping and
an adjacency matrix. If the priority queue is implemented as a more
sophisticated Fibonacci heap, the complexity becomes O(E + V log V).

=~=~=~=~= Sample Input =~=~=~=~=
7 7
0 1 4
1 2 6
2 0 3
3 4 1
4 5 2
5 6 3
6 4 4

=~=~=~=~= Sample Output =~=~=~=~=
Total distance: 13
0<->2
```

```
0<->1
3<->4
4<->5
5<->6

*/

#include <algorithm> /* std::fill() */
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

const int MAXN = 100;
vector<pair<int, int> > adj[MAXN], mst;

int prim(int nodes) {
  mst.clear();
  vector<bool> vis(nodes);
  int u, v, w, total_dist = 0;
  for (int i = 0; i < nodes; i++) {
    if (vis[i]) continue;
    vis[i] = true;
    priority_queue<pair<int, pair<int, int> > > pq;
    for (int j = 0; j < (int)adj[i].size(); j++)
      pq.push(make_pair(-adj[i][j].second,
                make_pair(i, adj[i][j].first)));
    while (!pq.empty()) {
      w = -pq.top().first;
      u = pq.top().second.first;
      v = pq.top().second.second;
      pq.pop();
      if (vis[u] && !vis[v]) {
        vis[v] = true;
        if (v != i) {
          mst.push_back(make_pair(u, v));
          total_dist += w;
        }
        for (int j = 0; j < (int)adj[v].size(); j++)
          pq.push(make_pair(-adj[v][j].second,
                    make_pair(v, adj[v][j].first)));
      }
    }
  }
  return total_dist;
}

int main() {
  int nodes, edges, u, v, w;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v >> w;
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
  }
  cout << "Total␣distance:␣" << prim(nodes) << "\n";
  for (int i = 0; i < (int)mst.size(); i++)
    cout << mst[i].first << "<->" << mst[i].second << "\n";
  return 0;
}
```

2.4.2 Kruskal's Algorithm

```
/*

Description: Given an undirected graph, its minimum spanning
tree (MST) is a tree connecting all nodes with a subset of its
edges such that their total weight is minimized. If the graph
is not connected, Kruskal's algorithm will produce the minimum
spanning forest. The input graph is stored in an edge list.

Complexity: O(E log V) on the number of edges and vertices.

=~=~=~=~= Sample Input =~=~=~=~=
7 7
0 1 4
1 2 6
2 0 3
3 4 1
4 5 2
5 6 3
6 4 4

=~=~=~=~= Sample Output =~=~=~=~=
Total distance: 13
3<->4
4<->5
2<->0
5<->6
0<->1

Note: If you already have a disjoint set data structure,
then the middle section of the program can be replaced by:

disjoint_set_forest<int> dsf;
for (int i = 0; i < nodes; i++) dsf.make_set(i);
for (int i = 0; i < E.size(); i++) {
  a = E[i].second.first;
  b = E[i].second.second;
  if (!dsf.is_united(a, b)) {
    ...
    dsf.unite(a, b);
  }
}

*/

#include <algorithm> /* std::sort() */
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 100;
int root[MAXN];
vector<pair<int, pair<int, int> > > E;
vector<pair<int, int> > mst;
```

```

int find_root(int x) {
  if (root[x] != x)
    root[x] = find_root(root[x]);
  return root[x];
}

int kruskal(int nodes) {
  mst.clear();
  sort(E.begin(), E.end());
  int u, v, total_dist = 0;
  for (int i = 0; i < nodes; i++) root[i] = i;
  for (int i = 0; i < (int)E.size(); i++) {
    u = find_root(E[i].second.first);
    v = find_root(E[i].second.second);
    if (u != v) {
      mst.push_back(E[i].second);
      total_dist += E[i].first;
      root[u] = root[v];
    }
  }
  return total_dist;
}

int main() {
  int nodes, edges, u, v, w;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v >> w;
    E.push_back(make_pair(w, make_pair(u, v)));
  }
  cout << "Total␣distance:␣" << kruskal(nodes) << "\n";
  for (int i = 0; i < (int)mst.size(); i++)
    cout << mst[i].first << "<->" << mst[i].second << "\n";
  return 0;
}
```

2.5 Maximum Flow

2.5.1 Ford-Fulkerson Algorithm

```
/*

Description: Given a flow network, find a flow from a single
source node to a single sink node that is maximized. Note
that in this implementation, the adjacency matrix cap[][]
will be modified by the function ford_fulkerson() after it's
been called. Make a back-up if you require it afterwards.

Complexity: O(V^2*|F|), where V is the number of
vertices and |F| is the magnitude of the max flow.

Real-valued capacities:
The Ford-Fulkerson algorithm is only optimal on graphs with
integer capacities; there exists certain real capacity inputs
for which it will never terminate. The Edmonds-Karp algorithm
```

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

=~=~=~=~= Sample Input =~=~=~=~=
6 8
0 1 3
0 2 3
1 2 2
1 3 3
2 4 2
3 4 1
3 5 2
4 5 3
0 5

=~=~=~=~= Sample Output =~=~=~=~=
5

*/

#include <algorithm> /* std::fill() */
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 100, INF = 0x3f3f3f3f;
int nodes, source, sink, cap[MAXN][MAXN];
vector<bool> vis(MAXN);

int dfs(int u, int f) {
  if (u == sink) return f;
  vis[u] = true;
  for (int v = 0; v < nodes; v++) {
    if (!vis[v] && cap[u][v] > 0) {
      int df = dfs(v, min(f, cap[u][v]));
      if (df > 0) {
        cap[u][v] -= df;
        cap[v][u] += df;
        return df;
      }
    }
  }
  return 0;
}

int ford_fulkerson() {
  int max_flow = 0;
  for (;;) {
    fill(vis.begin(), vis.end(), false);
    int df = dfs(source, INF);
    if (df == 0) break;
    max_flow += df;
  }
  return max_flow;
}

int main() {
  int edges, u, v, capacity;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v >> capacity;
    cap[u][v] = capacity;
  }
  cin >> source >> sink;
  cout << ford_fulkerson() << "\n";
  return 0;
}
```

2.5.2 Edmonds-Karp Algorithm

```
/*

Description: Given a flow network, find a flow from a single
source node to a single sink node that is maximized. Note
that in this implementation, the adjacency list adj[] will
be modified by the function edmonds_karp() after it's been called.

Complexity: O(min(V*E^2, E*|F|)), where V is the number of
vertices, E is the number of edges, and |F| is the magnitude of
the max flow. This improves the original Ford-Fulkerson algorithm,
which runs in O(E*|F|). As the Edmonds-Karp algorithm is also
bounded by O(E*|F|), it is guaranteed to be at least as fast as
Ford-Fulkerson. For an even faster algorithm, see Dinic's
algorithm in the next section, which runs in O(V^2*E).

Real-valued capacities:
Although the Ford-Fulkerson algorithm is only optimal on graphs
with integer capacities, the Edmonds-Karp algorithm also works
correctly on real-valued capacities.

=~=~=~= Sample Input =~=~=~=
6 8
0 1 3
0 2 3
1 2 2
1 3 3
2 4 2
3 4 1
3 5 2
4 5 3
0 5

=~=~=~= Sample Output =~=~=~=
5

*/

#include <algorithm> /* std::fill(), std::min() */
#include <iostream>
#include <vector>
using namespace std;

struct edge { int s, t, rev, cap, f; };

const int MAXN = 100, INF = 0x3f3f3f3f;
vector<edge> adj[MAXN];

```

```
void add_edge(int s, int t, int cap) {
  adj[s].push_back((edge){s, t, (int)adj[t].size(), cap, 0});
  adj[t].push_back((edge){t, s, (int)adj[s].size() - 1, 0, 0});
}

int edmonds_karp(int nodes, int source, int sink) {
  static int q[MAXN];
  int max_flow = 0;
  for (;;) {
    int qt = 0;
    q[qt++] = source;
    edge * pred[nodes];
    fill(pred, pred + nodes, (edge*)0);
    for (int qh = 0; qh < qt && !pred[sink]; qh++) {
      int u = q[qh];
      for (int j = 0; j < (int)adj[u].size(); j++) {
        edge * e = &adj[u][j];
        if (!pred[e->t] && e->cap > e->f) {
          pred[e->t] = e;
          q[qt++] = e->t;
        }
      }
    }
    if (!pred[sink]) break;
    int df = INF;
    for (int u = sink; u != source; u = pred[u]->s)
      df = min(df, pred[u]->cap - pred[u]->f);
    for (int u = sink; u != source; u = pred[u]->s) {
      pred[u]->f += df;
      adj[pred[u]->t][pred[u]->rev].f -= df;
    }
    max_flow += df;
  }
  return max_flow;
}

int main() {
  int nodes, edges, u, v, capacity, source, sink;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v >> capacity;
    add_edge(u, v, capacity);
  }
  cin >> source >> sink;
  cout << edmonds_karp(nodes, source, sink) << "\n";
  return 0;
}
```

2.5.3 Dinic's Algorithm

```
/*

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

```

```
Complexity: O(V^2*E) on the number of vertices and edges.

Comparison with Edmonds-Karp Algorithm:
Dinic's is similar to the Edmonds-Karp algorithm in that it
uses the shortest augmenting path. The introduction of the
concepts of the level graph and blocking flow enable Dinic's
algorithm to achieve its better performance. Hence, Dinic's
algorithm is also called Dinic's blocking flow algorithm.

=~=~=~=~= Sample Input =~=~=~=~=
6 8
0 1 3
0 2 3
1 2 2
1 3 3
2 4 2
3 4 1
3 5 2
4 5 3
0 5

=~=~=~=~= Sample Output =~=~=~=~=
5

*/

#include <algorithm> /* std::fill(), std::min() */
#include <iostream>
#include <vector>
using namespace std;

struct edge { int to, rev, cap, f; };

const int MAXN = 100, INF = 0x3f3f3f3f;
int dist[MAXN], ptr[MAXN];
vector<edge> adj[MAXN];

void add_edge(int s, int t, int cap) {
  adj[s].push_back((edge){t, (int)adj[t].size(), cap, 0});
  adj[t].push_back((edge){s, (int)adj[s].size() - 1, 0, 0});
}

bool dinic_bfs(int nodes, int source, int sink) {
  fill(dist, dist + nodes, -1);
  dist[source] = 0;
  int q[nodes], qh = 0, qt = 0;
  q[qt++] = source;
  while (qh < qt) {
    int u = q[qh++];
    for (int j = 0; j < (int)adj[u].size(); j++) {
      edge & e = adj[u][j];
      if (dist[e.to] < 0 && e.f < e.cap) {
        dist[e.to] = dist[u] + 1;
        q[qt++] = e.to;
      }
    }
  }
  return dist[sink] >= 0;
}
```

```

int dinic_dfs(int u, int f, int sink) {
  if (u == sink) return f;
  for (; ptr[u] < (int)adj[u].size(); ptr[u]++) {
    edge &e = adj[u][ptr[u]];
    if (dist[e.to] == dist[u] + 1 && e.f < e.cap) {
      int df = dinic_dfs(e.to, min(f, e.cap - e.f), sink);
      if (df > 0) {
        e.f += df;
        adj[e.to][e.rev].f -= df;
        return df;
      }
    }
  }
  return 0;
}

int dinic(int nodes, int source, int sink) {
  int max_flow = 0, delta;
  while (dinic_bfs(nodes, source, sink)) {
    fill(ptr, ptr + nodes, 0);
    while ((delta = dinic_dfs(source, INF, sink)) != 0)
      max_flow += delta;
  }
  return max_flow;
}

int main() {
  int nodes, edges, u, v, capacity, source, sink;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v >> capacity;
    add_edge(u, v, capacity);
  }
  cin >> source >> sink;
  cout << dinic(nodes, source, sink) << "\n";
  return 0;
}
```

2.5.4 Push-Relabel Algorithm

```
/*

Description: Given a flow network, find a flow from a single
source node to a single sink node that is maximized. The push-
relabel algorithm is considered one of the most efficient
maximum flow algorithms. However, unlike the Ford-Fulkerson or
Edmonds-Karp algorithms, it cannot take advantage of the fact
if max flow itself has a small magnitude.

Complexity: O(V^3) on the number of vertices.

=~=~=~=~= Sample Input =~=~=~=~=
6 8
0 1 3
0 2 3
1 2 2
```

```
1 3 3
2 4 2
3 4 1
3 5 2
4 5 3
0 5

=~=~=~=~= Sample Output =~=~=~=~=
5

*/

#include <algorithm> /* std::fill(), std::min() */
#include <iostream>
using namespace std;

const int MAXN = 100, INF = 0x3F3F3F3F;
int cap[MAXN][MAXN], f[MAXN][MAXN];

int push_relabel(int nodes, int source, int sink) {
  int e[nodes], h[nodes], maxh[nodes];
  fill(e, e + nodes, 0);
  fill(h, h + nodes, 0);
  fill(maxh, maxh + nodes, 0);
  for (int i = 0; i < nodes; i++)
    fill(f[i], f[i] + nodes, 0);
  h[source] = nodes - 1;
  for (int i = 0; i < nodes; i++) {
    f[source][i] = cap[source][i];
    f[i][source] = -f[source][i];
    e[i] = cap[source][i];
  }
  int sz = 0;
  for (;;) {
    if (sz == 0) {
      for (int i = 0; i < nodes; i++)
        if (i != source && i != sink && e[i] > 0) {
          if (sz != 0 && h[i] > h[maxh[0]]) sz = 0;
          maxh[sz++] = i;
        }
    }
    if (sz == 0) break;
    while (sz != 0) {
      int i = maxh[sz - 1];
      bool pushed = false;
      for (int j = 0; j < nodes && e[i] != 0; j++) {
        if (h[i] == h[j] + 1 && cap[i][j] - f[i][j] > 0) {
          int df = min(cap[i][j] - f[i][j], e[i]);
          f[i][j] += df;
          f[j][i] -= df;
          e[i] -= df;
          e[j] += df;
          if (e[i] == 0) sz--;
          pushed = true;
        }
      }
      if (!pushed) {
        h[i] = INF;
        for (int j = 0; j < nodes; j++)
```

```
          if (h[i] > h[j] + 1 && cap[i][j] - f[i][j] > 0)
            h[i] = h[j] + 1;
        if (h[i] > h[maxh[0]]) {
          sz = 0;
          break;
        }
      }
    }
  }
  int max_flow = 0;
  for (int i = 0; i < nodes; i++)
    max_flow += f[source][i];
  return max_flow;
}

int main() {
  int nodes, edges, u, v, capacity, source, sink;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v >> capacity;
    cap[u][v] = capacity;
  }
  cin >> source >> sink;
  cout << push_relabel(nodes, source, sink) << "\n";
  return 0;
}
```

2.6 Backtracking

2.6.1 Max Clique (Bron-Kerbosch Algorithm)

```
/*

Description: Given an undirected graph, determine a subset of
the graph's vertices such that every pair of vertices in the
subset are connected by an edge, and that the subset is as
large as possible. For the weighted version, each vertex is
assigned a weight and the objective is to find the clique in
the graph that has maximum total weight.

Complexity: O(3^(V/3)) where V is the number of vertices.

=~=~=~=~= Sample Input =~=~=~=~=
5 8
0 1
0 2
0 3
1 2
1 3
2 3
3 4
4 2
10 20 30 40 50

=~=~=~=~= Sample Output =~=~=~=~=
Max unweighted clique: 4
Max weighted clique: 120

*/

#include <algorithm> /* std::fill(), std::max() */
#include <bitset>
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 35;
typedef bitset<MAXN> bits;
typedef unsigned long long ull;

int w[MAXN];
bool adj[MAXN][MAXN];

int rec(int nodes, bits & curr, bits & pool, bits & excl) {
  if (pool.none() && excl.none()) return curr.count();
  int ans = 0, u = 0;
  for (int v = 0; v < nodes; v++)
    if (pool[v] || excl[v]) u = v;
  for (int v = 0; v < nodes; v++) {
    if (!pool[v] || adj[u][v]) continue;
    bits ncurr, npool, nexcl;
    for (int i = 0; i < nodes; i++) ncurr[i] = curr[i];
    ncurr[v] = true;
    for (int j = 0; j < nodes; j++) {
      npool[j] = pool[j] && adj[v][j];
      nexcl[j] = excl[j] && adj[v][j];
    }
    ans = max(ans, rec(nodes, ncurr, npool, nexcl));
    pool[v] = false;
    excl[v] = true;
  }
  return ans;
}

int bron_kerbosch(int nodes) {
  bits curr, excl, pool;
  pool.flip();
  return rec(nodes, curr, pool, excl);
}

//This is a fast implementation using bitmasks.
//Precondition: the number of nodes must be less than 64.
int bron_kerbosch_weighted(int nodes, ull g[], ull curr, ull pool, ull excl) {
  if (pool == 0 && excl == 0) {
    int res = 0, u = __builtin_ctzll(curr);
    while (u < nodes) {
      res += w[u];
      u += __builtin_ctzll(curr >> (u + 1)) + 1;
    }
    return res;
  }
  if (pool == 0) return -1;
  int res = -1, pivot = __builtin_ctzll(pool | excl);
  ull z = pool & ~g[pivot];
  int u = __builtin_ctzll(z);
  while (u < nodes) {
    res = max(res, bron_kerbosch_weighted(nodes, g, curr | (1LL << u),
                                          pool & g[u], excl & g[u]));
    pool ^= 1LL << u;
    excl |= 1LL << u;
    u += __builtin_ctzll(z >> (u + 1)) + 1;
  }
  return res;
}

int bron_kerbosch_weighted(int nodes) {
  ull g[nodes];
  for (int i = 0; i < nodes; i++) {
    g[i] = 0;
    for (int j = 0; j < nodes; j++)
      if (adj[i][j]) g[i] |= 1LL << j;
  }
  return bron_kerbosch_weighted(nodes, g, 0, (1LL << nodes) - 1, 0);
}

int main() {
  int nodes, edges, u, v;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v;
    adj[u][v] = adj[v][u] = true;
  }
  for (int i = 0; i < nodes; i++) cin >> w[i];
  cout << "Max␣unweighted␣clique:␣";
  cout << bron_kerbosch(nodes) << "\n";
  cout << "Max␣weighted␣clique:␣";
  cout << bron_kerbosch_weighted(nodes) << "\n";
  return 0;
}
```

2.6.2 Graph Coloring

```
/*

Description: Given an undirected graph, assign colors to each
of the vertices such that no pair of adjacent vertices have the
same color. Furthermore, do so using the minimum # of colors.

Complexity: Exponential on the number of vertices. The exact
running time is difficult to calculate due to several pruning
optimizations used here.

=~=~=~=~= Sample Input =~=~=~=~=
5 7
0 1
0 4
1 3
1 4
2 3
2 4
3 4

```

```
=~=~=~=~= Sample Output =~=~=~=~=
Colored using 3 color(s). The colorings are:
Color 1: 0 3
Color 2: 1 2
Color 3: 4

*/

#include <algorithm> /* std::fill(), std::max() */
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 30;
int cols[MAXN], adj[MAXN][MAXN];
int id[MAXN + 1], deg[MAXN + 1];
int min_cols, best_cols[MAXN];

void dfs(int from, int to, int cur, int used_cols) {
  if (used_cols >= min_cols) return;
  if (cur == to) {
    for (int i = from; i < to; i++)
      best_cols[id[i]] = cols[i];
    min_cols = used_cols;
    return;
  }
  vector<bool> used(used_cols + 1);
  for (int i = 0; i < cur; i++)
    if (adj[id[cur]][id[i]]) used[cols[i]] = true;
  for (int i = 0; i <= used_cols; i++) {
    if (!used[i]) {
      int tmp = cols[cur];
      cols[cur] = i;
      dfs(from, to, cur + 1, max(used_cols, i + 1));
      cols[cur] = tmp;
    }
  }
}

int color_graph(int nodes) {
  for (int i = 0; i <= nodes; i++) {
    id[i] = i;
    deg[i] = 0;
  }
  int res = 1;
  for (int from = 0, to = 1; to <= nodes; to++) {
    int best = to;
    for (int i = to; i < nodes; i++) {
      if (adj[id[to - 1]][id[i]]) deg[id[i]]++;
      if (deg[id[best]] < deg[id[i]]) best = i;
    }
    int tmp = id[to];
    id[to] = id[best];
    id[best] = tmp;
    if (deg[id[to]] == 0) {
      min_cols = nodes + 1;
      fill(cols, cols + nodes, 0);
      dfs(from, to, from, 0);
      from = to;
```

```cpp
      res = max(res, min_cols);
    }
  }
  return res;
}

int main() {
  int nodes, edges, u, v;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v;
    adj[u][v] = adj[v][u] = true;
  }
  cout << "Colored␣using␣" << color_graph(nodes);
  cout << "␣color(s).␣The␣colorings␣are:\n";
  for (int i = 0; i < min_cols; i++) {
    cout << "Color␣" << i + 1 << ":";
    for (int j = 0; j < nodes; j++)
      if (best_cols[j] == i) cout << "␣" << j;
    cout << "\n";
  }
  return 0;
}
```

2.7 Maximum Matching

2.7.1 Maximum Bipartite Matching (Kuhn's Algorithm)

```cpp
/*

Description: Given two sets of vertices A = {0, 1, ..., n1}
and B = {0, 1, ..., n2} as well as a set of edges E mapping
nodes from set A to set B, determine the largest possible
subset of E such that no pair of edges in the subset share
a common vertex. Precondition: n2 >= n1.

Complexity: O(V*E) on the number of vertices and edges.

=~=~=~=~= Sample Input =~=~=~=~=
3 4 6
0 1
1 0
1 1
1 2
2 2
2 3

=~=~=~=~= Sample Output =~=~=~=~=
Matched 3 pairs. Matchings are:
1 0
0 1
2 2

*/

#include <algorithm> /* std::fill() */
```

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 100;
int match[MAXN];
vector<bool> vis(MAXN);
vector<int> adj[MAXN];

bool dfs(int u) {
  vis[u] = true;
  for (int j = 0; j < (int)adj[u].size(); j++) {
    int v = match[adj[u][j]];
    if (v == -1 || (!vis[v] && dfs(v))) {
      match[adj[u][j]] = u;
      return true;
    }
  }
  return false;
}

int kuhn(int n1, int n2) {
  fill(vis.begin(), vis.end(), false);
  fill(match, match + n2, -1);
  int matches = 0;
  for (int i = 0; i < n1; i++) {
    for (int j = 0; j < n1; j++) vis[j] = 0;
    if (dfs(i)) matches++;
  }
  return matches;
}

int main() {
  int n1, n2, edges, u, v;
  cin >> n1 >> n2 >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v;
    adj[u].push_back(v);
  }
  cout << "Matched␣" << kuhn(n1, n2);
  cout << "␣pair(s).␣Matchings␣are:\n";
  for (int i = 0; i < n2; i++) {
    if (match[i] == -1) continue;
    cout << match[i] << "␣" << i << "\n";
  }
  return 0;
}
```

2.7.2 Maximum Bipartite Matching (Hopcroft-Karp Algorithm)

```cpp
/*

Description: Given two sets of vertices A = {0, 1, ..., n1}
and B = {0, 1, ..., n2} as well as a set of edges E mapping
nodes from set A to set B, determine the largest possible
subset of E such that no pair of edges in the subset share
a common vertex. Precondition: n2 >= n1.
```

```

Complexity: O(E sqrt V) on the number of edges and vertices.

=~=~=~=~= Sample Input =~=~=~=~=
3 4 6
0 1
1 0
1 1
1 2
2 2
2 3

=~=~=~=~= Sample Output =~=~=~=~=
Matched 3 pairs. Matchings are:
1 0
0 1
2 2

*/

#include <algorithm> /* std::fill() */
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 100;
int match[MAXN], dist[MAXN];
vector<bool> used(MAXN), vis(MAXN);
vector<int> adj[MAXN];

void bfs(int n1, int n2) {
  fill(dist, dist + n1, -1);
  int q[n2], qb = 0;
  for (int u = 0; u < n1; ++u) {
    if (!used[u]) {
      q[qb++] = u;
      dist[u] = 0;
    }
  }
  for (int i = 0; i < qb; i++) {
    int u = q[i];
    for (int j = 0; j < (int)adj[u].size(); j++) {
      int v = match[adj[u][j]];
      if (v >= 0 && dist[v] < 0) {
        dist[v] = dist[u] + 1;
        q[qb++] = v;
      }
    }
  }
}

bool dfs(int u) {
  vis[u] = true;
  for (int j = 0; j < (int)adj[u].size(); j++) {
    int v = match[adj[u][j]];
    if (v < 0 || (!vis[v] && dist[v] == dist[u] + 1 && dfs(v))) {
      match[adj[u][j]] = u;
      used[u] = true;
      return true;
```

```cpp
    }
  }
  return false;
}

int hopcroft_karp(int n1, int n2) {
  fill(match, match + n2, -1);
  fill(used.begin(), used.end(), false);
  int res = 0;
  for (;;) {
    bfs(n1, n2);
    fill(vis.begin(), vis.end(), false);
    int f = 0;
    for (int u = 0; u < n1; ++u)
      if (!used[u] && dfs(u)) f++;
    if (!f) return res;
    res += f;
  }
  return res;
}

int main() {
  int n1, n2, edges, u, v;
  cin >> n1 >> n2 >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v;
    adj[u].push_back(v);
  }
  cout << "Matched " << hopcroft_karp(n1, n2);
  cout << " pair(s). Matchings are:\n";
  for (int i = 0; i < n2; i++) {
    if (match[i] == -1) continue;
    cout << match[i] << " " << i << "\n";
  }
  return 0;
}
```

2.7.3 Maximum Graph Matching (Edmonds's Algorithm)

```cpp
/*

Description: Given a general directed graph, determine a maximal
subset of the edges such that no vertex is repeated in the subset.

Complexity: O(V^3) on the number of vertices.

=~=~=~=~= Sample Input =~=~=~=~=
4 8
0 1
1 0
1 2
2 1
2 3
3 2
3 0
0 3

```

```
=~=~=~=~= Sample Output =~=~=~=~=
Matched 2 pair(s). Matchings are:
0 1
2 3

*/

#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 100;
int p[MAXN], base[MAXN], match[MAXN];
vector<int> adj[MAXN];

int lca(int nodes, int a, int b) {
  vector<bool> used(nodes);
  for (;;) {
    a = base[a];
    used[a] = true;
    if (match[a] == -1) break;
    a = p[match[a]];
  }
  for (;;) {
    b = base[b];
    if (used[b]) return b;
    b = p[match[b]];
  }
}

void mark_path(vector<bool> & blossom, int v, int b, int children) {
  for (; base[v] != b; v = p[match[v]]) {
    blossom[base[v]] = blossom[base[match[v]]] = true;
    p[v] = children;
    children = match[v];
  }
}

int find_path(int nodes, int root) {
  vector<bool> used(nodes);
  for (int i = 0; i < nodes; ++i) {
    p[i] = -1;
    base[i] = i;
  }
  used[root] = true;
  int q[nodes], qh = 0, qt = 0;
  q[qt++] = root;
  while (qh < qt) {
    int v = q[qh++];
    for (int j = 0, to; j < (int)adj[v].size(); j++) {
      to = adj[v][j];
      if (base[v] == base[to] || match[v] == to) continue;
      if (to == root || (match[to] != -1 && p[match[to]] != -1)) {
        int curbase = lca(nodes, v, to);
        vector<bool> blossom(nodes);
        mark_path(blossom, v, curbase, to);
        mark_path(blossom, to, curbase, v);
        for (int i = 0; i < nodes; i++)
          if (blossom[base[i]]) {
            base[i] = curbase;
            if (!used[i]) {
              used[i] = true;
              q[qt++] = i;
            }
          }
      } else if (p[to] == -1) {
        p[to] = v;
        if (match[to] == -1) return to;
        to = match[to];
        used[to] = true;
        q[qt++] = to;
      }
    }
  }
  return -1;
}

int edmonds(int nodes) {
  for (int i = 0; i < nodes; i++) match[i] = -1;
  for (int i = 0; i < nodes; i++) {
    if (match[i] == -1) {
      int v, pv, ppv;
      for (v = find_path(nodes, i); v != -1; v = ppv) {
        ppv = match[pv = p[v]];
        match[v] = pv;
        match[pv] = v;
      }
    }
  }
  int matches = 0;
  for (int i = 0; i < nodes; i++)
    if (match[i] != -1) matches++;
  return matches / 2;
}

int main() {
  int nodes, edges, u, v;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v;
    adj[u].push_back(v);
  }
  cout << "Matched␣" << edmonds(nodes);
  cout << "␣pair(s).␣Matchings␣are:\n";
  for (int i = 0; i < nodes; i++) {
    if (match[i] != -1 && i < match[i])
      cout << i << "␣" << match[i] << "\n";
  }
  return 0;
}
```

2.8 Hamiltonian Path and Cycle

2.8.1 Shortest Hamiltonian Cycle (Travelling Salesman)

```
/*

Description: Given a weighted, directed graph, the shortest
hamiltonian cycle is a cycle of minimum distance that visits
each vertex exactly once and returns to the original vertex.
This is also known as the traveling salesman problem (TSP).
Since this is a bitmasking solution with 32-bit integers,
the number of vertices must be less than 32.

Complexity: O(2^V * V^2) on the number of vertices.

=~=~=~=~= Sample Input =~=~=~=~=
5 10
0 1 1
0 2 10
0 3 1
0 4 10
1 2 10
1 3 10
1 4 1
2 3 1
2 4 1
3 4 10

=~=~=~=~= Sample Output =~=~=~=~=
The shortest hamiltonian cycle has length 5.
Take the path: 0->3->2->4->1->0

*/

#include <algorithm> /* std::fill(), std::min() */
#include <iostream>
using namespace std;

const int MAXN = 20, INF = 0x3f3f3f3f;
int adj[MAXN][MAXN], order[MAXN];

int shortest_hamiltonian_cycle(int nodes) {
  int dp[1 << nodes][nodes];
  for (int i = 0; i < (1 << nodes); i++)
    fill(dp[i], dp[i] + nodes, INF);
  dp[1][0] = 0;
  for (int mask = 1; mask < (1 << nodes); mask += 2) {
    for (int i = 1; i < nodes; i++)
      if ((mask & 1 << i) != 0)
        for (int j = 0; j < nodes; j++)
          if ((mask & 1 << j) != 0)
            dp[mask][i] = min(dp[mask][i], dp[mask ^ (1 << i)][j] + adj[j][i]);
  }
  int res = INF + INF;
  for (int i = 1; i < nodes; i++)
    res = min(res, dp[(1 << nodes) - 1][i] + adj[i][0]);
  int cur = (1 << nodes) - 1, last = 0;
  for (int i = nodes - 1; i >= 1; i--) {
    int bj = -1;
    for (int j = 1; j < nodes; j++) {
      if ((cur & 1 << j) != 0 && (bj == -1 ||
            dp[cur][bj] + adj[bj][last] > dp[cur][j] + adj[j][last])) {
        bj = j;
      }
    }
    order[i] = bj;
    cur ^= 1 << bj;
    last = bj;
  }
  return res;
}

int main() {
  int nodes, edges, u, v, w;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v >> w;
    adj[u][v] = adj[v][u] = w; //only set adj[u][v] if directed edges
  }
  cout << "The␣shortest␣hamiltonian␣cycle␣has␣length␣";
  cout << shortest_hamiltonian_cycle(nodes) << ".\n";
  cout << "Take␣the␣path:␣";
  for (int i = 0; i < nodes; i++) cout << order[i] << "->";
  cout << order[0] << "\n";
  return 0;
}
```

2.8.2 Shortest Hamiltonian Path

```
/*

Description: Given a weighted, directed graph, the shortest
hamiltonian path is a path of minimum distance that visits
each vertex exactly once. Unlike the travelling salesman
problem, we don't have to return to the starting vertex.
Since this is a bitmasking solution with 32-bit integers,
the number of vertices must be less than 32.

Complexity: O(2^V * V^2) on the number of vertices.

=~=~=~=~= Sample Input =~=~=~=~=
3 6
0 1 1
0 2 1
1 0 7
1 2 2
2 0 3
2 1 5

=~=~=~=~= Sample Output =~=~=~=~=
The shortest hamiltonian path has length 3.
Take the path: 0->1->2

*/

#include <algorithm> /* std::fill(), std::min() */
#include <iostream>
using namespace std;

const int MAXN = 20, INF = 0x3f3f3f3f;
```

```

int adj[MAXN][MAXN], order[MAXN];

int shortest_hamiltonian_path(int nodes) {
  int dp[1 << nodes][nodes];
  for (int i = 0; i < (1 << nodes); i++)
    fill(dp[i], dp[i] + nodes, INF);
  for (int i = 0; i < nodes; i++) dp[1 << i][i] = 0;
  for (int mask = 1; mask < (1 << nodes); mask += 2) {
    for (int i = 0; i < nodes; i++)
      if ((mask & 1 << i) != 0)
        for (int j = 0; j < nodes; j++)
          if ((mask & 1 << j) != 0)
            dp[mask][i] = min(dp[mask][i], dp[mask ^ (1 << i)][j] + adj[j][i]);
  }
  int res = INF + INF;
  for (int i = 1; i < nodes; i++)
    res = min(res, dp[(1 << nodes) - 1][i]);
  int cur = (1 << nodes) - 1, last = -1;
  for (int i = nodes - 1; i >= 0; i--) {
    int bj = -1;
    for (int j = 0; j < nodes; j++) {
      if ((cur & 1 << j) != 0 && (bj == -1 ||
           dp[cur][bj] + (last == -1 ? 0 : adj[bj][last]) >
           dp[cur][j]  + (last == -1 ? 0 : adj[j][last]))) {
        bj = j;
      }
    }
    order[i] = bj;
    cur ^= 1 << bj;
    last = bj;
  }
  return res;
}

int main() {
  int nodes, edges, u, v, w;
  cin >> nodes >> edges;
  for (int i = 0; i < edges; i++) {
    cin >> u >> v >> w;
    adj[u][v] = w;
  }
  cout << "The shortest hamiltonian path has length ";
  cout << shortest_hamiltonian_path(nodes) << ".\n";
  cout << "Take the path: " << order[0];
  for (int i = 1; i < nodes; i++) cout << "->" << order[i];
  return 0;
}
```

Chapter 3

Data Structures

3.1 Disjoint Sets

3.1.1 Disjoint Set Forest (Simple)

```
/*

Description: This data structure dynamically keeps track
of items partitioned into non-overlapping sets (a disjoint
set forest). It is also known as a union-find data structure.

Time Complexity: Every function below is O(a(N)) amortized
on the number of items in the set due to the optimizations
of union by rank and path compression. Here, a(N) is the
extremely slow growing inverse of the Ackermann function.
For all practical values of n, a(n) is less than 5.

Space Complexity: O(N) total.

*/

const int MAXN = 1000;
int num_sets = 0, root[MAXN+1], rank[MAXN+1];

int find_root(int x) {
  if (root[x] != x) root[x] = find_root(root[x]);
  return root[x];
}

void make_set(int x) {
  root[x] = x;
  rank[x] = 0;
  num_sets++;
}

bool is_united(int x, int y) {
  return find_root(x) == find_root(y);
}

void unite(int x, int y) {
  int X = find_root(x), Y = find_root(y);
  if (X == Y) return;
  num_sets--;
  if (rank[X] < rank[Y]) root[X] = Y;
  else if (rank[X] > rank[Y]) root[Y] = X;
  else rank[root[Y] = X]++;
}

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;

int main() {
  for (char c = 'a'; c <= 'g'; c++) make_set(c);
  unite('a', 'b');
  unite('b', 'f');
  unite('d', 'e');
  unite('e', 'g');
  assert(num_sets == 3);
  assert(is_united('a', 'b'));
  assert(!is_united('a', 'c'));
  assert(!is_united('b', 'g'));
  assert(is_united('d', 'g'));
  return 0;
}
```

3.1.2 Disjoint Set Forest

```
/*

Description: This data structure dynamically keeps track
of items partitioned into non-overlapping sets (a disjoint
set forest). It is also known as a union-find data structure.
This particular templatized version employs an std::map for
built in storage and coordinate compression. That is, the
magnitude of values inserted is not limited.

Time Complexity: make_set(), unite() and is_united() are
O(a(N) + log N) = O(log N) on the number of elements in the
disjoint set forest. get_all_sets() is O(N). find() is is
O(a(N)) amortized on the number of items in the set due to
the optimizations of union by rank and path compression.
Here, a(N) is the extremely slow growing inverse of the
Ackermann function. For all practical values of n, a(n) is
less than 5.

Space Complexity: O(N) storage and auxiliary.

=~=~=~=~= Sample Output =~=~=~=~=
Elements: 7, Sets: 3
[[a,b,f],[c],[d,e,g]]

*/

#include <map>
```

```
#include <vector>

template<class T> class disjoint_set_forest {
  int num_elements, num_sets;
  std::map<T, int> ID;
  std::vector<int> root, rank;

  int find_root(int x) {
    if (root[x] != x) root[x] = find_root(root[x]);
    return root[x];
  }

 public:
  disjoint_set_forest(): num_elements(0), num_sets(0) {}
  int elements() { return num_elements; }
  int sets() { return num_sets; }

  bool is_united(const T & x, const T & y) {
    return find_root(ID[x]) == find_root(ID[y]);
  }

  void make_set(const T & x) {
    if (ID.find(x) != ID.end()) return;
    root.push_back(ID[x] = num_elements++);
    rank.push_back(0);
    num_sets++;
  }

  void unite(const T & x, const T & y) {
    int X = find_root(ID[x]), Y = find_root(ID[y]);
    if (X == Y) return;
    num_sets--;
    if (rank[X] < rank[Y]) root[X] = Y;
    else if (rank[X] > rank[Y]) root[Y] = X;
    else rank[root[Y] = X]++;
  }

  std::vector<std::vector<T> > get_all_sets() {
    std::map<int, std::vector<T> > tmp;
    for (typename std::map<T, int>::iterator
         it = ID.begin(); it != ID.end(); it++)
      tmp[find_root(it->second)].push_back(it->first);
    std::vector<std::vector<T> > ret;
    for (typename std::map<int, std::vector<T> >::
         iterator it = tmp.begin(); it != tmp.end(); it++)
      ret.push_back(it->second);
    return ret;
  }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  disjoint_set_forest<char> d;
  for (char c = 'a'; c <= 'g'; c++) d.make_set(c);
  d.unite('a', 'b');
  d.unite('b', 'f');
  d.unite('d', 'e');
  d.unite('e', 'g');
  cout << "Elements:␣" << d.elements();
  cout << ",␣Sets:␣" << d.sets() << endl;
  vector<vector<char> > s = d.get_all_sets();
  cout << "[";
  for (int i = 0; i < (int)s.size(); i++) {
    cout << (i > 0 ? ",[" : "[");
    for (int j = 0; j < (int)s[i].size(); j++)
      cout << (j > 0 ? "," : "") << s[i][j];
    cout << "]";
  }
  cout << "]\n";
  return 0;
}
```

3.2 Fenwick Trees

3.2.1 Simple Fenwick Tree

```
/*

Description: A Fenwick tree (a.k.a. binary indexed tree) is a
data structure that allows for the sum of an arbitrary range
of values in an array to be dynamically queried in logarithmic
time. Note that unlike the object-oriented versions of this
data structure found in later sections, the operations here
work on 1-based indices (i.e. between 1 and MAXN, inclusive).
The array a[] is always synchronized with the bit[] array and
should not be modified outside of the functions below.

Time Complexity: All functions are O(log MAXN).
Space Complexity: O(MAXN) storage and auxiliary.

*/

const int MAXN = 1000;
int a[MAXN + 1], bit[MAXN + 1];

//a[i] += v
void add(int i, int v) {
  a[i] += v;
  for (; i <= MAXN; i += i & -i)
    bit[i] += v;
}

//a[i] = v
void set(int i, int v) {
  int inc = v - a[i];
  add(i, inc);
}

//returns sum(a[i] for i = 1..hi inclusive)
int sum(int hi) {
  int ret = 0;
```

```cpp
  for (; hi > 0; hi -= hi & -hi)
    ret += bit[hi];
  return ret;
}

//returns sum(a[i] for i = lo..hi inclusive)
int sum(int lo, int hi) {
  return sum(hi) - sum(lo - 1);
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  for (int i = 1; i <= 5; i++) set(i, i);
  add(4, -5);
  cout << "BIT␣values:␣";
  for (int i = 1; i <= 5; i++)
    cout << a[i] << "␣"; //1 2 3 -1 5
  cout << "\nSum␣of␣range␣[1,3]␣is␣";
  cout << sum(1, 3) << ".\n"; //6
  return 0;
}
```

3.2.2 Fenwick Tree

```cpp
/*

Description: A Fenwick tree (a.k.a. binary indexed tree) is a
data structure that allows for the sum of an arbitrary range
of values in an array to be dynamically queried in logarithmic
time. All methods below work on 0-based indices (i.e. indices
in the range from 0 to size() - 1, inclusive, are valid).

Time Complexity: add(), set(), and sum() are all O(log N) on
the length of the array. size() and at() are O(1).

Space Complexity: O(N) storage and O(N) auxiliary on size().

*/

#include <vector>

template<class T> class fenwick_tree {
  int len;
  std::vector<int> a, bit;

 public:
  fenwick_tree(int n): len(n),
   a(n + 1), bit(n + 1) {}

  //a[i] += v
  void add(int i, const T & v) {
    a[++i] += v;
    for (; i <= len; i += i & -i)
```

```
      bit[i] += v;
  }

  //a[i] = v
  void set(int i, const T & v) {
    T inc = v - a[i + 1];
    add(i, inc);
  }

  //returns sum(a[i] for i = 1..hi inclusive)
  T sum(int hi) {
    T res = 0;
    for (hi++; hi > 0; hi -= hi & -hi)
      res += bit[hi];
    return res;
  }

  //returns sum(a[i] for i = lo..hi inclusive)
  T sum(int lo, int hi) {
    return sum(hi) - sum(lo - 1);
  }

  inline int size() { return len; }
  inline T at(int i) { return a[i + 1]; }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  int a[] = {10, 1, 2, 3, 4};
  fenwick_tree<int> t(5);
  for (int i = 0; i < 5; i++) t.set(i, a[i]);
  t.add(0, -5);
  cout << "BIT␣values:␣";
  for (int i = 0; i < t.size(); i++)
    cout << t.at(i) << "␣"; //5 1 2 3 4
  cout << "\nSum␣of␣range␣[1,␣3]␣is␣";
  cout << t.sum(1, 3) << ".\n"; //6
  return 0;
}
```

3.2.3 Fenwick Tree (Point Query)

```
/*

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

```

```
Time Complexity: add() and at() are O(log MAXN).
Space Complexity: O(N).

*/

const int MAXN = 1000;
int bit[MAXN + 1];

//a[i] += v
void add(int i, int v) {
  for (i++; i <= MAXN; i += i & -i) bit[i] += v;
}

//a[i] += v for i = lo..hi, inclusive
void add(int lo, int hi, int v) {
  add(lo, v);
  add(hi + 1, -v);
}

//returns a[i]
int at(int i) {
  int sum = 0;
  for (i++; i > 0; i -= i & -i) sum += bit[i];
  return sum;
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  add(1, 2, 5);
  add(2, 3, 5);
  add(3, 5, 10);
  cout << "BIT␣values:␣"; //5 10 15 10 10
  for (int i = 1; i <= 5; i++)
    cout << at(i) << "␣";
  cout << "\n";
  return 0;
}
```

3.2.4 Fenwick Tree (Range Update)

```
/*

Description: Using two arrays, a Fenwick tree can be made to
support range updates and range queries simultaneously. However,
the range updates can only be used to add an increment to all
values in a range, not set them to the same value. The latter
problem may be solved using a segment tree + lazy propagation.
All methods below operate 0-based indices (i.e. indices in the
range from 0 to size() - 1, inclusive, are valid).

Time Complexity: add(), set(), at(), and sum() are all O(log N)
on the length of the array. size() is O(1).

```

```
Space Complexity: O(N) storage and auxiliary.

=~=~=~=~= Sample Output =~=~=~=~=
BIT values: 15 6 7 -5 4
Sum of range [0, 4] is 27.

*/

#include <vector>

template<class T> class fenwick_tree {
  int len;
  std::vector<T> b1, b2;

  T sum(const std::vector<T> & b, int i) {
    T res = 0;
    for (; i != 0; i -= i & -i) res += b[i];
    return res;
  }

  void add(std::vector<T> & b, int i, const T & v) {
    for (; i <= len; i += i & -i) b[i] += v;
  }

 public:
  fenwick_tree(int n):
    len(n + 1), b1(n + 2), b2(n + 2) {}

  //a[i] += v for i = lo..hi, inclusive
  void add(int lo, int hi, const T & v) {
    lo++, hi++;
    add(b1, lo, v);
    add(b1, hi + 1, -v);
    add(b2, lo, v * (lo - 1));
    add(b2, hi + 1, -v * hi);
  }

  //a[i] = v
  void set(int i, const T & v) { add(i, i, v - at(i)); }

  //returns sum(a[i] for i = 1..hi inclusive)
  T sum(int hi) { return sum(b1, hi)*hi - sum(b2, hi); }

  //returns sum(a[i] for i = lo..hi inclusive)
  T sum(int lo, int hi) { return sum(hi + 1) - sum(lo); }

  inline int size() const { return len - 1; }
  inline T at(int i) { return sum(i, i); }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  int a[] = {10, 1, 2, 3, 4};
  fenwick_tree<int> t(5);
  for (int i = 0; i < 5; i++) t.set(i, a[i]);
  t.add(0, 2, 5); //15 6 7 3 4
  t.set(3, -5);   //15 6 7 -5 4
  cout << "BIT␣values:␣";
  for (int i = 0; i < t.size(); i++)
    cout << t.at(i) << "␣";
  cout << "\nSum␣of␣range␣[0,␣4]␣is␣";
  cout << t.sum(0, 4) << ".\n"; //27
  return 0;
}
```

3.2.5 Fenwick Tree (Map)

```
/*

Description: Using two std::maps to represent the Fenwick tree,
there no longer needs to be a restriction on the magnitude of
queried indices. All indices in range [0, MAXN] are valid.

Time Complexity: All functions are O(log^2 MAXN). If the
std::map is replaced with an std::unordered_map, then the
running time will become O(log MAXN) amortized.

Space Complexity: O(n) on the number of indices accessed.

*/

#include <map>

const int MAXN = 1000000000;
std::map<int, int> tmul, tadd;

void _add(int at, int mul, int add) {
  for (int i = at; i <= MAXN; i = (i | (i+1))) {
    tmul[i] += mul;
    tadd[i] += add;
  }
}

//a[i] += v for all i = lo..hi, inclusive
void add(int lo, int hi, int v) {
  _add(lo, v, -v * (lo - 1));
  _add(hi, -v, v * hi);
}

//returns sum(a[i] for i = 1..hi inclusive)
int sum(int hi) {
  int mul = 0, add = 0, start = hi;
  for (int i = hi; i >= 0; i = (i & (i + 1)) - 1) {
    if (tmul.find(i) != tmul.end())
      mul += tmul[i];
    if (tadd.find(i) != tadd.end())
      add += tadd[i];
  }
  return mul*start + add;
}

//returns sum(a[i] for i = lo..hi inclusive)
```

```cpp
int sum(int lo, int hi) {
  return sum(hi) - sum(lo - 1);
}

//a[i] = v
void set(int i, int v) {
  add(i, i, v - sum(i, i));
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  add(500000001, 500000010, 3);
  add(500000011, 500000015, 5);
  set(500000000, 10);
  cout << sum(500000000, 500000015) << "\n"; //65
  return 0;
}
```

3.2.6 2D Fenwick Tree

```cpp
/*

Description: A 2D Fenwick tree is abstractly a 2D array which also
supports efficient queries for the sum of values in the rectangle
with top-left (1, 1) and bottom-right (r, c). The implementation
below has indices accessible in the range [1...xmax][1...ymax].

Time Complexity: All functions are O(log(xmax)*log(ymax)).
Space Complexity: O(xmax*ymax) storage and auxiliary.

*/

const int xmax = 100, ymax = 100;

int a[xmax+1][ymax+1], bit[xmax+1][ymax+1];

//a[x][y] += v
void add(int x, int y, int v) {
  a[x][y] += v;
  for (int i = x; i <= xmax; i += i & -i)
    for (int j = y; j <= ymax; j += j & -j)
      bit[i][j] += v;
}

 //a[x][y] = v
void set(int x, int y, int v) {
  int inc = v - a[x][y];
  add(x, y, inc);
}

//returns sum(data[1..x][1..y], all inclusive)
int sum(int x, int y) {
  int ret = 0;
  for (int i = x; i > 0; i -= i & -i)
    for (int j = y; j > 0; j -= j & -j)
      ret += bit[i][j];
  return ret;
}

//returns sum(data[x1..x2][y1..y2], all inclusive)
int sum(int x1, int y1, int x2, int y2) {
  return sum(x2, y2) + sum(x1 - 1, y1 - 1) -
         sum(x1 - 1, y2) - sum(x2, y1 - 1);
}

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;

int main() {
  set(1, 1, 5);
  set(1, 2, 6);
  set(2, 1, 7);
  add(3, 3, 9);
  add(2, 1, -4);
/*
  5 6 0
  3 0 0
  0 0 9
*/
  cout << "2D␣BIT␣values:\n";
  for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++)
      cout << a[i][j] << "␣";
    cout << "\n";
  }
  assert(sum(1, 1, 1, 2) == 11);
  assert(sum(1, 1, 2, 1) == 8);
  assert(sum(1, 1, 3, 3) == 23);
  return 0;
}
```

3.2.7 2D Fenwick Tree (Range Update)

```
/*

Description: A 2D Fenwick tree is abstractly a 2D array which also
supports efficient queries for the sum of values in the rectangle
with top-left (1, 1) and bottom-right (r, c). The implementation
below has indices accessible in the range [0..xmax][0...ymax].

Time Complexity: All functions are O(log(xmax)*log(ymax)*log(N))
where N is the number of indices operated on so far. Use an array
or an unordered_map instead of a map to remove the log(N) factor.

Space Complexity: O(xmax*ymax) storage and auxiliary.

*/
```

```cpp

#include <map>
#include <utility>

template<class T> class fenwick_tree_2d {
  static const int xmax = 1000000000;
  static const int ymax = 1000000000;

  std::map<std::pair<int, int>, T> t1, t2, t3, t4;

  template<class Tree>
  void add(Tree & t, int x, int y, const T & v) {
    for (int i = x; i <= xmax; i += i & -i)
      for (int j = y; j <= ymax; j += j & -j)
        t[std::make_pair(i, j)] += v;
  }

  //a[i][j] += v for i = [1,x], j = [1,y]
  void add_pre(int x, int y, const T & v) {
    add(t1, 1, 1, v);

    add(t1, 1, y + 1, -v);
    add(t2, 1, y + 1, v * y);

    add(t1, x + 1, 1, -v);
    add(t3, x + 1, 1, v * x);

    add(t1, x + 1, y + 1, v);
    add(t2, x + 1, y + 1, -v * y);
    add(t3, x + 1, y + 1, -v * x);
    add(t4, x + 1, y + 1, v * x * y);
  }

 public:
  //a[i][j] += v for i = [x1,x2], j = [y1,y2]
  void add(int x1, int y1, int x2, int y2, const T & v) {
    x1++; y1++; x2++; y2++;
    add_pre(x2, y2, v);
    add_pre(x1 - 1, y2, -v);
    add_pre(x2, y1 - 1, -v);
    add_pre(x1 - 1, y1 - 1, v);
  }

  //a[x][y] += v
  void add(int x, int y, const T & v) {
    add(x, y, x, y, v);
  }

  //a[x][y] = v
  void set(int x, int y, const T & v) {
    add(x, y, v - at(x, y));
  }

  //returns sum(a[i][j] for i = [1,x], j = [1,y])
  T sum(int x, int y) {
    x++; y++;
    T s1 = 0, s2 = 0, s3 = 0, s4 = 0;
    for (int i = x; i > 0; i -= i & -i)
      for (int j = y; j > 0; j -= j & -j) {
        s1 += t1[std::make_pair(i, j)];
        s2 += t2[std::make_pair(i, j)];
        s3 += t3[std::make_pair(i, j)];
        s4 += t4[std::make_pair(i, j)];
      }
    return s1 * x * y + s2 * x + s3 * y + s4;
  }

  //returns sum(a[i][j] for i = [x1,x2], j = [y1,y2])
  T sum(int x1, int y1, int x2, int y2) {
    return sum(x2, y2) + sum(x1 - 1, y1 - 1) -
           sum(x1 - 1, y2) - sum(x2, y1 - 1);
  }

  T at(int x, int y) { return sum(x, y, x, y); }
};

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;

int main() {
  fenwick_tree_2d<long long> t;
  t.set(0, 0, 5);
  t.set(0, 1, 6);
  t.set(1, 0, 7);
  t.add(2, 2, 9);
  t.add(1, 0, -4);
  t.add(1, 1, 2, 2, 5);
/*
  5 6 0
  3 5 5
  0 5 14
*/
  cout << "2D␣BIT␣values:\n";
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++)
      cout << t.at(i, j) << "␣";
    cout << "\n";
  }
  assert(t.sum(0, 0, 0, 1) == 11);
  assert(t.sum(0, 0, 1, 0) == 8);
  assert(t.sum(1, 1, 2, 2) == 29);
  return 0;
}
```

3.3 1D Range Queries

3.3.1 Simple Segment Tree

```
/*

Description: A segment tree is a data structure used for
solving the dynamic range query problem, which asks to
```

```
determine the minimum (or maximum) value in any given
range in an array that is constantly being updated.

Time Complexity: Assuming merge() is O(1), build is O(n)
while query() and update() are O(log n). If merge() is
not O(1), then all running times are multiplied by a
factor of whatever complexity merge() runs in.

Space Complexity: O(MAXN). Note that a segment tree with
N leaves requires 2^(log2(N) - 1) = 4*N total nodes.

Note: This implementation is 0-based, meaning that all
indices from 0 to MAXN - 1, inclusive, are accessible.

=~=~=~=~= Sample Input =~=~=~=~=
5 10
35232
390942
649675
224475
18709
Q 1 3
M 4 475689
Q 2 3
Q 1 3
Q 1 2
Q 3 3
Q 2 3
M 2 645514
M 2 680746
Q 0 4

=~=~=~=~= Sample Output =~=~=~=~=
224475
224475
224475
390942
224475
224475
35232

*/

const int MAXN = 100000;
int N, M, a[MAXN], t[4*MAXN];

//define your custom nullv and merge() below.
//merge(x, nullv) must return x for all x

const int nullv = 1 << 30;

inline int merge(int a, int b) { return a < b ? a : b; }

void build(int n, int lo, int hi) {
  if (lo == hi) {
    t[n] = a[lo];
    return;
  }
  build(2*n + 1, lo, (lo + hi)/2);
  build(2*n + 2, (lo + hi)/2 + 1, hi);
  t[n] = merge(t[2*n + 1], t[2*n + 2]);
}

//x and y must be manually set before each call to the
//functions below. For query(), [x, y] is the range to
//be considered. For update(), a[x] is to be set to y.
int x, y;

//merge(a[i] for i = x..y, inclusive)
int query(int n, int lo, int hi) {
  if (hi < x || lo > y) return nullv;
  if (lo >= x && hi <= y) return t[n];
  return merge(query(2*n + 1, lo, (lo + hi) / 2),
               query(2*n + 2, (lo + hi) / 2 + 1, hi));
}

//a[x] = y
void update(int n, int lo, int hi) {
  if (hi < x || lo > x) return;
  if (lo == hi) {
    t[n] = y;
    return;
  }
  update(2*n + 1, lo, (lo + hi)/2);
  update(2*n + 2, (lo + hi)/2 + 1, hi);
  t[n] = merge(t[2*n + 1], t[2*n + 2]);
}

/*** Example Usage (wcipeg.com/problem/segtree) ***/

#include <cstdio>

int main() {
  scanf("%d%d", &N, &M);
  for (int i = 0; i < N; i++) scanf("%d", &a[i]);
  build(0, 0, N - 1);
  char op;
  for (int i = 0; i < M; i++) {
    scanf("␣%c%d%d", &op, &x, &y);
    if (op == 'Q') {
      printf("%d\n", query(0, 0, N - 1));
    } else if (op == 'M') {
      update(0, 0, N - 1);
    }
  }
  return 0;
}
```

3.3.2 Segment Tree

```
/*

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

```

Time Complexity: Assuming merge() is O(1), query(),
update(), and at() are O(log N). size() is O(1). If
merge() is not O(1), then all logarithmic running times
are multiplied by a factor of the complexity of merge().

Space Complexity: O(MAXN). Note that a segment tree with
N leaves requires 2^(log2(N) - 1) = 4*N total nodes.

Note: This implementation is 0-based, meaning that all
indices from 0 to N - 1, inclusive, are accessible.

*/

#include <limits> /* std::numeric_limits<T>::min() */
#include <vector>

template<class T> class segment_tree {
  int len, x, y;
  std::vector<T> t;
  T val, *init;

  //define the following yourself. merge(x, nullv) must return x for all x
  static inline T nullv() { return std::numeric_limits<T>::min(); }
  static inline T merge(const T & a, const T & b) { return a > b ? a : b; }

  void build(int n, int lo, int hi) {
    if (lo == hi) {
      t[n] = init[lo];
      return;
    }
    build(n * 2 + 1, lo, (lo + hi) / 2);
    build(n * 2 + 2, (lo + hi) / 2 + 1, hi);
    t[n] = merge(t[n * 2 + 1], t[n * 2 + 2]);
  }

  void update(int n, int lo, int hi) {
    if (x < lo || x > hi) return;
    if (lo == hi) {
      t[n] = val;
      return;
    }
    update(n * 2 + 1, lo, (lo + hi) / 2);
    update(n * 2 + 2, (lo + hi) / 2 + 1, hi);
    t[n] = merge(t[n * 2 + 1], t[n * 2 + 2]);
  }

  T query(int n, int lo, int hi) {
    if (hi < x || lo > y) return nullv();
    if (lo >= x && hi <= y) return t[n];
    return merge(query(n * 2 + 1, lo, (lo + hi) / 2),
                 query(n * 2 + 2, (lo + hi) / 2 + 1, hi));
  }

 public:
  segment_tree(int n, T * a = 0): len(n), t(4 * n, nullv()) {
    if (a != 0) {
      init = a;
      build(0, 0, len - 1);
    }
  }

  //a[i] = v
  void update(int i, const T & v) {
    x = i;
    val = v;
    update(0, 0, len - 1);
  }

  //merge(a[i] for i = lo..hi, inclusive)
  T query(int lo, int hi) {
    x = lo;
    y = hi;
    return query(0, 0, len - 1);
  }

  inline int size() { return len; }
  inline T at(int i) { return query(i, i); }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  int arr[5] = {6, -2, 1, 8, 10};
  segment_tree<int> T(5, arr);
  T.update(1, 4);
  cout << "Array␣contains:";
  for (int i = 0; i < T.size(); i++)
    cout << "␣" << T.at(i);
  cout << "\nThe␣max␣value␣in␣the␣range␣[0,␣3]␣is␣";
  cout << T.query(0, 3) << ".\n"; //8
  return 0;
}
```

3.3.3 Segment Tree (Range Updates)

```
/*

Description: A segment tree is a data structure used for
solving the dynamic range query problem, which asks to
determine the minimum (or maximum) value in any given
range in an array that is constantly being updated.
Lazy propagation is a technique applied to segment trees that
allows range updates to be carried out in O(log N) time. The
range updating mechanism is less versatile than the one
implemented in the next section.

Time Complexity: Assuming merge() is O(1), query(), update(),
at() are O(log(N)). If merge() is not constant time, then all
running times are multiplied by whatever complexity the merge
function runs in.

Space Complexity: O(N) on the size of the array. A segment
```

```
tree for an array of size N needs 2^(log2(N)-1) = 4N nodes.

Note: This implementation is 0-based, meaning that all
indices from 0 to size() - 1, inclusive, are accessible.

*/

#include <limits> /* std::numeric_limits<T>::min() */
#include <vector>

template<class T> class segment_tree {
  int len, x, y;
  std::vector<T> tree, lazy;
  T val, *init;

  //define the following yourself. merge(x, nullv) must return x for all valid x
  static inline T nullv() { return std::numeric_limits<T>::min(); }
  static inline T merge(const T & a, const T & b) { return a > b ? a : b; }

  void build(int n, int lo, int hi) {
    if (lo == hi) {
      tree[n] = init[lo];
      return;
    }
    build(n * 2 + 1, lo, (lo + hi) / 2);
    build(n * 2 + 2, (lo + hi) / 2 + 1, hi);
    tree[n] = merge(tree[n * 2 + 1], tree[n * 2 + 2]);
  }

  T query(int n, int lo, int hi) {
    if (x > hi || y < lo) return nullv();
    if (x <= lo && hi <= y) {
      if (lazy[n] == nullv()) return tree[n];
      return tree[n] = lazy[n];
    }
    int lchild = n * 2 + 1, rchild = n * 2 + 2;
    if (lazy[n] != nullv()) {
      lazy[lchild] = lazy[rchild] = lazy[n];
      lazy[n] = nullv();
    }
    return merge(query(lchild, lo, (lo + hi)/2),
                 query(rchild, (lo + hi)/2 + 1, hi));
  }

  void _update(int n, int lo, int hi) {
    if (x > hi || y < lo) return;
    if (lo == hi) {
      tree[n] = val;
      return;
    }
    if (x <= lo && hi <= y) {
      tree[n] = lazy[n] = merge(lazy[n], val);
      return;
    }
    int lchild = n * 2 + 1, rchild = n * 2 + 2;
    if (lazy[n] != nullv()) {
      lazy[lchild] = lazy[rchild] = lazy[n];
      lazy[n] = nullv();
    }
    _update(lchild, lo, (lo + hi) / 2);
    _update(rchild, (lo + hi) / 2 + 1, hi);
    tree[n] = merge(tree[lchild], tree[rchild]);
  }

 public:
  segment_tree(int n, T * a = 0):
   len(n), tree(4 * n, nullv()), lazy(4 * n, nullv()) {
    if (a != 0) {
      init = a;
      build(0, 0, len - 1);
    }
  }

  void update(int i, const T & v) {
    x = y = i;
    val = v;
    _update(0, 0, len - 1);
  }

  //a[i] = v for i = lo..hi, inclusive
  void update(int lo, int hi, const T & v) {
    x = lo; y = hi;
    val = v;
    _update(0, 0, len - 1);
  }

  //returns merge(a[i] for i = lo..hi, inclusive)
  T query(int lo, int hi) {
    x = lo;
    y = hi;
    return query(0, 0, len - 1);
  }

  inline int size() { return len; }
  inline T at(int i) { return query(i, i); }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  int arr[5] = {6, 4, 1, 8, 10};
  segment_tree<int> T(5, arr);
  cout << "Array␣contains:"; //6 4 1 8 10
  for (int i = 0; i < T.size(); i++)
    cout << "␣" << T.at(i);
  cout << "\n";
  T.update(2, 4, 12);
  cout << "Array␣contains:"; //6 4 12 12 12
  for (int i = 0; i < T.size(); i++)
    cout << "␣" << T.at(i);
  cout << "\nThe␣max␣value␣in␣the␣range␣[0,␣3]␣is␣";
  cout << T.query(0, 3) << ".\n"; //12
  return 0;
}
```

3.3.4 Segment Tree (Fast, Non-recursive)

```
/*

Description: A segment tree is a data structure used for
solving the dynamic range query problem, which asks to
determine the minimum (or maximum) value in any given
range in an array that is constantly being updated.
Lazy propagation is a technique applied to segment trees that
allows range updates to be carried out in O(log N) time.

Time Complexity: Assuming merge() is O(1), query(), update(),
at() are O(log(N)). If merge() is not constant time, then all
running times are multiplied by whatever complexity the merge
function runs in.

Space Complexity: O(N) on the size of the array.

Note: This implementation is 0-based, meaning that all
indices from 0 to T.size() - 1, inclusive, are accessible.

*/

#include <algorithm> /* std::fill(), std::max() */
#include <stdexcept> /* std::runtime_error */
#include <vector>

template<class T> class segment_tree {
  //Modify the following 5 methods to implement your custom
  //operations on the tree. This implements the Add/Max operations.
  //Operations like Add/Sum, Set/Max can also be implemented.
  static inline T modify_op(const T & x, const T & y) {
    return x + y;
  }

  static inline T query_op(const T & x, const T & y) {
    return std::max(x, y);
  }

  static inline T delta_on_segment(const T & delta, int seglen) {
    if (delta == nullv()) return nullv();
    //Here you must write a fast equivalent of following slow code:
    //  T result = delta;
    //  for (int i = 1; i < seglen; i++) result = query_op(result, delta);
    //  return result;
    return delta;
  }

  static inline T nullv() { return 0; }
  static inline T initv() { return 0; }

  int length;
  std::vector<T> value, delta;
  std::vector<int> len;

  static T join_value_with_delta(const T & val, const T & delta) {
    return delta == nullv() ? val : modify_op(val, delta);
  }

  static T join_deltas(const T & delta1, const T & delta2) {
    if (delta1 == nullv()) return delta2;
    if (delta2 == nullv()) return delta1;
    return modify_op(delta1, delta2);
  }

  T join_value_with_delta(int i) {
    return join_value_with_delta(value[i], delta_on_segment(delta[i], len[i]));
  }

  void push_delta(int i) {
    int d = 0;
    while ((i >> d) > 0) d++;
    for (d -= 2; d >= 0; d--) {
      int x = i >> d;
      value[x >> 1] = join_value_with_delta(x >> 1);
      delta[x] = join_deltas(delta[x], delta[x >> 1]);
      delta[x ^ 1] = join_deltas(delta[x ^ 1], delta[x >> 1]);
      delta[x >> 1] = nullv();
    }
  }

 public:
  segment_tree(int n):
   length(n), value(2 * n), delta(2 * n, nullv()), len(2 * n) {
    std::fill(len.begin() + n, len.end(), 1);
    for (int i = 0; i < n; i++) value[i + n] = initv();
    for (int i = 2 * n - 1; i > 1; i -= 2) {
      value[i >> 1] = query_op(value[i], value[i ^ 1]);
      len[i >> 1] = len[i] + len[i ^ 1];
    }
  }

  T query(int lo, int hi) {
    if (lo < 0 || hi >= length || lo > hi)
      throw std::runtime_error("Invalid␣query␣range.");
    push_delta(lo += length);
    push_delta(hi += length);
    T res = 0;
    bool found = false;
    for (; lo <= hi; lo = (lo + 1) >> 1, hi = (hi - 1) >> 1) {
      if ((lo & 1) != 0) {
        res = found ? query_op(res, join_value_with_delta(lo)) :
                      join_value_with_delta(lo);
        found = true;
      }
      if ((hi & 1) == 0) {
        res = found ? query_op(res, join_value_with_delta(hi)) :
                      join_value_with_delta(hi);
        found = true;
      }
    }
    if (!found) throw std::runtime_error("Not␣found.");
    return res;
  }

  void modify(int lo, int hi, const T & delta) {
    if (lo < 0 || hi >= length || lo > hi)
```

```
      throw std::runtime_error("Invalid␣modify␣range.");
    push_delta(lo += length);
    push_delta(hi += length);
    int ta = -1, tb = -1;
    for (; lo <= hi; lo = (lo + 1) >> 1, hi = (hi - 1) >> 1) {
      if ((lo & 1) != 0) {
        this->delta[lo] = join_deltas(this->delta[lo], delta);
        if (ta == -1) ta = lo;
      }
      if ((hi & 1) == 0) {
        this->delta[hi] = join_deltas(this->delta[hi], delta);
        if (tb == -1) tb = hi;
      }
    }
    for (int i = ta; i > 1; i >>= 1)
      value[i >> 1] = query_op(join_value_with_delta(i),
                               join_value_with_delta(i ^ 1));
    for (int i = tb; i > 1; i >>= 1)
      value[i >> 1] = query_op(join_value_with_delta(i),
                               join_value_with_delta(i ^ 1));
  }

  inline int size() { return length; }
  inline T at(int i) { return query(i, i); }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  segment_tree<int> T(10);
  T.modify(0, 0, 10);
  T.modify(1, 1, 5);
  T.modify(1, 1, 4);
  T.modify(2, 2, 7);
  T.modify(3, 3, 8);
  cout << T.query(0, 3) << "\n"; //10
  cout << T.query(1, 3) << "\n"; //9
  T.modify(0, 9, 5);
  cout << T.query(0, 9) << "\n"; //15
  cout << "Array␣contains:"; //15 14 12 13 5 5 5 5 5 5
  for (int i = 0; i < T.size(); i++)
    cout << "␣" << T.at(i);
  cout << "\n";
  return 0;
}
```

3.3.5 Implicit Treap

```
/*

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

```
O(log N), while decreasing the running time of insert() and erase()
from O(N) to O(log N). Furthermore, this version supports the same
operations as a segment tree with lazy propagation, allowing range
updates and queries to be performed in O(log N).

Time Complexity: Assuming the join functions have constant complexity:
insert(), push_back(), erase(), at(), modify(), and query() are all
O(log N), while walk() is O(N).

Space Complexity: O(N) on the size of the array.

Note: This implementation is 0-based, meaning that all
indices from 0 to size() - 1, inclusive, are accessible.

*/

#include <climits> /* INT_MIN */
#include <cstdlib> /* srand(), rand() */
#include <ctime>   /* time() */

template<class T> class implicit_treap {
  //Modify the following 5 functions to implement your custom
  //operations on the tree. This implements the Add/Max operations.
  //Operations like Add/Sum, Set/Max can also be implemented.
  static inline T join_values(const T & a, const T & b) {
    return a > b ? a : b;
  }

  static inline T join_deltas(const T & d1, const T & d2) {
    return d1 + d2;
  }

  static inline T join_value_with_delta(const T & v, const T & d, int len) {
    return v + d;
  }

  static inline T null_delta() { return 0; }
  static inline T null_value() { return INT_MIN; }

  struct node_t {
    static inline int rand32() {
      return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
    }

    T value, subtree_value, delta;
    int count, priority;
    node_t *L, *R;

    node_t(const T & val) {
      value = subtree_value = val;
      delta = null_delta();
      count = 1;
      L = R = 0;
      priority = rand32();
    }
  } *root;

  static int count(node_t * n) {
    return n ? n->count : 0;
```

```
}

static T subtree_value(node_t * n) {
  return n ? n->subtree_value : null_value();
}

static void update(node_t * n) {
  if (n == 0) return;
  n->subtree_value = join_values(join_values(subtree_value(n->L), n->value),
                                 subtree_value(n->R));
  n->count = 1 + count(n->L) + count(n->R);
}

static void apply_delta(node_t * n, const T & delta) {
  if (n == 0) return;
  n->delta = join_deltas(n->delta, delta);
  n->value = join_value_with_delta(n->value, delta, 1);
  n->subtree_value = join_value_with_delta(n->subtree_value, delta, n->count);
}

static void push_delta(node_t * n) {
  if (n == 0) return;
  apply_delta(n->L, n->delta);
  apply_delta(n->R, n->delta);
  n->delta = null_delta();
}

static void merge(node_t *& n, node_t * L, node_t * R) {
  push_delta(L);
  push_delta(R);
  if (L == 0) n = R;
  else if (R == 0) n = L;
  else if (L->priority < R->priority)
    merge(L->R, L->R, R), n = L;
  else
    merge(R->L, L, R->L), n = R;
  update(n);
}

static void split(node_t * n, node_t *& L, node_t *& R, int key) {
  push_delta(n);
  if (n == 0) L = R = 0;
  else if (key <= count(n->L))
    split(n->L, L, n->L, key), R = n;
  else
    split(n->R, n->R, R, key - count(n->L) - 1), L = n;
  update(n);
}

static void insert(node_t *& n, node_t * item, int idx) {
  push_delta(n);
  if (n == 0) n = item;
  else if (item->priority < n->priority)
    split(n, item->L, item->R, idx), n = item;
  else if (idx <= count(n->L))
    insert(n->L, item, idx);
  else
    insert(n->R, item, idx - count(n->L) - 1);
  update(n);
  }

  static T get(node_t * n, int idx) {
    push_delta(n);
    if (idx < count(n->L))
      return get(n->L, idx);
    else if (idx > count(n->L))
      return get(n->R, idx - count(n->L) - 1);
    return n->value;
  }

  static void erase(node_t *& n, int idx) {
    push_delta(n);
    if (idx == count(n->L)) {
      delete n;
      merge(n, n->L, n->R);
    } else if (idx < count(n->L)) {
      erase(n->L, idx);
    } else {
      erase(n->R, idx - count(n->L) - 1);
    }
  }

  template<class UnaryFunction>
  void walk(node_t * n, UnaryFunction f) {
    if (n == 0) return;
    push_delta(n);
    if (n->L) walk(n->L, f);
    f(n->value);
    if (n->R) walk(n->R, f);
  }

  void clean_up(node_t *& n) {
    if (n == 0) return;
    clean_up(n->L);
    clean_up(n->R);
    delete n;
  }

 public:
  implicit_treap(): root(0) { srand(time(0)); }
  ~implicit_treap() { clean_up(root); }

  int size() const { return count(root); }
  bool empty() const { return root == 0; }

  //list.insert(list.begin() + idx, val)
  void insert(int idx, const T & val) {
    if (idx < 0 || idx > size()) return;
    node_t * item = new node_t(val);
    insert(root, item, idx);
  }

  void push_back(const T & val) {
    insert(size(), val);
  }

  //list.erase(list.begin() + idx)
  void erase(int idx) {
    if (idx < 0 || idx >= size()) return;
    erase(root, idx);
  }

  T at(int idx) {
    if (root == 0 || idx < 0 || idx >= size())
      return null_value();
    return get(root, idx);
  }

  template<class UnaryFunction> void walk(UnaryFunction f) {
    walk(root, f);
  }

  //for (i = a; i <= b; i++)
  //  list[i] = join_value_with_delta(list[i], delta)
  void modify(int a, int b, const T & delta) {
    if (a < 0 || b < 0 || a >= size() || b >= size() || a > b)
      return;
    node_t *l1, *r1;
    split(root, l1, r1, b + 1);
    node_t *l2, *r2;
    split(l1, l2, r2, a);
    apply_delta(r2, delta);
    node_t *t;
    merge(t, l2, r2);
    merge(root, t, r1);
  }

  //return join_values(list[a..b])
  T query(int a, int b) {
    if (a < 0 || b < 0 || a >= size() || b >= size() || a > b)
      return null_value();
    node_t *l1, *r1;
    split(root, l1, r1, b + 1);
    node_t *l2, *r2;
    split(l1, l2, r2, a);
    int res = subtree_value(r2);
    node_t *t;
    merge(t, l2, r2);
    merge(root, t, r1);
    return res;
  }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

void print(int x) { cout << x << "␣"; }

int main() {
  implicit_treap<int> T;
  T.push_back(7);
  T.push_back(8);
  T.push_back(9);
  T.insert(1, 5);
  T.erase(3);
  T.walk(print); cout << "\n";   //7 5 8
  T.modify(0, 2, 2);
  T.walk(print); cout << "\n";   //9 7 10
  cout << T.at(1) << "\n";       //7
  cout << T.query(0, 2) << "\n"; //10
  cout << T.size() << "\n";      //3
  return 0;
}
```

3.3.6 Sparse Table

```
/*

Description: The static range minimum query problem can be solved
using a sparse table data structure. The RMQ for sub arrays of
length 2^k is pre-processed using dynamic programming with formula:

dp[i][j] = dp[i][j-1], if A[dp[i][j-1]] <= A[dp[i+2^(j-1)-1][j-1]]
           dp[i+2^(j-1)-1][j-1], otherwise

where dp[i][j] is the index of the minimum value in the sub array
starting at i having length 2^j.

Time Complexity: O(N log N) for build() and O(1) for min_idx()
Space Complexity: O(N log N) on the size of the array.

Note: This implementation is 0-based, meaning that all
indices from 0 to N - 1, inclusive, are valid.

*/

#include <vector>

const int MAXN = 100;
std::vector<int> logtable, dp[MAXN];

void build(int n, int a[]) {
  logtable.resize(n + 1);
  for (int i = 2; i <= n; i++)
    logtable[i] = logtable[i >> 1] + 1;
  for (int i = 0; i < n; i++) {
    dp[i].resize(logtable[n] + 1);
    dp[i][0] = i;
  }
  for (int k = 1; (1 << k) < n; k++) {
    for (int i = 0; i + (1 << k) <= n; i++) {
      int x = dp[i][k - 1];
      int y = dp[i + (1 << (k - 1))][k - 1];
      dp[i][k] = a[x] <= a[y] ? x : y;
    }
  }
}

//returns index of min element in [lo, hi]
int min_idx(int a[], int lo, int hi) {
  int k = logtable[hi - lo];
  int x = dp[lo][k];
  int y = dp[hi - (1 << k) + 1][k];
  return a[x] <= a[y] ? x : y;
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  int a[] = {7, -10, 5, 20};
  build(4, a);
  cout << min_idx(a, 0, 3) << "\n"; //1
  return 0;
}
```

3.3.7 Square Root Decomposition

```
/*

Description: To solve the dynamic range query problem using
square root decomposition, we split an array of size N into
sqrt(N) buckets, each bucket of size sqrt(N). As a result,
each query and update operation will be sqrt(N) in running time.

Time Complexity: O(N*sqrt(N)) to construct the initial
decomposition. After, query() and update() are O(sqrt N)/call.

Space Complexity: O(N) for the array. O(sqrt N) for the buckets.

Note: This implementation is 0-based, meaning that all
indices from 0 to N - 1, inclusive, are accessible.

=~=~=~=~= Sample Input =~=~=~=~=
5 10
35232
390942
649675
224475
18709
Q 1 3
M 4 475689
Q 2 3
Q 1 3
Q 1 2
Q 3 3
Q 2 3
M 2 645514
M 2 680746
Q 0 4

=~=~=~=~= Sample Output =~=~=~=~=
224475
224475
224475
390942
224475
```

```
224475
35232

*/

#include <cmath> /* sqrt() */
#include <limits> /* std::numeric_limits<T>::max() */
#include <vector>

template<class T> class sqrt_decomp {
  //define the following yourself. merge(x, nullv) must return x for all x
  static inline T nullv() { return std::numeric_limits<T>::max(); }
  static inline T merge(const T & a, const T & b) { return a < b ? a : b; }

  int len, blocklen, blocks;
  std::vector<T> array, block;

 public:
  sqrt_decomp(int n, T * a = 0): len(n), array(n) {
    blocklen = (int)sqrt(n);
    blocks = (n + blocklen - 1) / blocklen;
    block.resize(blocks);
    for (int i = 0; i < n; i++)
      array[i] = a ? a[i] : nullv();
    for (int i = 0; i < blocks; i++) {
      int h = (i + 1) * blocklen;
      if (h > n) h = n;
      block[i] = nullv();
      for (int j = i * blocklen; j < h; j++)
        block[i] = merge(block[i], array[j]);
    }
  }

  void update(int idx, const T & val) {
    array[idx] = val;
    int b = idx / blocklen;
    int h = (b + 1) * blocklen;
    if (h > len) h = len;
    block[b] = nullv();
    for (int i = b * blocklen; i < h; i++)
      block[b] = merge(block[b], array[i]);
  }

  T query(int lo, int hi) {
    T ret = nullv();
    int lb = ceil((double)lo / blocklen);
    int hb = (hi + 1) / blocklen - 1;
    if (lb > hb) {
      for (int i = lo; i <= hi; i++)
        ret = merge(ret, array[i]);
    } else {
      int l = lb * blocklen - 1;
      int h = (hb + 1) * blocklen;
      for (int i = lo; i <= l; i++)
        ret = merge(ret, array[i]);
      for (int i = lb; i <= hb; i++)
        ret = merge(ret, block[i]);
      for (int i = h; i <= hi; i++)
        ret = merge(ret, array[i]);
    }
    return ret;
  }

  inline int size() { return len; }
  inline int at(int idx) { return array[idx]; }
};

/*** Example Usage (wcipeg.com/problem/segtree) ***/

#include <cstdio>

int N, M, A, B, init[100005];

int main() {
  scanf("%d%d", &N, &M);
  for (int i = 0; i < N; i++) scanf("%d", &init[i]);
  sqrt_decomp<int> a(N, init);
  char op;
  for (int i = 0; i < M; i++) {
    scanf("␣%c%d%d", &op, &A, &B);
    if (op == 'Q') {
      printf("%d\n", a.query(A, B));
    } else if (op == 'M') {
      a.update(A, B);
    }
  }
  return 0;
}
```

3.3.8 Interval Tree (Augmented Treap)

```
/*

Description: An interval tree is structure used to store and efficiently
query intervals. An interval may be dynamically inserted, and range
queries of [lo, hi] may be performed to have the tree report all intervals
that intersect with the queried interval. Augmented trees, described in
CLRS (2009, Section 14.3: pp. 348354), is one way to represent these
intervals. This implementation uses a treap to maintain balance.
See: http://en.wikipedia.org/wiki/Interval_tree#Augmented_tree

Time Complexity: On average O(log N) for insert() and O(k) for query(),
where N is the number of intervals in the tree and k is the number of
intervals that will be reported by each query().

Space Complexity: O(N) on the number of intervals in the tree.

*/

#include <cstdlib>   /* srand() */
#include <ctime>     /* time() */
#include <utility>   /* std:pair */

class interval_tree {
  typedef std::pair<int, int> interval;

```

```
static bool overlap(const interval & a, const interval & b) {
  return a.first <= b.second && b.first <= a.second;
}

struct node_t {
  static inline int rand32() {
    return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
  }

  interval i;
  int maxh, priority;
  node_t *L, *R;

  node_t(const interval & i) {
    this->i = i;
    maxh = i.second;
    L = R = 0;
    priority = rand32();
  }

  void update() {
    maxh = i.second;
    if (L != 0 && L->maxh > maxh) maxh = L->maxh;
    if (R != 0 && R->maxh > maxh) maxh = R->maxh;
  }
} *root;

static void rotate_l(node_t *& k2) {
  node_t *k1 = k2->R;
  k2->R = k1->L;
  k1->L = k2;
  k2 = k1;
  k2->update();
  k1->update();
}

static void rotate_r(node_t *& k2) {
  node_t *k1 = k2->L;
  k2->L = k1->R;
  k1->R = k2;
  k2 = k1;
  k2->update();
  k1->update();
}

interval i; //temporary

void insert(node_t *& n) {
  if (n == 0) { n = new node_t(i); return; }
  if (i.first < (n->i).first) {
    insert(n->L);
    if (n->L->priority < n->priority) rotate_r(n);
  } else {
    insert(n->R);
    if (n->R->priority < n->priority) rotate_l(n);
  }
  n->update();
}

```

```
  template<class ReportFunction>
  void query(node_t * n, ReportFunction f) {
    if (n == 0 || n->maxh < i.first) return;
    if (overlap(n->i, i)) f(n->i.first, n->i.second);
    query(n->L, f);
    query(n->R, f);
  }

  static void clean_up(node_t * n) {
    if (n == 0) return;
    clean_up(n->L);
    clean_up(n->R);
    delete n;
  }

 public:
  interval_tree(): root(0) { srand(time(0)); }
  ~interval_tree() { clean_up(root); }

  void insert(int lo, int hi) {
    i = interval(lo, hi);
    insert(root);
  }

  template<class ReportFunction>
  void query(int lo, int hi, ReportFunction f) {
    i = interval(lo, hi);
    query(root, f);
  }
};

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;

void print(int lo, int hi) {
  cout << "[" << lo << "," << hi << "] ";
}

int cnt;
void count(int lo, int hi) { cnt++; }

int main() {
  int N = 6;
  int intv[6][2] = {{15, 20}, {10, 30}, {17, 19}, {5, 20}, {12, 15}, {30, 40}};
  interval_tree T;
  for (int i = 0; i < N; i++) {
    T.insert(intv[i][0], intv[i][1]);
  }
  T.query(10, 20, print); cout << "\n"; //[15,20] [10,30] [5,20] [12,15] [17,19]
  T.query(0, 5, print); cout << "\n";   //[5,20]
  T.query(25, 45, print); cout << "\n"; //[10,30] [30,40]
  //check correctness
  for (int l = 0; l <= 50; l++) {
    for (int h = l; h <= 50; h++) {
      cnt = 0;
      T.query(l, h, count);
      int cnt2 = 0;
      for (int i = 0; i < N; i++)
        if (intv[i][0] <= h && l <= intv[i][1])
          cnt2++;
      assert(cnt == cnt2);
    }
  }
  return 0;
}
```

3.4 2D Range Queries

3.4.1 Quadtree (Simple)

```
/*

Description: A quadtree can be used to dynamically query values
of rectangles in a 2D array. In a quadtree, every node has exactly
4 children. The following uses a statically allocated array to
store the nodes. This is less efficient than a 2D segment tree.

Time Complexity: For update(), query() and at(): O(log(N*M)) on
average and O(sqrt(N*M)) in the worst case, where N is the number
of rows and M is the number of columns in the 2D array.

Space Complexity: O(N*M)

Note: This implementation is 0-based. Valid indices for
all operations are [0..xmax][0..ymax]

*/

#include <climits> /* INT_MIN */

const int xmax = 100, ymax = 100;
int tree[4 * xmax * ymax];
int X, Y, X1, X2, Y1, Y2, V; //temporary value to speed up recursion

//define the following yourself. merge(x, nullv) must return x for all valid x
inline int nullv() { return INT_MIN; }
inline int merge(int a, int b) { return a > b ? a : b; }

void update(int n, int x1, int x2, int y1, int y2) {
  if (X < x1 || X > x2 || Y < y1 || Y > y2) return;
  if (x1 == x2 && y1 == y2) {
    tree[n] = V;
    return;
  }
  update(n * 4 + 1, x1, (x1 + x2) / 2, y1, (y1 + y2) / 2);
  update(n * 4 + 2, x1, (x1 + x2) / 2, (y1 + y2) / 2 + 1, y2);
  update(n * 4 + 3, (x1 + x2) / 2 + 1, x2, y1, (y1 + y2) / 2);
  update(n * 4 + 4, (x1 + x2) / 2 + 1, x2, (y1 + y2) / 2 + 1, y2);
  tree[n] = merge(merge(tree[n * 4 + 1], tree[n * 4 + 2]),
                  merge(tree[n * 4 + 3], tree[n * 4 + 4]));
}

```

```
void query(int n, int x1, int x2, int y1, int y2) {
  if (x1 > X2 || x2 < X1 || y2 < Y1 || y1 > Y2 || merge(tree[n], V) == V)
    return;
  if (x1 >= X1 && x2 <= X2 && y1 >= Y1 && y2 <= Y2) {
    V = merge(tree[n], V);
    return;
  }
  query(n * 4 + 1, x1, (x1 + x2) / 2, y1, (y1 + y2) / 2);
  query(n * 4 + 2, x1, (x1 + x2) / 2, (y1 + y2) / 2 + 1, y2);
  query(n * 4 + 3, (x1 + x2) / 2 + 1, x2, y1, (y1 + y2) / 2);
  query(n * 4 + 4, (x1 + x2) / 2 + 1, x2, (y1 + y2) / 2 + 1, y2);
}

void update(int x, int y, int v) {
  X = x;
  Y = y;
  V = v;
  update(0, 0, xmax - 1, 0, ymax - 1);
}

int query(int x1, int y1, int x2, int y2) {
  X1 = x1;
  X2 = x2;
  Y1 = y1;
  Y2 = y2;
  V = nullv();
  query(0, 0, xmax - 1, 0, ymax - 1);
  return V;
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  int arr[5][5] = {{1, 2, 3, 4, 5},
                   {5, 4, 3, 2, 1},
                   {6, 7, 8, 0, 0},
                   {0, 1, 2, 3, 4},
                   {5, 9, 9, 1, 2}};
  for (int r = 0; r < 5; r++)
    for (int c = 0; c < 5; c++)
      update(r, c, arr[r][c]);
  cout << "The␣maximum␣value␣in␣the␣rectangle␣with␣";
  cout << "upper␣left␣(0,2)␣and␣lower␣right␣(3,4)␣is␣";
  cout << query(0, 2, 3, 4) << ".\n"; //8
  return 0;
}
```

3.4.2 Quadtree

```
/*

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

```
store the nodes, which allows arbitrarily large indices to exist
without affecting the performance of operations.

Time Complexity: For update(), query() and at(): O(log(N*M)) on
average and O(sqrt(N*M)) in the worst case, where N is the number
of rows and M is the number of columns in the 2D array.

Space Complexity: O(N*M)

Note: This implementation is 0-based. Valid indices for
all operations are [0..XMAX][0..YMAX]

*/

#include <algorithm> /* std::max(), std::min() */
#include <limits>    /* std::numeric_limits<T>::min() */

template<class T> class quadtree {
  //these can be set to large values without affecting your memory usage!
  static const int xmax = 1000000000;
  static const int ymax = 1000000000;

  //define the following yourself. merge(x, nullv) must return x for all valid x
  static inline T nullv() { return std::numeric_limits<T>::min(); }
  static inline T merge(const T & a, const T & b) { return a > b ? a : b; }

  int X, Y, X1, X2, Y1, Y2;  T V; //temp vals for speed

  struct node_t {
    node_t * child[4];
    int x1, x2, y1, y2;
    T value;

    node_t(int x, int y) {
      x1 = x2 = x;
      y1 = y2 = y;
      child[0] = child[1] = child[2] = child[3] = 0;
      value = nullv();
    }
  } *root;

  void update(node_t *& n, int x1, int x2, int y1, int y2) {
    if (X < x1 || X > x2 || Y < y1 || Y > y2) return;
    if (n == 0) n = new node_t(X, Y);
    if (x1 == x2 && y1 == y2) {
      n->value = V;
      return;
    }
    int xmid = (x1 + x2)/2, ymid = (y1 + y2)/2;
    update(n->child[0], x1, xmid, y1, ymid);
    update(n->child[1], xmid + 1, x2, y1, ymid);
    update(n->child[2], x1, xmid, ymid + 1, y2);
    update(n->child[3], xmid + 1, x2, ymid + 1, y2);
    for (int i = 0; i < 4; i++) {
      if (!n->child[i] || n->child[i]->value == nullv()) continue;
      n->x1 = std::min(n->x1, n->child[i]->x1);
      n->x2 = std::max(n->x2, n->child[i]->x2);
      n->y1 = std::min(n->y1, n->child[i]->y1);
      n->y2 = std::max(n->y2, n->child[i]->y2);
      n->value = merge(n->value, n->child[i]->value);
    }
  }

  void query(node_t * n) {
    if (n == 0 || n->x1 > X2 || n->x2 < X1 || n->y2 < Y1 || n->y1 > Y2 ||
        merge(n->value, V) == V)
      return;
    if (n->x1 >= X1 && n->y1 >= Y1 && n->x2 <= X2 && n->y2 <= Y2) {
      V = merge(V, n->value);
      return;
    }
    for (int i = 0; i < 4; i++) query(n->child[i]);
  }

  static void clean_up(node_t * n) {
    if (n == 0) return;
    for (int i = 0; i < 4; i++) clean_up(n->child[i]);
    delete n;
  }

 public:
  quadtree() { root = 0; }
  ~quadtree() { clean_up(root); }

  void update(int x, int y, const T & v) {
    X = x;
    Y = y;
    V = v;
    update(root, 0, xmax - 1, 0, ymax - 1);
  }

  T query(int x1, int y1, int x2, int y2) {
    X1 = x1;
    X2 = x2;
    Y1 = y1;
    Y2 = y2;
    V = nullv();
    query(root);
    return V;
  }

  T at(int x, int y) {
    return query(x, y, x, y);
  }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  int arr[5][5] = {{1, 2, 3, 4, 5},
                   {5, 4, 3, 2, 1},
                   {6, 7, 8, 0, 0},
                   {0, 1, 2, 3, 4},
                   {5, 9, 9, 1, 2}};
  quadtree<int> T;
  for (int r = 0; r < 5; r++)
    for (int c = 0; c < 5; c++)
      T.update(r, c, arr[r][c]);
  cout << "The␣maximum␣value␣in␣the␣rectangle␣with␣";
  cout << "upper␣left␣(0,2)␣and␣lower␣right␣(3,4)␣is␣";
  cout << T.query(0, 2, 3, 4) << ".\n"; //8
  return 0;
}
```

3.4.3 2D Segment Tree

```
/*

Description: A quadtree is a segment tree but with 4 children
per node, making its running time proportional to the square
root of the number of leaves. However, a 2D segment tree is a
segment tree of segment trees, making its running time
proportional to the log of its size. The following implementation
is a highly optimized implementation with features such as
coordinate compression and path compression.

Time Complexity: O(log(xmax)*log(ymax)) for update(), query(),
and at() operations. size() is O(1).

Space Complexity: Left as an exercise for the reader.

Note: This implementation is 0-based. Valid indices for
all operations are [0..xmax][0..ymax]

*/

#include <limits> /* std::numeric_limits<T>::min() */

template<class T> class segment_tree_2d {
  //these can be set to large values without affecting your memory usage!
  static const int xmax = 1000000000;
  static const int ymax = 1000000000;

  //define the following yourself. merge(x, nullv) must return x for all valid x
  static inline T nullv() { return std::numeric_limits<T>::min(); }
  static inline T merge(const T & a, const T & b) { return a > b ? a : b; }

  struct layer2_node {
    int lo, hi;
    layer2_node *L, *R;
    T value;
    layer2_node(int l, int h) : lo(l), hi(h), L(0), R(0) {}
  };

  struct layer1_node {
    layer1_node *L, *R;
    layer2_node l2;
    layer1_node() : L(0), R(0), l2(0, ymax) {}
  } *root;

  void update2(layer2_node * node, int Q, const T & v) {
    int lo = node->lo, hi = node->hi, mid = (lo + hi)/2;
    if (lo + 1 == hi) {
      node->value = v;
      return;
    }
    layer2_node *& tgt = Q < mid ? node->L : node->R;
    if (tgt == 0) {
      tgt = new layer2_node(Q, Q + 1);
      tgt->value = v;
    } else if (tgt->lo <= Q && Q < tgt->hi) {
      update2(tgt, Q, v);
    } else {
      do {
        (Q < mid ? hi : lo) = mid;
        mid = (lo + hi)/2;
      } while ((Q < mid) == (tgt->lo < mid));
      layer2_node *nnode = new layer2_node(lo, hi);
      (tgt->lo < mid ? nnode->L : nnode->R) = tgt;
      tgt = nnode;
      update2(nnode, Q, v);
    }
    node->value = merge(node->L ? node->L->value : nullv(),
                        node->R ? node->R->value : nullv());
  }

  T query2(layer2_node * nd, int A, int B) {
    if (nd == 0 || B <= nd->lo || nd->hi <= A) return nullv();
    if (A <= nd->lo && nd->hi <= B) return nd->value;
    return merge(query2(nd->L, A, B), query2(nd->R, A, B));
  }

  void update1(layer1_node * node, int lo, int hi, int x, int y, const T & v) {
    if (lo + 1 == hi) update2(&node->l2, y, v);
    else {
      int mid = (lo + hi)/2;
      layer1_node *& nnode = x < mid ? node->L : node->R;
      (x < mid ? hi : lo) = mid;
      if (nnode == 0) nnode = new layer1_node();
      update1(nnode, lo, hi, x, y, v);
      update2(&node->l2, y, merge(
        node->L ? query2(&node->L->l2, y, y + 1) : nullv(),
        node->R ? query2(&node->R->l2, y, y + 1) : nullv())
      );
    }
  }

  T query1(layer1_node * nd, int lo, int hi, int A1, int B1, int A2, int B2) {
    if (nd == 0 || B1 <= lo || hi <= A1) return nullv();
    if (A1 <= lo && hi <= B1) return query2(&nd->l2, A2, B2);
    int mid = (lo + hi) / 2;
    return merge(query1(nd->L, lo, mid, A1, B1, A2, B2),
                 query1(nd->R, mid, hi, A1, B1, A2, B2));
  }

  void clean_up2(layer2_node * n) {
    if (n == 0) return;
    clean_up2(n->L);
    clean_up2(n->R);
    delete n;
  }

  void clean_up1(layer1_node * n) {
    if (n == 0) return;
    clean_up2(n->l2.L);
    clean_up2(n->l2.R);
    clean_up1(n->L);
    clean_up1(n->R);
    delete n;
  }

 public:
  segment_tree_2d() { root = new layer1_node(); }
  ~segment_tree_2d() { clean_up1(root); }

  void update(int x, int y, const T & v) {
    update1(root, 0, xmax, x, y, v);
  }

  T query(int x1, int y1, int x2, int y2) {
    return query1(root, 0, xmax, x1, x2 + 1, y1, y2 + 1);
  }

  T at(int x, int y) {
    return query(x, y, x, y);
  }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  int arr[5][5] = {{1, 2, 3, 4, 5},
                   {5, 4, 3, 2, 1},
                   {6, 7, 8, 0, 0},
                   {0, 1, 2, 3, 4},
                   {5, 9, 9, 1, 2}};
  segment_tree_2d<int> T;
  for (int r = 0; r < 5; r++)
    for (int c = 0; c < 5; c++)
      T.update(r, c, arr[r][c]);
  cout << "The maximum value in the rectangle with ";
  cout << "upper left (0,2) and lower right (3,4) is ";
  cout << T.query(0, 2, 3, 4) << ".\n"; //8
  return 0;
}
```

3.4.4 K-d Tree (2D Range Query)

```
/*

Description: k-d tree (short for k-dimensional tree) is a space-
partitioning data structure for organizing points in a k-
dimensional space. The following implementation supports
counting the number of points in rectangular ranges after the
tree has been build.
```

```

Time Complexity: O(N log N) for build(), where N is the number
of points in the tree. count() is O(sqrt N).

Space Complexity: O(N) on the number of points.

*/

#include <algorithm> /* nth_element(), max(), min() */
#include <climits>   /* INT_MIN, INT_MAX */
#include <utility>   /* std::pair */
#include <vector>

class kd_tree {
  typedef std::pair<int, int> point;

  static inline bool cmp_x(const point & a, const point & b) {
    return a.first < b.first;
  }

  static inline bool cmp_y(const point & a, const point & b) {
    return a.second < b.second;
  }

  std::vector<int> tx, ty, cnt, minx, miny, maxx, maxy;
  int x1, y1, x2, y2; //temporary values to speed up recursion

  void build(int lo, int hi, bool div_x, point P[]) {
    if (lo >= hi) return;
    int mid = (lo + hi) >> 1;
    std::nth_element(P + lo, P + mid, P + hi, div_x ? cmp_x : cmp_y);
    tx[mid] = P[mid].first;
    ty[mid] = P[mid].second;
    cnt[mid] = hi - lo;
    minx[mid] = INT_MAX; miny[mid] = INT_MAX;
    maxx[mid] = INT_MIN; maxy[mid] = INT_MIN;
    for (int i = lo; i < hi; i++) {
      minx[mid] = std::min(minx[mid], P[i].first);
      maxx[mid] = std::max(maxx[mid], P[i].first);
      miny[mid] = std::min(miny[mid], P[i].second);
      maxy[mid] = std::max(maxy[mid], P[i].second);
    }
    build(lo, mid, !div_x, P);
    build(mid + 1, hi, !div_x, P);
  }

  int count(int lo, int hi) {
    if (lo >= hi) return 0;
    int mid = (lo + hi) >> 1;
    int ax = minx[mid], ay = miny[mid];
    int bx = maxx[mid], by = maxy[mid];
    if (ax > x2 || x1 > bx || ay > y2 || y1 > by) return 0;
    if (x1 <= ax && bx <= x2 && y1 <= ay && by <= y2) return cnt[mid];
    int res = count(lo, mid) + count(mid + 1, hi);
    res += (x1 <= tx[mid] && tx[mid] <= x2 && y1 <= ty[mid] && ty[mid] <= y2);
    return res;
  }

 public:
```

```cpp
  kd_tree(int n, point P[]): tx(n), ty(n), cnt(n),
    minx(n), miny(n), maxx(n), maxy(n) {
     build(0, n, true, P);
  }

  int count(int x1, int y1, int x2, int y2) {
    this->x1 = x1;
    this->y1 = y1;
    this->x2 = x2;
    this->y2 = y2;
    return count(0, tx.size());
  }
};

/*** Example Usage ***/

#include <cassert>
using namespace std;

int main() {
  pair<int, int> P[4];
  P[0] = make_pair(0, 0);
  P[1] = make_pair(10, 10);
  P[2] = make_pair(0, 10);
  P[3] = make_pair(10, 0);
  kd_tree t(4, P);
  assert(t.count(0, 0, 10, 9) == 2);
  assert(t.count(0, 0, 10, 10) == 4);
  return 0;
}
```

3.4.5 K-d Tree (Nearest Neighbor)

```cpp
/*

Description: k-d tree (short for k-dimensional tree) is a space-
partitioning data structure for organizing points in a k-
dimensional space. The following implementation supports
querying the nearest neighboring point to (x, y) in terms of
Euclidean distance after the tree has been build. Note that
a point is not considered its own neighbour if it already exists
in the tree.

Time Complexity: O(N log N) for build(), where N is the number of
points in the tree. nearest_neighbor_id() is O(log(N)) on average.

Space Complexity: O(N) on the number of points.

*/

#include <algorithm> /* nth_element(), max(), min(), swap() */
#include <climits>   /* INT_MIN, INT_MAX */
#include <utility>
#include <vector>

class kd_tree {
  typedef std::pair<int, int> point;

  static inline bool cmp_x(const point & a, const point & b) {
    return a.first < b.first;
  }

  static inline bool cmp_y(const point & a, const point & b) {
    return a.second < b.second;
  }

  std::vector<int> tx, ty;
  std::vector<bool> div_x;

  void build(int lo, int hi, point P[]) {
    if (lo >= hi) return;
    int mid = (lo + hi) >> 1;
    int minx = INT_MAX, maxx = INT_MIN;
    int miny = INT_MAX, maxy = INT_MIN;
    for (int i = lo; i < hi; i++) {
      minx = std::min(minx, P[i].first);
      maxx = std::max(maxx, P[i].first);
      miny = std::min(miny, P[i].second);
      maxy = std::max(maxy, P[i].second);
    }
    div_x[mid] = (maxx - minx) >= (maxy - miny);
    std::nth_element(P + lo, P + mid, P + hi, div_x[mid] ? cmp_x : cmp_y);
    tx[mid] = P[mid].first;
    ty[mid] = P[mid].second;
    if (lo + 1 == hi) return;
    build(lo, mid, P);
    build(mid + 1, hi, P);
  }

  long long min_dist;
  int min_dist_id, x, y;

  void nearest_neighbor(int lo, int hi) {
    if (lo >= hi) return;
    int mid = (lo + hi) >> 1;
    int dx = x - tx[mid], dy = y - ty[mid];
    long long d = dx*(long long)dx + dy*(long long)dy;
    if (min_dist > d && d) {
      min_dist = d;
      min_dist_id = mid;
    }
    if (lo + 1 == hi) return;
    int delta = div_x[mid] ? dx : dy;
    long long delta2 = delta*(long long)delta;
    int l1 = lo, r1 = mid, l2 = mid + 1, r2 = hi;
    if (delta > 0) std::swap(l1, l2), std::swap(r1, r2);
    nearest_neighbor(l1, r1);
    if (delta2 < min_dist) nearest_neighbor(l2, r2);
  }

 public:
  kd_tree(int N, point P[]) {
    tx.resize(N);
    ty.resize(N);
    div_x.resize(N);
    build(0, N, P);
  }

  int nearest_neighbor_id(int x, int y) {
    this->x = x; this->y = y;
    min_dist = LLONG_MAX;
    nearest_neighbor(0, tx.size());
    return min_dist_id;
  }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  pair<int, int> P[3];
  P[0] = make_pair(0, 2);
  P[1] = make_pair(0, 3);
  P[2] = make_pair(-1, 0);
  kd_tree T(3, P);
  int res = T.nearest_neighbor_id(0, 0);
  cout << P[res].first << "␣" << P[res].second << "\n"; //-1, 0
  return 0;
}
```

3.4.6 R-Tree (Nearest Segment)

```
/*

Description: R-trees are tree data structures used for spatial
access methods, i.e., for indexing multi-dimensional information
such as geographical coordinates, rectangles or polygons. The
following implementation supports querying of the nearing line
segment to a point after a tree of line segments have been built.

Time Complexity: O(N log N) for build(), where N is the number of
points in the tree. nearest_neighbor_id() is O(log(N)) on average.

Space Complexity: O(N) on the number of points.

*/

#include <algorithm> /* nth_element(), max(), min(), swap() */
#include <cfloat>    /* DBL_MAX */
#include <climits>   /* INT_MIN, INT_MAX */
#include <vector>

struct segment { int x1, y1, x2, y2; };

class r_tree {

  static inline bool cmp_x(const segment & a, const segment & b) {
    return a.x1 + a.x2 < b.x1 + b.x2;
  }

  static inline bool cmp_y(const segment & a, const segment & b) {
```

```
  return a.y1 + a.y2 < b.y1 + b.y2;
}

std::vector<segment> s;
std::vector<int> minx, maxx, miny, maxy;

void build(int lo, int hi, bool div_x, segment s[]) {
  if (lo >= hi) return;
  int mid = (lo + hi) >> 1;
  std::nth_element(s + lo, s + mid, s + hi, div_x ? cmp_x : cmp_y);
  this->s[mid] = s[mid];
  for (int i = lo; i < hi; i++) {
    minx[mid] = std::min(minx[mid], std::min(s[i].x1, s[i].x2));
    miny[mid] = std::min(miny[mid], std::min(s[i].y1, s[i].y2));
    maxx[mid] = std::max(maxx[mid], std::max(s[i].x1, s[i].x2));
    maxy[mid] = std::max(maxy[mid], std::max(s[i].y1, s[i].y2));
  }
  build(lo, mid, !div_x, s);
  build(mid + 1, hi, !div_x, s);
}

double min_dist;
int min_dist_id, x, y;

void nearest_neighbor(int lo, int hi, bool div_x) {
  if (lo >= hi) return;
  int mid = (lo + hi) >> 1;
  double pdist = point_to_segment_squared(x, y, s[mid]);
  if (min_dist > pdist) {
    min_dist = pdist;
    min_dist_id = mid;
  }
  long long delta = div_x ? 2*x - s[mid].x1 - s[mid].x2 :
                            2*y - s[mid].y1 - s[mid].y2;
  if (delta <= 0) {
    nearest_neighbor(lo, mid, !div_x);
    if (mid + 1 < hi) {
      int mid1 = (mid + hi + 1) >> 1;
      long long dist = div_x ? seg_dist(x, minx[mid1], maxx[mid1]) :
                               seg_dist(y, miny[mid1], maxy[mid1]);
      if (dist*dist < min_dist) nearest_neighbor(mid + 1, hi, !div_x);
    }
  } else {
    nearest_neighbor(mid + 1, hi, !div_x);
    if (lo < mid) {
      int mid1 = (lo + mid) >> 1;
      long long dist = div_x ? seg_dist(x, minx[mid1], maxx[mid1]) :
                               seg_dist(y, miny[mid1], maxy[mid1]);
      if (dist*dist < min_dist) nearest_neighbor(lo, mid, !div_x);
    }
  }
}

static double point_to_segment_squared(int x, int y, const segment & s) {
  long long dx = s.x2 - s.x1, dy = s.y2 - s.y1;
  long long px = x - s.x1, py = y - s.y1;
  long long square_dist = dx*dx + dy*dy;
  long long dot_product = dx*px + dy*py;
  if (dot_product <= 0 || square_dist == 0) return px*px + py*py;
```

```
    if (dot_product >= square_dist)
      return (px - dx)*(px - dx) + (py - dy)*(py - dy);
    double q = (double)dot_product/square_dist;
    return (px - q*dx)*(px - q*dx) + (py - q*dy)*(py - q*dy);
  }

  static inline int seg_dist(int v, int lo, int hi) {
    return v <= lo ? lo - v : (v >= hi ? v - hi : 0);
  }

 public:
  r_tree(int N, segment s[]) {
    this->s.resize(N);
    minx.assign(N, INT_MAX);
    maxx.assign(N, INT_MIN);
    miny.assign(N, INT_MAX);
    maxy.assign(N, INT_MIN);
    build(0, N, true, s);
  }

  int nearest_neighbor_id(int x, int y) {
    min_dist = DBL_MAX;
    this->x = x; this->y = y;
    nearest_neighbor(0, s.size(), true);
    return min_dist_id;
  }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  segment s[4];
  s[0] = (segment){0, 0, 0, 4};
  s[1] = (segment){0, 4, 4, 4};
  s[2] = (segment){4, 4, 4, 0};
  s[3] = (segment){4, 0, 0, 0};
  r_tree t(4, s);
  int id = t.nearest_neighbor_id(-1, 2);
  cout << s[id].x1 << "␣" << s[id].y1 << "␣" <<
          s[id].x2 << "␣" << s[id].y2 << "\n"; //0 0 0 4
  return 0;
}
```

3.4.7 2D Range Tree

```
/*

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

```

Time Complexity: A range tree can build() in O(N log^(d-1)(N))
and query() in O(log^d(n) + k), where N is the number of points
stored in the tree, d is the dimension of each point and k is the
number of points reported by a given query. Thus for this 2D case
build() is O(N log N) and query() is O(log^2(N) + k).

Space Complexity: O(N log^(d-1)(N)) for a d-dimensional range tree.
Thus for this 2D case, the space complexity is O(N log N).

*/

#include <algorithm> /* lower_bound(), merge(), sort() */
#include <utility>   /* std::pair */
#include <vector>

class range_tree_2d {
  typedef std::pair<int, int> point;

  std::vector<point> P;
  std::vector<std::vector<point> > seg;

  static inline bool comp1(const point & a, const point & b) {
    return a.second < b.second;
  }

  static inline bool comp2(const point & a, int v) {
    return a.second < v;
  }

  void build(int n, int lo, int hi) {
    if (P[lo].first == P[hi].first) {
      for (int i = lo; i <= hi; i++)
        seg[n].push_back(point(i, P[i].second));
      return;
    }
    int l = n * 2 + 1, r = n * 2 + 2;
    build(l, lo, (lo + hi)/2);
    build(r, (lo + hi)/2 + 1, hi);
    seg[n].resize(seg[l].size() + seg[r].size());
    std::merge(seg[l].begin(), seg[l].end(), seg[r].begin(), seg[r].end(),
               seg[n].begin(), comp1);
  }

  int xl, xh, yl, yh;

  template<class ReportFunction>
  void query(int n, int lo, int hi, ReportFunction f) {
    if (P[hi].first < xl || P[lo].first > xh) return;
    if (xl <= P[lo].first && P[hi].first <= xh) {
      if (!seg[n].empty() && yh >= yl) {
        std::vector<point>::iterator it;
        it = std::lower_bound(seg[n].begin(), seg[n].end(), yl, comp2);
        for (; it != seg[n].end(); ++it) {
          if (it->second > yh) break;
          f(it->first); //or report P[it->first], the actual point
        }
      }
    } else if (lo != hi) {
      query(n * 2 + 1, lo, (lo + hi) / 2, f);
      query(n * 2 + 2, (lo + hi) / 2 + 1, hi, f);
    }
  }

 public:
  range_tree_2d(int n, point init[]): seg(4 *n + 1) {
    std::sort(init, init + n);
    P = std::vector<point>(init, init + n);
    build(0, 0, n - 1);
  }

  template<class ReportFunction>
  void query(int x1, int y1, int x2, int y2, ReportFunction f) {
    xl = x1; xh = x2;
    yl = y1; yh = y2;
    query(0, 0, P.size() - 1, f);
  }
};

/*** Example Usage (wcipeg.com/problem/boxl) ***/

#include <bitset>
#include <cstdio>
using namespace std;

int N, M; bitset<200005> b;
pair<int, int> pts[200005];
int x1[200005], y1[200005];
int x2[200005], y2[200005];

void mark(int i) {
  b[i] = true;
}

int main() {
  scanf("%d%d", &N, &M);
  for (int i = 0; i < N; i++)
    scanf("%d%d%d%d", x1 + i, y1 + i, x2 + i, y2 + i);
  for (int i = 0; i < M; i++)
    scanf("%d%d", &pts[i].first, &pts[i].second);
  range_tree_2d t(M, pts);
  for (int i = 0; i < N; i++)
    t.query(x1[i], y1[i], x2[i], y2[i], mark);
  printf("%d\n", b.count());
  return 0;
}
```

3.5 Search Trees and Alternatives

3.5.1 Binary Search Tree

```
/*

Description: A binary search tree (BST) is a node-based binary tree data
structure where the left sub-tree of every node has keys less than the
```

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

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

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

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

*/

template<class key_t, class val_t> class binary_search_tree {
  struct node_t {
    key_t key;
    val_t val;
    node_t *L, *R;

    node_t(const key_t & k, const val_t & v) {
      key = k;
      val = v;
      L = R = 0;
    }
  } *root;

  int num_nodes;

  static bool insert(node_t *& n, const key_t & k, const val_t & v) {
    if (n == 0) {
      n = new node_t(k, v);
      return true;
    }
    if (k < n->key) return insert(n->L, k, v);
    if (n->key < k) return insert(n->R, k, v);
    return false; //already exists
  }

  static bool erase(node_t *& n, const key_t & key) {
    if (n == 0) return false;
    if (key < n->key) return erase(n->L, key);
    if (n->key < key) return erase(n->R, key);
    if (n->L == 0) {
      node_t *temp = n->R;
      delete n;
      n = temp;
    } else if (n->R == 0) {
      node_t *temp = n->L;
      delete n;
      n = temp;
    } else {
      node_t *temp = n->R, *parent = 0;
      while (temp->L != 0) {
        parent = temp;
        temp = temp->L;
      }
      n->key = temp->key;
      n->val = temp->val;
      if (parent != 0)
        return erase(parent->L, parent->L->key);
      return erase(n->R, n->R->key);
    }
    return true;
  }

  template<class BinaryFunction>
  static void walk(node_t * n, BinaryFunction f) {
    if (n == 0) return;
    walk(n->L, f);
    f(n->key, n->val);
    walk(n->R, f);
  }

  static void clean_up(node_t * n) {
    if (n == 0) return;
    clean_up(n->L);
    clean_up(n->R);
    delete n;
  }

 public:
  binary_search_tree(): root(0), num_nodes(0) {}
  ~binary_search_tree() { clean_up(root); }
  int size() const { return num_nodes; }
  bool empty() const { return root == 0; }

  bool insert(const key_t & key, const val_t & val) {
    if (insert(root, key, val)) {
      num_nodes++;
      return true;
    }
    return false;
  }

  bool erase(const key_t & key) {
    if (erase(root, key)) {
      num_nodes--;
      return true;
    }
    return false;
  }

  template<class BinaryFunction> void walk(BinaryFunction f) {
    walk(root, f);
  }

  val_t * find(const key_t & key) {
    for (node_t *n = root; n != 0; ) {
      if (n->key == key) return &(n->val);
      n = (key < n->key ? n->L : n->R);
    }
    return 0; //key not found
```

```cpp
  }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

void printch(int k, char v) { cout << v; }

int main() {
  binary_search_tree<int, char> T;
  T.insert(2, 'b');
  T.insert(1, 'a');
  T.insert(3, 'c');
  T.insert(5, 'e');
  T.insert(4, 'x');
  *T.find(4) = 'd';
  cout << "In-order: ";
  T.walk(printch);  //abcde
  cout << "\nRemoving node with key 3...";
  cout << (T.erase(3) ? "Success!" : "Failed");
  cout << "\n";
  return 0;
}
```

3.5.2 Treap

```cpp
/*

Description: A binary search tree (BST) is a node-based binary tree data
structure where the left sub-tree of every node has keys less than the
node's key and the right sub-tree of every node has keys greater than the
node's key. A BST may be come degenerate like a linked list resulting in
an O(N) running time per operation. A self-balancing binary search tree
such as a randomized treap prevents the occurence of this known worst case.

Treaps use randomly generated priorities to reduce the height of the
tree. We assume that the rand() function in <cstdlib> is 16-bits, and
call it twice to generate a 32-bit number. For the treap to be
effective, the range of the randomly generated numbers should be
between 0 and around the number of elements in the treap.

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

Time Complexity: insert(), erase(), and find() are O(log(N)) on average
and O(N) in the worst case. Despite the technically O(N) worst case,
such cases are still extremely difficult to trigger, making treaps
very practice in many programming contest applications. walk() is O(N).

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

*/
```

```cpp

#include <cstdlib> /* srand(), rand() */
#include <ctime>   /* time() */

template<class key_t, class val_t> class treap {
  struct node_t {
    static inline int rand32() {
      return (rand() & 0x7fff) | ((rand() & 0x7fff) << 15);
    }

    key_t key;
    val_t val;
    int priority;
    node_t *L, *R;

    node_t(const key_t & k, const val_t & v): key(k), val(v), L(0), R(0) {
      priority = rand32();
    }
  } *root;

  int num_nodes;

  static void rotate_l(node_t *& k2) {
    node_t *k1 = k2->R;
    k2->R = k1->L;
    k1->L = k2;
    k2 = k1;
  }

  static void rotate_r(node_t *& k2) {
    node_t *k1 = k2->L;
    k2->L = k1->R;
    k1->R = k2;
    k2 = k1;
  }

  static bool insert(node_t *& n, const key_t & k, const val_t & v) {
    if (n == 0) {
      n = new node_t(k, v);
      return true;
    }
    if (k < n->key && insert(n->L, k, v)) {
      if (n->L->priority < n->priority) rotate_r(n);
      return true;
    } else if (n->key < k && insert(n->R, k, v)) {
      if (n->R->priority < n->priority) rotate_l(n);
      return true;
    }
    return false;
  }

  static bool erase(node_t *& n, const key_t & k) {
    if (n == 0) return false;
    if (k < n->key) return erase(n->L, k);
    if (k > n->key) return erase(n->R, k);
    if (n->L == 0 || n->R == 0) {
      node_t *temp = n;
      n = (n->L != 0) ? n->L : n->R;
      delete temp;
      return true;
    }
    if (n->L->priority < n->R->priority) {
      rotate_r(n);
      return erase(n->R, k);
    }
    rotate_l(n);
    return erase(n->L, k);
  }

  template<class BinaryFunction>
  static void walk(node_t * n, BinaryFunction f) {
    if (n == 0) return;
    walk(n->L, f);
    f(n->key, n->val);
    walk(n->R, f);
  }

  static void clean_up(node_t * n) {
    if (n == 0) return;
    clean_up(n->L);
    clean_up(n->R);
    delete n;
  }

 public:
  treap(): root(0), num_nodes(0) { srand(time(0)); }
  ~treap() { clean_up(root); }
  int size() const { return num_nodes; }
  bool empty() const { return root == 0; }

  bool insert(const key_t & key, const val_t & val) {
    if (insert(root, key, val)) {
      num_nodes++;
      return true;
    }
    return false;
  }

  bool erase(const key_t & key) {
    if (erase(root, key)) {
      num_nodes--;
      return true;
    }
    return false;
  }

  template<class BinaryFunction> void walk(BinaryFunction f) {
    walk(root, f);
  }

  val_t * find(const key_t & key) {
    for (node_t *n = root; n != 0; ) {
      if (n->key == key) return &(n->val);
      n = (key < n->key ? n->L : n->R);
    }
    return 0; //key not found
  }
};
```

```

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;

void printch(int k, char v) { cout << v; }

int main() {
  treap<int, char> T;
  T.insert(2, 'b');
  T.insert(1, 'a');
  T.insert(3, 'c');
  T.insert(5, 'e');
  T.insert(4, 'x');
  *T.find(4) = 'd';
  cout << "In-order:␣";
  T.walk(printch);  //abcde
  cout << "\nRemoving␣node␣with␣key␣3...";
  cout << (T.erase(3) ? "Success!" : "Failed");
  cout << "\n";

  //stress test - runs in <0.5 seconds
  //insert keys in an order that would break a normal BST
  treap<int, int> T2;
  for (int i = 0; i < 1000000; i++)
    T2.insert(i, i*1337);
  for (int i = 0; i < 1000000; i++)
    assert(*T2.find(i) == i*1337);
  return 0;
}
```

3.5.3 Size Balanced Tree (Order Statistics)

```
/*

Description: A binary search tree (BST) is a node-based binary tree data
structure where the left sub-tree of every node has keys less than the
node's key and the right sub-tree of every node has keys greater than the
node's key. A BST may be come degenerate like a linked list resulting in
an O(N) running time per operation. A self-balancing binary search tree
such as a randomized treap prevents the occurence of this known worst case.

The size balanced tree is a data structure first published in 2007 by
Chinese student Chen Qifeng. The tree is rebalanced by examining the sizes
of each node's subtrees. It is popular amongst Chinese OI competitors due
to its speed, simplicity to implement, and ability to double up as an
ordered statistics tree if necessary.
For more info, see: http://wcipeg.com/wiki/Size_Balanced_Tree

An ordered statistics tree is a BST that supports additional operations:
- Select(i): find the i-th smallest element stored in the tree
- Rank(x):   find the rank of element x in the tree,
             i.e. its index in the sorted list of elements of the tree
For more info, see: http://en.wikipedia.org/wiki/Order_statistic_tree

```

```
Note: The following implementation is used similar to an std::map. In order
to make it behave like an std::set, modify the code to remove the value
associated with each node. Making a size balanced tree behave like an
std::multiset or std::multimap is a more complex issue. Refer to the
articles above and determine the correct way to preserve the binary search
tree property with maintain() if equivalent keys are allowed.

Time Complexity: insert(), erase(), find(), select() and rank() are
O(log N) on the number of elements in the tree. walk() is O(N).

Space Complexity: O(N) on the number of nodes in the tree.

*/

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

template<class key_t, class val_t> class size_balanced_tree {
  struct node_t {
    key_t key;
    val_t val;
    int size;
    node_t * c[2];

    node_t(const key_t & k, const val_t & v) {
      key = k, val = v;
      size = 1;
      c[0] = c[1] = 0;
    }

    void update() {
      size = 1;
      if (c[0]) size += c[0]->size;
      if (c[1]) size += c[1]->size;
    }
  } *root;

  static inline int size(node_t * n) {
    return n ? n->size : 0;
  }

  static void rotate(node_t *& n, bool d) {
    node_t * p = n->c[d];
    n->c[d] = p->c[!d];
    p->c[!d] = n;
    n->update();
    p->update();
    n = p;
  }

  static void maintain(node_t *& n, bool d) {
    if (n == 0 || n->c[d] == 0) return;
    node_t *& p = n->c[d];
    if (size(p->c[d]) > size(n->c[!d])) {
      rotate(n, d);
    } else if (size(p->c[!d]) > size(n->c[!d])) {
      rotate(p, !d);
      rotate(n, d);
    } else return;
    maintain(n->c[0], 0);
    maintain(n->c[1], 1);
    maintain(n, 0);
    maintain(n, 1);
  }

  static void insert(node_t *& n, const key_t & k, const val_t & v) {
    if (n == 0) {
      n = new node_t(k, v);
      return;
    }
    if (k < n->key) {
      insert(n->c[0], k, v);
      maintain(n, 0);
    } else if (n->key < k) {
      insert(n->c[1], k, v);
      maintain(n, 1);
    } else return;
    n->update();
  }

  static void erase(node_t *& n, const key_t & k) {
    if (n == 0) return;
    bool d = k < n->key;
    if (k < n->key) {
      erase(n->c[0], k);
    } else if (n->key < k) {
      erase(n->c[1], k);
    } else {
      if (n->c[1] == 0 || n->c[0] == 0) {
        delete n;
        n = n->c[1] == 0 ? n->c[0] : n->c[1];
        return;
      }
      node_t * p = n->c[1];
      while (p->c[0] != 0) p = p->c[0];
      n->key = p->key;
      erase(n->c[1], p->key);
    }
    maintain(n, d);
    n->update();
  }

  template<class BinaryFunction>
  static void walk(node_t * n, BinaryFunction f) {
    if (n == 0) return;
    walk(n->c[0], f);
    f(n->key, n->val);
    walk(n->c[1], f);
  }

  static std::pair<key_t, val_t> select(node_t *& n, int k) {
    int r = size(n->c[0]);
    if (k < r) return select(n->c[0], k);
    if (k > r) return select(n->c[1], k - r - 1);
    return std::make_pair(n->key, n->val);
  }

  static int rank(node_t * n, const key_t & k) {
    if (n == 0)
      throw std::runtime_error("Cannot␣rank␣key␣not␣in␣tree.");
    int r = size(n->c[0]);
    if (k < n->key) return rank(n->c[0], k);
    if (n->key < k) return rank(n->c[1], k) + r + 1;
    return r;
  }

  static void clean_up(node_t * n) {
    if (n == 0) return;
    clean_up(n->c[0]);
    clean_up(n->c[1]);
    delete n;
  }

 public:
  size_balanced_tree() : root(0) {}
  ~size_balanced_tree() { clean_up(root); }
  int size() { return size(root); }
  bool empty() const { return root == 0; }

  void insert(const key_t & key, const val_t & val) {
    insert(root, key, val);
  }

  void erase(const key_t & key) {
    erase(root, key);
  }

  template<class BinaryFunction> void walk(BinaryFunction f) {
    walk(root, f);
  }

  val_t * find(const key_t & key) {
    for (node_t *n = root; n != 0; ) {
      if (n->key == key) return &(n->val);
      n = (key < n->key ? n->c[0] : n->c[1]);
    }
    return 0; //key not found
  }

  std::pair<key_t, val_t> select(int k) {
    if (k >= size(root))
      throw std::runtime_error("k␣must␣be␣smaller␣size␣of␣tree.");
    return select(root, k);
  }

  int rank(const key_t & key) {
    return rank(root, key);
  }
};

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;

void printch(int k, char v) { cout << v; }
```

```

int main() {
  size_balanced_tree<int, char> T;
  T.insert(2, 'b');
  T.insert(1, 'a');
  T.insert(3, 'c');
  T.insert(5, 'e');
  T.insert(4, 'x');
  *T.find(4) = 'd';
  cout << "In-order:␣";
  T.walk(printch);                  //abcde
  T.erase(3);
  cout << "\nRank␣of␣2:␣" << T.rank(2); //1
  cout << "\nRank␣of␣5:␣" << T.rank(5); //3
  cout << "\nValue␣of␣3rd␣smallest␣key:␣";
  cout << T.select(2).second;           //d
  cout << "\n";

  //stress test - runs in <1 second
  //insert keys in an order that would break a normal BST
  size_balanced_tree<int, int> T2;
  for (int i = 0; i < 1000000; i++)
    T2.insert(i, i*1337);
  for (int i = 0; i < 1000000; i++)
    assert(*T2.find(i) == i*1337);
  return 0;
}
```

3.5.4 Hashmap (Chaining)

```
/*

Description: A hashmap (std::unordered_map in C++11) is an
alternative to a binary search tree. Hashmaps use more memory than
BSTs, but are usually more efficient. The following implementation
uses the chaining method to handle collisions. You can use the
hash algorithms provided in the example, or define your own.

Time Complexity: insert(), remove(), find(), are O(1) amortized.
rehash() is O(N).

Space Complexity: O(N) on the number of entries.

*/

#include <list>

template<class key_t, class val_t, class Hash> class hashmap {
  struct entry_t {
    key_t key;
    val_t val;
    entry_t(const key_t & k, const val_t & v): key(k), val(v) {}
  };

  std::list<entry_t> * table;
  int table_size, map_size;

  /**
   * This doubles the table size, then rehashes every entry.
   * Rehashing is expensive; it is strongly suggested for the
   * table to be constructed with a large size to avoid rehashing.
   */
  void rehash() {
    std::list<entry_t> * old = table;
    int old_size = table_size;
    table_size = 2*table_size;
    table = new std::list<entry_t>[table_size];
    map_size = 0;
    typename std::list<entry_t>::iterator it;
    for (int i = 0; i < old_size; i++)
      for (it = old[i].begin(); it != old[i].end(); ++it)
        insert(it->key, it->val);
    delete[] old;
  }

 public:
  hashmap(int size = 1024): table_size(size), map_size(0) {
    table = new std::list<entry_t>[table_size];
  }

  ~hashmap() { delete[] table; }
  int size() const { return map_size; }

  void insert(const key_t & key, const val_t & val) {
    if (find(key) != 0) return;
    if (map_size >= table_size) rehash();
    unsigned int i = Hash()(key) % table_size;
    table[i].push_back(entry_t(key, val));
    map_size++;
  }

  void remove(const key_t & key) {
    unsigned int i = Hash()(key) % table_size;
    typename std::list<entry_t>::iterator it = table[i].begin();
    while (it != table[i].end() && it->key != key) ++it;
    if (it == table[i].end()) return;
    table[i].erase(it);
    map_size--;
  }

  val_t * find(const key_t & key) {
    unsigned int i = Hash()(key) % table_size;
    typename std::list<entry_t>::iterator it = table[i].begin();
    while (it != table[i].end() && it->key != key) ++it;
    if (it == table[i].end()) return 0;
    return &(it->val);
  }

  val_t & operator [] (const key_t & key) {
    val_t * ret = find(key);
    if (ret != 0) return *ret;
    insert(key, val_t());
    return *find(key);
  }
};
```

```
/*** Examples of Hash Algorithm Definitions ***/

#include <string>

struct class_hash {
  unsigned int operator () (int key) {
    return class_hash()((unsigned int)key);
  }

  unsigned int operator () (long long key) {
    return class_hash()((unsigned long long)key);
  }

  //Knuth's multiplicative method (one-to-one)
  unsigned int operator () (unsigned int key) {
    return key * 2654435761u; //or just return key
  }

  //Jenkins's 64-bit hash
  unsigned int operator () (unsigned long long key) {
    key += ~(key << 32); key ^= (key >> 22);
    key += ~(key << 13); key ^= (key >>  8);
    key +=  (key <<  3); key ^= (key >> 15);
    key += ~(key << 27); key ^= (key >> 31);
    return key;
  }

  //Jenkins's one-at-a-time hash
  unsigned int operator () (const std::string & key) {
    unsigned int hash = 0;
    for (unsigned int i = 0; i < key.size(); i++) {
      hash += ((hash += key[i]) << 10);
      hash ^= (hash >> 6);
    }
    hash ^= ((hash += (hash << 3)) >> 11);
    return hash + (hash << 15);
  }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  hashmap<string, int, class_hash> M;
  M["foo"] = 1;
  M.insert("bar", 2);
  cout << M["foo"] << M["bar"] << endl; //prints 12
  cout << M["baz"] << M["qux"] << endl; //prints 00
  M.remove("foo");
  cout << M.size() << endl;             //prints 3
  cout << M["foo"] << M["bar"] << endl; //prints 02
  return 0;
}
```

3.5.5 Skip List (Probabilistic)

```
/*

Description: A skip list is an alternative to binary search trees.
Fast search is made possible by maintaining a linked hierarchy of
subsequences, each skipping over fewer elements. Searching starts
in the sparsest subsequence until two consecutive elements have
been found, one smaller and one larger than the element searched for.
Skip lists are generally slower than binary search trees, but can
be easier to implement. The following version uses randomized levels.

Time Complexity: insert(), erase(), count() and find() are O(log(N))
on average, but O(N) in the worst case. walk() is O(N).

Space Complexity: O(N) on the number of elements inserted on average,
but O(N log N) in the worst case.

*/

#include <cmath>   /* log() */
#include <cstdlib> /* rand(), srand() */
#include <cstring> /* memset() */
#include <ctime>   /* time() */

template<class key_t, class val_t> struct skip_list {
  static const int MAX_LEVEL = 32; //~ log2(max # of keys)

  static int random_level() { //geometric distribution
    static const float P = 0.5;
    int lvl = log((float)rand()/RAND_MAX)/log(1.0 - P);
    return lvl < MAX_LEVEL ? lvl : MAX_LEVEL;
  }

  struct node_t {
    key_t key;
    val_t val;
    node_t **next;

    node_t(int level, const key_t & k, const val_t & v) {
      next = new node_t * [level + 1];
      memset(next, 0, sizeof(node_t*)*(level + 1));
      key = k;
      val = v;
    }

    ~node_t() { delete[] next; }
  } *head, *update[MAX_LEVEL + 1];

  int level, num_nodes;

  skip_list() {
    srand(time(0));
    head = new node_t(MAX_LEVEL, key_t(), val_t());
    level = num_nodes = 0;
  }

  ~skip_list() { delete head; }
  int size() { return num_nodes; }
  bool empty() { return num_nodes == 0; }
  int count(const key_t & k) { return find(k) != 0; }

  void insert(const key_t & k, const val_t & v) {
    node_t * n = head;
    memset(update, 0, sizeof(node_t*)*(MAX_LEVEL + 1));
    for (int i = level; i >= 0; i--) {
      while (n->next[i] && n->next[i]->key < k) n = n->next[i];
      update[i] = n;
    }
    n = n->next[0];
    if (!n || n->key != k) {
      int lvl = random_level();
      if (lvl > level) {
        for (int i = level + 1; i <= lvl; i++) update[i] = head;
        level = lvl;
      }
      n = new node_t(lvl, k, v);
      num_nodes++;
      for (int i = 0; i <= lvl; i++) {
        n->next[i] = update[i]->next[i];
        update[i]->next[i] = n;
      }
    } else if (n && n->key == k && n->val != v) {
      n->val = v;
    }
  }

  void erase(const key_t & k) {
    node_t * n = head;
    memset(update, 0, sizeof(node_t*)*(MAX_LEVEL + 1));
    for (int i = level;i >= 0; i--) {
      while (n->next[i] && n->next[i]->key < k) n = n->next[i];
      update[i] = n;
    }
    n = n->next[0];
    if (n->key == k) {
      for (int i = 0; i <= level; i++) {
        if (update[i]->next[i] != n) break;
        update[i]->next[i] = n->next[i];
      }
      delete n;
      num_nodes--;
      while (level > 0 && !head->next[level]) level--;
    }
  }

  val_t * find(const key_t & k) {
    node_t * n = head;
    for (int i = level; i >= 0; i--)
      while (n->next[i] && n->next[i]->key < k)
        n = n->next[i];
    n = n->next[0];
    if (n && n->key == k) return &(n->val);
    return 0; //not found
  }

  template<class BinaryFunction> void walk(BinaryFunction f) {
    node_t *n = head->next[0];
    while (n) {
      f(n->key, n->val);
      n = n->next[0];
    }
  }
};

/*** Example Usage: Random Tests ***/

#include <cassert>
#include <iostream>
#include <map>
using namespace std;

int main() {
  map<int, int> m;
  skip_list<int, int> s;
  for (int i = 0; i < 50000; i++) {
    int op = rand() % 3;
    int val1 = rand(), val2 = rand();
    if (op == 0) {
      m[val1] = val2;
      s.insert(val1, val2);
    } else if (op == 1) {
      if (!m.count(val1)) continue;
      m.erase(val1);
      s.erase(val1);
    } else if (op == 2) {
      assert(s.count(val1) == (int)m.count(val1));
      if (m.count(val1)) {
        assert(m[val1] == *s.find(val1));
      }
    }
  }
  return 0;
}
```

3.6 Tree Data Structures

3.6.1 Heavy-Light Decomposition

```
/*

Description: Given an undirected, connected graph that is a tree, the
heavy-light decomposition (HLD) on the graph is a partitioning of the
vertices into disjoint paths to later support dynamic modification and
querying of values on paths between pairs of vertices.
See: http://wcipeg.com/wiki/Heavy-light_decomposition
and: http://blog.anudeep2011.com/heavy-light-decomposition/
To support dynamic adding and removal of edges, see link/cut tree.

Note: The adjacency list tree[] that is passed to the constructor must
not be changed afterwards in order for modify() and query() to work.

Time Complexity: O(N) for the constructor and O(log N) in the worst
case for both modify() and query(), where N is the number of vertices.

Space Complexity: O(N) on the number of vertices in the tree.
```

```

*/

#include <algorithm> /* std::max(), std::min() */
#include <climits>   /* INT_MIN */
#include <vector>

template<class T> class heavy_light {
  //true if you want values on edges, false if you want values on vertices
  static const bool VALUES_ON_EDGES = true;

  //Modify the following 6 functions to implement your custom
  //operations on the tree. This implements the Add/Max operations.
  //Operations like Add/Sum, Set/Max can also be implemented.
  static inline T modify_op(const T & x, const T & y) {
    return x + y;
  }

  static inline T query_op(const T & x, const T & y) {
    return std::max(x, y);
  }

  static inline T delta_on_segment(const T & delta, int seglen) {
    if (delta == null_delta()) return null_delta();
    //Here you must write a fast equivalent of following slow code:
    //  T result = delta;
    //  for (int i = 1; i < seglen; i++) result = query_op(result, delta);
    //  return result;
    return delta;
  }

  static inline T init_value() { return 0; }
  static inline T null_delta() { return 0; }
  static inline T null_value() { return INT_MIN; }

  static inline T join_value_with_delta(const T & v, const T & delta) {
    return delta == null_delta() ? v : modify_op(v, delta);
  }

  static T join_deltas(const T & delta1, const T & delta2) {
    if (delta1 == null_delta()) return delta2;
    if (delta2 == null_delta()) return delta1;
    return modify_op(delta1, delta2);
  }

  int counter, paths;
  std::vector<int> *adj;
  std::vector<std::vector<T> > value, delta;
  std::vector<std::vector<int> > len;
  std::vector<int> size, parent, tin, tout;
  std::vector<int> path, pathlen, pathpos, pathroot;

  void precompute_dfs(int u, int p) {
    tin[u] = counter++;
    parent[u] = p;
    size[u] = 1;
    for (int j = 0, v; j < (int)adj[u].size(); j++) {
      if ((v = adj[u][j]) == p) continue;
      precompute_dfs(v, u);
      size[u] += size[v];
    }
    tout[u] = counter++;
  }

  int new_path(int u) {
    pathroot[paths] = u;
    return paths++;
  }

  void build_paths(int u, int path) {
    this->path[u] = path;
    pathpos[u] = pathlen[path]++;
    for (int j = 0, v; j < (int)adj[u].size(); j++) {
      if ((v = adj[u][j]) == parent[u]) continue;
      build_paths(v, 2*size[v] >= size[u] ? path : new_path(v));
    }
  }

  inline T join_value_with_delta0(int path, int i) {
    return join_value_with_delta(value[path][i],
              delta_on_segment(delta[path][i], len[path][i]));
  }

  void push_delta(int path, int i) {
    int d = 0;
    while ((i >> d) > 0) d++;
    for (d -= 2; d >= 0; d--) {
      int x = i >> d;
      value[path][x >> 1] = join_value_with_delta0(path, x >> 1);
      delta[path][x] = join_deltas(delta[path][x], delta[path][x >> 1]);
      delta[path][x ^ 1] = join_deltas(delta[path][x ^ 1], delta[path][x >> 1]);
      delta[path][x >> 1] = null_delta();
    }
  }

  T query(int path, int a, int b) {
    push_delta(path, a += value[path].size() >> 1);
    push_delta(path, b += value[path].size() >> 1);
    T res = null_value();
    for (; a <= b; a = (a + 1) >> 1, b = (b - 1) >> 1) {
      if ((a & 1) != 0)
        res = query_op(res, join_value_with_delta0(path, a));
      if ((b & 1) == 0)
        res = query_op(res, join_value_with_delta0(path, b));
    }
    return res;
  }

  void modify(int path, int a, int b, const T & delta) {
    push_delta(path, a += value[path].size() >> 1);
    push_delta(path, b += value[path].size() >> 1);
    int ta = -1, tb = -1;
    for (; a <= b; a = (a + 1) >> 1, b = (b - 1) >> 1) {
      if ((a & 1) != 0) {
        this->delta[path][a] = join_deltas(this->delta[path][a], delta);
        if (ta == -1) ta = a;
      }
      if ((b & 1) == 0) {
        this->delta[path][b] = join_deltas(this->delta[path][b], delta);
        if (tb == -1) tb = b;
      }
    }
    for (int i = ta; i > 1; i >>= 1)
      value[path][i >> 1] = query_op(join_value_with_delta0(path, i),
                                     join_value_with_delta0(path, i ^ 1));
    for (int i = tb; i > 1; i >>= 1)
      value[path][i >> 1] = query_op(join_value_with_delta0(path, i),
                                     join_value_with_delta0(path, i ^ 1));
  }

  inline bool is_ancestor(int p, int ch) {
    return tin[p] <= tin[ch] && tout[ch] <= tout[p];
  }

 public:
  heavy_light(int N, std::vector<int> tree[]): size(N), parent(N),
   tin(N), tout(N), path(N), pathlen(N), pathpos(N), pathroot(N) {
    adj = tree;
    counter = paths = 0;
    precompute_dfs(0, -1);
    build_paths(0, new_path(0));
    value.resize(paths);
    delta.resize(paths);
    len.resize(paths);
    for (int i = 0; i < paths; i++) {
      int m = pathlen[i];
      value[i].assign(2*m, init_value());
      delta[i].assign(2*m, null_delta());
      len[i].assign(2*m, 1);
      for (int j = 2*m - 1; j > 1; j -= 2) {
        value[i][j >> 1] = query_op(value[i][j], value[i][j ^ 1]);
        len[i][j >> 1] = len[i][j] + len[i][j ^ 1];
      }
    }
  }

  T query(int a, int b) {
    T res = null_value();
    for (int root; !is_ancestor(root = pathroot[path[a]], b); a = parent[root])
      res = query_op(res, query(path[a], 0, pathpos[a]));
    for (int root; !is_ancestor(root = pathroot[path[b]], a); b = parent[root])
      res = query_op(res, query(path[b], 0, pathpos[b]));
    if (VALUES_ON_EDGES && a == b) return res;
    return query_op(res, query(path[a], std::min(pathpos[a], pathpos[b]) +
                          VALUES_ON_EDGES, std::max(pathpos[a], pathpos[b])));
  }

  void modify(int a, int b, const T & delta) {
    for (int root; !is_ancestor(root = pathroot[path[a]], b); a = parent[root])
      modify(path[a], 0, pathpos[a], delta);
    for (int root; !is_ancestor(root = pathroot[path[b]], a); b = parent[root])
      modify(path[b], 0, pathpos[b], delta);
    if (VALUES_ON_EDGES && a == b) return;
    modify(path[a], std::min(pathpos[a], pathpos[b]) + VALUES_ON_EDGES,
           std::max(pathpos[a], pathpos[b]), delta);
  }
};
```

```

/*** Example Usage ***/

#include <iostream>
using namespace std;

const int MAXN = 1000;
vector<int> adj[MAXN];

/*
     w=10      w=20      w=40
  0---------1---------2---------3
                      \
                       ---------4
                         w=30
*/
int main() {
  adj[0].push_back(1);
  adj[1].push_back(0);
  adj[1].push_back(2);
  adj[2].push_back(1);
  adj[2].push_back(3);
  adj[3].push_back(2);
  adj[2].push_back(4);
  adj[4].push_back(2);
  heavy_light<int> hld(5, adj);
  hld.modify(0, 1, 10);
  hld.modify(1, 2, 20);
  hld.modify(2, 3, 40);
  hld.modify(2, 4, 30);
  cout << hld.query(0, 3) << "\n"; //40
  cout << hld.query(2, 4) << "\n"; //30
  hld.modify(3, 4, 50); //w[every edge from 3 to 4] += 50
  cout << hld.query(1, 4) << "\n"; //80
  return 0;
}
```

3.6.2 Link-Cut Tree

```
/*

Description: Given an unweighted forest of trees where each node
has an associated value, a link/cut tree can be used to dynamically
query and modify values on the path between pairs of nodes a tree.
This problem can be solved using heavy-light decomposition, which
also supports having values stored on edges rather than the nodes.
However in a link/cut tree, nodes in different trees may be
dynamically linked, edges between nodes in the same tree may be
dynamically split, and connectivity between two nodes (whether they
are in the same tree) may be checked.

Time Complexity: O(log N) amortized for make_root(), link(), cut(),
connected(), modify(), and query(), where N is the number of nodes
in the forest.

Space Complexity: O(N) on the number of nodes in the forest.

```

```
*/

#include <algorithm> /* std::max(), std::swap() */
#include <climits>   /* INT_MIN */
#include <map>
#include <stdexcept> /* std::runtime_error() */

template<class T> class linkcut_forest {
  //Modify the following 5 functions to implement your custom
  //operations on the tree. This implements the Add/Max operations.
  //Operations like Add/Sum, Set/Max can also be implemented.
  static inline T modify_op(const T & x, const T & y) {
    return x + y;
  }

  static inline T query_op(const T & x, const T & y) {
    return std::max(x, y);
  }

  static inline T delta_on_segment(const T & delta, int seglen) {
    if (delta == null_delta()) return null_delta();
    //Here you must write a fast equivalent of following slow code:
    //  T result = delta;
    //  for (int i = 1; i < seglen; i++) result = query_op(result, delta);
    //  return result;
    return delta;
  }

  static inline T null_delta() { return 0; }
  static inline T null_value() { return INT_MIN; }

  static inline T join_value_with_delta(const T & v, const T & delta) {
    return delta == null_delta() ? v : modify_op(v, delta);
  }

  static T join_deltas(const T & delta1, const T & delta2) {
    if (delta1 == null_delta()) return delta2;
    if (delta2 == null_delta()) return delta1;
    return modify_op(delta1, delta2);
  }

  struct node_t {
    T value, subtree_value, delta;
    int size;
    bool rev;
    node_t *L, *R, *parent;

    node_t(const T & v) {
      value = subtree_value = v;
      delta = null_delta();
      size = 1;
      rev = false;
      L = R = parent = 0;
    }

    bool is_root() { //is this the root of a splay tree?
      return parent == 0 || (parent->L != this && parent->R != this);
    }

    void push() {
      if (rev) {
        rev = false;
        std::swap(L, R);
        if (L != 0) L->rev = !L->rev;
        if (R != 0) R->rev = !R->rev;
      }
      value = join_value_with_delta(value, delta);
      subtree_value = join_value_with_delta(subtree_value,
                        delta_on_segment(delta, size));
      if (L != 0) L->delta = join_deltas(L->delta, delta);
      if (R != 0) R->delta = join_deltas(R->delta, delta);
      delta = null_delta();
    }

    void update() {
      subtree_value = query_op(query_op(get_subtree_value(L),
                                        join_value_with_delta(value, delta)),
                               get_subtree_value(R));
      size = 1 + get_size(L) + get_size(R);
    }
  };

  static inline int get_size(node_t * n) {
    return n == 0 ? 0 : n->size;
  }

  static inline int get_subtree_value(node_t * n) {
    return n == 0 ? null_value() : join_value_with_delta(n->subtree_value,
                                     delta_on_segment(n->delta, n->size));
  }

  static void connect(node_t * ch, node_t * p, char is_left) {
    if (ch != 0) ch->parent = p;
    if (is_left < 0) return;
    (is_left ? p->L : p->R) = ch;
  }

  /** rotates edge (n, n.parent)
   *         g          g
   *        /          /
   *       p          n
   *      / \  -->   / \
   *     n  p.r    n.l  p
   *    / \            / \
   *  n.l n.r        n.r p.r
   */
  static void rotate(node_t * n) {
    node_t *p = n->parent, *g = p->parent;
    bool is_rootp = p->is_root(), is_left = (n == p->L);
    connect(is_left ? n->R : n->L, p, is_left);
    connect(p, n, !is_left);
    connect(n, g, is_rootp ? -1 : (p == g->L));
    p->update();
  }

  /** brings n to the root, balancing tree
   *
   *  zig-zig case:
```

```
   *         g                                  n
   *        / \               p                / \
   *       p  g.r rot(p)    /   \     rot(n) n.l  p
   *      / \      -->    n       g    -->       / \
   *     n  p.r          / \     / \           n.r  g
   *    / \            n.l n.r p.r g.r             / \
   *  n.l n.r                                  p.r g.r
   *
   *  zig-zag case:
   *       g               g
   *      / \             / \               n
   *     p  g.r rot(n)   n  g.r rot(n)    /   \
   *    / \      -->    / \      -->    p       g
   *  p.l  n           p  n.r          / \     / \
   *      / \         / \            p.l n.l n.r g.r
   *    n.l n.r     p.l n.l
   */
  static void splay(node_t * n) {
    while (!n->is_root()) {
      node_t *p = n->parent, *g = p->parent;
      if (!p->is_root()) g->push();
      p->push();
      n->push();
      if (!p->is_root())
        rotate((n == p->L) == (p == g->L) ? p/*zig-zig*/ : n/*zig-zag*/);
      rotate(n);
    }
    n->push();
    n->update();
  }

  //makes node n the root of the virtual tree,
  //and also n becomes the leftmost node in its splay tree
  static node_t * expose(node_t * n) {
    node_t *prev = 0;
    for (node_t *i = n; i != 0; i = i->parent) {
      splay(i);
      i->L = prev;
      prev = i;
    }
    splay(n);
    return prev;
  }

  std::map<int, node_t*> nodes; //use array if ID compression not required
  node_t *u, *v; //temporary

  void get_uv(int a, int b) {
    static typename std::map<int, node_t*>::iterator it1, it2;
    it1 = nodes.find(a);
    it2 = nodes.find(b);
    if (it1 == nodes.end() || it2 == nodes.end())
      throw std::runtime_error("Error:␣a␣or␣b␣does␣not␣exist␣in␣forest.");
    u = it1->second;
    v = it2->second;
  }

 public:
  ~linkcut_forest() {
    static typename std::map<int, node_t*>::iterator it;
    for (it = nodes.begin(); it != nodes.end(); ++it)
      delete it->second;
  }

  void make_root(int id, const T & initv) {
    if (nodes.find(id) != nodes.end())
      throw std::runtime_error("Cannot␣make_root():␣ID␣already␣exists.");
    node_t * n = new node_t(initv);
    expose(n);
    n->rev = !n->rev;
    nodes[id] = n;
  }

  bool connected(int a, int b) {
    get_uv(a, b);
    if (a == b) return true;
    expose(u);
    expose(v);
    return u->parent != 0;
  }

  void link(int a, int b) {
    if (connected(a, b))
      throw std::runtime_error("Error:␣a␣and␣b␣are␣already␣connected.");
    get_uv(a, b);
    expose(u);
    u->rev = !u->rev;
    u->parent = v;
  }

  void cut(int a, int b) {
    get_uv(a, b);
    expose(u);
    u->rev = !u->rev;
    expose(v);
    if (v->R != u || u->L != 0)
      throw std::runtime_error("Error:␣edge␣(a,␣b)␣does␣not␣exist.");
    v->R->parent = 0;
    v->R = 0;
  }

  T query(int a, int b) {
    if (!connected(a, b))
      throw std::runtime_error("Error:␣a␣and␣b␣are␣not␣connected.");
    get_uv(a, b);
    expose(u);
    u->rev = !u->rev;
    expose(v);
    return get_subtree_value(v);
  }

  void modify(int a, int b, const T & delta) {
    if (!connected(a, b))
      throw std::runtime_error("Error:␣a␣and␣b␣are␣not␣connected.");
    get_uv(a, b);
    expose(u);
    u->rev = !u->rev;
    expose(v);
    v->delta = join_deltas(v->delta, delta);
  }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  linkcut_forest<int> F;
/*
 v=10      v=40      v=20      v=10
  0---------1---------2---------3
                      \
                       ---------4
                               v=30
*/
  F.make_root(0, 10);
  F.make_root(1, 40);
  F.make_root(2, 20);
  F.make_root(3, 10);
  F.make_root(4, 30);
  F.link(0, 1);
  F.link(1, 2);
  F.link(2, 3);
  F.link(2, 4);
  cout << F.query(1, 4) << "\n"; //40
  F.modify(1, 1, -10);
  F.modify(3, 4, -10);
/*
 v=10      v=30      v=10      v=0
  0---------1---------2---------3
                      \
                       ---------4
                               v=20
*/
  cout << F.query(0, 4) << "\n"; //30
  cout << F.query(3, 4) << "\n"; //20
  F.cut(1, 2);
  cout << F.connected(1, 2) << "\n"; //0
  cout << F.connected(0, 4) << "\n"; //0
  cout << F.connected(2, 3) << "\n"; //1
  return 0;
}
```

3.7 Lowest Common Ancestor

3.7.1 Sparse Tables

```
/*

Description: Given an undirected graph that is a tree, the
lowest common ancestor (LCA) of two nodes v and w is the
lowest (i.e. deepest) node that has both v and w as descendants,
where we define each node to be a descendant of itself (so if
v has a direct connection from w, w is the lowest common
ancestor). The following program uses sparse tables to solve
the problem on an unchanging graph.

Time Complexity: O(N log N) for build() and O(log N) for lca(),
where N is the number of nodes in the tree.

Space Complexity: O(N log N).

*/

#include <vector>

const int MAXN = 1000;
int len, timer, tin[MAXN], tout[MAXN];
std::vector<int> adj[MAXN], dp[MAXN];

void dfs(int u, int p) {
  tin[u] = timer++;
  dp[u][0] = p;
  for (int i = 1; i < len; i++)
    dp[u][i] = dp[dp[u][i - 1]][i - 1];
  for (int j = 0, v; j < (int)adj[u].size(); j++)
    if ((v = adj[u][j]) != p)
      dfs(v, u);
  tout[u] = timer++;
}

void build(int nodes, int root) {
  len = 1;
  while ((1 << len) <= nodes) len++;
  for (int i = 0; i < nodes; i++)
    dp[i].resize(len);
  timer = 0;
  dfs(root, root);
}

inline bool is_parent(int parent, int child) {
  return tin[parent] <= tin[child] && tout[child] <= tout[parent];
}

int lca(int a, int b) {
  if (is_parent(a, b)) return a;
  if (is_parent(b, a)) return b;
  for (int i = len - 1; i >= 0; i--)
    if (!is_parent(dp[a][i], b))
      a = dp[a][i];
  return dp[a][0];
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  adj[0].push_back(1);
  adj[1].push_back(0);
  adj[1].push_back(2);
  adj[2].push_back(1);
  adj[3].push_back(1);
  adj[1].push_back(3);
  adj[0].push_back(4);
  adj[4].push_back(0);
  build(5, 0);
  cout << lca(3, 2) << "\n"; //1
  cout << lca(2, 4) << "\n"; //0
  return 0;
}
```

3.7.2 Segment Trees

```
/*

Description: Given a rooted tree, the lowest common ancestor (LCA)
of two nodes v and w is the lowest (i.e. deepest) node that has
both v and w as descendants, where we define each node to be a
descendant of itself (so if v has a direct connection from w, w
is the lowest common ancestor). This problem can be reduced to the
range minimum query problem using Eulerian tours.

Time Complexity: O(N log N) for build() and O(log N) for lca(),
where N is the number of nodes in the tree.

Space Complexity: O(N log N).

*/

#include <algorithm> /* std::fill(), std::min(), std::max() */
#include <vector>

const int MAXN = 1000;
int len, counter;
int depth[MAXN], dfs_order[2*MAXN], first[MAXN], minpos[8*MAXN];
std::vector<int> adj[MAXN];

void dfs(int u, int d) {
  depth[u] = d;
  dfs_order[counter++] = u;
  for (int j = 0, v; j < (int)adj[u].size(); j++) {
    if (depth[v = adj[u][j]] == -1) {
      dfs(v, d + 1);
      dfs_order[counter++] = u;
    }
  }
}

void build_tree(int n, int l, int h) {
  if (l == h) {
    minpos[n] = dfs_order[l];
    return;
  }
  int lchild = 2 * n + 1, rchild = 2 * n + 2;
  build_tree(lchild, l, (l + h)/2);
  build_tree(rchild, (l + h) / 2 + 1, h);
  minpos[n] = depth[minpos[lchild]] < depth[minpos[rchild]] ?
```

```
              minpos[lchild] : minpos[rchild];
}

void build(int nodes, int root) {
  std::fill(depth, depth + nodes, -1);
  std::fill(first, first + nodes, -1);
  len = 2*nodes - 1;
  counter = 0;
  dfs(root, 0);
  build_tree(0, 0, len - 1);
  for (int i = 0; i < len; i++)
    if (first[dfs_order[i]] == -1)
      first[dfs_order[i]] = i;
}

int get_minpos(int a, int b, int n, int l, int h) {
  if (a == l && b == h) return minpos[n];
  int mid = (l + h) >> 1;
  if (a <= mid && b > mid) {
    int p1 = get_minpos(a, std::min(b, mid), 2 * n + 1, l, mid);
    int p2 = get_minpos(std::max(a, mid + 1), b, 2 * n + 2, mid + 1, h);
    return depth[p1] < depth[p2] ? p1 : p2;
  }
  if (a <= mid) return get_minpos(a, std::min(b, mid), 2 * n + 1, l, mid);
  return get_minpos(std::max(a, mid + 1), b, 2 * n + 2, mid + 1, h);
}

int lca(int a, int b) {
  return get_minpos(std::min(first[a], first[b]),
                    std::max(first[a], first[b]), 0, 0, len - 1);
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  adj[0].push_back(1);
  adj[1].push_back(0);
  adj[1].push_back(2);
  adj[2].push_back(1);
  adj[3].push_back(1);
  adj[1].push_back(3);
  adj[0].push_back(4);
  adj[4].push_back(0);
  build(5, 0);
  cout << lca(3, 2) << "\n"; //1
  cout << lca(2, 4) << "\n"; //0
  return 0;
}
```

Chapter 4

Mathematics

4.1 Mathematics Toolbox

```
/*

Useful math definitions. Excludes geometry (see next chapter).

*/

#include <algorithm> /* std::reverse() */
#include <cfloat>    /* DBL_MAX */
#include <cmath>     /* a lot of things */
#include <string>
#include <vector>

/* Definitions for Common Floating Point Constants */

const double PI = acos(-1.0), E = exp(1.0), root2 = sqrt(2.0);
const double phi = (1.0 + sqrt(5.0)) / 2.0; //golden ratio

//Sketchy but working defintions of +infinity, -infinity and quiet NaN
//A better way is using functions of std::numeric_limits<T> from <limits>
//See main() for identities involving the following special values.
const double posinf = 1.0 / 0.0, neginf = -1.0 / 0.0, NaN = -(0.0 / 0.0);

/*

Epsilon Comparisons

The range of values for which X compares EQ() to is [X - eps, X + eps].
For values to compare LT() and GT() x, they must fall outside of the range.

e.g. if eps = 1e-7, then EQ(1e-8, 2e-8) is true and LT(1e-8, 2e-8) is false.

*/

const double eps = 1e-7;
#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define NE(a, b) (fabs((a) - (b)) > eps)  /* not equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */
```

```
#define GT(a, b) ((a) > (b) + eps)        /* greater than */
#define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */
#define GE(a, b) ((a) >= (b) - eps)       /* greater than or equal to */

/*

Sign Function:

Returns: -1 (if x < 0), 0 (if x == 0), or 1 (if x > 0)
Doesn't handle the sign of NaN like signbit() or copysign()

*/

template<class T> int sgn(const T & x) {
  return (T(0) < x) - (x < T(0));
}

/*

signbit() and copysign() functions, only in C++11 and later.

signbit() returns whether the sign bit of the floating point
number is set to true. If signbit(x), then x is "negative."
Note that signbit(0.0) == 0 but signbit(-0.0) == 1. This
also works as expected on NaN, -NaN, posinf, and neginf.

We implement this by casting the floating point value to an
integer type with the same number of bits so we can perform
shift operations on it, then we extract the sign bit.
Another way is using unions, but this is non-portable
depending on endianess of the platform. Unfortunately, we
cannot find the signbit of long doubles using the method
below because there is no corresponding 96-bit integer type.
Note that this will cause complaints with the compiler.

copysign(x, y) returns a number with the magnitude of x but
the sign of y.

Assumptions:  sizeof(float) == sizeof(int) and
              sizeof(long long) == sizeof(double)
              CHAR_BITS == 8 (8 bits to a byte)

*/

inline bool signbit(float x) {
  return (*(int*)&x) >> (sizeof(float) * 8 - 1);
}

inline bool signbit(double x) {
  return (*(long long*)&x) >> (sizeof(double) * 8 - 1);
}

template<class Double>
inline Double copysign(Double x, Double y) {
  return signbit(y) ? -fabs(x) : fabs(x);
}

/*

```

```
Floating Point Rounding Functions

floor() in <cmath> asymmetrically rounds down, towards -infinity,
while ceil() in <cmath> asymmetrically rounds up, towards +infinity.
The following are common alternative ways to round.

*/

//symmetric round down, bias: towards zero (same as trunc() in C++11)
template<class Double> Double floor0(const Double & x) {
  Double res = floor(fabs(x));
  return (x < 0.0) ? -res : res;
}

//symmetric round up, bias: away from zero
template<class Double> Double ceil0(const Double & x) {
  Double res = ceil(fabs(x));
  return (x < 0.0) ? -res : res;
}

//round half up, bias: towards +infinity
template<class Double> Double roundhalfup(const Double & x) {
  return floor(x + 0.5);
}

//round half up, bias: towards -infinity
template<class Double> Double roundhalfdown(const Double & x) {
  return ceil(x - 0.5);
}

//symmetric round half down, bias: towards zero
template<class Double> Double roundhalfdown0(const Double & x) {
  Double res = roundhalfdown(fabs(x));
  return (x < 0.0) ? -res : res;
}

//symmetric round half up, bias: away from zero
template<class Double> Double roundhalfup0(const Double & x) {
  Double res = roundhalfup(fabs(x));
  return (x < 0.0) ? -res : res;
}

//round half to even (banker's rounding), bias: none
template<class Double>
Double roundhalfeven(const Double & x, const Double & eps = 1e-7) {
  if (x < 0.0) return -roundhalfeven(-x, eps);
  Double ipart;
  modf(x, &ipart);
  if (x - (ipart + 0.5) < eps)
    return (fmod(ipart, 2.0) < eps) ? ipart : ceil0(ipart + 0.5);
  return roundhalfup0(x);
}

//round alternating up/down for ties, bias: none for sequential calls
template<class Double> Double roundalternate(const Double & x) {
  static bool up = true;
  return (up = !up) ? roundhalfup(x) : roundhalfdown(x);
}

```

```
//symmetric round alternate, bias: none for sequential calls
template<class Double> Double roundalternate0(const Double & x) {
  static bool up = true;
  return (up = !up) ? roundhalfup0(x) : roundhalfdown0(x);
}

//round randomly for tie-breaking, bias: generator's bias
template<class Double> Double roundrandom(const Double & x) {
  return (rand() % 2 == 0) ? roundhalfup0(x) : roundhalfdown0(x);
}

//round x to N digits after the decimal using the specified round function
//e.g. roundplaces(-1.23456, 3, roundhalfdown0<double>) returns -1.235
template<class Double, class RoundFunction>
double roundplaces(const Double & x, unsigned int N, RoundFunction f) {
  return f(x * pow(10, N)) / pow(10, N);
}

/*

Error Function (erf() and erfc() in C++11)

erf(x) = 2/sqrt(pi) * integral of exp(-t^2) dt from 0 to x
erfc(x) = 1 - erf(x)
Note that the functions are co-dependent.

Adapted from: http://www.digitalmars.com/archives/cplusplus/3634.html#N3655

*/

//calculate 12 significant figs (don't ask for more than 1e-15)
static const double rel_error = 1e-12;

double erf(double x) {
  if (signbit(x)) return -erf(-x);
  if (fabs(x) > 2.2) return 1.0 - erfc(x);
  double sum = x, term = x, xsqr = x * x;
  int j = 1;
  do {
    term *= xsqr / j;
    sum -= term / (2 * (j++) + 1);
    term *= xsqr / j;
    sum += term / (2 * (j++) + 1);
  } while (fabs(term) / sum > rel_error);
  return 1.128379167095512574 * sum; //1.128 ~ 2/sqrt(pi)
}

double erfc(double x) {
  if (fabs(x) < 2.2) return 1.0 - erf(x);
  if (signbit(x)) return 2.0 - erfc(-x);
  double a = 1, b = x, c = x, d = x * x + 0.5, q1, q2 = 0, n = 1.0, t;
  do {
    t = a * n + b * x; a = b; b = t;
    t = c * n + d * x; c = d; d = t;
    n += 0.5;
    q1 = q2;
    q2 = b / d;
  } while (fabs(q1 - q2) / q2 > rel_error);
  return 0.564189583547756287 * exp(-x * x) * q2; //0.564 ~ 1/sqrt(pi)
```

```
}

/*

Gamma and Log-Gamma Functions (tgamma() and lgamma() in C++11)
Warning: unlike the actual standard C++ versions, the following
function only works on positive numbers (returns NaN if x <= 0).
Adapted from: http://www.johndcook.com/blog/cpp_gamma/

*/

double lgamma(double x);

double tgamma(double x) {
  if (x <= 0.0) return NaN;
  static const double gamma = 0.577215664901532860606512090;
  if (x < 1e-3) return 1.0 / (x * (1.0 + gamma * x));
  if (x < 12.0) {
    double y = x;
    int n = 0;
    bool arg_was_less_than_one = (y < 1.0);
    if (arg_was_less_than_one) y += 1.0;
    else y -= (n = static_cast<int>(floor(y)) - 1);
    static const double p[] = {
      -1.71618513886549492533811E+0, 2.47656508055759199108314E+1,
      -3.79804256470945635097577E+2, 6.29331155312818442661052E+2,
       8.66966202790413211295064E+2,-3.14512729688483675254357E+4,
      -3.61444134186911729807069E+4, 6.64561438202405440627855E+4
    };
    static const double q[] = {
      -3.08402300119738975254353E+1, 3.15350626979604161529144E+2,
      -1.01515636749021914166146E+3,-3.10777167157231109440444E+3,
       2.25381184209801510330112E+4, 4.75584627752788110767815E+3,
      -1.34659959864969306392456E+5,-1.15132259675553483497211E+5
    };
    double num = 0.0, den = 1.0, z = y - 1;
    for (int i = 0; i < 8; i++) {
      num = (num + p[i]) * z;
      den = den * z + q[i];
    }
    double result = num / den + 1.0;
    if (arg_was_less_than_one) result /= (y - 1.0);
    else for (int i = 0; i < n; i++) result *= y++;
    return result;
  }
  return (x > 171.624) ? DBL_MAX * 2.0 : exp(lgamma(x));
}

double lgamma(double x) {
  if (x <= 0.0) return NaN;
  if (x < 12.0) return log(fabs(tgamma(x)));
  static const double c[8] = {
    1.0/12.0, -1.0/360.0, 1.0/1260.0, -1.0/1680.0, 1.0/1188.0,
    -691.0/360360.0, 1.0/156.0, -3617.0/122400.0
  };
  double z = 1.0 / (x * x), sum = c[7];
  for (int i = 6; i >= 0; i--) sum = sum * z + c[i];
  static const double halflog2pi = 0.91893853320467274178032973640562;
  return (x - 0.5) * log(x) - x + halflog2pi + sum / x;
```

```
}

/*

Base Conversion - O(N) on the number of digits

Given the digits of an integer x in base a, returns x’s digits in base b.
Precondition: the base-10 value of x must be able to fit within an unsigned
long long. In other words, the value of x must be between 0 and 2^64 - 1.

Note: vector[0] stores the most significant digit in all usages below.

e.g. if x = {1, 2, 3} and a = 5 (i.e. x = 123 in base 5 = 38 in base 10),
then convert_base(x, 5, 3) returns {1, 1, 0, 2} (1102 in base 2).

*/

std::vector<int> convert_base(const std::vector<int> & x, int a, int b) {
  unsigned long long base10 = 0;
  for (int i = 0; i < (int)x.size(); i++)
    base10 += x[i] * pow(a, x.size() - i - 1);
  int N = ceil(log(base10 + 1) / log(b));
  std::vector<int> baseb;
  for (int i = 1; i <= N; i++)
    baseb.push_back(int(base10 / pow(b, N - i)) % b);
  return baseb;
}

//returns digits of a number in base b
std::vector<int> base_digits(int x, int b = 10) {
  std::vector<int> baseb;
  while (x != 0) {
    baseb.push_back(x % b);
    x /= b;
  }
  std::reverse(baseb.begin(), baseb.end());
  return baseb;
}

/*

Integer to Roman Numerals Conversion

Given an integer x, this function returns the Roman numeral representation
of x as a C++ string. More ’M’s are appended to the front of the resulting
string if x is greater than 1000. e.g. to_roman(1234) returns "MCCXXXIV"
and to_roman(5678) returns "MMMMMDCLXXVIII".

*/

std::string to_roman(unsigned int x) {
  static std::string h[] = {"","C","CC","CCC","CD","D","DC","DCC","DCCC","CM"};
  static std::string t[] = {"","X","XX","XXX","XL","L","LX","LXX","LXXX","XC"};
  static std::string o[] = {"","I","II","III","IV","V","VI","VII","VIII","IX"};
  std::string res(x / 1000, ’M’);
  x %= 1000;
  return res + h[x / 100] + t[x / 10 % 10] + o[x % 10];
}

```

```cpp
/*** Example Usage ***/

#include <algorithm>
#include <cassert>
#include <iostream>
using namespace std;

int main() {
  cout << "PI:␣" << PI << "\n";
  cout << "E:␣" << E << "\n";
  cout << "sqrt(2):␣" << root2 << "\n";
  cout << "Golden␣ratio:␣" << phi << "\n";

  //some properties of posinf, neginf, and NaN:
  double x = -1234.567890; //any normal value of x will work
  assert((posinf > x) && (neginf < x) && (posinf == -neginf));
  assert((posinf + x == posinf) && (posinf - x == posinf));
  assert((neginf + x == neginf) && (neginf - x == neginf));
  assert((posinf + posinf == posinf) && (neginf - posinf == neginf));
  assert((NaN != x) && (NaN != NaN) && (NaN != posinf) && (NaN != neginf));
  assert(!(NaN < x) && !(NaN > x) && !(NaN <= x) && !(NaN >= x));
  assert(isnan(0.0*posinf) && isnan(0.0*neginf) && isnan(posinf/neginf));
  assert(isnan(NaN) && isnan(-NaN) && isnan(NaN*x + x - x/-NaN));
  assert(isnan(neginf-neginf) && isnan(posinf-posinf) && isnan(posinf+neginf));
  assert(!signbit(NaN) && signbit(-NaN) && !signbit(posinf) && signbit(neginf));

  assert(copysign(1.0, +2.0) == +1.0 && copysign(posinf, -2.0) == neginf);
  assert(copysign(1.0, -2.0) == -1.0 && signbit(copysign(NaN, -2.0)));
  assert(sgn(-1.234) == -1 && sgn(0.0) == 0 && sgn(5678) == 1);

  assert(EQ(floor0(1.5), 1.0) && EQ(floor0(-1.5), -1.0));
  assert(EQ(ceil0(1.5), 2.0) && EQ(ceil0(-1.5), -2.0));
  assert(EQ(roundhalfup(1.5), 2.0) && EQ(roundhalfup(-1.5), -1.0));
  assert(EQ(roundhalfdown(1.5), 1.0) && EQ(roundhalfdown(-1.5), -2.0));
  assert(EQ(roundhalfup0(1.5), 2.0) && EQ(roundhalfup0(-1.5), -2.0));
  assert(EQ(roundhalfdown0(1.5), 1.0) && EQ(roundhalfdown0(-1.5), -1.0));
  assert(EQ(roundhalfeven(1.5), 2.0) && EQ(roundhalfeven(-1.5), -2.0));
  assert(NE(roundalternate(1.5), roundalternate(1.5)));
  assert(EQ(roundplaces(-1.23456, 3, roundhalfdown0<double>), -1.235));

  assert(EQ(erf(1.0), 0.8427007929) && EQ(erf(-1.0), -0.8427007929));
  assert(EQ(tgamma(0.5), 1.7724538509) && EQ(tgamma(1.0), 1.0));
  assert(EQ(lgamma(0.5), 0.5723649429) && EQ(lgamma(1.0), 0.0));

  int base10digs[] = {1, 2, 3, 4, 5, 6}, a = 20, b = 10;
  vector<int> basea = base_digits(123456, a);
  vector<int> baseb = convert_base(basea, a, b);
  assert(equal(baseb.begin(), baseb.end(), base10digs));

  assert(to_roman(1234) == "MCCXXXIV");
  assert(to_roman(5678) == "MMMMMDCLXXVIII");
  return 0;
}
```

4.2 Combinatorics

4.2.1 Combinatorial Calculations

```cpp
/*

The meanings of the following functions can respectively be
found with quick searches online. All of them computes the
answer modulo m, since contest problems typically ask us for
this due to the actual answer being potentially very large.
All functions using tables to generate every answer below
n and k can be optimized using recursion and memoization.

Note: The following are only defined for nonnegative inputs.

*/

#include <vector>

typedef std::vector<std::vector<long long> > table;

//n! mod m in O(n)
long long factorial(int n, int m = 1000000007) {
  long long res = 1;
  for (int i = 2; i <= n; i++) res = (res * i) % m;
  return res % m;
}

//n! mod p, where p is a prime number, in O(p log n)
long long factorialp(long long n, long long p = 1000000007) {
  long long res = 1, h;
  while (n > 1) {
    res = (res * ((n / p) % 2 == 1 ? p - 1 : 1)) % p;
    h = n % p;
    for (int i = 2; i <= h; i++) res = (res * i) % p;
    n /= p;
  }
  return res % p;
}

//first n rows of pascal's triangle (mod m) in O(n^2)
table binomial_table(int n, long long m = 1000000007) {
  table t(n + 1);
  for (int i = 0; i <= n; i++)
    for (int j = 0; j <= i; j++)
      if (i < 2 || j == 0 || i == j)
        t[i].push_back(1);
      else
        t[i].push_back((t[i - 1][j - 1] + t[i - 1][j]) % m);
  return t;
}

//if the product of two 64-bit ints (a*a, a*b, or b*b) can
//overflow, you must use mulmod (multiplication by adding)
long long powmod(long long a, long long b, long long m) {
  long long x = 1, y = a;
  for (; b > 0; b >>= 1) {
    if (b & 1) x = (x * y) % m;
    y = (y * y) % m;
  }
  return x % m;
}

//n choose k (mod a prime number p) in O(min(k, n - k))
//powmod is used to find the mod inverse to get num / den % m
long long choose(int n, int k, long long p = 1000000007) {
  if (n < k) return 0;
  if (k > n - k) k = n - k;
  long long num = 1, den = 1;
  for (int i = 0; i < k; i++)
    num = (num * (n - i)) % p;
  for (int i = 1; i <= k; i++)
    den = (den * i) % p;
  return num * powmod(den, p - 2, p) % p;
}

//n multichoose k (mod a prime number p) in O(k)
long long multichoose(int n, int k, long long p = 1000000007) {
  return choose(n + k - 1, k, p);
}

//n permute k (mod m) on O(k)
long long permute(int n, int k, long long m = 1000000007) {
  if (n < k) return 0;
  long long res = 1;
  for (int i = 0; i < k; i++)
    res = (res * (n - i)) % m;
  return res % m;
}

//number of partitions of n (mod m) in O(n^2)
long long partitions(int n, long long m = 1000000007) {
  std::vector<long long> p(n + 1, 0);
  p[0] = 1;
  for (int i = 1; i <= n; i++)
    for (int j = i; j <= n; j++)
      p[j] = (p[j] + p[j - i]) % m;
  return p[n] % m;
}

//partitions of n into exactly k parts (mod m) in O(n * k)
long long partitions(int n, int k, long long m = 1000000007) {
  table t(n + 1, std::vector<long long>(k + 1, 0));
  t[0][1] = 1;
  for (int i = 1; i <= n; i++)
    for (int j = 1, h = k < i ? k : i; j <= h; j++)
      t[i][j] = (t[i - 1][j - 1] + t[i - j][j]) % m;
  return t[n][k] % m;
}

//unsigned Stirling numbers of the 1st kind (mod m) in O(n * k)
long long stirling1(int n, int k, long long m = 1000000007) {
  table t(n + 1, std::vector<long long>(k + 1, 0));
  t[0][0] = 1;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= k; j++) {
      t[i][j] = ((i - 1) * t[i - 1][j]) % m;
      t[i][j] = (t[i][j] + t[i - 1][j - 1]) % m;
    }
  return t[n][k] % m;
}

//Stirling numbers of the 2nd kind (mod m) in O(n * k)
long long stirling2(int n, int k, long long m = 1000000007) {
  table t(n + 1, std::vector<long long>(k + 1, 0));
  t[0][0] = 1;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= k; j++) {
      t[i][j] = (j * t[i - 1][j]) % m;
      t[i][j] = (t[i][j] + t[i - 1][j - 1]) % m;
    }
  return t[n][k] % m;
}

//Eulerian numbers of the 1st kind (mod m) in O(n * k)
//precondition: n > k
long long eulerian1(int n, int k, long long m = 1000000007) {
  if (k > n - 1 - k) k = n - 1 - k;
  table t(n + 1, std::vector<long long>(k + 1, 1));
  for (int j = 1; j <= k; j++) t[0][j] = 0;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= k; j++) {
      t[i][j] = ((i - j) * t[i - 1][j - 1]) % m;
      t[i][j] = (t[i][j] + ((j + 1) * t[i - 1][j]) % m) % m;
    }
  return t[n][k] % m;
}

//Eulerian numbers of the 2nd kind (mod m) in O(n * k)
//precondition: n > k
long long eulerian2(int n, int k, long long m = 1000000007) {
  table t(n + 1, std::vector<long long>(k + 1, 1));
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= k; j++) {
      if (i == j) {
        t[i][j] = 0;
      } else {
        t[i][j] = ((j + 1) * t[i - 1][j]) % m;
        t[i][j] = (((2 * i - 1 - j) * t[i - 1][j - 1]) % m
                  + t[i][j]) % m;
      }
    }
  return t[n][k] % m;
}

//nth Catalan number (mod a prime number p) in O(n)
long long catalan(int n, long long p = 1000000007) {
  return choose(2 * n, n, p) * powmod(n + 1, p - 2, p) % p;
}

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;
```

```

int main() {
  table t = binomial_table(10);
  for (int i = 0; i < (int)t.size(); i++) {
    for (int j = 0; j < (int)t[i].size(); j++)
      cout << t[i][j] << "␣";
    cout << "\n";
  }
  assert(factorial(10)      == 3628800);
  assert(factorialp(123456) == 639390503);
  assert(choose(20, 7)      == 77520);
  assert(multichoose(20, 7) == 657800);
  assert(permute(10, 4)     == 5040);
  assert(partitions(4)      == 5);
  assert(partitions(100, 5) == 38225);
  assert(stirling1(4, 2)    == 11);
  assert(stirling2(4, 3)    == 6);
  assert(eulerian1(9, 5)    == 88234);
  assert(eulerian2(8, 3)    == 195800);
  assert(catalan(10)        == 16796);
  return 0;
}
```

4.2.2 Enumerating Arrangements

```
/*

We shall consider an arrangement to be a permutation of
all the integers from 0 to n - 1. For our purposes, the
difference between an arrangement and a permutation is
simply that a permutation can pertain to a set of any
given values, not just distinct integers from 0 to n-1.

*/

#include <algorithm> /* std::copy(), std::fill() */
#include <vector>

/*

Changes a[] to the next lexicographically greater
permutation of any k distinct integers in range [0, n).
The values of a[] that's passed should be k distinct
integers, each in range [0, n).

returns: whether the function could rearrange a[] to
a lexicographically greater arrangement.

examples:
next_arrangement(4, 3, {0, 1, 2}) => 1,  a[] = {0, 1, 3}
next_arrangement(4, 3, {0, 1, 3}) => 1,  a[] = {0, 2, 1}
next_arrangement(4, 3, {3, 2, 1}) => 0,  a[] unchanged

*/

bool next_arrangement(int n, int k, int a[]) {
  std::vector<bool> used(n);
  for (int i = 0; i < k; i++) used[a[i]] = true;
  for (int i = k - 1; i >= 0; i--) {
    used[a[i]] = false;
    for (int j = a[i] + 1; j < n; j++) {
      if (!used[j]) {
        a[i++] = j;
        used[j] = true;
        for (int x = 0; i < k; x++)
          if (!used[x]) a[i++] = x;
        return true;
      }
    }
  }
  return false;
}

/*

Computes n permute k using formula: nPk = n!/(n - k)!
Complexity: O(k).  E.g. n_permute_k(10, 7) = 604800

*/

long long n_permute_k(int n, int k) {
  long long res = 1;
  for (int i = 0; i < k; i++) res *= n - i;
  return res;
}

/*

Given an integer rank x in range [0, n permute k), returns
a vector of integers representing the x-th lexicographically
smallest permutation of any k distinct integers in [0, n).

examples: arrangement_by_rank(4, 3, 0) => {0, 1, 2}
          arrangement_by_rank(4, 3, 5) => {0, 3, 2}

*/

std::vector<int> arrangement_by_rank(int n, int k, long long x) {
  std::vector<int> free(n), res(k);
  for (int i = 0; i < n; i++) free[i] = i;
  for (int i = 0; i < k; i++) {
    long long cnt = n_permute_k(n - 1 - i, k - 1 - i);
    int pos = (int)(x / cnt);
    res[i] = free[pos];
    std::copy(free.begin() + pos + 1, free.end(),
              free.begin() + pos);
    x %= cnt;
  }
  return res;
}

/*

Given an array a[] of k integers each in range [0, n), returns
the (0-based) lexicographical rank (counting from least to
greatest) of the arrangement specified by a[] in all possible
```

```
permutations of k distinct integers in range [0, n).

examples: rank_by_arrangement(4, 3, {0, 1, 2}) => 0
          rank_by_arrangement(4, 3, {0, 3, 2}) => 5

*/

long long rank_by_arrangement(int n, int k, int a[]) {
  long long res = 0;
  std::vector<bool> used(n);
  for (int i = 0; i < k; i++) {
    int cnt = 0;
    for (int j = 0; j < a[i]; j++)
      if (!used[j]) cnt++;
    res += n_permute_k(n - i - 1, k - i - 1) * cnt;
    used[a[i]] = true;
  }
  return res;
}

/*

Changes a[] to the next lexicographically greater
permutation of k (not-necessarily distinct) integers in
range [0, n). The values of a[] should be in range [0, n).
If a[] was interpreted as a base-n integer that is k digits
long, this function would be equivalent to incrementing a.
Ergo, there are n^k arrangements if repeats are allowed.

returns: whether the function could rearrange a[] to a
lexicographically greater arrangement with repeats.

examples:
n_a_w_r(4, 3, {0, 0, 0}) => 1,  a[] = {0, 0, 1}
n_a_w_r(4, 3, {0, 1, 3}) => 1,  a[] = {0, 2, 0}
n_a_w_r(4, 3, {3, 3, 3}) => 0,  a[] unchanged

*/

bool next_arrangement_with_repeats(int n, int k, int a[]) {
  for (int i = k - 1; i >= 0; i--) {
    if (a[i] < n - 1) {
      a[i]++;
      std::fill(a + i + 1, a + k, 0);
      return true;
    }
  }
  return false;
}

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;

template<class it> void print(it lo, it hi) {
  for (; lo != hi; ++lo) cout << *lo << "␣";
  cout << "\n";
}

int main() {
  {
    int n = 4, k = 3, a[] = {0, 1, 2};
    cout << n << "␣permute␣" << k << "␣arrangements:\n";
    int cnt = 0;
    do {
      print(a, a + k);
      vector<int> b = arrangement_by_rank(n, k, cnt);
      assert(equal(a, a + k, b.begin()));
      assert(rank_by_arrangement(n, k, a) == cnt);
      cnt++;
    } while (next_arrangement(n, k, a));
    cout << "\n";
  }

  {
    int n = 4, k = 2, a[] = {0, 0};
    cout << n << "^" << k << "␣arrangements␣with␣repeats:\n";
    do {
      print(a, a + k);
    } while (next_arrangement_with_repeats(n, k, a));
  }
  return 0;
}
```

4.2.3 Enumerating Permutations

```
/*

We shall consider a permutation of n objects to be an
ordered list of size n that contains all n elements,
where order is important. E.g. 1 1 2 0 and 0 1 2 1
are considered two different permutations of 0 1 1 2.
Compared to our prior definition of an arrangement, a
permutable range of size n may contain repeated values
of any type, not just the integers from 0 to n - 1.

*/

#include <algorithm> /* copy, iter_swap, reverse, swap */
#include <vector>

//identical to std::next_permutation()
template<class It> bool _next_permutation(It lo, It hi) {
  if (lo == hi) return false;
  It i = lo;
  if (++i == hi) return false;
  i = hi; --i;
  for (;;) {
    It j = i; --i;
    if (*i < *j) {
      It k = hi;
      while (!(*i < *--k)) /* pass */;
      std::iter_swap(i, k);
      std::reverse(j, hi);
      return true;
    }
    if (i == lo) {
      std::reverse(lo, hi);
      return false;
    }
  }
}

//array version
template<class T> bool next_permutation(int n, T a[]) {
  for (int i = n - 2; i >= 0; i--)
    if (a[i] < a[i + 1])
      for (int j = n - 1; ; j--)
        if (a[i] < a[j]) {
          std::swap(a[i++], a[j]);
          for (j = n - 1; i < j; i++, j--)
            std::swap(a[i], a[j]);
          return true;
        }
  return false;
}

/*

Calls the custom function f(vector) on all permutations
of the integers from 0 to n - 1. This is more efficient
than making many consecutive calls to next_permutation(),
however, here, the permutations will not be printed in
lexicographically increasing order.

*/

template<class ReportFunction>
void gen_permutations(int n, ReportFunction report,
                      std::vector<int> & p, int d) {
  if (d == n) {
    report(p);
    return;
  }
  for (int i = 0; i < n; i++) {
    if (p[i] == 0) {
      p[i] = d;
      gen_permutations(n, report, p, d + 1);
      p[i] = 0;
    }
  }
}

template<class ReportFunction>
void gen_permutations(int n, ReportFunction report) {
  std::vector<int> perms(n, 0);
  gen_permutations(n, report, perms, 0);
}

/*

Finds the next lexicographically greater permutation of
the binary digits of x. In other words, next_permutation()
```

```
simply returns the smallest integer greater than x which
has the same number of 1 bits (i.e. same popcount) as x.

examples: next_permutation(10101 base 2) =  10110
          next_permutation(11100 base 2) = 100011

This can also be used to generate combinations as follows:
If we let k = popcount(x), then we can use this to generate
all possible masks to tell us which k items to take out of
n total items (represented by the first n bits of x).

*/

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

/*

Given an integer rank x in range [0, n!), returns a vector
of integers representing the x-th lexicographically smallest
permutation of the integers in [0, n).

examples: permutation_by_rank(4, 0) => {0, 1, 2, 3}
          permutation_by_rank(4, 5) => {0, 3, 2, 1}

*/

std::vector<int> permutation_by_rank(int n, long long x) {
  long long fact[n];
  fact[0] = 1;
  for (int i = 1; i < n; i++)
    fact[i] = i * fact[i - 1];
  std::vector<int> free(n), res(n);
  for (int i = 0; i < n; i++) free[i] = i;
  for (int i = 0; i < n; i++) {
    int pos = x / fact[n - 1 - i];
    res[i] = free[pos];
    std::copy(free.begin() + pos + 1, free.end(),
              free.begin() + pos);
    x %= fact[n - 1 - i];
  }
  return res;
}

/*

Given an array a[] of n integers each in range [0, n), returns
the (0-based) lexicographical rank (counting from least to
greatest) of the arrangement specified by a[] in all possible
permutations of the integers from 0 to n - 1.

examples: rank_by_permutation(3, {0, 1, 2}) => 0
          rank_by_permutation(3, {2, 1, 0}) => 5

*/

template<class T> long long rank_by_permutation(int n, T a[]) {
  long long fact[n];
  fact[0] = 1;
  for (int i = 1; i < n; i++)
    fact[i] = i * fact[i - 1];
  long long res = 0;
  for (int i = 0; i < n; i++) {
    int v = a[i];
    for (int j = 0; j < i; j++)
      if (a[j] < a[i]) v--;
    res += v * fact[n - 1 - i];
  }
  return res;
}

/*

Given a permutation a[] of the integers from 0 to n - 1,
returns a decomposition of the permutation into cycles.
A permutation cycle is a subset of a permutation whose
elements trade places with one another. For example, the
permutation {0, 2, 1, 3} decomposes to {0, 3, 2} and {1}.
Here, the notation {0, 3, 2} means that starting from the
original ordering {0, 1, 2, 3}, the 0th value is replaced
by the 3rd, the 3rd by the 2nd, and the 2nd by the first,
See: http://mathworld.wolfram.com/PermutationCycle.html

*/

typedef std::vector<std::vector<int> > cycles;

cycles decompose_into_cycles(int n, int a[]) {
  std::vector<bool> vis(n);
  cycles res;
  for (int i = 0; i < n; i++) {
    if (vis[i]) continue;
    int j = i;
    std::vector<int> cur;
    do {
      cur.push_back(j);
      vis[j] = true;
      j = a[j];
    } while (j != i);
    res.push_back(cur);
  }
  return res;
}

/*** Example Usage ***/

#include <bitset>
#include <cassert>
#include <iostream>
using namespace std;

void printperm(const vector<int> & perm) {
  for (int i = 0; i < (int)perm.size(); i++)
    cout << perm[i] << "␣";
  cout << "\n";
}
```

```cpp

template<class it> void print(it lo, it hi) {
  for (; lo != hi; ++lo) cout << *lo << "␣";
  cout << "\n";
}

int main() {
  { //method 1: ordered
    int n = 4, a[] = {0, 1, 2, 3};
    int b[n], c[n];
    for (int i = 0; i < n; i++) b[i] = c[i] = a[i];
    cout << "Ordered␣permutations␣of␣0␣to␣" << n-1 << ":\n";
    int cnt = 0;
    do {
      print(a, a + n);
      assert(equal(b, b + n, a));
      assert(equal(c, c + n, a));
      vector<int> d = permutation_by_rank(n, cnt);
      assert(equal(d.begin(), d.end(), a));
      assert(rank_by_permutation(n, a) == cnt);
      cnt++;
      std::next_permutation(b, b + n);
      _next_permutation(c, c + n);
    } while (next_permutation(n, a));
    cout << "\n";
  }

  { //method 2: unordered
    int n = 3;
    cout << "Unordered␣permutations␣of␣0␣to␣" << n-1 << ":\n";
    gen_permutations(n, printperm);
    cout << "\n";
  }

  { //permuting binary digits
    const int n = 5;
    cout << "Permutations␣of␣2␣zeros␣and␣3␣ones:\n";
    long long lo = 7;  // 00111 in base 2
    long long hi = 35; //100011 in base 2
    do {
      cout << bitset<n>(lo).to_string() << "\n";
    } while ((lo = next_permutation(lo)) != hi);
    cout << "\n";
  }

  { //permutation cycles
    int n = 4, a[] = {3, 1, 0, 2};
    cout << "Decompose␣0␣2␣1␣3␣into␣cycles:\n";
    cycles c = decompose_into_cycles(n, a);
    for (int i = 0; i < (int)c.size(); i++) {
      cout << "Cycle␣" << i + 1 << ":";
      for (int j = 0; j < (int)c[i].size(); j++)
        cout << "␣" << c[i][j];
      cout << "\n";
    }
  }
  return 0;
}
```

4.2.4 Enumerating Combinations

```
/*

We shall consider a combination n choose k to be an
set of k elements chosen from a total of n elements.
Unlike n permute k, the order here doesn’t matter.
That is, 0 1 2 is considered the same as 0 2 1, so
we will consider the sorted representation of each
combination for purposes of the functions below.

*/

#include <algorithm> /* iter_swap, rotate, swap, swap_ranges */
#include <iterator>  /* std::iterator_traits */
#include <vector>

/*

Rearranges the values in the range [lo, hi) such that
elements in the range [lo, mid) becomes the next
lexicographically greater combination of the values from
[lo, hi) than it currently is, and returns whether the
function could rearrange [lo, hi) to a lexicographically
greater combination. If the range [lo, hi) contains n
elements and the range [lo, mid) contains k elements,
then starting off with a sorted range [lo, hi) and
calling next_combination() repeatedly will return true
for n choose k iterations before returning false.

*/

template<class It>
bool next_combination(It lo, It mid, It hi) {
  if (lo == mid || mid == hi) return false;
  It l(mid - 1), h(hi - 1);
  int sz1 = 1, sz2 = 1;
  while (l != lo && !(*l < *h)) --l, ++sz1;
  if (l == lo && !(*l < *h)) {
    std::rotate(lo, mid, hi);
    return false;
  }
  for (; mid < h; ++sz2) if (!(*l < *--h)) { ++h; break; }
  if (sz1 == 1 || sz2 == 1) {
    std::iter_swap(l, h);
  } else if (sz1 == sz2) {
    std::swap_ranges(l, mid, h);
  } else {
    std::iter_swap(l, h);
    ++l; ++h; --sz1; --sz2;
    int total = sz1 + sz2, gcd = total;
    for (int i = sz1; i != 0; ) std::swap(gcd %= i, i);
    int skip = total / gcd - 1;
    for (int i = 0; i < gcd; i++) {
      It curr(i < sz1 ? l + i : h + (i - sz1));
      int k = i;
      typename std::iterator_traits<It>::value_type v(*curr);
      for (int j = 0; j < skip; j++) {
        k = (k + sz1) % total;
        It next(k < sz1 ? l + k : h + (k - sz1));
        *curr = *next;
        curr = next;
      }
      *curr = v;
    }
  }
  return true;
}

/*

Changes a[] to the next lexicographically greater
combination of any k distinct integers in range [0, n).
The values of a[] that’s passed should be k distinct
integers, each in range [0, n).

*/

bool next_combination(int n, int k, int a[]) {
  for (int i = k - 1; i >= 0; i--) {
    if (a[i] < n - k + i) {
      for (++a[i]; ++i < k; ) a[i] = a[i - 1] + 1;
      return true;
    }
  }
  return false;
}

/*

Finds the "mask" of the next combination of x. This is
equivalent to the next lexicographically greater permutation
of the binary digits of x. In other words, the function
simply returns the smallest integer greater than x which
has the same number of 1 bits (i.e. same popcount) as x.

examples: next_combination_mask(10101 base 2) =  10110
          next_combination_mask(11100 base 2) = 100011

If we arbitrarily number the n items of our collection from
0 to n-1, then generating all combinations n choose k can
be done as follows: initialize x such that popcount(x) = k
and the first (least-significant) k bits are all set to 1
(e.g. to do 5 choose 3, start at x = 00111 (base 2) = 7).
Then, we repeatedly call x = next_combination_mask(x) until
we reach 11100 (the lexicographically greatest mask for 5
choose 3), after which we stop. At any point in the process,
we can say that the i-th item is being "taken" (0 <= i < n)
iff the i-th bit of x is set.

Note: this does not produce combinations in the same order
as next_combination, nor does it work if your n items have
repeated values (in that case, repeated combos will be
generated).

*/

```

```
long long next_combination_mask(long long x) {
  long long s = x & -x, r = x + s;
  return r | (((x ^ r) >> 2) / s);
}

//n choose k in O(min(k, n - k))
long long n_choose_k(long long n, long long k) {
  if (k > n - k) k = n - k;
  long long res = 1;
  for (int i = 0; i < k; i++)
    res = res * (n - i) / (i + 1);
  return res;
}

/*

Given an integer rank x in range [0, n choose k), returns
a vector of integers representing the x-th lexicographically
smallest combination k distinct integers in [0, n).

examples: combination_by_rank(4, 3, 0) => {0, 1, 2}
          combination_by_rank(4, 3, 2) => {0, 2, 3}

*/

std::vector<int> combination_by_rank(int n, int k, long long x) {
  std::vector<int> res(k);
  int cnt = n;
  for (int i = 0; i < k; i++) {
    int j = 1;
    for (;; j++) {
      long long am = n_choose_k(cnt - j, k - 1 - i);
      if (x < am) break;
      x -= am;
    }
    res[i] = i > 0 ? (res[i - 1] + j) : (j - 1);
    cnt -= j;
  }
  return res;
}

/*

Given an array a[] of k integers each in range [0, n), returns
the (0-based) lexicographical rank (counting from least to
greatest) of the combination specified by a[] in all possible
combination of k distinct integers in range [0, n).

examples: rank_by_combination(4, 3, {0, 1, 2}) => 0
          rank_by_combination(4, 3, {0, 2, 3}) => 2

*/

long long rank_by_combination(int n, int k, int a[]) {
  long long res = 0;
  int prev = -1;
  for (int i = 0; i < k; i++) {
    for (int j = prev + 1; j < a[i]; j++)
      res += n_choose_k(n - 1 - j, k - 1 - i);
    prev = a[i];
  }
  return res;
}

/*

Changes a[] to the next lexicographically greater
combination of any k (not necessarily distinct) integers
in range [0, n). The values of a[] that's passed should
be k integers, each in range [0, n). Note that there are
a total of n multichoose k combinations with repetition,
where n multichoose k = (n + k - 1) choose k

*/

bool next_combination_with_repeats(int n, int k, int a[]) {
  for (int i = k - 1; i >= 0; i--) {
    if (a[i] < n - 1) {
      for (++a[i]; ++i < k; ) a[i] = a[i - 1];
      return true;
    }
  }
  return false;
}

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;

template<class it> void print(it lo, it hi) {
  for (; lo != hi; ++lo) cout << *lo << "␣";
  cout << "\n";
}

int main() {
  { //like std::next_permutation(), repeats in the range allowed
    int k = 3;
    string s = "11234";
    cout << s << "␣choose␣" << k << ":\n";
    do {
      cout << s.substr(0, k) << "\n";
    } while (next_combination(s.begin(), s.begin() + k, s.end()));
    cout << "\n";
  }

  { //unordered combinations with masks
    int n = 5, k = 3;
    string s = "abcde"; //must be distinct values
    cout << s << "␣choose␣" << k << "␣with␣masks:\n";
    long long mask = 0, dest = 0;
    for (int i = 0; i < k; i++) mask |= 1 << i;
    for (int i = k - 1; i < n; i++) dest |= 1 << i;
    do {
      for (int i = 0; i < n; i++)
        if ((mask >> i) & 1) cout << s[i];
      cout << "\n";
      mask = next_combination_mask(mask);
    } while (mask != dest);
    cout << "\n";
  }

  { //only combinations of distinct integers from 0 to n - 1
    int n = 5, k = 3, a[] = {0, 1, 2};
    cout << n << "␣choose␣" << k << ":\n";
    int cnt = 0;
    do {
      print(a, a + k);
      vector<int> b = combination_by_rank(n, k, cnt);
      assert(equal(a, a + k, b.begin()));
      assert(rank_by_combination(n, k, a) == cnt);
      cnt++;
    } while (next_combination(n, k, a));
    cout << "\n";
  }

  { //combinations with repetition
    int n = 3, k = 2, a[] = {0, 0};
    cout << n << "␣multichoose␣" << k << ":\n";
    do {
      print(a, a + k);
    } while (next_combination_with_repeats(n, k, a));
  }
  return 0;
}
```

4.2.5 Enumerating Partitions

```
/*

We shall consider a partition of an integer n to be an
unordered multiset of positive integers that has a total
sum equal to n. Since both 2 1 1 and 1 2 1 represent the
same partition of 4, we shall consider only descending
sorted lists as "valid" partitions for functions below.

*/

#include <vector>

/*

Given a vector representing a partition of some
integer n (the sum of all values in the vector),
changes p to the next lexicographically greater
partition of n and returns whether the change was
successful (whether a lexicographically greater
partition existed). Note that the "initial" value
of p must be a vector of size n, all initialized 1.

e.g. next_partition({2, 1, 1}) => 1, p becomes {2, 2}
     next_partition({2, 2})    => 1, p becomes {3, 1}
     next_partition({4})       => 0, p is unchanged

```

```
*/

bool next_partition(std::vector<int> & p) {
  int n = p.size();
  if (n <= 1) return false;
  int s = p[n - 1] - 1, i = n - 2;
  p.pop_back();
  for (; i > 0 && p[i] == p[i - 1]; i--) {
    s += p[i];
    p.pop_back();
  }
  for (p[i]++; s-- > 0; ) p.push_back(1);
  return true;
}

/* Returns the number of partitions of n. */

long long count_partitions(int n) {
  std::vector<long long> p(n + 1, 0);
  p[0] = 1;
  for (int i = 1; i <= n; i++)
    for (int j = i; j <= n; j++)
      p[j] += p[j - i];
  return p[n];
}

/* Helper function for partitioning by rank */

std::vector< std::vector<long long> >
  p(1, std::vector<long long>(1, 1)); //memoization

long long partition_function(int a, int b) {
  if (a >= (int)p.size()) {
    int old = p.size();
    p.resize(a + 1);
    p[0].resize(a + 1);
    for (int i = 1; i <= a; i++) {
      p[i].resize(a + 1);
      for (int j = old; j <= i; j++)
        p[i][j] = p[i - 1][j - 1] + p[i - j][j];
    }
  }
  return p[a][b];
}

/*

Given an integer n to partition and a 0-based rank x,
returns a vector of integers representing the x-th
lexicographically smallest partition of n (if values
in each partition were sorted in decreasing order).

examples: partition_by_rank(4, 0) => {1, 1, 1, 1}
          partition_by_rank(4, 3) => {3, 1}

*/

std::vector<int> partition_by_rank(int n, long long x) {
  std::vector<int> res;
  for (int i = n; i > 0; ) {
    int j = 1;
    for (;; j++) {
      long long cnt = partition_function(i, j);
      if (x < cnt) break;
      x -= cnt;
    }
    res.push_back(j);
    i -= j;
  }
  return res;
}

/*

Given a partition of an integer n (sum of all values
in vector p), returns a 0-based rank x of the partition
represented by p, considering partitions from least to
greatest in lexicographical order (if each partition
had values sorted in descending order).

examples: rank_by_partition({1, 1, 1, 1}) => 0
          rank_by_partition({3, 1})       => 3

*/

long long rank_by_partition(const std::vector<int> & p) {
  long long res = 0;
  int sum = 0;
  for (int i = 0; i < (int)p.size(); i++) sum += p[i];
  for (int i = 0; i < (int)p.size(); i++) {
    for (int j = 0; j < p[i]; j++)
      res += partition_function(sum, j);
    sum -= p[i];
  }
  return res;
}

/*

Calls the custom function f(vector) on all partitions
which consist of strictly *increasing* integers.
This will exclude partitions such as {1, 1, 1, 1}.

*/

template<class ReportFunction>
void gen_increasing_partitons(int left, int prev, int i,
                 ReportFunction f, std::vector<int> & p) {
  if (left == 0) {
    //warning: slow constructor - modify accordingly
    f(std::vector<int>(p.begin(), p.begin() + i));
    return;
  }
  for (p[i] = prev + 1; p[i] <= left; p[i]++)
    gen_increasing_partitons(left - p[i], p[i], i + 1, f, p);
}

template<class ReportFunction>
```

```
void gen_increasing_partitons(int n, ReportFunction f) {
  std::vector<int> partitions(n, 0);
  gen_increasing_partitons(n, 0, 0, f, partitions);
}

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;

void print(const vector<int> & v) {
  for (int i = 0; i < (int)v.size(); i++)
    cout << v[i] << "␣";
  cout << "\n";
}

int main() {
  assert(count_partitions(5) == 7);
  assert(count_partitions(20) == 627);
  assert(count_partitions(30) == 5604);
  assert(count_partitions(50) == 204226);
  assert(count_partitions(100) == 190569292);

  {
    int n = 4;
    vector<int> a(n, 1);
    cout << "Partitions␣of␣" << n << ":\n";
    int cnt = 0;
    do {
      print(a);
      vector<int> b = partition_by_rank(n, cnt);
      assert(equal(a.begin(), a.end(), b.begin()));
      assert(rank_by_partition(a) == cnt);
      cnt++;
    } while (next_partition(a));
    cout << "\n";
  }

  {
    int n = 8;
    cout << "Increasing␣partitions␣of␣" << n << ":\n";
    gen_increasing_partitons(n, print);
  }
  return 0;
}
```

4.2.6 Enumerating Generic Combinatorial Sequences

```
/*

The follow provides a universal method for enumerating
abstract combinatorial sequences in O(n^2) time.

*/

#include <vector>
```

```

class abstract_enumeration {
 protected:
  int range, length;

  abstract_enumeration(int r, int l): range(r), length(l) {}

  virtual long long count(const std::vector<int> & pre) {
    return 0;
  }

  std::vector<int> next(std::vector<int> & seq) {
    return from_number(to_number(seq) + 1);
  }

  long long total_count() {
    return count(std::vector<int>(0));
  }

public:
  long long to_number(const std::vector<int> & seq) {
    long long res = 0;
    for (int i = 0; i < (int)seq.size(); i++) {
      std::vector<int> pre(seq.begin(), seq.end());
      pre.resize(i + 1);
      for (pre[i] = 0; pre[i] < seq[i]; ++pre[i])
        res += count(pre);
    }
    return res;
  }

  std::vector<int> from_number(long long x) {
    std::vector<int> seq(length);
    for (int i = 0; i < (int)seq.size(); i++) {
      std::vector<int> pre(seq.begin(), seq.end());
      pre.resize(i + 1);
      for (pre[i] = 0; pre[i] < range; ++pre[i]) {
        long long cur = count(pre);
        if (x < cur) break;
        x -= cur;
      }
      seq[i] = pre[i];
    }
    return seq;
  }

  template<class ReportFunction>
  void enumerate(ReportFunction report) {
    long long total = total_count();
    for (long long i = 0; i < total; i++) {
      //assert(i == to_number(from_number(i));
      report(from_number(i));
    }
  }
};

class arrangements: public abstract_enumeration {
 public:
  arrangements(int n, int k) : abstract_enumeration(n, k) {}

  long long count(const std::vector<int> & pre) {
    int sz = pre.size();
    for (int i = 0; i < sz - 1; i++)
      if (pre[i] == pre[sz - 1]) return 0;
    long long res = 1;
    for (int i = 0; i < length - sz; i++)
      res *= range - sz - i;
    return res;
  }
};

class permutations: public arrangements {
 public:
  permutations(int n) : arrangements(n, n) {}
};

class combinations: public abstract_enumeration {
  std::vector<std::vector<long long> > binomial;

 public:
  combinations(int n, int k) : abstract_enumeration(n, k),
   binomial(n + 1, std::vector<long long>(n + 1, 0)) {
    for (int i = 0; i <= n; i++)
      for (int j = 0; j <= i; j++)
        binomial[i][j] = (j == 0) ? 1 :
              binomial[i - 1][j - 1] + binomial[i - 1][j];
  }

  long long count(const std::vector<int> & pre) {
    int sz = pre.size();
    if (sz >= 2 && pre[sz - 1] <= pre[sz - 2]) return 0;
    int last = sz > 0 ? pre[sz - 1] : -1;
    return binomial[range - 1 - last][length - sz];
  }
};

class partitions: public abstract_enumeration {
  std::vector<std::vector<long long> > p;

 public:
  partitions(int n) : abstract_enumeration(n + 1, n),
   p(n + 1, std::vector<long long>(n + 1, 0)) {
    std::vector<std::vector<long long> > pp(p);
    pp[0][0] = 1;
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= i; j++)
        pp[i][j] = pp[i - 1][j - 1] + pp[i - j][j];
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++)
        p[i][j] = pp[i][j] + p[i][j - 1];
  }

  long long count(const std::vector<int> & pre) {
    int size = pre.size(), sum = 0;
    for (int i = 0; i < (int)pre.size(); i++) sum += pre[i];
    if (sum == range - 1) return 1;
    if (sum > range - 1 || (size > 0 && pre[size - 1] == 0) ||
        (size >= 2 && pre[size - 1] > pre[size - 2])) return 0;
    int last = size > 0 ? pre[size - 1] : range - 1;
    return p[range - 1 - sum][last];
  }
};

/*** Example Usage ***/

#include <iostream>
using namespace std;

void print(const std::vector<int> & v) {
  for (int i = 0; i < (int)v.size(); i++)
    cout << v[i] << "␣";
  cout << "\n";
}

int main() {
  cout << "Arrangement(3,␣2):\n";
  arrangements arrg(3, 2);
  arrg.enumerate(print);

  cout << "Permutation(3):\n";
  permutations perm(3);
  perm.enumerate(print);

  cout << "Combination(4,␣3):\n";
  combinations comb(4, 3);
  comb.enumerate(print);

  cout << "Partition(4):\n";
  partitions part(4);
  part.enumerate(print);
  return 0;
}
```

4.3 Number Theory

4.3.1 GCD, LCM, Mod Inverse, Chinese Remainder

```
/*

GCD, LCM, Modular Inverse, Chinese Remainder Theorem

*/

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

//C++98 does not have abs() declared for long long
template<class T> inline T _abs(const T & x) {
  return x < 0 ? -x : x;
}

//GCD using Euclid's algorithm - O(log(a + b))
template<class Int> Int gcd(Int a, Int b) {
  return b == 0 ? _abs(a) : gcd(b, a % b);
}

//non-recursive version
template<class Int> Int gcd2(Int a, Int b) {
  while (b != 0) {
    Int t = b;
    b = a % b;
    a = t;
  }
  return _abs(a);
}

template<class Int> Int lcm(Int a, Int b) {
  return _abs(a / gcd(a, b) * b);
}

//returns <gcd(a, b), <x, y>> such that gcd(a, b) = ax + by
template<class Int>
std::pair<Int, std::pair<Int, Int> > euclid(Int a, Int b) {
  Int x = 1, y = 0, x1 = 0, y1 = 1;
  //invariant: a = a * x + b * y, b = a * x1 + b * y1
  while (b != 0) {
    Int q = a / b, _x1 = x1, _y1 = y1, _b = b;
    x1 = x - q * x1;
    y1 = y - q * y1;
    b = a - q * b;
    x = _x1;
    y = _y1;
    a = _b;
  }
  return a > 0 ? std::make_pair(a, std::make_pair(x, y)) :
                 std::make_pair(-a, std::make_pair(-x, -y));
}

//recursive version
template<class Int>
std::pair<Int, std::pair<Int, Int> > euclid2(Int a, Int b) {
  if (b == 0) {
    return a > 0 ? std::make_pair(a, std::make_pair(1, 0)) :
                   std::make_pair(-a, std::make_pair(-1, 0));
  }
  std::pair<Int, std::pair<Int, Int> > r = euclid2(b, a % b);
  return std::make_pair(r.first, std::make_pair(r.second.second,
                    r.second.first - a / b * r.second.second));
}

/*

Modulo Operation - Euclidean Definition

The % operator in C/C++ returns the remainder of division (which
may be positive or negative) The true Euclidean definition of
modulo, however, defines the remainder to be always nonnegative.
For positive operators, % and mod are the same. But for negative
operands, they differ. The result here is consistent with the
Euclidean division algorithm.

e.g. -21 % 4 == -1 since -21 / 4 == -5 and 4 * -5 + (-1) == -21
      however, -21 mod 4 is equal to 3 because -21 + 4 * 6 is 3.
```

```

*/

template<class Int> Int mod(Int a, Int m) {
  Int r = (Int)(a % m);
  return r >= 0 ? r : r + m;
}

//returns x such that a * x = 1 (mod m)
//precondition: m > 0 && gcd(a, m) = 1
template<class Int> Int mod_inverse(Int a, Int m) {
  a = mod(a, m);
  return a == 0 ? 0 : mod((1 - m * mod_inverse(m % a, a)) / a, m);
}

//precondition: m > 0 && gcd(a, m) = 1
template<class Int> Int mod_inverse2(Int a, Int m) {
  return mod(euclid(a, m).second.first, m);
}

//returns a vector where i*v[i] = 1 (mod p) in O(p) time
//precondition: p is prime
std::vector<int> generate_inverses(int p) {
  std::vector<int> res(p);
  res[1] = 1;
  for (int i = 2; i < p; i++)
    res[i] = (p - (p / i) * res[p % i] % p) % p;
  return res;
}

/*

Chinese Remainder Theorem

Let r and s be positive integers which are relatively prime and
let a and b be any two integers. Then there exists an integer N
such that N = a (mod r) and N = b (mod s). Moreover, N is
uniquely determined modulo rs.

More generally, given a set of simultaneous congruences for
which all values in p[] are pairwise relative prime:

 x = a[i] (mod p[i]), for i = 1..n

the solution of the set of congruences is:

 x = a[1] * b[1] * (M/p[1]) + ... + a[n] * b[n] * (M/p[n]) (mod M)

where M = p[1] * p[2] ... * p[n] and the b[i] are determined for

 b[i] * (M/p[i]) = 1 (mod p[i]).

The following functions solves for this value of x, with the
first function computed using the method above while the
second function using a special case of Garner's algorithm.

http://e-maxx-eng.github.io/algebra/chinese-remainder-theorem.html

*/
```

```cpp

long long simple_restore(int n, int a[], int p[]) {
  long long res = 0, m = 1;
  for (int i = 0; i < n; i++) {
    while (res % p[i] != a[i]) res += m;
    m *= p[i];
  }
  return res;
}

long long garner_restore(int n, int a[], int p[]) {
  int x[n];
  for (int i = 0; i < n; i++) x[i] = a[i];
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++)
      x[i] = mod_inverse((long long)p[j], (long long)p[i]) *
                         (long long)(x[i] - x[j]);
      x[i] = (x[i] % p[i] + p[i]) % p[i];
  }
  long long res = x[0], m = 1;
  for (int i = 1; i < n; i++) {
    m *= p[i - 1];
    res += x[i] * m;
  }
  return res;
}

/*** Example Usage ***/

#include <cassert>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

int main() {
  {
    srand(time(0));
    for (int steps = 0; steps < 10000; steps++) {
      int a = rand() % 200 - 10;
      int b = rand() % 200 - 10;
      int g1 = gcd(a, b), g2 = gcd2(a, b);
      assert(g1 == g2);
      if (g1 == 1 && b > 1) {
        int inv1 = mod_inverse(a, b);
        int inv2 = mod_inverse2(a, b);
        assert(inv1 == inv2 && mod(a * inv1, b) == 1);
      }
      pair<int, pair<int, int> > euc1 = euclid(a, b);
      pair<int, pair<int, int> > euc2 = euclid2(a, b);
      assert(euc1.first == g1 && euc1 == euc2);
      int x = euc1.second.first;
      int y = euc1.second.second;
      assert(g1 == a * x + b * y);
    }
  }

  {
    long long a = 6, b = 9;
    pair<int, pair<int, int> > r = euclid(6, 9);
    cout << r.second.first << "␣*␣(" << a << ")" << "␣+␣";
    cout << r.second.second << "␣*␣(" << b << ")␣=␣gcd(";
    cout << a << "," << b << ")␣=␣" << r.first << "\n";
  }

  {
    int prime = 17;
    std::vector<int> res = generate_inverses(prime);
    for (int i = 0; i < prime; i++) {
      if (i > 0) assert(mod(i * res[i], prime) == 1);
      cout << res[i] << "␣";
    }
    cout << "\n";
  }

  {
    int n = 3, a[] = {2, 3, 1}, m[] = {3, 4, 5};
    //solves for x in the simultaneous congruences:
    //x = 2 (mod 3)
    //x = 3 (mod 4)
    //x = 1 (mod 5)
    int x1 = simple_restore(n, a, m);
    int x2 = garner_restore(n, a, m);
    assert(x1 == x2);
    for (int i = 0; i < n; i++)
      assert(mod(x1, m[i]) == a[i]);
    cout << "Solution:␣" << x1 << "\n"; //11
  }

  return 0;
}
```

4.3.2 Generating Primes

```
/*

The following are three methods to generate primes.
Although the latter two functions are theoretically
linear, the former function with the sieve of
Eratosthenes is still significantly the fastest even
for n under 1 billion, since its constant factor is
so much better because of its minimal arithmetic
operations. For this reason, it should be favored
over the other two algorithms in most contest
applications. For the computation of larger primes,
you should replace int with long long or an arbitrary
precision class.

*/

#include <cmath> /* ceil(), sqrt() */
#include <vector>

//Sieve of Eratosthenes in ~ O(n log log n)
//returns: a vector of all primes under n
std::vector<int> gen_primes(int n) {
  std::vector<bool> prime(n + 1, true);
  int sqrtn = (int)ceil(sqrt(n));
  for (int i = 2; i <= sqrtn; i++) {
    if (prime[i])
      for (int j = i * i; j <= n; j += i)
        prime[j] = false;
  }
  std::vector<int> res;
  for (int i = 2; i <= n; i++)
    if (prime[i]) res.push_back(i);
  return res;
}

//Technically O(n), but on -O2, this is about
//as fast as the above sieve for n = 100 million
std::vector<int> gen_primes_linear(int n) {
  std::vector<int> lp(n + 1), res;
  for (int i = 2; i <= n; i++) {
    if (lp[i] == 0) {
      lp[i] = i;
      res.push_back(i);
    }
    for (int j = 0; j < (int)res.size(); j++) {
      if (res[j] > lp[i] || i * res[j] > n)
        break;
      lp[i * res[j]] = res[j];
    }
  }
  return res;
}

//Sieve of Atkins in O(n), somewhat slow due to
//its heavier arithmetic compared to the above
std::vector<int> gen_primes_atkins(int n) {
  std::vector<bool> prime(n + 1, false);
  std::vector<int> res;
  prime[2] = true;
  prime[3] = true;
  int num, lim = ceil(sqrt(n));
  for (int x = 1; x <= lim; x++) {
    for (int y = 1; y <= lim; y++) {
      num = 4 * x * x + y * y;
      if (num <= n && (num % 12 == 1 || num % 12 == 5))
        prime[num] = true;
      num = 3 * x * x + y * y;
      if (num <= n && (num % 12 == 7))
        prime[num] = true;
      if (x > y) {
        num = (3 * x * x - y * y);
        if (num <= n && num % 12 == 11)
          prime[num] = true;
      }
    }
  }
  for (int i = 5; i <= lim; i++) {
    if (prime[i])
      for (int j = i * i; j <= n; j += i)
        prime[j] = false;
  }
  for (int i = 2; i <= n; i++)
    if (prime[i]) res.push_back(i);
  return res;
}

//Double sieve to find primes in [l, h]
//Approximately O(sqrt(h) * log log(h - l))
std::vector<int> gen_primes(int l, int h) {
  int sqrth = (int)ceil(sqrt(h));
  int sqrtsqrth = (int)ceil(sqrt(sqrth));
  std::vector<bool> prime1(sqrth + 1, true);
  std::vector<bool> prime2(h - l + 1, true);
  for (int i = 2; i <= sqrtsqrth; i++) {
    if (prime1[i])
      for (int j = i * i; j <= sqrth; j += i)
        prime1[j] = false;
  }
  for (int i = 2, n = h - l; i <= sqrth; i++) {
    if (prime1[i])
      for (int j = l / i * i - l; j <= n; j += i)
        if (j >= 0 && j + l != i)
          prime2[j] = false;
  }
  std::vector<int> res;
  for (int i = l > 1 ? l : 2; i <= h; i++)
    if (prime2[i - l]) res.push_back(i);
  return res;
}

/*** Example Usage ***/

#include <cassert>
#include <ctime>
#include <iostream>
using namespace std;

template<class It> void print(It lo, It hi) {
  while (lo != hi) cout << *(lo++) << " ";
  cout << "\n";
}

int main() {
  int pmax = 10000000;
  vector<int> p;
  time_t start;
  double delta;

  cout << "Generating primes up to " << pmax << "...\n";
  start = clock();
  p = gen_primes(pmax);
  delta = (double)(clock() - start)/CLOCKS_PER_SEC;
  cout << "gen_primes() took " << delta << "s.\n";

  start = clock();
  p = gen_primes_linear(pmax);
  delta = (double)(clock() - start)/CLOCKS_PER_SEC;
  cout << "gen_primes_linear() took " << delta << "s.\n";

  start = clock();
  p = gen_primes_atkins(pmax);
  delta = (double)(clock() - start)/CLOCKS_PER_SEC;
  cout << "gen_primes_atkins() took " << delta << "s.\n";

  cout << "Generated " << p.size() << " primes.\n";
  //print(p.begin(), p.end());

  for (int i = 0; i <= 1000; i++) {
    assert(gen_primes(i) == gen_primes_linear(i));
    assert(gen_primes(i) == gen_primes_atkins(i));
  }

  int l = 1000000000, h = 1000000500;
  cout << "Generating primes in [" << l << ", " << h << "]...\n";
  start = clock();
  p = gen_primes(l, h);
  delta = (double)(clock() - start)/CLOCKS_PER_SEC;
  cout << "Generated " << p.size() << " primes in " << delta << "s.\n";
  print(p.begin(), p.end());
  return 0;
}
```

4.3.3 Primality Testing

```
/*

Primality Testing

*/

#include <cstdlib>  /* rand(), srand() */
#include <ctime>    /* time() */
#include <stdint.h> /* uint64_t */

/*

Trial division in O(sqrt(n)) to return whether n is prime
Applies an optimization based on the fact that all
primes greater than 3 take the form 6n + 1 or 6n - 1.

*/

template<class Int> bool is_prime(Int n) {
  if (n == 2 || n == 3) return true;
  if (n < 2 || !(n % 2) || !(n % 3)) return false;
  for (Int i = 5, w = 4; i * i <= n; i += (w = 6 - w))
    if (n % i == 0) return false;
  return true;
}

/*

Miller-Rabin Primality Test (Probabilistic)

Checks whether a number n is probably prime. If n is prime,
the function is guaranteed to return 1. If n is composite,
the function returns 1 with a probability of (1/4)^k,
where k is the number of iterations. With k = 1, the
probability of a composite being falsely predicted to be a
prime is 25%. If k = 5, the probability for this error is
just less than 0.1%. Thus, k = 18 to 20 is accurate enough
for most applications. All values of n < 2^63 is supported.

Complexity: O(k log^3(n)). In comparison to trial division,
the Miller-Rabin algorithm on 32-bit ints take ~45
operations for k = 10 iterations (~0.0001% error), while the
former takes ~10,000.

Warning: Due to the overflow of modular exponentiation,
         this will only work on inputs less than 2^63.

*/

uint64_t mulmod(uint64_t a, uint64_t b, uint64_t m) {
  uint64_t x = 0, y = a % m;
  for (; b > 0; b >>= 1) {
    if (b & 1) x = (x + y) % m;
    y = (y << 1) % m;
  }
  return x % m;
}

uint64_t powmod(uint64_t a, uint64_t b, uint64_t m) {
  uint64_t x = 1, y = a;
  for (; b > 0; b >>= 1) {
    if (b & 1) x = mulmod(x, y, m);
    y = mulmod(y, y, m);
  }
  return x % m;
}

//5 calls to rand() is unnecessary if RAND_MAX is 2^31-1
uint64_t rand64u() {
  return ((uint64_t)(rand() & 0xf) << 60) |
         ((uint64_t)(rand() & 0x7fff) << 45) |
         ((uint64_t)(rand() & 0x7fff) << 30) |
         ((uint64_t)(rand() & 0x7fff) << 15) |
         ((uint64_t)(rand() & 0x7fff));
}

bool is_probable_prime(long long n, int k = 20) {
  if (n < 2 || (n != 2 && !(n & 1))) return false;
  uint64_t s = n - 1, p = n - 1, x, r;
  while (!(s & 1)) s >>= 1;
  for (int i = 0; i < k; i++) {
    r = powmod(rand64u() % p + 1, s, n);
    for (x = s; x != p && r != 1 && r != p; x <<= 1)
      r = mulmod(r, r, n);
    if (r != p && !(x & 1)) return false;
  }
  return true;
}

/*

Miller-Rabin - Deterministic for all unsigned long long
```

```

Although Miller-Rabin is generally probabilistic, the seven
bases 2, 325, 9375, 28178, 450775, 9780504, 1795265022 have
been proven to deterministically test the primality of all
numbers under 2^64. See: http://miller-rabin.appspot.com/

Complexity: O(log^3(n)).
Warning: Due to the overflow of modular exponentiation,
         this will only work on inputs less than 2^63.

*/

bool is_prime_fast(long long n) {
  static const uint64_t witnesses[] =
    {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
  if (n <= 1) return false;
  if (n <= 3) return true;
  if ((n & 1) == 0) return false;
  uint64_t d = n - 1;
  int s = 0;
  for (; ~d & 1; s++) d >>= 1;
  for (int i = 0; i < 7; i++) {
    if (witnesses[i] > (uint64_t)n - 2) break;
    uint64_t x = powmod(witnesses[i], d, n);
    if (x == 1 || x == (uint64_t)n - 1) continue;
    bool flag = false;
    for (int j = 0; j < s; j++) {
      x = powmod(x, 2, n);
      if (x == 1) return false;
      if (x == (uint64_t)n - 1) {
        flag = true;
        break;
      }
    }
    if (!flag) return false;
  }
  return true;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  int len = 20;
  unsigned long long v[] = {
    0, 1, 2, 3, 4, 5, 11,
    1000000ull,
    772023803ull,
    792904103ull,
    813815117ull,
    834753187ull,
    855718739ull,
    876717799ull,
    897746119ull,
    2147483647ull,
    5705234089ull,
    5914686649ull,
    6114145249ull,
    6339503641ull,
    6548531929ull
  };
  for (int i = 0; i < len; i++) {
    bool p = is_prime(v[i]);
    assert(p == is_prime_fast(v[i]));
    assert(p == is_probable_prime(v[i]));
  }
  return 0;
}
```

4.3.4 Integer Factorization

```
/*

Integer Factorization

*/

#include <algorithm> /* std::sort() */
#include <cmath>     /* sqrt() */
#include <cstdlib>   /* rand(), srand() */
#include <stdint.h>  /* uint64_t */
#include <vector>

/*

Trial division in O(sqrt(n))

Returns a vector of pair<prime divisor, exponent>
e.g. prime_factorize(15435) => {(3,2),(5,1),(7,3)}
because 3^2 * 5^1 * 7^3 = 15435

*/

template<class Int>
std::vector<std::pair<Int, int> > prime_factorize(Int n) {
  std::vector<std::pair<Int, int> > res;
  for (Int d = 2; ; d++) {
    int power = 0, quot = n / d, rem = n - quot * d;
    if (d > quot || (d == quot && rem > 0)) break;
    for (; rem == 0; rem = n - quot * d) {
      power++;
      n = quot;
      quot = n / d;
    }
    if (power > 0) res.push_back(std::make_pair(d, power));
  }
  if (n > 1) res.push_back(std::make_pair(n, 1));
  return res;
}

/*

Trial division in O(sqrt(n))

Returns a sorted vector of all divisors of n.
```

```
e.g. get_all_divisors(28) => {1, 2, 4, 7, 14, 28}

*/

template<class Int>
std::vector<Int> get_all_divisors(Int n) {
  std::vector<Int> res;
  for (int d = 1; d * d <= n; d++) {
    if (n % d == 0) {
      res.push_back(d);
      if (d * d != n)
        res.push_back(n / d);
    }
  }
  std::sort(res.begin(), res.end());
  return res;
}

/*

Fermat's Method ~ O(sqrt(N))

Given a number n, returns one factor of n that is
not necessary prime. Fermat's algorithm is pretty
good when the number you wish to factor has two
factors very near to sqrt(n). Otherwise, it is just
as slow as the basic trial division algorithm.

e.g. 14917627 => 1 (it's a prime), or
     1234567 => 127 (because 127*9721 = 1234567)

*/

long long fermat(long long n) {
  if (n % 2 == 0) return 2;
  long long x = sqrt(n), y = 0;
  long long r = x * x - y * y - n;
  while (r != 0) {
    if (r < 0) {
      r += x + x + 1;
      x++;
    } else {
      r -= y + y + 1;
      y++;
    }
  }
  return x != y ? x - y : x + y;
}

/*

Pollard's rho Algorithm with Brent's Optimization

Brent's algorithm is a much faster variant of Pollard's
rho algorithm using Brent's cycle-finding method. The
following function returns a (not necessarily prime) factor
of n, or n if n is prime. Note that this is not necessarily
guaranteed to always work perfectly. brent(9) may return 9
instead of 3. However, it works well when coupled with trial
```

```
division in the function prime_factorize_big() below.

*/

uint64_t mulmod(uint64_t a, uint64_t b, uint64_t m) {
  uint64_t x = 0, y = a % m;
  for (; b > 0; b >>= 1) {
    if (b & 1) x = (x + y) % m;
    y = (y << 1) % m;
  }
  return x % m;
}

//5 calls to rand() is unnecessary if RAND_MAX is 2^31-1
uint64_t rand64u() {
  return ((uint64_t)(rand() & 0xf) << 60) |
         ((uint64_t)(rand() & 0x7fff) << 45) |
         ((uint64_t)(rand() & 0x7fff) << 30) |
         ((uint64_t)(rand() & 0x7fff) << 15) |
         ((uint64_t)(rand() & 0x7fff));
}

uint64_t gcd(uint64_t a, uint64_t b) {
  return b == 0 ? a : gcd(b, a % b);
}

long long brent(long long n) {
  if (n % 2 == 0) return 2;
  long long y = rand64u() % (n - 1) + 1;
  long long c = rand64u() % (n - 1) + 1;
  long long m = rand64u() % (n - 1) + 1;
  long long g = 1, r = 1, q = 1, ys = 0, hi = 0, x = 0;
  while (g == 1) {
    x = y;
    for (int i = 0; i < r; i++)
      y = (mulmod(y, y, n) + c) % n;
    for (long long k = 0; k < r && g == 1; k += m) {
      ys = y;
      hi = std::min(m, r - k);
      for (int j = 0; j < hi; j++) {
        y = (mulmod(y, y, n) + c) % n;
        q = mulmod(q, x > y ? x - y : y - x, n);
      }
      g = gcd(q, n);
    }
    r *= 2;
  }
  if (g == n) do {
    ys = (mulmod(ys, ys, n) + c) % n;
    g = gcd(x > ys ? x - ys : ys - x, n);
  } while (g <= 1);
  return g;
}

/*

Combines Brent's method with trial division to efficiently
generate the prime factorization of large integers.

```

```
Returns a vector of prime divisors that multiply to n.
e.g. prime_factorize(15435) => {3, 3, 5, 7, 7, 7}
     because 3^2 * 5^1 * 7^3 = 15435

*/

std::vector<long long> prime_factorize_big(long long n) {
  if (n <= 0) return std::vector<long long>(0);
  if (n == 1) return std::vector<long long>(1, 1);
  std::vector<long long> res;
  for (; n % 2 == 0; n /= 2) res.push_back(2);
  for (; n % 3 == 0; n /= 3) res.push_back(3);
  int mx = 1000000; //trial division for factors <= 1M
  for (int i = 5, w = 2; i <= mx; i += w, w = 6 - w) {
    for (; n % i == 0; n /= i) res.push_back(i);
  }
  for (long long p = 0, p1; n > mx; n /= p1) { //brent
    for (p1 = n; p1 != p; p1 = brent(p)) p = p1;
    res.push_back(p1);
  }
  if (n != 1) res.push_back(n);
  sort(res.begin(), res.end());
  return res;
}

/*** Example Usage ***/

#include <cassert>
#include <iostream>
#include <ctime>
using namespace std;

template<class It> void print(It lo, It hi) {
  while (lo != hi) cout << *(lo++) << "␣";
  cout << "\n";
}

template<class It> void printp(It lo, It hi) {
  for (; lo != hi; ++lo)
    cout << "(" << lo->first << "," << lo->second << ")␣";
  cout << "\n";
}

int main() {
  srand(time(0));

  vector< pair<int, int> > v1 = prime_factorize(15435);
  printp(v1.begin(), v1.end());

  vector<int> v2 = get_all_divisors(28);
  print(v2.begin(), v2.end());

  long long n = 1000003ll*100000037ll;
  assert(fermat(n) == 1000003ll);

  vector<long long> v3 = prime_factorize_big(n);
  print(v3.begin(), v3.end());

  return 0;
}
```

4.3.5 Euler's Totient Function

```
/*

Euler's totient function (or Euler's phi function) counts
the positive integers less than or equal to n that are
relatively prime to n. (These integers are sometimes
referred to as totatives of n.) Thus, phi(n) is the number
of integers k in the range [1, n] for which gcd(n, k) = 1.

E.g. if n = 9. Then gcd(9, 3) = gcd(9, 6) = 3 and gcd(9, 9)
= 9. The other six numbers in the range [1, 9], i.e. 1, 2,
4, 5, 7 and 8 are relatively prime to 9. Thus, phi(9) = 6.

*/

#include <vector>

int phi(int n) {
  int res = n;
  for (int i = 2; i * i <= n; i++)
    if (n % i == 0) {
      while (n % i == 0) n /= i;
      res -= res / i;
    }
  if (n > 1) res -= res / n;
  return res;
}

std::vector<int> phi_table(int n) {
  std::vector<int> res(n + 1);
  for (int i = 1; i <= n; i++)
    res[i] = i;
  for (int i = 1; i <= n; i++)
    for (int j = i + i; j <= n; j += i)
      res[j] -= res[i];
  return res;
}

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;

int main() {
  cout << phi(1) << "\n";       //1
  cout << phi(9) << "\n";       //6
  cout << phi(1234567) << "\n"; //1224720

  int n = 1000;
  vector<int> v = phi_table(n);
  for (int i = 0; i <= n; i++)
    assert(v[i] == phi(i));
  return 0;
}
```

4.4 Arbitrary Precision Arithmetic

4.4.1 Big Integers (Simple)

```
/*

Description: Integer arbitrary precision functions.
To use, pass bigints to the functions by addresses.
e.g. add(&a, &b, &c) stores the sum of a and b into c.

Complexity: comp(), to_string(), digit_shift(), add(),
and sub() are O(N) on the number of digits. mul() and
div() are O(N^2). zero_justify() is amortized constant.

*/

#include <string>

struct bigint {
  static const int maxdigits = 1000;

  char dig[maxdigits], sign;
  int last;

  bigint(long long x = 0): sign(x < 0 ? -1 : 1) {
    for (int i = 0; i < maxdigits; i++) dig[i] = 0;
    if (x == 0) { last = 0; return; }
    if (x < 0) x = -x;
    for (last = -1; x > 0; x /= 10) dig[++last] = x % 10;
  }

  bigint(const std::string & s): sign(s[0] == '-' ? -1 : 1) {
    for (int i = 0; i < maxdigits; i++) dig[i] = 0;
    last = -1;
    for (int i = s.size() - 1; i >= 0; i--)
      dig[++last] = (s[i] - '0');
    if (dig[last] + '0' == '-') dig[last--] = 0;
  }
};

void zero_justify(bigint * x) {
  while (x->last > 0 && !x->dig[x->last]) x->last--;
  if (x->last == 0 && x->dig[0] == 0) x->sign = 1;
}

void add(bigint * a, bigint * b, bigint * c);
void sub(bigint * a, bigint * b, bigint * c);

//returns: -1 if a < b, 0 if a == b, or 1 if a > b
int comp(bigint * a, bigint * b) {
  if (a->sign != b->sign) return b->sign;
  if (b->last > a->last) return a->sign;
  if (a->last > b->last) return -a->sign;
  for (int i = a->last; i >= 0; i--) {
    if (a->dig[i] > b->dig[i]) return -a->sign;
    if (b->dig[i] > a->dig[i]) return  a->sign;
  }
  return 0;
}

void add(bigint * a, bigint * b, bigint * c) {
  if (a->sign != b->sign) {
    if (a->sign == -1)
      a->sign = 1, sub(b, a, c), a->sign = -1;
    else
      b->sign = 1, sub(a, b, c), b->sign = -1;
    return;
  }
  c->sign = a->sign;
  c->last = (a->last > b->last ? a->last : b->last) + 1;
  for (int i = 0, carry = 0; i <= c->last; i++) {
    c->dig[i] = (carry + a->dig[i] + b->dig[i]) % 10;
    carry = (carry + a->dig[i] + b->dig[i]) / 10;
  }
  zero_justify(c);
}

void sub(bigint * a, bigint * b, bigint * c) {
  if (a->sign == -1 || b->sign == -1) {
    b->sign *= -1, add(a, b, c), b->sign *= -1;
    return;
  }
  if (comp(a, b) == 1) {
    sub(b, a, c), c->sign = -1;
    return;
  }
  c->last = (a->last > b->last) ? a->last : b->last;
  for (int i = 0, borrow = 0, v; i <= c->last; i++) {
    v = a->dig[i] - borrow;
    if (i <= b->last) v -= b->dig[i];
    if (a->dig[i] > 0) borrow = 0;
    if (v < 0) v += 10, borrow = 1;
    c->dig[i] = v % 10;
  }
  zero_justify(c);
}

void digit_shift(bigint * x, int n) {
  if (!x->last && !x->dig[0]) return;
  for (int i = x->last; i >= 0; i--)
    x->dig[i + n] = x->dig[i];
  for (int i = 0; i < n; i++) x->dig[i] = 0;
  x->last += n;
}

void mul(bigint * a, bigint * b, bigint * c) {
  bigint row = *a, tmp;
  for (int i = 0; i <= b->last; i++) {
    for (int j = 1; j <= b->dig[i]; j++) {
      add(c, &row, &tmp);
      *c = tmp;
    }
    digit_shift(&row, 1);
  }
  c->sign = a->sign * b->sign;
  zero_justify(c);
}

void div(bigint * a, bigint * b, bigint * c) {
  bigint row, tmp;
  int asign = a->sign, bsign = b->sign;
  a->sign = b->sign = 1;
  c->last = a->last;
  for (int i = a->last; i >= 0; i--) {
    digit_shift(&row, 1);
    row.dig[0] = a->dig[i];
    c->dig[i] = 0;
    for (; comp(&row, b) != 1; row = tmp) {
      c->dig[i]++;
      sub(&row, b, &tmp);
    }
  }
  c->sign = (a->sign = asign) * (b->sign = bsign);
  zero_justify(c);
}

std::string to_string(bigint * x) {
  std::string s(x->sign == -1 ? "-" : "");
  for (int i = x->last; i >= 0; i--)
    s += (char)('0' + x->dig[i]);
  return s;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  bigint a("-9899819294989142124"), b("12398124981294214");
  bigint sum; add(&a, &b, &sum);
  bigint dif; sub(&a, &b, &dif);
  bigint prd; mul(&a, &b, &prd);
  bigint quo; div(&a, &b, &quo);
  assert(to_string(&sum) == "-9887421170007847910");
  assert(to_string(&dif) == "-9912217419970436338");
  assert(to_string(&prd) == "-122739196911503356525379735104870536");
  assert(to_string(&quo) == "-798");
  return 0;
}
```

4.4.2 Big Integer and Rational Class

```
/*

The following bigint class is implemented by storing "chunks"
of the big integer in a large base that is a power of 10 so
it can be efficiently stored, operated on, and printed.

It has extensive features including karatsuba multiplication,
exponentiation by squaring, and n-th root using binary search.
```

```
The class is thoroughly templatized, so you can use it as
easily as you do for normal ints. For example, you may use
operators with a bigint and a string (e.g. bigint(1234)+"-567"
and the result will be correctly promoted to a bigint that has
a value of 667). I/O is done using <iostream>. For example:
  bigint a, b; cin >> a >> b; cout << a + b << "\n";
adds two integers together and prints the result, just as you
would expect for a normal int, except with arbitrary precision.
The class also supports other streams such as fstream.

After the bigint class, a class for rational numbers is
implemented, using two bigints to store its numerators and
denominators. It is useful for when exact results of division
operations are needed.

*/

#include <algorithm> /* std::max(), std::swap() */
#include <cmath>     /* sqrt() */
#include <cstdlib>   /* rand() */
#include <iomanip>   /* std::setw(), std::setfill() */
#include <istream>
#include <ostream>
#include <sstream>
#include <stdexcept> /* std::runtime_error() */
#include <string>
#include <utility>   /* std::pair */
#include <vector>

struct bigint {
  //base should be a power of 10 for I/O to work
  //base and base_digits should be consistent
  static const int base = 1000000000, base_digits = 9;

  typedef std::vector<int> vint;
  typedef std::vector<long long> vll;

  vint a; //a[0] stores right-most (least significant) base-digit
  int sign;

  bigint() : sign(1) {}
  bigint(int v) { *this = (long long)v; }
  bigint(long long v) { *this = v; }
  bigint(const std::string & s) { read(s); }
  bigint(const char * s) { read(std::string(s)); }

  void trim() {
    while (!a.empty() && a.back() == 0) a.pop_back();
    if (a.empty()) sign = 1;
  }

  void read(const std::string & s) {
    sign = 1;
    a.clear();
    int pos = 0;
    while (pos < (int)s.size() && (s[pos] == '-' || s[pos] == '+')) {
      if (s[pos] == '-') sign = -sign;
      pos++;
    }
    for (int i = s.size() - 1; i >= pos; i -= base_digits) {
      int x = 0;
      for (int j = std::max(pos, i - base_digits + 1); j <= i; j++)
        x = x * 10 + s[j] - '0';
      a.push_back(x);
    }
    trim();
  }

  void operator = (const bigint & v) {
    sign = v.sign;
    a = v.a;
  }

  void operator = (long long v) {
    sign = 1;
    if (v < 0) sign = -1, v = -v;
    a.clear();
    for (; v > 0; v /= base) a.push_back(v % base);
  }

  bigint operator + (const bigint & v) const {
    if (sign == v.sign) {
      bigint res = v;
      int carry = 0;
      for (int i = 0; i < (int)std::max(a.size(), v.a.size()) || carry; i++) {
        if (i == (int)res.a.size()) res.a.push_back(0);
        res.a[i] += carry + (i < (int)a.size() ? a[i] : 0);
        carry = res.a[i] >= base;
        if (carry) res.a[i] -= base;
      }
      return res;
    }
    return *this - (-v);
  }

  bigint operator - (const bigint & v) const {
    if (sign == v.sign) {
      if (abs() >= v.abs()) {
        bigint res(*this);
        for (int i = 0, carry = 0; i < (int)v.a.size() || carry; i++) {
          res.a[i] -= carry + (i < (int)v.a.size() ? v.a[i] : 0);
          carry = res.a[i] < 0;
          if (carry) res.a[i] += base;
        }
        res.trim();
        return res;
      }
      return -(v - *this);
    }
    return *this + (-v);
  }

  void operator *= (int v) {
    if (v < 0) sign = -sign, v = -v;
    for (int i = 0, carry = 0; i < (int)a.size() || carry; i++) {
      if (i == (int)a.size()) a.push_back(0);
      long long cur = a[i] * (long long)v + carry;
      carry = (int)(cur / base);
      a[i] = (int)(cur % base);
      //asm("divl %%ecx" : "=a"(carry), "=d"(a[i]) : "A"(cur), "c"(base));
    }
    trim();
  }

  bigint operator * (int v) const {
    bigint res(*this);
    res *= v;
    return res;
  }

  static vint convert_base(const vint & a, int l1, int l2) {
    vll p(std::max(l1, l2) + 1);
    p[0] = 1;
    for (int i = 1; i < (int)p.size(); i++) p[i] = p[i - 1] * 10;
    vint res;
    long long cur = 0;
    for (int i = 0, cur_digits = 0; i < (int)a.size(); i++) {
      cur += a[i] * p[cur_digits];
      cur_digits += l1;
      while (cur_digits >= l2) {
        res.push_back((int)(cur % p[l2]));
        cur /= p[l2];
        cur_digits -= l2;
      }
    }
    res.push_back((int)cur);
    while (!res.empty() && res.back() == 0) res.pop_back();
    return res;
  }

  //complexity: O(3N^log2(3)) ~ O(3N^1.585)
  static vll karatsuba_multiply(const vll & a, const vll & b) {
    int n = a.size();
    vll res(n + n);
    if (n <= 32) {
      for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
          res[i + j] += a[i] * b[j];
      return res;
    }
    int k = n >> 1;
    vll a1(a.begin(), a.begin() + k), a2(a.begin() + k, a.end());
    vll b1(b.begin(), b.begin() + k), b2(b.begin() + k, b.end());
    vll a1b1 = karatsuba_multiply(a1, b1);
    vll a2b2 = karatsuba_multiply(a2, b2);
    for (int i = 0; i < k; i++) a2[i] += a1[i];
    for (int i = 0; i < k; i++) b2[i] += b1[i];
    vll r = karatsuba_multiply(a2, b2);
    for (int i = 0; i < (int)a1b1.size(); i++) r[i] -= a1b1[i];
    for (int i = 0; i < (int)a2b2.size(); i++) r[i] -= a2b2[i];
    for (int i = 0; i < (int)r.size(); i++) res[i + k] += r[i];
    for (int i = 0; i < (int)a1b1.size(); i++) res[i] += a1b1[i];
    for (int i = 0; i < (int)a2b2.size(); i++) res[i + n] += a2b2[i];
    return res;
  }

  bigint operator * (const bigint & v) const {
    //if really big values cause overflow, use smaller _base
    static const int _base = 10000, _base_digits = 4;
    vint _a = convert_base(this->a, base_digits, _base_digits);
    vint _b = convert_base(v.a, base_digits, _base_digits);
    vll a(_a.begin(), _a.end());
    vll b(_b.begin(), _b.end());
    while (a.size() < b.size()) a.push_back(0);
    while (b.size() < a.size()) b.push_back(0);
    while (a.size() & (a.size() - 1)) {
      a.push_back(0);
      b.push_back(0);
    }
    vll c = karatsuba_multiply(a, b);
    bigint res;
    res.sign = sign * v.sign;
    for (int i = 0, carry = 0; i < (int)c.size(); i++) {
      long long cur = c[i] + carry;
      res.a.push_back((int)(cur % _base));
      carry = (int)(cur / _base);
    }
    res.a = convert_base(res.a, _base_digits, base_digits);
    res.trim();
    return res;
  }

  bigint operator ^ (const bigint & v) const {
    if (v.sign == -1) return bigint(0);
    bigint x(*this), n(v), res(1);
    while (!n.is_zero()) {
      if (n.a[0] % 2 == 1) res *= x;
      x *= x;
      n /= 2;
    }
    return res;
  }

  friend std::pair<bigint, bigint> divmod(const bigint & a1, const bigint & b1) {
    int norm = base / (b1.a.back() + 1);
    bigint a = a1.abs() * norm;
    bigint b = b1.abs() * norm;
    bigint q, r;
    q.a.resize(a.a.size());
    for (int i = a.a.size() - 1; i >= 0; i--) {
      r *= base;
      r += a.a[i];
      int s1 = r.a.size() <= b.a.size() ? 0 : r.a[b.a.size()];
      int s2 = r.a.size() <= b.a.size() - 1 ? 0 : r.a[b.a.size() - 1];
      int d = ((long long)base * s1 + s2) / b.a.back();
      for (r -= b * d; r < 0; r += b) d--;
      q.a[i] = d;
    }
    q.sign = a1.sign * b1.sign;
    r.sign = a1.sign;
    q.trim();
    r.trim();
    return std::make_pair(q, r / norm);
  }

  bigint operator / (const bigint & v) const { return divmod(*this, v).first; }
  bigint operator % (const bigint & v) const { return divmod(*this, v).second; }

  bigint & operator /= (int v) {
    if (v < 0) sign = -sign, v = -v;
    for (int i = a.size() - 1, rem = 0; i >= 0; i--) {
      long long cur = a[i] + rem * (long long)base;
      a[i] = (int)(cur / v);
      rem = (int)(cur % v);
    }
    trim();
    return *this;
  }

  bigint operator / (int v) const {
    bigint res(*this);
    res /= v;
    return res;
  }

  int operator % (int v) const {
    if (v < 0) v = -v;
    int m = 0;
    for (int i = a.size() - 1; i >= 0; i--)
      m = (a[i] + m * (long long)base) % v;
    return m * sign;
  }

  bigint operator ++(int) { bigint t(*this); operator++(); return t; }
  bigint operator --(int) { bigint t(*this); operator--(); return t; }
  bigint & operator ++() { *this = *this + bigint(1); return *this; }
  bigint & operator --() { *this = *this - bigint(1); return *this; }
  bigint & operator += (const bigint & v) { *this = *this + v; return *this; }
  bigint & operator -= (const bigint & v) { *this = *this - v; return *this; }
  bigint & operator *= (const bigint & v) { *this = *this * v; return *this; }
  bigint & operator /= (const bigint & v) { *this = *this / v; return *this; }
  bigint & operator %= (const bigint & v) { *this = *this % v; return *this; }
  bigint & operator ^= (const bigint & v) { *this = *this ^ v; return *this; }

  bool operator < (const bigint & v) const {
    if (sign != v.sign) return sign < v.sign;
    if (a.size() != v.a.size())
      return a.size() * sign < v.a.size() * v.sign;
    for (int i = a.size() - 1; i >= 0; i--)
      if (a[i] != v.a[i])
        return a[i] * sign < v.a[i] * sign;
    return false;
  }

  bool operator >  (const bigint & v) const { return v < *this; }
  bool operator <= (const bigint & v) const { return !(v < *this); }
  bool operator >= (const bigint & v) const { return !(*this < v); }
  bool operator == (const bigint & v) const { return !(*this < v) && !(v < *this); }
  bool operator != (const bigint & v) const { return *this < v || v < *this; }

  int size() const {
    if (a.empty()) return 1;
    std::ostringstream oss;
    oss << a.back();
    return oss.str().length() + base_digits*(a.size() - 1);
  }

  bool is_zero() const {
    return a.empty() || (a.size() == 1 && !a[0]);
  }

  bigint operator - () const {
    bigint res(*this);
    res.sign = -sign;
    return res;
  }

  bigint abs() const {
    bigint res(*this);
    res.sign *= res.sign;
    return res;
  }

  friend bigint abs(const bigint & a) {
    return a.abs();
  }

  friend bigint gcd(const bigint & a, const bigint & b) {
    return b.is_zero() ? a : gcd(b, a % b);
  }

  friend bigint lcm(const bigint & a, const bigint & b) {
    return a / gcd(a, b) * b;
  }

  friend bigint sqrt(const bigint & x) {
    bigint a = x;
    while (a.a.empty() || a.a.size() % 2 == 1) a.a.push_back(0);
    int n = a.a.size();
    int firstdig = sqrt((double)a.a[n - 1] * base + a.a[n - 2]);
    int norm = base / (firstdig + 1);
    a *= norm;
    a *= norm;
    while (a.a.empty() || a.a.size() % 2 == 1) a.a.push_back(0);
    bigint r = (long long)a.a[n - 1] * base + a.a[n - 2];
    firstdig = sqrt((double)a.a[n - 1] * base + a.a[n - 2]);
    int q = firstdig;
    bigint res;
    for (int j = n / 2 - 1; j >= 0; j--) {
      for (;; q--) {
        bigint r1 = (r - (res * 2 * base + q) * q) * base * base + (j > 0 ?
                      (long long)a.a[2 * j - 1] * base + a.a[2 * j - 2] : 0);
        if (r1 >= 0) {
          r = r1;
          break;
        }
      }
      res = (res * base) + q;
      if (j > 0) {
        int d1 = res.a.size() + 2 < r.a.size() ? r.a[res.a.size() + 2] : 0;
        int d2 = res.a.size() + 1 < r.a.size() ? r.a[res.a.size() + 1] : 0;
        int d3 = res.a.size() < r.a.size() ? r.a[res.a.size()] : 0;
        q = ((long long)d1*base*base + (long long)d2*base + d3)/(firstdig * 2);
      }
    }
    res.trim();
    return res / norm;
  }

  friend bigint nthroot(const bigint & x, const bigint & n) {
    bigint hi = 1;
    while ((hi ^ n) <= x) hi *= 2;
    bigint lo = hi / 2, mid, midn;
    while (lo < hi) {
      mid = (lo + hi) / 2;
      midn = mid ^ n;
      if (lo < mid && midn < x) {
        lo = mid;
      } else if (mid < hi && x < midn) {
        hi = mid;
      } else {
        return mid;
      }
    }
    return mid + 1;
  }

  friend std::istream & operator >> (std::istream & in, bigint & v) {
    std::string s;
    in >> s;
    v.read(s);
    return in;
  }

  friend std::ostream & operator << (std::ostream & out, const bigint & v) {
    if (v.sign == -1) out << '-';
    out << (v.a.empty() ? 0 : v.a.back());
    for (int i = v.a.size() - 2; i >= 0; i--)
      out << std::setw(base_digits) << std::setfill('0') << v.a[i];
    return out;
  }

  std::string to_string() const {
    std::ostringstream oss;
    if (sign == -1) oss << '-';
    oss << (a.empty() ? 0 : a.back());
    for (int i = a.size() - 2; i >= 0; i--)
      oss << std::setw(base_digits) << std::setfill('0') << a[i];
    return oss.str();
  }

  long long to_llong() const {
    long long res = 0;
    for (int i = a.size() - 1; i >= 0; i--)
      res = res * base + a[i];
    return res * sign;
  }

  double to_double() const {
    std::stringstream ss(to_string());
    double res;
    ss >> res;
    return res;
  }

  long double to_ldouble() const {
    std::stringstream ss(to_string());
    long double res;
    ss >> res;
    return res;
  }

  static bigint rand(int len) {
    if (len == 0) return bigint(0);
    std::string s(1, '1' + (::rand() % 9));
    for (int i = 1; i < len; i++) s += '0' + (::rand() % 10);
    return bigint(s);
  }
};

template<class T> bool operator > (const T & a, const bigint & b) { return bigint(a) > b; }
template<class T> bool operator < (const T & a, const bigint & b) { return bigint(a) < b; }
template<class T> bool operator >= (const T & a, const bigint & b) { return bigint(a) >= b; }
template<class T> bool operator <= (const T & a, const bigint & b) { return bigint(a) <= b; }
template<class T> bool operator == (const T & a, const bigint & b) { return bigint(a) == b; }
template<class T> bool operator != (const T & a, const bigint & b) { return bigint(a) != b; }
template<class T> bigint operator + (const T & a, const bigint & b) { return bigint(a) + b;  }
template<class T> bigint operator - (const T & a, const bigint & b) { return bigint(a) - b;  }
template<class T> bigint operator ^ (const T & a, const bigint & b) { return bigint(a) ^ b;  }

/*

Exclude *, /, and % to force a user decision between int and bigint algorithms

bigint operator * (bigint a, bigint b) vs. bigint operator * (bigint a, int b)
bigint operator / (bigint a, bigint b) vs. bigint operator / (bigint a, int b)
bigint operator % (bigint a, bigint b) vs. int operator % (bigint a, int b)

*/

struct rational {
  bigint num, den;

  rational(): num(0), den(1) {}
  rational(long long n): num(n), den(1) {}
  rational(const bigint & n) : num(n), den(1) {}

  template<class T1, class T2>
  rational(const T1 & n, const T2 & d): num(n), den(d) {
    if (den == 0)
      throw std::runtime_error("Rational␣division␣by␣zero.");
    if (den < 0) {
      num = -num;
      den = -den;
    }
    bigint a(num < 0 ? -num : num), b(den), tmp;
    while (a != 0 && b != 0) {
      tmp = a % b;
      a = b;
      b = tmp;
    }
    bigint gcd = (b == 0) ? a : b;
    num /= gcd;
    den /= gcd;
  }

  bool operator < (const rational & r) const {
    return num * r.den < r.num * den;
  }

  bool operator > (const rational & r) const {
    return r.num * den < num * r.den;
  }

  bool operator <= (const rational & r) const {
    return !(r < *this);
  }

  bool operator >= (const rational & r) const {
    return !(*this < r);
  }

  bool operator == (const rational & r) const {
    return num == r.num && den == r.den;
  }

  bool operator != (const rational & r) const {
    return num != r.num || den != r.den;
  }

  rational operator + (const rational & r) const {
    return rational(num * r.den + r.num * den, den * r.den);
  }

  rational operator - (const rational & r) const {
    return rational(num * r.den - r.num * den, r.den * den);
  }

  rational operator * (const rational & r) const {
    return rational(num * r.num, r.den * den);
  }

  rational operator / (const rational & r) const {
    return rational(num * r.den, den * r.num);
  }

  rational operator % (const rational & r) const {
    return *this - r * rational(num * r.den / (r.num * den), 1);
  }

  rational operator ^ (const bigint & p) const {
    return rational(num ^ p, den ^ p);
  }

  rational operator ++(int) { rational t(*this); operator++(); return t; }
  rational operator --(int) { rational t(*this); operator--(); return t; }
  rational & operator ++() { *this = *this + 1; return *this; }
  rational & operator --() { *this = *this - 1; return *this; }
  rational & operator += (const rational & r) { *this = *this + r; return *this; }
  rational & operator -= (const rational & r) { *this = *this - r; return *this; }
  rational & operator *= (const rational & r) { *this = *this * r; return *this; }
  rational & operator /= (const rational & r) { *this = *this / r; return *this; }
  rational & operator %= (const rational & r) { *this = *this % r; return *this; }
  rational & operator ^= (const bigint & r) { *this = *this ^ r; return *this; }

  rational operator - () const {
    return rational(-num, den);
  }

  rational abs() const {
    return rational(num.abs(), den);
  }

  long long to_llong() const {
    return num.to_llong() / den.to_llong();
  }

  double to_double() const {
    return num.to_double() / den.to_double();
  }

  friend rational abs(const rational & r) {
    return rational(r.num.abs(), r.den);
  }

  friend std::istream & operator >> (std::istream & in, rational & r) {
    std::string s;
    in >> r.num;
    r.den = 1;
    return in;
  }

  friend std::ostream & operator << (std::ostream & out, const rational & r) {
    out << r.num << "/" << r.den;
    return out;
  }

  //rational in range [0, 1] with precision no greater than prec
  static rational rand(int prec) {
    rational r(bigint::rand(prec), bigint::rand(prec));
    if (r.num > r.den) std::swap(r.num, r.den);
    return r;
  }
};

template<class T> bool operator > (const T & a, const rational & b) { return rational(a) > b; }
template<class T> bool operator < (const T & a, const rational & b) { return rational(a) < b; }
template<class T> bool operator >= (const T & a, const rational & b) { return rational(a) >= b; }
template<class T> bool operator <= (const T & a, const rational & b) { return rational(a) <= b; }
template<class T> bool operator == (const T & a, const rational & b) { return rational(a) == b; }
template<class T> bool operator != (const T & a, const rational & b) { return rational(a) != b; }
template<class T> rational operator + (const T & a, const rational & b) { return rational(a) + b;  }
template<class T> rational operator - (const T & a, const rational & b) { return rational(a) - b;  }
template<class T> rational operator * (const T & a, const rational & b) { return rational(a) * b;  }
template<class T> rational operator / (const T & a, const rational & b) { return rational(a) / b;  }
template<class T> rational operator % (const T & a, const rational & b) { return rational(a) % b;  }
template<class T> rational operator ^ (const T & a, const rational & b) { return rational(a) ^ b;  }

/*** Example Usage ***/

```

```
#include <cassert>
#include <cstdio>
#include <ctime>
#include <iostream>
using namespace std;

int main() {
  for (int i = 0; i < 20; i++) {
    int n = rand() % 100 + 1;
    bigint a = bigint::rand(n);
    bigint res = sqrt(a);
    bigint xx(res * res);
    bigint yy(res + 1);
    yy *= yy;
    assert(xx <= a && yy > a);
    int m = rand() % n + 1;
    bigint b = bigint::rand(m) + 1;
    res = a / b;
    xx = res * b;
    yy = b * (res + 1);
    assert(a >= xx && a < yy);
  }

  assert("995291497" ==
    nthroot(bigint("981298591892498189249182998429898124"), 4));

  bigint x(5);
  x = -6;
  assert(x.to_llong() == -6ll);
  assert(x.to_string() == "-6");

  clock_t start;

  start = clock();
  bigint c = bigint::rand(10000) / bigint::rand(2000);
  cout << "Div␣took␣" << (float)(clock() - start)/CLOCKS_PER_SEC << "s\n";

  start = clock();
  assert((20^bigint(12345)).size() == 16062);
  cout << "Pow␣took␣" << (float)(clock() - start)/CLOCKS_PER_SEC << "s\n";

  int nn = -21, dd = 2;
  rational n(nn, 1), d(dd);
  cout << (nn % dd) << "\n";
  cout << (n % d) << "\n";
  cout << fmod(-5.3, -1.7) << "\n";
  cout << rational(-53, 10) % rational(-17, 10) << "\n";
  cout << rational(-53, 10).abs() << "\n";
  cout << (rational(-53, 10) ^ 20) << "\n";
  cout << rational::rand(20) << "\n";
  return 0;
}
```

4.4.3 FFT and Multiplication

```
/*

```

```
A discrete Fourier transform (DFT) converts a list of equally
spaced samples of a function into the list of coefficients of
a finite combination of complex sinusoids, ordered by their
frequencies, that has those same sample values. A Fast Fourier
Transform (FFT) rapidly computes the DFT by factorizing the
DFT matrix into a product of sparse (mostly zero) factors.
The FFT can be used to solve problems such as efficiently
multiplying big integers or polynomials

The fft() function below is a generic function that will
work well in many applications beyond just multiplying
big integers. While Karatsuba multiplication is ~ O(n^1.58),
the complexity of the fft multiplication is only O(n log n).

Note that mul(string, string) in the following implementation
only works for strings of strictly digits from '0' to '9'.
It is also easy to adapt this for the bigint class in the
previous section. Simply replace the old bigint operator *
definition with the following modified version of mul():

  bigint operator * (const bigint & v) const {
    static const int _base = 10000, _base_digits = 4;
    vint _a = convert_base(this->a, base_digits, _base_digits);
    vint _b = convert_base(v.a, base_digits, _base_digits);
    int len = 32 - __builtin_clz(std::max(_a.size(), _b.size()) - 1);
    len = 1 << (len + 1);
    vcd a(len), b(len);
    for (int i = 0; i < _a.size(); i++) a[i] = cd(_a[i], 0);
    for (int i = 0; i < _b.size(); i++) b[i] = cd(_b[i], 0);
    a = fft(a);
    b = fft(b);
    for (int i = 0; i < len; i++) {
      double real = a[i].real() * b[i].real() - a[i].imag() * b[i].imag();
      a[i].imag() = a[i].imag() * b[i].real() + b[i].imag() * a[i].real();
      a[i].real() = real;
    }
    a = fft(a, true);
    vll c(len);
    for (int i = 0; i < len; i++) c[i] = (long long)(a[i].real() + 0.5);
    bigint res;
    res.sign = sign * v.sign;
    for (int i = 0, carry = 0; i < c.size(); i++) {
      long long cur = c[i] + carry;
      res.a.push_back((int)(cur % _base));
      carry = (int)(cur / _base);
    }
    res.a = convert_base(res.a, _base_digits, base_digits);
    res.trim();
    return res;
  }

*/

#include <algorithm> /* std::max(), std::reverse() */
#include <cmath>     /* M_PI, cos(), sin() */
#include <complex>
#include <iomanip>   /* std::setw(), std::setfill() */
#include <sstream>
#include <string>
```

```
#include <vector>

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

vcd fft(const vcd & v, bool inverse = false) {
  static const double PI = acos(-1.0);
  int n = v.size(), k = 0, high1 = -1;
  while ((1 << k) < n) k++;
  std::vector<int> rev(n);
  rev[0] = 0;
  for (int i = 1; i < n; i++) {
    if ((i & (i - 1)) == 0) high1++;
    rev[i] = rev[i ^ (1 << high1)];
    rev[i] |= (1 << (k - high1 - 1));
  }
  vcd roots(n), res(n);
  for (int i = 0; i < n; i++) {
    double alpha = 2 * PI * i / n;
    roots[i] = cd(cos(alpha), sin(alpha));
  }
  for (int i = 0; i < n; i++) res[i] = v[rev[i]];
  for (int len = 1; len < n; len <<= 1) {
    vcd tmp(n);
    int rstep = roots.size() / (len * 2);
    for (int pdest = 0; pdest < n;) {
      int p1 = pdest;
      for (int i = 0; i < len; i++) {
        cd val = roots[i * rstep] * res[p1 + len];
        tmp[pdest] = res[p1] + val;
        tmp[pdest + len] = res[p1] - val;
        pdest++, p1++;
      }
      pdest += len;
    }
    res.swap(tmp);
  }
  if (inverse) {
    for (int i = 0; i < (int)res.size(); i++) res[i] /= v.size();
    std::reverse(res.begin() + 1, res.end());
  }
  return res;
}

typedef std::vector<long long> vll;

vll mul(const vll & va, const vll & vb) {
  int len = 32 - __builtin_clz(std::max(va.size(), vb.size()) - 1);
  len = 1 << (len + 1);
  vcd a(len), b(len);
  for (int i = 0; i < (int)va.size(); i++) a[i] = cd(va[i], 0);
  for (int i = 0; i < (int)vb.size(); i++) b[i] = cd(vb[i], 0);
  a = fft(a);
  b = fft(b);
  for (int i = 0; i < len; i++) {
    double real = a[i].real() * b[i].real() - a[i].imag() * b[i].imag();
    a[i].imag() = a[i].imag() * b[i].real() + b[i].imag() * a[i].real();
    a[i].real() = real;
  }
  a = fft(a, true);
  vll res(len);
  for (int i = 0; i < len; i++) res[i] = (long long)(a[i].real() + 0.5);
  return res;
}

const int base = 10000, base_digits = 4;

std::string mul(const std::string & as, const std::string & bs) {
  vll a, b;
  for (int i = as.size() - 1; i >= 0; i -= base_digits) {
    int x = 0;
    for (int j = std::max(0, i - base_digits + 1); j <= i; j++)
      x = x * 10 + as[j] - '0';
    a.push_back(x);
  }
  for (int i = bs.size() - 1; i >= 0; i -= base_digits) {
    int x = 0;
    for (int j = std::max(0, i - base_digits + 1); j <= i; j++)
      x = x * 10 + bs[j] - '0';
    b.push_back(x);
  }
  vll c = mul(a, b);
  long long carry = 0;
  for (int i = 0; i < (int)c.size(); i++) {
    c[i] += carry;
    carry = c[i] / base;
    c[i] %= base;
  }
  while (c.back() == 0) c.pop_back();
  if (c.empty()) c.push_back(0);
  std::ostringstream oss;
  oss << (c.empty() ? 0 : c.back());
  for (int i = c.size() - 2; i >= 0; i--)
    oss << std::setw(base_digits) << std::setfill('0') << c[i];
  return oss.str();
}

/*** Example Usage ***/

#include <cassert>

int main() {
  assert(mul("98904189", "244212") == "24153589804068");
  return 0;
}
```

4.5 Linear Algebra

4.5.1 Matrix Class

```
/*

Basic matrix class with support for arithmetic operations
as well as matrix multiplication and exponentiation. You
can access/modify indices using m(r, c) or m[r][c]. You
can also treat it as a 2d vector, since the cast operator
to a reference to its internal 2d vector is defined. This
makes it compatible with the 2d vector functions such as
det() and lu_decompose() in later sections.

*/

#include <ostream>
#include <stdexcept> /* std::runtime_error() */
#include <vector>

template<class val_t> class matrix {
  int r, c;
  std::vector<std::vector<val_t> > mat;

 public:
  matrix(int rows, int cols, val_t init = val_t()) {
    r = rows;
    c = cols;
    mat.resize(r, std::vector<val_t>(c, init));
  }

  matrix(const std::vector<std::vector<val_t> > & m) {
    r = m.size();
    c = m[0].size();
    mat = m;
    mat.resize(r, std::vector<val_t>(c));
  }

  template<size_t rows, size_t cols>
  matrix(val_t (&init)[rows][cols]) {
    r = rows;
    c = cols;
    mat.resize(r, std::vector<val_t>(c));
    for (int i = 0; i < r; i++)
      for (int j = 0; j < c; j++)
        mat[i][j] = init[i][j];
  }

  operator std::vector<std::vector<val_t> > &() { return mat; }
  val_t & operator() (int r, int c) { return mat[r][c]; }
  std::vector<val_t> & operator[] (int r) { return mat[r]; }
  val_t at(int r, int c) const { return mat[r][c]; }
  int rows() const { return r; }
  int cols() const { return c; }

  friend bool operator <  (const matrix & a, const matrix & b) { return a.mat < b.mat; }
  friend bool operator >  (const matrix & a, const matrix & b) { return a.mat > b.mat; }
  friend bool operator <= (const matrix & a, const matrix & b) { return a.mat <= b.mat; }
  friend bool operator >= (const matrix & a, const matrix & b) { return a.mat >= b.mat; }
  friend bool operator == (const matrix & a, const matrix & b) { return a.mat == b.mat; }
  friend bool operator != (const matrix & a, const matrix & b) { return a.mat != b.mat; }

  friend matrix operator + (const matrix & a, const matrix & b) {
    if (a.r != b.r || a.c != b.c)
      throw std::runtime_error("Matrix dimensions don't match.");
    matrix res(a);
    for (int i = 0; i < res.r; i++)
      for (int j = 0; j < res.c; j++)
```

```
        res.mat[i][j] += b.mat[i][j];
    return res;
  }

  friend matrix operator - (const matrix & a, const matrix & b) {
    if (a.r != b.r || a.c != b.c)
      throw std::runtime_error("Matrix␣dimensions␣don’t␣match.");
    matrix res(a);
    for (int i = 0; i < a.r; i++)
      for (int j = 0; j < a.c; j++)
        res.mat[i][j] -= b.mat[i][j];
    return res;
  }

  friend matrix operator * (const matrix & a, const matrix & b) {
    if (a.c != b.r)
      throw std::runtime_error("#␣of␣a␣cols␣must␣equal␣#␣of␣b␣rows.");
    matrix res(a.r, b.c, 0);
    for (int i = 0; i < a.r; i++)
      for (int j = 0; j < b.c; j++)
        for (int k = 0; k < a.c; k++)
          res.mat[i][j] += a.mat[i][k] * b.mat[k][j];
    return res;
  }

  friend matrix operator + (const matrix & a, const val_t & v) {
    matrix res(a);
    for (int i = 0; i < a.r; i++)
      for (int j = 0; j < a.c; j++) res.mat[i][j] += v;
    return res;
  }

  friend matrix operator - (const matrix & a, const val_t & v) {
    matrix res(a);
    for (int i = 0; i < a.r; i++)
      for (int j = 0; j < a.c; j++) res.mat[i][j] -= v;
    return res;
  }

  friend matrix operator * (const matrix & a, const val_t & v) {
    matrix res(a);
    for (int i = 0; i < a.r; i++)
      for (int j = 0; j < a.c; j++) res.mat[i][j] *= v;
    return res;
  }

  friend matrix operator / (const matrix & a, const val_t & v) {
    matrix res(a);
    for (int i = 0; i < a.r; i++)
      for (int j = 0; j < a.c; j++)
        res.mat[i][j] /= v;
    return res;
  }

  //raise matrix to the n-th power. precondition: a must be a square matrix
  friend matrix operator ^ (const matrix & a, unsigned int n) {
    if (a.r != a.c)
      throw std::runtime_error("Matrix␣must␣be␣square␣for␣exponentiation.");
    if (n == 0) return identity_matrix(a.r);
    if (n % 2 == 0) return (a * a) ^ (n / 2);
    return a * (a ^ (n - 1));
  }

  //returns a^1 + a^2 + ... + a^n
  friend matrix powsum(const matrix & a, unsigned int n) {
    if (n == 0) return matrix(a.r, a.r);
    if (n % 2 == 0)
      return powsum(a, n / 2) * (identity_matrix(a.r) + (a ^ (n / 2)));
    return a + a * powsum(a, n - 1);
  }

  matrix & operator += (const matrix & m) { *this = *this + m; return *this; }
  matrix & operator -= (const matrix & m) { *this = *this - m; return *this; }
  matrix & operator *= (const matrix & m) { *this = *this * m; return *this; }
  matrix & operator += (const val_t & v)  { *this = *this + v; return *this; }
  matrix & operator -= (const val_t & v)  { *this = *this - v; return *this; }
  matrix & operator *= (const val_t & v)  { *this = *this * v; return *this; }
  matrix & operator /= (const val_t & v)  { *this = *this / v; return *this; }
  matrix & operator ^= (unsigned int n)   { *this = *this ^ n; return *this; }

  static matrix identity_matrix(int n) {
    matrix res(n, n);
    for (int i = 0; i < n; i++) res[i][i] = 1;
    return res;
  }

  friend std::ostream & operator << (std::ostream & out, const matrix & m) {
    out << "[";
    for (int i = 0; i < m.r; i++) {
      out << (i > 0 ? ",[" : "[");
      for (int j = 0; j < m.c; j++)
        out << (j > 0 ? "," : "") << m.mat[i][j];
      out << "]";
    }
    out << "]";
    return out;
  }
};

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;

int main() {
  int a[2][2] = {{1,8}, {5,9}};
  matrix<int> m(5, 5, 10), m2(a);
  m += 10;
  m[0][0] += 10;
  assert(m[0][0] == 30 && m[1][1] == 20);
  assert(powsum(m2, 3) == m2 + m2*m2 + (m2^3));
  return 0;
}
```

4.5.2 Determinant (Gauss)

```
/*

The following are ways to compute the determinant of a
matrix directly using Gaussian elimination. See the
following section for a generalized solution using LU
decompositions. Since the determinant can get very large,
look out for overflows and floating-point inaccuracies.
Bignums are recommended for maximal correctness.

Complexity: O(N^3), except for the adjustment for
overflow in the integer det() function.

Precondition: All input matrices must be square.

*/

#include <algorithm> /* std::swap() */
#include <cassert>
#include <cmath>     /* fabs() */
#include <map>
#include <vector>

static const double eps = 1e-10;
typedef std::vector<std::vector<int> > vvi;
typedef std::vector<std::vector<double> > vvd;

double det(vvd a) {
  int n = a.size();
  assert(!a.empty() && n == (int)a[0].size());
  double res = 1;
  std::vector<bool> used(n, false);
  for (int i = 0; i < n; i++) {
    int p;
    for (p = 0; p < n; p++)
      if (!used[p] && fabs(a[p][i]) > eps)
        break;
    if (p >= n) return 0;
    res *= a[p][i];
    used[p] = true;
    double z = 1 / a[p][i];
    for (int j = 0; j < n; j++) a[p][j] *= z;
    for (int j = 0; j < n; j++) {
      if (j == p) continue;
      z = a[j][i];
      for (int k = 0; k < n; k++)
        a[j][k] -= z * a[p][k];
    }
  }
  return res;
}

/*

Determinant of Integer Matrix

This is prone to overflow, so it is recommended you use your
own bigint class instead of long long. At the end of this
function, the final answer is found as a product of powers.
You have two choices: change the "#if 0" to "#if 1" and use
```

```
the naive method to compute this product and risk overflow,
or keep it as "#if 0" and try to make the situation better
through prime factorization (less efficient). Note that
even in the prime factorization method, overflow may happen
if the final answer is too big for a long long.

*/

//C++98 doesn't have an abs() for long long
template<class T> inline T _abs(const T & x) {
  return x < 0 ? -x : x;
}

long long det(const vvi & a) {
  int n = a.size();
  assert(!a.empty() && n == (int)a[0].size());
  long long b[n][n], det = 1;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) b[i][j] = a[i][j];
  int sign = 1, exponent[n];
  for (int i = 0; i < n; i++) {
    exponent[i] = 0;
    int k = i;
    for (int j = i + 1; j < n; j++) {
      if (b[k][i] == 0 || (b[j][i] != 0 && _abs(b[k][i]) > _abs(b[j][i])))
        k = j;
    }
    if (b[k][i] == 0) return 0;
    if (i != k) {
      sign = -sign;
      for (int j = 0; j < n; j++)
        std::swap(b[i][j], b[k][j]);
    }
    exponent[i]++;
    for (int j = i + 1; j < n; j++)
      if (b[j][i] != 0) {
        for (int p = i + 1; p < n; ++p)
          b[j][p] = b[j][p] * b[i][i] - b[i][p] * b[j][i];
        exponent[i]--;
      }
  }

#if 0
  for (int i = 0; i < n; i++)
    for (; exponent[i] > 0; exponent[i]--)
      det *= b[i][i];
  for (int i = 0; i < n; i++)
    for (; exponent[i] < 0; exponent[i]++)
      det /= b[i][i];
#else
  std::map<long long, int> m;
  for (int i = 0; i < n; i++) {
    long long x = b[i][i];
    for (long long d = 2; ; d++) {
      long long power = 0, quo = x / d, rem = x - quo * d;
      if (d > quo || (d == quo && rem > 0)) break;
      for (; rem == 0; rem = x - quo * d) {
        power++;
        x = quo;
        quo = x / d;
      }
      if (power > 0) m[d] += power * exponent[i];
    }
    if (x > 1) m[x] += exponent[i];
  }
  std::map<long long, int>::iterator it;
  for (it = m.begin(); it != m.end(); ++it)
    for (int i = 0; i < it->second; i++)
      det *= it->first;
#endif

  return sign < 0 ? -det : det;
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  const int n = 3;
  int a[n][n] = {{6,1,1},{4,-2,5},{2,8,7}};
  vvi v1(n);
  vvd v2(n);
  for (int i = 0; i < n; i++) {
    v1[i] = vector<int>(a[i], a[i] + n);
    v2[i] = vector<double>(a[i], a[i] + n);
  }
  int d1 = det(v1);
  int d2 = (int)det(v2);
  assert(d1 == d2 && d2 == -306);
  return 0;
}
```

4.5.3 Gaussian Elimination

```
/*

Given a system of m linear equations with n unknowns:

A(1,1)*x(1) + A(1,2)*x(2) + ... + A(1,n)*x(n) = B(1)
A(2,1)*x(1) + A(2,2)*x(2) + ... + A(2,n)*x(n) = B(2)
                        ...
A(m,1)*x(1) + A(m,2)*x(2) + ... + A(m,n)*x(n) = B(m)

For any system of linear equations, there will either
be no solution (in 2d, lines are parallel), a single
solution (in 2d, the lines intersect at a point), or
or infinite solutions (in 2d, lines are the same).

Using Gaussian elimination in O(n^3), this program
solves for the values of x(1) ... x(n) or determines
that no unique solution of x() exists. Note that
the implementation below uses 0-based indices.

*/
```

```

#include <algorithm> /* std::swap() */
#include <cmath>     /* fabs() */
#include <vector>

const double eps = 1e-9;
typedef std::vector<double> vd;
typedef std::vector<vd> vvd;

//note: A[i][n] stores B[i]
//if no unique solution found, returns empty vector
vd solve_system(vvd A) {
  int m = A.size(), n = A[0].size() - 1;
  vd x(n);
  if (n > m) goto fail;
  for (int k = 0; k < n; k++) {
    double mv = 0;
    int mi = -1;
    for (int i = k; i < m; i++)
      if (mv < fabs(A[i][k])) {
        mv = fabs(A[i][k]);
        mi = i;
      }
    if (mv < eps) goto fail;
    for (int i = 0; i <= n; i++)
      std::swap(A[mi][i], A[k][i]);
    for (int i = k + 1; i < m; i++) {
      double v = A[i][k] / A[k][k];
      for (int j = k; j <= n; j++)
        A[i][j] -= v * A[k][j];
      A[i][k] = 0;
    }
  }
  for (int i = n; i < m; i++)
    if (fabs(A[i][n]) > eps) goto fail;
  for (int i = n - 1; i >= 0; i--) {
    if (fabs(A[i][i]) < eps) goto fail;
    double v = 0;
    for (int j = i + 1; j < n; j++)
      v += A[i][j] * x[j];
    v = A[i][n] - v;
    x[i] = v / A[i][i];
  }
  return x;
fail:
  return vd();
}

/*** Example Usage (wcipeg.com/problem/syssolve) ***/

#include <iostream>
using namespace std;

int main() {
  int n, m;
  cin >> n >> m;
  vvd a(m, vd(n + 1));
  for (int i = 0; i < m; i++)
    for (int j = 0; j <= n; j++)
```

```
      cin >> a[i][j];
  vd x = solve_system(a);
  if (x.empty()) {
    cout << "NO␣UNIQUE␣SOLUTION\n";
  } else {
    cout.precision(6);
    for (int i = 0; i < n; i++)
      cout << fixed << x[i] << "\n";
  }
  return 0;
}
```

4.5.4 LU Decomposition

```
/*

The LU (lower upper) decomposition of a matrix is a factorization
of a matrix as the product of a lower triangular matrix and an
upper triangular matrix. With the LU decomposition, we can solve
many problems, including the determinant of the matrix, a systems
of linear equations, and the inverse of a matrix.

Note: in the following implementation, each call to det(),
solve_system(), and inverse() recomputes the lu decomposition.
For the same matrix, you should precompute the lu decomposition
and reuse it for several of these operations afterwards.

Complexity: O(n^3) for lu_decompose(). det() uses the running time
of lu_decompose(), plus an addition O(n) term. solve_system() and
inverse() both have the running time of lu_decompose(), plus an
additional O(n^3) term.

*/

#include <algorithm> /* std::swap() */
#include <cassert>
#include <cmath>     /* fabs() */
#include <vector>

static const double eps = 1e-10;
typedef std::vector<double> vd;
typedef std::vector<vd> vvd;

/*

LU decomposition with Gauss-Jordan elimination. This is generalized
for rectangular matrices. Since the resulting L and U matrices have
all mutually exclusive 0's (except when i == j), we can merge them
into a single LU matrix to save memory. Note: l[i][i] = 1 for all i.

Optionally determine the permutation vector p. If an array p is
passed, p[i] will be populated such that p[i] is the only column of
the i-th row of the permutation matrix that is equal to 1.

Returns: a matrix m, the merged lower/upper triangular matrix:
         m[i][j] = l[i][j] (for i > j) or u[i][j] (for i <= j)

```

```
*/

vvd lu_decompose(vvd a, int * detsign = 0, int * p = 0) {
  int n = a.size(), m = a[0].size();
  int sign = 1;
  if (p != 0)
    for (int i = 0; i < n; i++) p[i] = i;
  for (int r = 0, c = 0; r < n && c < m; r++, c++) {
    int pr = r;
    for (int i = r + 1; i < n; i++)
      if (fabs(a[i][c]) > fabs(a[pr][c]))
        pr = i;
    if (fabs(a[pr][c]) <= eps) {
      r--;
      continue;
    }
    if (pr != r) {
      if (p != 0) std::swap(p[r], p[pr]);
      sign = -sign;
      for (int i = 0; i < m; i++)
        std::swap(a[r][i], a[pr][i]);
    }
    for (int s = r + 1; s < n; s++) {
      a[s][c] /= a[r][c];
      for (int d = c + 1; d < m; d++)
        a[s][d] -= a[s][c] * a[r][d];
    }
  }
  if (detsign != 0) *detsign = sign;
  return a;
}

double getl(const vvd & lu, int i, int j) {
  if (i > j) return lu[i][j];
  return i < j ? 0.0 : 1.0;
}

double getu(const vvd & lu, int i, int j) {
  return i <= j ? lu[i][j] : 0.0;
}

//Precondition: A is square matrix.
double det(const vvd & a) {
  int n = a.size(), detsign;
  assert(!a.empty() && n == (int)a[0].size());
  vvd lu = lu_decompose(a, &detsign);
  double det = 1;
  for (int i = 0; i < n; i++)
    det *= lu[i][i];
  return detsign < 0 ? -det : det;
}

/*

Solves system of linear equations with forward/backwards
substitution. Precondition: A must be n*n and B must be n*m.
Returns: an n by m matrix X such that A*X = B.

*/
```

```

vvd solve_system(const vvd & a, const vvd & b) {
  int n = b.size(), m = b[0].size();
  assert(!a.empty() && n == (int)a.size() && n == (int)a[0].size());
  int detsign, p[a.size()];
  vvd lu = lu_decompose(a, &detsign, p);
  //forward substitute for Y in L*Y = B
  vvd y(n, vd(m));
  for (int j = 0; j < m; j++) {
    y[0][j] = b[p[0]][j] / getl(lu, 0, 0);
    for (int i = 1; i < n; i++) {
      double s = 0;
      for (int k = 0; k < i; k++)
        s += getl(lu, i, k) * y[k][j];
      y[i][j] = (b[p[i]][j] - s) / getl(lu, i, i);
    }
  }
  //backward substitute for X in U*X = Y
  vvd x(n, vd(m));
  for (int j = 0; j < m; j++) {
    x[n - 1][j] = y[n - 1][j] / getu(lu, n-1, n-1);
    for (int i = n - 2; i >= 0; i--) {
      double s = 0;
      for (int k = i + 1; k < n; k++)
        s += getu(lu, i, k) * x[k][j];
      x[i][j] = (y[i][j] - s) / getu(lu, i, i);
    }
  }
  return x;
}

/*

Find the inverse A^-1 of a matrix A. The inverse of a matrix
satisfies A * A^-1 = I, where I is the identity matrix (for
all pairs (i, j), I[i][j] = 1 iff i = j, else I[i][j] = 0).
The inverse of a matrix exists if and only if det(a) is not 0.
We're lazy, so we just generate I and call solve_system().

Precondition: A is a square and det(A) != 0.

*/

vvd inverse(const vvd & a) {
  int n = a.size();
  assert(!a.empty() && n == (int)a[0].size());
  vvd I(n, vd(n));
  for (int i = 0; i < n; i++) I[i][i] = 1;
  return solve_system(a, I);
}

/*** Example Usage ***/

#include <cstdio>
#include <iostream>
using namespace std;

void print(const vvd & m) {
  cout << "[";
  for (int i = 0; i < (int)m.size(); i++) {
    cout << (i > 0 ? ",[" : "[");
    for (int j = 0; j < (int)m[0].size(); j++)
      cout << (j > 0 ? "," : "") << m[i][j];
    cout << "]";
  }
  cout << "]\n";
}

void printlu(const vvd & lu) {
  printf("L:\n");
  for (int i = 0; i < (int)lu.size(); i++) {
    for (int j = 0; j < (int)lu[0].size(); j++)
      printf("%10.5f␣", getl(lu, i, j));
    printf("\n");
  }
  printf("U:\n");
  for (int i = 0; i < (int)lu.size(); i++) {
    for (int j = 0; j < (int)lu[0].size(); j++)
      printf("%10.5f␣", getu(lu, i, j));
    printf("\n");
  }
}

int main() {
  { //determinant of 3x3
    const int n = 3;
    double a[n][n] = {{1,3,5},{2,4,7},{1,1,0}};
    vvd v(n);
    for (int i = 0; i < n; i++)
      v[i] = vector<double>(a[i], a[i] + n);
    printlu(lu_decompose(v));
    cout << "determinant:␣" << det(v) << "\n"; //4
  }

  { //determinant of 4x4
    const int n = 4;
    double a[n][n] = {{11,9,24,2},{1,5,2,6},{3,17,18,1},{2,5,7,1}};
    vvd v(n);
    for (int i = 0; i < n; i++)
      v[i] = vector<double>(a[i], a[i] + n);
    printlu(lu_decompose(v));
    cout << "determinant:␣" << det(v) << "\n"; //284
  }

  { //solve for [x, y] in x + 3y = 4 && 2x + 3y = 6
    const int n = 2;
    double a[n][n] = {{1,3},{2,3}};
    double b[n] = {4, 6};
    vvd va(n), vb(n);
    for (int i = 0; i < n; i++) {
      va[i] = vector<double>(a[i], a[i] + n);
      vb[i] = vector<double>(1, b[i]);
    }
    vvd x = solve_system(va, vb);
    for (int i = 0; i < n; i++) {
      assert(fabs(a[i][0]*x[0][0] + a[i][1]*x[1][0] - b[i]) < eps);
    }
  }

  { //find inverse by solving a system
    const int n = 2;
    double a[n][n] = {{2,3},{1,2}};
    vvd v(n);
    for (int i = 0; i < n; i++)
      v[i] = vector<double>(a[i], a[i] + n);
    print(inverse(v)); //[[2,-3],[-1,2]]
  }
  return 0;
}
```

4.5.5 Simplex Algorithm

```
/*

Description: The canonical form of a linear programming
problem is to maximize c^T*x, subject to Ax <= b, and x >= 0.
where x is the vector of variables (to be solved), c and b
are vectors of (known) coefficients, A is a (known) matrix of
coefficients, and (.)^T is the matrix transpose. The following
implementation solves n variables in a system of m constraints.

Precondition: ab has dimensions m by n+1 and c has length n+1.

Complexity: The simplex method is remarkably efficient in
practice, usually taking 2m or 3m iterations, converging in
expected polynomial time for certain distributions of random
inputs. However, its worst-case complexity is exponential,
and can be demonstrated with carefully constructed examples.

*/

#include <algorithm> /* std::swap() */
#include <cfloat>    /* DBL_MAX */
#include <cmath>     /* fabs() */
#include <vector>

typedef std::vector<double> vd;
typedef std::vector<vd> vvd;

//ab[i][0..n-1] stores A and ab[i][n] stores B
vd simplex(const vvd & ab, const vd & c, bool max = true) {
  const double eps = 1e-10;
  int n = c.size() - 1, m = ab.size();
  vvd ts(m + 2, vd(n + 2));
  ts[1][1] = max ? c[n] : -c[n];
  for (int j = 1; j <= n; j++)
    ts[1][j + 1] = max ? c[j - 1] : -c[j - 1];
  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++)
      ts[i + 1][j + 1] = -ab[i - 1][j - 1];
    ts[i + 1][1] = ab[i - 1][n];
  }
  for (int j = 1; j <= n; j++)
    ts[0][j + 1] = j;
  for (int i = n + 1; i <= n + m; i++)
```

```
    ts[i - n + 1][0] = i;
  double p1 = 0.0, p2 = 0.0;
  bool done = true;
  do {
    double mn = DBL_MAX, xmax = 0.0, v;
    for (int j = 2; j <= n + 1; j++)
      if (ts[1][j] > 0.0 && ts[1][j] > xmax) {
        p2 = j;
        xmax = ts[1][j];
      }
    for (int i = 2; i <= m + 1; i++) {
      v = fabs(ts[i][1] / ts[i][p2]);
      if (ts[i][p2] < 0.0 && mn > v) {
        mn = v;
        p1 = i;
      }
    }
    std::swap(ts[p1][0], ts[0][p2]);
    for (int i = 1; i <= m + 1; i++) {
      if (i == p1) continue;
      for (int j = 1; j <= n + 1; j++)
        if (j != p2)
          ts[i][j] -= ts[p1][j] * ts[i][p2] / ts[p1][p2];
    }
    ts[p1][p2] = 1.0 / ts[p1][p2];
    for (int j = 1; j <= n + 1; j++) {
      if (j != p2)
        ts[p1][j] *= fabs(ts[p1][p2]);
    }
    for (int i = 1; i <= m + 1; i++) {
      if (i != p1)
        ts[i][p2] *= ts[p1][p2];
    }
    for (int i = 2; i <= m + 1; i++)
      if (ts[i][1] < 0.0) return vd(); //no solution
    done = true;
    for (int j = 2; j <= n + 1; j++)
      if (ts[1][j] > 0) done = false;
  } while (!done);
  vd res;
  for (int i = 1; i <= n; i++)
    for (int j = 2; j <= m + 1; j++)
      if (fabs(ts[j][0] - i) <= eps)
        res.push_back(ts[j][1]);
  //the solution is stored in ts[1][1]
  return res;
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

/*
  Maximize 3x + 4y + 5, subject to x, y >= 0 and:
      -2x +    1y <=  0
       1x + 0.85y <=  9
       1x +    2y <= 14

  Note: The solution is 38.3043 at (5.30435, 4.34783).
*/

int main() {
  const int n = 2, m = 3;
  double ab[m][n + 1] = {{-2, 1, 0}, {1, 0.85, 9}, {1, 2, 14}};
  double c[n + 1] = {3, 4, 5};
  vvd vab(m, vd(n + 1));
  vd vc(c, c + n + 1);
  for (int i = 0; i < m; i++) {
    for (int j = 0; j <= n; j++)
      vab[i][j] = ab[i][j];
  }
  vd x = simplex(vab, vc);
  if (x.empty()) {
    cout << "No␣solution.\n";
  } else {
    double solval = c[n];
    for (int i = 0; i < (int)x.size(); i++)
      solval += c[i] * x[i];
    cout << "Solution␣=␣" << solval;
    cout << "␣at␣(" << x[0];
    for (int i = 1; i < (int)x.size(); i++)
      cout << ",␣" << x[i];
    cout << ").\n";
  }
  return 0;
}
```

4.6 Root-Finding

4.6.1 Real Root Finding (Differentiation)

```
/*

Real roots can be found via binary searching, a.k.a the bisection
method. If two x-coordinates evaluate to y-coordinates that have
opposite signs, a root must exist between them. For a polynomial
function, at most 1 root lies between adjacent local extrema.
Since local extrema exist where the derivative equals 0, we can
break root-finding into the subproblem of finding the roots of
the derivative. Recursively solve for local extrema until we get
to a base case of degree 0. For each set of local extrema found,
binary search between pairs of extrema for a root. This method is
easy, robust, and allows us to find the root to an arbitrary level
of accuracy. We're limited only by the precision of the arithmetic.

Complexity: For a degree N polynomial, repeatedly differentiating
it will take N + (N-1) + ... + 1 = O(N^2) operations. At each step
we binary search the number of times equal to the current degree.
If we want to make roots precise to eps=10^-P, each binary search
will take O(log P). Thus the overall complexity is O(N^2 log P).

*/

#include <cmath>   /* fabsl(), powl() */
```

```
#include <limits>  /* std::numeric_limits<>::quiet_NaN() */
#include <utility> /* std::pair<> */
#include <vector>

typedef long double Double;
typedef std::vector<std::pair<Double, int> > poly;

const Double epsa = 1e-11; //required precision of roots in absolute error
const Double epsr = 1e-15; //required precision of roots in relative error
const Double eps0 = 1e-17; //x is considered a root if fabs(eval(x))<=eps0
const Double inf = 1e20;   //[-inf, inf] is the range of roots to consider
const Double NaN = std::numeric_limits<Double>::quiet_NaN();

Double eval(const poly & p, Double x) {
  Double res = 0;
  for (int i = 0; i < (int)p.size(); i++)
    res += p[i].first * powl(x, p[i].second);
  return res;
}

Double find_root(const poly & p, Double x1, Double x2) {
  Double y1 = eval(p, x1), y2 = eval(p, x2);
  if (fabsl(y1) <= eps0) return x1;
  bool neg1 = (y1 < 0), neg2 = (y2 < 0);
  if (fabsl(y2) <= eps0 || neg1 == neg2) return NaN;
  while (x2 - x1 > epsa && x1 * (1 + epsr) < x2 && x2 * (1 + epsr) > x1) {
    Double x = (x1 + x2) / 2;
    ((eval(p, x) < 0) == neg1 ? x1 : x2) = x;
  }
  return x1;
}

std::vector<Double> find_all_roots(const poly & p) {
  poly dif;
  for (int i = 0; i < (int)p.size(); i++)
    if (p[i].second > 0)
      dif.push_back(std::make_pair(p[i].first * p[i].second, p[i].second - 1));
  if (dif.empty()) return std::vector<Double>();
  std::vector<Double> res, r = find_all_roots(dif);
  r.insert(r.begin(), -inf);
  r.push_back(inf);
  for (int i = 0; i < (int)r.size() - 1; i++) {
    Double root = find_root(p, r[i], r[i + 1]);
    if (root != root) continue; //NaN, not found
    if (res.empty() || root != res.back())
      res.push_back(root);
  }
  return res;
}

/*** Example Usage (http://wcipeg.com/problem/rootsolve) ***/

#include <iostream>
using namespace std;

int main() {
  int n, d;
  Double c;
  poly p;
  cin >> n;
  for (int i = 0; i < n; i++) {
    cin >> c >> d;
    p.push_back(make_pair(c, d));
  }
  vector<Double> sol = find_all_roots(p);
  if (sol.empty()) {
    cout << "NO␣REAL␣ROOTS\n";
  } else {
    cout.precision(9);
    for (int i = 0; i < (int)sol.size(); i++)
      cout << fixed << sol[i] << "\n";
  }
  return 0;
}
```

4.6.2 Complex Root Finding (Laguerre's)

```
/*

Laguerre's method can be used to not only find complex roots of
a polynomial, the polynomial may also have complex coefficients.
From extensive empirical study, Laguerre's method is observed to
be very close to being a "sure-fire" method, as it is almost
guaranteed to always converge to some root of the polynomial
regardless of what initial guess is chosen.

*/

#include <complex>
#include <cstdlib> /* rand(), RAND_MAX */
#include <vector>

typedef long double Double;
typedef std::complex<Double> cdouble;
typedef std::vector<cdouble> poly;

const Double eps = 1e-12;

std::pair<poly, cdouble> horner(const poly & a, const cdouble & x) {
  int n = a.size();
  poly b = poly(std::max(1, n - 1));
  for (int i = n - 1; i > 0; i--)
    b[i - 1] = a[i] + (i < n - 1 ? b[i] * x : 0);
  return std::make_pair(b, a[0] + b[0] * x);
}

cdouble eval(const poly & p, const cdouble & x) {
  return horner(p, x).second;
}

poly derivative(const poly & p) {
  int n = p.size();
  poly r(std::max(1, n - 1));
  for(int i = 1; i < n; i++)
    r[i - 1] = p[i] * cdouble(i);
  return r;
```

```
}

int comp(const cdouble & x, const cdouble & y) {
  Double diff = std::abs(x) - std::abs(y);
  return diff < -eps ? -1 : (diff > eps ? 1 : 0);
}

cdouble find_one_root(const poly & p, cdouble x) {
  int n = p.size() - 1;
  poly p1 = derivative(p), p2 = derivative(p1);
  for (int step = 0; step < 10000; step++) {
    cdouble y0 = eval(p, x);
    if (comp(y0, 0) == 0) break;
    cdouble G = eval(p1, x) / y0;
    cdouble H = G * G - eval(p2, x) / y0;
    cdouble R = std::sqrt(cdouble(n - 1) * (H * cdouble(n) - G * G));
    cdouble D1 = G + R, D2 = G - R;
    cdouble a = cdouble(n) / (comp(D1, D2) > 0 ? D1 : D2);
    x -= a;
    if (comp(a, 0) == 0) break;
  }
  return x;
}

std::vector<cdouble> find_all_roots(const poly & p) {
  std::vector<cdouble> res;
  poly q = p;
  while (q.size() > 2) {
    cdouble z(rand()/Double(RAND_MAX), rand()/Double(RAND_MAX));
    z = find_one_root(q, z);
    z = find_one_root(p, z);
    q = horner(q, z).first;
    res.push_back(z);
  }
  res.push_back(-q[0] / q[1]);
  return res;
}

/*** Example Usage ***/

#include <cstdio>
#include <iostream>
using namespace std;

void print_roots(vector<cdouble> roots) {
  for (int i = 0; i < (int)roots.size(); i++) {
    printf("(%9.5f, ", (double)roots[i].real());
    printf("%9.5f)\n", (double)roots[i].imag());
  }
}

int main() {
  { // x^3 - 8x^2 - 13x + 140 = (x + 4)(x - 5)(x - 7)
    printf("Roots of x^3 - 8x^2 - 13x + 140:\n");
    poly p;
    p.push_back(140);
    p.push_back(-13);
    p.push_back(-8);
    p.push_back(1);
    vector<cdouble> roots = find_all_roots(p);
    print_roots(roots);
  }

  { //(-6+4i)x^4 + (-26+12i)x^3 + (-30+40i)x^2 + (-26+12i)x + (-24+36i)
    // = ((2+3i)x + 6)*(x + i)*(2x + (6+4i))*(x*i + 1)
    printf("Roots␣of␣((2+3i)x␣+␣6)(x␣+␣i)(2x␣+␣(6+4i))(x*i␣+␣1):\n");
    poly p;
    p.push_back(cdouble(-24, 36));
    p.push_back(cdouble(-26, 12));
    p.push_back(cdouble(-30, 40));
    p.push_back(cdouble(-26, 12));
    p.push_back(cdouble(-6, 4));
    vector<cdouble> roots = find_all_roots(p);
    print_roots(roots);
  }
  return 0;
}
```

4.6.3 Complex Root Finding (RPOLY)

```
/*

Determine the complex roots of a polynomial with real coefficients.
This is the variant of the Jenkins-Traub algorithm for polynomials
with real coefficient, known as RPOLY. RPOLY follows follows the
same pattern as the CPOLY algorithm, but computes two roots at a
time, either two real roots or a pair of conjugate complex roots.
See: https://en.wikipedia.org/wiki/Jenkins%E2%80%93Traub_algorithm

The following is a translation of TOMS493 (www.netlib.org/toms/)
from FORTRAN to C++, with a simple wrapper at the end for the C++
<complex> class. Although the code is not meant to be read, it is
extremely efficient and robust, capable of achieving an accuracy
of at least 5 decimal places for even the most strenuous inputs.

*/

#include <cfloat> /* LDBL_EPSILON, LDBL_MAX, LDBL_MIN */
#include <cmath>  /* cosl, expl, fabsl, logl, powl, sinl, sqrtl */

typedef long double LD;

void divide_quadratic(int n, LD u, LD v, LD p[], LD q[], LD * a, LD * b) {
  q[0] = *b = p[0];
  q[1] = *a = -((*b) * u) + p[1];
  for (int i = 2; i < n; i++) {
    q[i] = -((*a) * u + (*b) * v) + p[i];
    *b = *a;
    *a = q[i];
  }
}

int get_flag(int n, LD a, LD b, LD * a1, LD * a3, LD * a7,
             LD * c, LD * d, LD * e, LD * f, LD * g, LD * h,
             LD k[], LD u, LD v, LD qk[]) {
  divide_quadratic(n, u, v, k, qk, c, d);
  if (fabsl(*c) <= 100.0 * LDBL_EPSILON * fabsl(k[n - 1]) &&
      fabsl(*d) <= 100.0 * LDBL_EPSILON * fabsl(k[n - 2])) return 3;
  *h = v * b;
  if (fabsl(*d) >= fabsl(*c)) {
    *e = a / (*d);
    *f = (*c) / (*d);
    *g = u * b;
    *a1 = (*f) * b - a;
    *a3 = (*e) * ((*g) + a) + (*h) * (b / (*d));
    *a7 = (*h) + ((*f) + u) * a;
    return 2;
  }
  *e = a / (*c);
  *f = (*d) / (*c);
  *g = (*e) * u;
  *a1 = -(a * ((*d) / (*c))) + b;
  *a3 = (*e) * a + ((*g) + (*h) / (*c)) * b;
  *a7 = (*g) * (*d) + (*h) * (*f) + a;
  return 1;
}

void find_polynomials(int n, int flag, LD a, LD b, LD a1, LD * a3,
                      LD * a7, LD k[], LD qk[], LD qp[]) {
  if (flag == 3) {
    k[1] = k[0] = 0.0;
    for (int i = 2; i < n; i++) k[i] = qk[i - 2];
    return;
  }
  if (fabsl(a1) > 10.0 * LDBL_EPSILON * fabsl(flag == 1 ? b : a)) {
    *a7 /= a1;
    *a3 /= a1;
    k[0] = qp[0];
    k[1] = qp[1] - (*a7) * qp[0];
    for (int i = 2; i < n; i++)
      k[i] = qp[i] - ((*a7) * qp[i - 1]) + (*a3) * qk[i - 2];
  } else {
    k[0] = 0.0;
    k[1] = -(*a7) * qp[0];
    for (int i = 2; i < n; i++)
      k[i] = (*a3) * qk[i - 2] - (*a7) * qp[i - 1];
  }
}

void estimate_coeff(int flag, LD * uu, LD * vv, LD a, LD a1, LD a3, LD a7,
                    LD b, LD c, LD d, LD f, LD g, LD h, LD u, LD v, LD k[],
                    int n, LD p[]) {
  LD a4, a5, b1, b2, c1, c2, c3, c4, temp;
  *vv = *uu = 0.0;
  if (flag == 3) return;
  if (flag != 2) {
    a4 = a + u * b + h * f;
    a5 = c + (u + v * f) * d;
  } else {
    a4 = (a + g) * f + h;
    a5 = (f + u) * c + v * d;
  }
  b1 = -k[n - 1] / p[n];
  b2 = -(k[n - 2] + b1 * p[n - 1]) / p[n];
  c1 = v * b2 * a1;
  c2 = b1 * a7;
  c3 = b1 * b1 * a3;
  c4 = c1 - c2 - c3;
  temp = b1 * a4 - c4 + a5;
  if (temp != 0.0) {
    *uu= u - (u * (c3 + c2) + v * (b1 * a1 + b2 * a7)) / temp;
    *vv = v * (1.0 + c4 / temp);
  }
}

void solve_quadratic(LD a, LD b1, LD c, LD * sr, LD * si, LD * lr, LD * li) {
  LD b, d, e;
  *sr = *si = *lr = *li = 0.0;
  if (a == 0) {
    *sr = (b1 != 0) ? -c / b1 : *sr;
    return;
  }
  if (c == 0) {
    *lr = -b1 / a;
    return;
  }
  b = b1 / 2.0;
  if (fabsl(b) < fabsl(c)) {
    e = (c >= 0) ? a : -a;
    e = b * (b / fabsl(c)) - e;
    d = sqrtl(fabsl(e)) * sqrtl(fabsl(c));
  } else {
    e = 1.0 - (a / b) * (c / b);
    d = sqrtl(fabsl(e)) * fabsl(b);
  }
  if (e >= 0) {
    d = (b >= 0) ? -d : d;
    *lr = (d - b) / a;
    *sr = (*lr != 0) ? (c / *lr / a) : *sr;
  } else {
    *lr = *sr = -b / a;
    *si = fabsl(d / a);
    *li = -(*si);
  }
}

void quadratic_iterate(int N, int * NZ, LD uu, LD vv,
                       LD * szr, LD * szi, LD * lzr, LD * lzi, LD qp[],
                       int n, LD * a, LD * b, LD p[], LD qk[],
                       LD * a1, LD * a3, LD * a7, LD * c, LD * d, LD * e,
                       LD * f, LD * g, LD * h, LD k[]) {
  int steps = 0, flag, tried_flag = 0;
  LD ee, mp, omp = 0.0, relstp = 0.0, t, u, ui, v, vi, zm;
  *NZ = 0;
  u = uu;
  v = vv;
  do {
    solve_quadratic(1.0, u, v, szr, szi, lzr, lzi);
    if (fabsl(fabsl(*szr) - fabsl(*lzr)) > 0.01 * fabsl(*lzr)) break;
    divide_quadratic(n, u, v, p, qp, a, b);
    mp = fabsl(-((*szr) * (*b)) + *a) + fabsl((*szi) * (*b));
    zm = sqrtl(fabsl(v));
    ee = 2.0 * fabsl(qp[0]);
    t = -(*szr) * (*b);
    for (int i = 1; i < N; i++) ee = ee * zm + fabsl(qp[i]);
    ee = ee * zm + fabsl(*a + t);
    ee = ee * 9.0 + 2.0 * fabsl(t) - 7.0 * (fabsl(*a + t) + zm * fabsl(*b));
    ee *= LDBL_EPSILON;
    if (mp <= 20.0 * ee) {
      *NZ = 2;
      break;
    }
    if (++steps > 20) break;
    if (steps >= 2 && relstp <= 0.01 && mp >= omp && !tried_flag) {
      relstp = (relstp < LDBL_EPSILON) ? sqrtl(LDBL_EPSILON) : sqrtl(relstp);
      u -= u * relstp;
      v += v * relstp;
      divide_quadratic(n, u, v, p, qp, a, b);
      for (int i = 0; i < 5; i++) {
        flag = get_flag(N, *a, *b, a1, a3, a7, c, d, e, f, g, h, k, u, v, qk);
        find_polynomials(N, flag, *a, *b, *a1, a3, a7, k, qk, qp);
      }
      tried_flag = 1;
      steps = 0;
    }
    omp = mp;
    flag = get_flag(N, *a, *b, a1, a3, a7, c, d, e, f, g, h, k, u, v, qk);
    find_polynomials(N, flag, *a, *b, *a1, a3, a7, k, qk, qp);
    flag = get_flag(N, *a, *b, a1, a3, a7, c, d, e, f, g, h, k, u, v, qk);
    estimate_coeff(flag, &ui, &vi, *a, *a1, *a3, *a7, *b, *c, *d, *f, *g, *h,
                   u, v, k, N, p);
    if (vi != 0) {
      relstp = fabsl((-v + vi) / vi);
      u = ui;
      v = vi;
    }
  } while (vi != 0);
}

void real_iterate(int * flag, int * nz, LD * sss, int n, LD p[],
                  int nn, LD qp[], LD * szr, LD * szi, LD k[], LD qk[]) {
  int steps = 0;
  LD ee, kv, mp, ms, omp = 0.0, pv, s, t = 0.0;
  *flag = *nz = 0;
  for (s = *sss; ; s += t) {
    pv = p[0];
    qp[0] = pv;
    for (int i = 1; i < nn; i++) qp[i] = pv = pv * s + p[i];
    mp = fabsl(pv);
    ms = fabsl(s);
    ee = 0.5 * fabsl(qp[0]);
    for (int i = 1; i < nn; i++) ee = ee * ms + fabsl(qp[i]);
    if (mp <= 20.0 * LDBL_EPSILON * (2.0 * ee - mp)) {
      *nz = 1;
      *szr = s;
      *szi = 0.0;
      break;
    }
    if (++steps > 10) break;
    if (steps >= 2 && fabsl(t) <= 0.001 * fabsl(s - t) && mp > omp) {
      *flag = 1;
      *sss = s;
      break;
    }
    omp = mp;
    qk[0] = kv = k[0];
    for (int i = 1; i < n; i++) qk[i] = kv = kv * s + k[i];
    if (fabsl(kv) > fabsl(k[n - 1]) * 10.0 * LDBL_EPSILON) {
      t = -pv / kv;
      k[0] = qp[0];
      for (int i = 1; i < n; i++)
        k[i] = t * qk[i - 1] + qp[i];
    } else {
      k[0] = 0.0;
      for (int i = 1; i < n; i++)
        k[i] = qk[i - 1];
    }
    kv = k[0];
    for (int i = 1; i < n; i++) kv = kv * s + k[i];
    t = fabsl(kv) > (fabsl(k[n - 1]) * 10.0 * LDBL_EPSILON) ? -pv / kv : 0.0;
  }
}

void solve_fixedshift(int l2, int * nz, LD sr, LD v, LD k[], int n,
                      LD p[], int nn, LD qp[], LD u, LD qk[], LD svk[],
                      LD * lzi, LD * lzr, LD * szi, LD * szr) {
  int flag, _flag, __flag = 1, spass, stry, vpass, vtry;
  LD a, a1, a3, a7, b, betas, betav, c, d, e, f, g, h;
  LD oss, ots = 0.0, otv = 0.0, ovv, s, ss, ts, tss, tv, tvv, ui, vi, vv;
  *nz = 0;
  betav = betas = 0.25;
  oss = sr;
  ovv = v;
  divide_quadratic(nn, u, v, p, qp, &a, &b);
  flag = get_flag(n, a, b, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h,
                  k, u, v, qk);
  for (int j = 0; j < l2; j++) {
    _flag = 1;
    find_polynomials(n, flag, a, b, a1, &a3, &a7, k, qk, qp);
    flag = get_flag(n, a, b, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h,
                    k, u, v, qk);
    estimate_coeff(flag, &ui, &vi, a, a1, a3, a7, b, c, d, f, g, h,
                   u, v, k, n, p);
    vv = vi;
    ss = k[n - 1] != 0.0 ? -p[n] / k[n - 1] : 0.0;
    ts = tv = 1.0;
    if (j != 0 && flag != 3) {
      tv = (vv != 0.0) ? fabsl((vv - ovv) / vv) : tv;
      ts = (ss != 0.0) ? fabsl((ss - oss) / ss) : ts;
      tvv = (tv < otv) ? tv * otv : 1.0;
      tss = (ts < ots) ? ts * ots : 1.0;
      vpass = (tvv < betav) ? 1 : 0;
      spass = (tss < betas) ? 1 : 0;
      if (spass || vpass) {
        for (int i = 0; i < n; i++) svk[i] = k[i];
        s = ss; stry = vtry = 0;
        for (;;) {
          if (!(_flag && spass && (!vpass || tss < tvv))) {
            quadratic_iterate(n, nz, ui, vi, szr, szi, lzr, lzi, qp, nn,
                  &a, &b, p, qk, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h, k);
            if (*nz > 0) return;
            __flag = vtry = 1;
            betav *= 0.25;
            if (stry || !spass) {
              __flag = 0;
            } else {
              for (int i = 0; i < n; i++) k[i] = svk[i];
            }
          }
          _flag = 0;
          if (__flag != 0) {
            real_iterate(&__flag, nz, &s, n, p, nn, qp, szr, szi, k, qk);
            if (*nz > 0) return;
            stry = 1;
            betas *= 0.25;
            if (__flag != 0) {
              ui = -(s + s);
              vi = s * s;
              continue;
            }
          }
          for (int i = 0; i < n; i++) k[i] = svk[i];
          if (!vpass || vtry) break;
        }
        divide_quadratic(nn, u, v, p, qp, &a, &b);
        flag = get_flag(n, a, b, &a1, &a3, &a7, &c, &d, &e, &f, &g, &h,
                        k, u, v, qk);
      }
    }
    ovv = vv;
    oss = ss;
    otv = tv;
    ots = ts;
  }
}

void find_roots(int degree, LD co[], LD re[], LD im[]) {
  int j, jj, n, nm1, nn, nz, zero, SZ = degree + 1;
  LD k[SZ], p[SZ], pt[SZ], qp[SZ], temp[SZ], qk[SZ], svk[SZ];
  LD bnd, df, dx, factor, ff, moduli_max, moduli_min, sc, x, xm;
  LD aa, bb, cc, lzi, lzr, sr, szi, szr, t, u, xx, xxx, yy;
  n = degree;
  xx = sqrtl(0.5);
  yy = -xx;
  for (j = 0; co[n] == 0; n--, j++) re[j] = im[j] = 0.0;
  nn = n + 1;
  for (int i = 0; i < nn; i++) p[i] = co[i];
  while (n >= 1) {
    if (n <= 2) {
      if (n < 2) {
        re[degree - 1] = -p[1] / p[0];
        im[degree - 1] = 0.0;
      } else {
        solve_quadratic(p[0], p[1], p[2], &re[degree - 2], &im[degree - 2],
                                          &re[degree - 1], &im[degree - 1]);
      }
      break;
    }
    moduli_max = 0.0;
    moduli_min = LDBL_MAX;
    for (int i = 0; i < nn; i++) {
      x = fabsl(p[i]);
      if (x > moduli_max) moduli_max = x;
      if (x != 0 && x < moduli_min) moduli_min = x;
    }
    sc = LDBL_MIN / LDBL_EPSILON / moduli_min;
    if ((sc <= 1.0 && moduli_max >= 10) ||
        (sc > 1.0 && LDBL_MAX / sc >= moduli_max)) {
      sc = (sc == 0) ? LDBL_MIN : sc;
      factor = powl(2.0, logl(sc) / logl(2.0));
      if (factor != 1.0)
        for (int i = 0; i < nn; i++) p[i] *= factor;
    }
    for (int i = 0; i < nn; i++) pt[i] = fabsl(p[i]);
    pt[n] = -pt[n];
    nm1 = n - 1;
    x = expl((logl(-pt[n]) - logl(pt[0])) / (LD)n);
    if (pt[nm1] != 0) {
      xm = -pt[n] / pt[nm1];
      if (xm < x) x = xm;
    }
    xm = x;
    do {
      x = xm;
      xm = 0.1 * x;
      ff = pt[0];
      for (int i = 1; i < nn; i++) ff = ff * xm + pt[i];
    } while (ff > 0);
    dx = x;
    do {
      df = ff = pt[0];
      for (int i = 1; i < n; i++) {
        ff = x * ff + pt[i];
        df = x * df + ff;
      }
      ff = x * ff + pt[n];
      dx = ff / df;
      x -= dx;
    } while (fabsl(dx / x) > 0.005);
    bnd = x;
    for (int i = 1; i < n; i++)
      k[i] = (LD)(n - i) * p[i] / (LD)n;
    k[0] = p[0];
    aa = p[n];
    bb = p[nm1];
    zero = (k[nm1] == 0) ? 1 : 0;
    for (jj = 0; jj < 5; jj++) {
      cc = k[nm1];
      if (zero) {
        for (int i = 0; i < nm1; i++) {
          j = nm1 - i;
          k[j] = k[j - 1];
        }
        k[0] = 0;
        zero = (k[nm1] == 0) ? 1 : 0;
      } else {
        t = -aa / cc;
        for (int i = 0; i < nm1; i++) {
          j = nm1 - i;
          k[j] = t * k[j - 1] + p[j];
        }
        k[0] = p[0];
        zero = (fabsl(k[nm1]) <= fabsl(bb) * LDBL_EPSILON * 10.0) ? 1 : 0;
      }
    }
    for (int i = 0; i < n; i++) temp[i] = k[i];
    static const LD DEG = 0.01745329251994329576923690768489L;
    for (jj = 1; jj <= 20; jj++) {
      xxx = -sinl(94.0 * DEG) * yy + cosl(94.0 * DEG) * xx;
      yy = sinl(94.0 * DEG) * xx + cosl(94.0 * DEG) * yy;
      xx = xxx;
      sr = bnd * xx;
      u = -2.0 * sr;
      for (int i = 0; i < nn; i++) qk[i] = svk[i] = 0.0;
      solve_fixedshift(20 * jj, &nz, sr, bnd, k, n, p, nn, qp, u,
                       qk, svk, &lzi, &lzr, &szi, &szr);
      if (nz != 0) {
        j = degree - n;
        re[j] = szr;
        im[j] = szi;
        nn = nn - nz;
        n = nn - 1;
        for (int i = 0; i < nn; i++) p[i] = qp[i];
        if (nz != 1) {
          re[j + 1] = lzr;
          im[j + 1] = lzi;
        }
        break;
      } else {
        for (int i = 0; i < n; i++) k[i] = temp[i];
      }
    }
    if (jj > 20) break;
  }
  }

  /*** Wrapper ***/

  #include <algorithm> /* std::reverse(), std::sort() */
  #include <complex>
  #include <vector>

  typedef std::complex<LD> root;

  bool comp(const root & a, const root & b) {
    if (real(a) != real(b)) return real(a) < real(b);
    return imag(a) < imag(b);
  }

  std::vector<root> find_roots(int degree, LD coefficients[]) {
    std::reverse(coefficients, coefficients + degree + 1);
    LD re[degree], im[degree];
    find_roots(degree, coefficients, re, im);
    std::vector<root> res;
    for (int i = 0; i < degree; i++)
    res.push_back(root(re[i], im[i]));
    std::sort(res.begin(), res.end(), comp);
    return res;
  }
```

```

/*** Example Usage (http://wcipeg.com/problem/rootsolve) ***/

#include <iostream>
using namespace std;

int T, degree, p;
LD c, coeff[101];

int main() {
  degree = 0;
  cin >> T;
  for (int i = 0; i < T; i++) {
    cin >> c >> p;
    if (p > degree) degree = p;
    coeff[p] = c;
  }
  std::vector<root> roots = find_roots(degree, coeff);
  bool printed = false;
  cout.precision(6);
  for (int i = 0; i < (int)roots.size(); i++) {
    if (fabsl(roots[i].imag()) < LDBL_EPSILON) {
      cout << fixed << roots[i].real() << "\n";
      printed = true;
    }
  }
  if (!printed) cout << "NO␣REAL␣ROOTS\n";
  return 0;
}
```

4.7 Integration

4.7.1 Simpson's Rule

```
/*

Simpson's rule is a method for numerical integration, the
numerical approximation of definite integrals. The rule is:

Integral of f(x) dx from a to b ~=
  [f(a) + 4*f((a + b)/2) + f(b)] * (b - a)/6

*/

#include <cmath> /* fabs() */

template<class DoubleFunction>
double simpsons(DoubleFunction f, double a, double b) {
  return (f(a) + 4 * f((a + b)/2) + f(b)) * (b - a)/6;
}

template<class DoubleFunction>
double integrate(DoubleFunction f, double a, double b) {
  static const double eps = 1e-10;
  double m = (a + b) / 2;
  double am = simpsons(f, a, m);
  double mb = simpsons(f, m, b);
  double ab = simpsons(f, a, b);
  if (fabs(am + mb - ab) < eps) return ab;
  return integrate(f, a, m) + integrate(f, m, b);
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

double f(double x) { return sin(x); }

int main () {
  double PI = acos(-1.0);
  cout << integrate(f, 0.0, PI/2) << "\n"; //1
  return 0;
}
```

Chapter 5

Geometry

5.1 Geometric Classes

5.1.1 Point

```
/*

This class is very similar to std::complex, except it uses epsilon
comparisons and also supports other operations such as reflection
and rotation. In addition, this class supports many arithmetic
operations (e.g. overloaded operators for vector addition, subtraction,
multiplication, and division; dot/cross products, etc.) pertaining to
2D cartesian vectors.

All operations are O(1) in time and space.

*/

#include <cmath>   /* atan(), fabs(), sqrt() */
#include <ostream>
#include <utility> /* std::pair */

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */

struct point {

  double x, y;

  point() : x(0), y(0) {}
  point(const point & p) : x(p.x), y(p.y) {}
  point(const std::pair<double, double> & p) : x(p.first), y(p.second) {}
  point(const double & a, const double & b) : x(a), y(b) {}

  bool operator < (const point & p) const {
    return EQ(x, p.x) ? LT(y, p.y) : LT(x, p.x);
  }

  bool operator > (const point & p) const {
    return EQ(x, p.x) ? LT(p.y, y) : LT(p.x, x);
  }

  bool operator == (const point & p) const { return EQ(x, p.x) && EQ(y, p.y); }
  bool operator != (const point & p) const { return !(*this == p); }
  bool operator <= (const point & p) const { return !(*this > p); }
  bool operator >= (const point & p) const { return !(*this < p); }
  point operator + (const point & p) const { return point(x + p.x, y + p.y); }
  point operator - (const point & p) const { return point(x - p.x, y - p.y); }
  point operator + (const double & v) const { return point(x + v, y + v); }
  point operator - (const double & v) const { return point(x - v, y - v); }
  point operator * (const double & v) const { return point(x * v, y * v); }
  point operator / (const double & v) const { return point(x / v, y / v); }
  point & operator += (const point & p) { x += p.x; y += p.y; return *this; }
  point & operator -= (const point & p) { x -= p.x; y -= p.y; return *this; }
  point & operator += (const double & v) { x += v; y += v; return *this; }
  point & operator -= (const double & v) { x -= v; y -= v; return *this; }
  point & operator *= (const double & v) { x *= v; y *= v; return *this; }
  point & operator /= (const double & v) { x /= v; y /= v; return *this; }
  friend point operator + (const double & v, const point & p) { return p + v; }
  friend point operator * (const double & v, const point & p) { return p * v; }

  double norm() const { return x * x + y * y; }
  double abs() const { return sqrt(x * x + y * y); }
  double arg() const { return atan2(y, x); }
  double dot(const point & p) const { return x * p.x + y * p.y; }
  double cross(const point & p) const { return x * p.y - y * p.x; }
  double proj(const point & p) const { return dot(p) / p.abs(); } //onto p
  point rot90() const { return point(-y, x); }

  //proportional unit vector of (x, y) such that x^2 + y^2 = 1
  point normalize() const {
    return (EQ(x, 0) && EQ(y, 0)) ? point(0, 0) : (point(x, y) / abs());
  }

  //rotate t radians CW about origin
  point rotateCW(const double & t) const {
    return point(x * cos(t) + y * sin(t), y * cos(t) - x * sin(t));
  }

  //rotate t radians CCW about origin
  point rotateCCW(const double & t) const {
    return point(x * cos(t) - y * sin(t), x * sin(t) + y * cos(t));
  }

  //rotate t radians CW about point p
  point rotateCW(const point & p, const double & t) const {
    return (*this - p).rotateCW(t) + p;
  }

  //rotate t radians CCW about point p
  point rotateCCW(const point & p, const double & t) const {
    return (*this - p).rotateCCW(t) + p;
  }

  //reflect across point p
  point reflect(const point & p) const {
    return point(2 * p.x - x, 2 * p.y - y);
  }

  //reflect across the line containing points p and q
  point reflect(const point & p, const point & q) const {
    if (p == q) return reflect(p);
    point r(*this - p), s = q - p;
    r = point(r.x * s.x + r.y * s.y, r.x * s.y - r.y * s.x) / s.norm();
    r = point(r.x * s.x - r.y * s.y, r.x * s.y + r.y * s.x) + p;
    return r;
  }

  friend double norm(const point & p) { return p.norm(); }
  friend double abs(const point & p) { return p.abs(); }
  friend double arg(const point & p) { return p.arg(); }
  friend double dot(const point & p, const point & q) { return p.dot(q); }
  friend double cross(const point & p, const point & q) { return p.cross(q); }
  friend double proj(const point & p, const point & q) { return p.proj(q); }
  friend point rot90(const point & p) { return p.rot90(); }
  friend point normalize(const point & p) { return p.normalize(); }
  friend point rotateCW(const point & p, const double & t) { return p.rotateCW(t); }
  friend point rotateCCW(const point & p, const double & t) { return p.rotateCCW(t); }
  friend point rotateCW(const point & p, const point & q, const double & t) { return p.rotateCW(q, t); }
  friend point rotateCCW(const point & p, const point & q, const double & t) { return p.rotateCCW(q, t);
    }
  friend point reflect(const point & p, const point & q) { return p.reflect(q); }
  friend point reflect(const point & p, const point & a, const point & b) { return p.reflect(a, b); }

  friend std::ostream & operator << (std::ostream & out, const point & p) {
    out << "(";
    out << (fabs(p.x) < eps ? 0 : p.x) << ",";
    out << (fabs(p.y) < eps ? 0 : p.y) << ")";
    return out;
  }
};

/*** Example Usage ***/

#include <cassert>
#define pt point

const double PI = acos(-1.0);

int main() {
  pt p(-10, 3);
  assert(pt(-18, 29) == p + pt(-3, 9) * 6 / 2 - pt(-1, 1));
  assert(EQ(109, p.norm()));
  assert(EQ(10.44030650891, p.abs()));
  assert(EQ(2.850135859112, p.arg()));
  assert(EQ(0,  p.dot(pt(3, 10))));
  assert(EQ(0,  p.cross(pt(10, -3))));
  assert(EQ(10, p.proj(pt(-10, 0))));
  assert(EQ(1,  p.normalize().abs()));
  assert(pt(-3, -10) == p.rot90());
  assert(pt(3, 12)   == p.rotateCW(pt(1, 1), PI / 2));
  assert(pt(1, -10)  == p.rotateCCW(pt(2, 2), PI / 2));
  assert(pt(10, -3)  == p.reflect(pt(0, 0)));
  assert(pt(-10, -3) == p.reflect(pt(-2, 0), pt(5, 0)));
  return 0;
}
```

5.1.2 Line

```
/*

A 2D line is expressed in the form Ax + By + C = 0. All lines can be
"normalized" to a canonical form by insisting that the y-coefficient
equal 1 if it is non-zero. Otherwise, we set the x-coefficient to 1.
If B is non-zero, then we have the common case where the slope = -A
after normalization.

All operations are O(1) in time and space.

*/

#include <cmath>   /* fabs() */
#include <limits>  /* std::numeric_limits */
#include <ostream>
#include <utility> /* std::pair */

const double eps = 1e-9, NaN = std::numeric_limits<double>::quiet_NaN();

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */

typedef std::pair<double, double> point;
#define x first
#define y second

struct line {

  double a, b, c;

  line(): a(0), b(0), c(0) {} //invalid or uninitialized line

  line(const double & A, const double & B, const double & C) {
    a = A;
    b = B;
    c = C;
    if (!EQ(b, 0)) {
      a /= b; c /= b; b = 1;
    } else {
      c /= a; a = 1; b = 0;
    }
  }

  line(const double & slope, const point & p) {
    a = -slope;
    b = 1;
    c = slope * p.x - p.y;
  }

  line(const point & p, const point & q): a(0), b(0), c(0) {
    if (EQ(p.x, q.x)) {
      if (EQ(p.y, q.y)) return; //invalid line
      //vertical line
      a = 1;
      b = 0;
      c = -p.x;
      return;
    }
    a = -(p.y - q.y) / (p.x - q.x);
    b = 1;
    c = -(a * p.x) - (b * p.y);
  }

  bool operator == (const line & l) const {
    return EQ(a, l.a) && EQ(b, l.b) && EQ(c, l.c);
  }

  bool operator != (const line & l) const {
    return !(*this == l);
  }

  //whether the line is initialized and normalized
  bool valid() const {
    if (EQ(a, 0)) return !EQ(b, 0);
    return EQ(b, 1) || (EQ(b, 0) && EQ(a, 1));
  }

  bool horizontal() const { return valid() && EQ(a, 0); }
  bool vertical() const { return valid() && EQ(b, 0); }

  double slope() const {
    if (!valid() || EQ(b, 0)) return NaN; //vertical
    return -a;
  }

  //solve for x, given y
  //for horizontal lines, either +inf, -inf, or nan is returned
  double x(const double & y) const {
    if (!valid() || EQ(a, 0)) return NaN; //invalid or horizontal
    return (-c - b * y) / a;
  }

  //solve for y, given x
  //for vertical lines, either +inf, -inf, or nan is returned
  double y(const double & x) const {
    if (!valid() || EQ(b, 0)) return NaN; //invalid or vertical
    return (-c - a * x) / b;
  }

  //returns whether p exists on the line
  bool contains(const point & p) const {
    return EQ(a * p.x + b * p.y + c, 0);
  }

  //returns whether the line is parallel to l
  bool parallel(const line & l) const {
    return EQ(a, l.a) && EQ(b, l.b);
  }

  //returns whether the line is perpendicular to l
  bool perpendicular(const line & l) const {
    return EQ(-a * l.a, b * l.b);
  }

  //return the parallel line passing through point p
  line parallel(const point & p) const {
    return line(a, b, -a * p.x - b * p.y);
  }

  //return the perpendicular line passing through point p
  line perpendicular(const point & p) const {
    return line(-b, a, b * p.x - a * p.y);
  }

  friend std::ostream & operator << (std::ostream & out, const line & l) {
    out << (fabs(l.a) < eps ? 0 : l.a) << "x" << std::showpos;
    out << (fabs(l.b) < eps ? 0 : l.b) << "y";
    out << (fabs(l.c) < eps ? 0 : l.c) << "=0" << std::noshowpos;
    return out;
  }
};

/*** Example Usage ***/

#include <cassert>

int main() {
  line l(2, -5, -8);
  line para = line(2, -5, -8).parallel(point(-6, -2));
  line perp = line(2, -5, -8).perpendicular(point(-6, -2));
  assert(l.parallel(para) && l.perpendicular(perp));
  assert(l.slope() == 0.4);
  assert(para == line(-0.4, 1, -0.4)); //-0.4x+1y-0.4=0
  assert(perp == line(2.5, 1, 17));    //2.5x+1y+17=0
  return 0;
}
```

5.1.3 Circle

```
/*

A 2D circle with center at (h, k) and a radius of r can be expressed by
the relation (x - h)^2 + (y - k)^2 = r^2. In the following definition,
the radius used to construct it is forced to be a positive number.

All operations are O(1) in time and space.

*/

#include <cmath>     /* fabs(), sqrt() */
#include <ostream>
#include <stdexcept> /* std::runtime_error() */
#include <utility>   /* std::pair */

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define GT(a, b) ((a) > (b) + eps)        /* greater than */
#define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */

typedef std::pair<double, double> point;
#define x first
```

```
#define y second

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double abs(const point & a) { return sqrt(norm(a)); }

struct circle {

  double h, k, r;

  circle(): h(0), k(0), r(0) {}
  circle(const double & R): h(0), k(0), r(fabs(R)) {}
  circle(const point & o, const double & R): h(o.x), k(o.y), r(fabs(R)) {}
  circle(const double & H, const double & K, const double & R):
    h(H), k(K), r(fabs(R)) {}

  //circumcircle with the diameter equal to the distance from a to b
  circle(const point & a, const point & b) {
    h = (a.x + b.x) / 2.0;
    k = (a.y + b.y) / 2.0;
    r = abs(point(a.x - h, a.y - k));
  }

  //circumcircle of 3 points - throws exception if abc are collinear/equal
  circle(const point & a, const point & b, const point & c) {
    double an = norm(point(b.x - c.x, b.y - c.y));
    double bn = norm(point(a.x - c.x, a.y - c.y));
    double cn = norm(point(a.x - b.x, a.y - b.y));
    double wa = an * (bn + cn - an);
    double wb = bn * (an + cn - bn);
    double wc = cn * (an + bn - cn);
    double w = wa + wb + wc;
    if (fabs(w) < eps)
      throw std::runtime_error("No␣circle␣from␣collinear␣points.");
    h = (wa * a.x + wb * b.x + wc * c.x) / w;
    k = (wa * a.y + wb * b.y + wc * c.y) / w;
    r = abs(point(a.x - h, a.y - k));
  }

  //circle from 2 points and a radius - many possible edge cases!
  //in the "normal" case, there will be 2 possible circles, one
  //centered at (h1, k1) and the other (h2, k2). Only one is used.
  //note that (h1, k1) equals (h2, k2) if dist(a, b) = 2 * r = d
  circle(const point & a, const point & b, const double & R) {
    r = fabs(R);
    if (LE(r, 0) && a == b) { //circle is a point
      h = a.x;
      k = a.y;
      return;
    }
    double d = abs(point(b.x - a.x, b.y - a.y));
    if (EQ(d, 0))
      throw std::runtime_error("Identical␣points,␣infinite␣circles.");
    if (GT(d, r * 2.0))
      throw std::runtime_error("Points␣too␣far␣away␣to␣make␣circle.");
    double v = sqrt(r * r - d * d / 4.0) / d;
    point m((a.x + b.x) / 2.0, (a.y + b.y) / 2.0);
    h = m.x + (a.y - b.y) * v;
    k = m.y + (b.x - a.x) * v;
    //other answer is (h, k) = (m.x-(a.y-b.y)*v, m.y-(b.x-a.x)*v)
  }

  bool operator == (const circle & c) const {
    return EQ(h, c.h) && EQ(k, c.k) && EQ(r, c.r);
  }

  bool operator != (const circle & c) const {
    return !(*this == c);
  }

  bool contains(const point & p) const {
    return LE(norm(point(p.x - h, p.y - k)), r * r);
  }

  bool on_edge(const point & p) const {
    return EQ(norm(point(p.x - h, p.y - k)), r * r);
  }

  point center() const {
    return point(h, k);
  }

  friend std::ostream & operator << (std::ostream & out, const circle & c) {
    out << std::showpos;
    out << "(x" << -(fabs(c.h) < eps ? 0 : c.h) << ")^2+";
    out << "(y" << -(fabs(c.k) < eps ? 0 : c.k) << ")^2";
    out << std::noshowpos;
    out << "=" << (fabs(c.r) < eps ? 0 : c.r * c.r);
    return out;
  }
};

//circle inscribed within points a, b, and c
circle incircle(const point & a, const point & b, const point & c) {
  double al = abs(point(b.x - c.x, b.y - c.y));
  double bl = abs(point(a.x - c.x, a.y - c.y));
  double cl = abs(point(a.x - b.x, a.y - b.y));
  double p = al + bl + cl;
  if (EQ(p, 0)) return circle(a.x, a.y, 0);
  circle res;
  res.h = (al * a.x + bl * b.x + cl * c.x) / p;
  res.k = (al * a.y + bl * b.y + cl * c.y) / p;
  res.r = fabs((a.x - c.x) * (b.y - c.y) - (a.y - c.y) * (b.x - c.x)) / p;
  return res;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  circle c(-2, 5, sqrt(10)); //(x+2)^2+(y-5)^2=10
  assert(c == circle(point(-2, 5), sqrt(10)));
  assert(c == circle(point(1, 6), point(-5, 4)));
  assert(c == circle(point(-3, 2), point(-3, 8), point(-1, 8)));
  assert(c == incircle(point(-12, 5), point(3, 0), point(0, 9)));
  assert(c.contains(point(-2, 8)) && !c.contains(point(-2, 9)));
  assert(c.on_edge(point(-1, 2)) && !c.on_edge(point(-1.01, 2)));
  return 0;
```

```
}
```

5.2 Geometric Calculations

5.2.1 Angles

```
/*

Angle calculations in 2 dimensions. All returned angles are in radians,
except for reduce_deg(). If x is an angle in radians, then you may use
x * DEG to convert x to degrees, and vice versa to radians with x * RAD.

All operations are O(1) in time and space.

*/

#include <cmath>     /* acos(), fabs(), sqrt(), atan2() */
#include <utility>   /* std::pair */

typedef std::pair<double, double> point;
#define x first
#define y second

const double PI = acos(-1.0), RAD = 180 / PI, DEG = PI / 180;

double abs(const point & a) { return sqrt(a.x * a.x + a.y * a.y); }

//reduce angles to the range [0, 360) degrees. e.g. reduce_deg(-630) = 90
double reduce_deg(const double & t) {
  if (t < -360) return reduce_deg(fmod(t, 360));
  if (t < 0) return t + 360;
  return t >= 360 ? fmod(t, 360) : t;
}

//reduce angles to the range [0, 2*pi) radians. e.g. reduce_rad(720.5) = 0.5
double reduce_rad(const double & t) {
  if (t < -2 * PI) return reduce_rad(fmod(t, 2 * PI));
  if (t < 0) return t + 2 * PI;
  return t >= 2 * PI ? fmod(t, 2 * PI) : t;
}

//like std::polar(), but returns a point instead of an std::complex
point polar_point(const double & r, const double & theta) {
  return point(r * cos(theta), r * sin(theta));
}

//angle of segment (0, 0) to p, relative (CCW) to the +'ve x-axis in radians
double polar_angle(const point & p) {
  double t = atan2(p.y, p.x);
  return t < 0 ? t + 2 * PI : t;
}

//smallest angle formed by points aob (angle is at point o) in radians
double angle(const point & a, const point & o, const point & b) {
  point u(o.x - a.x, o.y - a.y), v(o.x - b.x, o.y - b.y);
  return acos((u.x * v.x + u.y * v.y) / (abs(u) * abs(v)));
}

//angle of line segment ab relative (CCW) to the +'ve x-axis in radians
double angle_between(const point & a, const point & b) {
  double t = atan2(a.x * b.y - a.y * b.x, a.x * b.x + a.y * b.y);
  return t < 0 ? t + 2 * PI : t;
}

//Given the A, B values of two lines in Ax + By + C = 0 form, finds the
//minimum angle in radians between the two lines in the range [0, PI/2]
double angle_between(const double & a1, const double & b1,
                     const double & a2, const double & b2) {
  double t = atan2(a1 * b2 - a2 * b1, a1 * a2 + b1 * b2);
  if (t < 0) t += PI; //force angle to be positive
  if (t > PI / 2) t = PI - t; //force angle to be <= 90 degrees
  return t;
}

//magnitude of the 3D cross product with Z component implicitly equal to 0
//the answer assumes the origin (0, 0) is instead shifted to point o.
//this is equal to 2x the signed area of the triangle from these 3 points.
double cross(const point & o, const point & a, const point & b) {
  return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

//does the path a->o->b form:
// -1 ==> a left turn on the plane?
//  0 ==> a single straight line segment? (i.e. are a,o,b collinear?) or
// +1 ==> a right turn on the plane?
//warning: the order of parameters is a,o,b, and NOT o,a,b as in cross()
int turn(const point & a, const point & o, const point & b) {
  double c = cross(o, a, b);
  return c < 0 ? -1 : (c > 0 ? 1 : 0);
}

/*** Example Usage ***/

#include <cassert>
#define pt point
#define EQ(a, b) (fabs((a) - (b)) <= 1e-9)

int main() {
  assert(EQ(123,    reduce_deg(-(8 * 360) + 123)));
  assert(EQ(1.2345, reduce_rad(2 * PI * 8 + 1.2345)));
  point p = polar_point(4, PI), q = polar_point(4, -PI / 2);
  assert(EQ(p.x, -4) && EQ(p.y, 0));
  assert(EQ(q.x, 0) && EQ(q.y, -4));
  assert(EQ(45,  polar_angle(pt(5, 5)) * RAD));
  assert(EQ(135, polar_angle(pt(-4, 4)) * RAD));
  assert(EQ(90,  angle(pt(5, 0), pt(0, 5), pt(-5, 0)) * RAD));
  assert(EQ(225, angle_between(pt(0, 5), pt(5, -5)) * RAD));
  assert(EQ(90,  angle_between(-1, 1, -1, -1) * RAD)); //y=x and y=-x
  assert(-1 == cross(pt(0, 0), pt(0, 1), pt(1, 0)));
  assert(+1 == turn(pt(0, 1), pt(0, 0), pt(-5, -5)));
  return 0;
}
```

5.2.2 Distances

```
/*

Distance calculations in 2 dimensions between points, lines, and segments.
All operations are O(1) in time and space.

*/

#include <algorithm> /* std::max(), std::min() */
#include <cmath>     /* fabs(), sqrt() */
#include <utility>   /* std::pair */

typedef std::pair<double, double> point;
#define x first
#define y second

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps)  /* equal to */
#define LE(a, b) ((a) <= (b) + eps)        /* less than or equal to */
#define GE(a, b) ((a) >= (b) - eps)        /* greater than or equal to */

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double abs(const point & a) { return sqrt(norm(a)); }

//distance from point a to point b
double dist(const point & a, const point & b) {
  return abs(point(b.x - a.x, b.y - a.y));
}

//squared distance from point a to point b
double dist2(const point & a, const point & b) {
  return norm(point(b.x - a.x, b.y - a.y));
}

//minimum distance from point p to line l denoted by ax + by + c = 0
//if a = b = 0, then -inf, nan, or +inf is returned depending on sgn(c)
double dist_line(const point & p,
                 const double & a, const double & b, const double & c) {
  return fabs(a * p.x + b * p.y + c) / sqrt(a * a + b * b);
}

//minimum distance from point p to the infinite line containing a and b
//if a = b, then the point distance from p to the single point is returned
double dist_line(const point & p, const point & a, const point & b) {
  double ab2 = dist2(a, b);
  if (EQ(ab2, 0)) return dist(p, a);
  double u = ((p.x - a.x) * (b.x - a.x) + (p.y - a.y) * (b.y - a.y)) / ab2;
  return abs(point(a.x + u * (b.x - a.x) - p.x, a.y + u * (b.y - a.y) - p.y));
}

//distance between two lines each denoted by the form ax + by + c = 0
//if the lines are nonparallel, then the distance is 0, otherwise
//it is the perpendicular distance from a point on one line to the other
double dist_lines(const double & a1, const double & b1, const double & c1,
                  const double & a2, const double & b2, const double & c2) {
  if (EQ(a1 * b2, a2 * b1)) {
    double factor = EQ(b1, 0) ? (a1 / a2) : (b1 / b2);
    if (EQ(c1, c2 * factor)) return 0;
    return fabs(c2 * factor - c1) / sqrt(a1 * a1 + b1 * b1);
  }
  return 0;
}

//distance between two infinite lines respectively containing ab and cd
//same results as above, except we solve for the lines here first.
double dist_lines(const point & a, const point & b,
                  const point & c, const point & d) {
  double A1 = a.y - b.y, B1 = b.x - a.x;
  double A2 = c.y - d.y, B2 = d.x - c.x;
  double C1 = -A1 * a.x - B1 * a.y, C2 = -A2 * c.x - B2 * c.y;
  return dist_lines(A1, B1, C1, A2, B2, C2);
}

//minimum distance from point p to any point on segment ab
double dist_seg(const point & p, const point & a, const point & b) {
  if (a == b) return dist(p, a);
  point ab(b.x - a.x, b.y - a.y), ap(p.x - a.x, p.y - a.y);
  double n = norm(ab), d = ab.x * ap.x + ab.y * ap.y;
  if (LE(d, 0) || EQ(n, 0)) return abs(ap);
  if (GE(d, n)) return abs(point(ap.x - ab.x, ap.y - ab.y));
  return abs(point(ap.x - ab.x * (d / n), ap.y - ab.y * (d / n)));
}

double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }

//minimum distance from any point on segment ab to any point on segment cd
double dist_segs(const point & a, const point & b,
                 const point & c, const point & d) {
  //check if segments are touching or intersecting - if so, distance is 0
  point ab(b.x - a.x, b.y - a.y);
  point ac(c.x - a.x, c.y - a.y);
  point cd(d.x - c.x, d.y - c.y);
  double c1 = cross(ab, cd), c2 = cross(ac, ab);
  if (EQ(c1, 0) && EQ(c2, 0)) {
    double t0 = dot(ac, ab) / norm(ab);
    double t1 = t0 + dot(cd, ab) / norm(ab);
    if (LE(std::min(t0, t1), 1) && LE(0, std::max(t0, t1)))
      return 0;
  } else {
    double t = cross(ac, cd) / c1, u = c2 / c1;
    if (!EQ(c1, 0) && LE(0, t) && LE(t, 1) && LE(0, u) && LE(u, 1))
      return 0;
  }
  //find min distances across each endpoint to opposing segment
  return std::min(std::min(dist_seg(a, c, d), dist_seg(b, c, d)),
                  std::min(dist_seg(c, a, b), dist_seg(d, a, b)));
}

/*** Example Usage ***/

#include <cassert>
#define pt point

int main() {
  assert(EQ(5, dist(pt(-1, -1), pt(2, 3))));
  assert(EQ(25, dist2(pt(-1, -1), pt(2, 3))));
  assert(EQ(1.2, dist_line(pt(2, 1), -4, 3, -1)));
  assert(EQ(0.8, dist_line(pt(3, 3), pt(-1, -1), pt(2, 3))));
  assert(EQ(1.2, dist_line(pt(2, 1), pt(-1, -1), pt(2, 3))));
  assert(EQ(0.0, dist_lines(-4, 3, -1, 8, 6, 2)));
  assert(EQ(0.8, dist_lines(-4, 3, -1, -8, 6, -10)));
  assert(EQ(1.0, dist_seg(pt(3, 3), pt(-1, -1), pt(2, 3))));
  assert(EQ(1.2, dist_seg(pt(2, 1), pt(-1, -1), pt(2, 3))));
  assert(EQ(0.0, dist_segs(pt(0, 2), pt(3, 3), pt(-1, -1), pt(2, 3))));
  assert(EQ(0.6, dist_segs(pt(-1, 0), pt(-2, 2), pt(-1, -1), pt(2, 3))));
  return 0;
}
```

5.2.3 Line Intersections

```
/*

Intersections between straight lines, as well as between line segments
in 2 dimensions. Also included are functions to determine the closest
point to a line, which is done by finding the intersection through the
perpendicular. Note that you should modify the TOUCH_IS_INTERSECT flag
used for line segment intersection, depending on whether you wish for
the algorithm to consider barely touching segments to intersect.

All operations are O(1) in time and space.

*/

#include <algorithm> /* std::min(), std::max() */
#include <cmath>     /* fabs(), sqrt() */
#include <utility>   /* std::pair */

typedef std::pair<double, double> point;
#define x first
#define y second

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps)  /* equal to */
#define LT(a, b) ((a) < (b) - eps)         /* less than */
#define LE(a, b) ((a) <= (b) + eps)        /* less than or equal to */

//intersection of line l1 and line l2, each in ax + by + c = 0 form
//returns: -1, if lines do not intersect,
//          0, if there is exactly one intersection point, or
//         +1, if there are infinite intersection
//in the 2nd case, the intersection point is optionally stored into p
int line_intersection(const double & a1, const double & b1, const double & c1,
                      const double & a2, const double & b2, const double & c2,
                      point * p = 0) {
  if (EQ(a1 * b2, a2 * b1))
    return (EQ(a1 * c2, a2 * c1) || EQ(b1 * c2, b2 * c1)) ? 1 : -1;
  if (p != 0) {
    p->x = (b1 * c1 - b1 * c2) / (a2 * b1 - a1 * b2);
    if (!EQ(b1, 0)) p->y = -(a1 * p->x + c1) / b1;
    else p->y = -(a2 * p->x + c2) / b2;
  }
  return 0;
}

//intersection of line through p1, p2, and line through p2, p3
//returns: -1, if lines do not intersect,
//          0, if there is exactly one intersection point, or
//         +1, if there are infinite intersections
//in the 2nd case, the intersection point is optionally stored into p
int line_intersection(const point & p1, const point & p2,
                      const point & p3, const point & p4, point * p = 0) {
  double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
  double c1 = -(p1.x * p2.y - p2.x * p1.y);
  double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
  double c2 = -(p3.x * p4.y - p4.x * p3.y);
  double x = -(c1 * b2 - c2 * b1), y = -(a1 * c2 - a2 * c1);
  double det = a1 * b2 - a2 * b1;
  if (EQ(det, 0))
    return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
  if (p != 0) *p = point(x / det, y / det);
  return 0;
}

//Line Segment Intersection (http://stackoverflow.com/a/565282)

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double abs(const point & a) { return sqrt(norm(a)); }
double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }

//should we consider barely touching segments an intersection?
const bool TOUCH_IS_INTERSECT = true;

//does [l, h] contain m?
//precondition: l <= h
bool contain(const double & l, const double & m, const double & h) {
  if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
  return LT(l, m) && LT(m, h);
}

//does [l1, h1] overlap with [l2, h2]?
//precondition: l1 <= h1 and l2 <= h2
bool overlap(const double & l1, const double & h1,
             const double & l2, const double & h2) {
  if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
  return LT(l1, h2) && LT(l2, h1);
}

//intersection of line segment ab with line segment cd
//returns: -1, if segments do not intersect,
//          0, if there is exactly one intersection point
//         +1, if the intersection is another line segment
//In case 2, the intersection point is stored into p
//In case 3, the intersection segment is stored into p and q
int seg_intersection(const point & a, const point & b,
                     const point & c, const point & d,
                     point * p = 0, point * q = 0) {
  point ab(b.x - a.x, b.y - a.y);
  point ac(c.x - a.x, c.y - a.y);
  point cd(d.x - c.x, d.y - c.y);
  double c1 = cross(ab, cd), c2 = cross(ac, ab);
  if (EQ(c1, 0) && EQ(c2, 0)) { //collinear
    double t0 = dot(ac, ab) / norm(ab);
    double t1 = t0 + dot(cd, ab) / norm(ab);
    if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
      point res1 = std::max(std::min(a, b), std::min(c, d));
      point res2 = std::min(std::max(a, b), std::max(c, d));
      if (res1 == res2) {
        if (p != 0) *p = res1;
        return 0; //collinear, meeting at an endpoint
      }
      if (p != 0 && q != 0) *p = res1, *q = res2;
      return 1; //collinear and overlapping
    } else {
      return -1; //collinear and disjoint
    }
  }
  if (EQ(c1, 0)) return -1; //parallel and disjoint
  double t = cross(ac, cd) / c1, u = c2 / c1;
  if (contain(0, t, 1) && contain(0, u, 1)) {
    if (p != 0) *p = point(a.x + t * ab.x, a.y + t * ab.y);
    return 0; //non-parallel with one intersection
  }
  return -1; //non-parallel with no intersections
}

//determines the point on line ax + by + c = 0 that is closest to point p
//this always lies on the line through p perpendicular to l.
point closest_point(const double & a, const double & b, const double & c,
                    const point & p) {
  if (EQ(a, 0)) return point(p.x, -c); //horizontal line
  if (EQ(b, 0)) return point(-c, p.y); //vertical line
  point res;
  line_intersection(a, b, c, -b, a, b * p.x - a * p.y, &res);
  return res;
}

//determines the point on segment ab closest to point p
point closest_point(const point & a, const point & b, const point & p) {
  if (a == b) return a;
  point ap(p.x - a.x, p.y - a.y), ab(b.x - a.x, b.y - a.y);
  double t = dot(ap, ab) / norm(ab);
  if (t <= 0) return a;
  if (t >= 1) return b;
  return point(a.x + t * ab.x, a.y + t * ab.y);
}

/*** Example Usage ***/

#include <cassert>
#define pt point

int main() {
  point p;
  assert(line_intersection(-1, 1, 0, 1, 1, -3, &p) == 0);
  assert(p == pt(1.5, 1.5));
  assert(line_intersection(pt(0, 0), pt(1, 1), pt(0, 4), pt(4, 0), &p) == 0);
  assert(p == pt(2, 2));

  //tests for segment intersection (examples in order from link below)
  //http://martin-thoma.com/how-to-check-if-two-line-segments-intersect/
  {
#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)
    pt p, q;
    //intersection is a point
    assert(0 == test(-4, 0, 4, 0, 0, -4, 0, 4));   assert(p == pt(0, 0));
    assert(0 == test(0, 0, 10, 10, 2, 2, 16, 4));  assert(p == pt(2, 2));
    assert(0 == test(-2, 2, -2, -2, -2, 0, 0, 0)); assert(p == pt(-2, 0));
    assert(0 == test(0, 4, 4, 4, 4, 0, 4, 8));     assert(p == pt(4, 4));

    //intersection is a segment
    assert(1 == test(10, 10, 0, 0, 2, 2, 6, 6));
    assert(p == pt(2, 2) && q == pt(6, 6));
    assert(1 == test(6, 8, 14, -2, 14, -2, 6, 8));
    assert(p == pt(6, 8) && q == pt(14, -2));

    //no intersection
    assert(-1 == test(6, 8, 8, 10, 12, 12, 4, 4));
    assert(-1 == test(-4, 2, -8, 8, 0, 0, -4, 6));
    assert(-1 == test(4, 4, 4, 6, 0, 2, 0, 0));
    assert(-1 == test(4, 4, 6, 4, 0, 2, 0, 0));
    assert(-1 == test(-2, -2, 4, 4, 10, 10, 6, 6));
    assert(-1 == test(0, 0, 2, 2, 4, 0, 1, 4));
    assert(-1 == test(2, 2, 2, 8, 4, 4, 6, 4));
    assert(-1 == test(4, 2, 4, 4, 0, 8, 10, 0));
  }
  assert(pt(2.5, 2.5) == closest_point(-1, -1, 5, pt(0, 0)));
  assert(pt(3, 0)     == closest_point(1, 0, -3, pt(0, 0)));
  assert(pt(0, 3)     == closest_point(0, 1, -3, pt(0, 0)));

  assert(pt(3, 0)  == closest_point(pt(3, 0), pt(3, 3), pt(0, 0)));
  assert(pt(2, -1) == closest_point(pt(2, -1), pt(4, -1), pt(0, 0)));
  assert(pt(4, -1) == closest_point(pt(2, -1), pt(4, -1), pt(5, 0)));
  return 0;
}
```

5.2.4 Circle Intersections

```
/*

Tangent lines to circles, circle-line intersections, and circle-circle
intersections (intersection point(s) as well as area) in 2 dimensions.

All operations are O(1) in time and space.

*/

#include <algorithm> /* std::min(), std::max() */
#include <cmath>     /* acos(), fabs(), sqrt() */
#include <utility>   /* std::pair */

typedef std::pair<double, double> point;
#define x first
#define y second

```

```
const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define NE(a, b) (fabs((a) - (b)) > eps)  /* not equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */
#define GT(a, b) ((a) > (b) + eps)        /* greater than */
#define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */
#define GE(a, b) ((a) >= (b) - eps)       /* greater than or equal to */

struct circle {
  double h, k, r;

  circle(const double & h, const double & k, const double & r) {
    this->h = h;
    this->k = k;
    this->r = r;
  }
};

//note: this is a simplified version of line that is not canonicalized.
// e.g. comparing lines with == signs will not work as intended. For a
//      fully featured line class, see the whole geometry library.
struct line {
  double a, b, c;

  line() { a = b = c = 0; }

  line(const double & a, const double & b, const double & c) {
    this->a = a;
    this->b = b;
    this->c = c;
  }

  line(const point & p, const point & q) {
    a = p.y - q.y,
    b = q.x - p.x;
    c = -a * p.x - b * p.y;
  }
};

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double abs(const point & a) { return sqrt(norm(a)); }
double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }

//tangent line(s) to circle c passing through p. there are 3 cases:
//returns: 0, if there are no lines (p is strictly inside c)
//         1, if there is 1 tangent line (p is on the edge)
//         2, if there are 2 tangent lines (p is strictly outside)
//If there is only 1 tangent, then the line will be stored in l1.
//If there are 2, then they will be stored in l1 and l2 respectively.
int tangents(const circle & c, const point & p, line * l1 = 0, line * l2 = 0) {
  point vop(p.x - c.h, p.y - c.k);
  if (EQ(norm(vop), c.r * c.r)) { //on an edge, get perpendicular through p
    if (l1 != 0) {
      *l1 = line(point(c.h, c.k), p);
      *l1 = line(-l1->b, l1->a, l1->b * p.x - l1->a * p.y);
    }
    return 1;
  }
  if (LE(norm(vop), c.r * c.r)) return 0; //inside circle
  point q(vop.x / c.r, vop.y / c.r);
  double n = norm(q), d = q.y * sqrt(norm(q) - 1.0);
  point t1((q.x - d) / n, c.k), t2((q.x + d) / n, c.k);
  if (NE(q.y, 0)) { //common case
    t1.y += c.r * (1.0 - t1.x * q.x) / q.y;
    t2.y += c.r * (1.0 - t2.x * q.x) / q.y;
  } else { //point at center horizontal, y = 0
    d = c.r * sqrt(1.0 - t1.x * t1.x);
    t1.y += d;
    t2.y -= d;
  }
  t1.x = t1.x * c.r + c.h;
  t2.x = t2.x * c.r + c.h;
  //note: here, t1 and t2 are the two points of tangencies
  if (l1 != 0) *l1 = line(p, t1);
  if (l2 != 0) *l2 = line(p, t2);
  return 2;
}

//determines the intersection(s) between a circle c and line l
//returns: 0, if the line does not intersect with the circle
//         1, if the line is tangent (one intersection)
//         2, if the line crosses through the circle
//If there is 1 intersection point, it will be stored in p
//If there are 2, they will be stored in p and q respectively
int intersection(const circle & c, const line & l,
                 point * p = 0, point * q = 0) {
  double v = c.h * l.a + c.k * l.b + l.c;
  double aabb = l.a * l.a + l.b * l.b;
  double disc = v * v / aabb - c.r * c.r;
  if (disc > eps) return 0;
  double x0 = -l.a * l.c / aabb, y0 = -l.b * v / aabb;
  if (disc > -eps) {
    if (p != 0) *p = point(x0 + c.h, y0 + c.k);
    return 1;
  }
  double k = sqrt((disc /= -aabb) < 0 ? 0 : disc);
  if (p != 0) *p = point(x0 + k * l.b + c.h, y0 - k * l.a + c.k);
  if (q != 0) *q = point(x0 - k * l.b + c.h, y0 + k * l.a + c.k);
  return 2;
}

//determines the intersection points between two circles c1 and c2
//returns: -2, if circle c2 completely encloses circle c1
//         -1, if circle c1 completely encloses circle c2
//          0, if the circles are completely disjoint
//          1, if the circles are tangent (one intersection point)
//          2, if the circles intersect at two points
//          3, if the circles intersect at infinite points (c1 = c2)
//If one intersection, the intersection point is stored in p
//If two, the intersection points are stored in p and q respectively
int intersection(const circle & c1, const circle & c2,
                 point * p = 0, point * q = 0) {
  if (EQ(c1.h, c2.h) && EQ(c1.k, c2.k))
    return EQ(c1.r, c2.r) ? 3 : (c1.r > c2.r ? -1 : -2);
  point d12(point(c2.h - c1.h, c2.k - c1.k));
  double d = abs(d12);
  if (GT(d, c1.r + c2.r)) return 0;
  if (LT(d, fabs(c1.r - c2.r))) return c1.r > c2.r ? -1 : -2;
  double a = (c1.r * c1.r - c2.r * c2.r + d * d) / (2 * d);
  double x0 = c1.h + (d12.x * a / d);
  double y0 = c1.k + (d12.y * a / d);
  double s = sqrt(c1.r * c1.r - a * a);
  double rx = -d12.y * s / d, ry = d12.x * s / d;
  if (EQ(rx, 0) && EQ(ry, 0)) {
    if (p != 0) *p = point(x0, y0);
    return 1;
  }
  if (p != 0) *p = point(x0 - rx, y0 - ry);
  if (q != 0) *q = point(x0 + rx, y0 + ry);
  return 2;
}

const double PI = acos(-1.0);

//intersection area of circles c1 and c2
double intersection_area(const circle & c1, const circle c2) {
  double r = std::min(c1.r, c2.r), R = std::max(c1.r, c2.r);
  double d = abs(point(c2.h - c1.h, c2.k - c1.k));
  if (LE(d, R - r)) return PI * r * r;
  if (GE(d, R + r)) return 0;
  return r * r * acos((d * d + r * r - R * R) / 2 / d / r) +
         R * R * acos((d * d + R * R - r * r) / 2 / d / R) -
         0.5 * sqrt((-d + r + R) * (d + r - R) * (d - r + R) * (d + r + R));
}

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;
#define pt point

int main() {
  line l1, l2;
  assert(0 == tangents(circle(0, 0, 4), pt(1, 1), &l1, &l2));
  assert(1 == tangents(circle(0, 0, sqrt(2)), pt(1, 1), &l1, &l2));
  cout << l1.a << "␣" << l1.b << "␣" << l1.c << "\n"; // -x - y + 2 = 0
  assert(2 == tangents(circle(0, 0, 2), pt(2, 2), &l1, &l2));
  cout << l1.a << "␣" << l1.b << "␣" << l1.c << "\n"; //    -2y + 4 = 0
  cout << l2.a << "␣" << l2.b << "␣" << l2.c << "\n"; // 2x     - 4 = 0

  pt p, q;
  assert(0 == intersection(circle(1, 1, 3), line(5, 3, -30), &p, &q));
  assert(1 == intersection(circle(1, 1, 3), line(0, 1, -4), &p, &q));
  assert(p == pt(1, 4));
  assert(2 == intersection(circle(1, 1, 3), line(0, 1, -1), &p, &q));
  assert(p == pt(4, 1));
  assert(q == pt(-2, 1));

  assert(-2 == intersection(circle(1, 1, 1), circle(0, 0, 3), &p, &q));
  assert(-1 == intersection(circle(0, 0, 3), circle(1, 1, 1), &p, &q));
  assert(0 == intersection(circle(5, 0, 4), circle(-5, 0, 4), &p, &q));
  assert(1 == intersection(circle(-5, 0, 5), circle(5, 0, 5), &p, &q));
  assert(p == pt(0, 0));
  assert(2 == intersection(circle(-0.5, 0, 1), circle(0.5, 0, 1), &p, &q));
  assert(p == pt(0, -sqrt(3) / 2));
  assert(q == pt(0, sqrt(3) / 2));

  //example where each circle passes through the other circle's center
  //http://math.stackexchange.com/a/402891
  double r = 3;
  double a = intersection_area(circle(-r / 2, 0, r), circle(r / 2, 0, r));
  assert(EQ(a, r * r * (2 * PI / 3 - sqrt(3) / 2)));
  return 0;
}
```

5.3 Common Geometric Computations

5.3.1 Polygon Sorting and Area

```
/*

centroid() - Simply returns the geometric average point of all the
points given. This could be used to find the reference center point
for the following function. An empty range will result in (0, 0).
Complexity: O(n) on the number of points in the given range.

cw_comp() - Given a set of points, these points could possibly form
many different polygons. The following sorting comparators, when
used in conjunction with std::sort, will produce one such ordering
of points which is sorted in clockwise order relative to a custom-
defined center point that must be set beforehand. This could very
well be the result of mean_point(). ccw_comp() is the opposite
function, which produces the points in counterclockwise order.
Complexity: O(1) per call.

polygon_area() - A given range of points is interpreted as a polygon
based on the ordering they're given in. The shoelace formula is used
to determine its area. The polygon does not necessarily have to be
sorted using one of the functions above, but may be any ordering that
produces a valid polygon. You may optionally pass the last point in
the range equal to the first point and still expect the correct result.
Complexity: O(n) on the number of points in the range, assuming that
the points are already sorted in the order that specifies the polygon.

*/

#include <algorithm> /* std::sort() */
#include <cmath>     /* fabs() */
#include <utility>   /* std::pair */

typedef std::pair<double, double> point;
#define x first
#define y second

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */
#define GE(a, b) ((a) >= (b) - eps)       /* greater than or equal to */

//magnitude of the 3D cross product with Z component implicitly equal to 0
```

```
//the answer assumes the origin (0, 0) is instead shifted to point o.
//this is equal to 2x the signed area of the triangle from these 3 points.
double cross(const point & o, const point & a, const point & b) {
  return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

point ctr;

template<class It> point centroid(It lo, It hi) {
  if (lo == hi) return point(0, 0);
  double xtot = 0, ytot = 0, points = hi - lo;
  for (; lo != hi; ++lo) {
    xtot += lo->x;
    ytot += lo->y;
  }
  return point(xtot / points, ytot / points);
}

//ctr must be defined beforehand
bool cw_comp(const point & a, const point & b) {
  if (GE(a.x - ctr.x, 0) && LT(b.x - ctr.x, 0)) return true;
  if (LT(a.x - ctr.x, 0) && GE(b.x - ctr.x, 0)) return false;
  if (EQ(a.x - ctr.x, 0) && EQ(b.x - ctr.x, 0)) {
    if (GE(a.y - ctr.y, 0) || GE(b.y - ctr.y, 0))
      return a.y > b.y;
    return b.y > a.y;
  }
  double det = cross(ctr, a, b);
  if (EQ(det, 0))
    return (a.x - ctr.x) * (a.x - ctr.x) + (a.y - ctr.y) * (a.y - ctr.y) >
           (b.x - ctr.x) * (b.x - ctr.x) + (b.y - ctr.y) * (b.y - ctr.y);
  return det < 0;
}

bool ccw_comp(const point & a, const point & b) {
  return cw_comp(b, a);
}

//area of a polygon specified by range [lo, hi) - shoelace formula in O(n)
//[lo, hi) must point to the polygon vertices, sorted in CW or CCW order
template<class It> double polygon_area(It lo, It hi) {
  if (lo == hi) return 0;
  double area = 0;
  if (*lo != *--hi)
    area += (lo->x - hi->x) * (lo->y + hi->y);
  for (It i = hi, j = hi - 1; i != lo; --i, --j)
    area += (i->x - j->x) * (i->y + j->y);
  return fabs(area / 2.0);
}

/*** Example Usage ***/

#include <cassert>
#include <vector>
using namespace std;
#define pt point

int main() {
  //irregular pentagon with only (1, 2) not on the convex hull
  //the ordering here is already sorted in ccw order around their centroid
  //we will scramble them and see if our comparator works
  pt pts[] = {pt(1, 3), pt(1, 2), pt(2, 1), pt(0, 0), pt(-1, 3)};
  vector<pt> v(pts, pts + 5);
  std::random_shuffle(v.begin(), v.end());
  ctr = centroid(v.begin(), v.end()); //note: ctr is a global variable
  assert(EQ(ctr.x, 0.6) && EQ(ctr.y, 1.8));
  sort(v.begin(), v.end(), cw_comp);
  for (int i = 0; i < (int)v.size(); i++) assert(v[i] == pts[i]);
  assert(EQ(polygon_area(v.begin(), v.end()), 5));
  return 0;
}
```

5.3.2 Point in Polygon Query

```
/*

Given a single point p and another range of points specifying a
polygon, determine whether p lies within the polygon. Note that
you should modify the EDGE_IS_INSIDE flag, depending on whether
you wish for the algorithm to consider points lying on an edge of
the polygon to be inside it.

Complexity: O(n) on the number of vertices in the polygon.

*/

#include <algorithm> /* std::sort() */
#include <cmath>     /* fabs() */
#include <utility>   /* std::pair */

typedef std::pair<double, double> point;
#define x first
#define y second

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define GT(a, b) ((a) > (b) + eps)        /* greater than */
#define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */

//should we consider points lying on an edge to be inside the polygon?
const bool EDGE_IS_INSIDE = true;

//magnitude of the 3D cross product with Z component implicitly equal to 0
//the answer assumes the origin (0, 0) is instead shifted to point o.
//this is equal to 2x the signed area of the triangle from these 3 points.
double cross(const point & o, const point & a, const point & b) {
  return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

//return whether point p is in polygon specified by range [lo, hi) in O(n)
//[lo, hi) must point to the polygon vertices, sorted in CW or CCW order
template<class It> bool point_in_polygon(const point & p, It lo, It hi) {
  int cnt = 0;
  for (It i = lo, j = hi - 1; i != hi; j = i++) {
    if (EQ(i->y, p.y) && (EQ(i->x, p.x) ||
                         (EQ(j->y, p.y) && (LE(i->x, p.x) || LE(j->x, p.x)))))
      return EDGE_IS_INSIDE; //on an edge
    if (GT(i->y, p.y) != GT(j->y, p.y)) {
      double det = cross(p, *i, *j);
      if (EQ(det, 0)) return EDGE_IS_INSIDE; //on an edge
      if (GT(det, 0) != GT(j->y, i->y)) cnt++;
    }
  }
  return cnt % 2 == 1;
}

/*** Example Usage ***/

#include <cassert>
using namespace std;
#define pt point

int main() {
  //irregular trapezoid
  pt p[] = {pt(-1, 3), pt(1, 3), pt(2, 1), pt(0, 0)};
  assert(point_in_polygon(pt(1, 2), p, p + 4));
  assert(point_in_polygon(pt(0, 3), p, p + 4));
  assert(!point_in_polygon(pt(0, 3.01), p, p + 4));
  assert(!point_in_polygon(pt(2, 2), p, p + 4));
  return 0;
}
```

5.3.3 Convex Hull

```
/*

Determines the convex hull from a range of points, that is, the
smallest convex polygon (a polygon such that every line which
crosses through it will only cross through it once) that contains
all of the points. This function uses the monotone chain algorithm
to compute the upper and lower hulls separately.

Returns: a vector of the convex hull points in clockwise order.
Complexity: O(n log n) on the number of points given

Notes: To yield the hull points in counterclockwise order,
       replace every usage of GE() in the function with LE().
       To have the first point on the hull repeated as the last,
       replace the last line of the function to res.resize(k);

*/

#include <algorithm> /* std::sort() */
#include <cmath>     /* fabs() */
#include <utility>   /* std::pair */
#include <vector>

typedef std::pair<double, double> point;
#define x first
#define y second

//change < 0 comparisons to > 0 to produce hull points in CCW order
```

```cpp
double cw(const point & o, const point & a, const point & b) {
  return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x) < 0;
}

//convex hull from a range [lo, hi) of points
//monotone chain in O(n log n) to find hull points in CW order
//notes: the range of input points will be sorted lexicographically
template<class It> std::vector<point> convex_hull(It lo, It hi) {
  int k = 0;
  if (hi - lo <= 1) return std::vector<point>(lo, hi);
  std::vector<point> res(2 * (int)(hi - lo));
  std::sort(lo, hi); //compare by x, then by y if x-values are equal
  for (It it = lo; it != hi; ++it) {
    while (k >= 2 && !cw(res[k - 2], res[k - 1], *it)) k--;
    res[k++] = *it;
  }
  int t = k + 1;
  for (It it = hi - 2; it != lo - 1; --it) {
    while (k >= t && !cw(res[k - 2], res[k - 1], *it)) k--;
    res[k++] = *it;
  }
  res.resize(k - 1);
  return res;
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  //irregular pentagon with only (1, 2) not on the convex hull
  vector<point> v;
  v.push_back(point(1, 3));
  v.push_back(point(1, 2));
  v.push_back(point(2, 1));
  v.push_back(point(0, 0));
  v.push_back(point(-1, 3));
  std::random_shuffle(v.begin(), v.end());
  vector<point> h = convex_hull(v.begin(), v.end());
  cout << "hull␣points:";
  for (int i = 0; i < (int)h.size(); i++)
    cout << "␣(" << h[i].x << "," << h[i].y << ")";
  cout << "\n";
  return 0;
}
```

5.3.4 Minimum Enclosing Circle

```cpp
/*

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

```
be preserved.

Time Complexity: O(n) average on the number of points given.

*/

#include <algorithm>
#include <cmath>
#include <stdexcept>
#include <utility>

const double eps = 1e-9;

#define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */

typedef std::pair<double, double> point;
#define x first
#define y second

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double abs(const point & a) { return sqrt(norm(a)); }

struct circle {

  double h, k, r;

  circle(): h(0), k(0), r(0) {}
  circle(const double & H, const double & K, const double & R):
    h(H), k(K), r(fabs(R)) {}

  //circumcircle with the diameter equal to the distance from a to b
  circle(const point & a, const point & b) {
    h = (a.x + b.x) / 2.0;
    k = (a.y + b.y) / 2.0;
    r = abs(point(a.x - h, a.y - k));
  }

  //circumcircle of 3 points - throws exception if abc are collinear/equal
  circle(const point & a, const point & b, const point & c) {
    double an = norm(point(b.x - c.x, b.y - c.y));
    double bn = norm(point(a.x - c.x, a.y - c.y));
    double cn = norm(point(a.x - b.x, a.y - b.y));
    double wa = an * (bn + cn - an);
    double wb = bn * (an + cn - bn);
    double wc = cn * (an + bn - cn);
    double w = wa + wb + wc;
    if (fabs(w) < eps)
      throw std::runtime_error("No␣circle␣from␣collinear␣points.");
    h = (wa * a.x + wb * b.x + wc * c.x) / w;
    k = (wa * a.y + wb * b.y + wc * c.y) / w;
    r = abs(point(a.x - h, a.y - k));
  }

  bool contains(const point & p) const {
    return LE(norm(point(p.x - h, p.y - k)), r * r);
  }

};

```

```
template<class It> circle smallest_circle(It lo, It hi) {
  if (lo == hi) return circle(0, 0, 0);
  if (lo + 1 == hi) return circle(lo->x, lo->y, 0);
  std::random_shuffle(lo, hi);
  circle res(*lo, *(lo + 1));
  for (It i = lo + 2; i != hi; ++i) {
    if (res.contains(*i)) continue;
    res = circle(*lo, *i);
    for (It j = lo + 1; j != i; ++j) {
      if (res.contains(*j)) continue;
      res = circle(*i, *j);
      for (It k = lo; k != j; ++k)
        if (!res.contains(*k)) res = circle(*i, *j, *k);
    }
  }
  return res;
}

/*** Example Usage ***/

#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<point> v;
  v.push_back(point(0, 0));
  v.push_back(point(0, 1));
  v.push_back(point(1, 0));
  v.push_back(point(1, 1));
  circle res = smallest_circle(v.begin(), v.end());
  cout << "center:␣(" << res.h << ",␣" << res.k << ")\n";
  cout << "radius:␣" << res.r << "\n";
  return 0;
}
```

5.3.5 Diameter of Point Set

```
/*

Determines the diametral pair of a range of points. The diamter
of a set of points is the largest distance between any two
points in the set. A diametral pair is a pair of points in the
set whose distance is equal to the set's diameter. The following
program uses rotating calipers method to find a solution.

Time Complexity: O(n log n) on the number of points in the set.

*/

#include <algorithm> /* std::sort() */
#include <cmath>     /* fabs(), sqrt() */
#include <utility>   /* std::pair */
#include <vector>

typedef std::pair<double, double> point;
#define x first
```

```
#define y second

double sqdist(const point & a, const point & b) {
  double dx = a.x - b.x, dy = a.y - b.y;
  return sqrt(dx * dx + dy * dy);
}

double cross(const point & o, const point & a, const point & b) {
  return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

bool cw(const point & o, const point & a, const point & b) {
  return cross(o, a, b) < 0;
}

double area(const point & o, const point & a, const point & b) {
  return fabs(cross(o, a, b));
}

template<class It> std::vector<point> convex_hull(It lo, It hi) {
  int k = 0;
  if (hi - lo <= 1) return std::vector<point>(lo, hi);
  std::vector<point> res(2 * (int)(hi - lo));
  std::sort(lo, hi); //compare by x, then by y if x-values are equal
  for (It it = lo; it != hi; ++it) {
    while (k >= 2 && !cw(res[k - 2], res[k - 1], *it)) k--;
    res[k++] = *it;
  }
  int t = k + 1;
  for (It it = hi - 2; it != lo - 1; --it) {
    while (k >= t && !cw(res[k - 2], res[k - 1], *it)) k--;
    res[k++] = *it;
  }
  res.resize(k - 1);
  return res;
}

template<class It> std::pair<point, point> diametral_pair(It lo, It hi) {
  std::vector<point> h = convex_hull(lo, hi);
  int m = h.size();
  if (m == 1) return std::make_pair(h[0], h[0]);
  if (m == 2) return std::make_pair(h[0], h[1]);
  int k = 1;
  while (area(h[m - 1], h[0], h[(k + 1) % m]) > area(h[m - 1], h[0], h[k]))
    k++;
  double maxdist = 0, d;
  std::pair<point, point> res;
  for (int i = 0, j = k; i <= k && j < m; i++) {
    d = sqdist(h[i], h[j]);
    if (d > maxdist) {
      maxdist = d;
      res = std::make_pair(h[i], h[j]);
    }
    while (j < m && area(h[i], h[(i + 1) % m], h[(j + 1) % m]) >
                    area(h[i], h[(i + 1) % m], h[j])) {
      d = sqdist(h[i], h[(j + 1) % m]);
      if (d > maxdist) {
        maxdist = d;
        res = std::make_pair(h[i], h[(j + 1) % m]);
    }
    j++;
   }
  }
  return res;
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  vector<point> v;
  v.push_back(point(0, 0));
  v.push_back(point(3, 0));
  v.push_back(point(0, 3));
  v.push_back(point(1, 1));
  v.push_back(point(4, 4));
  pair<point, point> res = diametral_pair(v.begin(), v.end());
  cout << "diametral␣pair:␣(" << res.first.x << "," << res.first.y << ")␣";
  cout << "(" << res.second.x << "," << res.second.y << ")\n";
  cout << "diameter:␣" << sqrt(sqdist(res.first, res.second)) << "\n";
  return 0;
}
```

5.3.6 Closest Point Pair

```
/*

Given a range containing distinct points on the Cartesian plane,
determine two points which have the closest possible distance.
A divide and conquer algorithm is used. Note that the ordering
of points in the input range may be changed by the function.

Time Complexity: O(n log^2 n) where n is the number of points.

*/

#include <algorithm> /* std::min, std::sort */
#include <cfloat>    /* DBL_MAX */
#include <cmath>     /* fabs */
#include <utility>   /* std::pair */

typedef std::pair<double, double> point;
#define x first
#define y second

double sqdist(const point & a, const point & b) {
  double dx = a.x - b.x, dy = a.y - b.y;
  return dx * dx + dy * dy;
}

bool cmp_x(const point & a, const point & b) { return a.x < b.x; }
bool cmp_y(const point & a, const point & b) { return a.y < b.y; }

template<class It>
```

```
double rec(It lo, It hi, std::pair<point, point> & res, double mindist) {
  if (lo == hi) return DBL_MAX;
  It mid = lo + (hi - lo) / 2;
  double midx = mid->x;
  double d1 = rec(lo, mid, res, mindist);
  mindist = std::min(mindist, d1);
  double d2 = rec(mid + 1, hi, res, mindist);
  mindist = std::min(mindist, d2);
  std::sort(lo, hi, cmp_y);
  int size = 0;
  It t[hi - lo];
  for (It it = lo; it != hi; ++it)
    if (fabs(it->x - midx) < mindist)
      t[size++] = it;
  for (int i = 0; i < size; i++) {
    for (int j = i + 1; j < size; j++) {
      point a = *t[i], b = *t[j];
      if (b.y - a.y >= mindist) break;
      double dist = sqdist(a, b);
      if (mindist > dist) {
        mindist = dist;
        res = std::make_pair(a, b);
      }
    }
  }
  return mindist;
}

template<class It> std::pair<point, point> closest_pair(It lo, It hi) {
  std::pair<point, point> res;
  std::sort(lo, hi, cmp_x);
  rec(lo, hi, res, DBL_MAX);
  return res;
}

/*** Example Usage ***/

#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<point> v;
  v.push_back(point(2, 3));
  v.push_back(point(12, 30));
  v.push_back(point(40, 50));
  v.push_back(point(5, 1));
  v.push_back(point(12, 10));
  v.push_back(point(3, 4));
  pair<point, point> res = closest_pair(v.begin(), v.end());
  cout << "closest␣pair:␣(" << res.first.x << "," << res.first.y << ")␣";
  cout << "(" << res.second.x << "," << res.second.y << ")\n";
  cout << "dist:␣" << sqrt(sqdist(res.first, res.second)) << "\n"; //1.41421
  return 0;
}
```

5.3.7 Segment Intersection Finding

```
/*

Given a range of segments on the Cartesian plane, identify one
pair of segments which intersect each other. This is done using
a sweep line algorithm.

Time Complexity: O(n log n) where n is the number of segments.

*/

#include <algorithm> /* std::min(), std::max(), std::sort() */
#include <cmath>     /* fabs() */
#include <set>
#include <utility>   /* std::pair */

typedef std::pair<double, double> point;
#define x first
#define y second

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps)  /* equal to */
#define LT(a, b) ((a) < (b) - eps)         /* less than */
#define LE(a, b) ((a) <= (b) + eps)        /* less than or equal to */

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }
double cross(const point & o, const point & a, const point & b) {
  return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

const bool TOUCH_IS_INTERSECT = true;

bool contain(const double & l, const double & m, const double & h) {
  if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
  return LT(l, m) && LT(m, h);
}

bool overlap(const double & l1, const double & h1,
             const double & l2, const double & h2) {
  if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
  return LT(l1, h2) && LT(l2, h1);
}

int seg_intersection(const point & a, const point & b,
                     const point & c, const point & d) {
  point ab(b.x - a.x, b.y - a.y);
  point ac(c.x - a.x, c.y - a.y);
  point cd(d.x - c.x, d.y - c.y);
  double c1 = cross(ab, cd), c2 = cross(ac, ab);
  if (EQ(c1, 0) && EQ(c2, 0)) {
    double t0 = dot(ac, ab) / norm(ab);
    double t1 = t0 + dot(cd, ab) / norm(ab);
    if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
      point res1 = std::max(std::min(a, b), std::min(c, d));
      point res2 = std::min(std::max(a, b), std::max(c, d));
      return (res1 == res2) ? 0 : 1;
    }
    return -1;
  }
  if (EQ(c1, 0)) return -1;
  double t = cross(ac, cd) / c1, u = c2 / c1;
  if (contain(0, t, 1) && contain(0, u, 1)) return 0;
  return -1;
}

struct segment {
  point p, q;

  segment() {}
  segment(const point & p, const point & q) {
    if (p < q) {
      this->p = p;
      this->q = q;
    } else {
      this->p = q;
      this->q = p;
    }
  }

  bool operator < (const segment & rhs) const {
    if (p.x < rhs.p.x) {
      double c = cross(p, q, rhs.p);
      if (c != 0) return c > 0;
    } else if (p.x > rhs.p.x) {
      double c = cross(rhs.p, rhs.q, q);
      if (c != 0) return c < 0;
    }
    return p.y < rhs.p.y;
  }
};

template<class SegIt> struct event {
  point p;
  int type;
  SegIt seg;

  event() {}
  event(const point & p, const int type, SegIt seg) {
    this->p = p;
    this->type = type;
    this->seg = seg;
  }

  bool operator < (const event & rhs) const {
    if (p.x != rhs.p.x) return p.x < rhs.p.x;
    if (type != rhs.type) return type > rhs.type;
    return p.y < rhs.p.y;
  }
};

bool intersect(const segment & s1, const segment & s2) {
  return seg_intersection(s1.p, s1.q, s2.p, s2.q) >= 0;
}

//returns whether any pair of segments in the range [lo, hi) intersect
//if the result is true, one such intersection pair will be stored
```

```
//into values pointed to by res1 and res2.
template<class It>
bool find_intersection(It lo, It hi, segment * res1, segment * res2) {
  int cnt = 0;
  event<It> e[2 * (hi - lo)];
  for (It it = lo; it != hi; ++it) {
    if (it->p > it->q) std::swap(it->p, it->q);
    e[cnt++] = event<It>(it->p, 1, it);
    e[cnt++] = event<It>(it->q, -1, it);
  }
  std::sort(e, e + cnt);
  std::set<segment> s;
  std::set<segment>::iterator it, next, prev;
  for (int i = 0; i < cnt; i++) {
    It seg = e[i].seg;
    if (e[i].type == 1) {
      it = s.lower_bound(*seg);
      if (it != s.end() && intersect(*it, *seg)) {
        *res1 = *it; *res2 = *seg;
        return true;
      }
      if (it != s.begin() && intersect(*--it, *seg)) {
        *res1 = *it; *res2 = *seg;
        return true;
      }
      s.insert(*seg);
    } else {
      it = s.lower_bound(*seg);
      next = prev = it;
      prev = it;
      if (it != s.begin() && it != --s.end()) {
        ++next;
        --prev;
        if (intersect(*next, *prev)) {
          *res1 = *next; *res2 = *prev;
          return true;
        }
      }
      s.erase(it);
    }
  }
  return false;
}

/*** Example Usage ***/

#include <iostream>
#include <vector>
using namespace std;

void print(const segment & s) {
  cout << "(" << s.p.x << "," << s.p.y << ")<->";
  cout << "(" << s.q.x << "," << s.q.y << ")\n";
}

int main() {
  vector<segment> v;
  v.push_back(segment(point(0, 0), point(2, 2)));
  v.push_back(segment(point(3, 0), point(0, -1)));
  v.push_back(segment(point(0, 2), point(2, -2)));
  v.push_back(segment(point(0, 3), point(9, 0)));
  segment res1, res2;
  bool res = find_intersection(v.begin(), v.end(), &res1, &res2);
  if (res) {
    print(res1);
    print(res2);
  } else {
    cout << "No␣intersections.\n";
  }
  return 0;
}
```

5.4 Advanced Geometric Computations

5.4.1 Convex Polygon Cut

```
/*

Given a range of points specifying a polygon on the Cartesian
plane, as well as two points specifying an infinite line, "cut"
off the right part of the polygon with the line and return the
resulting polygon that is the left part.

Time Complexity: O(n) on the number of points in the poylgon.

*/

#include <cmath>   /* fabs() */
#include <utility> /* std::pair */
#include <vector>

typedef std::pair<double, double> point;
#define x first
#define y second

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */
#define GT(a, b) ((a) > (b) + eps)        /* greater than */

double cross(const point & o, const point & a, const point & b) {
  return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

int orientation(const point & o, const point & a, const point & b) {
  double c = cross(o, a, b);
  return LT(c, 0) ? -1 : (GT(c, 0) ? 1 : 0);
}

int line_intersection(const point & p1, const point & p2,
                      const point & p3, const point & p4, point * p = 0) {
  double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
  double c1 = -(p1.x * p2.y - p2.x * p1.y);
  double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
  double c2 = -(p3.x * p4.y - p4.x * p3.y);
  double x = -(c1 * b2 - c2 * b1), y = -(a1 * c2 - a2 * c1);
  double det = a1 * b2 - a2 * b1;
  if (EQ(det, 0))
    return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
  if (p != 0) *p = point(x / det, y / det);
  return 0;
}

template<class It>
std::vector<point> convex_cut(It lo, It hi, const point & p, const point & q) {
  std::vector<point> res;
  for (It i = lo, j = hi - 1; i != hi; j = i++) {
    int d1 = orientation(p, q, *j), d2 = orientation(p, q, *i);
    if (d1 >= 0) res.push_back(*j);
    if (d1 * d2 < 0) {
      point r;
      line_intersection(p, q, *j, *i, &r);
      res.push_back(r);
    }
  }
  return res;
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  //irregular pentagon with only (1, 2) not on the convex hull
  vector<point> v;
  v.push_back(point(1, 3));
  v.push_back(point(1, 2));
  v.push_back(point(2, 1));
  v.push_back(point(0, 0));
  v.push_back(point(-1, 3));
  //cut using the vertical line through (0, 0)
  vector<point> res = convex_cut(v.begin(), v.end(), point(0, 0), point(0, 1));
  cout << "left␣cut:\n";
  for (int i = 0; i < (int)res.size(); i++)
    cout << "(" << res[i].x << "," << res[i].y << ")\n";
  return 0;
}
```

5.4.2 Polygon Union and Intersection

```
/*

Given two ranges of points respectively denoting the vertices of
two polygons, determine the intersection area of those polygons.
Using this, we can easily calculate their union with the forumla:
  union_area(A, B) = area(A) + area(B) - intersection_area(A, B)

Time Complexity: O(n^2 log n), where n is the total number of vertices.

*/
```

```cpp

#include <algorithm> /* std::sort() */
#include <cmath>     /* fabs(), sqrt() */
#include <set>
#include <utility>   /* std::pair */
#include <vector>

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */
#define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */

typedef std::pair<double, double> point;
#define x first
#define y second

inline int sgn(const double & x) {
  return (0.0 < x) - (x < 0.0);
}

//Line and line segment intersection (see their own sections)

int line_intersection(const point & p1, const point & p2,
                      const point & p3, const point & p4, point * p = 0) {
  double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
  double c1 = -(p1.x * p2.y - p2.x * p1.y);
  double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
  double c2 = -(p3.x * p4.y - p4.x * p3.y);
  double x = -(c1 * b2 - c2 * b1), y = -(a1 * c2 - a2 * c1);
  double det = a1 * b2 - a2 * b1;
  if (EQ(det, 0))
    return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
  if (p != 0) *p = point(x / det, y / det);
  return 0;
}

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }

const bool TOUCH_IS_INTERSECT = true;

bool contain(const double & l, const double & m, const double & h) {
  if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
  return LT(l, m) && LT(m, h);
}

bool overlap(const double & l1, const double & h1,
             const double & l2, const double & h2) {
  if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
  return LT(l1, h2) && LT(l2, h1);
}

int seg_intersection(const point & a, const point & b,
                     const point & c, const point & d,
                     point * p = 0, point * q = 0) {
  point ab(b.x - a.x, b.y - a.y);
  point ac(c.x - a.x, c.y - a.y);
  point cd(d.x - c.x, d.y - c.y);
  double c1 = cross(ab, cd), c2 = cross(ac, ab);
  if (EQ(c1, 0) && EQ(c2, 0)) { //collinear
    double t0 = dot(ac, ab) / norm(ab);
    double t1 = t0 + dot(cd, ab) / norm(ab);
    if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
      point res1 = std::max(std::min(a, b), std::min(c, d));
      point res2 = std::min(std::max(a, b), std::max(c, d));
      if (res1 == res2) {
        if (p != 0) *p = res1;
        return 0; //collinear, meeting at an endpoint
      }
      if (p != 0 && q != 0) *p = res1, *q = res2;
      return 1; //collinear and overlapping
    } else {
      return -1; //collinear and disjoint
    }
  }
  if (EQ(c1, 0)) return -1; //parallel and disjoint
  double t = cross(ac, cd) / c1, u = c2 / c1;
  if (contain(0, t, 1) && contain(0, u, 1)) {
    if (p != 0) *p = point(a.x + t * ab.x, a.y + t * ab.y);
    return 0; //non-parallel with one intersection
  }
  return -1; //non-parallel with no intersections
}

struct event {
  double y;
  int mask_delta;

  event(double y = 0, int mask_delta = 0) {
    this->y = y;
    this->mask_delta = mask_delta;
  }

  bool operator < (const event & e) const {
    if (y != e.y) return y < e.y;
    return mask_delta < e.mask_delta;
  }
};

template<class It>
double intersection_area(It lo1, It hi1, It lo2, It hi2) {
  It plo[2] = {lo1, lo2}, phi[] = {hi1, hi2};
  std::set<double> xs;
  for (It i1 = lo1; i1 != hi1; ++i1) xs.insert(i1->x);
  for (It i2 = lo2; i2 != hi2; ++i2) xs.insert(i2->x);
  for (It i1 = lo1, j1 = hi1 - 1; i1 != hi1; j1 = i1++) {
    for (It i2 = lo2, j2 = hi2 - 1; i2 != hi2; j2 = i2++) {
      point p;
      if (seg_intersection(*i1, *j1, *i2, *j2, &p) == 0)
        xs.insert(p.x);
    }
  }
  std::vector<double> xsa(xs.begin(), xs.end());
  double res = 0;
  for (int k = 0; k < (int)xsa.size() - 1; k++) {
    double x = (xsa[k] + xsa[k + 1]) / 2;
    point sweep0(x, 0), sweep1(x, 1);
    std::vector<event> events;
    for (int poly = 0; poly < 2; poly++) {
      It lo = plo[poly], hi = phi[poly];
      double area = 0;
      for (It i = lo, j = hi - 1; i != hi; j = i++)
        area += (j->x - i->x) * (j->y + i->y);
      for (It j = lo, i = hi - 1; j != hi; i = j++) {
        point p;
        if (line_intersection(*j, *i, sweep0, sweep1, &p) == 0) {
          double y = p.y, x0 = i->x, x1 = j->x;
          if (x0 < x && x1 > x) {
            events.push_back(event(y,  sgn(area) * (1 << poly)));
          } else if (x0 > x && x1 < x) {
            events.push_back(event(y, -sgn(area) * (1 << poly)));
          }
        }
      }
    }
    std::sort(events.begin(), events.end());
    double a = 0.0;
    int mask = 0;
    for (int j = 0; j < (int)events.size(); j++) {
      if (mask == 3)
        a += events[j].y - events[j - 1].y;
      mask += events[j].mask_delta;
    }
    res += a * (xsa[k + 1] - xsa[k]);
  }
  return res;
}

template<class It> double polygon_area(It lo, It hi) {
  if (lo == hi) return 0;
  double area = 0;
  if (*lo != *--hi)
    area += (lo->x - hi->x) * (lo->y + hi->y);
  for (It i = hi, j = hi - 1; i != lo; --i, --j)
    area += (i->x - j->x) * (i->y + j->y);
  return fabs(area / 2.0);
}

template<class It>
double union_area(It lo1, It hi1, It lo2, It hi2) {
  return polygon_area(lo1, hi1) + polygon_area(lo2, hi2) -
         intersection_area(lo1, hi1, lo2, hi2);
}

/*** Example Usage ***/

#include <cassert>
using namespace std;

int main() {
  vector<point> p1, p2;

  //irregular pentagon with area 1.5 triangle in quadrant 2
  p1.push_back(point(1, 3));
  p1.push_back(point(1, 2));
  p1.push_back(point(2, 1));
  p1.push_back(point(0, 0));
  p1.push_back(point(-1, 3));
  //a big square in quadrant 2
  p2.push_back(point(0, 0));
  p2.push_back(point(0, 3));
  p2.push_back(point(-3, 3));
  p2.push_back(point(-3, 0));

  assert(EQ(1.5, intersection_area(p1.begin(), p1.end(),
                                   p2.begin(), p2.end())));
  assert(EQ(12.5, union_area(p1.begin(), p1.end(),
                             p2.begin(), p2.end())));
  return 0;
}
```

5.4.3 Delaunay Triangulation (Simple)

```
/*

Given a range of points P on the Cartesian plane, the Delaunay
Triangulation of said points is a set of non-overlapping triangles
covering the entire convex hull of P, such that no point in P lies
within the circumcircle of any of the resulting triangles. The
triangulation maximizes the minimum angle of all the angles of the
triangles in the triangulation. In addition, for any point p in the
convex hull (not necessarily in P), the nearest point is guaranteed
to be a vertex of the enclosing triangle from the triangulation.
See: https://en.wikipedia.org/wiki/Delaunay_triangulation

The triangulation may not exist (e.g. for a set of collinear points)
or it may not be unique (multiple possible triangulations may exist).
The triangulation may not exist (e.g. for a set of collinear points)
or it may not be unique (multiple possible triangulations may exist).
The following program assumes that a triangulation exists, and
produces one such valid result using one of the simplest algorithms
to solve this problem. It involves encasing the simplex in a circle
and rejecting the simplex if another point in the tessellation is
within the generalized circle.

Time Complexity: O(n^4) on the number of input points.

*/

#include <algorithm> /* std::sort() */
#include <cmath>     /* fabs(), sqrt() */
#include <utility>   /* std::pair */
#include <vector>

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */
#define GT(a, b) ((a) > (b) + eps)        /* greater than */
#define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */
#define GE(a, b) ((a) >= (b) - eps)       /* greater than or equal to */

```

```
typedef std::pair<double, double> point;
#define x first
#define y second

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }

const bool TOUCH_IS_INTERSECT = false;

bool contain(const double & l, const double & m, const double & h) {
  if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
  return LT(l, m) && LT(m, h);
}

bool overlap(const double & l1, const double & h1,
             const double & l2, const double & h2) {
  if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
  return LT(l1, h2) && LT(l2, h1);
}

int seg_intersection(const point & a, const point & b,
                     const point & c, const point & d) {
  point ab(b.x - a.x, b.y - a.y);
  point ac(c.x - a.x, c.y - a.y);
  point cd(d.x - c.x, d.y - c.y);
  double c1 = cross(ab, cd), c2 = cross(ac, ab);
  if (EQ(c1, 0) && EQ(c2, 0)) {
    double t0 = dot(ac, ab) / norm(ab);
    double t1 = t0 + dot(cd, ab) / norm(ab);
    if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
      point res1 = std::max(std::min(a, b), std::min(c, d));
      point res2 = std::min(std::max(a, b), std::max(c, d));
      return (res1 == res2) ? 0 : 1;
    }
    return -1;
  }
  if (EQ(c1, 0)) return -1;
  double t = cross(ac, cd) / c1, u = c2 / c1;
  if (contain(0, t, 1) && contain(0, u, 1)) return 0;
  return -1;
}

struct triangle { point a, b, c; };

template<class It>
std::vector<triangle> delaunay_triangulation(It lo, It hi) {
  int n = hi - lo;
  std::vector<double> x, y, z;
  for (It it = lo; it != hi; ++it) {
    x.push_back(it->x);
    y.push_back(it->y);
    z.push_back((it->x) * (it->x) + (it->y) * (it->y));
  }
  std::vector<triangle> res;
  for (int i = 0; i < n - 2; i++) {
    for (int j = i + 1; j < n; j++) {
      for (int k = i + 1; k < n; k++) {
        if (j == k) continue;
        double nx = (y[j] - y[i]) * (z[k] - z[i]) - (y[k] - y[i]) * (z[j] - z[i]);
        double ny = (x[k] - x[i]) * (z[j] - z[i]) - (x[j] - x[i]) * (z[k] - z[i]);
        double nz = (x[j] - x[i]) * (y[k] - y[i]) - (x[k] - x[i]) * (y[j] - y[i]);
        if (GE(nz, 0)) continue;
        bool done = false;
        for (int m = 0; m < n; m++)
          if (x[m] - x[i]) * nx + (y[m] - y[i]) * ny + (z[m] - z[i]) * nz > 0) {
            done = true;
            break;
          }
        if (!done) { //handle 4 points on a circle
          point s1[] = { *(lo + i), *(lo + j), *(lo + k), *(lo + i) };
          for (int t = 0; t < (int)res.size(); t++) {
            point s2[] = { res[t].a, res[t].b, res[t].c, res[t].a };
            for (int u = 0; u < 3; u++)
              for (int v = 0; v < 3; v++)
                if (seg_intersection(s1[u], s1[u + 1], s2[v], s2[v + 1]) == 0)
                  goto skip;
          }
          res.push_back((triangle){*(lo + i), *(lo + j), *(lo + k)});
        }
skip:;
      }
    }
  }
  return res;
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  vector<point> v;
  v.push_back(point(1, 3));
  v.push_back(point(1, 2));
  v.push_back(point(2, 1));
  v.push_back(point(0, 0));
  v.push_back(point(-1, 3));
  vector<triangle> dt = delaunay_triangulation(v.begin(), v.end());
  for (int i = 0; i < (int)dt.size(); i++) {
    cout << "Triangle: ";
    cout << "(" << dt[i].a.x << "," << dt[i].a.y << ") ";
    cout << "(" << dt[i].b.x << "," << dt[i].b.y << ") ";
    cout << "(" << dt[i].c.x << "," << dt[i].c.y << ")\n";
  }
  return 0;
}
```

5.4.4 Delaunay Triangulation (Fast)

```
/*

Given a range of points P on the Cartesian plane, the Delaunay
Triangulation of said points is a set of non-overlapping triangles
covering the entire convex hull of P, such that no point in P lies
```

```
within the circumcircle of any of the resulting triangles. The
triangulation maximizes the minimum angle of all the angles of the
triangles in the triangulation. In addition, for any point p in the
convex hull (not necessarily in P), the nearest point is guaranteed
to be a vertex of the enclosing triangle from the triangulation.
See: https://en.wikipedia.org/wiki/Delaunay_triangulation

The triangulation may not exist (e.g. for a set of collinear points)
or it may not be unique (multiple possible triangulations may exist).
The following program assumes that a triangulation exists, and
produces one such valid result. The following is a C++ adaptation of
a FORTRAN90 program, which applies a divide and conquer algorithm
with complex linear-time merging. The original program can be found
via the following link. It contains more thorough documentation,
comments, and debugging messages associated with the current asserts().
http://people.sc.fsu.edu/~burkardt/f_src/table_delaunay/table_delaunay.html

Time Complexity: O(n log n) on the number of input points.

*/

#include <algorithm> /* std::min(), std::max() */
#include <cassert>
#include <cmath>     /* fabs(), sqrt() */
#include <utility>   /* std::pair */
#include <vector>

int wrap(int ival, int ilo, int ihi) {
  int jlo = std::min(ilo, ihi), jhi = std::max(ilo, ihi);
  int wide = jhi + 1 - jlo, res = jlo;
  if (wide != 1)  {
    assert(wide != 0);
    int tmp = (ival - jlo) % wide;
    if (tmp < 0) res += abs(wide);
    res += tmp;
  }
  return res;
}

double epsilon() {
  double r = 1;
  while (1 < (double)(r + 1)) r /= 2;
  return 2 * r;
}

void permute(int n, double a[][2], int p[]) {
  for (int istart = 1; istart <= n; istart++) {
    if (p[istart - 1] < 0) continue;
    if (p[istart - 1] == istart) {
      p[istart - 1] = -p[istart - 1];
      continue;
    }
    double tmp0 = a[istart - 1][0];
    double tmp1 = a[istart - 1][1];
    int iget = istart;
    for (;;) {
      int iput = iget;
      iget = p[iget - 1];
      p[iput - 1] = -p[iput - 1];
      assert(!(iget < 1 || n < iget));
      if (iget == istart) {
        a[iput - 1][0] = tmp0;
        a[iput - 1][1] = tmp1;
        break;
      }
      a[iput - 1][0] = a[iget - 1][0];
      a[iput - 1][1] = a[iget - 1][1];
    }
  }
  for (int i = 0; i < n; i++) p[i] = -p[i];
  return;
}

int * sort_heap(int n, double a[][2]) {
  double aval[2];
  int i, ir, j, l, idxt;
  int *idx;
  if (n < 1) return NULL;
  if (n == 1) {
    idx = new int[1];
    idx[0] = 1;
    return idx;
  }
  idx = new int[n];
  for (int i = 0; i < n; i++) idx[i] = i + 1;
  l = n / 2 + 1;
  ir = n;
  for (;;) {
    if (1 < l) {
      l--;
      idxt = idx[l - 1];
      aval[0] = a[idxt - 1][0];
      aval[1] = a[idxt - 1][1];
    } else {
      idxt = idx[ir - 1];
      aval[0] = a[idxt - 1][0];
      aval[1] = a[idxt - 1][1];
      idx[ir - 1] = idx[0];
      if (--ir == 1) {
        idx[0] = idxt;
        break;
      }
    }
    i = l;
    j = 2 * l;
    while (j <= ir) {
      if (j < ir && (a[idx[j - 1] - 1][0] <  a[idx[j] - 1][0] ||
                     (a[idx[j - 1] - 1][0] == a[idx[j] - 1][0] &&
                      a[idx[j - 1] - 1][1] <  a[idx[j] - 1][1]))) {
        j++;
      }
      if ( aval[0]  < a[idx[j - 1] - 1][0] ||
          (aval[0] == a[idx[j - 1] - 1][0] &&
           aval[1]  < a[idx[j - 1] - 1][1])) {
        idx[i - 1] = idx[j - 1];
        i = j;
        j *= 2;
      } else {
        j = ir + 1;
      }
    }
    idx[i - 1] = idxt;
  }
  return idx;
}

int lrline(double xu, double yu, double xv1, double yv1,
           double xv2, double yv2, double dv) {
  double tol = 1e-7;
  double dx = xv2 - xv1, dy = yv2 - yv1;
  double dxu = xu - xv1, dyu = yu - yv1;
  double t = dy * dxu - dx * dyu + dv * sqrt(dx * dx + dy * dy);
  double tolabs = tol * std::max(std::max(fabs(dx), fabs(dy)),
                         std::max(fabs(dxu), std::max(fabs(dyu), fabs(dv))));
  if (tolabs < t) return 1;
  if (-tolabs <= t) return 0;
  return -1;
}

void vbedg(double x, double y, int point_num, double point_xy[][2],
           int tri_num, int tri_nodes[][3], int tri_neigh[][3],
           int *ltri, int *ledg, int *rtri, int *redg) {
  int a, b;
  double ax, ay, bx, by;
  bool done;
  int e, l, t;
  if (*ltri == 0) {
    done = false;
    *ltri = *rtri;
    *ledg = *redg;
  } else {
    done = true;
  }
  for (;;) {
    l = -tri_neigh[(*rtri) - 1][(*redg) - 1];
    t = l / 3;
    e = l % 3 + 1;
    a = tri_nodes[t - 1][e - 1];
    if (e <= 2) {
      b = tri_nodes[t - 1][e];
    } else {
      b = tri_nodes[t - 1][0];
    }
    ax = point_xy[a - 1][0];
    ay = point_xy[a - 1][1];
    bx = point_xy[b - 1][0];
    by = point_xy[b - 1][1];
    if (lrline(x, y, ax, ay, bx, by, 0.0) <= 0) break;
    *rtri = t;
    *redg = e;
  }
  if (done) return;
  t = *ltri;
  e = *ledg;
  for (;;) {
    b = tri_nodes[t - 1][e - 1];
    e = wrap(e - 1, 1, 3);
    while (0 < tri_neigh[t - 1][e - 1]) {
      t = tri_neigh[t - 1][e - 1];
      if (tri_nodes[t - 1][0] == b) {
        e = 3;
      } else if (tri_nodes[t - 1][1] == b) {
        e = 1;
      } else {
        e = 2;
      }
    }
    a = tri_nodes[t - 1][e - 1];
    ax = point_xy[a - 1][0];
    ay = point_xy[a - 1][1];
    bx = point_xy[b - 1][0];
    by = point_xy[b - 1][1];
    if (lrline(x, y, ax, ay, bx, by, 0.0) <= 0) break;
  }
  *ltri = t;
  *ledg = e;
  return;
}

int diaedg(double x0, double y0, double x1, double y1,
           double x2, double y2, double x3, double y3) {
  double ca, cb, s, tol, tola, tolb;
  int value;
  tol = 100.0 * epsilon();
  double dx10 = x1 - x0, dy10 = y1 - y0;
  double dx12 = x1 - x2, dy12 = y1 - y2;
  double dx30 = x3 - x0, dy30 = y3 - y0;
  double dx32 = x3 - x2, dy32 = y3 - y2;
  tola = tol * std::max(std::max(fabs(dx10), fabs(dy10)),
                        std::max(fabs(dx30), fabs(dy30)));
  tolb = tol * std::max(std::max(fabs(dx12), fabs(dy12)),
                        std::max(fabs(dx32), fabs(dy32)));
  ca = dx10 * dx30 + dy10 * dy30;
  cb = dx12 * dx32 + dy12 * dy32;
  if (tola < ca && tolb < cb) {
    value = -1;
  } else if (ca < -tola && cb < -tolb) {
    value = 1;
  } else {
    tola = std::max(tola, tolb);
    s = (dx10 * dy30 - dx30 * dy10) * cb + (dx32 * dy12 - dx12 * dy32) * ca;
    if (tola < s) {
      value = -1;
    } else if (s < -tola) {
      value = 1;
    } else {
      value = 0;
    }
  }
  return value;
}

int swapec(int i, int *top, int *btri, int *bedg,
           int point_num, double point_xy[][2],
           int tri_num, int tri_nodes[][3], int tri_neigh[][3], int stack[]) {
  int a, b, c, e, ee, em1, ep1, f, fm1, fp1, l, r, s, swap, t, tt, u;
  double x = point_xy[i - 1][0];
  double y = point_xy[i - 1][1];
  for (;;) {
    if (*top <= 0) break;
    t = stack[*top - 1];
    *top = *top - 1;
    if (tri_nodes[t - 1][0] == i) {
      e = 2;
      b = tri_nodes[t - 1][2];
    } else if (tri_nodes[t - 1][1] == i) {
      e = 3;
      b = tri_nodes[t - 1][0];
    } else {
      e = 1;
      b = tri_nodes[t - 1][1];
    }
    a = tri_nodes[t - 1][e - 1];
    u = tri_neigh[t - 1][e - 1];
    if (tri_neigh[u - 1][0] == t) {
      f = 1;
      c = tri_nodes[u - 1][2];
    } else if (tri_neigh[u - 1][1] == t) {
      f = 2;
      c = tri_nodes[u - 1][0];
    } else {
      f = 3;
      c = tri_nodes[u - 1][1];
    }
    swap = diaedg(x, y, point_xy[a - 1][0], point_xy[a - 1][1],
                        point_xy[c - 1][0], point_xy[c - 1][1],
                        point_xy[b - 1][0], point_xy[b - 1][1]);
    if (swap == 1) {
      em1 = wrap(e - 1, 1, 3);
      ep1 = wrap(e + 1, 1, 3);
      fm1 = wrap(f - 1, 1, 3);
      fp1 = wrap(f + 1, 1, 3);
      tri_nodes[t - 1][ep1 - 1] = c;
      tri_nodes[u - 1][fp1 - 1] = i;
      r = tri_neigh[t - 1][ep1 - 1];
      s = tri_neigh[u - 1][fp1 - 1];
      tri_neigh[t - 1][ep1 - 1] = u;
      tri_neigh[u - 1][fp1 - 1] = t;
      tri_neigh[t - 1][e - 1] = s;
      tri_neigh[u - 1][f - 1] = r;
      if (0 < tri_neigh[u - 1][fm1 - 1]) {
        *top = *top + 1;
        stack[*top - 1] = u;
      }
      if (0 < s) {
        if (tri_neigh[s - 1][0] == u) {
          tri_neigh[s - 1][0] = t;
        } else if (tri_neigh[s - 1][1] == u) {
          tri_neigh[s - 1][1] = t;
        } else {
          tri_neigh[s - 1][2] = t;
        }
        *top = *top + 1;
        if (point_num < *top) return 8;
        stack[*top - 1] = t;
      } else {
        if (u == *btri && fp1 == *bedg) {
          *btri = t;
          *bedg = e;
        }
        l = - (3 * t + e - 1);
        tt = t;
        ee = em1;
        while (0 < tri_neigh[tt - 1][ee - 1]) {
          tt = tri_neigh[tt - 1][ee - 1];
          if (tri_nodes[tt - 1][0] == a) {
            ee = 3;
          } else if (tri_nodes[tt - 1][1] == a) {
            ee = 1;
          } else {
            ee = 2;
          }
        }
        tri_neigh[tt - 1][ee - 1] = l;
      }
      if (0 < r) {
        if (tri_neigh[r - 1][0] == t) {
          tri_neigh[r - 1][0] = u;
        } else if (tri_neigh[r - 1][1] == t) {
          tri_neigh[r - 1][1] = u;
        } else {
          tri_neigh[r - 1][2] = u;
        }
      } else {
        if (t == *btri && ep1 == *bedg) {
          *btri = u;
          *bedg = f;
        }
        l = -(3 * u + f - 1);
        tt = u;
        ee = fm1;
        while (0 < tri_neigh[tt - 1][ee - 1]) {
          tt = tri_neigh[tt - 1][ee - 1];
          if (tri_nodes[tt - 1][0] == b) {
            ee = 3;
          } else if (tri_nodes[tt - 1][1] == b) {
            ee = 1;
          } else {
            ee = 2;
          }
        }
        tri_neigh[tt - 1][ee - 1] = l;
      }
    }
  }
  return 0;
}

void perm_inv(int n, int p[]) {
  int i, i0, i1, i2;
  assert(n > 0);
  for (i = 1; i <= n; i++) {
    i1 = p[i - 1];
    while (i < i1) {
      i2 = p[i1 - 1];
      p[i1 - 1] = -i2;
      i1 = i2;
    }
    p[i - 1] = -p[i - 1];
  }
  for (i = 1; i <= n; i++) {
    i1 = -p[i - 1];
    if (0 <= i1) {
      i0 = i;
      for (;;) {
        i2 = p[i1 - 1];
        p[i1 - 1] = i0;
        if (i2 < 0) break;
        i0 = i1;
        i1 = i2;
      }
    }
  }
  return;
}

int dtris2(int point_num, double point_xy[][2],
           int tri_nodes[][3], int tri_neigh[][3]) {
  double cmax;
  int e, error;
  int i, j, k, l, m, m1, m2, n;
  int ledg, lr, ltri, redg, rtri, t, top;
  double tol;
  int *stack = new int[point_num];
  tol = 100.0 * epsilon();
  int *idx = sort_heap(point_num, point_xy);
  permute(point_num, point_xy, idx);
  m1 = 0;
  for (i = 1; i < point_num; i++) {
    m = m1;
    m1 = i;
    k = -1;
    for (j = 0; j <= 1; j++) {
      cmax = std::max(fabs(point_xy[m][j]), fabs(point_xy[m1][j]));
      if (tol * (cmax + 1.0) < fabs(point_xy[m][j] - point_xy[m1][j])) {
        k = j;
        break;
      }
    }
    assert(k != -1);
  }
  m1 = 1;
  m2 = 2;
  j = 3;
  for (;;) {
    assert(point_num >= j);
    m = j;
    lr = lrline(point_xy[m - 1][0], point_xy[m - 1][1],
                point_xy[m1 - 1][0], point_xy[m1 - 1][1],
                point_xy[m2 - 1][0], point_xy[m2 - 1][1], 0.0);
    if (lr != 0) break;
    j++;
  }
  int tri_num = j - 2;
  if (lr == -1) {
    tri_nodes[0][0] = m1;
    tri_nodes[0][1] = m2;
    tri_nodes[0][2] = m;
    tri_neigh[0][2] = -3;
    for (i = 2; i <= tri_num; i++) {
      m1 = m2;
      m2 = i + 1;
      tri_nodes[i - 1][0] = m1;
      tri_nodes[i - 1][1] = m2;
      tri_nodes[i - 1][2] = m;
      tri_neigh[i - 1][0] = -3 * i;
      tri_neigh[i - 1][1] = i;
      tri_neigh[i - 1][2] = i - 1;
    }
    tri_neigh[tri_num - 1][0] = -3 * tri_num - 1;
    tri_neigh[tri_num - 1][1] = -5;
    ledg = 2;
    ltri = tri_num;
  } else {
    tri_nodes[0][0] = m2;
    tri_nodes[0][1] = m1;
    tri_nodes[0][2] = m;
    tri_neigh[0][0] = -4;
    for (i = 2; i <= tri_num; i++) {
      m1 = m2;
      m2 = i+1;
      tri_nodes[i - 1][0] = m2;
      tri_nodes[i - 1][1] = m1;
      tri_nodes[i - 1][2] = m;
      tri_neigh[i - 2][2] = i;
      tri_neigh[i - 1][0] = -3 * i - 3;
      tri_neigh[i - 1][1] = i - 1;
    }
    tri_neigh[tri_num - 1][2] = -3 * (tri_num);
    tri_neigh[0][1] = -3 * (tri_num) - 2;
    ledg = 2;
    ltri = 1;
  }
  top = 0;
  for (i = j + 1; i <= point_num; i++) {
    m = i;
    m1 = tri_nodes[ltri - 1][ledg - 1];
    if (ledg <= 2) {
      m2 = tri_nodes[ltri - 1][ledg];
    } else {
      m2 = tri_nodes[ltri - 1][0];
    }
    lr = lrline(point_xy[m - 1][0], point_xy[m - 1][1],
                point_xy[m1 - 1][0], point_xy[m1 - 1][1],
                point_xy[m2 - 1][0], point_xy[m2 - 1][1], 0.0);
    if (0 < lr) {
      rtri = ltri;
      redg = ledg;
      ltri = 0;
    } else {
      l = -tri_neigh[ltri - 1][ledg - 1];
      rtri = l / 3;
      redg = (l % 3) + 1;
    }
    vbedg(point_xy[m - 1][0], point_xy[m - 1][1],
          point_num, point_xy, tri_num, tri_nodes, tri_neigh,
          &ltri, &ledg, &rtri, &redg);
    n = tri_num + 1;
    l = -tri_neigh[ltri - 1][ledg - 1];
    for (;;) {
      t = l / 3;
      e = (l % 3) + 1;
      l = -tri_neigh[t - 1][e - 1];
      m2 = tri_nodes[t - 1][e - 1];
      if (e <= 2) {
        m1 = tri_nodes[t - 1][e];
      } else {
        m1 = tri_nodes[t - 1][0];
      }
      tri_num++;
      tri_neigh[t - 1][e - 1] = tri_num;
      tri_nodes[tri_num - 1][0] = m1;
      tri_nodes[tri_num - 1][1] = m2;
      tri_nodes[tri_num - 1][2] = m;
      tri_neigh[tri_num - 1][0] = t;
      tri_neigh[tri_num - 1][1] = tri_num - 1;
      tri_neigh[tri_num - 1][2] = tri_num + 1;
      top++;
      assert(point_num >= top);
      stack[top - 1] = tri_num;
      if (t == rtri && e == redg) break;
    }
    tri_neigh[ltri - 1][ledg - 1] = -3 * n - 1;
    tri_neigh[n - 1][1] = -3 * tri_num - 2;
    tri_neigh[tri_num - 1][2] = -l;
    ltri = n;
    ledg = 2;
    error = swapec(m, &top, &ltri, &ledg, point_num, point_xy,
                   tri_num, tri_nodes, tri_neigh, stack);
    assert(error == 0);
  }
  for (i = 0; i < 3; i++)
    for (j = 0; j < tri_num; j++)
      tri_nodes[j][i] = idx[tri_nodes[j][i] - 1];
  perm_inv(point_num, idx);
  permute(point_num, point_xy, idx);
  delete[] idx;
  delete[] stack;
  return tri_num;
}

/*** C++ Wrapper ***/

typedef std::pair<double, double> point;
#define x first
#define y second

struct triangle { point a, b, c; };

template<class It>
std::vector<triangle> delaunay_triangulation(It lo, It hi) {
  int n = hi - lo;
  double points[n][2];
  int tri_nodes[3 * n][3], tri_neigh[3 * n][3];
  int curr = 0;
  for (It it = lo; it != hi; ++curr, ++it) {
    points[curr][0] = it->x;
    points[curr][1] = it->y;
  }
  int m = dtris2(n, points, tri_nodes, tri_neigh);
  std::vector<triangle> res;
  for (int i = 0; i < m; i++)
    res.push_back((triangle){*(lo + (tri_nodes[i][0] - 1)),
                             *(lo + (tri_nodes[i][1] - 1)),
                             *(lo + (tri_nodes[i][2] - 1))});
  return res;
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  vector<point> v;
  v.push_back(point(1, 3));
  v.push_back(point(1, 2));
  v.push_back(point(2, 1));
  v.push_back(point(0, 0));
  v.push_back(point(-1, 3));
  vector<triangle> dt = delaunay_triangulation(v.begin(), v.end());
  for (int i = 0; i < (int)dt.size(); i++) {
    cout << "Triangle: ";
    cout << "(" << dt[i].a.x << "," << dt[i].a.y << ") ";
    cout << "(" << dt[i].b.x << "," << dt[i].b.y << ") ";
    cout << "(" << dt[i].c.x << "," << dt[i].c.y << ")\n";
  }
  return 0;
}
```

Chapter 6

Strings

6.1 Strings Toolbox

```
/*

Useful or trivial string operations. These functions are not particularly
algorithmic. They are typically naive implementations using C++ features.
They depend on many features of the C++ <string> library, which tend to
have an unspecified complexity. They may not be optimally efficient.

*/

#include <cstdlib>
#include <sstream>
#include <string>
#include <vector>

//integer to string conversion and vice versa using C++ features

//note that a similar std::to_string is introduced in C++0x
template<class Int>
std::string to_string(const Int & i) {
  std::ostringstream oss;
  oss << i;
  return oss.str();
}

//like atoi, except during special cases like overflows
int to_int(const std::string & s) {
  std::istringstream iss(s);
  int res;
  if (!(iss >> res)) /* complain */;
  return res;
}

/*

itoa implementation (fast)
documentation: http://www.cplusplus.com/reference/cstdlib/itoa/
taken from: http://www.jb.man.ac.uk/~slowe/cpp/itoa.html
```

```

*/

char* itoa(int value, char * str, int base = 10) {
  if (base < 2 || base > 36) {
    *str = '\0';
    return str;
  }
  char *ptr = str, *ptr1 = str, tmp_c;
  int tmp_v;
  do {
    tmp_v = value;
    value /= base;
    *ptr++ = "zyxwvutsrqponmlkjihgfedcba9876543210123456789"
             "abcdefghijklmnopqrstuvwxyz"[35 + (tmp_v - value * base)];
  } while (value);
  if (tmp_v < 0) *ptr++ = '-';
  for (*ptr-- = '\0'; ptr1 < ptr; *ptr1++ = tmp_c) {
    tmp_c = *ptr;
    *ptr-- = *ptr1;
  }
  return str;
}

/*

Trimming functions (in place). Given a string and optionally a series
of characters to be considered for trimming, trims the string's ends
(left, right, or both) and returns the string. Note that the ORIGINAL
string is trimmed as it's passed by reference, despite the original
reference being returned for convenience.

*/

std::string& ltrim(std::string & s, const std::string & delim = "␣\n\t\v\f\r") {
  unsigned int pos = s.find_first_not_of(delim);
  if (pos != std::string::npos) s.erase(0, pos);
  return s;
}

std::string& rtrim(std::string & s, const std::string & delim = "␣\n\t\v\f\r") {
  unsigned int pos = s.find_last_not_of(delim);
  if (pos != std::string::npos) s.erase(pos);
  return s;
}

std::string& trim(std::string & s, const std::string & delim = "␣\n\t\v\f\r") {
  return ltrim(rtrim(s));
}

/*

Returns a copy of the string s with all occurrences of the given
string search replaced with the given string replace.

Time Complexity: Unspecified, but proportional to the number of times
the search string occurs and the complexity of std::string::replace,
which is unspecified.

```

```cpp
*/

std::string replace(std::string s,
                    const std::string & search,
                    const std::string & replace) {
  if (search.empty()) return s;
  unsigned int pos = 0;
  while ((pos = s.find(search, pos)) != std::string::npos) {
    s.replace(pos, search.length(), replace);
    pos += replace.length();
  }
  return s;
}

/*

Tokenizes the string s based on single character delimiters.

Version 1: Simpler. Only one delimiter character allowed, and this will
not skip empty tokens.
  e.g. split("a::b", ":") yields {"a", "b"}, not {"a", "", "b"}.

Version 2: All of the characters in the delim parameter that also exists
in s will be removed from s, and the token(s) of s that are left over will
be added sequentially to a vector and returned. Empty tokens are skipped.
  e.g. split("a::b", ":") yields {"a", "b"}, not {"a", "", "b"}.

Time Complexity: O(s.length() * delim.length())

*/

std::vector<std::string> split(const std::string & s, char delim) {
  std::vector<std::string> res;
  std::stringstream ss(s);
  std::string curr;
  while (std::getline(ss, curr, delim))
    res.push_back(curr);
  return res;
}

std::vector<std::string> split(const std::string & s,
                               const std::string & delim = "␣\n\t\v\f\r") {
  std::vector<std::string> res;
  std::string curr;
  for (int i = 0; i < (int)s.size(); i++) {
    if (delim.find(s[i]) == std::string::npos) {
      curr += s[i];
    } else if (!curr.empty()) {
      res.push_back(curr);
      curr = "";
    }
  }
  if (!curr.empty()) res.push_back(curr);
  return res;
}

/*

Like the explode() function in PHP, the string s is tokenized based
```

```
on delim, which is considered as a whole boundary string, not just a
sequence of possible boundary characters like the split() function above.
This will not skip empty tokens.
  e.g. explode("a::b", ":") yields {"a", "", "b"}, not {"a", "b"}.

Time Complexity: O(s.length() * delim.length())

*/

std::vector<std::string> explode(const std::string & s,
                                 const std::string & delim) {
  std::vector<std::string> res;
  unsigned int last = 0, next = 0;
  while ((next = s.find(delim, last)) != std::string::npos) {
    res.push_back(s.substr(last, next - last));
    last = next + delim.size();
  }
  res.push_back(s.substr(last));
  return res;
}

/*** Example Usage ***/

#include <cassert>
#include <cstdio>
#include <iostream>
using namespace std;

void print(const vector<string> & v) {
  cout << "[";
  for (int i = 0; i < (int)v.size(); i++)
    cout << (i ? "\",␣\"" : "\"") << v[i];
  cout << "\"]\n";
}

int main() {
  assert(to_string(123) + "4" == "1234");
  assert(to_int("1234") == 1234);
  char buffer[50];
  assert(string(itoa(1750, buffer, 10)) == "1750");
  assert(string(itoa(1750, buffer, 16)) == "6d6");
  assert(string(itoa(1750, buffer, 2)) == "11011010110");

  string s("␣␣␣abc␣\n");
  string t = s;
  assert(ltrim(s) == "abc␣\n");
  assert(rtrim(s) == trim(t));
  assert(replace("abcdabba", "ab", "00") == "00cd00ba");

  vector<string> tokens;

  tokens = split("a\nb\ncde\nf", '\n');
  cout << "split␣v1:␣";
  print(tokens); //["a", "b", "cde", "f"]

  tokens = split("a::b,cde:,f", ":,");
  cout << "split␣v2:␣";
  print(tokens); //["a", "b", "cde", "f"]

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

6.2 Expression Parsing

6.2.1 Recursive Descent

```
/*

Evaluate a mathematical expression in accordance to the order
of operations (parentheses, exponents, multiplication, division,
addition, subtraction). Does not handle unary operators like '-'.

*/

/*** Example Usage ***/

#include <cctype>
#include <cmath>
#include <sstream>
#include <stdexcept>
#include <string>

class parser {
  int pos;
  double tokval;
  std::string s;

  bool is_dig_or_dot(char c) {
    return isdigit(c) || c == '.';
  }

  double to_double(const std::string & s) {
    std::stringstream ss(s);
    double res;
    ss >> res;
    return res;
  }

 public:
  char token;

  parser(const std::string & s) {
    this->s = s;
    pos = 0;
  }

  int next() {
    for (;;) {
      if (pos == (int)s.size())
        return token = -1;
      char c = s[pos++];
      if (std::string("+-*/^()\n").find(c) != std::string::npos)
```

```
        return token = c;
      if (isspace(c)) continue;
      if (isdigit(c) || c == '.') {
        std::string operand(1, c);
        while (pos < (int)s.size() && is_dig_or_dot(s[pos]))
          operand += (c = s[pos++]);
        tokval = to_double(operand);
        return token = 'n';
      }
      throw std::runtime_error(std::string("Bad␣character:␣") + c);
    }
  }

  void skip(int ch) {
    if (token != ch)
      throw std::runtime_error(std::string("Bad␣character:␣") + token + std::string(",␣expected:␣") +
    (char)ch);
    next();
  }

  double number() {
    if (token == 'n') {
      double v = tokval;
      skip('n');
      return v;
    }
    skip('(');
    double v = expression();
    skip(')');
    return v;
  }

  // factor ::= number | number '^' factor
  double factor() {
    double v = number();
    if (token == '^') {
      skip('^');
      v = pow(v, factor());
    }
    return v;
  }

  // term ::= factor | term '*' factor | term '/' factor
  double term() {
    double v = factor();
    for (;;) {
      if (token == '*') {
        skip('*');
        v *= factor();
      } else if (token == '/') {
        skip('/');
        v /= factor();
      } else {
        return v;
      }
    }
  }

  // expression ::= term | expression '+' term | expression '-' term
  double expression() {
    double v = term();
    for (;;) {
      if (token == '+') {
        skip('+');
        v += term();
      } else if (token == '-') {
        skip('-');
        v -= term();
      } else {
        return v;
      }
    }
  }
};

#include <iostream>
using namespace std;

int main() {
  parser p("1+2*3*4+3*(2+2)-100\n");
  p.next();
  while (p.token != -1) {
    if (p.token == '\n') {
      p.skip('\n');
      continue;
    }
    cout << p.expression() << "\n";
  }
  return 0;
}
```

6.2.2 Recursive Descent (Simple)

```
/*

Evaluate a mathematica expression in accordance to the order
of operations (parentheses, exponents, multiplication, division,
addition, subtraction). This handles unary operators like '-'.

*/

#include <string>

template<class It> int eval(It & it, int prec) {
  if (prec == 0) {
    int sign = 1, ret = 0;
    for (; *it == '-'; it++) sign *= -1;
    if (*it == '(') {
      ret = eval(++it, 2);
      it++;
    } else while (*it >= '0' && *it <= '9') {
      ret = 10 * ret + (*(it++) - '0');
    }
    return sign * ret;
  }
  int num = eval(it, prec - 1);
  while (!((prec == 2 && *it != '+' && *it != '-') ||
           (prec == 1 && *it != '*' && *it != '/'))) {
    switch (*(it++)) {
      case '+': num += eval(it, prec - 1); break;
      case '-': num -= eval(it, prec - 1); break;
      case '*': num *= eval(it, prec - 1); break;
      case '/': num /= eval(it, prec - 1); break;
    }
  }
  return num;
}

/*** Wrapper Function ***/

int eval(const std::string & s) {
  std::string::iterator it = std::string(s).begin();
  return eval(it, 2);
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  cout << eval("1+2*3*4+3*(2+2)-100") << "\n";
  return 0;
}
```

6.2.3 Shunting Yard Algorithm

```
/*

Evaluate a mathematica expression in accordance to the order
of operations (parentheses, exponents, multiplication, division,
addition, subtraction). This also handles unary operators like '-'.
We use strings for operators so we can even define things like "sqrt"
and "mod" as unary operators by changing prec() and split_expr()
accordingly.

Time Complexity: O(n) on the total number of operators and operands.

*/

#include <cstdlib>   /* strtol() */
#include <stack>
#include <stdexcept> /* std::runtime_error */
#include <string>
#include <vector>

// Classify the precedences of operators here.
inline int prec(const std::string & op, bool unary) {
  if (unary) {
    if (op == "+" || op == "-") return 3;
    return 0; // not a unary operator
  }
  if (op == "*" || op == "/") return 2;
```

```cpp
  if (op == "+" || op == "-") return 1;
  return 0;  // not a binary operator
}

inline int calc1(const std::string & op, int val) {
  if (op == "+") return +val;
  if (op == "-") return -val;
  throw std::runtime_error("Invalid␣unary␣operator:␣" + op);
}

inline int calc2(const std::string & op, int L, int R) {
  if (op == "+") return L + R;
  if (op == "-") return L - R;
  if (op == "*") return L * R;
  if (op == "/") return L / R;
  throw std::runtime_error("Invalid␣binary␣operator:␣" + op);
}

inline bool is_operand(const std::string & s) {
  return s != "(" && s != ")" && !prec(s, 0) && !prec(s, 1);
}

int eval(std::vector<std::string> E) { // E stores the tokens
  E.insert(E.begin(), "(");
  E.push_back(")");
  std::stack<std::pair<std::string, bool> > ops;
  std::stack<int> vals;
  for (int i = 0; i < (int)E.size(); i++) {
    if (is_operand(E[i])) {
      vals.push(strtol(E[i].c_str(), 0, 10)); // convert to int
      continue;
    }
    if (E[i] == "(") {
      ops.push(std::make_pair("(", 0));
      continue;
    }
    if (prec(E[i], 1) && (i == 0 || E[i - 1] == "(" || prec(E[i - 1], 0))) {
      ops.push(std::make_pair(E[i], 1));
      continue;
    }
    while(prec(ops.top().first, ops.top().second) >= prec(E[i], 0)) {
      std::string op = ops.top().first;
      bool is_unary = ops.top().second;
      ops.pop();
      if (op == "(") break;
      int y = vals.top(); vals.pop();
      if (is_unary) {
        vals.push(calc1(op, y));
      } else {
        int x = vals.top(); vals.pop();
        vals.push(calc2(op, x, y));
      }
    }
    if (E[i] != ")") ops.push(std::make_pair(E[i], 0));
  }
  return vals.top();
}

/*
```

```

Split a string expression to tokens, ignoring whitespace delimiters.
A vector of tokens is a more flexible format since you can decide to
parse the expression however you wish just by modifying this function.
e.g. "1+(51 * -100)" converts to {"1","+","(","51","*","-","100",")"}

*/

std::vector<std::string> split_expr(const std::string &s,
                const std::string &delim = "␣\n\t\v\f\r") {
  std::vector<std::string> ret;
  std::string acc = "";
  for (int i = 0; i < (int)s.size(); i++)
    if (s[i] >= '0' && s[i] <= '9') {
      acc += s[i];
    } else {
      if (i > 0 && s[i - 1] >= '0' && s[i - 1] <= '9')
        ret.push_back(acc);
      acc = "";
      if (delim.find(s[i]) != std::string::npos) continue;
      ret.push_back(std::string("") + s[i]);
    }
  if (s[s.size() - 1] >= '0' && s[s.size() - 1] <= '9')
    ret.push_back(acc);
  return ret;
}

int eval(const std::string & s) {
  return eval(split_expr(s));
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  cout << eval("1+2*3*4+3*(2+2)-100") << endl;
  return 0;
}
```

6.3 String Searching

6.3.1 Longest Common Substring

```
/*

Given an text and a pattern to be searched for within the text,
determine the first position in which the pattern occurs in
the text. The KMP algorithm is much faster than the naive,
quadratic time, string searching algorithm that is found in
string.find() in the C++ standard library.

KMP generates a table using a prefix function of the pattern.
Then, the precomputed table of the pattern can be used indefinitely
for any number of texts.
```

```

Time Complexity: O(n + m) where n is the length of the text
and m is the length of the pattern.

Space Complexity: O(m) auxiliary on the length of the pattern.

*/

#include <string>
#include <vector>

int find(const std::string & text, const std::string & pattern) {
  if (pattern.empty()) return 0;
  //generate table using pattern
  std::vector<int> p(pattern.size());
  for (int i = 0, j = p[0] = -1; i < (int)pattern.size(); ) {
    while (j >= 0 && pattern[i] != pattern[j])
      j = p[j];
    i++;
    j++;
    p[i] = (pattern[i] == pattern[j]) ? p[j] : j;
  }
  //use the precomputed table to search within text
  //the following can be repeated on many different texts
  for (int i = 0, j = 0; j < (int)text.size(); ) {
    while (i >= 0 && pattern[i] != text[j])
      i = p[i];
    i++;
    j++;
    if (i >= (int)pattern.size())
      return j - i;
  }
  return std::string::npos;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  assert(15 == find("ABC␣ABCDAB␣ABCDABCDABDE", "ABCDABD"));
  return 0;
}
```

6.3.2 Longest Common Subsequence

```
/*

Given a text and multiple patterns to be searched for within the
text, simultaneously determine the position of all matches.
All of the patterns will be first required for precomputing
the automata, after which any input text may be given without
having to recompute the automata for the pattern.

Time Complexity: O(n) for build_automata(), where n is the sum of
all pattern lengths, and O(1) amortized for next_state(). However,
since it must be called m times for an input text of length m, and
```

```
if there are z matches throughout the entire text, then the entire
algorithm will have a running time of O(n + m + z).

Note that in this implementation, a bitset is used to speed up
build_automata() at the cost of making the later text search cost
O(n * m). To truly make the algorithm O(n + m + z), bitset must be
substituted for an unordered_set, which will not encounter any
blank spaces during iteration of the bitset. However, for simply
counting the number of matches, bitsets are clearly advantages.

Space Complexity: O(l * c), where l is the sum of all pattern
lengths and c is the size of the alphabet.

*/

#include <bitset>
#include <cstring>
#include <queue>
#include <string>
#include <vector>

const int MAXP = 1000;  //maximum number of patterns
const int MAXL = 10000; //max possible sum of all pattern lengths
const int MAXC = 26;    //size of the alphabet (e.g. 'a'..'z')

//This function should be customized to return a mapping from
//the input alphabet (e.g. 'a'..'z') to the integers 0..MAXC-1
inline int map_alphabet(char c) {
  return (int)(c - 'a');
}

std::bitset<MAXP> out[MAXL];  //std::unordered_set<int> out[MAXL]
int fail[MAXL], g[MAXL][MAXC + 1];

int build_automata(const std::vector<std::string> & patterns) {
  memset(fail, -1, sizeof fail);
  memset(g, -1, sizeof g);
  for (int i = 0; i < MAXL; i++)
    out[i].reset();  //out[i].clear();
  int states = 1;
  for (int i = 0; i < (int)patterns.size(); i++) {
    const std::string & pattern = patterns[i];
    int curr = 0;
    for (int j = 0; j < (int)pattern.size(); j++) {
      int c = map_alphabet(pattern[j]);
      if (g[curr][c] == -1)
        g[curr][c] = states++;
      curr = g[curr][c];
    }
    out[curr][i] = out[curr][i] | 1;  //out[curr].insert(i);
  }
  for (int c = 0; c < MAXC; c++)
    if (g[0][c] == -1) g[0][c] = 0;
  std::queue<int> q;
  for (int c = 0; c <= MAXC; c++) {
    if (g[0][c] != -1 && g[0][c] != 0) {
      fail[g[0][c]] = 0;
      q.push(g[0][c]);
    }
  }
  while (!q.empty()) {
    int s = q.front(), t;
    q.pop();
    for (int c = 0; c <= MAXC; c++) {
      t = g[s][c];
      if (t != -1) {
        int f = fail[s];
        while (g[f][c] == -1)
          f = fail[f];
        f = g[f][c];
        fail[t] = f;
        out[t] |= out[f];  //out[t].insert(out[f].begin(), out[f].end());
        q.push(t);
      }
    }
  }
  return states;
}

int next_state(int curr, char ch) {
  int next = curr, c = map_alphabet(ch);
  while (g[next][c] == -1)
    next = fail[next];
  return g[next][c];
}

/*** Example Usage (en.wikipedia.org/wiki/AhoCorasick_algorithm) ***/

#include <iostream>
using namespace std;

int main() {
  vector<string> patterns;
  patterns.push_back("a");
  patterns.push_back("ab");
  patterns.push_back("bab");
  patterns.push_back("bc");
  patterns.push_back("bca");
  patterns.push_back("c");
  patterns.push_back("caa");
  build_automata(patterns);

  string text("abccab");
  int state = 0;
  for (int i = 0; i < (int)text.size(); i++) {
    state = next_state(state, text[i]);
    cout << "Matches␣ending␣at␣position␣" << i << ":" << endl;
    if (out[state].any())
      for (int j = 0; j < (int)out[state].size(); j++)
        if (out[state][j])
          cout << "'" << patterns[j] << "'" << endl;
  }
  return 0;
}
```

6.3.3 Edit Distance

```cpp
/*

Given an text and a pattern to be searched for within the text,
determine the positions of all patterns within the text. This
is as efficient as KMP, but does so through computing the
"Z function." For a string S, Z[i] stores the length of the longest
substring starting from S[i] which is also a prefix of S, i.e. the
maximum k such that S[j]= S [ i + j ] for all 0 <=  j  < k .

Time Complexity: O(n + m) where n is the length of the text
and m is the length of the pattern.

Space Complexity: O(m) auxiliary on the length of the pattern.

*/

#include <algorithm>
#include <string>
#include <vector>

std::vector<int> z_function(const std::string & s) {
  std::vector<int> z(s.size());
  for (int i = 1, l = 0, r = 0; i < (int)z.size(); i++) {
    if (i <= r)
      z[i] = std::min(r - i + 1, z[i - l]);
    while (i + z[i] < (int)z.size() && s[z[i]] == s[i + z[i]])
      z[i]++;
    if (r < i + z[i] - 1) {
      l = i;
      r = i + z[i] - 1;
    }
  }
  return z;
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  string text = "abcabaaaababab";
  string pattern = "aba";
  vector<int> z = z_function(pattern + "$" + text);
  for (int i = (int)pattern.size() + 1; i < (int)z.size(); i++) {
    if (z[i] == (int)pattern.size())
      cout << "Pattern␣found␣starting␣at␣index␣"
           << (i - (int)pattern.size() - 1) << "." << endl;
  }
  return 0;
}
```

6.4 Dynamic Programming

6.4.1 Longest Common Substring

```
/*

A substring is a consecutive part of a longer string (e.g. "ABC" is
a substring of "ABCDE" but "ABD" is not). Using dynamic programming,
determine the longest string which is a substring common to any two
input strings.

Time Complexity: O(n * m) where n and m are the lengths of the two
input strings, respectively.

Space Complexity: O(min(n, m)) auxiliary.

*/

#include <string>

std::string longest_common_substring
(const std::string & s1, const std::string & s2) {
  if (s1.empty() || s2.empty()) return "";
  if (s1.size() < s2.size())
    return longest_common_substring(s2, s1);
  int * A = new int[s2.size()];
  int * B = new int[s2.size()];
  int startpos = 0, maxlen = 0;
  for (int i = 0; i < (int)s1.size(); i++) {
    for (int j = 0; j < (int)s2.size(); j++) {
      if (s1[i] == s2[j]) {
        A[j] = (i > 0 && j > 0) ? 1 + B[j - 1] : 1;
        if (maxlen < A[j]) {
          maxlen = A[j];
          startpos = i - A[j] + 1;
        }
      } else {
        A[j] = 0;
      }
    }
    int * temp = A;
    A = B;
    B = temp;
  }
  delete[] A;
  delete[] B;
  return s1.substr(startpos, maxlen);
}

/*** Example Usage ***/

#include <cassert>

int main() {
  assert(longest_common_substring("bbbabca", "aababcd") == "babc");
  return 0;
}
```

6.4.2 Longest Common Subsequence

```
/*
```

```

A subsequence is a sequence that can be derived from another sequence
by deleting some elements without changing the order of the remaining
elements (e.g. "ACE" is a subsequence of "ABCDE", but "BAE" is not).
Using dynamic programming, determine the longest string which
is a subsequence common to any two input strings.

In addition, the shortest common supersequence between two strings is
a closely related problem, which involves finding the shortest string
which has both input strings as subsequences (e.g. "ABBC" and "BCB" has
the shortest common supersequence of "ABBCB"). The answer is simply:
  (sum of lengths of s1 and s2) - (length of LCS of s1 and s2)

Time Complexity: O(n * m) where n and m are the lengths of the two
input strings, respectively.

Space Complexity: O(n * m) auxiliary.

*/

#include <string>
#include <vector>

std::string longest_common_subsequence
(const std::string & s1, const std::string & s2) {
  int n = s1.size(), m = s2.size();
  std::vector< std::vector<int> > dp;
  dp.resize(n + 1, std::vector<int>(m + 1, 0));
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (s1[i] == s2[j]) {
        dp[i + 1][j + 1] = dp[i][j] + 1;
      } else if (dp[i + 1][j] > dp[i][j + 1]) {
        dp[i + 1][j + 1] = dp[i + 1][j];
      } else {
        dp[i + 1][j + 1] = dp[i][j + 1];
      }
    }
  }
  std::string ret;
  for (int i = n, j = m; i > 0 && j > 0; ) {
    if (s1[i - 1] == s2[j - 1]) {
      ret = s1[i - 1] + ret;
      i--;
      j--;
    } else if (dp[i - 1][j] < dp[i][j - 1]) {
      j--;
    } else {
      i--;
    }
  }
  return ret;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  assert(longest_common_subsequence("xmjyauz", "mzjawxu") == "mjau");
  return 0;
}
```

6.4.3 Edit Distance

```
/*

Given two strings s1 and s2, the edit distance between them is the
minimum number of operations required to transform s1 into s2,
where each operation can be any one of the following:
  - insert a letter anywhere into the current string
  - delete any letter from the current string
  - replace any letter of the current string with any other letter

Time Complexity: O(n * m) where n and m are the lengths of the two
input strings, respectively.

Space Complexity: O(n * m) auxiliary.

*/

#include <algorithm>
#include <string>
#include <vector>

int edit_distance(const std::string & s1, const std::string & s2) {
  int n = s1.size(), m = s2.size();
  std::vector< std::vector<int> > dp;
  dp.resize(n + 1, std::vector<int>(m + 1, 0));
  for (int i = 0; i <= n; i++) dp[i][0] = i;
  for (int j = 0; j <= m; j++) dp[0][j] = j;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (s1[i] == s2[j]) {
        dp[i + 1][j + 1] = dp[i][j];
      } else {
        dp[i + 1][j + 1] = 1 + std::min(dp[i][j],        //replace
                               std::min(dp[i + 1][j],    //insert
                                        dp[i][j + 1]));  //delete
      }
    }
  }
  return dp[n][m];
}

/*** Example Usage ***/

#include <cassert>

int main() {
  assert(edit_distance("abxdef", "abcdefg") == 2);
  return 0;
}
```

6.5 Suffix Array and LCP

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

```
/*

A suffix array SA of a string S[1..n] is a sorted array of indices of
all the suffixes of S ("abc" has suffixes "abc", "bc", and "c").
SA[i] contains the starting position of the i-th smallest suffix in S,
ensuring that for all 1 < i <= n, S[SA[i - 1], n] < S[A[i], n] holds.
It is a simple, space efficient alternative to suffix trees.
By binary searching on a suffix array, one can determine whether a
substring exists in a string in O(log n) time per query.

The longest common prefix array (LCP array) stores the lengths of the
longest common prefixes between all pairs of consecutive suffixes in
a sorted suffix array and can be found in O(n) given the suffix array.

The following algorithm uses a "gap" partitioning algorithm
explained here: http://stackoverflow.com/a/17763563

Time Complexity: O(n log^2 n) for suffix_array() and O(n) for
lcp_array(), where n is the length of the input string.

Space Complexity: O(n) auxiliary.

*/

#include <algorithm>
#include <string>
#include <vector>

std::vector<long long> rank2;

bool comp(const int & a, const int & b) {
  return rank2[a] < rank2[b];
}

std::vector<int> suffix_array(const std::string & s) {
  int n = s.size();
  std::vector<int> sa(n), rank(n);
  for (int i = 0; i < n; i++) {
    sa[i] = i;
    rank[i] = (int)s[i];
  }
  rank2.resize(n);
  for (int len = 1; len < n; len *= 2) {
    for (int i = 0; i < n; i++)
      rank2[i] = ((long long)rank[i] << 32) +
                 (i + len < n ? rank[i + len] + 1 : 0);
    std::sort(sa.begin(), sa.end(), comp);
    for (int i = 0; i < n; i++)
      rank[sa[i]] = (i > 0 && rank2[sa[i - 1]] == rank2[sa[i]]) ?
                    rank[sa[i - 1]] : i;
  }
  return sa;
}
```

```

std::vector<int> lcp_array(const std::string & s,
                           const std::vector<int> & sa) {
  int n = sa.size();
  std::vector<int> rank(n), lcp(n - 1);
  for (int i = 0; i < n; i++)
    rank[sa[i]] = i;
  for (int i = 0, h = 0; i < n; i++) {
    if (rank[i] < n - 1) {
      int j = sa[rank[i] + 1];
      while (std::max(i, j) + h < n && s[i + h] == s[j + h])
        h++;
      lcp[rank[i]] = h;
      if (h > 0) h--;
    }
  }
  return lcp;
}

/*** Example Usage ***/

#include <cassert>
using namespace std;

int main() {
  string s("banana");
  vector<int> sa = suffix_array(s);
  vector<int> lcp = lcp_array(s, sa);
  int sa_ans[] = {5, 3, 1, 0, 4, 2};
  int lcp_ans[] = {1, 3, 0, 0, 2};
  assert(equal(sa.begin(), sa.end(), sa_ans));
  assert(equal(lcp.begin(), lcp.end(), lcp_ans));
  return 0;
}
```

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

```
/*

A suffix array SA of a string S[1..n] is a sorted array of indices of
all the suffixes of S ("abc" has suffixes "abc", "bc", and "c").
SA[i] contains the starting position of the i-th smallest suffix in S,
ensuring that for all 1 < i <= n, S[SA[i - 1], n] < S[A[i], n] holds.
It is a simple, space efficient alternative to suffix trees.
By binary searching on a suffix array, one can determine whether a
substring exists in a string in O(log n) time per query.

The longest common prefix array (LCP array) stores the lengths of the
longest common prefixes between all pairs of consecutive suffixes in
a sorted suffix array and can be found in O(n) given the suffix array.

The following algorithm uses a "gap" partitioning algorithm
explained here: http://stackoverflow.com/a/17763563, except that the
O(n log n) comparison-based sort is substituted for an O(n) counting
sort to reduce the running time by an order of log n.

Time Complexity: O(n log n) for suffix_array() and O(n) for
```

```
lcp_array(), where n is the length of the input string.

Space Complexity: O(n) auxiliary.

*/

#include <algorithm>
#include <string>
#include <vector>

const std::string * str;

bool comp(const int & a, const int & b) {
  return (*str)[a] < (*str)[b];
}

std::vector<int> suffix_array(const std::string & s) {
  int n = s.size();
  std::vector<int> sa(n), order(n), rank(n);
  for (int i = 0; i < n; i++)
    order[i] = n - 1 - i;
  str = &s;
  std::stable_sort(order.begin(), order.end(), comp);
  for (int i = 0; i < n; i++) {
    sa[i] = order[i];
    rank[i] = (int)s[i];
  }
  std::vector<int> r(n), cnt(n), _sa(n);
  for (int len = 1; len < n; len *= 2) {
    r = rank;
    _sa = sa;
    for (int i = 0; i < n; i++)
      cnt[i] = i;
    for (int i = 0; i < n; i++) {
      if (i > 0 && r[sa[i - 1]] == r[sa[i]] && sa[i - 1] + len < n &&
          r[sa[i - 1] + len / 2] == r[sa[i] + len / 2]) {
        rank[sa[i]] = rank[sa[i - 1]];
      } else {
        rank[sa[i]] = i;
      }
    }
    for (int i = 0; i < n; i++) {
      int s1 = _sa[i] - len;
      if (s1 >= 0)
        sa[cnt[rank[s1]]++] = s1;
    }
  }
  return sa;
}

std::vector<int> lcp_array(const std::string & s,
                           const std::vector<int> & sa) {
  int n = sa.size();
  std::vector<int> rank(n), lcp(n - 1);
  for (int i = 0; i < n; i++)
    rank[sa[i]] = i;
  for (int i = 0, h = 0; i < n; i++) {
    if (rank[i] < n - 1) {
      int j = sa[rank[i] + 1];
```

```cpp
      while (std::max(i, j) + h < n && s[i + h] == s[j + h])
        h++;
      lcp[rank[i]] = h;
      if (h > 0) h--;
    }
  }
  return lcp;
}

/*** Example Usage ***/

#include <cassert>
using namespace std;

int main() {
  string s("banana");
  vector<int> sa = suffix_array(s);
  vector<int> lcp = lcp_array(s, sa);
  int sa_ans[] = {5, 3, 1, 0, 4, 2};
  int lcp_ans[] = {1, 3, 0, 0, 2};
  assert(equal(sa.begin(), sa.end(), sa_ans));
  assert(equal(lcp.begin(), lcp.end(), lcp_ans));
  return 0;
}
```

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

```cpp
/*

A suffix array SA of a string S[1, n] is a sorted array of indices of
all the suffixes of S ("abc" has suffixes "abc", "bc", and "c").
SA[i] contains the starting position of the i-th smallest suffix in S,
ensuring that for all 1 < i <= n, S[SA[i - 1], n] < S[A[i], n] holds.
It is a simple, space efficient alternative to suffix trees.
By binary searching on a suffix array, one can determine whether a
substring exists in a string in O(log n) time per query.

The longest common prefix array (LCP array) stores the lengths of the
longest common prefixes between all pairs of consecutive suffixes in
a sorted suffix array and can be found in O(n) given the suffix array.

The following implementation uses the sophisticated DC3/skew algorithm
by Karkkainen & Sanders (2003), using radix sort on integer alphabets
for linear construction. The function suffix_array(s, SA, n, K) takes
in s, an array [0, n - 1] of ints with n values in the range [1, K].
It stores the indices defining the suffix array into SA. The last value
of the input array s[n - 1] must be equal 0, the sentinel character. A
C++ wrapper function suffix_array(std::string) is implemented below it.

Time Complexity: O(n) for suffix_array() and lcp_array(), where n is
the length of the input string.

Space Complexity: O(n) auxiliary.

*/

inline bool leq(int a1, int a2, int b1, int b2) {
  return a1 < b1 || (a1 == b1 && a2 <= b2);
}

inline bool leq(int a1, int a2, int a3, int b1, int b2, int b3) {
  return a1 < b1 || (a1 == b1 && leq(a2, a3, b2, b3));
}

static void radix_pass(int * a, int * b, int * r, int n, int K) {
  int *c = new int[K + 1];
  for (int i = 0; i <= K; i++)
    c[i] = 0;
  for (int i = 0; i < n; i++)
    c[r[a[i]]]++;
  for (int i = 0, sum = 0; i <= K; i++) {
    int tmp = c[i];
    c[i] = sum;
    sum += tmp;
  }
  for (int i = 0; i < n; i++)
    b[c[r[a[i]]]++] = a[i];
  delete[] c;
}

void suffix_array(int * s, int * sa, int n, int K) {
  int n0 = (n + 2) / 3, n1 = (n + 1) / 3, n2 = n / 3, n02 = n0 + n2;
  int *s12 = new int[n02 + 3], *SA12 = new int[n02 + 3];
  s12[n02] = s12[n02 + 1] = s12[n02 + 2] = 0;
  SA12[n02] = SA12[n02 + 1] = SA12[n02 + 2] = 0;
  int *s0 = new int[n0], *SA0 = new int[n0];
  for (int i = 0, j = 0; i < n + n0 - n1; i++)
    if (i % 3 != 0) s12[j++] = i;
  radix_pass(s12 , SA12, s + 2, n02, K);
  radix_pass(SA12, s12 , s + 1, n02, K);
  radix_pass(s12 , SA12, s , n02, K);
  int name = 0, c0 = -1, c1 = -1, c2 = -1;
  for (int i = 0; i < n02; i++) {
    if (s[SA12[i]] != c0 || s[SA12[i] + 1] != c1 || s[SA12[i] + 2] != c2) {
      name++;
      c0 = s[SA12[i]];
      c1 = s[SA12[i] + 1];
      c2 = s[SA12[i] + 2];
    }
    if (SA12[i] % 3 == 1)
      s12[SA12[i] / 3] = name;
    else
      s12[SA12[i] / 3 + n0] = name;
  }
  if (name < n02) {
    suffix_array(s12, SA12, n02, name);
    for (int i = 0; i < n02; i++)
      s12[SA12[i]] = i + 1;
  } else {
    for (int i = 0; i < n02; i++)
      SA12[s12[i] - 1] = i;
  }
  for (int i = 0, j = 0; i < n02; i++)
    if (SA12[i] < n0)
      s0[j++] = 3 * SA12[i];
  radix_pass(s0, SA0, s, n0, K);
```

```
#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
  for (int p = 0, t = n0 - n1, k = 0; k < n; k++) {
    int i = GetI(), j = SA0[p];
    if (SA12[t] < n0 ? leq(s[i], s12[SA12[t] + n0],s[j], s12[j/3]) :
        leq(s[i], s[i + 1], s12[SA12[t] - n0 + 1], s[j], s[j + 1], s12[j / 3 + n0])) {
      sa[k] = i;
      if (++t == n02)
        for (k++; p < n0; p++, k++)
          sa[k] = SA0[p];
    } else {
      sa[k] = j;
      if (++p == n0)
        for (k++; t < n02; t++, k++)
          sa[k] = GetI();
    }
  }
#undef GetI
  delete[] s12;
  delete[] SA12;
  delete[] SA0;
  delete[] s0;
}

#include <string>
#include <vector>

// C++ wrapper function
std::vector<int> suffix_array(const std::string & s) {
  int n = s.size();
  int *str = new int[n + 5], *sa = new int[n + 1];
  for (int i = 0; i < n + 5; i++) str[i] = 0;
  for (int i = 0; i < n; i++) str[i] = (int)s[i];
  suffix_array(str, sa, n + 1, 256);
  return std::vector<int>(sa + 1, sa + n + 1);
}

std::vector<int> lcp_array(const std::string & s,
                           const std::vector<int> & sa) {
  int n = sa.size();
  std::vector<int> rank(n), lcp(n - 1);
  for (int i = 0; i < n; i++)
    rank[sa[i]] = i;
  for (int i = 0, h = 0; i < n; i++) {
    if (rank[i] < n - 1) {
      int j = sa[rank[i] + 1];
      while (std::max(i, j) + h < n && s[i + h] == s[j + h])
        h++;
      lcp[rank[i]] = h;
      if (h > 0) h--;
    }
  }
  return lcp;
}

/*** Example Usage ***/

#include <cassert>
using namespace std;

```

```
int main() {
  string s("banana");
  vector<int> sa = suffix_array(s);
  vector<int> lcp = lcp_array(s, sa);
  int sa_ans[] = {5, 3, 1, 0, 4, 2};
  int lcp_ans[] = {1, 3, 0, 0, 2};
  assert(equal(sa.begin(), sa.end(), sa_ans));
  assert(equal(lcp.begin(), lcp.end(), lcp_ans));
  return 0;
}
```

6.6 String Data Structures

6.5.1 Simple Trie

```
/*

A trie, digital tree, or prefix tree, is an ordered tree data
structure that is used to store a dynamic set or associative array
where the keys are strings. Each leaf node represents a string that
has been inserted into the trie. This makes tries easier to implement
than balanced binary search trees, and also potentially faster.

Time Complexity: O(n) for insert(), contains(), and erase(), where
n is the length of the string being inserted, searched, or erased.

Space Complexity: At worst O(l * ALPHABET_SIZE), where l is the
sum of all lengths of strings that have been inserted so far.

*/

#include <string>

class trie {
  static const int ALPHABET_SIZE = 26;

  static int map_alphabet(char c) {
    return (int)(c - 'a');
  }

  struct node_t {
    bool leaf;

    node_t * children[ALPHABET_SIZE];

    node_t(): leaf(false) {
      for (int i = 0; i < ALPHABET_SIZE; i++)
        children[i] = 0;
    }

    bool is_free() {
      for (int i = 0; i < ALPHABET_SIZE; i++)
        if (this->children[i] != 0) return true;
      return false;
    }
  } *root;

  bool erase(const std::string & s, node_t * n, int depth) {
    if (n == 0) return false;
    if (depth == (int)s.size()) {
      if (n->leaf) {
        n->leaf = false;
        return n->is_free();
      }
    } else {
      int idx = map_alphabet(s[depth]);
      if (erase(s, n->children[idx], depth + 1)) {
        delete n->children[idx];
        return !n->leaf && n->is_free();
      }
    }
    return false;
  }

  static void clean_up(node_t * n) {
    if (n == 0 || n->leaf) return;
    for (int i = 0; i < ALPHABET_SIZE; i++)
      clean_up(n->children[i]);
    delete n;
  }

 public:
  trie() { root = new node_t(); }
  ~trie() { clean_up(root); }

  void insert(const std::string & s) {
    node_t * n = root;
    for (int i = 0; i < (int)s.size(); i++) {
      int c = map_alphabet(s[i]);
      if (n->children[c] == 0)
        n->children[c] = new node_t();
      n = n->children[c];
    }
    n->leaf = true;
  }

  bool contains(const std::string & s) {
    node_t *n = root;
    for (int i = 0; i < (int)s.size(); i++) {
      int c = map_alphabet(s[i]);
      if (n->children[c] == 0)
        return false;
      n = n->children[c];
    }
    return n != 0 && n->leaf;
  }

  bool erase(const std::string & s) {
    return erase(s, root, 0);
  }
};

/*** Example Usage ***/

#include <cassert>
```

```
using namespace std;

int main() {
  string s[8] = {"a", "to", "tea", "ted", "ten", "i", "in", "inn"};
  trie t;
  for (int i = 0; i < 8; i++)
    t.insert(s[i]);
  assert(t.contains("ten"));
  t.erase("tea");
  assert(!t.contains("tea"));
  return 0;
}
```

6.5.2 Radix Trie

```
/*

A radix tree, radix trie, patricia trie, or compressed trie is a
data structure that is used to store a dynamic set or associative
array where the keys are strings. Each leaf node represents a string
that has been inserted into the trie. Unlike simple tries, radix
tries are space-optimized by merging each node that is an only child
with its parent.

Time Complexity: O(n) for insert(), contains(), and erase(), where
n is the length of the string being inserted, searched, or erased.

Space Complexity: At worst O(l), where l is the sum of all lengths
of strings that have been inserted so far.

*/

#include <string>
#include <vector>

class radix_trie {
  struct node_t {
    std::string label;
    std::vector<node_t*> children;

    node_t(const std::string & s = "") {
      label = s;
    }
  } *root;

  unsigned int lcplen(const std::string & s, const std::string & t) {
    int minsize = (t.size() < s.size()) ? t.size() : s.size();
    if (minsize == 0) return 0;
    unsigned int res = 0;
    for (int i = 0; i < minsize && s[i] == t[i]; i++)
      res++;
    return res;
  }

  void insert(const std::string & s, node_t * n) {
    unsigned int lcp = lcplen(s, n->label);
    if (lcp == 0 || n == root ||
        (lcp > 0 && lcp < s.size() && lcp >= n->label.size())) {
      bool inserted = false;
      std::string newstr = s.substr(lcp, s.size() - lcp);
      for (int i = 0; i < (int)n->children.size(); i++) {
        if (n->children[i]->label[0] == newstr[0]) {
          inserted = true;
          insert(newstr, n->children[i]);
        }
      }
      if (!inserted)
        n->children.push_back(new node_t(newstr));
    } else if (lcp < s.size()) {
      node_t * t = new node_t();
      t->label = n->label.substr(lcp, n->label.size() - lcp);
      t->children.assign(n->children.begin(), n->children.end());
      n->label = s.substr(0, lcp);
      n->children.assign(1, t);
      n->children.push_back(new node_t(s.substr(lcp, s.size() - lcp)));
    }
  }

  void erase(const std::string & s, node_t * n) {
    unsigned int lcp = lcplen(s, n->label);
    if (lcp == 0 || n == root ||
        (lcp > 0 && lcp < s.size() && lcp >= n->label.size())) {
      std::string newstr = s.substr(lcp, s.size() - lcp);
      for (int i = 0; i < (int)n->children.size(); i++) {
        if (n->children[i]->label[0] == newstr[0]) {
          if (newstr == n->children[i]->label &&
              n->children[i]->children.empty()) {
            n->children.erase(n->children.begin() + i);
            return;
          }
          erase(newstr, n->children[i]);
        }
      }
    }
  }

  bool contains(const std::string & s, node_t * n) {
    unsigned int lcp = lcplen(s, n->label);
    if (lcp == 0 || n == root ||
        (lcp > 0 && lcp < s.size() && lcp >= n->label.size())) {
      std::string newstr = s.substr(lcp, s.size() - lcp);
      for (int i = 0; i < (int)n->children.size(); i++)
      if (n->children[i]->label[0] == newstr[0])
        return contains(newstr, n->children[i]);
      return false;
    }
    return lcp == n->label.size();
  }

  static void clean_up(node_t * n) {
    if (n == 0) return;
    for (int i = 0; i < (int)n->children.size(); i++)
      clean_up(n->children[i]);
    delete n;
  }

```

```
 public:
  template <class UnaryFunction>
  void walk(node_t * n, UnaryFunction f) {
    if (n == 0) return;
    if (n != root) f(n->label);
    for (int i = 0; i < (int)n->children.size(); i++)
      walk(n->children[i], f);
  }

  radix_trie() { root = new node_t(); }
  ~radix_trie() { clean_up(root); }

  void insert(const std::string & s) { insert(s, root); }
  void erase(const std::string & s) { erase(s, root); }
  bool contains(const std::string & s) { return contains(s, root); }

  template <class UnaryFunction> void walk(UnaryFunction f) {
    walk(root, f);
  }
};

/*** Example Usage ***/

#include <cassert>
using namespace std;

string preorder;

void concat(const string & s) {
  preorder += (s + "␣");
}

int main() {
  {
    string s[8] = {"a", "to", "tea", "ted", "ten", "i", "in", "inn"};
    radix_trie t;
    for (int i = 0; i < 8; i++)
      t.insert(s[i]);
    assert(t.contains("ten"));
    t.erase("tea");
    assert(!t.contains("tea"));
  }
  {
    radix_trie t;
    t.insert("test");
    t.insert("toaster");
    t.insert("toasting");
    t.insert("slow");
    t.insert("slowly");
    preorder = "";
    t.walk(concat);
    assert(preorder == "t␣est␣oast␣er␣ing␣slow␣ly␣");
  }
  return 0;
}
```

6.5.3 Suffix Trie

```
/*

A suffix tree of a string S is a compressed trie of all the suffixes
of S. While it can be constructed in O(n^2) time on the length of S
by simply inserting the suffixes into a radix tree, Ukkonen (1995)
provided an algorithm to construct one in O(n * ALPHABET_SIZE).

Suffix trees can be used for string searching, pattern matching, and
solving the longest common substring problem. The implementation
below is optimized for solving the latter.

Time Complexity: O(n) for construction of suffix_tree() and
per call to longest_common_substring(), respectively.

Space Complexity: O(n) auxiliary.

*/

#include <cstdio>
#include <string>

struct suffix_tree {

  static const int ALPHABET_SIZE = 38;

  static int map_alphabet(char c) {
    static const std::string ALPHABET(
      "abcdefghijklmnopqrstuvwxyz0123456789\01\02"
    );
    return ALPHABET.find(c);
  }

  struct node_t {
    int begin, end, depth;
    node_t *parent, *suffix_link;
    node_t *children[ALPHABET_SIZE];

    node_t(int begin, int end, int depth, node_t * parent) {
      this->begin = begin;
      this->end = end;
      this->depth = depth;
      this->parent = parent;
      for (int i = 0; i < ALPHABET_SIZE; i++)
        children[i] = 0;
    }
  } *root;

  suffix_tree(const std::string & s) {
    int n = s.size();
    int * c = new int[n];
    for (int i = 0; i < n; i++) c[i] = map_alphabet(s[i]);
    root = new node_t(0, 0, 0, 0);
    node_t *node = root;
    for (int i = 0, tail = 0; i < n; i++, tail++) {
      node_t *last = 0;
      while (tail >= 0) {
        node_t *ch = node->children[c[i - tail]];
        while (ch != 0 && tail >= ch->end - ch->begin) {
          tail -= ch->end - ch->begin;
          node = ch;
          ch = ch->children[c[i - tail]];
        }
        if (ch == 0) {
          node->children[c[i]] = new node_t(i, n,
                                  node->depth + node->end - node->begin, node);
          if (last != 0) last->suffix_link = node;
          last = 0;
        } else {
          int aftertail = c[ch->begin + tail];
          if (aftertail == c[i]) {
            if (last != 0) last->suffix_link = node;
            break;
          } else {
            node_t *split = new node_t(ch->begin, ch->begin + tail,
                                 node->depth + node->end - node->begin, node);
            split->children[c[i]] = new node_t(i, n, ch->depth + tail, split);
            split->children[aftertail] = ch;
            ch->begin += tail;
            ch->depth += tail;
            ch->parent = split;
            node->children[c[i - tail]] = split;
            if (last != 0)
              last->suffix_link = split;
            last = split;
          }
        }
        if (node == root) {
          tail--;
        } else {
          node = node->suffix_link;
        }
      }
    }
  }
};

int lcs_begin, lcs_len;

int lcs_rec(suffix_tree::node_t * n, int i1, int i2) {
  if (n->begin <= i1 && i1 < n->end) return 1;
  if (n->begin <= i2 && i2 < n->end) return 2;
  int mask = 0;
  for (int i = 0; i < suffix_tree::ALPHABET_SIZE; i++) {
    if (n->children[i] != 0)
      mask |= lcs_rec(n->children[i], i1, i2);
  }
  if (mask == 3) {
    int curr_len = n->depth + n->end - n->begin;
    if (lcs_len < curr_len) {
      lcs_len = curr_len;
      lcs_begin = n->begin;
    }
  }
  return mask;
}

std::string longest_common_substring
(const std::string & s1, const std::string & s2) {
  std::string s(s1 + '\01' + s2 + '\02');
  suffix_tree tree(s);
  lcs_begin = lcs_len = 0;
  lcs_rec(tree.root, s1.size(), s1.size() + s2.size() + 1);
  return s.substr(lcs_begin - 1, lcs_len);
}

/*** Example Usage ***/

#include <cassert>

int main() {
  assert(longest_common_substring("bbbabca", "aababcd") == "babc");
  return 0;
}
```

6.5.4 Suffix Automaton

```
/*

A suffix automaton is a data structure to efficiently represent the
suffixes of a string. It can be considered a compressed version of
a suffix tree. The data structure supports querying for substrings
within the text from with the automaton is constructed in linear
time. It also supports computation of the longest common substring
in linear time.

Time Complexity: O(n * ALPHABET_SIZE) for construction, and O(n)
for find_all(), as well as longest_common_substring().

Space Complexity: O(n * ALPHABET_SIZE) auxiliary.

*/

#include <algorithm>
#include <queue>
#include <string>
#include <vector>

struct suffix_automaton {

  static const int ALPHABET_SIZE = 26;

  static int map_alphabet(char c) {
    return (int)(c - 'a');
  }

  struct state_t {
    int length, suffix_link;
    int firstpos, next[ALPHABET_SIZE];
    std::vector<int> invlinks;

    state_t() {
      length = 0;
      suffix_link = 0;
      firstpos = -1;
      for (int i = 0; i < ALPHABET_SIZE; i++)
        next[i] = -1;
    }
  };

  std::vector<state_t> states;

  suffix_automaton(const std::string & s) {
    int n = s.size();
    states.resize(std::max(2, 2 * n - 1));
    states[0].suffix_link = -1;
    int last = 0;
    int size = 1;
    for (int i = 0; i < n; i++) {
      int c = map_alphabet(s[i]);
      int curr = size++;
      states[curr].length = i + 1;
      states[curr].firstpos = i;
      int p = last;
      while (p != -1 && states[p].next[c] == -1) {
        states[p].next[c] = curr;
        p = states[p].suffix_link;
      }
      if (p == -1) {
        states[curr].suffix_link = 0;
      } else {
        int q = states[p].next[c];
        if (states[p].length + 1 == states[q].length) {
          states[curr].suffix_link = q;
        } else {
          int clone = size++;
          states[clone].length = states[p].length + 1;
          for (int i = 0; i < ALPHABET_SIZE; i++)
            states[clone].next[i] = states[q].next[i];
          states[clone].suffix_link = states[q].suffix_link;
          while (p != -1 && states[p].next[c] == q) {
            states[p].next[c] = clone;
            p = states[p].suffix_link;
          }
          states[q].suffix_link = clone;
          states[curr].suffix_link = clone;
        }
      }
      last = curr;
    }
    for (int i = 1; i < size; i++)
      states[states[i].suffix_link].invlinks.push_back(i);
    states.resize(size);
  }

  std::vector<int> find_all(const std::string & s) {
    std::vector<int> res;
    int node = 0;
    for (int i = 0; i < (int)s.size(); i++) {
      int next = states[node].next[map_alphabet(s[i])];
      if (next == -1) return res;
      node = next;
    }
    std::queue<int> q;
    q.push(node);
```

```
    while (!q.empty()) {
      int curr = q.front();
      q.pop();
      if (states[curr].firstpos != -1)
        res.push_back(states[curr].firstpos - (int)s.size() + 1);
      for (int j = 0; j < (int)states[curr].invlinks.size(); j++)
        q.push(states[curr].invlinks[j]);
    }
    return res;
  }

  std::string longest_common_substring(const std::string & s) {
    int len = 0, bestlen = 0, bestpos = -1;
    for (int i = 0, cur = 0; i < (int)s.size(); i++) {
      int c = map_alphabet(s[i]);
      if (states[cur].next[c] == -1) {
        while (cur != -1 && states[cur].next[c] == -1)
          cur = states[cur].suffix_link;
        if (cur == -1) {
          cur = len = 0;
          continue;
        }
        len = states[cur].length;
      }
      len++;
      cur = states[cur].next[c];
      if (bestlen < len) {
        bestlen = len;
        bestpos = i;
      }
    }
    return s.substr(bestpos - bestlen + 1, bestlen);
  }
};

/*** Example Usage ***/

#include <algorithm>
#include <cassert>
using namespace std;

int main() {
  {
    suffix_automaton sa("bananas");
    vector<int> pos_a, pos_an, pos_ana;
    int ans_a[] = {1, 3, 5};
    int ans_an[] = {1, 3};
    int ans_ana[] = {1, 3};
    pos_a = sa.find_all("a");
    pos_an = sa.find_all("an");
    pos_ana = sa.find_all("ana");
    assert(equal(pos_a.begin(), pos_a.end(), ans_a));
    assert(equal(pos_an.begin(), pos_an.end(), ans_an));
    assert(equal(pos_ana.begin(), pos_ana.end(), ans_ana));
  }
  {
    suffix_automaton sa("bbbabca");
    assert(sa.longest_common_substring("aababcd") == "babc");
  }
  return 0;
}
```