Sorting Algorithms

```cpp
/*
These functions are equivalent to std::sort(), taking
RandomAccessIterators
as a range [lo, hi) to be sorted. Elements between lo and hi
(including the
element pointed to by lo but excluding the element pointed to by
hi) will be
sorted into ascending order after the function call. Optionally, a
comparison
function object specifying a strict weak ordering may be specified
to replace
the default operator <.

These functions are not meant to compete with standard library
implementations
in terms terms of speed. Instead, they are meant to demonstrate
how common
sorting algorithms can be concisely implemented in C++.
*/

#include <algorithm>
#include <functional>
#include <iterator>
#include <vector>

/*
Quicksort repeatedly selects a pivot and partitions the range so
that elements
comparing less than the pivot precede the pivot, and elements
comparing greater
or equal follow 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 often faster in practice than merge sort and
heapsort, which both
have a worst case time complexity of O(n log n).

The pivot chosen in this implementation is always a middle element
of the range
to be sorted. To reduce the likelihood of encountering the worst
case, the pivot
can be chosen in better ways (e.g. randomly, or using the "median
of three"
technique).

Time Complexity (Average): O(n log n).
Time Complexity (Worst): O(n^2).
Space Complexity: O(log n) auxiliary stack space.
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 first divides a list into n sublists of one element
each, then
recursively merges the sublists into sorted order until only a
single sorted
sublist remains. Merge sort is a stable sort, meaning that it
preserves the
relative order of elements which compare equal by operator < or
the custom
comparator given.

An analogous function in the C++ standard library is
std::stable_sort(), except
that the implementation here requires sufficient memory to be
available. When
O(n) auxiliary memory is not available, std::stable_sort() falls
back to a time
complexity of O(n log^2 n) whereas the implementation here will
simply fail.
```

```
Time Complexity (Average): O(n log n).
Time Complexity (Worst): O(n log n).
Space Complexity: O(log n) auxiliary stack space and O(n)
auxiliary heap space.
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;
  std::vector<T> merged;
  while (a <= mid && c < hi) {
    merged.push_back(comp(*c, *a) ? *c++ : *a++);
  }
  if (a > mid) {
    for (It it = c; it < hi; ++it) {
      merged.push_back(*it);
    }
  } else {
    for (It it = a; it <= mid; ++it) {
      merged.push_back(*it);
    }
  }
  for (int i = 0; i < hi - lo; i++) {
    *(lo + i) = merged[i];
  }
}

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 first rearranges an array to satisfy the max-heap
property. Then, it
repeatedly pops the max element of the heap (the left, unsorted
subrange),
moving it to the beginning of the right, sorted subrange until the
entire range
is sorted. Heapsort has a better worst case time complexity than
quicksort and
also a better space complexity than merge sort.
```

The C++ standard library equivalent is calling std::make_heap(lo, hi), followed
by std::sort_heap(lo, hi).

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 tmp;
  It i = lo + (hi - lo)/2, j = hi, parent, child;
  for (;;) {
    if (i <= lo) {
      if (--j == lo) {
        return;
      }
      tmp = *j;
      *j = *lo;
    } else {
      tmp = *(--i);
    }
    parent = i;
    child = lo + 2*(i - lo) + 1;
    while (child < j) {
      if (child + 1 < j && comp(*child, *(child + 1))) {
        child++;
      }
      if (!comp(tmp, *child)) {
        break;
      }
      *parent = *child;
      parent = child;
      child = lo + 2*(parent - lo) + 1;
    }
    *(lo + (parent - lo)) = tmp;
  }
}

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 is an improved bubble sort. While bubble sort increments the gap

```
between swapped elements for every inner loop iteration, comb sort
fixes the gap
size in the inner loop, decreasing it by a particular shrink
factor in every
iteration of the outer loop. The shrink factor of 1.3 is
empirically determined
to be the most effective.

Even though the worst case time complexity is O(n^2), a well
chosen shrink
factor ensures that the gap sizes are co-prime, in turn requiring
astronomically
large n to make the algorithm exceed O(n log n) steps. On random
arrays, comb
sort is only 2-3 times slower than merge sort. Its short code
length length
relative to its good performance makes it a worthwhile algorithm
to remember.

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)((double)gap / 1.3);
    }
    swapped = false;
    for (It it = lo; it + gap < hi; ++it) {
      if (comp(*(it + gap), *it)) {
        std::swap(*it, *(it + 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 is used to sort integer elements with a constant number
of bits in
```

```
linear time. This implementation only works on ranges pointing to
unsigned
integer primitives. The elements in the input range do not
strictly have to be
unsigned types, as long as their values are nonnegative integers.

In this implementation, a power of two is chosen to be the base
for the sort
so that bitwise operations can be easily used to extract digits.
This avoids the
need to use modulo and exponentiation, which are much more
expensive operations.
In practice, it's been demonstrated that 2^8 is the best choice
for sorting
32-bit integers (approximately 5 times faster than std::sort(),
and typically
2-4 faster than radix sort using any other power of two chosen as
the base).

Time Complexity: O(n*w) for n integers of w bits each.
Space Complexity: O(n + w) auxiliary.
*/

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 *buf = 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 = buf;
    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(buf, buf + (hi - lo), lo);
  }
  delete[] buf;
}
```

```
/*** Example Usage and 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():  0.355s
quicksort():  0.426s
mergesort():  1.263s
heapsort():   1.093s
combsort():   0.827s
radix_sort(): 0.076s

***/

#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 sorted(It lo, It hi) {
  while (++lo != hi) {
    if (*lo < *(lo - 1)) {
      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(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(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(sorted(v.rbegin(), v.rend()));
  }
  {  // We can sort doubles just as well.
    double a[] = {1.1, -5.0, 6.23, 4.123, 155.2};
    vector<double> v(a, a + 5);
    combsort(v.begin(), v.end());
    assert(sorted(v.begin(), v.end()));
  }
  {  // Must use radix_sort with unsigned values, but sorting in
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(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(sort_function) {                                  \
  clock_t start = clock();                                     \
  sort_function(v.begin(), v.end());                           \
  double t = (double)(clock() - start) / CLOCKS_PER_SEC;      \
  cout << setw(14) << left << #sort_function "(): ";          \
  cout << fixed << t << "s" << endl;                           \
  assert(sorted(v.begin(), v.end()));                          \
  v = v2;                                                      \
}
  test(std::sort);
  test(quicksort);
  test(mergesort);
  test(heapsort);
  test(combsort);
  test(radix_sort);
  return 0;
}
```

Array Rotation

```
/*
These functions are equivalent to std::rotate(), taking three
iterators lo, mid,
and hi (lo <= mid <= hi) to perform a left rotation on the range
[lo, hi). After
the function call, [lo, hi) will comprise of the concatenation of
the elements
originally in [mid, hi) + [lo, mid). That is, the range [lo, hi)
will be
rearranged 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 while
preserving the relative ordering of elements within the two
rotated subarrays.

All three versions below achieve the same result using in-place
algorithms.
Version 1 uses a straightforward swapping algorithm requiring
ForwardIterators.
Version 2 requires BidirectionalIterators, employing a well-known
trick with
three simple inversions. Version 3 requires RandomAccessIterators,
applying a
juggling algorithm which first divides the range into gcd(hi - lo,
mid - lo)
sets and then rotates the corresponding elements in each set.

Time Complexity:
```

```
- O(n) per call to both functions, where n is the distance between
lo and hi.

Space Complexity:
- O(1) auxiliary.
*/

#include <algorithm>

template<class It>
void rotate1(It lo, It mid, It hi) {
  It next = mid;
  while (lo != next) {
    std::iter_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::iter_swap(lo + curr, lo + next);
      curr = next;
    }
  }
}

/*** Example Usage and 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() {
  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;
}
```

Counting Inversions

```
/*
The number of inversions for an array a[] is defined as 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 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 two functions are each techniques to efficiently count
inversions.
*/

#include <algorithm>
#include <iterator>
#include <vector>

/*
Version 1: Merge Sort

Returns the number of inversions given two RandomAccessIterators
as a range
[lo, hi). The range will become sorted after the function call.
This requires
operator < to be defined on the iterator's value type.

Time Complexity:
- O(n log n) per call to both functinos, where n is distance
between lo and hi
  in the first version and the number of array elements in the
second version.

Space Complexity:
```

```
- O(n) auxiliary heap space for both functions.
*/

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;
  std::vector<T> merged;
  while (a <= mid && c < hi) {
    if (*c < *a) {
      merged.push_back(*(c++));
      res += (mid - a) + 1;
    } else {
      merged.push_back(*(a++));
    }
  }
  if (a > mid) {
    for (It it = c; it != hi; ++it) {
      merged.push_back(*it);
    }
  } else {
    for (It it = a; it <= mid; ++it) {
      merged.push_back(*it);
    }
  }
  for (It it = lo; it != hi; ++it) {
    *it = merged[it - lo];
  }
  return res;
}

/*
Version 2: Power-of-Two Trick

Returns the number of inversions for an array a[] of n nonnegative
integers.
After calling the function, every value of a[] will be set to 0.

Here, the time and space complexities depend on the magnitude of
the maximum
value in a[]. Therefore for a running time of O(n log n),
coordinate compression
may be applied to a[] so its maximum is strictly less than the
length n itself.
```

```
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]);
  }
  long long res = 0;
  std::vector<int> count(mx);
  while (mx > 0) {
    std::fill(count.begin(), count.end(), 0);
    for (int i = 0; i < n; i++) {
      if (a[i] % 2 == 0) {
        res += count[a[i] / 2];
      } else {
        count[a[i] / 2]++;
      }
    }
    mx = 0;
    for (int i = 0; i < n; i++) {
      mx = std::max(mx, a[i] /= 2);
    }
  }
  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;
}
```

Coordinate Compression

```
/*
Given two ForwardIterators as a range [lo, hi) of n numerical
elements, reassign
each element in the range to an integer in [0, k), where k is the
```

number of
distinct elements in the original range, while preserving the initial relative
ordering of elements. That is, if a[] is an array of the original values and b[]
is the compressed values, then every pair of indices i, j (0 <= i, j < n) shall
satisfy a[i] < a[j] if and only if b[i] < b[j].

Both implementations below require operator < to be defined on the iterator's
value type. Version 1 performs the compression by sorting the array, removing
duplicates, and binary searching for the position of each original value.
Version 2 achieves the same result by inserting all values in a balanced binary
search tree (std::map) which automatically removes duplicate values and supports
efficient lookups of the compressed values.

Time Complexity:
- O(n log n) per call to either function, where n is the distance between lo and
  hi.

Space Complexity:
- O(n) auxiliary heap space.
*/

```
#include <algorithm>
#include <iterator>
#include <map>
#include <vector>

template<class It> void compress1(It lo, It hi) {
  typedef typename std::iterator_traits<It>::value_type T;
  std::vector<T> v;
  for (It it = lo; it != hi; ++it) {
    v.push_back(*it);
  }
  std::sort(v.begin(), v.end());
  v.resize(std::unique(v.begin(), v.end()) - v.begin());
  for (It it = lo; it != hi; ++it) {
    *it = (int)(std::lower_bound(v.begin(), v.end(), *it) -
v.begin());
  }
}

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 it = m.begin();
  for (int id = 0; it != m.end(); it++) {
    it->second = id++;
  }
  for (It it = lo; it != hi; ++it) {
    *it = m[*it];
  }
}

/*** Example Usage and 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);
  }
  {  // Non-integral types work too, as long as ints can be
assigned to them.
    double a[] = {0.5, -1.0, 3, -1.0, 20, 0.5};
    compress1(a, a + 6);
    print_range(a, a + 6);
  }
  return 0;
}
```

Selection(Quickselect)

```
/*
nth_element2() is equivalent to std::nth_element(), taking
RandomAccessIterators
lo, nth, and hi as the range [lo, hi) to be partially sorted. The
values in
[lo, hi) are rearranged such that the value pointed to by nth is
the element
that would be there if the range were sorted. Furthermore, the
range is
partitioned such that no value in [lo, nth) compares greater than
the value
pointed to by nth and no value in (nth, hi) compares less. This
implementation
requires operator < to be defined on the iterator's value type.

Time Complexity:
- O(n) on average per call to nth_element2(), where n is the
distance between lo
  and hi.

Space Complexity:
- O(1) auxiliary.
*/

#include <algorithm>
#include <cstdlib>
#include <iterator>

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

template<class It>
void nth_element2(It lo, It nth, It hi) {
  for (;;) {
    std::iter_swap(lo + rand32() % (hi - lo), hi - 1);
    typename std::iterator_traits<It>::value_type mid = *(hi - 1);
    It k = lo - 1;
    for (It it = lo; it != hi; ++it) {
      if (!(mid < *it)) {
        std::iter_swap(++k, it);
      }
    }
    if (nth < k) {
      hi = k;
    } else if (k < nth) {
      lo = k + 1;
    } else {
      return;
    }
```

```
  }
}

/*** Example Usage and Output:

2 3 3 4 5 6 6 7 9

***/

#include <cassert>
#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 n = 9;
  int a[] = {5, 6, 4, 3, 2, 6, 7, 9, 3};
  nth_element2(a, a + n/2, a + n);
  assert(a[n/2] == 5);
  print_range(a, a + n);
  return 0;
}
```

Longest Increasing Subsequence

```
/*
Given two RandomAccessIterators lo and hi specifying a range [lo,
hi), determine
a longest subsequence of the range such that all of its elements
are in strictly
ascending order. This implementation requires operator < to be
defined on the
iterator's value type. The subsequence is not necessarily
contiguous or unique,
so only one such answer will be found. The answer is computed
using binary
search and dynamic programming.

Time Complexity:
- O(n log n) per call to longest_increasing_subsequence(), where n
is the
  distance between lo and hi.

Space Complexity:
- O(n) auxiliary heap space for longest_increasing_subsequence().
```

```
*/

#include <iterator>
#include <vector>

template<class It>
std::vector<typename std::iterator_traits<It>::value_type>
longest_increasing_subsequence(It lo, It hi) {
  int len = 0, n = hi - lo;
  std::vector<int> prev(n), tail(n);
  for (int i = 0; i < n; i++) {
    int l = -1, h = len;
    while (h - l > 1) {
      int mid = (l + h)/2;
      if (*(lo + tail[mid]) < *(lo + i)) {
        l = mid;
      } else {
        h = mid;
      }
    }
    if (len < h + 1) {
      len = h + 1;
    }
    prev[i] = h > 0 ? tail[h - 1] : -1;
    tail[h] = i;
  }
  std::vector<typename std::iterator_traits<It>::value_type>
res(len);
  for (int i = tail[len - 1]; i != -1; i = prev[i]) {
    res[--len] = *(lo + i);
  }
  return res;
}

/*** Example Usage and 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 = longest_increasing_subsequence(a, a + 10);
  print_range(res.begin(), res.end());
  return 0;
}
```

Maximal Subarray Sum (Kadane's)

```
/*

Given an array of numbers (at least one of which must be
positive), determine
the maximum possible sum of any contiguous subarray. Kadane's
algorithm scans
through the array, at each index computing the maximum positive
sum subarray
ending there. Either this subarray is empty (in which case its sum
is zero) or
it consists of one more element than the maximum sequence ending
at the previous
position. This can be adapted to compute the maximal submatrix sum
as well.

*/

#include <algorithm>
#include <cstdlib>
#include <iterator>
#include <limits>
#include <vector>

/*

Returns the maximal subarray sum for the range [lo, hi), where lo
and hi are
RandomAccessIterators to numerical types. This implementation
requires operators
+ and < to be defined on the iterator's value type. Optionally,
two int pointers
may be passed to store the inclusive boundary indices [res_lo,
res_hi] of the
resulting subarray. By convention, an input range consisting of
only negative
values will yield a size 1 subarray consisting of the maximum
value.

Time Complexity:
- O(n) per call to max_subarray_sum(), where n is the distance
between lo and
  hi.

Space Complexity:
```

```
- O(1) auxiliary.

*/

template<class It>
typename std::iterator_traits<It>::value_type
max_subarray_sum(It lo, It hi, int *res_lo = NULL, int *res_hi =
NULL) {
  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;
    }
  }
  if (end == -1) {
    // All values negative. By convention, just return the maximum
value.
    for (It it = lo; it != hi; ++it) {
      if (max_sum < *it) {
        max_sum = *it;
        begin = it - lo;
        end = begin;
      }
    }
  }
  if (res_lo != NULL && res_hi != NULL) {
    *res_lo = begin;
    *res_hi = end;
  }
  return max_sum;
}

/*

Returns the largest sum of any rectangular submatrix for a matrix
of n rows by
m columns. The matrix should be given as a 2-dimensional vector,
where the outer
vector must contain n vectors each of size m. This implementation
requires
operators + and < to be defined on the iterator's value type.
Optionally, four
int pointers may be passed to store the boundary indices of the
```

```
resulting
subarray, with (r1, c1) specifiying the top-left index and (r2,
c2) specifying
the bottom-right index. By convention, an input matrix consisting
of only
negative values will yield a size 1 submatrix consisting of the
maximum value.

Time Complexity:
- O(n*m^2) per call to max_submatrix_sum(), where n is the number
of rows and m
  is the number of columns in the matrix.

Space Complexity:
- O(n) auxiliary heap space for max_submatrix_sum(), where n is
the number of
  rows in the matrix.

*/

template<class T>
T max_submatrix_sum(const std::vector<std::vector<T> > &matrix,
    int *r1 = NULL, int *c1 = NULL, int *r2 = NULL, int *c2 =
NULL) {
  int n = matrix.size(), m = matrix[0].size();
  std::vector<T> sums(n);
  T sum, max_sum = std::numeric_limits<T>::min();
  for (int clo = 0; clo < m; clo++) {
    std::fill(sums.begin(), sums.end(), 0);
    for (int chi = clo; chi < m; chi++) {
      for (int i = 0; i < n; i++) {
        sums[i] += matrix[i][chi];
      }
      int rlo, rhi;
      sum = max_subarray_sum(sums.begin(), sums.end(), &rlo,
&rhi);
      if (max_sum < sum) {
        max_sum = sum;
        if (r1 != NULL && c1 != NULL && r2 != NULL && c2 != NULL)
{
          *r1 = rlo;
          *c1 = clo;
          *r2 = rhi;
          *c2 = chi;
        }
      }
    }
  }
  return max_sum;
}
```

```
/*** Example Usage and Output:

Maximal sum subarray:
4 -1 2 1

Maximal sum submatrix:
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 lo = 0, hi = 0;
    assert(max_subarray_sum(a, a + 3) == -1);
    assert(max_subarray_sum(a, a + 9, &lo, &hi) == 6);
    cout << "Maximal sum subarray:" << endl;
    for (int i = lo; i <= hi; 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> > matrix(n);
    for (int i = 0; i < n; i++) {
      matrix[i] = vector<int>(a[i], a[i] + m);
    }
    int r1 = 0, c1 = 0, r2 = 0, c2 = 0;
    assert(max_submatrix_sum(matrix, &r1, &c1, &r2, &c2) == 15);
    cout << "\nMaximal sum submatrix:" << endl;
    for (int i = r1; i <= r2; i++) {
      for (int j = c1; j <= c2; j++) {
        cout << matrix[i][j] << " ";
      }
      cout << endl;
    }
  }
  return 0;
}
```

Majority Element (Boyer-Moore)

```
/*

Given two ForwardIterators lo and hi specifying a range [lo, hi)
of n elements,
return an iterator to the first occurrence of the majority
element, or the
iterator hi if there is no majority element. The majority element
is defined as
an element which occurs strictly more than floor(n/2) times in the
range. This
implementation requires operator == to be defined on the
iterator's value type.

Time Complexity:
- O(n) per call to majority(), where n is the size of the array.

Space Complexity:
- O(1) auxiliary.

*/

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

/*** Example Usage ***/

#include <cassert>
```

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

Subset Sum (Meet-in-the-Middle)

```
/*

Given RandomAccessIterators lo and hi specifying a range [lo, hi)
of integers,
return the minimum sum of any subset of the range that is greater
than or equal
to a given integer v. This is a generalization of the NP-complete
subset sum
problem, which asks whether a subset summing to 0 exists
(equivalent in this
case to checking if v = 0 yields an answer of 0). This
implementation uses a
meet-in-the-middle algorithm to precompute and search for a lower
bound. Note
that 64-bit integers are used in intermediate calculations to
avoid overflow.

Time Complexity:
- O(n*2^(n/2)) per call to sum_lower_bound(), where n is the
distance between lo
  and hi.

Space Complexity:
- O(n) auxiliary heap space, where n is the number of array
elements.

*/

#include <algorithm>
#include <limits>
#include <vector>

template<class It>
long long sum_lower_bound(It lo, It hi, long long v) {
  int n = hi - lo, llen = 1 << (n/2), hlen = 1 << (n - n/2);
  std::vector<long long> lsum(llen), hsum(hlen);
  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.begin(), lsum.end());
  std::sort(hsum.begin(), hsum.end());
  int l = 0, h = hlen - 1;
  long long curr = std::numeric_limits<long long>::min();
  while (l < llen && h >= 0) {
    if (lsum[l] + hsum[h] <= v) {
      curr = std::max(curr, lsum[l] + hsum[h]);
      l++;
    } else {
      h--;
    }
  }
  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;
}
```

Maximal Zero Submatrix

```
/*

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

Time Complexity:
```

- O(n*m) per call to max_zero_submatrix(), where n is the number of rows and m
  is the number of columns in the matrix.

Space Complexity:
- O(m) auxiliary heap space, where m is the number of columns in the matrix.

*/

```
#include <algorithm>
#include <stack>
#include <vector>

int max_zero_submatrix(const std::vector<std::vector<bool> >
&matrix) {
  int n = matrix.size(), m = matrix[0].size(), res = 0;
  std::vector<int> d(m, -1), d1(m), d2(m);
  for (int r = 0; r < n; r++) {
    for (int c = 0; c < m; c++) {
      if (matrix[r][c]) {
        d[c] = r;
      }
    }
    std::stack<int> s;
    for (int c = 0; c < m; c++) {
      while (!s.empty() && d[s.top()] <= d[c]) {
        s.pop();
      }
      d1[c] = s.empty() ? -1 : s.top();
      s.push(c);
    }
    while (!s.empty()) {
      s.pop();
    }
    for (int c = m - 1; c >= 0; c--) {
      while (!s.empty() && d[s.top()] <= d[c]) {
        s.pop();
      }
      d2[c] = s.empty() ? m : s.top();
      s.push(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}};
  vector<vector<bool> > matrix(n);
  for (int i = 0; i < n; i++) {
    matrix[i] = vector<bool>(a[i], a[i] + m);
  }
  assert(max_zero_submatrix(matrix) == 6);
  return 0;
}
```

Binary Search

```
/*

Binary search can be generally used to find the input value
corresponding to any
output value of a monotonic (strictly non-increasing or strictly
non-decreasing)
function in O(log n) time with respect to the domain size. 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,
making it useful
for handling infinitely large search spaces such as real number
intervals.

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
smallest integer k in [lo, hi) for which the predicate pred(k)
tests true. If
pred(k) tests false for every k in [lo, hi), then hi is returned.
This function
must be used on a range in which there exists a constant k such
that pred(x)
tests false for every x in [lo, k) and true for every x in [k,
hi).

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
largest integer k in [lo, hi) for which the predicate pred(k)
tests true. If
pred(k) tests false for every k in [lo, hi), then hi is returned.
This function
must be used on a range in which there exists a constant k such
that pred(x)
tests true for every x in [lo, k] and false for every x in (k,
hi).

Time Complexity:
- O(log n) calls will be made to pred() in either function, where
n is the
  distance between lo and hi.

Space Complexity:
- O(1) auxiliary.

*/

template<class Int, class IntPredicate>
Int binary_search_first_true(Int lo, Int hi, IntPredicate pred)
{  // 000[1]11
  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;
}

template<class Int, class IntPredicate>
Int binary_search_last_true(Int lo, Int hi, IntPredicate pred)
{  // 11[1]000
  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 false.
  }
  return lo;
}

/*

fbinary_search() is the equivalent of binary_search_first_true()
on floating
point predicates. Since any interval of real numbers is dense, the
exact target
cannot be found due to floating point error. Instead, the function
returns 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
performs. Since
each repetition bisects the search space, the absolute error of
the answer is
1/(2^r) times the distance between lo and hi after r repetitions.
Although it is
possible to control the error by looping while hi - lo is greater
than an
arbitrary epsilon, it is simpler to let the loop run for a desired
number of
iterations until floating point arithmetic break down. 100
iterations is usually
sufficient, since the search space will be reduced to 2^-100
(roughly 10^-30)
times its original size.

This implementation can be modified to find the "last true" point
in the range
by simply interchanging the assignments of lo and hi in the if-
else statements.

Time Complexity:
- O(log n) calls will be made to pred(), where n is the distance
between lo and
  hi divided by the desired absolute error (based on the number of
iterations).

Space Complexity:
- O(1) auxiliary.

*/

template<class DoublePredicate>
double fbinary_search(double lo, double hi, DoublePredicate pred)
```

```
{  // 000[1]11
  double mid;
  for (int i = 0; i < 100; i++) {
    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)  == 6);
  assert(fabs(fbinary_search(-10.0, 10.0, pred5) - 1.2345) < 1e-
15);
  return 0;
}
```

Ternary Search

```
/*

Given a unimodal function f(x) taking a single double argument,
find its global
maximum or minimum point to a 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 the function
to be correct
and deterministic, such an 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 the function
to be correct
and deterministic, such an x must exist and be unique.

Time Complexity:
- O(log n) calls will be made to f(), where n is the distance
between lo and hi
  divided by the specified 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-12;
  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-12;
  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 <cassert>
#include <cmath>

bool equal(double a, double b) {
  return fabs(a - b) < 1e-7;
}

// 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) ~ (0.105,
88.063).
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(equal(ternary_search_min(-1000, 1000, f1), -2));
  assert(equal(ternary_search_max(-1000, 1000, f2), 2.0/19));
  assert(equal(ternary_search_min(-1000, 1000, f3), 30));
  return 0;
}
```

Hill Climbing

```
/*

Given a continuous function f(x, y) to double and a (possibly
arbitrary) initial
guess (x0, y0), return a potential global minimum found through
hill-climbing.
Optionally, two double pointers critical_x and critical_y may be
passed to store
the input points to f at which the returned minimum value is
attained.

Hill-climbing is a heuristic which starts at the guess, then
considers taking
a single step in each of a fixed number of directions. The
direction with the
best (in this case, minimum) value is chosen, and further steps
```

```
are taken in it
until the answer no longer improves. When this happens, the step
size is reduced
and the same process repeats until a desired absolute error is
reached. The
technique's success heavily depends on the behavior of f and the
initial guess.
Therefore, the result is not guaranteed to be the global minimum.

Time Complexity:
- O(d log n) call will be made to f, where d is the number of
directions
  considered at each position and n is the search space that is
approximately
  proportional to the maximum possible step size divided by the
minimum possible
  step size.

Space Complexity:
- O(1) auxiliary.

*/

#include <cstdlib>
#include <cmath>

template<class ContinuousFunction>
double find_min(ContinuousFunction f, double x0, double y0,
                double *critical_x = NULL, double *critical_y =
NULL) {
  static const double PI = acos(-1.0);
  static const double STEP_MIN = 1e-9, STEP_MAX = 1e6;
  static const int NUM_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 < NUM_DIRECTIONS; i++) {
      double a = 2.0*PI*i / NUM_DIRECTIONS;
      double x2 = x + step*cos(a), y2 = y + step*sin(a);
      double value = f(x2, y2);
      if (best > value) {
        best_x = x2;
        best_y = y2;
        best = value;
        found = true;
      }
    }
    if (!found) {
      step /= 2.0;
    } else {
```

```
      x = best_x;
      y = best_y;
      res = best;
    }
  }
  if (critical_x != NULL && critical_y != NULL) {
    *critical_x = x;
    *critical_y = y;
  }
  return res;
}

/*** Example Usage ***/

#include <cassert>
#include <cmath>

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

// Paraboloid with global minimum 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;
}
```

Convex Hull Trick (Semi-Dynamic)

```
/*

Given a set of pairs (m, b) specifying lines of the form y = mx +
b, process a
set of x-coordinate queries each asking to find the minimum y-
value when any of
the given lines are evaluated at the specified x. For each
add_line(m, b) call,
m must be the minimum m of all lines added so far. For each
query(x) call, x
must be the maximum x of all queries made so far.

The following implementation is a concise, semi-dynamic version of
the convex
hull optimization technique. It supports an an interlaced sequence
of add_line()
```

```
and query() calls, as long as the preconditions of descending m
and ascending x
are satisfied. As a result, it may be necessary to sort the lines
and queries
before calling the functions. In that case, the overall time
complexity will be
dominated by the sorting step.

Time Complexity:
- O(n) for any interlaced sequence of add_line() and query()
calls, where n is
  the number of lines added. This is because the overall number of
steps taken
  by add_line() and query() are respectively bounded by the number
of lines.
  Thus a single call to either add_line() or query() will have an
amortized O(1)
  running time.

Space Complexity:
- O(n) for storage of the lines.
- O(1) auxiliary for add_line() and query().

*/

#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 query(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 ***/

#include <cassert>

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

Convex Hull Trick (Fully Dynamic)

```
/*

Given a set of pairs (m, b) specifying lines of the form y = mx +
b, process a
set of x-coordinate queries each asking to find the minimum y-
value when any of
the given lines are evaluated at the specified x. To instead have
the queries
optimize for maximum y-value, call the constructor with
query_max=true.

The following implementation is a fully dynamic variant of the
convex hull
optimization technique, using a self-balancing binary search tree
(std::set) to
support the ability to call add_line() and query() in any desired
order.

Time Complexity:
- O(n) for any interlaced sequence of add_line() and query()
calls, where n
  is the number of lines added. This is because the overall number
of steps
  taken by add_line() and query() are respectively bounded by the
number of
  lines. Thus a single call to either add_line() or query() will
have an O(1)
  amortized running time.

Space Complexity:
- O(n) for storage of the lines.
```

```
- O(1) auxiliary for add_line() and query().

*/

#include <limits>
#include <set>

class hull_optimizer {
  struct line {
    long long m, b, value;
    double xlo;
    bool is_query, query_max;

    line(long long m, long long b, long long v, bool is_query,
bool query_max)
        : m(m), b(b), value(v), xlo(-
std::numeric_limits<double>::max()),
          is_query(is_query), query_max(query_max) {}

    double intersect(const line &l) const {
      if (m == l.m) {
        return std::numeric_limits<double>::max();
      }
      return (double)(l.b - b)/(m - l.m);
    }

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

  std::set<line> hull;
  bool query_max;

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

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

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

  bool irrelevant(hulliter it) const {
    if (!has_prev(it) || !has_next(it)) {
      return false;
    }
```

```
    hulliter prev = it, next = it;
    --prev;
    ++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 value = it->intersect(query_max ? *--it2 : *++it2);
    line l(*it);
    l.xlo = value;
    hull.erase(it++);
    return hull.insert(it, l);
  }

 public:
  hull_optimizer(bool query_max = false) : query_max(query_max) {}

  void add_line(long long m, long long b) {
    line l(m, b, 0, false, query_max);
    hulliter it = hull.lower_bound(l);
    if (it != hull.end() && it->m == l.m) {
      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 query(long long x) const {
    line q(0, 0, x, true, query_max);
    hulliter it = hull.lower_bound(q);
    if (query_max) {
      --it;
    }
    return it->m*x + it->b;
  }
};

/*** Example Usage ***/

#include <cassert>

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

Cycle Detection (Floyd's)

```
/*

Given a function f mapping a set of integers to itself and an x-
coordinate in
the set, return a pair containing the (position, length) of a
cycle in the
sequence of numbers obtained from repeatedly composing f with
itself starting
with the initial x. Formally, since f maps a finite set S to
itself, some value
is guaranteed to eventually repeat in the sequence:
  x[0], x[1]=f(x[0]), x[2]=f(x[1]), ..., x[n]=f(x[n - 1]), ...

There must exist a pair of indices i and j (i < j) such that x[i]
= x[j]. When
this happens, the rest of the sequence will consist of the
subsequence from x[i]
to x[j − 1] repeating indefinitely. The cycle detection problem
```

asks to find
such an i, along with the length of the repeating subsequence. A well-known
special case is the problem of cycle-detection in a degenerate linked list.

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(m + n) per call to find_cycle_floyd(), where m is the smallest index of the
  sequence which is the beginning of a cycle, and n is the cycle's length.

Space Complexity:
- O(1) auxiliary.

*/

```
#include <utility>

template<class IntFunction>
std::pair<int, int> find_cycle_floyd(IntFunction f, int x0) {
  int tortoise = f(x0), hare = f(f(x0));
  while (tortoise != hare) {
    tortoise = f(tortoise);
    hare = f(f(hare));
  }
  int start = 0;
  tortoise = x0;
  while (tortoise != hare) {
    tortoise = f(tortoise);
    hare = f(hare);
    start++;
  }
  int length = 1;
  hare = f(tortoise);
  while (tortoise != hare) {
    hare = f(hare);
    length++;
  }
```

```
  return std::make_pair(start, length);
}

/*** Example Usage ***/

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

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

Cycle Detection (Brent's)

```
/*

Given a function f mapping a set of integers to itself and an x-coordinate
in
the set, return a pair containing the (position, length) of a cycle in the
sequence of numbers obtained from repeatedly composing f with itself
starting
with the initial x. Formally, since f maps a finite set S to itself, some
value
is guaranteed to eventually repeat in the sequence:
```

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

There must exist a pair of indices i and j (i < j) such that x[i] = x[j]. When
this happens, the rest of the sequence will consist of the subsequence from
x[i]
to x[j - 1] repeating indefinitely. The cycle detection problem asks to
find
such an i, along with the length of the repeating subsequence. A well-known
special case is the problem of cycle-detection in a degenerate linked list.

While Floyd's cycle-finding algorithm finds cycles by simultaneously
moving two
pointers at different speeds, Brent's algorithm keeps the tortoise
pointer
stationary in every iteration, only teleporting it to the hare pointer at
every
power of two. The smallest power of two at 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 calls made to f.

Time Complexity:
- O(m + n) per call to find_cycle_brent(), where m is the smallest index of the
 sequence which is the beginning of a cycle, and n is the cycle's length.

Space Complexity:
- O(1) auxiliary.

*/

#include <utility>

template<class IntFunction>
std::pair<int, int> find_cycle_brent(IntFunction f, int x0) {
 int power = 1, length = 1, 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;

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

Binary Exponentiation

```
/*

Given three unsigned 64-bit integers x, n, and m, powmod() returns x raised to
the power of n (modulo m). mulmod() returns x multiplied by n (modulo m).
Despite the fact that both functions use unsigned 64-bit integers for
arguments
and intermediate calculations, arguments x and n must not exceed 2^63 - 1
(the
maximum value of a signed 64-bit integer) for the result to be correctly
computed without overflow.

Binary exponentiation, also known as exponentiation by squaring,
decomposes the
exponentiation into a logarithmic number of multiplications while
avoiding
overflow. To further prevent overflow in the intermediate squaring
computations,
multiplication is performed using a similar principle of repeated
addition.

Time Complexity:
- O(log n) per call to mulmod() and powmod(), where n is the second argument.

Space Complexity:
- O(1) auxiliary.

*/

typedef unsigned long long uint64;

uint64 mulmod(uint64 x, uint64 n, uint64 m) {
 uint64 a = 0, b = x % m;
 for (; n > 0; n >>= 1) {
  if (n & 1) {
   a = (a + b) % m;
  }
  b = (b << 1) % m;
 }
 return a % m;
}

uint64 powmod(uint64 x, uint64 n, uint64 m) {
 uint64 a = 1, b = x;
 for (; n > 0; n >>= 1) {
  if (n & 1) {
   a = mulmod(a, b, m);
  }
```

```
  b = mulmod(b,b,m);
 }
 return a % 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;
}
```