CPSC 490

Input

Input will always arrive on `stdin`. You may assume input is well-formed with respect to the problem specification; inappropriate input (e.g. text where a number was specified, number out of range, string too long) may be handled by crashing. Normally you want to read input a "word" at a time:

C++	Java

<pre>#include <iostream>
#include <string>
using namespace std;
...
int i; string s; double d;
cin >> i >> s >> d;
 	
// Check for end-of-file
if (cin >> var) // read OK
else // EOF</pre>
	<pre>import java.util.*;
	...
	Scanner sc = new Scanner(System.in);
	int i = sc.nextInt();
	String s = sc.next();
	double d = sc.nextDouble();
	
	// Check for end-of-file
	if (sc.hasNext()) // can read
	else // EOF</pre>

Occasionally you want to deal with a line at a time instead (e.g. dealing with strings that might have whitespace in them or variable numbers of words without counts specified):

C++	Java

<pre>#include <iostream>
#include <string>
using namespace std;
...
string line;
getline(cin, line);</pre>
	<pre>import java.util.*;
	...
	Scanner sc = new Scanner(System.in);
	String line = sc.nextLine();</pre>

NOTE: If you use both word-based and line-based functions in the same program, if you reach the end of a line using the word-based functions, the line-based functions will see an empty line before the following line. Therefore, if you do this, call the line function one extra time and discard its output. For example, given the following input file:

```
17
The quick brown fox jumps over the lazy dog.
```

the following code will parse it properly:

C++	Java

<pre>#include <iostream>
#include <string>
using namespace std;
...
int number;
string sentence;
cin >> number;
// Discard the pseudo-line
getline(cin, sentence);
// Read the real line
getline(cin, sentence);</pre>
	<pre>import java.util.*;
	...
	Scanner sc = new Scanner(System.in);
	int number = sc.nextInt();
	// Discard the pseudo-line
	sc.nextLine();
	// Read the real line
	String sentence = sc.nextLine();</pre>

Output

Output always goes to `stdout`. For most problems, the output your program generates must be *exactly* byte-for-byte identical to the "correct" output. This means the following will result in errors:

- wrong answers
- incorrect spelling
- incorrect capitalization
- incorrect number of decimal places
- incorrect use of whitespace, including blanks at end of line
- confusing "a blank line between test cases" with "a blank line after each test case"

The following example prints out the following (excluding the braces): {`$1234 hello world 27.35`} and then moves to the next line.

C++

```cpp
#include <iostream>
using namespace std;
...
int dollars = 1234;
double foo = 27.35;
string word = "hello";
// # decimal places for doubles &
// floats
cout << fixed << setprecision(2);
cout << '$' << dollars << ' '
  << word << " world " << foo << endl;
```

Java

```java
import java.util.*;
...
int dollars = 1234;
double foo = 27.35;
String word = "hello";
// # decimal places for doubles & floats
// inside format string
System.out.printf("$%d hello %s %.2f\n",
  dollars, word, foo);
```

Containers: Lists

A *list* stores a collection of items (of the same type) in an order specified as the items are added. Each item has a position in the list, and is normally identified by its position (positions are counted starting from zero). Lists generally allow duplicates.

If you have a reasonable upper bound on the amount of data and you don't need to store the exact size (e.g. because it's implied by other parts of the problem), consider using a simple array declared to be the maximum size. In Java, dynamically allocating arrays with sizes not known at compile time can be useful; in C++ you have to remember to free such arrays manually so vectors are often a better choice.

One more advanced list is the *vector*, which is implemented on top of an array which is reallocated as needed. Vectors are fast for finding an element by its position and adding and removing elements at the end of the list, but slow for finding an element by its value and adding and removing elements other than at the end of the list.

C++

```cpp
#include <vector>
using namespace std;
...
vector<int> numbers;
numbers.push_back(5);
numbers.push_back(7);

// Insert in the middle (SLOW!):
numbers.insert(numbers.begin() + 1, 6);

assert(numbers.size() == 3);
assert(numbers[0] == 5);
assert(numbers[1] == 6);
assert(numbers[2] == 7);
numbers.clear();
assert(numbers.empty());
```

Java

```java
import java.util.*;
...
List<Integer> numbers =
  new ArrayList<Integer>();
numbers.add(5);
numbers.add(7);

// Insert in the middle (SLOW!):
numbers.add(1, 6);

assert(numbers.size() == 3);
assert(numbers.get(0) == 5);
assert(numbers.get(1) == 6);
assert(numbers.get(2) == 7);
numbers.clear();
```

Another type of list is the *linked list*, which is implemented as a string of nodes each of which knows how to find the node before and after itself. Linked lists are fast for inserting and removing elements anywhere (at the beginning or end or any location at which one holds an *iterator*) and iterating forward or backwards through all elements, but slow for finding an element by its position or value.

C++

```
#include <list>
using namespace std;
...
list<int> numbers;
numbers.push_back(7);
numbers.push_front(5);

// Get iterator by position (SLOW!):
list<int>::iterator iter
  = numbers.begin() + 1;
assert(*iter == 7);

// Insert before iterator (fast):
numbers.insert(iter, 6);
assert(numbers.size() == 3);

// Get elements by position (SLOW!):
assert(*(numbers.begin() + 0) == 5);
assert(*(numbers.begin() + 1) == 6);
assert(*(numbers.begin() + 2) == 7);
```

Java

```
import java.util.*;
...
List<Integer> numbers =
  new LinkedList<Integer>();
numbers.add(7); // at end
numbers.add(0, 5); // at start

// Get iterator by position (SLOW!):
ListIterator<Integer> iter =
  numbers.listIterator(1);
assert(iter.next() == 7);
// The iterator moved. Move it back.
iter.previous();

// Insert before iterator (fast):
iter.add(6);
assert(numbers.size() = 3);

// Get elements by position (SLOW!):
assert(numbers.get(0) == 5);
assert(numbers.get(1) == 6);
assert(numbers.get(2) == 7);
```

These are the general-purpose lists. There is also a *deque*, or double-ended queue, which provides slightly different services in C++ and in Java. In C++, `deque` acts very like a vector except allowing fast adding and removing of elements at both ends (not only the back). In Java, `ArrayDeque` (added in version 1.6.0) also allows adding and removing elements at both ends, but does not allow efficiently accessing an element by its position. In this course, deques may be used in certain special cases where the front-and-back semantics are required, but we will not be accessing elements by position.

C++

```
#include <deque>
using namespace std;
...
deque<int> numbers;
numbers.push_back(7);
numbers.push_back(8);
numbers.push_front(6);
numbers.push_front(5);
assert(numbers.size() == 4);
for (int i = 5; i <= 8; i++) {
  assert(numbers.front() == i);
  numbers.pop_front();
}
```

Java

```
import java.util.*;
...
Deque<Integer> numbers =
  new ArrayDeque<Integer>();
numbers.addLast(7);
numbers.addLast(8);
numbers.addFirst(6);
numbers.addFirst(5);
assert(numbers.size() == 4);
for (int i = 5; i <= 8; i++) {
  assert(numbers.peekFirst() == i);
  numbers.removeFirst();
}
```

Finally, there is the *queue*, which is similar to a deque but is intended for use in situations where all elements are added at one end and removed at the other—in other words, a FIFO. In C++, `queue` is a class which uses a template parameter to choose which implementation to use (typical choices are `list` and `deque`, with `deque` being the default). In Java, `Queue` is an interface implemented by both `LinkedList` and `ArrayDeque`.

C++

```cpp
#include <queue>
using namespace std;
...
queue<int> numbers;
numbers.push(5);
numbers.push(6);
numbers.push(7);
numbers.push(8);
assert(numbers.size() == 4);
for (int i = 5; i <= 8; i++) {
  assert(numbers.front() == i);
  numbers.pop();
}
```

Java

```java
import java.util.*;
...
Queue<Integer> numbers =
  new ArrayDeque<Integer>();
numbers.add(5);
numbers.add(6);
numbers.add(7);
numbers.add(8);
assert(numbers.size() == 4);
for (int i = 5; i <= 8; i++) {
  assert(numbers.peek() == i);
  numbers.remove();
}
```

Containers: Maps

Maps are collections of key-value pairs, optimized for looking up the value associated with a known key. There are two variants: *hashtables* perform most operations faster (many in constant time) but do not keep their keys in sorted order, while *trees* are slower (performing most operations in logarithmic time) but keep their keys sorted, which may be useful for iteration. Maps normally do not allow duplicate keys. In general trees are fast enough for most purposes, and defining hash functions in C++ is somewhat nonintuitive. Therefore, we will use trees in this class:

C++

```cpp
#include <map>
using namespace std;
...
map<string, int> days;

// Add elements:
days["Monday"]    = 1;
days["Tuesday"]   = 2;
days["Wednesday"] = 3;
days["Thursday"]  = 4;
days["Friday"]    = 5;
days["Saturday"]  = 6;
days["Sunday"]    = 7;

// Check presence:
assert(days.count("Monday")   == 1);
assert(days.count("NotADay") == 0);

// Get values:
assert(days["Monday"]  == 1);
assert(days["Tuesday"] == 2);

// Iterate keys in order of <:
map<string, int>::iterator i, iend;
for (i = days.begin(),
     iend = days.end(); i != iend; ++i) {
  string key = i->first;
  int value = i->second;
}

// Erase by key.
days.erase("Tuesday");
```

Java

```java
import java.util.*;
...
Map<String, Integer> days =
  new TreeMap<String, Integer>();

// Add elements:
days.put("Monday",    1);
days.put("Tuesday",   2);
days.put("Wednesday", 3);
days.put("Thursday",  4);
days.put("Friday",    5);
days.put("Saturday",  6);
days.put("Sunday",    7);

// Check presence:
assert(days.containsKey("Monday"));
assert(!days.containsKey("NotADay"));

// Get values:
assert(days.get("Monday")  == 1);
assert(days.get("Tuesday") == 2);

// Iterate keys in order of Comparable:
for (Map.Entry<String, Integer> i
    : days.entrySet()) {
  String key = i.getKey();
  int value = i.getValue();
}

// Erase by key.
days.remove("Tuesday");
```

Containers: Sets

A *set* is a map without values: its only purpose is to contain values (not allowing duplicates) and permit efficient checking of whether or not a value is contained in the set. As with maps, typical implementations are hashtable-based or tree-based. Again, hashtables keep their values unordered but are faster, while tree-based sets keep their values in their natural order, and we will use tree sets in most cases for this class to avoid defining hash functions.

C++

```cpp
#include <set>
using namespace std;
...
set<string> words;

// Add elements:
words.insert("Foo");
words.insert("Bar");
words.insert("Baz");

// Check presence:
assert(words.count("Foo") == 1);
assert(words.count("Quux") == 0);

// Iterate elements in order of <:
set<string>::iterator i, iend;
for (i = words.begin(),
     iend = words.end();
     i != iend; ++i) {
  string elem = *i;
}

// Erase by element.
days.erase("Baz");
```

Java

```java
import java.util.*;
...
Set<String> words =
  new TreeSet<String>();

// Add elements:
words.add("Foo");
words.add("Bar");
words.add("Baz");

// Check presence:
assert(days.contains("Foo"));
assert(!days.contains("Quux"));

// Iterate elements in order of
// Comparable:
for (String elem : words) {
  // ...
}

// Erase by element.
words.remove("Baz");
```

Sorting: Natural and Custom Orders

Every data type can have a *natural ordering*, which is used to determine in which order objects of that type should be sorted. Data types can also have any number of *custom orderings*, which can be explicitly used to sort objects in a different order. Functions are available in the standard libraries to efficiently sort arrays and vectors (quicksort).

C++

```cpp
#include <set>
#include <map>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;
...

// Define a custom type:
class mytype {
  public:
    int foo;
    string bar;
};

// Natural ordering:
// Returns true if x < y,
// false if x >= y.
bool operator<(const mytype &x,
    const mytype &y) {
  if (x.foo != y.foo)
    return x.foo < y.foo;
  else return x.bar < y.bar;
}

// Custom ordering:
bool otherorder(const mytype &x,
    const mytype &y) {
  if (x.bar != y.bar)
    return x.bar < y.bar;
  else return x.foo < y.foo;
}

// Custom order set/map:
set<mytype, typeof(&otherorder)>
  s(&otherorder);
map<mytype, string,
  typeof(&otherorder)> m(&otherorder);

// Sorting:
vector<mytype> vec;
mytype ary[27];
sort(v.begin(), v.end());
sort(v.begin(), v.end(), &otherorder);
sort(ary, ary + 27);
sort(ary, ary + 27, &otherorder);
```

Java

```java
import java.util.*;
...
// Define a custom type:
class MyType implements Comparable {
  public int foo;
  public String bar;

  // Natural ordering: returns >0 for >,
  // 0 for =, <0 for <
  public int compareTo(MyType other) {
    if (foo != other.foo)
      return foo - other.foo;
    else
      return bar.compareTo(other.bar);
  }
}

// Custom ordering:
class OtherOrder implements
    Comparator<MyType> {
  public int compare(MyType x, MyType y)
  {
    if (!x.bar.equals(y.bar))
      return x.bar.compareTo(y.bar);
    else
      return x.foo - y.foo;
  }
}

// Custom order set/map:
new TreeSet<MyType>(new OtherOrder());
new TreeMap<MyType, String>(
  new OtherOrder());

// Sorting:
List<MyType> vec;
MyType[] ary;
Collections.sort(vec);
Collections.sort(vec, new OtherOrder());
Arrays.sort(ary);
Arrays.sort(ary, new OtherOrder());
```