About This Contest Sun Oct 12 04:05:44 PM EDT 2008

You should have the following 4 handouts:

(1) Help Files: this handout (index is next item).
(2) Demo Files: demonstration code.
(3) Schedule: one page, with room numbers.
(4) Problems: not given out till start of contest.

The most important thing for new teams is that they run their program against the Sample Input and verify that their program produces the Sample Output, BEFORE they submit. For program 'ppp', this is most easily done by typing the Sample Input into ppp.in, running 'make', and looking at the program output in ppp.out. Remember, 'make' sends ppp.in to your program's standard input and copies your programs standard output to ppp.out; you MUST NOT open any files in your program.

The command 'help' will display this file. You have also been given a printout of the important help and demonstration files.

This is a 'formal' contest. The command

```
help formal_contest
```

displays more details about formal contests.

To display a list of the help files available use the command 'help index'. Note that if something in the index has the name 'help/topic' you use the command 'help topic' to view it.

The contest managers will log in for you. You will NOT be given a password. If you have an xterminal, the xcsh command will make more windows: see 'help xterminals'.

The contest managers will inform you in writing which account you will be using. It is your responsibility to double check that the contest managers have logged you into the right account.

The command 'hpcm_get' will get a demonstration problem into your directory ~/demos/count (where ~ denotes your home directory). Read ~/demos/count/README and ~/demos/count/Makefile and practice submitting the demonstration problem.

Hpcm_get will also get all the contest problems into your '~/problems' directory AFTER the contest officially starts. At the contest start time you will be given a printout of all the problem descriptions. The command 'help problems' will display a list of all the problems AFTER the contest starts. Also see 'help solving'.

The command 'scoreboard' will display the scoreboard. The time of a problem is the time between the start time of the contest and the time a solution is submitted. See 'help scoreboard'.

There is no time penalty for an incorrect submission for this contest.

To send a question to the contest manager DO NOT try to e-mail the manager directly. Instead edit the question into a file qqq and use the command:

```
hpcm_sendmail < qqq
```

Output of the print commands (see 'help print') will be brought to you. Please do NOT try to find the printer. For more information see 'help print'.

You CANNOT use the alternative submit commands that make 'in-submit', 'inout-submit', or 'solution-submit'.

You CANNOT use a web browser. You may use any printed material you like, but cannot communicate electronically with the world outside the contest.

The 'javahelp' and 'stlhelp' commands give access to on-line Java and C++ STL documentation: see 'help java' and 'help c++'.

The day after the contest proper, the contest will be converted to a different type of contest. You will then be given your account name and password, and can log in

and in a sense continue the contest. 'in-submit', 'inout-submit', and 'solution-submit' will be allowed. The judging will be completely automatic. You will be able to print by making a ~/PRINTER file: see 'help print'. The scoreboard will use the submission type ('submit', 'in-submit', 'inout-submit') of incorrect submissions in place of time. A correct submission with no previous incorrect submissions will have a score of 100.0, and the penalties for incorrect submissions depend on submission type as follows:

incorrect 'submit'	10%
incorrect 'in-submit'	20%
incorrect 'inout-submit'	30%

See 'help scoreboard', feedback contests, for details.

There is a 'bug' in some of the software we are using that scrambles the screen sometimes. You can get it unscrambled by typing control-L. So far only javahelp and stlhelp seem to be affected.

Good Hunting!

```
Help Index                  Sun Oct 12 04:08:27 PM EDT 2008


To display the file listed as ‘help/topic’ below use the
command:
                      help topic

E.g., ‘help index’ displays this file.


Indices:
--------

    help/index
        This file.

    help/demos
        Index of contest demonstration problems.

    help/problems
        Index of contest problems.



Introductory Information:
-------------------------


    help/this_contest
        Information about this contest.


Contest Description
------- -----------

    help/formal_contest
        About formal contests, including logging in.

    help/xterminals
        How to make windows if you are using an
        xterminal emulator.

Solving Problems:
-----------------

    help/solving
        How to write a solution file, how to test it,
        how to submit it, if you have been given a
        contestant account.

    help/input
        Advice on input formats.

    help/output
        Advice on output formats.

    help/print
        How to print files.

    help/commonlisp
        How to write COMMONLISP solutions.


Scores and the Scoreboard:
--------------------------

    help/scores
        How to find out your scores, and also how to
        ask questions of the judge.

    help/scoreboard
        Interpreting the scoreboard notation.


Advice:
-------

    help/c++
        Special C++ advice.

    help/java
        Special JAVA advice.

    help/advice
        Polonic advice on how to select problems and
        find errors.
```

Advice on Specific Kinds of Problems
------ -- -------- ----- -- --------

Formal Contest Help Thu Sep 28 17:14:22 EDT 2006

A formal contest is a contest in which you are provided with an account and at which problem descriptions are passed out at the start of the contest.

Normally all contestants run on the same fast computer. This means that all contestants are subject to the same load. There may be differences in terminals assigned to contestants. To ensure fairness, terminals are assigned to contestants at random (e.g., by using dice throws).

Rules for Formal Contests:
----- --- ------ --------

The contest managers will log in for you. You may or may not be given an X-Terminal equivalent. If you are, you can make more windows using xcsh: see 'help xterminals'. If not, you must ask the contest managers to make more windows for you. Usually in this case all teams are given the same number of windows to start with: typically 4. You will NOT be given any password.

For formal contests you are assigned a location and account name. Please type 'who am i' to be sure you have the right account.

In a formal contest your printouts will be brought to you. Do NOT try to find the printer. If a printout has not been brought to you promptly, ask the contest support personnel to check the printer.

During a formal contest you may not use web browsers or electronic communication that is not directly part of the contest. You may bring and use any printed material you like. There is a contest command, javahelp, for on-line Java documentation: see 'help java'. There is a contest command, stlhelp, for viewing on-line C++ Standard Template Library documentation, if this documentation is available: see 'help c++'.

During a formal contest the alternative submit commands 'make in-submit', 'make inout-submit', and 'make solution-submit' are NOT allowed.

A few days or so after the contest ends you will be given the name and password of your contest account so you can collect the code you wrote. At this point the contest will be reconstituted as an untimed contest, so you can additionally submit new code to the autojudge, though of course it will not count in the already finished formal contest.

File: formal_contest
Author: Bob Walton <walton@deas.harvard.edu>
Date: See top of file.

RCS Info (may not be true date or author):

$Author: walton $
$Date: 2006/09/28 21:15:16 $
$RCSfile: formal_contest,v $
$Revision: 1.10 $

X-Terminals Help Tue Apr 25 06:11:29 EDT 2006

This file is for those who are using an account provided by the contest managers, and who are logging into that account using ssh with X-windows. Note that to do this from UNIX you may need to use the -X option to ssh, as in

```
ssh -X someaccount@somecomputer
```

Creating Windows with Xcsh

To create a new window you may use the command:

xcsh	Start a new X terminal window running csh.
	The new window tends to form directly over the top of the leftmost existing window, so one of these windows has to be moved.

Xcsh takes options that change the font and location of the window formed. To try a different font, use the command

```
xcsh PPm
```

where PP is the point size (e.g., 12, 16, 18, 24) desired and ‘m’ indicates you want a medium font. This command will list the available font names that begin with PPm, and any of these can then be used as an option to xcsh. For example, if the command ‘xcsh 18m’ outputs ‘use one of 18ml 18ma 18mb’, the command ‘xcsh 18ml’ would use an 18 point medium font of style ‘l’.

If you want a bold font, replace ‘m’ in ‘PPm’ by ‘b’.

To get more complete documentation of xcsh type the command ‘xcsh -doc’.

Destroying a Window

To destroy a window just type the command ‘exit’, or if that does not work, ‘logout’.

BUT BE CAREFUL not to destroy your original (root) window as doing that may destroy all other windows.

File: xterminals
Author: Bob Walton <walton@deas.harvard.edu>
Date: See top of file.

RCS Info (may not be true date or author):

```
$Author: hc3 $
$Date: 2006/04/25 10:12:13 $
$RCSfile: xterminals,v $
$Revision: 1.8 $
```

Solving Problems Help Sat Dec 23 02:10:02 EST 2006

This help file is for contestants who have been given an account, or for contestants using their own accounts with the tools described in the email_unix_tools help file. Contestants using their own accounts without the tools just mentioned should read the email_solving help file instead of this help file.

What Problems?

When you first log in, there are no problems. You have to get the problems with the 'hpcm_get' command.

To get the demonstration problems before the contest starts, you can just type the command

```
hpcm_get
```

The demonstration problems are put in the 'demos' subdirectory of your home directory (you can always get to this with the 'cd ~/demos' UNIX command, where '~' denotes your home directory).

In a formal contest, you need to REPEAT the above 'hpcm_get' command again, JUST ONCE, AFTER the contest starts, to get all the problems.

In some informal contests you need to get each problem one at a time, using the command

```
hpcm_get problems/pppp
```

to get the problem named 'pppp'.

In other contests, the first 'hpcm_get' gets ALL the problems, not just the demonstration problems.

The contest problems are put in the 'problems' subdirectory of your home directory (you can always get to this with the 'cd ~/problems' UNIX command).

Each problem has its own subdirectory in the 'problems' subdirectory. Thus a problem named 'pppp' would have a problem directory named '~/problems/pppp'.

You can display an index of the problems and sometimes an estimate of their relative difficulty by using the command:

```
help problems
```

Asking Questions:

You may think that a problem description is ambiguous, and ask the judges for clarification, though you should be careful to think the situation through thoroughly first. You may e-mail such questions to the judges, but the method to do this from an account provide to you is a little weird. First, edit your email into a file of your own choosing. If your first line has the form

```
Subject: ....
```

and the next line is completely empty (no spaces), then the first line will be the subject line of the message. Then

```
hpcm_sendmail < filename
```

to send the file as email to the judges.

Note that the subject must NOT begin with the words 'submit' or 'get'.

If you ask for a clarification by e-mail, and a significant answer is given, that answer will be given by posting it on the scoreboard so everyone can see it. You will not receive email back in this case.

Working on a Problem

For a problem named ‘pppp’ you should write a file named ‘pppp.c’, ‘pppp.cc’, ‘pppp.java’, or ‘pppp.lsp’ in your ‘problems/pppp’ directory. The suffix of the file you write determines the programming language you are using:

.c	for C
.cc	for C++
.java	for JAVA
.lsp	for COMMONLISP

You must write only ONE of these files; you CANNOT have a solution in two different languages at once.

The description of a problem named ‘pppp’ is in a file named ‘pppp.txt’, ‘pppp.html’, ‘pppp.htm’, ‘pppp.ps’, or or similar printable file. This file may or many not be given to you electronically. In a formal contest, it is not, but instead a printout of the file is given to you at the start of the contest. In informal contests, the problem description file is gotten into the problem directory by hpcm_get, and you must print it yourself or look at it with an editor, browser, or postscript display program. In untimed contests, the problem description file is typically a subpage of the contest web page, although the file may also appear in the problem directory.

Your program should be written to take input from the terminal and put output to the terminal. In C this means using functions such as ‘gets’, ‘scanf’, and ‘printf’ that implicitly use ‘stdin’ for input and ‘stdout’ for output. In C++ this means using ‘cin’ and ‘cout’. In JAVA this means using ‘System.in’ and ‘System.out’. In COMMONLISP this means using ‘*standard-input*’ and ‘*standard-output*’.

You should NOT, repeat NOT, write output to the standard error output, ‘stderr’, ‘cerr’, ‘System.err’, or ‘*standard-error*’. Such output will result in a ‘Program Crashed’ or ‘Cpu Time Limit Exceeded’ score.

You should NOT under any circumstances open a file in your code. In the judge’s program execution environment, output operations on opened files fail, in order to protect the judge’s software from wayward submissions. Since you will likely not be checking for output failures in your code, your code will mysteriously produce no output in the judge’s environment if you open an output file, and will be scored ‘Program Crashed’ if your program terminates, or ‘Cpu Time Limit Exceeded’ if your program fails to terminate because all your input instructions have become no-operations.

When the autojudge executes your program, it will be called with NO program arguments. You can use this fact to write out debugging information to the standard output IF your program is called with one or more program arguments.

If you edit an input file named ‘pppp.in’ in the problem directory, then the following UNIX commands can be executed in the problem directory. You can try them out using the demonstration problem in the ‘~/demos/count’ directory. You can try ‘make submit’ in this directory, and the judges will ‘judge’ the demo submission, though of course it will not count. See the README file in this directory for more instructions.

make — Same as ‘make pppp.out’ (see below).

make pppp — Makes the binary program file ‘pppp’ by running gcc on pppp.c, or g++ on pppp.cc, javac on pppp.java, or a commonlisp compiler on pppp.lsp, depending upon which of pppp.c, pppp.cc, pppp.java, or pppp.lsp exist. Does nothing if ‘pppp’ is more up to date than pppp.c, pppp.cc, etc.

In addition to making ‘pppp’, other files may be made for some languages, such as pppp.class which is made from pppp.java, or pppp.fas which is made from pppp.lsp.

make pppp.out — Makes ‘pppp’ as above and then runs it with standard input coming from the file pppp.in. Puts the standard output in the file pppp.out, and then copies that to the screen. Puts any standard error output to the screen before the standard output from pppp.out. Does nothing, however, if pppp.out is more recent than both pppp.in and pppp.

make pppp.debug — Ditto but runs ‘pppp debug’ instead of ‘pppp’ and puts the standard output in the file ‘pppp.debug’ instead of ‘pppp.out’. You should write your program to output debugging information to the standard output if the program is given any arguments.

make debug — Same as ‘make pppp.debug’.

make submit — Makes ‘pppp.out’ just to be sure that nothing crashes, and then e-mails pppp.c, pppp.cc, ppp.java, or pppp.lsp to the judges. Note the pppp.in file MUST exist to submit, to make pppp.out, but pppp.in can be the minimum needed to keep your program from crashing (often pppp.in can be empty).

There are alternative forms of this command used in untimed contests: see ‘Submission Alternatives’ below.

make clean — Removes ‘pppp’ and pppp.out.

The ‘make’ UNIX commands work because of the way the ‘Makefile’ file in the problem directory is written. It is this file that causes ‘pppp.in’ to be presented to your program as if it were input from a terminal, and causes output your program writes to a terminal to be stored in ‘pppp.out’ instead.

The ‘Makefile’ file contains some oddities resulting from the fact that judges, who use the same ‘Makefile’ as contestants, run solutions in a sandbox account that has permission problems accessing judge’s files. Any file, directory, or program that must be accessed from the sandbox account may need permissions making it accessible to everyone.

Common Mistakes

If you get a ‘Program Crashed’ or ‘Cpu Time Limit Exceeded’ score, check that you DO NOT OPEN any files. You should just read the standard input and write the standard output. If you open .in and .out files, it will work for you, but not for the judge. The judge runs your program in a protected mode in which opening a file fails because you do not have permissions.

Assuming you do not check for opening or input/output errors, all your input and output operations may turn into no-operations. If because input operations are no-operations you loop infinitely, you will get the score ‘Cpu Time Limit Exceeded’. Otherwise you will get the score ‘Program Crashed’, either because your program wrote no output since output operations are no-ops, or because JAVA checks for errors automatically and you get an uncaught input/output exception.

If ‘make submit’ tells you it cannot make a .in file, that is because you must edit a pppp.in file in order to keep ‘make submit’ happy. The pppp.in file can be the minimum needed to keep your program from crashing (often pppp.in can be empty).

It is important to run your program on the Sample Input and carefully check that the output produced matches the Sample Output. In particular, be sure any lines that begin or separate test cases are correct.

Submission Alternatives

In some contests, particularly untimed contests with ‘feedback’ scoring, there are alternative ways of submitting a problem:

`make submit`	Returns just score.
`make in-submit`	Returns score, and when practical, returns the judge’s input for the first failed test case.
`make inout-submit`	Ditto but returns both judge’s input and judge’s output for the first failed test case.
`make solution-submit`	Returns score, and if the score is ‘Completely Correct’, also returns the judge’s solution to the problem.

The scoreboard in a ‘feedback scored’ contest is not based on time, but is instead based on the kinds of submission you do when your solution is still incorrect. Using ‘make submit’ to submit an incorrect solution invokes the least penalty, using ‘make in-submit’ a greater penalty, and using ‘make inout-submit’ the greatest penalty. There is no penalty for a correct submission, and ‘make solution-submit’ behaves like ‘make submit’ for an incorrect submission. Timed contests and some other contests disallow most of these kinds of submit, in which case the disallowed submissions behave like ‘make submit’ (but also return a note saying they are not allowed).

If you have already submitted a correct solution, you can resubmit with a ‘solution’ qualifier to get the judge’s solution, without any affect on your score.

Timed contests and some other contests disallow most submission qualifiers, in which case the submissions with disallowed qualifiers behave as if they had no qualifier (except they also return a note saying the qualifier was disallowed).

Resource Limits

If you look at the ‘Makefile’ in the problem directory you will see that it contains memory and time resource limits which constrain your problem. Memory limits are typically several megabytes and time limits are typically 10 to 60 seconds. Harder problems require care to be sure you stay within these limits.

Most problems are such that solutions in C or C++ will run in a few seconds. If you have long running times, you may have chosen an algorithm too inefficient to avoid the time limit.

Debugging

The best way to debug is to include code that prints extra debugging information to the standard output if your program is passed any arguments. For C, such code might look like:

```
int debug;
#define dprintf if ( debug ) printf

int main ( int argc )
{
    debug = ( argc > 1 );       // # arguments = argc-1.
    . . .
    dprintf ( ... );
}
```

See the C++, java, and commonlisp help files for debugging in other languages.

The ‘make debug’ command (see above) calls your program with one argument and puts the output in ‘pppp.debug’.

For program crashes and infinite loops, it is quickest to use a debugger, if you have minimal familiarity with one of the available debuggers. A typical usage is

```
gdb pppp
run < pppp.in
... program crashes, or control-C is pressed ...
... to get out of an infinite loop ...
back            [this lists frames]
frame #         [# is number of correct frame]
list            [lists code near crash]
p EXP           [print value of expression EXP]
```

To debug, you do not have to use pppp.in and make pppp.out. You can just make pppp and run it as a UNIX command, typing input at it. BUT, to submit you must have some pppp.in file, though it can be the minimum needed to keep your program from crashing (often it can be a zero length file). Also, the resource limits will not be applied if you do not use pppp.in and 'make'.

The contest staff will answer questions about debuggers and editors as best they can.

Other Issues

The following help commands may be useful for:

Printing files:	help print
Understanding problem scores:	help scores
Understanding the scoreboard:	help scoreboard

File: solving
Author: Bob Walton <walton@deas.harvard.edu>
Date: See top of file.

RCS Info (may not be true date or author):

```
$Author: walton $
$Date: 2006/12/23 07:09:23 $
$RCSfile: solving,v $
$Revision: 1.20 $
```

```
Input Help                    Tue Apr 25 00:15:36 EDT 2006

Input comes from the standard input.  This is 'stdin' in
C, 'cin' in C++, 'System.in' in JAVA, and '*standard-
input*' in COMMONLISP.  These input streams do NOT have
to be opened; you should NOT open any files.

You may assume that input is correctly formatted, except
for the rare problem where you are told to produce
special output if the input contains a formatting error.
Do not waste time checking for input errors when you do
not have to.

In some problems newlines are NOT syntactically signifi-
cant, and newlines are just like any other space charac-
ter on input.

If you are writing in C you may need the ungetc func-
tion and the gets function.

If you are writing in C++ you may need cin.peek() and
cin.getline(buffer, sizeof buffer).

If you are writing in JAVA you may need either the
StreamTokenizer or StringTokenizer classes.  See the
code in demos/count/count1.java and demos/javaio/
javaio.java.

If you are writing in COMMONLISP, you may need calls
like
        (read-line t nil 'eof)
        (read-char t nil 'eof)
        (peek-char nil t nil 'eof)
```

```
File:           input
Author:         Bob Walton <walton@deas.harvard.edu>
Date:           See top of file.

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

    $Author: hc3 $
    $Date: 2006/04/25 04:17:59 $
    $RCSfile: input,v $
    $Revision: 1.11 $
```

Output Help Tue Apr 25 00:15:45 EDT 2006

Output is written to the standard output. This is ‘stdout’ in C, ‘cout’ in C++, ‘System.out’ in JAVA, and ‘*standard-output*’ in COMMONLISP. These output streams do NOT have to be opened; you should NOT open any files.

In general, to be correct your program must produce EXACTLY the one and only correct sequence of output characters.

The only whitespace characters you should use are new-line and the single space ’ ’, unless the problem de-scription contains explicit instructions to use hori-zontal tabs.

You should never use two single spaces in a row unless explicitly instructed to do so. Usually the only time you use two spaces in a row is when you are asked to line things up in columns.

You should never put space characters at the beginning or ending of lines unless explicitly instructed to do so (which is never, except for beginning spaces to line something up in a column).

You must use the correct upper or lower case.

The judging is done by comparing your program’s output to a judge’s solution’s output on specially prepared judge’s input. The comparison checks for differences in column position of non-numeric non-blank characters and the ends of numbers, and checks the differences of numbers. The comparison is likely to require an EXACT match for output that does not include floating point numbers, and an EXACT match except for numbers when floating point numbers are included. BEWARE.

To format floating point numbers within a fixed number of columns, you should use formats like %10.6f in C and ~10,6F in COMMONLISP. See the help file on C++ for information on how to output floating point numbers in C++, and the help file on JAVA for information about how to output floating point numbers in JAVA.

If you are asked to output floating point numbers but are NOT given the number of columns in which to place the number, you may assume that the number of columns occupied by the number is unimportant. This is because numbers that are only slightly different may take dif-ferent numbers of columns: e.g., 10.000 and 9.999 where the accuracy is supposed to be 0.001 and the true value might be close to 9.9995.

Numbers that are output should NEVER have commas in them, unless the problem specifically states otherwise.

In order to get sufficient precision, floating point computations should be done in double precision, and constants like PI should have double precision accuracy.

File: output
Author: Bob Walton <walton@deas.harvard.edu>
Date: See top of file.

RCS Info (may not be true date or author):

$Author: hc3 $
$Date: 2006/04/25 04:17:59 $
$RCSfile: output,v $
$Revision: 1.9 $

Print Help Tue Apr 25 01:08:22 EDT 2006

In a fully formal contest the contest staff will bring your printouts to you, and you are NOT permitted to fetch them from the printer yourself. The 'printer' command below may tell you something about the status of your printout.

In a formal contest run for practice purposes, you may be asked to get your own printouts. The 'printer' command will tell you which printer printed them.

In other contests in which you are given an account into which you log from your own personal computer, you have to define your printer yourself in the ~/PRINTER file of your account: see below for details.

Printer Commands:
---

print filename ...

Prints files with 80 column width.

print2 filename ...

Prints files with 80 column width using a more compact two pages per page format and very small font.

fprint filename ...

Ditto but assumes 56 column width and uses a normal sized font. All our documentation and sample code is written with a 56 column width (and where appropriate, a 40 line per page length).

printer

Shows the printer status.

Printer Definition Using The ~/PRINTER File

If you are given a contest account into which you log from your own personal computer, you have to define your printer by putting its name in the ~/PRINTER file in your contest account. The name of the printer can be an email address, in which case postscript files will be emailed to that address, instead of being sent to a normal printer.

If ~/PRINTER contains an email address, postscript files will be the bodies of the messages mailed. But if ~/PRINTER contains an email address preceded by an initial '@' (so ~/PRINTER has a line with two '@'s) then the postscript files will be attachments in the messages mailed.

If postscript files are being emailed in message bodies to an address like 'fee<fi@fo>' on a system that uses 'procmail' (e.g., Linux), one typically puts the following entry in the '.procmailrc' file of the 'fi@fo' account:

```
:0 b
* ^To:[         ]+fee<fi
| lpr -Plp
```

This pipes the body of any email received with a 'To' address of 'fee<fi...' to 'lpr -Plp'. Here the []'s contain a space followed by a tab.

If you are working in the same building as a printer known to the computer on which your contest account runs, and you want to use that printer, you can just put its name in ~/PRINTER. This name must not have any '@'s.

File: print
Author: Bob Walton <walton@deas.harvard.edu>
Date: See top of file.

RCS Info (may not be true date or author):

```
$Author: hc3 $
$Date: 2006/04/25 05:18:35 $
$RCSfile: print,v $
$Revision: 1.7 $
```

Commonlisp Help Thu Sep 28 19:21:57 EDT 2006

Commonlisp

There is no reasonably standard COMMONLISP, so HPCM hides the exact commonlisp it is using inside the hpcm_clisp program. The standard way of running a COMMONLISP solution interpretively is

```
hpcm_clisp -I pppp.lsp
```

This loads pppp.lsp into the interpreter and issues the '(main)' call. The programmer should write a '(defun main ...)' for a function that reads '*standard-input*' and writes '*standard-output*'. The output of this command is just what the 'main' function writes to the standard output, and nothing more.

The pppp.lsp file can be compiled via the command

```
hpcm_clisp -c pppp.lsp
```

and the compiled code can then be executed by

```
hpcm_clisp -I pppp.fas
```

Program Structure

The following can be used as the structure of a pppp.lsp file:

```
(defvar debug)
(defun dformat (&rest r)
    (if debug (apply #'format t r)))

(defun main (&rest r)
  (setq debug r)

  (loop
    (let ( (line (read-line t nil 'eof)) )
      (if (eq line 'eof) (return))

      (dformat "..." . . .)

      . . . ))

   (format t "..." . . .))
```

If the program is called with one or more arguments, the 'debug' defvar is set non-nil, and '(dformat ...)' will do nothing. The program can be called with arguments by executing

```
hpcm_clisp -I pppp.fas argument ...
```

The '(read-line t nil 'eof)' expression will read a line and return it, but return the symbol eof if an end of file is encountered instead of a line.

```
File:           commonlisp
Author:         Bob Walton <walton@deas.harvard.edu>
Date:           See top of file.

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2006/09/28 23:24:23 $
    $RCSfile: commonlisp,v $
    $Revision: 1.5 $
```

Scores Help Sat Dec 23 01:59:54 EST 2006

Here we describe the scores of individual problems. See the ‘scoreboard’ help file for a description of how these individual scores are put together into a total score for each team or individual contestant.

Results Come by Email:

You will get your results back by e-mail. Correct results will also be posted on the scoreboard (see below).

Whatever you submit using ‘make submit’ will be Cc’ed to your account. If you are using a personal account not provided by the contest, you may not be using ‘make submit’, but may be simply sending email yourself to the autojudge, in which case it is your responsibility to Cc the mail to yourself if you wish.

Asking Questions:

You may e-mail questions to the judges but the method to do this from a contest provided account is a little weird. First, edit your email into a file of your own choosing. If your first line has the form

```
Subject: ....
```

and the next line is completely empty (no spaces), then the first line will be the subject line of the message. Then

```
hpcm_sendmail < filename
```

to send the file as email to the judges.

Note that the subject must NOT begin with the words ‘submit’ or ‘get’.

If you are using your own personal account not provided by the contest, just send ordinary mail to the auto-judge, taking care that it’s subject does not begin with ‘submit’ or ‘get’. A human judge will be notified and eventually respond.

If you are using a personal account with the UNIX tools described in the ‘email_unix_tools’ help file, you can also use hpcm_sendmail if you choose.

If you ask for a clarification by e-mail, and a significant answer is given, that answer will be given by posting it on the scoreboard so everyone can see it. You will not receive email back in this case.

What Scores are Possible

The possible scores for a particular submission are:

- Completely Correct
- Formatting Error
- Incomplete Output
- Cpu Time Limit Exceeded
- Output Size Limit Exceeded
- Program Crashed
- Incorrect Output

The first score, ‘Completely Correct’, means the submission is correct. ALL the other scores mean the submission is incorrect. Sometimes the word ‘accepted’ is used instead of ‘correct’ and the word ‘rejected’ is used instead of ‘incorrect’. With this terminology, the score ‘Completely Correct’ means the submission was accepted, and ALL the other scores listed above mean the submission was rejected.

What Do the Scores Mean

Scoring is done by running the judge’s input through your program to produce ‘your output’, which is compared with the ‘judge’s output’, that is produced by running the judge’s solution on the same input. With this in mind, the precise meanings of the possible submission scores are as follows:

Completely Correct

Your output was essentially the same as the judge’s output. For some problems, ‘essentially the same’ means exactly the same, character by character. For problems with floating point numeric output, numbers may not differ by more than a tolerance specified in the problem description.

If the number of columns occupied by numbers and other output is specified by the problem description, then numbers are required to end in the correct column, even when they differ. If the number of decimal places in output numbers is specified, your numbers and the judge’s numbers must have the same number of decimal places. If the presence or absence of exponents in output numbers is specified, your numbers must agree with the judge’s numbers on having or not having an exponent. The absence of an exponent is implicit if an absolute number tolerance is specified, such as 0.001; but specification of a relative number tolerance, such as 0.01%, implies that exponents are optional.

If white spaces in the output are specified by the problem description, no extra spaces are allowed and no required spaces may be missing. Most problem descriptions specify the number of lines to be output, in which case no extra blank lines may be output and no required blank or empty lines may be missing.

‘Completely Correct’ is the ONLY score that means your solution was accepted by the judges, and you have solved the problem. All other scores mean your solution has been rejected by the judges.

Formatting Error

Your output has the right words and the right numbers (within any designated tolerance) in the right order, but there are unallowed differences in whitespace, columnization, letter case, number of decimal places in numbers, or presence or absence of exponents in numbers. These differences can be as little as one extra blank line in your output!

Although omitting a blank line is usually a Formatting Error, if the line is used to separate test cases, omitting it may be an ‘Incorrect Output’ error.

For problems in which only the numbers really matter, omitted, extra, or misspelled words may be considered to be just formatting errors. However, even in this case words like ‘Case’ and ‘Data Set’ that announce the beginning of a new test case in the output are considered to really matter, and if incorrect, will cause an ‘Incorrect Output’ score and not a ‘Formatting Error’ score.

Incomplete Output

Your output is not empty, and is either completely correct or has only formatting errors, but your output stops prematurely.

Cpu Time Limit Exceeded

The program was terminated by the system because it exceeded the CPU time limit imposed by the problem Makefile.

If you open files instead of using the standard input and output you may get a Cpu Time Limit Exceeded score: see Common Mistakes in the solving help file.

Output Size Limit Exceeded

The program was terminated by the system because it exceeded the amount of output allowed by the problem Makefile.

Program Crashed

Your program did not produce any output, or your program sent output to the standard error stream, or your program terminated abnormally; but your program did not exceed either CPU time or output size limits. Exceeding memory limits will cause this score, because it is unfortunately difficult to distinguish memory limit exceedments from other kinds of program crashes (e.g., using unallocated memory).

If you open files instead of using the standard input and output you may get a Program Crashed score: see Common Mistakes in the solving help file.

Incorrect Output

The program did not crash or terminate prematurely, and some word or number in your output does not match the corresponding word or number in the judge’s output, even when letter case and whitespace are ignored and number differences within any tolerance stated in the problem description are ignored.

Omitting a blank line that is used to separate test cases may be an ‘Incorrect Output’ error instead of a ‘Formatting Error’. In general, failure to output correctly lines that begin or separate test cases may be an ‘Incorrect Output’ error, instead of a ‘Formatting Error’.

The autojudge is generally very good at giving the right score, with the exception of certain cases where the autojudge scores ‘Incorrect Output’ but a human judge would score ‘Formatting Error’. Misspelled, omitted, out-of-order, or extra words may be such cases.

Although a human judge who is manually reviewing scores could change the score in such a case, she does not for most contests, because the extra effort to do so is not worth the minimal benefit to the contestants, who should find it fairly easy to catch such errors, and should not need the extra hint involved in changing the score.

File: scores
Author: Bob Walton <walton@deas.harvard.edu>
Date: See top of file.

RCS Info (may not be true date or author):

```
$Author: walton $
$Date: 2006/12/23 07:03:53 $
$RCSfile: scores,v $
$Revision: 1.16 $
```

Scoreboard Help Thu Sep 28 19:29:21 EDT 2006

The scores of individual problems are described in the ‘scores’ help file. This file describes how individual problem scores are put together into a scoreboard.

The Scoreboard

If you do not have a contest account provided by the contest judges, you will typically have been given a web URL at which a one page contest scoreboard is displayed.

But in a contest account provided by the contest judges the scoreboard is best displayed by allocating a separate window to show the scoreboard and in that window running the command:

```
scoreboard
```

This command will show the contest scoreboard refreshing every 30 seconds. The command is best executed in its own window. To terminate type control-C.

The scoreboard may have several pages, and in such a case this command cycles through all the pages, going to the next page every 30 seconds.

```
scoreboard N
```

This command just displays the N’th scoreboard page, refreshing every 30 seconds.

```
scoreboard CONTEST-NAME
scoreboard CONTEST-NAME N
```

Sometimes one account can be used to enter many different contests. In this case you have to identify which contest you want to see the scoreboard of. Usually the ‘scoreboard’ command by itself will list all the possible contests.

The part of a scoreboard that shows team scores may be frozen during the last hour of a formal contest. This means the scoreboard only shows results as of the time when it was frozen, which is one hour before the contest ends. This is done to create suspense, so people will come to the post-contest meeting at which the final contest results will be announced.

Ranking Submitters

The scoreboard gives for each problem and submitter information about the submissions made by the submitter for the problem. For each submitter the scoreboard typically gives the total number of problems correctly solved by the submitter, and a ranking score that ranks submitters who have the same number of solved problems.

A submitter (contest account) may be a team or an individual, depending on the type of contest. Here we use ‘submitter’ to mean either ‘a team’ or ‘an individual’.

Some contests are completely unscored. For these the submitters are listed in alphabetical order, and for each problem and submitter an indication is given of whether the submitter has solved the problem, and if so, the date the problem was solved.

On the other hand, if the contest is being scored, submitters with more correctly solved problems appear earlier in the scoreboard.

When a contest is being scored, there may be a way of ranking submitters who have the same number of solved problems. If there is no ranking for a scored contest, submitters are listed by alphabetical order within the group of submitters all of which have the same number of solved problems.

There are two different methods of ranking that can be used. The classical way, used in all limited duration contests, is to measure the time required to solved each problem. This is called 'timed' ranking. The other way, usable in untimed contests, is to use the kinds of feedback provided for incorrect submissions. Depending upon the submission type, either the judge's input or both the judge's input and output for the first failed test case can be provided for a submission. This is called 'feedback' ranking.

For either kind of ranking, each solution is assigned a numerical score, these scores are either added or averaged to produce a score for each submitter, and problems a submitter never solves do not count in any way toward scoring or ranking submitters.

Below we discuss in separate sections timed, feedback, and unranked contests, so you may read just the appropriate section. However the following is common to all kinds of contest.

A scoreboard has an entry for each submitter and each problem.

If there are no submissions for a problem by a submitter, '.....' appears in the scoreboard for the submitter and problem. This also appears if there are no correct submissions and incorrect submissions are NOT being reported. But if incorrect submissions are being reported and there are some for a problem not yet correctly submitted by a submitter, then '..../N' appears, where N is the number of incorrect submissions by the submitter for the problem.

For each submitter in a ranked contest the number of solutions N and ranking score RRR of submitter are reported in the format 'N/RRR'. For timed contests higher RRR is worse, but for feedback contests higher RRR is better. For scored but unranked contests, only N is reported, and there is no '/RRR'.

In some contests, problems are scored first by an autojudge, and then if they are not completely correct, the score is reviewed by a human judge. If you see a '*' prefixed or suffixed to any item of scoreboard information, this means that one or more submissions used to compute the item have been scored incorrect by the autojudge, but are still awaiting review by the human judge.

Formal Timed Contests
------ ----- --------

A formal contest is a contest of limited duration in which the problem descriptions are passed out at the start of the contest. Each solved problem is given a score that is the difference between the time the correct solution was submitted and the start time of the contest.

In a formal contest a submitter's ranking score is the sum (NOT the average) of all the solution scores for the solutions submitted by the submitter. Lower time scores are better, higher time scores are worse. Also, if you think of problems as being solved consecutively, and a submitter solves, say, three problems, the first in time T1, the second in time T2, and the third in time T3, then the submitter's ranking score is 3*T1 + 2*T2 + T3, so it pays to solve the easiest problems first.

Solution times are computed in seconds. Some contests apply a penalty of typically 20 minutes (1200 seconds) for each incorrect submission of a problem made before the first correct submission of the problem.

Solution times may be represented on the scoreboard as 'M:SSsN', with decimal numbers M, SS, and N, meaning that the solution time is M minutes and SS seconds, and there were N submissions, including N-1 incorrect submissions, before the first correct submission. The 's' means the time SS is in seconds. N may be omitted if incorrect submissions are not being reported on the scoreboard.

'H:MMmN' instead means the time is H hours and MM minutes, rounded down to the nearest minute. 'D:HHhN' instead means the time is D days and HH hours, rounded down to the nearest hour. 'DdN' means the time is D days, rounded down to the nearest day. The choice of time unit, s, m, h, or d, is made to give the elapsed time to the greatest accuracy within 6 display characters, including the unit character. However the actual solution time used for computing a submitter's ranking score as described above is always in seconds.

Informal Timed Contests

Informal contests derive timed scores without having a single contest start time or giving out paper problem descriptions. In an informal contest a submitter gets a problem named pppp by executing

```
hpcm_get problems/pppp
```

This obtains a description of the problem as a file in the submitter's problems/pppp directory.

In general, the problem score for a timed contest for a submitter is the difference between the time the submitter submits a correct solution and the start time of the problem. There are two ways of scoring informal contests, depending on how problem start times are assigned.

If the problem start times for a particular submitter are ALL the first time the submitter gets ANY problem in the contest, this time acts as the start time of the contest for the particular submitter, and the ranking score is the same as that of the formal contest above. In this case, an informal contest is just like a formal contest except each submitter (each team) can start at a different time.

If a start time for a particular problem and submitter is the time the submitter hpcm_get's the problem, the ranking score is the AVERAGE of the problem scores, which is just the average time to solve any problem the submitter has solved. Again lower scores are better, higher scores are worse. In contests of this kind there is often a maximum problem score, i.e. a maximum problem solution time, so that any problem whose solution takes longer than this maximum is counted as if it took just this maximum amount of time to solve.

Feedback Contests
-------- --------

When a scored contest is not of limited duration and problems are not being timed, three different kinds of submission may be allowed, and different penalties are assigned for each kind of submission when that submission is incorrect.

The methods for making different kinds of submissions are as follows:

Contest Provided Account Command:	Personal Account; Send email with:
make submit	Subject: submit ppp.ext
make in-submit	Subject: submit ppp.ext in
make inout-submit	Subject: submit ppp.ext inout
make solution-submit	Subject: submit ppp.ext solution

You use the ‘make’ commands in an account provided to you by the contest managers, and you send email with the given ‘Subject:’ fields from your own personal account that is not provided by the contest. You can also use the ‘make’ commands in a personal account if you use the email UNIX tools: see the email_unix_tools help file.

In what follows we refer to the different types of submission as ‘submit’, ‘in-submit’, ‘inout-submit’, and ‘solution-submit’.

The penalties typically assigned for different submission types in feedback scoring are:

submit	10%
in-submit	20%
inout-submit	30%

‘Submit’ returns just the submission score. ‘In-submit’ also returns the judge’s input for the first test case that was incorrect, when this is practical. ‘Inout-submit’ returns both judge’s input and judge’s output for this test case.

The score for each solution starts at 100.00, and is multiplied by 90% for each incorrect ‘submit’, 80% for each incorrect ‘in-submit’, and 70% for each incorrect ‘inout-submit’. Thus each penalty is applied to the remaining score for a problem. Note that only incorrect submissions before the first correct submission count.

A submitter’s ranking score is the average (NOT the sum) of the solution scores for all problems solved by the submitter.

‘Solution-submit’ does not affect scoring. It returns the judge’s solution if the score is ‘Completely Correct’. After receiving a ‘Completely Correct’ score for some other type of submission, you may resubmit using ‘solution-submit’ to get the judge’s solution without affecting your problem score.

If incorrect submissions are being reported, then for each solution the solution score and the number of incorrect submissions are both reported using the format ‘IIiOOoXX:SS’. Here the upper case letter pairs denote numbers and the lower case letters ‘i’ and ‘o’ appear literally on the scoreboard. II is the number of in-submit’ incorrect submissions, OO the number of ‘inout-submit’ incorrect submissions, XX the number of ‘submit’ incorrect submissions, and SS is the solution score rounded to the nearest integer. If there are no incorrect submissions, just ‘100’ is reported instead.

If incorrect submissions are NOT are being reported, then only SS is reported.

Although SS is rounded to the nearest integer, in computing the ranking score of a submitter, the solution score to many decimal places is used.

Unranked Contests

If a contest is unranked, there are no ranking scores, and the entry for a problem solution is just the date of the solution, followed by '/N' if N is the total number of submissions up to and including the first correct submission, provided that incorrect submissions are being reported.

The scoreboard for an unranked contest may or may not report the number of correct submissions each submitter has. If it does, the submitters are sorted first by their numbers of correct submissions, and then alphabetically by submitter name, and the scoring is called 'unranked scoring'. Otherwise submitters are all just sorted alphabetically, without reporting the number of correct submissions for each submitter, and the contest is said to be 'unscored'.

File: scoreboard
Author: Bob Walton <walton@deas.harvard.edu>
Date: See top of file.

RCS Info (may not be true date or author):

```
$Author: walton $
$Date: 2006/09/28 23:31:48 $
$RCSfile: scoreboard,v $
$Revision: 1.19 $
```

C++ Help Tue Sep 9 08:34:05 EDT 2008

Include Files

The following are typical includes in modern C++:

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <cstring>
#include <cctype>
#include <cassert>

using namespace std;
```

It is common to have a conflict with some name defined in the includes. E.g., you may have trouble naming a global variable ‘time’. To fix this, change the name of your variable. A simple way of doing this is to put the following right after the includes:

```
#define time Time
```

If you want your code to last as include files change, replace the using statement above with statements such as the following:

```
using std::cin;
using std::cout;
using std::endl;
```

Program Structure and Debugging

Typical program structure including end of file detection and debugging is:

```
#include ...
. . .

bool debug;
#define dout if ( debug ) cout

int main ( int argc )
{
    // argc-1 == number of program arguments.
    debug = ( argc > 1 );
    . . . .
    while ( true )
    {
        . . . . .
        cin >> x >> y >> z;
        if ( cin.eof() ) break;
        . . . . .
        dout << . . .
        . . . . .
    }
}
```

After reading with ‘cin >> ...’, cin.eof() is true if and only if an end of file has occurred on cin. No other error indication is given. Note that ‘cin.eof()’ is normally false after successfully reading some value, as the input is still before the end of the line containing the value.

Many problems require you to detect certain input values as end of data markers, instead of looking for eof. E.g., the problem might specify that x, y, and z are all 0 to indicate end of data, in which case you replace the above 'if' statement by

```
if ( x == 0 && y == 0 && z == 0 ) break;
```

Above 'dout' behaves just like 'cout' except no output is produced unless your program is called with an argument. One must be careful, however, to remember that 'dout' contains an 'if' statement and cannot be followed directly by an 'else if'; i.e., the following will NOT work:

```
if (...) dout << ...;
else if ...
```

Put {} brackets around 'dout << ...;' in this situation.

Debugging is best done with information printed by 'dout', and not with a debugger like 'gdb'. The exception is debugging programs that crash or go into an infinite loop, which is best done with a debugger.

It is also a good idea to use 'assert' statements to check that assumptions you have made are valid during actual program execution.

Numeric Constants

The following code imports useful constants, which are the minimum and maximum values of various number types, and the values of PI and E.

INCLUDE	IMPORTS
#include <climits>	int INT_MAX; int INT_MIN; long LONG_MAX; long LONG_MIN; unsigned long ULONG_MIN;
#include <cfloat>	double DBL_MAX; double DBL_MIN; float FLT_MAX; float FLT_MIN;
#include <cmath>	double M_PI; double M_E;

Fixed Width Output

The following is useful for producing fixed width format numbers.

To output a right adjusted integer I in N columns use:

```
cout << setw (N) << I;
```

Setw sets the width of the next output; but each output resets this to 0 so you must reset the width just before outputting each N column number. The width of 0 means 'use as many columns as necessary'.

To output a right adjusted floating point number F in N columns with P decimal places use:

```
    cout << setiosflags ( ios::showpoint | ios::fixed );
    cout << setprecision (P);
    cout << setw (N) << F;
```

The precision P and flags do not get reset after the number is output. They can be reset to defaults by

```
  cout << resetiosflags ( ios::showpoint | ios::fixed );
  cout << setprecision (6);
```

Sometimes you are asked to use '0' as a high order fill character, instead of ' ', or to output in hexadecimal or octal instead of decimal. This can be done with

```
   cout << setfill ('0');     // Set fill character.
   cout << setfill (' ');     // Reset fill character.
   cout << hex;               // Set base to 16.
   cout << dec;               // Set base to 10.
   cout << oct;               // Set base to 8.
```

To use setw etc. you need to:

```
    #include <iomanip.h>
```

You may also need

```
    using std::setw;
    using std::hex;
    using std::dec;
    using std::setprecision;
    using std::ios;
```

etc. However, somewhat counter intuitively

```
    using std::setiosflags;
    using std::resetiosflags;
```

are not needed.

To left adjust use

```
    cout << setw ( my_width )
         << setiosflags ( ios::left )
         << my_string
         << resetiosflags ( ios::left )
         ...
```

Character Input

You can use the following for character input:

```
    int c = cin.get();  // Get next character.
    int c = cin.peek(); // Return next character
                        // without skipping over it.
```

These return EOF for end of file.

Inputting Lines

To input a line use the getline function, as in

```
        char buffer [SOME_SIZE];
        cin.getline ( buffer, sizeof ( buffer ) );
```

The new line at the end of the line is NOT stored in the buffer; a NUL character is stored at the end of the line in the buffer. Do NOT try to use the 'get' function in place of 'getline': its behavior is similar but it does NOT skip over the new line in the input stream, and therefore after reading the first line it reads empty lines forever.

If you use non-line-oriented code to input a value V, you CANNOT then use 'getline' to get a following line, unless you first skip by the end of line after V. Failing to do this causes 'getline' to read an empty line, consisting of just the line feed after V. Typical correct code is:

```
int x;
cin >> x;       // Does not skip line end.
while ( cin.get() != '\n' );
cin.getline ( ... );
```

Parsing Lines

You can parse a line after it is input by code such as

```
#include <sstream>
using std::istringstream;

istringstream in ( buffer );
```

Now 'in' is an input stream whose input is taken from the buffer (actually, the input is copied from the buffer into another buffer internal to the instring-stream when the latter is created).

Here in.eof() will be true at the end of the string. This is likely to be a problem because immediately after reading a number at the end of the buffer in.eof() will be true. To compensate, check for eof BEFORE reading each value by first executing 'in >> ws' to skip past whitespace, then testing in.eof(), and only then reading the value. Typical code is:

```
in >> ws;
if ( in.eof() ) break;
in >> x;
```

Writing Strings

Strings can be written using:

```
#include <sstream>
using std::ostringstream;

ostringstream out;
. . . out << . . .
out << ends;
const char * s = out.str().data();
cout << s;
```

Here 'out' is an output stream whose output is written into an INTERNAL buffer (you CANNOT make a stream that writes into your buffer). You can get the starting address of a string that contains the contents of the internal buffer with 'out.str().data()'. Before doing this, use 'out << ends' to write a NUL into the buffer.

You can also use 'cout << out.str()' to write the internal buffer to 'cout' directly, but do NOT use 'out << ends' in this case, as the NUL will be written to 'cout' if you do.

Writing Your Own << And >> Operators

It is often convenient to define your own << operator to output something. Some examples are:

```
using std::ostream;
ostream & operator <<
        ( ostream & out, mytype & value )
{
    . . .
    out << . . .
    . . .
    return out;
}
```

Suppose you want to output integers in a distinctive format. A simple way is

```
struct myformat {
    int value;
    myformat ( int value ) : value ( value ) {}
};
ostream & operator <<
        ( ostream & out, myformat s )
{
    out << ( ... s.value ... );
    . . .
    return out;
}
. . .
cout << myformat ( 99 );
```

Here we have invented a typed structure to encapsulate the value when it is to be printed. Note the argument to << may NOT be ‘myformat & s’ because here s is not constant, and in use the ‘myformat’ value is a temporary and temporaries are read-only.

Input operators can be define by:

```
using std::istream;
istream & operator >>
        ( istream & in, mytype & value )
{
    . . .
    in >> . . .
    . . .
    return in;
}
```

STL API Documentation

Standard Template Library (STL) API documentation is usually available on-line during a formal contest. The command to access it is:

```
stlhelp
```

In a formal contest, you should NOT use other means to access such documentation, as using the internet is a violation of formal contest rules.

```
File:           c++
Author:         Bob Walton <walton@deas.harvard.edu>
Date:           See top of file.

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2008/09/09 12:31:49 $
    $RCSfile: c++,v $
    $Revision: 1.14 $
```

```
JAVA Help                    Thu Sep 28 19:35:45 EDT 2006

API Documentation
--- -------------

JAVA API documentation is usually available on-line
during a formal contest.  The command to access it is:

        javahelp

In a formal contest, you should NOT use other means to
access such documentation, as using the internet is a
violation of formal contest rules.


Program Structure
------- ---------

The following is a suitable structure for a program that
solves the problem named PPPP:
```

```
import java.io.*;
import java.util.StringTokenizer;

public class PPPP {

    public static boolean debug;

    public static void dprintln ( String s )
    {
        if ( debug ) System.out.println ( s );
    }

    public static void main (String[] args)
            throws IOException
    {

        debug = ( args.length > 0 );

        BufferedReader reader
           = new BufferedReader
                 ( new InputStreamReader
                       ( System.in ) );

        while ( true )
        {

            String line = reader.readLine();
            if ( line == null ) break;

            StringTokenizer tokenizer
                = new StringTokenizer ( line );
            . . .

            dprintln ( . . . );

            System.out.println ( . . . );
        }
    }
}
```

Some parts of this structure are described below, and examples are given in demos/count/count1.java and demos/javaio/javaio.java.

Debugging

In the above program structure, debugging is turned on if the program is called with any arguments (the main function args is of non-zero length). In this case the dprintln function prints, and in the opposite case, where the program is called with no arguments, dprintln does nothing.

Input

Line oriented input is best done with

```
java.io.BufferedReader.readLine
java.util.StringTokenizer.getToken
```

An example is in the count1.java demo. When using this method, strings returned by getToken may have to be converted to numbers. This can be done by code such as

```
String s = ...;
int i = Integer.parseInt ( s );
long l = Long.parseLong ( s );
double d = Double.parseDouble ( s );
```

If line ends are to be treated just like space characters, using

```
java.io.StreamTokenizer
```

is a good idea. An example is in the javaio.java demo.

Output

Output of floating point numbers with a specified number of decimal places is done with

```
java.text.DecimalFormat
```

See the javaio.java demo. It is important in some contests to use a DecimalFormat object created with the ENGLISH Locale.

Numbers that are output should NEVER have commas in them, unless the problem specifically states otherwise. This can be a problem if one uses NumberFormat.get_ instance and then fails to use applyPattern (see the javaio.java demo).

String and free format number output examples merely use

```
System.out.println
```

and are given in the count1.java an javaio.java demos.

Crashes

It is fairly common for JAVA programs which run well for the contestant to crash when run by the judge. The usual cause is judge's input data that triggers an exception not observed with contestant input data. Examples are when an out-of-range array index is used, or the program one runs off the end of an input line and calls getToken with no more tokens. The best debugging strategy is to try to find input data that breaks the program.

```
File:           java
Author:         Bob Walton <walton@deas.harvard.edu>
Date:           See top of file.

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2006/09/28 23:37:30 $
    $RCSfile: java,v $
    $Revision: 1.7 $
```

Help from Polonius Thu Sep 28 19:37:40 EDT 2006

Read the Demonstration Problems

Every contest has rules for how to write your solution programs; e.g., rules for submitting files, rules for doing input and output, etc. The best way to BE SURE you are following these rules is to read a demonstration problem solution in the language you are using, and mimic that solution when you do input/output, submit your files, etc.

Input/Output Difficulties

A surprising number of solutions have simple bugs in input or output. You need to master these with a bit of practice. For example, be sure you can detect the end of file and not go into an infinite loop when it happens.

Completeness

Often solutions are correct except that their author ignored one smallish instruction in the problem statement. You need to test that your solution meets EVERY stipulation in the problem statement. There is no partial credit in a contest like this.

Be sure you can handle the formatting issues of a problem before you attempt it.

In a Formal Contest Do the Easiest Problems First

Take great care to do easiest problems first. Your primary score is the number of completely correct problems you have done. Your secondary score, used in the event of a primary score tie, is the sum of the number of seconds you took for each problem, where the seconds taken for a problem is computed by subtracting the start time of the contest from the submit time of your first correct solution of the problem. The secondary score is lowest best (the primary score is highest best). In effect, if you do N problems, the time you take on the first problem counts N times, the time on the second N-1 times, etc. So you want the problems you solve first to be the ones that take you the least amount of time. You also want to do easiest problems first so you will solve more problems, and your primary score will be higher.

Formal Contest Goals

You should try hard to complete half the problems. Often completing a bit more than half the problems is sufficient to be one of the several contest ‘winners’ (i.e. to go on to the next round).

Contestants who are completely new to this kind of all-or-nothing problem scoring usually find it difficult to get even one problem correct during a timed contest. Such contestants should have as their goal to get just one problem during the contest.

```
File:           advice
Author:         Bob Walton <walton@deas.harvard.edu>
Date:           See top of file.

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2006/09/28 23:38:36 $
    $RCSfile: advice,v $
    $Revision: 1.9 $
```

Dynamic Programming Problem Help Sat Sep 30 06:59:20 EDT 2006

A dynamic programming algorithm is an algorithm with two characteristics:

A. A problem P is parameterized in such a way that some parametric set of subproblems can be solved recursively. For example, the original problem P might be parameterized as P(i,m) for some integers 0 <= i < N, 0 <= m < N. And each problem P(i,m) for 0 < m might be solvable fairly quickly (say in time proportional to N) from the solutions to all the problems P(i,m-1) for 0 <= i < N, while the problems P(i,0) are readily solvable.

An example is the problem of finding the length of the shortest path from node 0 to node N-1 in an undirected graph with nodes 0, 1, ..., N-1 and edges each of which have some length > 0. Then let P(i,m) be the problem of finding the length of the shortest path from node 0 to node i that has no more than m edges. Then we want to solve P(N-1,N-1), because any path with more than N-1 edges would have a cycle that could be removed to make the path shorter.

Treat two nodes that are NOT connected as if they were connected by an edge whose length is infinity (or some number much larger than any possible shortest path). Then P(0,0) == 0 and P(i,0) = infinity for 0 < i < N. And for m > 0, P(i,m) is the minimum of P(k,m-1)+edge_length(k,i) for all nodes 0 <= k < N.

Note that if the length of every present edge is 1, this problem is just the breadth first search problem with N-1 as the only goal node (see the help file on breadth_first_search). It can be modified to have a set of goal nodes, and is easier to program than breadth first search. But dynamic programming does not generally work as well as breadth first search if the nodes cannot be organized into a simple array.

B. The solution to one of the parameterized problems (e.g. P(i,m)) is typically used very many times in computing the solution to the final problem recursively, so it is important to remember this solution as a table entry and not recompute it every time it is needed.

Indeed, in our example P(k,m-1) is used to compute P(i,m) for every i, therefore it is used N times.

The word ‘programming’ in ‘dynamic programming’ refers to describing the table of subproblem solutions and the order in which the table will be computed.

Dynamic programming can also be used to compute an actual shortest path, and not just the length of such a path. There are two ways of doing this:

C. For each P(i,m) record in P_previous(i,m) the last node k that is just before i on some shortest path from node 0 to node i. This can be computed when computing P(i,m):

```
when setting P(i,m) = P(k,m-1)
                    + edge_length(k,i)
     also set P_previous(i,m) = k
```

D. After computing P(i,m) for all i and m, backtrack to build an end segment of the path as follows:

```
initially the end segment is just the node N-1
          and m = N-1
while 0 is not the first node in the end
      segment:
    let i be the first node in the end segment
    find a k such that
        P(i,m) = P(k,m-1) + edge_length(k,i)
        and add k to the beginning of the end
        segment
    set m = m-1
```

The first step in solving any dynamic programming problem is to parameterize the problem. But notice that in our example, the parameter m is not even hinted at by the original problem statement, which is just to find the distance between 0 and N-1. We say that m is a 'hidden parameter'. The essence of solving a dynamic programming problem is to find the hidden parameter (or parameters) that are needed to make a fast recursive algorithm.

The Traveling Salesman Problem

Some 'intractable' problems have good dynamic programming solutions WHEN THE PROBLEM PARAMETERS ARE SMALL ENOUGH. An example is the traveling salesman problem: find the shortest path from node 0 to node N-1 that visits all the nodes just once. The subproblem for this is P(i,S): find the shortest path from node 0 to node i that visits every node in the set S of nodes just once. Then P(i,{0,1,...,N-1}) is the answer.

However to compute the answer one needs to solve the subproblem for 2**N values of S, and there are on the order of N*(2**N) subproblems overall. For N <= 15 this is doable.

File: dynamic_programming
Author: Bob Walton <walton@deas.harvard.edu>
Date: See top of file.

RCS Info (may not be true date or author):

```
$Author: walton $
$Date: 2006/09/30 10:58:15 $
$RCSfile: dynamic_programming,v $
$Revision: 1.11 $
```

2D Geometry Problem Help Sat Oct 14 13:51:23 EDT 2006

For some reason many young computer scientists think that geometry problems are hard, and avoid them. The purpose of this discussion is to demonstrate that geometry problems are fairly easy after all.

We will restrict ourselves here to 2D geometry.

Basics

Let

$$p = (px,py) \text{ and } q = (qx,qy)$$

be points in the plane. The difference

$$v = (vx,vy) = q - p = (qx-px,qy-py)$$

is the vector from p to q. Its length is

$$|v| = \text{sqrt}(vx^{**}2 + vy^{**}2)$$

The scalar product of two vectors

$$u = (ux,uy) \text{ and } v = (vx,vy)$$

is

$$u.v = ux*vx + uy*vy$$

$$= |u|\,|v| \cos(\text{theta})$$

where theta is the angle between u and v. The sign of theta does not matter because cos(theta) = cos(-theta). Note that

$$|v| = \text{sqrt}\,(v.v)$$

Note that u.v = 0 if and only if u and v are orthogonal (at right angles) to each other, i.e., if theta = + or - 90 degrees and cos(theta) = 0.

A 'unit' vector v is a vector of length 1 (|v| = 1). If u and v are unit vectors at right angles to each other, which is the same as saying that

$$|u| = 1, \quad |v| = 1, \quad u.v = 0$$

then one can ask what the coordinates of a point p would be in the coordinate system in which u is in the positive direction of the x-axis and v is in the positive direction of the y-axis. The answer is

$$(u.p,\ v.p)$$

Note that

$$\begin{vmatrix} u.p \\ v.p \end{vmatrix} = \begin{vmatrix} ux & uy \\ vx & vy \end{vmatrix} \begin{vmatrix} px \\ py \end{vmatrix}$$

so u and v are the rows of the 2x2 matrix which by left multiplication translates from old xy coordinates to new uv coordinates.

A counterclockwise rotation R(theta) of a vector v=(vx,vy) by an angle theta is

$$R(\text{theta}).v = \begin{vmatrix} \cos(\text{theta}) & -\sin(\text{theta}) \\ \sin(\text{theta}) & \cos(\text{theta}) \end{vmatrix} \begin{vmatrix} vx \\ vy \end{vmatrix}$$

$$= \begin{vmatrix} vx\cos(\text{theta}) - vy\sin(\text{theta}) \\ vx\sin(\text{theta}) + vy\cos(\text{theta}) \end{vmatrix}$$

Most particularly, if theta = 90 degrees, then

$$R(90\text{ degrees}) = \begin{vmatrix} 0 & -1 \\ 1 & 0 \end{vmatrix}$$

$$R(90\text{ degrees}).v = (-vy,vx)$$

Given two points p and q, it is common to want to change coordinates so these points are on the x-axis. Let v = q - p be the vector from p to q. Let w = R(90 deg).v, so w is orthogonal to v and |w| = |v|. Scalar products by v measure distance in the v direction, and scalar products by w measure distance in the w direction. w.q = w.p; to check this we see that w.p = w.q + w.(p-q) and w.(p-q) = w.v = (R(90 deg).v).v = 0. Let f = w.q = w.p. Then for any point r use the new coordinates (rx',ry') = (v.r,w.r-f). We have (px',py') = (v.p,0) and (qx',qy') = (v.q,0) because w.p-f = 0 = w.q - f. Because v and w are orthogonal and both have the same length |v| = |w| = |R(90 deg).v|, distances between points in the new coordinates are just |v| times distances in the old coordinates. For example, the distance between p and q in the new coordinate system is |v.q - v.p| = |v.v| = |v|**2 which is |v| times the distance between p and q in the old coordinate system.

Since (wx,wy) = (-vy,vx) we have

```
f = w.p = - vy * px + vx * py
  = w.q = - vy * qx + vx * qy
w.r = - vy * rx + vx * ry
ry' = - vy * rx + vx * ry - f
```

So the summary of the part of the computation that you need to remember is

```
vx = qx - px
vy = qy - py
f = - vy * px + vx * py = -vy * qx + vx * qy
for r = (rx,ry) the new coordinates are
        rx' = - vy * rx + vx * ry - f
        ry' = - vy * rx + vx * ry - f
distances between points in new coordinates are
    |v| = sqrt ( vx*vx + vy*vy )
times distances between points in old coordinates
```

Note that if all coordinates in the old coordinate system are integers, all coordinates in the new coordinate system are integers. This is a big advantage if exact arithmetic is needed, as when one must determine if a point lies exactly on a line. This is why we do NOT divide all the new coordinates by |v|, and thus must deal with new coordinate distances that are not identical to old coordinate distances.

The last coordinate change is basic to much elementary computational geometry, and you should learn it very well.

Lines Dividing Planes

The line through points p and q, oriented in the direction from p to q (p != q is assumed), divides the plane into three parts: points to the right of the line (facing in the direction from p to q), points to the left of the line, and points on the line. How do we find out whether a point r is to the right, left, or on the line?

If we change coordinates as indicate in the last section then for r

```
vx = qx - px
vy = qy - py
f = - vy * px + vx * py = - vy * qx + vx * qy
ry' = - vy * rx + vx * ry - f

ry' > 0     if r is to the left of the line;
ry' == 0    if r is on the line;
ry' < 0     if r is to the right of the line;
```

The distance from r to the infinite line through p and q is |ry'| in the new coordinates or |ry'|/|v| in the original coordinates.

The Distance of a Point to a Finite Line

We just indicated how to find the distance between a point r and an infinite line that runs through two points p and q. What if the line is finite with ends p and q?

Again shift to the new coordinates so that:

```
vx = qx - px
vy = qy - py
f = - vy * px + vx * py = - vy * qx + vx * qy
rx' =   vx * rx + vy * ry
ry' = - vy * rx + vx * ry - f
px' =   vx * px + vy * py
py' =   0
qx' =   vx * qx + vy * qy
qy' =   0
```

So p and q are now on the x'-axis and the problem is fairly easy. There are two cases:

```
if rx' < px' and rx' < qx'
or rx' > px' and rx' > qx'
   then r is off the end of the line from p to q
   and the distance is the minimum of |r-p| and
   |r-q|

if px' <= rx' <= qx'
or qx' <= rx' <= px'
   then r is over the line segment and the distance
   is |ry'|/|v|, the same as the distance from r
   to the infinite line through p and q
```

Intersection of a Finite Line and an Infinite Line

When does the infinite line through p1 and p2 intersect the interior of the finite line from q1 to q2? By interior we mean the part of the line that excludes the endpoints. Answer: when q1 is on one side of the infinite line and q2 is on the other side, if we ignore the special case where q1 and q2 are BOTH on the infinite line.

So we can compute:

```
vx = p2x - p1x
vy = p2y - p1y
f = - vy * p1x + vx * p1y = - vy * p2x + vx * p2y
q1y' = - vy * q1x + vx * q1y - f
q2y' = - vy * q2x + vx * q2y - f
```

then q1 and q2 are on opposite sides of the infinite line through p1 and p2 if and only if:

```
q1y' > 0 and q2y' < 0
or q1y' < 0 and q2y' > 0
```

or equivalently: q1y' * q2y' < 0

When does the infinite line through p1 and p2 intersect the finite line from q1 to q2, including the possibility of intersecting at an end point, i.e., at q1 or q2. Answer: if and only if q1y' * q2y' <= 0. This equation also handles the case where q1 and q2 are BOTH on the infinite line.

Convex Hulls and Polygons

The clockwise convex hull of a set of points V in a plane is a sequence of points of V, p1, p2, p3, ..., p(N), such that for every i, all the points of V are on or to the right of the infinite line from p(i) to p((i-1 mod N)+1), and such that no point p(j) is on this line except of course p(i) and p((i-1 mod N)+1). Then the finite lines from p(i) to p((i-1 mod N)+1) for i = 1, ..., N are the sides of the smallest convex polygon such that all the points of V are inside this polygon or on its boundary.

To specify a polygon we give its boundary, which for a convex polygon is either its clockwise or counterclockwise convex hull. The counterclockwise hull has the same property as the clockwise hull with 'left' replacing 'right' in the above definition.

Inside Convex Polygons

Suppose we have a clockwise convex hull p1, ..., p(N) that defines a convex polygon. So the sides of the polygon with a clockwise orientation are the lines from p(i) to p((i-1 mod N)+1), for i = 1, ..., N, and no three of the points p1, ..., p(N) lie on the same line.

Then a point r is inside the convex polygon but not on the boundary of the polygon if and only if r is to the right of the infinite line that extends each clockwise oriented side of the polygon.

A point r is inside OR ON the boundary of a polygon if and only if r is to the right of OR ON the infinite line that extends each clockwise oriented side of the polygon.

Finding the Convex Hull

To find the convex hull of a finite set of points V, first find some point p1 on the hull, which can be done say by choosing a leftmost point in V and given two leftmost points choosing the highest. Then extend the hull recursively from p(i) to p(i+1) by using the following.

Given a point p, define a relation among points q1, q2 that are not equal to p as follows:

```
    definition:
        q1 > q2 if and only if
                    q2 is to the right of the infinite
                       line from p through q1,
                 or q2 is on this line and closer to p
                       than q1 is.
```

If p is a hull point, this relation is antisymmetric and transitive (proof to reader). So given the hull up to p(i), choose p(i+1) to be the maximum point in V according to this relation. Stop when p(i+1) = p1, in which case N=i.

This is not the fastest algorithm, as it has time O(|V|*|H|) where |V| is the number of points in V and |H| is the number of points on the hull. A faster algorithm is the Graham-scan algorithm that begins with a sort of V and has running time dominated by the sort time, O(|V| log|V|). Usually the extra speed is unnecessary, but just in case, the Graham-scan algorithm is:

```
    S = V
    compute p1 as above and remove it from S
    let j = 1
    while S is not empty:
        let p(j+1) be any leftmost point in S and remove
            it from S
        while j >= 2 and p(j+1) is to the left of the
              directed infinite line from p(j-1) to pj:
            set pj = p(j+1)
            set j = j - 1
        if j < 2 or if p(j+1) is NOT on the directed in-
           finite line from p(j-1) to pj, set j = j + 1
        else if p(j-1) is closer to pj than to p(j+1),
             set pj = p(j+1)
```

When this algorithm stops only half the hull has been found. To find the other half, reset S to V minus all the points on the hull, and continue the algorithm with 'leftmost' replaced by 'rightmost'.

Intersection of Two Finite Lines

Suppose we are given four points, p1, p2, q1, q2.

Question: Does the interior of the finite line from p1 to p2 intersect the interior of the finite line from q1 to q2. By the interior of a line we mean the part of the line that excludes its end points. Also, we DO NOT COUNT as intersections parallel lines that overlap, which we will treat as a special case below.

Answer: The interiors of the lines intersect if and only if the infinite line through p1 and p2 intersects the interior of the line from q1 to q2, and the infinite line through q1 and q2 intersects the interior of the line from p1 to p2.

If we wanted to know whether the finite lines including end points intersect, we ask whether the infinite lines intersect the finite lines, endpoints included. Thus we use the test q1y' * q2y' <= 0 and similarly with the p's and q's exchanged. Again there is a special case where p1, p2, q1, and q2 are all on the same straight line and the finite lines may or may not intersect.

What if p1, p2, q1, and q2 are ALL on the same straight line? Then p1y′ = p2y′ = q1y′ = q2y′ = 0. We compute p1x′, p2x′, q1x′, q2x′. There is overlap (endpoints included) if and only if at least of the line ends is on the other finite line, i.e., if at least one of the following is true:

```
    p1x′ <= q1x′ <= p2x′        q1x′ <= p1x′ <= q2x′
    p2x′ <= q1x′ <= p1x′        q2x′ <= p1x′ <= q1x′
    p1x′ <= q2x′ <= p2x′        q1x′ <= p2x′ <= q2x′
    p2x′ <= q2x′ <= p1x′        q2x′ <= p2x′ <= q1x′
```

Lines Intersecting Polygons

So when does the interior of a line from r1 to r2 intersect the inside of a convex polygon? It does if the interior of the line from r1 to r2 intersects the interior of any side of the polygon, but r1 and r2 are not both on the infinite line extending this side. It also does if r1 and r2 are both inside or on the boundary of the polygon, but are not both on the same single side of the polygon.

These are the only two cases where the interior of the line from r1 to r2 can intersect the interior of the polygon, UNLESS some of the convex hull points p1, ..., p(N) are in the interior of the line from r1 to r2. In this last case, use a convex hull point in the interior of the line to divide the line in two, and recursively ask if the interiors of either of the two new line segments intersect the interior of the polygon.

So all one has to do is take each clockwise side of the polygon and check that r1 and r2 are not both on that side and either the interior of the finite line from r1 and r2 intersects the interior of the side, or r1 and r2 are both on or to the right of the infinite line extending the side. During this process one checks each convex hull point p to see if it is in the interior of the line from r1 to r2, and if it is, one then divides the line from r1 to r2 up into two segments, one from r1 to p and one from p to r2, and then one applies the algorithm recursively to see if the interior of either of these two segments intersects the interior of the polygon.

Polygon Maze

Problem: Given a set of convex polygons in a plane, and two points p and q outside any convex polygon, find the shortest path from p to q that does not go inside any convex polygon. Paths may travel on the edges of a polygon if these are not inside some other polygon.

Solution: Let V be the set of vertexes of the polygons plus the two points p and q. Then the path to be found can be represented as a sequence of straight line segments with vertexes in V (proof to reader). So it is a shortest path in an undirected graph whose vertexes are V such that given points r1 and r2 in V, there is an edge in this undirected graph between r1 and r2 if and only if the interior of the line from r1 to r2 does not intersect the interior of any polygon. Actually, we can make the computation simpler by also deleting an edge from r1 to r2 if the interior of the line from r1 to r2 contains any member of V, as the shortest path can be composed of line segments between members of V that do not contain other members of V.

Circular Anti-Maze

Find the shortest path between two points in a plane that has circular holes in it. The part of a path that transverses a hole does NOT count toward the length of the path.

Solution: let V be the set consisting of the two points and the centers of the holes. Then the path consists of straight line segments with end points in V. The length of each segment is the distance between its ends minus the radius of any hole centered at one end, or is 0 if this length is negative. The reader should try to prove that his works, even when holes overlap.

Robot Arms

Suppose we have a planar robot arm. Such an arm consists of line segments in an order. The beginning of each segment is a pivot point, around which a servo can rotate the segment and anything attached to its end. The end of a segment is attached to the beginning of the next segment, or to the robot hand if the segment is the last segment of the arm. We will define the end of the last segment to also be a pivot point: it could have a servo to rotate the hand.

The parameters of the arm are the lengths of the segments and the angular settings of the servos. A servo is typically set to have 0 angle if the segment following it continues in the same direction as the segment preceding it. We will assume that a positive angle means the arm following the servo is rotated counterclockwise by that angle. Particular robot arms may use other conventions for servo angles. We will also assume that the first pivot point is at the origin, and that the setting of 0 degrees for the first servo points the first segment along the positive x-axis.

The position of the pivot points of such an arm can be computed recursively by induction on the number of segments in the arm. If there are 0 segments, there is only one pivot point (the hand's), and it is at the origin. The setting of the servo at that pivot point is irrelevant to the position of the pivot points.

If there are more than 0 segments, assume for the moment that the first segment is such that its end (not its beginning) is at the origin and its orientation is in the direction of the positive x-axis, so its servo setting is 0 degrees.

Next compute the positions of the pivot points at the ends of the other segments by a recursive call, pretending that the first segment does not exist, so the arm has one fewer segments than it actually has.

Now translate the origin to the beginning of the first segment by adding the length of this segment to the x coordinate of every pivot point. The beginning of the first segment is now the origin. The setting of the first servo is still 0 degrees.

Now rotate all the pivot points about the origin by the amount indicated by the setting of the first servo. This finishes the computation.

This computation is easy because we use recursion to
build the arm from its end, and not from its beginning.

File: 2D_geometry
Author: Bob Walton <walton@deas.harvard.edu>
Date: See top of file.

RCS Info (may not be true date or author):

```
$Author: walton $
$Date: 2006/10/14 17:52:21 $
$RCSfile: 2D_geometry,v $
$Revision: 1.21 $
```

Breadth First Search Fri Sep 29 07:57:17 EDT 2006
Problem Help

Breadth first search is most commonly used in maze problems, but can be used in other problems. The idea behind search is that one has a set of nodes, each node has a list of other nodes that are its children, and the search begins at a start node and progresses from the current node to one of its children until the search finds a goal node.

The idea behind breadth first search is that one makes a 'visited list' of the nodes in the order that they are first visited by the search, and one has a pointer into this list at the first node whose children have not yet been examined. Then one iteratively examines the children of the node pointed at, adds any children that are not yet on the visited list to the end of the visited list, and bumps the pointer to the next node on the visited list. One stops when one gets to a goal node, or fails if one runs out of nodes to examine.

(Depth first search is similar, except that the children are added to the visited list immediately after the node that is their parent, the node currently being examined for children. Depth first search with a visited list is almost unheard of in programming contests, but recursive exhaustive search which visits ALL nodes in depth first order is common.)

There are three approaches to making a visited list:

A. If the set of possible nodes is small and can be organized into an array, don't do breadth first search, but instead use dynamic programming. See the help file on dynamic programming.

B. If the size of a node description is fixed and not large, and the maximum number of nodes that might be visited is not too large, make a large fixed size vector of nodes and two pointers: one to the next empty element and one to the first element whose children have not yet been examined.

C. Otherwise allocate each node as it is visited. You will need a pointer in each node to the next node (or NULL if there is none), a pointer to the first node, a pointer to the last node, and a pointer to the first node whose children have not yet been examined. When an input test case is complete, you will need to deallocate all the nodes starting with the first node.

Given your choice of visited list data structure, you need the following functions:

1. A function which given the parameters describing a node, determines whether the node is already in the visited list. This can be done by a linear search of the list. In some cases this is not fast enough, and a hash table is needed: see below.

2. A function which given the parameters describing a node creates a new node and adds it to the end of the visited list.

3. A function which checks a node to see if it is a goal node.

4. A function which given a node generates for each child of the node a call to function 1 to see if the child is on the visited list already, and if not, a call to function 2 to add the child to the list and then a call to function 3 to see if the added child is a goal.

Given these it is easy to code the algorithm in a few lines.

In order to construct the shortest path to the goal node, each node that is visited needs to record its parent. Then starting from the goal node and working backward a shortest path can be constructed.

If a linear search of all the visited nodes is not fast enough, a hash table can be used. The idea of a hash table is that instead of having one long lookup list, we have many short lookup lists. Suppose we want M lookup lists. We write a hash function that takes the parameters describing a node and returns an integer i, the hash value of the node, where 0 <= i < M. The node's hash value i names the lookup list on which the node will be placed. The hash table is a vector H such that H[i] is the head of lookup list i. Each node is now on two lists, a lookup list and the list of all visited nodes.

Then the function to find a node in the visited list computes the node's hash value i and linearly searches just list H[i]. The function to put a node on the end of the visited list must computed the node's hash value i and add the node to the H[i] list. All this works well if for typical sets of actual nodes the hash function generates its M possible values about equally often. Then the expected length of each lookup list will be N/M, where N is the total number of nodes and M is the number of lookup lists.

File: breadth_first_search
Author: Bob Walton <walton@deas.harvard.edu>
Date: See top of file.

RCS Info (may not be true date or author):

```
$Author: walton $
$Date: 2006/09/29 12:33:13 $
$RCSfile: breadth_first_search,v $
$Revision: 1.6 $
```