About This Contest Thu Oct 13 18:56:50 EDT 2005

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 command 'help' will display this file.

This is a 'formal' contest. The command

```
help formal_contest
```

displays more details about formal contests.

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'.

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.

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'. These are only for practice contests.

You CANNOT use a web browser. The 'javahelp' command gives on-line Java documentation: see 'help java'. You may use any printed matterial you like, but cannot communicate electronically with the world outside the contest.

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

Good hunting!

```
HELP INDEX                    Thu Oct 13 19:26:56 EDT 2005

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

formal_contest
        About formal contests, including logging in.

practice_contest
        About practice contests, including logging in.

informal_contest
        About informal contests, including logging in.

xterminals
        How to make windows if you are using and
        xterminal emulator.

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

solving
        How to write a solution file, how to test it,
        how to submit it.

input
        Advice on input formats.

output
        Advice on output formats.

print
        How to print files.

commonlisp
        How to write COMMONLISP solutions.


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

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

scoreboard
        Interpreting the scoreboard notation.


Advice:
-------

c++
        Special C++ advice.

java
        Special JAVA advice.

advice
        Polonic advice on how to select problems and
        find errors.


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

dynamic_programming
        Solving dynamic programming problems.

2D_geometry
        Solving 2D geometry problems.

breadth_first_search
        Solving problems (e.g. mazes) with breadth first
        search.
```

```
Formal Contest Help         Thu Oct  6 11:19:45 EDT 2005


In order to participate in a contest, you need to log
into your contest account.

Normally we make all contestants run on the same fast
computer.  This means that all contestants are subject
to the same load.

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
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 sup-
port 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 matter-
ial you like.  There is a contest command, javahelp, for
on-line Java documentation: see `help java'.

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 a practice contest,
so you can additionally submit new code to the auto-
judge, 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.

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: 2005/10/06 15:20:58 $
    $RCSfile: formal_contest,v $
    $Revision: 1.7 $
```

```
Practice Contest Help      Thu Oct  6 09:54:02 EDT 2005


Rules for Practice Contests:
----- --- -------- --------

The purpose of a practice contest is to practice for
other kinds of contest.

In practice contests it is possible to use 'in-submit'
and 'inout-submit' to get back the judge's input and
judge's output for the first test case for which a sub-
mission fails.  This CANNOT be done in formal contests,
so you need to be careful to wean yourself away from
this feature as you practice for a formal contest.

You will be given a contest account name and a password,
either by email or on a sheet of paper.   You can log
into the contest account from anywhere (e.g., if there
are not enough terminals in one place you can go to any
other place, or you can work at 'home').

You must use the secure shell (ssh) to log into the
contest computers.  You may use an X-Terminal equivalent
(X-server) in combination with ssh.  If you use an
X-Terminal equivalent, you can make new windows easily
with xcsh: see 'help xterminals'.

You may be able to use the print commands in an practice
contest, if you are located in the same building as the
printer.  The command 'printer' will tell you the name
of the printer.  Otherwise you can set up your account
to either use a different printer or to e-mail post-
script files of printouts to the e-mail address of your
choice.  See 'help print'.

The 'scoreboard' and 'help' commands work for a practice
contest.

Practice contests usually have a web page.  Help files
and problem descriptions are placed on the web page.
Sometimes paper handouts are used in place of a web
page.

There are no time limits on a practice contest, and no
timing of how long it takes for you to submit a
solution.  Instead scoring depends upon the number of
incorrect submissions you make, and the kind of these
submissions.  The 3 kinds of submissions and the
penalties usually assessed for each are

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

The first just returns your score, the second returns
the judge's input for the first case where your program
gave incorrect output or a formatting error, and the
third is similar but returns both the judge's input and
the judge's output.  The penalties are applied to your
current score for a problem, which is initialized to
100.00.  Scores for problems you never solve do not
count on the scoreboard.  See 'help scoreboard' for more
details.

For some practice contests you may be able to get the
judge's solution by executing

        make solution-submit

when you yourself have a CORRECT solution.  Doing an
extra submit in this way does not change your score.
```

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

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

```
    $Author: hc3 $
    $Date: 2005/10/06 14:29:56 $
    $RCSfile: practice_contest,v $
    $Revision: 1.2 $
```

Informal Contest Help Thu Oct 6 11:37:44 EDT 2005

Informal contests are a way of practicing for formal contests using timed scoring. In an informal contest you get each problem individually, and get the problem description as a file only when you get the problem. The start time of the problem is either the time you get the problem, or the time you get the first problem in the contest.

An informal contest can be set up so each team can participate at a different time, say one on Sunday afternoon and another on Tuesday evening. Or an informal contest can be set up so each contestant can solve each problem at a different time.

Rules for Informal Contests:
----- --- -------- --------

You will be given a contest account name and a password, either by email or on a sheet of paper. You can log into the contest account from anywhere (e.g., if there are not enough terminals in one place you can go to any other place, or you can work at 'home').

You must use the secure shell (ssh) to log into the contest computers. You may use an X-Terminal equivalent (X-server) in combination with ssh. If you use an X-Terminal equivalent, you can make new windows easily with xcsh: see 'help xterminals'.

You may be able to use the print commands in an informal contest, if you are located in the same building as the printer. The command 'printer' will tell you the name of the printer. Otherwise you can set up your account to either use a different printer or to e-mail postscript files of printouts to the e-mail address of your choice. See 'help print'.

The 'scoreboard' and 'help' commands work for an informal contest.

If you are simulating a formal contest precisely, you will not be allowed to use the alternative make commands that 'in-submit' or 'inout-submit'. However 'solution-submit' may work.

Your contest account will normally remain for some longish period of time after the contest so you can log in and email out your solutions, in order to save them.

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

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

```
    $Author: hc3 $
    $Date: 2005/10/06 15:48:23 $
    $RCSfile: informal_contest,v $
    $Revision: 1.6 $
```

X-Terminals Help Fri Nov 17 21:58:04 EST 2000

Creating Windows with Xcsh

If your ‘terminal’ is emulating an X terminal, 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: 2001/09/15 10:25:12 $
    $RCSfile: xterminals,v $
    $Revision: 1.7 $
```

Solving Problems Help Thu Oct 13 19:10:15 EDT 2005

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 practice contest, the above ‘hpcm_get’ gets ALL the problems, not just the demonstration problems.

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 an informal contest you need to get each problem one at a time, using the command

```
        hpcm_get problems/pppp
```

to get the problem names ‘pppp’. In an informal contest, problem start times are computed using the times you get problems using hpcm_get (see ‘help scoreboard’ for details).

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
```

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 a practice contest, the file is usually a subpage of the contest web page, although the file may also appear in the problem directory. In an informal contest, the problem description file is usually 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.

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 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.

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 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 practice 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 has to have permissions making it accessible to everyone.

Common Mistakes

If you get ‘Program Crashed’, 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 an output file crashes your program.

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 empty.

Submission Alternatives

In a practice contest there are alternative ways of submitting a problem.

make submit	Returns just score.
make in-submit	Returns score, and if score is ‘Incorrect Output’ or ‘Formatting Error’, 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 practice 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. Some contests may disallow some 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).

Formal or informal contests do not normally allow these alternatives.

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

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 gdb debugger works for C and C++. Gdb works well under EMACS.

For JAVA, making pppp also makes pppp.class, which can be run with the debugger jdb.

See 'help commonlisp' for COMMONLISP debugging.

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

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

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

    $Author: hc3 $
    $Date: 2005/10/13 23:14:34 $
    $RCSfile: solving,v $
    $Revision: 1.12 $

Input Help Thu Oct 7 08:04:59 EDT 2004

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 significant, and newlines are just like any other space character on input.

If you are writing in C you may need the ungetc function 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)
```

Generally speaking floating point computations should be done in double precision, and constants like PI should have double precision accuracy. Code such as

```
        double PI = 4 * atan ( 1.0 );
```

is useful.

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

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

```
    $Author: hc3 $
    $Date: 2004/10/07 12:16:37 $
    $RCSfile: input,v $
    $Revision: 1.10 $
```

Output Help Thu Oct 7 08:05:20 EDT 2004

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 description contains explicit instructions to use horizontal 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 different 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.

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: 2004/10/07 12:16:39 $
    $RCSfile: output,v $
    $Revision: 1.8 $

Print Help Wed Oct 5 07:53:01 EDT 2005

In a formal contest the contest staff will bring your printouts to you. The 'printer' command below may tell you something about the status of your printout.

In an informal contest run at a particular location and time, your printer may be defined for you, but you will have to go get your printouts yourself. Otherwise, in informal contests you typically have to define your printer yourself in the ~/PRINTER file or 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 in Informal Contests
------- ---------- -- -------- --------

In an informal contest you typically have to define your printer by putting its name in the ~/PRINTER file in your 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 postscript files are being emailed 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.

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: 2005/10/05 11:51:16 $
    $RCSfile: print,v $
    $Revision: 1.4 $

Commonlisp Help Mon Feb 18 07:53:00 EST 2002

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
```

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

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

```
    $Author: hc3 $
    $Date: 2002/02/18 12:57:14 $
    $RCSfile: commonlisp,v $
    $Revision: 1.1 $
```

Scores Help Thu Oct 6 09:33:52 EDT 2005

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 (or send by hpcm_sendmail, see below) will be Cc'ed to your account.

Asking Questions:

You may e-mail questions to the judges but the method is a little weird. First, edit your email in 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.

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 are 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 white-space, 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!

For problems in which only the numbers really matter, omitted, extra, or misspelled words are often 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.

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).

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.

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.

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 command 'help scoreboard' will display a description of the scoreboard format.

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 the end of a formal contest the final results will be announced in a post-contest meeting, as soon as the final submissions have been judged.

```
File:           scores
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: 2005/10/06 13:28:22 $
    $RCSfile: scores,v $
    $Revision: 1.11 $
```

Scoreboard Help Thu Oct 13 04:04:03 EDT 2005

Ranking Submitters
------- ----------

The scoreboard gives for each submitter (contest account) and for each problem information about the submissions made by the submitter for the problem. For each submitter the scoreboard also gives the total number of problems correctly solved by the submitter, and typically also gives a total score that distinguishes submitters who have the same number of solved problems.

A submitter may be a team or an individual, depending on the type of contest. Here we use the term 'submitter' to mean either 'team' or '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 correct solutions appear earlier in the scoreboard, and each problem solution is given a score indicated on the scoreboard.

When a contest is being scored, there needs to be some way of ranking submitters who have the same number of problems solved. There are two different methods of ranking. The classical way, used in all limited duration contests, is to measure the time required to solved each problem. The other way that we provide, usable in untimed contests, is to use the number and kind of incorrect submissions made for each solved problem. In either case, 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, untimed, and unscored 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, then '..../N' appears, where N is the number of incorrect submissions by the submitter for the problem.

For each submitter in a scored contest the number of solutions N and total score TTT of submitter are reported in the format 'N/TTT'. For timed contests higher TTT is worse, but for untimed contests higher TTT is better.

In some contests, problems are scored first by an auto-judge, 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 auto-judge, but are still awaiting review by the human judge.

Timed Contests
----- --------

When a contest is of limited duration and problems are being timed to produce a score, each problem is assigned a start time, which is the time the submitter first sees the problem description. For formal contests this time is the time the contest starts, and the submitters are handed problem descriptions on paper when the contest starts. A solution score is just the difference between the time the solution was submitted and the start time of the problem. The submitter's score is the sum (NOT the average) of all the solution scores for the solutions submitted by the submitter.

Most importantly note that lower time scores are better, higher time scores are worse. Also note that if you think of problems as being solved consecutively in a formal contest where all problems have the same start time, 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 submitters 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 total time score as described above is always in seconds.

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 directories. One kind of informal contest assigns each problem an individual start time that is the time of the hpcm_get command that gets the problem description. The other kind of informal contest is intended to closely simulate a formal contest but allow each submitter to start at a different time. In this kind of informal contest, each submitter is assigned a start time equal to the time the submitter first gets a problem using hpcm_get, and then each solution of that submitter is given this start time.

Untimed 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 commands used to make the different kinds of submission and the penalties typically assigned are:

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

The first, 'make submit', returns just the submission score. The second, 'make in-submit', also returns the judge's input for the first test case that was incorrect, provided the score was 'Incorrect Output' or 'Formatting Error'. The third, 'make 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 total score is the average (NOT the sum) of the solution scores for all problems solved by the submitter.

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 total score of a submitter, the solution score to many decimal places is used.

Unscored Contests

If a contest is unscored, 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.

```
File:           scoreboard
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: 2005/10/13 08:25:12 $
    $RCSfile: scoreboard,v $
    $Revision: 1.13 $
```

C++ Help Thu Feb 5 13:33:14 EST 2004

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;
using std::ws;
using std::setprecision;
using std::ios;
```

Detecting End of File

Typical program structure including end of file detection is:

```
int main ( )
{
    . . . .
    while ( true )
    {
        . . . . .
        cin >> x >> y >> z;
        if ( cin.eof() ) break;
        . . . . .
    }
}
```

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.

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;
```

Numeric Constants and Rounding
------- --------- --- --------

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);
```

To use setw etc. you need to:

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

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.

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

        #include <sstream>
        using std::istringstream;

        istringstream in ( buffer );

Here in.eof() will be true at the end of the string.

You can use the following for character input:

        int c = cin.get();
        int c = cin.peek();

These return EOF on end of file.



What May Not Work
-----------------

You should NOT try to use C++ templates or exceptions.
Features like these are not yet standardized across C++
compilers, and we have little faith in the ability of
our compiler to handle them sufficiently correctly
(templates mostly work, but ... ).

The 'bool' data type does work, however.
```


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



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

    $Author: hc3 $
    $Date: 2004/10/06 10:55:24 $
    $RCSfile: c++,v $
    $Revision: 1.7 $
```

JAVA Help Mon Oct 6 00:39:06 EDT 2003

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

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

```
        javahelp
```

You should NOT use other means to access such documentation, and in particular, using the internet typically is a violation of contest rules.

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.

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: hc3 $
    $Date: 2003/10/06 04:37:42 $
    $RCSfile: java,v $
    $Revision: 1.4 $
```

Help from Polonius Thu Oct 6 09:35:00 EDT 2005

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.

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.

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

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: hc3 $
    $Date: 2005/10/06 13:33:00 $
    $RCSfile: advice,v $
    $Revision: 1.8 $
```

DYNAMIC PROGRAMMING PROBLEM HELP

Tue Aug 6 07:50:17 EDT 2002

A dynamic programming algorithm is an algorithm with two characteristics:

A. A problem P is parameterized in such a way that the parametric set of subproblems can be solved recursively. For example, the original problem P might be parameterized as P(i,j,M) for some integers 0 <= i < N, 0 <= j < N, 2**M >= N with N > 0. And each problem P(i,j,m) for 0 <= i < N, 0 <= j < N, 0 < m <= M might be solvable fairly quickly (say in time proportional to N) from the solutions to all the problems P(i,j,m-1) for 0 <= i < N, 0 <= j < N, while the problems P(i,j,0) are readily solvable.

B. The solution to one of the parameterized problems (e.g. P(i,j,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.

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

For example, suppose we have an undirected graph with vertices connected by edges that have lengths. Suppose there a N vertices, numbered from 0 through N-1. Then the problem is to find the length of a shortest path between two given vertices, i and j.

Let P(i,j,m) denote the length of the shortest path between vertices i and j such that the path does not have more than 2**m edges. Then P(i,j,0) is easily computed: it is just the length of the edge from vertex i to vertex j, or infinity if there is no such edge, or 0 if i = j. Also, if m > 0, then P(i,j,m) can be computed from the solutions to P(i,k,m-1) and P(k,j,m-1) for all 0 <= k < N: it is just

```
    min { P(i,k,m-1) + P(k,j,m-1) : 0 <= k < N }
```

Going a little deeper into this, the values of P(i,j,m) for all i and j with m fixed form an NxN matrix, which we might call M(m). Then M(0) is just the original graph with edges expressed as lengths, infinities where there are no edges, and zeros down the diagonal. And for m > 0, M(m) is just the 'matrix product' of M(m-1) with itself, where instead of using * and + as in the normal matrix product, we use + and 'min'. Thus M(m) can be computed from M(m-1) in roughly N**3 operations, and the original problem can be solved in N**3 * (log base 2 of N) time.

Notice that one does not need to store M(m) for every value of m at one time. One only needs space for M(m-1) and M(m) to compute M(m). Also, instead of using infinity as a matrix element value, one can use any number known to be larger than the solution path length.

Dynamic programming can also be used to compute an actual shortest path, and not just the length of such a path. Note that the shortest path algorithm above will work even if edges have negative lengths, as long as no loop in the graph has a negative length (if a loop had a negative length the algorithm could not stop after log base 2 of N iterations). We will assume here that the length of each loop is not equal to zero. One can always make this true by replacing each path length L in the original problem by L*N+e, where e is the number of edges in the path.

Then we extend the value of P(i,j,m) to include the second vertex on a shortest path from i to j that does not have more than 2**m edges. After the computation is done, a path from i to j can be constructed by going from i to the computed second vertex on a best path from i to j, and iterating until one gets to j. This works because if one best path goes from i to k to j then replacing the part of this path from k to j by any best path from k to j also gives a best path, and also because best paths cannot have loops as the length of every loop is > 0.

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 i and j. 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.

Dynamic programming solutions are often easier to code than other solutions to a problem. One possible use of this is as an alternative to breadth first search for finding a shortest path, in the sense of a path having a minimum number of edges, between two vertices of a graph. Note, however, that breadth first search takes O(|E|) time, where |E| is the number of graph edges, whereas the dynamic programming algorithm we have presented takes O(|V|**3 log |V|) time, where |V| is the number of vertices. But it is not necessary to find the shortest path between any two pairs of vertices to solve the breadth first search problem. You can use dynamic programming to find the set of vertices that can be reached from a given starting point by a path with at most m edges by recursion on m. This gives the same answer as breadth first search with less bookkeeping but in O(|S||E|) time, where |S| is the length of the shortest path being sought.

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: hc3 $
    $Date: 2002/08/06 11:55:24 $
    $RCSfile: dynamic_programming,v $
    $Revision: 1.7 $

2D GEOMETRY PROBLEM HELP   Sun Aug 4 09:36:13 EDT 2002

For some reason many young computer scientists think that geometry problems are hard, and avoid them. The purpose of this short 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

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

is the vector from p to q. Its length is

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

The scalar product of two vectors

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

is

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

$$= |u|\ |v| \cos(\theta)$$

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

$$|v| = \mathrm{sqrt}\ (v.v)$$

Note that u.v = 0 iff 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.

If u is a unit vector and v is any vector, we call (u.v)u the projection of v onto u. Taking vectors as denoting points (the ends of the vectors if all have the same origin), then the projection of v onto u is the point on the infinite line going through the origin and u that is closest to the point v. That is, u.v is the number K which minimizes $|\ v - Ku\ |$. You can check this with a little algebra.

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

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

$$= \left| \begin{matrix} vx \cos(\text{theta}) + vy \sin(\text{theta}) \\ -vx \sin(\text{theta}) + vy \cos(\text{theta}) \end{matrix} \right|$$

Most particularly, if theta = 90 degrees, then

$$R(90 \text{ degrees}) = \left| \begin{matrix} 0 & 1 \\ -1 & 0 \end{matrix} \right|$$

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

These are all the basics you need to know. However, you need to know these basics 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?

Simply compute:

    v = R(90 degrees).(q-p) = (qy-py,-qx+px)

    Note: then v.q = v.p.

    if v.r > v.p then r is to the right of the line;
    if v.r < v.p then r is to the left of the line;
    if v.r = v.p then r is on the line.

v is the vector from p to q rotated 90 degrees clockwise, so it points to the right of the line through p and q. Then

$$\frac{q - p}{|q - p|} \qquad \text{and} \qquad \frac{v}{|v|}$$

are a pair of unit length vectors at right angles to each other and form a coordinate system. Therefore

$$\frac{v.r}{|v|}$$

is the coordinate in this coordinate system in the direction perpendicular to the line through p and q, and this is >, =, or < to v.p/|v| according to whether r is to the right, on, or to the left of this line.

Note that if p, q, and r have integer coordinates, then v.r and v.p are integers and are exact, and the classification of r as to the right, on, or to the left of the line is exact, with no chance of roundoff error.

Sometimes the distance from r to the line from p to q is desired. This is just the absolute value of

```
          v.r
          ---
          |v|
```

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.

This can be readily solved if the line is parallel to the x-axis. Suppose it is and the x coordinates of p and q are px < qx, and the x coordinate of r is rx. Then if rx <= px, the distance from r to the line is the distance from r to p. Similarly if rx >= qx then the distance from r to the line is the distance from r to q. But if px < rx < qx, then distance from r to the line is the distance from r to the point on the line with the same x coordinate as r. This is just the absolute value of ry - py (note py = qy).

In order to find the distance when the line is not parallel to the x-axis, we introduce new coordinates in which the line will be parallel to the x-axis. To do this we find orthogonal unit vectors u and v such that u is parallel to the line. Then the new x coordinates are u.r, u.p, u.q and the new y coordinates are v.r and v.p = v.q.

Intersection

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.

Answer: It does iff the infinite line through p1 and p2 splits q1 and q2, and the infinite line through q1 and q2 splits p1 and p2, where for an infinite line to split two points means that one point must be to the right of the line and the other point must be to the left of the line. Note that we can assume p1 != p2 and q1 != q2, as otherwise at least one of the two finite lines would have no interior.

Suppose instead we wanted to know whether the finite lines including end points intersect. Then we change the definition of split so that an infinite line splits two points iff one point is on or to the right of the line and the other point is on or to the left. Also we can mix our two definitions of ‘split’ to test whether one finite line, including its end points, intersects the interior of another finite line.

Convex Hull

The 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 line through p(i) and p((i mod N)+1), and such that no point p(j) is on this line except of course p(i) and p((i mod N)+1). Then the finite lines from p(i) to p((i mode 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 find the convex hull, 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 iff q2 is to the right of
                             the line from p to q1
                             or q2 is on this line and
                             closer to p that 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|**2) where |V| is the number of points in V. The fastest algorithm is the Gram-Schmidt algorithm that begins with a sort of V and has running time dominated by the sort time, O(|V| log|V|).

Inside Convex Polygons

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

Then a point r is inside the convex polygon but not on the boundary of the polygon iff r is to the right of each clockwise oriented side of the polygon. Or more specifically, r is to the right of the infinite line that extends the side.

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. 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 the clockwise sides of the polygon and check that the interior of the finite line from r1 and r2 intersects the interior of some side, or that for each polygon side, r1 and r2 are both on or to the right of the infinite line extending the side, and r1 and r2 are not both on that line. 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 vertices 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 vertices in V (proof to reader). So it is a shortest path in an undirected graph whose vertices are V such that given points r1 and r2 in V, there is an edge in this undirected graph between r1 and r2 iff 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.

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 clockwise 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. Otherwise 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 server setting is 0 degrees. Next compute the positions of the pivot points at the ends of the other segments by induction.

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 induction 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: hc3 $
    $Date: 2002/08/04 15:10:13 $
    $RCSfile: 2D_geometry,v $
    $Revision: 1.12 $
```

BREADTH FIRST SEARCH PROBLEM HELP
Wed Aug 7 12:12:15 EDT 2002

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.

The key to programming breadth first search is to understand how you are going to represent the following data structures:

A. The 'visited list' of all the nodes that have been visited by the search.

B. An 'allocated list' of all nodes. This is needed to free memory allocated to the nodes when the algorithm is finished. Sometimes this can be the same as the visited list A.

C. For each node, a list of the node's children.

D. For each node, a bit indicating whether the node is on the visited list A. Of course this is just an efficiency matter, but it is important except for very small problems.

E. For each node, a 'parent pointer' to the parent node which caused the node to be put on the visited list A. This pointer is used to construct the shortest path list F.

F. For each node on the shortest path from the start node to a goal, a 'shortest path pointer' to the next node on the shortest path. This pointer can be computed by traversing the shortest path in reverse order from goal to start node using the 'parent pointer' E.

The various items above can be encoded in a 'node' data structure. The members of this data structure can be:

0. Information naming the node.

A. A next_visited pointer that points at the next node in the visited list A.

B. A previous_allocated pointer that points at the previous node on the node list B. This is needed only if nodes must be allocated before each search begins and freed after each search ends.

C. A children pointer at a list of cons cells that list all the children of this node. This is needed only if the list of children of a node cannot easily be generated on the fly from some simpler data structure; e.g., if the list is read from input.

D. A 'visited' flag if nodes are allocated before the search and the list of children C exists. Otherwise, some way is needed of quickly looking up a node name to see if the node is already on the visited list. It may be necessary to employ a hash table, putting nodes in a hash bin and having a next_hashed pointer from each node to the next node in its hash bin.

E. A 'parent' pointer to the parent node which caused
   this node to be put on the visited list A.

F. A 'next_shortest' pointer to the next node on the
   shortest path.

Note that there are a number of variants to basic breadth first search. Sometimes the starting node can also be a goal, sometimes there are many goal nodes, etc.

Given the right definition of a node structure, breadth first search is easy enough to code.

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: hc3 $
    $Date: 2002/08/07 16:24:55 $
    $RCSfile: breadth_first_search,v $
    $Revision: 1.3 $
```