```
About This Contest         Wed Oct 15 10:26:01 EDT 2003

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 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 demonstra-
tion problem.

Hpcm_get will also get all the contest problems into
your '~/problems' directory AFTER the contest official-
ly 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 submit-
ted.

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 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'.

Good hunting!
```

```
HELP INDEX                    Wed Oct 15 10:11:20 EDT 2003

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

formal_contest
        About formal contests, including logging in.

informal_contest
        About informal contests, including logging in.

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

email_contest
        How to be a contestant in an email contest.

email_unix_tools
        How to use Bourne or tclsh shell tools in email
        contests.

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.


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

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

dynamic_programming
        Solving dynamic programming problems.

2D_geometry
        Solving 2D geometry problems.

3D_geometry
        Solving 3D geometry problems.

network_flows
        Solving network flow problems.

optimal_matchings
        Solving optimal matching problems.
```

```
Formal Contest Help        Fri Nov 17 22:00:22 EST 2000


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.

At the end of the contest you should email your code
to your accounts in the world outside so you have
a record of what you did.  Note that email coming
into the contest computer may be controlled, but email
out usually is not.
```

```
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: hc3 $
    $Date: 2001/09/15 10:25:13 $
    $RCSfile: formal_contest,v $
    $Revision: 1.6 $
```

```
Informal Contest Help      Sat Nov 18 06:54:21 EST 2000


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 ast the
printer.  The command `lpq' will tell you the name of
the printer.  Otherwise you will have to email files
you want to print from your contest account to an
account that can access a printer hear you.

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

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.

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: 2001/09/15 10:25:13 $
    $RCSfile: informal_contest,v $
    $Revision: 1.4 $
```

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 $
```

Email Contest Help Tue Sep 24 06:53:53 EDT 2002

In order to participate in an email contest, you need an account on a machine of your own in which to solve problems. This account should support the programming languages you want to use, and should support email. This account will communicate with the contest autojudge through email.

The Simplest Contest

For the simplest contest you are given a web page with complete instructions and problem descriptions. The word 'contest' is something of a misnomer for this activity, as there is no scoring, and no accounting of how long someone takes to solve a problem.

The contest web page contains the email address of an autojudge. In order to submit a solution, you simply email a solution file to the autojudge. The subject of this message must be

```
        Subject: submit pppp.eee
```

where pppp is the name of the problem and .eee is an extension that designates the programming language in which the solution file is written. Some possible extensions are

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

The body of the email message must contain the solution file and nothing else (the file must NOT be an attachment). The solution file must be a one file program that reads from the standard input and writes to the standard output and does NOT open any files.

For the simplest contest problems are not usually timed and submitters are not ranked in any order. There is typically a 'scoreboard' that lists submitters who have solved a problem in alphabetic order and shows for each submitter the problems the submitter has solved and the date of each solution.

Sometimes the set of contestant email addresses is restricted. For example, only email with a sender address ending in '.harvard.edu' might be accepted by the autojudge.

Private Contests

Private contests are protected by a password. The email address of the autojudge typically has the form 'fee<fie@fo.fum>' where 'fee' is a password that is not to be revealed to the public. The contest typically has a web page with URL of the form

```
        www.fiddle/~diddle/private/fee
```

where fee is the same password as in the email address, and 'private' is a directory that cannot be read by the public, so to access the web page a person has to know the password.

You should not put the URL in any public web page, so as not to reveal this password. The scoreboard is posted on the web page, along with information about how to participate in the contest.

```
Getting Help Files
------- ---- -----

Help files giving details about contests are usually on
public web pages.  Given an email contest, one can get
a complete set of help files by sending email to the
autojudge with the subject 'get help' and an empty mes-
sage body.  The reply will contain many files (including
this file).  The files are separated by lines of the
form:

        ---<>--<>---<>--<>---<>----- filename:

You should first search the reply for and read the file
named 'help/this_contest'.  This gives information about
the specific email contest.

If you have a UNIX system, follow the directions in the
file 'help/email_unix_tools' that is part of the reply,
and do NOT read the rest of the help file you are cur-
rently reading.  If you do NOT have a UNIX system, con-
tinue reading.


Getting Started on a Non-UNIX System
------- ------- -- - -------- ------

If you do NOT have a UNIX system, you need NOT extract
any files from email replies.  You will probably want to
print the reply to the 'get help' message and read many
of the 'help' files therein.  You can ignore all the
'help/sh' and 'help/tcl' files, as these are UNIX tools.

Demonstration problems are also typically on the web,
but a complete set of demonstration files can be
obtained by sending a message to the autojudge with the
subject 'get demos' and empty body.

You can practice submitting a solution by sending one
of the 'demos/count' solutions to the autojudge.  For
example, if you send a message to the autojudge with
the subject 'submit count.c' and a body equal to the
contents of the 'demos/count/count1.c' file, you should
get back a reply saying the submission was 'Completely
Correct'.  Read the 'demos/count/README' file for more
details.

Needless to say, NONE of the UNIX commands that support
a contest account will work if you do not have a UNIX
system.  You CANNOT use 'print', 'scoreboard', 'make',
'help', etc.



Working Problems on a Non-UNIX System
------- -------- -- - -------- ------

You can find out what problems are available by looking
at the file 'help/problems' (in the reply to the
'get help' message).  Suppose you decide to attempt a
problem named 'pppp' in the C++ language.  Send email to
the autojudge with the subject 'get problems/pppp' and
an empty body.  The reply will contain a description of
the problem in the form of a file with a name like one
of the following:

    problems/pppp/pppp.txt      ASCII text.
    problems/pppp/pppp.html     HTML.
    problems/pppp/pppp.htm      HTML.
    problems/pppp/pppp.ps       Postscript.

The reply also contains a file named 'problems/pppp/
Makefile' which is not of direct use, since you do not
have a UNIX system, but which does contain the resource
limits applied to the problem during judging.
```

Then write a solution to the problem in C++ as a file named pppp.cc. To submit the solution, send email to the autojudge with subject 'submit pppp.cc' and message body equal to just the contents of the file 'pppp.cc'. If it is more convenient, you can use a name other than 'pppp.cc' for the file name, but you MUST use 'submit pppp.cc' as the email subject.

Programming languages other than C++ can be used by changing the .cc to a different extension, as indicated near the beginning of this help file.

```
File:           email_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: 2002/09/24 11:26:16 $
    $RCSfile: email_contest,v $
    $Revision: 1.12 $
```

Email UNIX Tools Help Thu Oct 31 06:18:12 EST 2002

Before you read this file, read the ‘help/email_contest’ file up to the section in that file entitled ‘Getting Started on a Non-UNIX System’.

Getting Started in a UNIX System

First make a ‘home directory’ for the contest you wish to enter. You can enter many contests, and should have a separate home directory for each. In this home directory place a file named ‘HPCM_ADDRESS’ that contains the email address of the contest (more specifically, of the autojudge for the contest). Thus if the contest email address is ‘fee<fi@fo.fum>’, execute in the contest home directory:

```
echo 'fee<fi@fo.fum>' >HPCM_ADDRESS
```

From now on work in this contest home directory.

Note that it is a bad idea to use your UNIX home directory ‘~’ as a contest home directory. Each contest you enter should have its own unique home directory.

You will be running various HPCM programs written in the UNIX Bourne shell (sh) language. Some versions of sh do not support the ‘-r’ option to the ‘read’ command, the ‘-e’ option to the ‘test’ command, or the use of backslash as a non-interpreted output character by the ‘echo’ command. If ‘/bin/sh’ has any of these problems, you have to choose one of two fixes. If your system has a GNU bash shell, be sure this is available using your your PATH under the name ‘bash’. Otherwise, if your system has a TCL shell, be sure this is available using your PATH under the name ‘tclsh’. If our system has neither, get either from:

```
www.gun.org/software/bash/bash.html   for bash
www.scriptics.com                     for tclsh
```

Next, send email to the contest email address with the subject ‘get help’ and an empty message body. The reply will contain many files (including the file you are now reading).

Copy this reply into the file named help.email in your contest home directory. Manually extract either of the files help/sh/hpcm_extract or help/tcl/hpcm_extract from this help.email file into your contest home directory as the hpcm_extract file. Choose sh or tcl according to whether you have bash or another adequate Bourne shell, or you have chosen instead to use tclsh. The files in help.email are separated by lines that match the regular expression ‘^---<>’, that is, that begin with the characters ‘---<>’. Using this fact you can fairly easily extract the correct hpcm_extract file with an editor.

Next execute the following UNIX commands in your contest home directory:

```
chmod 700 hpcm_extract
./hpcm_extract < help.email
chmod 700 help/sh/* help/tcl/*
```

This should extract all the files into the ‘help’ subdirectory and make the files in ‘help/sh’ and ‘help/tcl’ executable. The hpcm_extract file, when it executes, will log its actions to the standard output.

At this point you can remove the files help.email and hpcm_extract from the contest home directory. There will be other copies of hpcm_extract in the ‘help/sh’ and ‘help/tcl’ subdirectories.

Next add the ‘help/sh’ subdirectory to your PATH so you can execute the programs in this directory. Assuming your contest home directory is named ‘~/hpcm/practice’, and assuming you use the ‘csh’ shell, add

```
set path=(~/hpcm/practice/help/sh $path)
```

in your ‘~/.cshrc’ file. If instead you use the Bourne shell, add

```
PATH=$HOME/hpcm/practice/help/sh:$PATH
```

in your ‘~/.profile’ file.

Next, if you chose to use tclsh, add ‘help/tcl’ to your PATH just as you added ‘help/sh’. Be sure ‘help/tcl’ is BEFORE ‘help/sh’ in the path.

In any case, after editing the relevant file, log out and log back in again, and then check for errors by typing the UNIX command:

```
hpcm_get -doc | more
```

This should output the documentation for the hpcm_get program. You can output the documentation of the other programs in ‘help/sh’ or ‘help/tcl’ by executing them with just the ‘-doc’ option, if you wish, but this should not be necessary. If you chose to use tclsh, the command ‘which hpcm_get’ should return the name of the hpcm_get file in the ‘help/tcl’ subdirectory.

Lastly, to automatically extract files from email replies to requests generated by hpcm_get, add a copy of ‘~/hpcm/help/sh/procmailrc’ to your ‘~/.procmailrc’ file, and edit the copy according to the instructions in what you copied. This is all you need to do if your UNIX system uses the procmail program to deliver mail (as does RedHat Linux). If your system delivers mail using a different mailer (as does SUN UNIX), you may have to put the following line in your ‘~/.forward’ file:

```
"| exec /usr/bin/procmail"
```

to cause your ‘~/.procmailrc’ file to be used (include the "’s in the ‘~/.forward’ file). Here ‘/usr/bin/procmail’ should be identical to the output of the UNIX command ‘which procmail’.

If you cannot arrange for automatic extraction of files gotten by hpcm_get, you will have to manually input each reply containing files to the ‘hpcm_extract’ UNIX command. E.g., copy a reply into a file ‘foo’ and execute the command:

```
hpcm_extract < foo
```

within the contest home directory.

To add more contests later, you merely create a contest home directory for each contest, and a HPCM_ADDRESS file within that directory. You do NOT have to repeat any of the other actions described above.

Programs like hpcm_extract, hpcm_get, and hpcm_submit work by searching the current directory and its ancestors for a readable HPCM_ADDRESS file, and using the first directory they find as the contest home directory. Thus if hpcm_submit is executed in the ‘problems/ppp’ subdirectory of the contest home directory, it will identify ‘../..’ as the contest home directory.

If you want more up to date versions of the files obtained by 'get help' or 'get demos', use the command

```
        hpcm_get -force
```

to repeat these messages. Then read the output from hpcm_extract's processing of the reply. This output will tell you which files have changed and therefore been updated. You then delete your local versions of these changed files and execute 'hpcm_get -force' one more time to get the updated versions of these files.

If you get an updated version of any file in the 'help/sh' or 'help/tcl' subdirectories, re-execute:

```
        chmod 700 help/sh/* help/tcl/*
```

Commands that Do NOT Work
-------- ---- -- --- ----

The following commands do NOT work for an email contest.

```
    print, print2,      Use your favorite file print
    fprint, lpq         program instead.

    scoreboard          See the scoreboard URL instead.

    help                Instead you can print the email
                        reply to 'get help' or files
                        extracted therefrom.
```

Testing the Demo in a UNIX System
------- --- ---- -- - ---- ------

The UNIX command 'hpcm_get', executed in the contest home directory, will get all the demonstration files into the 'demos' subdirectory of the contest home directory. If you go to the '/demos/count' subdirectory, you will find a demonstration problem complete with solutions in all the supported programming languages. The README file in this directory will tell you how to install any of these solutions and practice running it and submitting it.

Working Problems in a UNIX System
------- -------- -- - ---- ------

You can find out what problems are available by looking at the contest file 'help/problems'. Suppose you decide to attempt a problem named 'ppp' in the C++ language. Then within the contest home directory execute the UNIX command 'hpcm_get problems/ppp' which will send email to the autojudge to get that problem. After files are extracted from the reply, the problem will be in the 'problems/ppp' subdirectory of the contest home directory. A 'ppp.txt', 'ppp.html', 'ppp.ps' (postscript), or similar file in that directory will describe the problem. The 'Makefile' in that directory will permit you to compile, run, and submit the problem, and will contain the resource limits applied to the problem during judging.

For example, to write a solution to the problem in C++, you need to write the file 'problems/ppp/ppp.cc'.
See 'help/solving' for more details.

```
File:           email_unix_tools
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: 2002/10/31 11:18:41 $
    $RCSfile: email_unix_tools,v $
    $Revision: 1.16 $
```

Solving Problems Help Mon Feb 18 07:05:52 EST 2002

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, all you need to do to get the problems is type the command

```
        hpcm_get
```

AGAIN, after the contest starts. 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
```

In an informal or email contest you have to get each problem one at a time, because the scoreboard uses the time you get a problem as the problem start time (or may use the time your team gets its first problem as the start time for all your team's problems). To get a problem named 'pppp' in an informal or email contest use the command:

```
        hpcm_get problems/pppp
```

In email contests the 'demos' and 'problems' directories may NOT be subdirectories of your home directory '~'. They might, instead, be subdirectories of '~/hpcm', or of some other directory. See 'help email_contest' and 'help email_unix_tools'.

Working on a Problem

For a problem named 'pppp' you should write a file named 'pppp.c', 'pppp.cc', 'pppp.java', 'pppp.p', 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
        .p      for PASCAL
        .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. In a formal contest, a printout of this file is given to you at the start of the contest, but the file does NOT appear in the problem directory. In an informal or email contest, 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.

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 write 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, p2c on
            pppp.p., or a commonlisp compiler on
            pppp.lsp, depending upon which of
            pppp.c, pppp.cc, pppp.java, pppp.p,
            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.p2c which is made from
            pppp.p, 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, pppp.p, 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).
```

    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.

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: 2002/02/18 12:57:14 $
    $RCSfile: solving,v $
    $Revision: 1.8 $

Input Help Tue Nov 5 00:40:53 EST 2002

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 PASCAL, you may want to use the code in demos/pascal/pascal.p for free-format input.

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: 2002/11/05 06:28:50 $
$RCSfile: input,v $
$Revision: 1.9 $
```

Output Help Tue Nov 5 00:32:47 EST 2002

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 newline 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, :10:6 in PASCAL, 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: 2002/11/05 06:28:51 $
    $RCSfile: output,v $
    $Revision: 1.7 $
```

```
Print Help                    Mon Feb 18 05:27:12 EST 2002

In a formal contest the contest staff will bring your
printouts to you.  The lpq 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).

    lpq

        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.

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: 2002/02/18 10:49:28 $
    $RCSfile: print,v $
    $Revision: 1.3 $
```

```
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 31 06:16:04 EST 2002

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 symbols 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!

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 and was not terminated, and some symbol or number in your output does not match the corresponding symbol or number in the judge's output, even when letter case is 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 are such cases. Although a human judge who is manually reviewing scores could change the score in such a case, she may not, on the grounds that contestants 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: 2002/10/31 11:16:34 $
    $RCSfile: scores,v $
    $Revision: 1.9 $
```

Scoreboard Help Mon Sep 23 05:41:31 EDT 2002

Interpreting the Scoreboard

The scoreboard tells for each submitter (contest account) and for each problem whether the submitter has submitted a correct solution to the problem and when this solution was submitted. The total number of submissions by the submitter for the problem, including incorrect submissions before the first correct submission, may also be given.

A submitter may be a team or an individual, depending on the type of contest.

If the contest is being scored, the number of correct submissions by each submitter is displayed, and submitters with greater numbers of correct submissions appear earlier in the scoreboard.

Problems may or may not be timed. If they are not timed, then only the date of the first correct submission of a problem is displayed. If problems are timed, the elapsed time between the problem start time and the time of the first correct submission is displayed. For a formal contest the start time of all problems is the start time of the contest. For informal contests, and for email contests that are timed, the start time of a problem may be the time the problem was gotten by the submitter using hpcm_get, or the time the submitter got its first problem using hpcm_get.

If a contest is being scored and problems are being timed, submitters with the same number of correct problems are ranked according to their total time, which is just a sum the times the submitter needed to solve each problem. The submitter with the smallest total time ranks highest among all submitters with the same number of correct problems, and appears first in the scoreboard.

More specifically, the total time of a submitter is the sum of the problem elapsed times, in seconds, for all problems the submitter has solved. Optionally a penalty may added for each incorrect submission of a problem before the correct submission of that problem. Typical penalties, if they are being used, are 10 to 20 minutes. There is no penalty for incorrect submission of a problem never submitted correctly by a submitter.

If there are no submissions for a problem by a submitter, `......' appears in the scoreboard for the submitter and problem. If there are only incorrect submissions, and incorrect submissions are being reported on the scoreboard, then `..../N' appears, where N is the number of incorrect submissions for the problem.

If a submitter has submitted a correct solution for a problem, and the elapsed time between the problem start time and the time of the correct solution is being reported, this time may be represented as `M:SSsN', with decimal numbers M, SS, and N, meaning that the problem 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 problem time used for computing a submitter's total time as described above is always in seconds.

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 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.

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


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

```
    $Author: hc3 $
    $Date: 2002/09/23 09:42:49 $
    $RCSfile: scoreboard,v $
    $Revision: 1.8 $
```

C++ Help Mon Feb 18 04:55:19 EST 2002

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.

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.

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: 2002/02/18 10:04:06 $
    $RCSfile: c++,v $
    $Revision: 1.4 $
```

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.

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 Mon Oct 6 00:31:19 EDT 2003

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

Many times 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).

```
File:           advice
Author:         Bob Walton &lt;walton@deas.harvard.edu&gt;
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 04:34:20 $
    $RCSfile: advice,v $
    $Revision: 1.7 $
```

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.

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

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

$$|v| = sqrt(vx**2 + vy**2)$$

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

$$|v| = sqrt\ (v.v)$$

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

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

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

Most particularly, if theta = 90 degrees, then

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

R(90 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

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

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 $
```

3D GEOMETRY PROBLEM HELP   Mon Aug 5 12:13:35 EDT 2002

Read 2D Geometry Problem Help first.

Basics

The basics of 3D geometry are mostly identical to the basics of 2D geometry. We review.

If

```
        p = (px,py,pz) and q = (qx,qy,qz)
```

are points in space, the difference

```
        v = (vx,vy,vz) = p - q = (px-qx,py-qy,pz-qz)
```

is the vector from p to q. Its length is

```
        |v| = sqrt( vx**2 + vy**2 + vz**2 )
```

The scalar product of two vectors

```
        u = (ux,uy,uz) and v = (vx,vy,vz)
```

is

```
        u.v = ux*vx + uy*vy + uz*vz

            = |u| |v| cos(theta)
```

where theta is the angle between u and v, measured in the plane through the origin and the end points of u and v, assuming u and v have a common origin. The sign of theta does not matter as cos(theta) = cos(-theta). Note that

```
        |v| = 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, v, and w are unit vectors at right angles to each other, which is the same as saying that

```
        |u| = 1,  |v| = 1,  |w| = 1
        u.v = 0,  v.w = 0,  v.u = 0
```

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

```
        (u.p, v.p, w.p)
```

Note that

```
        | u.p |   | ux  uy  uz | | px |
        |     |   |            | |    |
        | v.p | = | vx  vy  uz | | py |
        |     |   |            | |    |
        | w.p |   | wx  wy  wz | | pz |
```

so u, v, and w are the rows of the 3x3 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.

In 3D, rotations occur around an axis. The following is a rotation Rz(theta) around the z-axis of a vector v=(vx,vy,vz) by an angle theta:

$$Rz(theta).v = \begin{vmatrix} \cos(theta) & \sin(theta) & 0 \\ -\sin(theta) & \cos(theta) & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} vx \\ vy \\ vz \end{vmatrix}$$

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

Most particularly, if theta = 90 degrees, then

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

$$Rz(90\ degrees).v = (vy,-vx,vz)$$

We can see from this last that the signs of the sin(theta) elements of the rotation matrix have been chosen so that if the x-axis goes toward your right, the z-axis goes up, and the y-axis goes away from you, then a rotation with theta > 0 is clockwise looking down on the xy-plane.

One can also write matrices easily for rotations about the x-axis or y-axis.

Next we ask: suppose we are given a unit vector w. Can we find a coordinate system u, v, w. That is, can we find unit vectors u and v such that u, v, and w are all orthogonal to each other.

First we find any unit vector v orthogonal to w. To do this we pick a candidate v' and compute

$$v'' = v' - (v'.w)w$$

Then v''.w = 0 and therefore

$$v = \frac{v''}{|v''|}$$

is a unit vector orthogonal to w. HOWEVER, what if |v''| is nearly zero. We must avoid this situation as it will lead to inaccuracies or computational failure.

We do this by trying the following three values of for v':

```
ix = (1,0,0)
iy = (0,1,0)
iz = (0,0,1)
```

If |v''| is near zero, then v' must be nearly parallel to w. But not all three of these ix, iy, and iz can be nearly parallel to w, so we can choose one of them to be v' and have |v''| not be very small (in fact, it will be at least 1/3).

Thus we have found a suitable v. To find u one plays the same game. Choose u' and compute

$$u'' = u - (u.v)v - (u.w)w$$

and

$$u = \frac{u''}{|u''|}$$

To avoid too small a value of u'', try using all ix, iy, and iz as u'. One will give a non-small value of |u''| (in fact, it will be at least 1/3).

These are all the basics you need to know for 3D.

The Distance of a Point to a Line

In order to find the distance of a point r to the infinite line through points p and q, we need to find the x coordinate of points in a coordinate system in which the line is parallel to the x-axis. So we need a unit vector u parallel to the line, and this can be readily found from the vector from p to q. Then the new x coordinates are u.p, u.q, and u.r.

The points on the line are: $p + K(q-p)$

for all the real numbers K. The new x coordinates of these points are:

$$u.p + K(u.q-u.p)$$

Solving: $u.p + K(u.q-u.p) = u.r$

we get:

$$K = \frac{u.r-u.p}{u.q-u.p}$$

which allows us to determine the point on the line with the same new x coordinate as r. This is the point closest to r, so we first calculate this point, and then its distance to r.

If instead we want to know the distance between r and the finite line from p to q, we can calculate K as above, then replace any K < 0 by 0, replace any K > 1 by 1, and then the point on the line closest to r is just the point on the line defined by this K. As an example of the effect of limiting K to the range [0,1], if u.r < u.p < u.q, K < 0 is replaced by K = 0, so the closest point to r on the line is p.

Closest Point of Two Lines

Suppose we want to find the closest point of approach of two infinite lines L1 and L2.

If one of the lines L1 is parallel to the z-axis, then for each point p2 on the other line L2, the point p1 on L1 that is closest to p2 has the same z coordinate as p2. From this it follows that the distance between L1 and L2 is the same as the distance between the projections of L1 and L2 on the xy-plane. The projection of L1 on the xy-plane is just a single point; while the projection of L2 is in general an infinite line. The problem of finding the distance between a point and an infinite line in 2D has been solved elsewhere.

So we can solve the problem if we can find a coordinate system in which L1 is parallel to the z-axis. This can be done by taking the unit vector w in the direction of L1 and finding u and v such that u,v,w are orthogonal unit vectors: see above.

Now consider the problem of finding how close two finite lines L1 and L2 are. Let L1 be parallel to the z-axis, and let its end points p1 and p2 have coordinates z1 and z2, with z1 < z2. Then break L2 into three parts:

```
    L2a = the part of L2 in the z-halfspace z <= z1
    L2b = the part of L2 in z-partspace z1 <= z <= z2
    L2c = the part of L2 in the z-halfspace z2 <= z
```

Then the nearest point on L1 to any point on L2a is p1, so the distance from L2a to L1 is the distance from p1 to L2a. The nearest point on L1 to any point on L2b has the same z coordinate as the point on L2b, so the distance from L2b to L1 is the distance of the projections of L2b and L1 on the xy-plane. The projection of L1 is a single point, so this is the 2D distance between a finite line and a point. Lastly the distance from L2c to L1 is the distance from L2c to p2. Taking the minimum of the three distances from L2a, L2b, and L2c to L1 gives the distance from L2 to L1.

Robot Arms
----- ----

Robot arms in 3D can have the positions of their pivot points computed just like planar (2D) robot arms. The only difference is that each servo is generally oriented along some axis, e.g. the z-axis, and the rotation defined by the servo is around the servo's axis.

File: 3D_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: 2002/08/05 16:21:22 $
    $RCSfile: 3D_geometry,v $
    $Revision: 1.5 $

NETWORK FLOWS HELP   Wed Aug 7 23:35:32 EDT 2002

A network flow problem can be reduced to the problem of finding a shortest path with non-zero flow. Alternatively it can be reduced to the dynamic programming problem of finding a path with maximum flow.

A flow network is a directed graph in which every arrow is labeled with a capacity, and we are asked to find the maximum amount of fluid (or whatever) that can flow from a given source vertex to a given destination vertex, assuming each arrow is a uni-directional pipe whose capacity is the maximum fluid flow on that pipe. See the Maximum Flow chapter of Cormen, Leiserson, and Rivest or the Advanced Combinatorial Algorithms chapter of Atallah.

The simpler algorithms find an path flow P from source to destination and subtract P from the original flow network N to get a residual flow network R. P is a path from source to destination each of whose edges has positive, non-zero capacity in N. The flow of P is the minimum capacity of all P's edges. It is this flow that must be subtracted from the capacities of N, for each edge in P, to get the capacities of R. R and N are then related, in that every flow FR of R that obeys R's capacity restrictions is related to a flow FN of N that obeys N's capacity restrictions with the FN equal to FR plus the flow of P.

The path flow P is called an 'augmenting path' because the maximum flow of N is the sum of P and the maximum flow of R.

One iterates the above algorithm until the residual network has no remaining path with non-zero flow. There is then an easy lemma saying that the maximum flow of this residual network is zero. It follows that the maximum flow of the original network is the sum of all the augmenting paths found by the iterations.

Some cleverness is required in computing residual networks. If the original network has capacities Cxy from vertex x to vertex y, and Cyx from vertex y to vertex x, and if the path being subtracted has flow f > 0 from x to y, then the residual network has capacities Cxy-f from x to y and Cyx+f from y to x. Even if Cyx was originally 0, and there was no explicit arrow from y to x. This is because a flow of f from y to x in the residual network can be obtained by reducing the flow subtracted out of the original network to 0.

The hard part of all this is finding the right augmenting paths. A sloppy algorithm can find paths that have too little flow, and the require too many iterations. E.g., consider the network:

```
          1,000,000,001       1,000,000,000
       +-----------------> C ---------------> +
       |                   |                  |
       |                   |                  |
       |     2,000,000,000|                  |
       |                   |                  |
       |                   v                  |
       |                   D                  |
       |                   |                  v
       B                   |2                 G
       |                   v                  ^
       |                   E                  |
       |                  ^ |                 |
       |                  | |                 |
       |    2,000,000,000| |2                |
       |                  | |                 |
       |  1,000,000,000   | v 1,000,000,000   |
       + ----------------> F -----------------+
```

Here the problem is to find the maximum flow from B to G. A depth first search that prefers the arrow of greatest capacity and does not revisit vertices would find the path B->C->D->E->F->G in the first iteration, which only has a flow of 2, and then find the path B->F->E->D->C->G in the second iteration, which only has the flow of 2, and then repeat these two iterations over and over. There would be about a billion iterations in all. But if the first iteration found B->C->G and the second B->F->G, the algorithm would complete in just two iterations.

The easiest good algorithm for finding augmenting paths is to just find a shortest augmenting path. The number of augmenting paths that must be found can be shown to be $O(|E|\ |V|)$, where $|E|$ is the number of edges and $|V|$ is the number of vertices in the original flow network.

A slightly better solution is to find an augmenting path with maximum flow. Calculating the maximum flow of any path from source to destination is an easy dynamic programming problem (see help file on dynamic programming). It can be shown that the number of maximum flow augmenting paths that must be found is at most $O(|E| \log C)$ where $|E|$ is the number of edges and C is the maximum capacity of any edge in the original flow network.

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

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

    $Author: hc3 $
    $Date: 2002/08/08 03:37:08 $
    $RCSfile: network_flows,v $
    $Revision: 1.6 $

OPTIMAL MATCHINGS HELP Sat Sep 21 05:13:37 EDT 2002

A matching is a 1-1 function from some domain set D into some range set R. As a function, it is a subset of ordered pairs (x,y) in DxR. Suppose a weight number w(x,y) is assigned to each such pair. Define the weight of a function to be the sum of the weights of its pairs. Then an optimal matching is a matching that either minimizes or maximizes the weight of the matching.

In what follows we will define V to be the set D union R and call elements of V 'vertices', and we will refer to pairs (x,y) in DxR as 'edges'. For the rest of this document we will refer to the same fixed D, R, and V.

A signed graph is an assignment of one of the numbers -1, 0, or +1 to each edge (x,y) in DxR. We will consider a function (and therefore a matching) to be a signed graph that assigns 0 to each (x,y) not in the function and +1 to each (x,y) in the function. If G is a signed graph, the edges assigned +1 are called the positive edges of G, and the edges assigned -1 are called the negative edges of G. Edges assigned 0 are said to be not in G.

We define a path P to be a signed graph such that every vertex in V is an endpoint of at most one positive edge of P and at most one negative edge of P. Vertices that are endpoints of a positive edge but not a negative edge are called positive endpoints of the path, and vertices that are endpoints of a negative edge but not a positive edge are called negative endpoints of the path. Note that a path can have any number of positive or negative endpoints, including none.

A path P is connected if its edges can be written in a sequence such that the signs of the edges alternate and each pair of consecutive edges shares at least one endpoint. A connected path either has no endpoints, and is then called a loop, or has exactly two endpoints.

The sum of two signed graphs G1 and G2 is the signed graph G1 + G2 that assigns to each edge the sum of the numbers assigned to the edge by G1 and G2. G1 + G2 will not exist if G1 and G2 share a positive edge or share a negative edge, as no edge in G1 + G2 can be assigned +2 or -2.

The difference of two signed graphs G1 and G2 is the signed graph G1 - G2 that assigns to each edge the number assigned to the edge by G1 minus the number assigned to the edge by G2. G1 - G2 will not exist if some edge is positive in G1 or G2 and negative in the other graph, as no edge in G1 - G2 can be assigned +2 or -2.

Two signed graphs are said to be disjoint if no edge is assigned a non-zero value in both graphs. A signed graph G1 is said to be a subgraph of a signed graph G2 if and only if G1 can be made from G2 by changing the assignment of some edges to zero. It can be shown that any path is the sum of disjoint connected subpaths.

If M is a matching and P a path, then P is said to be an augmenting path of M if M + P exists and is also a matching. If M1 and M2 are matchings, it can be shown that M2 - M1 is a path, and therefore it is an augmenting path of M1, with M2 = M1 + (M2 - M1).

It can be shown that a path P is an augmenting path for M if and only if (1) all negative edges in P are positive in M, (2) each edge of P has one endpoint in D and one endpoint in R, and (3) all positive endpoints of P are not in the domain or range of M. If follows that the connected components of an augmenting path for M are also augmenting paths of M.

If M1 and M2 are matchings with

    domain(M2) = domain(M1) union {x}

for some element x of D, then it can be shown that the connected components of M2 - M1 are of three kinds. First, there may be loops. Second, there is exactly one non-loop that has two positive endpoints one of which is x and the other of which is in R. Third, there may be non-loops that have one positive and one negative end-point, both in R.

Let w(x,y) be a weighting function from edges (x,y) in DxR to numbers. Define the weight of a signed graph G, w(G), to be the sum over all edges (x,y) of w(x,y) mul-tiplied by the number (-1, 0, or +1) assigned to (x,y) by G. Then if G1 and G2 are signed graphs, w(G1 + G2) = w(G1) + w(G2) if G1 + G2 exists, and w(G1 - G2) = w(G1) - w(G2) if G1 - G2 exists.

Suppose we search for a matching M with domain(M)=D and minimal weight w(M). We can build M by adding one element at a time to its domain. Let M1 be a minimal matching on its domain which is a strict subset of D, and let x be an element of D not in the domain of M1. Let M2 be a minimal matching with domain(M2) = domain(M1) union {x}. Then M2 - M1 is an augmenting path with w(M2) = w(M1) + w(M2 - M1). Therefore to find M2 we merely need to find an augmenting path P for M1 which has minimal w(P), and then set M2 = M1 + P.

Suppose this P has a component P' which does NOT have x as an endpoint. P' is an augmenting path for M1 and if we set M1' = M1 + P', we find that M1' and M1 have the same domain and w(M1') = w(M1) + w(P'). Since M1 is minimal for its domain, we have w(P') >= 0 necessarily. w(P) is the sum of the weights of the components of P, so if w(P') were > 0, then w(P) could not be minimal. Therefore w(P') = 0, and we can omit P' from P and still get a minimal matching on domain(M1) union {x}. More specifically, if P'' = P - P', then P'' is a subpath of P and an augmenting path of M1, and if M2'' = M1 + P'', w(M2'') = w(M2) is minimal.

So to find M2 we need merely find a minimal connected augmenting path P for M1, and this path will have x as one of its positive endpoints, and some point in R but not in range(M1) as its other positive endpoint.

Such a path P is determined by a sequence of vertices x, x1, x2, ..., xk, y where x1, ..., xk are elements of domain(M1) and y is an element of R not in range(M1). There are unique elements y1, y2, ..., yk in range(M1) such that (x1,y1), (x2,y2), ..., (xk,yk) are edges of M1, and these become the negative edges of P. The posi-tive edges of P are (x,y1), (x1,y2), (x2,y3), ..., (xk,y). We merely need to find x1, x2, ..., xk, y such that w(P) is minimal.

Consider the directed graph H with vertices in domain(M1) union {x} and weighted edges as follows. There one edge from xa to xb if xb is in domain(M1), and the weight of (xa,xb) is w(xa,yb) minus w(xb,yb), where (xb,yb) is the edge in M1 with xb as an endpoint. Then w(P) is the weight in H of the path x, x1, x2, ..., xk plus w(xk,y), so since w(P) is minimal, so is the path x, x1, x2, ..., xk minimal from x to xk in H. Also, since augmenting paths that are loops for M1 have zero weight, loops in H can also be shown to have zero weight, and we can choose a minimal path from x to xk that has no loop. So to find P, we need merely find minimal paths without loops in H from x to each point in xk in domain(M1), and then exhaustively search over all xk and all endpoints y in R.

In summary, we build a minimal matching one domain point at a time, finding a minimal augmenting path P for each new domain point. To find P, we compute H which depends upon the matching found so far, find minimal paths without loops in H, and then find the minimal P by an exhaustive search over points in the domain of the matching found so far and points in R. Maximal matchings can be built similarly by finding maximal augmenting paths.

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

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

```
    $Author: hc3 $
    $Date: 2002/09/21 09:22:53 $
    $RCSfile: optimal_matchings,v $
    $Revision: 1.4 $
```