ACS Programming Challenge

Welcome to the ACS programming challenge!

Introduction

The goal of this challenge is to give you a chance to show how you would solve a small programming task, while at the same time give you a glimpse into the kind of problems we come across on a regular basis. Please write this program as if it were to be a small part of a larger collection of tools that will be maintained and modified over time by various people.

Problem

Given a list of files with their sizes and a list of compute nodes with their available space, write a program that produces a *distribution plan* of how to place the files on the nodes. The plan should make the total amount of data distributed to each node as balanced as reasonably possible. Note that this is different from balancing the amount of available space remaining on each node after the distribution.

In the case where not all the input data will fit on the nodes, the program should place what data it can, and the output will indicate which files were not included in the plan (see the Output section, below).

Your solution should run as a UNIX command-line program and be written in one of these languages:

- Python (2.x)
- Java
- C or C++

The following sections provide a precise definition of the inputs the program must handle, the output that is expected, and how the program will be invoked from the UNIX command-line.

Inputs

There are two input files: `files` and `nodes`. Both have the same structure: simple text files, with one item per line, using the newline character `\n` to separate lines.

Specifically:

- Any blank line or line beginning with the comment character ('#') should be ignored.
- Any other line will have two fields separated by white space.
    - The first field is string representing either the file or node name
    - The second field is a positive integer representing the size in bytes of either the file or the available space for the given node.

Example of `files`:

```
# filename size
tom.dat 1024
jerry.dat 16553
tweety.out 12345
elmerfudd.txt 987654321
```

Example of `nodes`:

```
# node-name available-space
node1 65536
node2 32768
```

Output

The program should produce a distribution plan showing the node to which each file is assigned, including any files that could not be assigned to a node. The format of this plan is a simple text file, with one item per line, using the newline character `\n` to separate lines. Each line should have two items separated by whitespace: the name of the file and the name of the node to which the file was assigned. If the file could not be assigned to a node, i.e. because there wasn't enough space for all the files, then use the name `NULL` for that node (you can assume that no input node will be called `NULL`). Please do not include any other lines in the output, i.e. no headers, blank lines, or comment lines.

Example `distribution`:

```
tom.dat node1
tweety.out node1
jerry.dat node2
elmerfudd.txt NULL
```

Invocation

To make it easier for us to evaluate the solution, your program must be named in a certain way and handle a certain set of command-line arguments. From within the top-level directory of the solution you submit, we should be able to run your program like this:

- Java: `java Solution [OPTIONS]` OR `java -jar solution.jar [OPTIONS]`
- C/C++: `./solution [OPTIONS]`
- Python: `python solution.py [OPTIONS]`

In all three cases, the [OPTIONS] supported should be, in any order:

- `-f filename`: Input file for *files*, e.g. `-f files.txt`. Required.
- `-n filename`: Input file for *nodes*, e.g. `-n nodes.txt`. Required.
- `-o filename`: Output file, e.g. `-o result.txt`. If this option is not given, assume standard output.
- `-h`: Print usage information to standard error and stop

Errors

Although the overall focus of this challenge is to see how well the program solves the challenge for correct and feasible inputs, the program should gracefully handle incorrect inputs of all types.

If there are errors in the invocation or inputs the program should exit with an indication of the problem and a usage message.

Submission

Please submit your answer as an archive of a directory which is your first and last (family) name separated by an underscore, with the archive given the same name. The archive can be in tar, compressed tar, or zip format (i.e. .tar, .tar.gz, or .zip). For example, if Elmer Fudd was to submit his

program, he would name the submission file `Elmer_Fudd.tar`; running `tar xf Elmer_Fudd.tar` will create a directory, `Elmer_Fudd/`, that contains the solution.

For Java solutions please include the source files (.java) as well as compiled class files. Do not include any external dependencies: all files necessary to run the program should be present in the submitted archive, or part of the J2SE.

For C/C++ programs, please include a simple UNIX shell script called `build.sh` that will compile and link the program. The source code should build and link with a modern GNU compiler (4.x), without any additional external libraries. You may use a Makefile, but no other build systems (automake, SCons, IDEs).

For Python programs, just make sure the program can be run from within the submitted directory, as specified above.

You do not need to submit additional input or output files, or documentation. Any description or explanations should be placed as comments in the source files themselves.

** Thanks, and happy programming! **