Bin packing

Tech firm `dowhatnow.com` has just hired you as a consultant to help them reduce costs in their customer service department. They digitally record customer service phone calls and store all the sound files locally on portable disks at the end of each week. Since your marginal cost is proportional to the total number of disks used (not the exact amount of space required, as in cloud storage fees), your task is to assign the sound files to disks using as few disks as possible.

Unfortunately, this type of optimization calculation belongs to a class of algorithms that are "NP-hard"; essentially this means it is impossible to calculate an *optimal* solution in a reasonable amount of time for large data sets (though optimal solutions can be *verified* fairly quickly). Instead, your goal is to design heuristics that run fast and produce high quality (though not necessarily optimal) solutions.

We formulate the *bin packing* problem as follows: given a set of N file sizes between 0 and 1,000,000 KB (f 1 GB), find a way to assign them to a minimal number of disks, each of capacity 1 GB. The *worst-fit* heuristic is a simple rule that considers the file sizes in the order they are presented: if the sound file won't fit on any disk, create a new disk; otherwise place the file on a disk that has the *most* remaining space. For example, this algorithm would put the sizes 700,000, 800,000, 200,000, 150,000, 150,000 onto three disks: {700,000, 200,000}, {800,000, 150,000}, {150,000}. Note that this does not necessarily lead to the best solution since the five files could fit on two disks.

Perspective. The bin packing problem is a fundamental problem for minimizing the consumption of a scarce resource, usually space. Applications include:

- Packing the data for Internet phone calls (VoIP) into ATM packets.
- Allocating blocks of computer memory.
- Assigning commercials to TV station breaks.
- Cloth, paper, and sheet metal manufacturers use a two-dimensional version of the problem to optimally cut pieces of material (according to customer demand) from standard sized rectangular sheets.
- Shipping companies use a three-dimensional version to optimally pack containers or trucks.
- Minimizing the number of identical processors needed to process a set of tasks by a given deadline.
    - In this example, the scarce resource is time instead of space

Disk data type. First, implement a data type `Disk.java` that represents a 1GB disk, and contains a list of all of the files it is storing. This data type should implement the `Comparable<Disk>` interface so that you can use it with a ***priority queue***, a data structure that organizes elements based on their priority (in the case, the most remaining storage space). See the powerpoint for more info.

Input and output. Your client program `WorstFit.java` will read in the set of file sizes (guaranteed to be between zero and a million) from standard input and implement the "Worst-fit" heuristic describe previously. Your program should output the number of disks used by the heuristic and the sum of all file sizes divided by 1 million (a lower bound on the number of disks required). If the number of files is less than 100, you should also print out the disks in decreasing order of remaining space. For each disk, print out its remaining space and its contents (in the order the file sizes were inserted). Optionally, you may also print out a unique ID associated with each disk (assigned in the order the disks were created) to aid in debugging.

Test your WorstFit class with the "input20.txt" file. You output should match the following:

```
Total size  = 6.580996 GB
Disks req'd = 8

     # Avail
     5 325754: 347661 326585
     0 227744: 420713 351543
     7 224009: 383972 392019
     4 190811: 324387 484802
     6 142051: 340190 263485 254274
     3 116563: 347560 204065 331812
     2 109806: 396295 230973 262926
     1  82266: 311011 286309 320414
```

Worst-fit *decreasing*. Experienced travelers know that if small items are packed last, they can fit snugly in the odd gaps in nearly filled luggage. This motivates a smarter strategy: process the items (files) from biggest to smallest. The *worst-fit decreasing* heuristic first preprocesses the file sizes so that they are in descending order. Create a new client program `WorstFitDecreasing.java` that implements this small optimization technique, your output should change as follows:

```
Total size  = 6.580996 GB
Disks req'd = 7

      #  Avail
      1  211686:  396295 392019
      0   94485:  484802 420713
      6   47762:  262926 254274 230973 204065
      5   44188:  324387 320414 311011
      3   18470:  347661 347560 286309
      4    1413:  340190 331812 326585
      2    1000:  383972 351543 263485
```

(Advanced) Better heuristics

Design and implement a heuristic that consistently beats worst-fit decreasing.

Your algorithm should work for large N, so it must take time proportional to N log N on average (as with worst-fit). One idea is *first-fit decreasing* where you insert the next file in the first disk that has enough room to store it. Use a heap-like structure called a *tournament tree*. Another idea is *best-fit decreasing* where you insert the next file in the disk that has the least remaining space among disks capable of storing the file. Add a method `ceil()` to a binary search tree that takes a disk x and returns the disk whose remaining capacity is closest to x without going under.