Announcements

Test 3 will be on Monday, December 11 (for the Monday/Wednesday section) or Thursday, December 14 (for the Tuesday/Thursday section).

Projects 6 and 7 are due by midnight on December 14 (I will check them on December 15 and email you complaints, with fixes due by 5:00 p.m. on December 18).

All Routine Exercises that you have not gotten checked off and are forced to give to me in writing or send in email will follow the same schedule. Recall that in order to get at least a C for the course you must have all Routine Exercises checked off, and get at least around 70% average on the three tests.

⇒ **Routine Exercise 36** (Branch and Bound for 0-1 Knapsack)

Perform the branch and bound algorithm for the 0-1 Knapsack problem following the rules and node format given in the pre-reading/video for the instance with

$W = 15$ and

i	p_i	w_i	p_i/w_i
1	64	4	
2	90	6	
3	91	7	
4	48	4	
5	33	3	
6	10	1	

⇒ **Project 6**

Your job on this project is to write code in Java to implement the branch and bound algorithm for the 0-1 knapsack problem.

Create a Java application named `Proj6` that will ask the user for the name of a data file, read the information from the data file, and then solve that instance of the 0-1 knapsack problem using the branch and bound method, following exactly the algorithm demonstrated in Embedded Exercise 42, but without the display of the tree, of course.

The data file must contain the capacity of the knapsack, followed by the number of items, followed by followed by the profit and weight information (on one line) for each item. All these values are integers. You may assume that the items are sorted in order from most profitable per unit of weight to least profitable.

As a small but important requirement to avoid irritating me, you *must* label the items starting with 1, rather than 0.

Here is a sample data file:

```
12
6
100 4
120 5
88 4
80 4
54 3
80 5
```

Here is a sample run on this data file, showing the output your program must produce (within cosmetic differences), and, implicitly, the algorithm it must use:

```
Capacity of knapsack is 12
Items are:
1: 100 4
2: 120 5
3: 88 4
4: 80 4
5: 54 3
6: 80 5

Begin exploration of the possibilities tree:

Exploring <Node 1:   items: [] level: 0 profit: 0 weight: 0 bound: 286.0>
      Left child is <Node 2:   items: [] level: 1 profit: 0 weight: 0 bound: 268.0>
         explore further
      Right child is <Node 3:   items: [1] level: 1 profit: 100 weight: 4 bound: 286.0>
         explore further
         note achievable profit of 100

Exploring <Node 3:   items: [1] level: 1 profit: 100 weight: 4 bound: 286.0>
      Left child is <Node 4:   items: [1] level: 2 profit: 100 weight: 4 bound: 268.0>
         explore further
      Right child is <Node 5:   items: [1, 2] level: 2 profit: 220 weight: 9 bound: 286.0>
         explore further
         note achievable profit of 220

Exploring <Node 5:   items: [1, 2] level: 2 profit: 220 weight: 9 bound: 286.0>
      Left child is <Node 6:   items: [1, 2] level: 3 profit: 220 weight: 9 bound: 280.0>
         explore further
      Right child is <Node 7:   items: [1, 2, 3] level: 3 profit: 308 weight: 13 bound: 308.0>
         pruned because too heavy

Exploring <Node 6:   items: [1, 2] level: 3 profit: 220 weight: 9 bound: 280.0>
      Left child is <Node 8:   items: [1, 2] level: 4 profit: 220 weight: 9 bound: 274.0>
         explore further
      Right child is <Node 9:   items: [1, 2, 4] level: 4 profit: 300 weight: 13 bound: 300.0>
         pruned because too heavy

Exploring <Node 8:   items: [1, 2] level: 4 profit: 220 weight: 9 bound: 274.0>
      Left child is <Node 10:   items: [1, 2] level: 5 profit: 220 weight: 9 bound: 268.0>
         explore further
      Right child is <Node 11:   items: [1, 2, 5] level: 5 profit: 274 weight: 12 bound: 274.0>
         hit capacity exactly so don't explore further
         note achievable profit of 274

Exploring <Node 2:   items: [] level: 1 profit: 0 weight: 0 bound: 268.0>
```

```
      pruned, don't explore children because bound 268.0 is smaller than known achievable profit 274

Exploring <Node 4:   items: [1] level: 2 profit: 100 weight: 4 bound: 268.0>
      pruned, don't explore children because bound 268.0 is smaller than known achievable profit 274

Exploring <Node 10:   items: [1, 2] level: 5 profit: 220 weight: 9 bound: 268.0>
      pruned, don't explore children because bound 268.0 is smaller than known achievable profit 274

 Best node: <Node 11:   items: [1, 2, 5] level: 5 profit: 274 weight: 12 bound: 274.0>
```

Your application must display on screen each node when it is first reached, showing the items selected at that node, the total profit for those items, the total weight for those items, and the bound on that node.

Note that your algorithm must use a priority queue to replace manually looking at all the nodes in the current tree to determine which has the best bound. Whenever a node is explored, it should either be pruned, or should be removed from the priority queue with both its children added to the priority queue.

You may implement the priority queue without worrying about efficiency (if you use a heap data structure removing the highest priority item and adding an item will both have efficiency in $\Theta(\log n)$, but it is okay to just use a list of some kind and have removing the highest priority—best bound—node take $\Theta(n)$, with adding being in $\Theta(1)$.

Be sure to test your algorithm thoroughly.

To submit your work, send me an email with the single file `proj6.jar` (or `proj6.zip` if you prefer) attached, containing all the source code for your project, in portable form (no meaningless `package` statements or hard-coded file names).

Announcement I'm canceling Project 7, due to lack of time. But, note that the last Routine Exercise, given below, will require you to be able to solve a TSP instance by using branch-and-bound with the bound coming from solution of an LP (Project 7 was just going to be a harder version of this work, so is not a great loss).

⇒ **Routine Exercise 37** (Heuristic Solution of TSP)