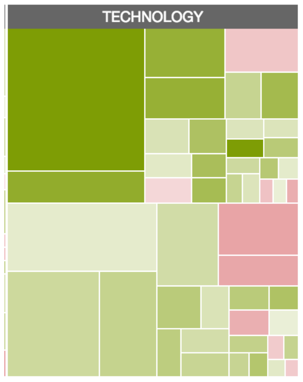
Treemaps

A treemap is a common information visualization tool that compactly represents hierarchical data. A square or rectangle is partitioned into smaller rectangles which are in turn partitioned into smaller rectangles. This process continues until one is left with a set of base rectangles where each base rectangle represents some information or data. The area of the rectangle is proportional to the size of the data. The color of the rectangle is often used to display some other aspect of the data. For example, below is a treemap displaying stock data for the technology sector of the market. The size of the rectangle is proportional to the share of the market and the color is how the stock has risen or fallen over the past month.



Treemaps and Binary Trees

Although it may not be clear at first, treemaps are intimately related to binary trees. If you look closely at the image above, the rectangle is partitioned horizontally at the center into two smaller rectangles. These two rectangles are then vertically partitioned, respectively, into two smaller rectangles. This process continues until you reach the individual rectangles that represent the data and collectively tile the entire space.

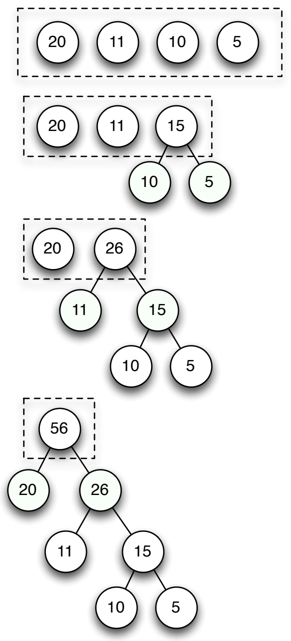
There are many procedures for building treemaps, but here we will study a *greedy* algorithm—one that makes choices in the present using only current information. Here's how it works. We start by creating a tree for each item in the list of data. For simplicity, we just assume the data is a list of n weights, but in practice it could be a list of any objections, just as long as there was some notion of weight or amount. In each step, we find the two trees with the smallest weight. We combine these trees to form a new tree. The weight of this tree is the sum of the weights of its two children. After $n-1$ iterations, we are left with a single tree.

Algorithm 1 BUILDTREEMAP$(data)$

Require: A list of n data items. We assume each item is a weight.

```
$T \leftarrow$ a list of $n$ trees $T = T_1 \ldots T_n$ where each tree is a single (a leaf) with one item of data
Sort $T$ from highest weight to lowest weight
while $|T| > 0$ do
    $Z_1 \leftarrow$ the last tree in $T$
    $Z_2 \leftarrow$ the second-to-last tree in $T$
    Replace the final two trees in $T$ with a new tree $Z$. The left child of $Z$ is $Z_1$. The right child of $Z$ is $Z_2$. The weight of $Z$ is the sum of the weights of $Z_1$ and $Z_2$.
end while
return The final tree in $T$
```

Below is a figure that shows how the algorithm runs on a set of four data items.



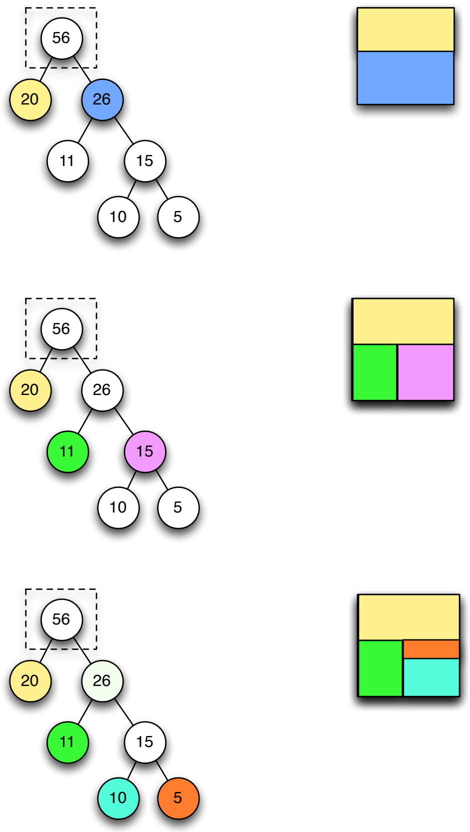
Here is the Python code. A few notes:

- Notice that instead of searching for the two trees with the smallest weights, we just sort the trees by weight.
- We sort in reverse order so that the trees with smaller weight are at the end. To sort in this way, use the optional keyword argument `reverse`.
- We also use the optional keyword `key`, which is a function that is applied to the data before comparing it. This allows us to sort trees by comparing their root values.

```
def build_treemap(data):
    """
    Takes a list of Stocks and returns a binary tree, constructed greedily,
    where each internal node represents the market cap of its two children,
    and each leaf is a Stock.
    """
    trees = [Tree(d) for d in data]
    trees.sort(reverse=True, key=lambda t: t.root)
    while len(trees) > 1:
        tree1 = trees.pop()
        tree2 = trees.pop()
        trees.append(Tree(tree1.root + tree2.root, tree1, tree2))
        trees.sort(reverse=True, key=lambda t: t.root)
    return trees[0]
```

Trees to Treemaps

The following figure shows how we take a binary tree of weights, along with, say, the unit square, and partition the square into rectangles that tile it.



Here is an outline of the algorithm, which returns a list of rectangles that collectively partition the unit square (i.e., the square with side lengths 1.0). Each rectangle corresponds to a leaf of the tree. The algorithm is recursive. Given a rectangle r, an orientation o (i.e., either *horizontal* or *vertical*), and a tree t:

leaf If the t is a leaf, then return the the list containing r; and

non-leaf if t is not a leaf, then

1. split r into two smaller rectangles r_1 and r_2 along the axis given by o using weight proportional to the left and right subtrees respectively;
2. recursively find the partition of r_1 by making a recursive call on the left subtree, passing r_1 and the opposite orientation of o;
3. recursively find the partition of r_2 by making a recursive call on the right subtree, passing r_2 and the opposite orientation of o;
4. return the concatenation of these two partitions.