Here the greedy-algorithm design pattern applied to another problem.

HUFFMAN CODING ....

input: An increasing sequence $p = (p_1, \ldots, p_n)$ of *frequencies* $(0 \le p_1 \le p_2 \le \cdots \le p_n)$.

output: A rooted binary tree T whose n leaves consist of the n frequencies. The tree should have minimum cost, defined as $\mathsf{cost}(T) = \sum_{i=1}^{n} p_i \, \mathsf{depth}_T(p_i)$, where $\mathsf{depth}_T(p_i)$ is the depth of the leaf for p_i in T. The cost is the (weighted) average leaf depth.

The problem models compressing a given file in a certain way. (See e.g. here.)

It's not obvious a-priori what a greedy algorithm for this problem should look like, because a solution T is not just a subset of objects that can be chosen one by one. But we apply the same basic idea: to start, we look for some piece that must be "part" of an optimal tree. Let's consider some examples. Our goal is to discover properties that an optimal solution should have.

Consider an example where $p = (0.5, 0.5)$. The optimal tree T has two leaves, each of depth 1, and has cost 1. In fact, this tree is optimal for any instance with $n = 2$ frequencies. Next consider an example where $p = (0.25, 0.25, 0.5)$. There is (up to isomorphism) just one tree T with $n = 3$ leaves. (Why is that?) One leaf has depth 1, the other two are siblings at depth 2. We should make p_1 and p_2 the depth-2 siblings. This way the tree has cost $1 + 0.25 + 0.25 = 1.5$

Fix an arbitrary instance p. Define $\mathcal{T}$ to be the set of trees T for p in which leaves p_1 and p_2 are siblings. We argue next that there must be an optimal tree for p that is in $\mathcal{T}$.

First, does an optimal tree T always have to have *some* two leaves that are siblings? Yes — consider any deepest leaf ℓ in T; if ℓ doesn't have a sibling, then its parent only has one child. So we could replace the parent by ℓ and get a better tree.

Observation 1. *There is an optimal tree T for p in which two of the deepest leaves are siblings.*

Next we show that, by an exchange argument, these siblings may as well be p_1 and p_2.

Lemma 1. *There is an optimal tree T for p that is in $\mathcal{T}$.*

Proof (short form). The proof is an exchange argument. Let T be *any* optimal tree. Let p_i and p_j (with $i < j$) be two deepest leaves that are siblings (Observation 1). Let tree T' be obtained from T by exchanging p_1 and p_i. By calculation, $\mathsf{cost}(T') \le \mathsf{cost}(T)$. Here is the calculation:

$$\begin{aligned}
\mathsf{cost}(T') - \mathsf{cost}(T) &= p_1[\mathsf{depth}_{T'}(p_1) - \mathsf{depth}_T(p_1)] + p_i[d\,\mathsf{depth}_{T'}(p_i) - \mathsf{depth}_T(p_i)] \\
&= p_1[\mathsf{depth}_{\,T}\,(p_i) - \mathsf{depth}_T(p_1)] + p_i[\mathsf{depth}_{\,T}\,(p_1) - \mathsf{depth}_T(p_i)]. \\
&= (p_1 - p_i)[\mathsf{depth}_T(p_i) - \mathsf{depth}_T(p_1)] \\
&\le 0.
\end{aligned}$$

The last line follows from $p_1 \le p_i$ and and $\mathsf{depth}_T(p_i) \ge \mathsf{depth}_T(p_1)$.

Next, let T'' be the tree obtained from T' by exchanging p_2 and p_j. Note that $p_2 \le p_j$ (as $j > i \ge 1$), and $\mathsf{depth}_T(p_j) \ge \mathsf{depth}_T(p_2)$, so the same calculation as above shows $\mathsf{cost}(T'') \le \mathsf{cost}(T')$. From this it follows that T'' is also optimal. And since p_1 and p_2 in T'' replace p_i and p_j in T, leaves p_1 and p_2 are siblings in T''. So $T'' \in \mathcal{T}$. □

By Lemma 1, p has an optimal tree in $\mathcal{T}$. So it's enough to find a tree that is optimal among trees in $\mathcal{T}$? How do we do that? The next key insight is that in any tree $T \in \mathcal{T}$, *we can think of the 3-node subtree consisting of p_1, p_2, and their parent as a single leaf having frequency $p_1 + p_2$.*

Next we make this intuition precise.

Definition 1. *Define p' to be the instance obtained from instance p by removing the two smallest frequencies p_1 and p_2, replacing them by a single frequency $p_1 + p_2$, and reordering the frequencies as necessary. So p' has $n - 1$ frequencies.*

Define $\mathcal{T}'$ to be the set of possible trees for p'.

Given any $T \in \mathcal{T}$, let R be the tree obtained from T by replacing the leaves p_1, p_2, and their parent (a 3-node subtree) by a single new leaf with frequency $p'_i = p_1 + p_2$. Call the process of obtaining R from T in this way merging p_1 and p_2 in T.

Let $\mu : \mathcal{T} \to \mathcal{T}'$ be the function where $\mu(T) = R$, where R is obtained by merging p_1 and p_2 in T.

The merging process is invertible as follows. Given any tree $R \in \mathcal{T}'$, obtain tree T from R by finding the leaf with the frequency $p_1 + p_2$ in R, and replacing it by a new node having two new children: leaves p_1 and p_2. Call the process of obtaining T this way *splitting p'_i in R*. If we then merge p_1 and p_2 in T, we get R back. That is, $\mu(T) = R$. So $T = \mu^{-1}(R)$.

In short, μ is a bijection between the set $\mathcal{T}$ of trees for p (having p_1 and p_2 as siblings) and the set $\mathcal{T}'$ of trees for p'. So solutions for p (of the kind that we are after) correspond one-to-one with solutions for p'. But does the bijection preserve the *cost*? For each $T \in \mathcal{T}$, if $R = \mu(T)$, is it the case that $\mathsf{cost}(T) = \mathsf{cost}(R)$? By considering an example, we can see that it's not the case. However, the function just shifts the cost, by $p_1 + p_2$:

Observation 2. *For each tree $T \in \mathcal{T}$ and tree $R = \mu(T)$, $\mathsf{cost}(T) = \mathsf{cost}(R) + p_1 + p_2$.*

To see why the observation holds, let d be the depth of p_1 and p_2 in T. The leaves p_1 and p_2 in T contribute $d \cdot (p_1 + p_2)$ to $\mathsf{cost}(T)$, whereas the leaf p'_i in R contributes $(d - 1) \cdot (p_1 + p_2)$ to $\mathsf{cost}(R)$. (And otherwise T and R are identical.)

In short, the function μ (merging p_1 and p_2) is a bijection (a one-to-one mapping, a.k.a. correspondence) between $\mathcal{T}$ (the trees for p in which p_1 and p_2 are siblings) and $\mathcal{T}'$ (the trees for p'). Furthermore, the bijection preserves relative cost: given two trees T_1 and T_2 for p, it is the case that $\mathsf{cost}(T_1) \le \mathsf{cost}(T_2)$ if and only if $\mathsf{cost}(R_1) \le \mathsf{cost}(R_2)$, where $R_1 = \mu(T_2)$ and $R_2 = \mu(T_1)$.

It follows that a tree $T \in \mathcal{T}$ is optimal for p if and only if $\mu(T)$ is optimal for p'. So, to compute an optimal tree T, it *is* enough to compute an optimal tree R for p', and then take $T = \mu^{-1}(R)$. (That is, obtain T by splitting its leaf of frequency $p_1 + p_2$.) This is the gist of the next lemma.

Recall that p is an arbitrary instance with $n \ge 2$, and p' is the instance obtained from p by replacing p_1 and p_2 with $p_1 + p_2$.

Lemma 2 (optimal substructure)**.** *Let R be any optimal tree for p'. Let $T = \mu^{-1}(R)$ be obtained by splitting the leaf of frequency $p_1 + p_2$ in R. Then T is optimal for p.*

Proof (long form).

1. Consider any p, p', R, and $T = \mu^{-1}(R)$ as defined above. We need to show T is optimal for p.
2. Let T^* be an optimal tree for p in $\mathcal{T}$. (T^* exists by Lemma 1.)
3. Let $R^* = \mu(T^*)$ be obtained by contracting p_1 and p_2 in T^*.
4. By Observation 2, $\mathsf{cost}(T^*) = \mathsf{cost}(R^*) + p_1 + p_2$, and $\mathsf{cost}(T) = \mathsf{cost}(R) + p_1 + p_2$.
5. Tree R is optimal for p', so $\mathsf{cost}(R) \le \mathsf{cost}(R^*)$.
6. By Steps 4 and 5, $\mathsf{cost}(T) = \mathsf{cost}(R) + p_1 + p_2 \le \mathsf{cost}(R^*) + p_1 + p_2 = \mathsf{cost}(T^*)$.
7. That is, $\mathsf{cost}(T) \le \mathsf{cost}(T^*)$.
8. Since T^* has minimum cost for p, it follows that T does also. So T is optimal for p. $\square$

This gives us the following recursive algorithm:

```
Huffman(p = (p_1, p_2, ..., p_n)):          — recursive algorithm for Huffman Coding —
1. If $n = 1$: return the tree consisting of a single leaf $p_1$.
2. Let $p'$ be the instance obtained from $p$ by replacing $p_1$ and $p_2$ by $p_1 + p_2$ and reordering.
3. Let $R = \mathsf{Huffman}(p')$.                          (Recursively compute a tree for $p'$)
4. Obtain new tree $T = \mu^{-1}(R)$ for $p$ by splitting the leaf of frequency $p_1 + p_2$ in $R$
   (replacing leaf $p_1 + p_2$ by a new node with two new children, leaves $p_1$ and $p_2$).
5. Return $T$.
```

The correctness of the recursive algorithm follows from Lemma 2 by a simple induction.

Theorem 1. *The recursive algorithm is correct.*

Proof (long form).

1. We show by induction on $n = |p|$ that Huffman is correct for all instances p.
2. For any instance with $n = 1$, the one-node tree is optimal, so the algorithm is correct.

3.1. Consider any p with $n \geq 1$.

3.2.1. Assume that the algorithm is correct for all smaller instances.

3.2.2. Consider the execution of the algorithm on p.

3.2.3. Let p', $R = \mathsf{Huffman}(p')$, and $T = \mu^{-1}(R)$ be as computed in the algorithm.

3.2.4. By the inductive assumption, R is optimal for the smaller instance p'.

3.2.5. So, by Lemma 2, the solution $T = \mu^{-1}(R)$ returned by $\mathsf{Huffman}(p)$ is optimal for T.

3.3. By Block 3.2, if Huffman is correct for all smaller instances, then it is correct for p.

4. By Block 3, for any p with $n \geq 1$, if Huffman is correct for smaller instances, it is correct for p.
5. By Step 2, Huffman is correct for all instances of size 1.
6. So, inductively, Huffman is correct for all instances. □

We proved correctness of the recursive algorithm using the optimal-substructure property for the greedy choice. We showed how to *reduce* the problem of computing an optimal tree for p to the problem of computing an optimal tree for the smaller instance p'.

The recursive algorithm is "mostly" tail recursive. Try executing it on few examples to get intuition. We can unwind the tail recursion to get the following iterative algorithm:

```
iHuffman(p = (p_1, p_2, ..., p_n)):          — iterative algorithm for Huffman Coding —
1. Initialize $C \leftarrow (t_1, t_2, \ldots, t_n)$, where $t_i$ is a subtree consisting of a single leaf $p_i$.
2. For $t \leftarrow 1, 2, \ldots, n-1$:
3.1. Remove the two minimum-frequency subtrees $t_1$ and $t_2$ from $C$.
3.2. Construct a new subtree $T$, whose root is a new node with left and right subtrees $t_1$ and $t_2$.
     The frequency of $t$ is the sum of the frequencies of $t_1$ and $t_2$.
3.3. Insert $t$ into $C$, and reorder the trees in $C$ by increasing frequency.
4. Return the single tree $t_1$ in $C$.
```

The iterative algorithm maintains a collection $C = (t_1, t_2, \ldots, t_k)$ of subtrees. Initially, there are n subtrees: subtree t_i consists of a single leaf with frequency p_i. It maintains the *frequency* of each subtree, which is defined to be the sum of the frequencies of its leaves. It keeps the collection C ordered by increasing frequency. The algorithm repeats the following step: it removes the two minimum-frequency subtrees t_1 and t_2 from C, then replaces them by a single new subtree T, formed by a new root, having left subtree t_i and right subtree t_j. It stops after $n-1$ iterations, when the collection has just one tree, and returns that tree.

What is the greedy invariant? In this design pattern, we expect to prove that the iterative algorithm is correct by showing that it maintains the greedy invariant:

greedy invariant: *The current partial solution can be extended to a correct solution.*

But, in this context, what is "the current partial solution", and what does it mean to say it can be "extended" to some correct solution? Here's the right interpretation:

greedy invariant: *There is an optimal tree T that includes each subtree t_i in C.*

One can prove that (i) this invariant is maintained in each iteration, and (ii) after the final iteration (when there is just one tree t_1 in C) the invariant implies that t_1 is an optimal tree for p. The proof of (i) generalizes the proof of Lemma 1. We leave the proofs of (i) and (ii) as an exercise.

Making the recursive algorithm fully tail-recursive. To get the iterative algorithm from the recursive algorithm, we first modify the recursive algorithm to make it fully tail-recursive. The original version is not fully To do that, we make it precompute the subtree for each p_i "in advance", and pass these subtrees along with each recursive call. For example, in the top-level call, we know that in the final tree, the two frequencies p_1 and p_2 will be siblings in a 3-node subtree, say t', with total frequency $p_1 + p_2$. Instead of waiting until the end to build that 3-node subtree t' and insert it into the tree, we build t' in advance, and pass it down to the recursive call. When the recursive call builds its tree, it just uses t', instead of the single leaf with frequency $p_1 + p_2$.

Generally, at any point in the original algorithm, when it constructs some new frequency p_1+p_2 by summing two previously constructed frequencies, we already know what the subtrees for p_1 and p_2 are going to have to be in the end. So, we also know what the subtree for p_1+p_2 will have to be. Inductively, for each frequency that the algorithm constructs, we know, at the time the frequency is constructed what the subtree associated with that frequency is going to end up being. Instead of waiting to after the recursive all ends to construct it, we pass the subtree down into the recursive call, which takes care of it. The pseudo-code for the fully tail-recursive algorithm is below. Initially, the n subtrees $t_1, t_2, \ldots, t_n$ that we pass in are just singleton trees: t_i consists of just one leaf with frequency p_i.

```
mHuffman$(p = (p_1, p_2, \ldots, p_n), C = (t_1, t_2, \ldots, t_n))$:        — modified recursive algorithm —
1. Note: $t_i$ is the "pre-constructed" subtree for frequency $p_i$.
2. If $n = 1$: return the tree $t_1$
3. Let $p'$ be the instance obtained from $p$ by replacing $p_1$ and $p_2$ by $p_1 + p_2$ and reordering.
4. Construct new subtree $t$ with a new root and left and right subtrees $t_1$ and $t_2$.
5. Let $C'$ be the collection of subtrees obtained from $C$ by replacing $t_1$ and $t_2$ by $t$ ...
                                                   ... and ordering $C'$ to match $p'$.
6. Return mHuffman$(p', C')$.
```

Once we have the recursive algorithm in fully tail-recursive form above, we just unwind the tail recursion. This gives the iterative algorithm, iHuffman.