Some greedy algorithms you have seen before.

Huffman coding from CS 240

Given symbols (e.g. a, b, c, ..., z) each with a frequency (e.g. e frequent, q infrequent) encode them in binary so that

- Encodings are short - so use short string for e, longer for q
- Can decode - so use a prefix code - no code is a prefix of another.

**Example**

<table>
<thead>
<tr>
<th>freq.</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.45</td>
</tr>
<tr>
<td>B</td>
<td>.1</td>
</tr>
<tr>
<td>C</td>
<td>.25</td>
</tr>
<tr>
<td>D</td>
<td>.2</td>
</tr>
</tbody>
</table>

To decode:

\[
\begin{align*}
&0 \quad 1 \quad 1 \quad 1 \quad 0 \\
&A \quad C \quad D \quad A
\end{align*}
\]

Average code length:

\[
\frac{.35(1) + .1(3) + .25(2) + .2(3)}{\text{internal nodes}} = \frac{2}{2} = .35 + .55 + 1.0
\]

**Greedy Algorithm**

- Let x, y be two least frequent letters
- Construct \( x \overset{z}{\rightarrow} y \) new letter \( z \)
  \( \text{freq}(z) = \text{freq}(x) + \text{freq}(y) \)
- Remove x, y. Add z. Repeat
You saw a proof that this greedy algorithm gives the prefix code that minimizes average code length.

Idea of proof: an exchange proof that if \( x \) or \( y \) is not a child of the lowest internal node, we could swap and average code length does not go up.

Thus

\[
\text{any solution} \quad \rightarrow \quad \text{swap} \quad \rightarrow \quad \text{greedy solution}
\]

and avg. code length does not go up.

So greedy solution has smallest avg. code length.

Invented by David Huffman as PhD student at MIT. Also known for mathematics of origami.
Another example of greedy algorithm: optimal caching
Recall from CS 240

![Diagram showing large slow memory and small fast memory, with blocks indicated.]

When the cache is full and we want a new block, we must evict some block. Which one?

Goal: minimize the total number of evictions over the sequence of block requests. (The actual sequence is hidden)

E.g. requests 1, 2, 3, 1 with k = 2

<table>
<thead>
<tr>
<th>request</th>
<th>cache</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

```
versus some

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

total = 2
```

Block eviction strategies

- Least Recently Used (LRU) evict the block that was last accessed longest ago
- Least Frequently Used (LFU) evict the block that's been accessed least often
- First In, First Out (FIFO) just use a queue
In fact LRU is better. Better than LFU in a theoretical sense (competitive analysis). Better than FIFO in practice (though equally in comp. analysis).

To compare some strategy to optimum (with knowledge of the future) we need to compute the opt.

Use this greedy strategy:
- Evict the block that is next accessed furthest in the future.

Ex. Prove that this works - use exchange proof.
( It’s tricky.)

LRU pretends that the future looks like the past (reversed)
**Knapsack Problem**

You’re going on a 5 day canoeing trip to Algonquin Park. You want to pack your knapsack to maximize value and minimize weight.

Given n items, item i has weight $w_i$ and value $v_i$.

Weight limit of knapsack is $W$. Put items in knapsack, sum of weights $\leq W$, maximize sum of values.

**[Notation: find $S \subseteq \{1, \ldots, n\}$, $\sum w_i : i \in S \leq W$ and maximize $\sum v_i : i \in S$]**

Two versions of the problem:

- 0-1 knapsack. Items are indivisible (tent, flashlight)
- fractional knapsack. Can use fractions of items (oatmeal, cheese)

Question: Do you think it’s ever good to use fractions?

We’ll see a dynamic programming algorithm for 0-1 knapsack, but (in some sense) the alg. is not efficient and the problem is hard.

Today: a greedy algorithm for the fractional knapsack

**Example:**

<table>
<thead>
<tr>
<th>$i$</th>
<th>$v_i$</th>
<th>$w_i$</th>
<th>$v_i/w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>3</td>
<td>2 1/3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

$W = 6$

Note: it makes sense to order items by value per weight.
For the 0-1 case, greedy gives item 1, value 12 (nothing else fits)
but taking items 2 and 3 gives value 13.
For fractional case, greedy takes item 1, leaving weight
of 2 free, so take 2/3 of item 2. Value: 12 + \( \frac{2}{3} \cdot 7 \).

Greedy Algorithm

\[
x_i = \text{weight of item } i \text{ that we take} \\
\text{free-}W \leftarrow W \\
\text{for } i = 1 \ldots n \text{ (items ordered by } \frac{v_i}{w_i} \text{)} \\
x_i \leftarrow \min \left\{ \frac{x_i}{w_i}, \text{free-}W \right\} \\
\text{free-}W \leftarrow \text{free-}W - x_i
\]

Note that the solution will look like:

<table>
<thead>
<tr>
<th>Item</th>
<th>1</th>
<th>2</th>
<th>\ldots</th>
<th>j</th>
<th>j+1</th>
<th>\ldots</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_i )</td>
<td>( \frac{x_1}{w_1} )</td>
<td>( \frac{x_2}{w_2} )</td>
<td>\ldots</td>
<td>( \frac{x_j}{w_j} )</td>
<td>0</td>
<td>\ldots</td>
<td>0</td>
</tr>
</tbody>
</table>

\( \frac{x_1}{w_1}, \frac{x_2}{w_2}, \ldots, \frac{x_j}{w_j} \) \( \uparrow \) use none of
items \( j+1 \ldots n \)

\( \frac{x_1}{w_1}, \frac{x_2}{w_2}, \ldots, \frac{x_j}{w_j} \) \( \uparrow \) use fraction
of item \( j \) \( 0 < x_j < w_j \)

Final weight: \( \sum x_i = W \) (if \( \sum w_i \geq W \))

Final value: \( \sum \left( \frac{v_i}{w_i} \right) x_i \)

Running time \( O(n \log n) \) to sort by \( \frac{v_i}{w_i} \).
Claim: The greedy alg. gives the opt. soln. to the fractional knapsack problem.

Proof: greedy solution \( x_1, x_2, \ldots, x_{k-1}, x_k, \ldots, x_n \)
opt. solution \( y_1, y_2, \ldots, y_{k-1}, y_k, \ldots, y_n \)

Suppose \( y \) is an opt. soln. that matches \( x \) on max \# indices.
If \( x = y \) done. Let \( k \) = first index where \( x_k \neq y_k \).
Then \( x_k > y_k \) since greedy maximizes \( x_k \).
Since \( 2y = 2x = N \), there is a later index \( l > k \) with \( y_l > x_l \).
Exchange weight \( \Delta \) of item \( l \) for equal weight of item \( k \).

\[ y_k' = y_k + \Delta \]
\[ y_l' = y_l - \Delta \]

Choose \( \Delta \) so \( x_k = y_k' \) or \( x_l = y_l' \).

\[ \Delta = \min \left( \frac{y_l - x_l}{2}, \frac{x_k - y_k}{2} \right) \]

so \( \Delta > 0 \)

Amount we can move from \( l \) to \( k \).

Change in value

\[ \Delta \left( \frac{v_k}{w_k} \right) - \Delta \left( \frac{v_l}{w_l} \right) = \Delta \left( \frac{v_k}{w_k} - \frac{v_l}{w_l} \right) \]

This is non-neg. because \( \frac{v_k}{w_k} \geq \frac{v_l}{w_l} \) (we sorted this way).

But \( y \) was an opt. soln., so this can't be better.

Therefore it's a new opt. soln. that matches \( x \) on one more index. Contra to choice of \( y \).

We will see more greedy algos. for graph problems.