# Greedy Algorithms

## CS 624 — Analysis of Algorithms

April 2, 2024

# Greedy Algorithms

**Greedy algorithms**, like dynamic programming, are used to solve optimization problems.

- Problems exhibit optimal substructure, as in DP.
- Problems also exhibit the greedy-choice property:

  Instead of having to *search* over results of sub-problems, we have a criterion (a locally optimal choice) that lets us *predict* the choice that leads to a globally optimal solution.

# Character Encoding

Goal: Encode a text message as a bit string.

The message is 100,000 characters, with only the letters $\{a, b, c, d, e, f\}$.
The frequency of each character is given by the following table:

| character | times used |
|---|---|
| $a$ | 45,000 |
| $b$ | 13,000 |
| $c$ | 12,000 |
| $d$ | 16,000 |
| $e$ | 9,000 |
| $f$ | 5,000 |

# Fixed-Length Encoding

An example fixed-length encoding:

| character | code |
|---|---|
| $a$ | 000 |
| $b$ | 001 |
| $c$ | 010 |
| $d$ | 011 |
| $e$ | 100 |
| $f$ | 101 |

We need three bits for each character, so the entire message will take 300,000 bits to encode. Can we do better?

# Variable Length Code

Idea: use a *variable-length* encoding, where *more frequent characters* are given *shorter codes*.

| character | times used |
|---|---|
| $a$ | 45,000 |
| $b$ | 13,000 |
| $c$ | 12,000 |
| $d$ | 16,000 |
| $e$ | 9,000 |
| $f$ | 5,000 |

For example "$a$" should have a shorter code than "$f$".

# Prefix Codes

**Definition (Prefix Code)**

A **prefix code** (aka **prefix-free code**) is a mapping from an alphabet to codes (typically, bit strings), such that no code is a prefix of another code.

This property allows *variable-length* codes to be uniquely parsed.

# Prefix Codes

For example:

| character | frequency | code |
|---|---|---|
| $a$ | .45 | 0 |
| $b$ | .13 | 101 |
| $c$ | .12 | 100 |
| $d$ | .16 | 111 |
| $e$ | .09 | 1101 |
| $f$ | .05 | 1100 |

The total size of the encoded message is now

$$\begin{aligned}(1(.45) + 3(.13) + 3(.12) + 3(.16) + 4(.09) + 4(.05)) \cdot 100,000 \text{ bits} \\ = 224,000 \text{ bits}\end{aligned}$$

which is a significant improvement, even though some of the code words are actually longer in this encoding.

# Prefix Codes

If we treat the frequency as the relative number of times a character appears in the code, then we can re-write the former equation as:

$$1(.45) + 3(.13) + 3(.12) + 3(.16) + 4(.09) + 4(.05) = 2.24$$

This is the expected number (or “average” number) of bits per character, as opposed to 3 bits per character in our fixed-length encoding.

# Prefix Codes

The **efficiency** of a code is the expected number of bits per character (given a distribution of characters).

- Let $C$ be the set of characters.
- Let $f(x)$ be the frequency of the character $x \in C$.
  Assume that $\sum_{x \in C} f(x) = 1$.
- Let $\mathit{length}(x)$ be the length of the code word for $x \in C$.

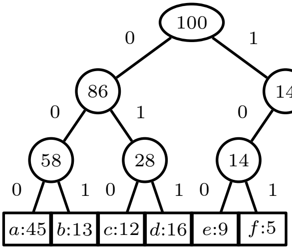
Then the average number of bits per character for this encoding is

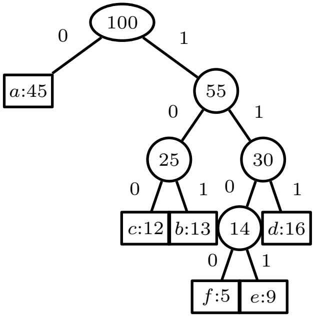
$$\sum_{x \in C} f(x) \cdot \mathit{length}(x)$$

Our problem is this: Given the set $C$ and the frequency function $f$, find a prefix code that minimizes this value.

# Codes as Binary Trees

Codes can be represented by binary trees.




Left: fixed code, right: variable code.

# Codes as Binary Trees

- The depth of a leaf in the tree is just the length of the code word for that character.
- Let $d_T(x)$ be the depth of a leaf node corresponding to the character $x$ in the tree $T$.
- The average cost AC per character in the encoding scheme defined by the tree $T$ is

$$AC(T) = \sum_{x \in C} f(x) d_T(x)$$

# Exhaustive Search

**Strategy #1: Exhaustive search**

- Enumerate all possible prefix trees and find the one with the smallest average cost per character.
- Without performing an exact analysis, the cost of this algorithm would be exponential in the number of characters, and therefore completely useless.

# Optimal Substructure

## Lemma

*If $T$ is the tree corresponding to an optimal prefix encoding, and if $T_L$ and $T_R$ are its left and right subtrees, respectively, then $T_L$ and $T_R$ are also trees corresponding to optimal prefix encodings for the alphabets they cover.*

## Proof.

- Let us say that $C_L$ is the set of characters that are leaf nodes in $T_L$ and similarly for $C_R$ and $T_R$.
- If $x \in C_L$, then certainly $d_{T_L}(x) = d_T(x) - 1$, and the same is true for $C_R$ and $T_R$.

# Optimal Substructure

## Proof (cont.)

- Therefore we can see from our basic cost formula that

$$\begin{aligned} AC(T) &= \sum_{x \in C} f(x) d_T(x) \\ &= \sum_{x \in C_L} f(x)\big(d_{T_L}(x) + 1\big) + \sum_{x \in C_R} f(x)\big(d_{T_R}(x) + 1\big) \\ &= \sum_{x \in C_L} f(x) d_{T_L}(x) + \sum_{x \in C_R} f(x) d_{T_R}(x) + \sum_{x \in C} f(x) \end{aligned}$$

- If $T_R$ were not an optimal encoding tree, then we could replace it by a more efficient one (with the same leaves and the same frequencies), and this would show in turn that $T$ could not have been optimal, a contradiction.

# Optimal Substructure

## Corollary

*If $T$ is the tree corresponding to an optimal prefix encoding, then every subtree of $T$ also corresponds to an optimal prefix encoding.*

## Proof.

This follows immediately by induction. $\square$

Since this problem has the optimal substructure property, we could use dynamic programming to solve it recursively.

# Recursive (Top-Down) Algorithm

**Strategy #2: Recursive algorithm**

- For a given alphabet of characters $C$ where $|C| > 1$, choose a partition of $C$ into two non-empty sets $C_L$ and $C_R$.
- Solve the sub-problems corresponding to $C_L$ and $C_R$ recursively, and form a binary tree from the results.
- Minimize over $AC(T)$ for every candidate $T$.
- There are overlapping subproblems when we hit the same subset of $C$ along different paths.

  A subproblem is identified by a subset of the original $C$.

Criticism:

- If $|C| = n$, then there are $2^n - 2$ candidate splits to choose from!

# Worklist (Bottom-Up) Algorithm

**Strategy #3: Worklist algorithm**

- Start with a worklist consisting of $n$ trees, each tree consisting of exactly 1 character.
- From these trees construct other trees bottom-up and add them to the worklist.
- As each new tree is constructed, check the worklist to see if a tree with the same leaves is in it.
- Keep the tree with the smallest cost in the worklist and remove any others with the same set of leaves.
- At the end of this process there will be one tree in the worklist that contains all the characters in $C$ as leaves, and that tree represents an optimal encoding.
- This algorithm will definitely give the correct answer, but is still inefficient, although it is better than exhaustive search.

# Finding the Optimal Encoding

- The optimal substructure property should remind us of dynamic programming.
- If there were also an *overlapping subproblems* property of this problem, we could try such a solution.
- Actually we have something even better: We don't actually have to form all possible trees on the way up and check them all.
- We actually can tell at each step exactly which tree to form.

# Greedy Choice Property

**Lemma (Greedy Choice Property)**

*Let $x$ and $y$ be two characters in $C$ having the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differ only in the last bit.*

**Proof.**

- Suppose that the tree $T$ represents an optimal prefix code for our problem.
- If $x$ and $y$ are sibling nodes of greatest depth, then we are done.
- Otherwise, suppose that $p$ and $q$ are sibling nodes of greatest depth.
- We will exchange $x$ and $p$, and we will also exchange $y$ and $q$.

# Finding the Optimal Encoding

## Proof (cont.)

- We know that

$$
\begin{aligned}
d_T(x) &\le d_T(p)\\
d_T(y) &\le d_T(q)\\
f(x) &\le f(p)\\
f(y) &\le f(q)
\end{aligned}
$$

- Suppose the tree $T$, after these two switches, is turned into the tree $T'$. Then we have:

$$
\begin{aligned}
d_{T'}(x) &= d_T(p)\\
d_{T'}(p) &= d_T(x)\\
d_{T'}(y) &= d_T(q)\\
d_{T'}(q) &= d_T(y)
\end{aligned}
$$

# Finding the Optimal Encoding

## Proof (cont.)

$$
\begin{aligned}
AC(T') - AC(T) &= \sum_{z \in C} f(z)\big(d_{T'}(z) - d_T(z)\big) \\
&= f(p)\big(d_{T'}(p) - d_T(p)\big) + f(x)\big(d_{T'}(x) - d_T(x)\big) \\
&\quad + f(q)\big(d_{T'}(q) - d_T(q)\big) + f(y)\big(d_{T'}(y) - d_T(y)\big) \\
&= f(p)\big(d_T(x) - d_T(p)\big) + f(x)\big(d_T(p) - d_T(x)\big) \\
&\quad + f(q)\big(d_T(y) - d_T(q)\big) + f(y)\big(d_T(q) - d_T(y)\big) \\
&= \big(f(p) - f(x)\big)\big(d_T(x) - d_T(p)\big) \\
&\quad + \big(f(q) - f(y)\big)\big(d_T(y) - d_T(q)\big) \\
&\leq 0
\end{aligned}
$$

so $AC(T') \leq AC(T)$, which shows that $T$ was not an optimal tree to begin with, and this is a contradiction. $\square$

# Finding the Optimal Encoding – Huffman's Algorithm

- We can start out with our initial worklist, and we can take two nodes of smallest frequency and build a tree from them (in which they are the two leaves).
- Then we delete those two nodes from the worklist, because we know that they will definitely be part of the little tree we have just constructed – we will never have to look at them again.
- By exactly the same argument, we can take the two elements of the worklist that are now of smallest cost, and build a little tree from them, and then throw them away.
- When we are done, we have the tree we are looking for.
- The algorithm: We keep a minimum-priority queue $Q$ of subtrees. $Q$ initially consists of the $n$ characters. The priority of any element in $Q$ will be the cost of that subtree.

# Huffman's Algorithm

**Algorithm 1** Huffman(C)

$n \leftarrow |C|$
$Q \leftarrow C$
**for** $i \leftarrow 1$ **to** $n - 1$ **do**
    allocate a new node z
    $left[z] \leftarrow ExtractMin(Q)$
    $right[z] \leftarrow ExtractMin(Q)$
    $f[z] \leftarrow f[x] + f[y]$
    $Insert(Q.z)$
**end for**
**return** $ExtractMin(Q)$    //Return the root of the tree.

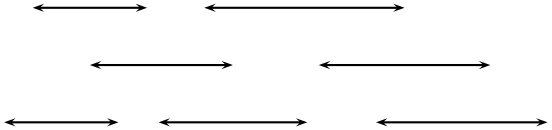
# Finding the Optimal Encoding – Huffman's Algorithm

- This algorithm works even better than a dynamic programming algorithm: we don't have to memoize intermediate results for later use.
- We know exactly at each step what we need to do.
- This is called a "greedy" algorithm because we chose the locally best solution at each step.
- In effect, we act as if we were "greedy".
- What is is the best at each step is guaranteed (in this case) to turn to out to be the best overall.

# Activity Selection

- **Input:** Set S of n activities – $\{a_1, a_2, \ldots, a_n\}$.
- $s_i$ = start time of activity i.
- $f_i$ = finish time of activity i.
- **Output:** Subset A of maximum number of compatible activities.
- Two activities are compatible, if their intervals do not overlap.

Example (activities in each line are compatible):



# Optimal Substructure

- Assume activities are sorted by finishing times – $f_1 \leq f_2 \leq \cdots \leq f_n$.
- Suppose an optimal solution includes activity $a_k$.
- This generates two subproblems:
  - Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before $a_k$ starts (compatible with $a_k$).
  - Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after $a_k$ finishes.
- The solutions to the two subproblems must be optimal.
- Prove using the cut-and-paste approach.

# Optimal Substructure

- Let $S_{ij}$ = subset of activities in S that start after $a_i$ finishes and finish before $a_j$ starts.
- Subproblems: Selecting maximum number of mutually compatible activities from $S_{ij}$.
- Let c[i,j] = size of maximum-size subset of mutually compatible activities in $S_{ij}$.
- The recursive solution is:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{i<k<j}\{c[i,k] + c[k,j] + 1\} & \text{otherwise} \end{cases}$$

# Greedy Choice Property

- The problem also exhibits the greedy-choice property.
- There is an optimal solution to the subproblem $S_{ij}$, that includes the activity with the smallest finish time in set $S_{ij}$.
- It can be proved easily (how?).
- Hence, there is an optimal solution to S that includes $a_1$.
- Therefore, make this greedy choice without solving subproblems first and evaluating them.
- Solve the subproblem that ensues as a result of making this greedy choice.
- Combine the greedy choice and the solution to the subproblem.

# Recursive Solution

**Algorithm 2** Recursive-Activity-Selector (s, f, i, j)

$m \leftarrow i + 1$
**while** $m < j$ and $s_m < f_i$ **do**
    $m \leftarrow m + 1$
**end while**
**if** $m < j$ **then**
    **return** $a_m \cup Recursive - Activity - Selector(s, f, m, j)$
**else**
    **return** $\emptyset$
**end if**

- Top level call: $Recursive - Activity - Selector(s, f, 0, n+1)$
- Complexity??
- See text for iterative version

# Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there is always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and solve top-down.
- May have to preprocess input to put it into greedy order.
- Example: Sorting activities by finish time.