
Observations.

1. Problem exhibits the optimal substructure property that characterizes dynamic programming algorithms.

2. However, one of the competitors for the optimal subproblem seems to have an advantage, which we will come to call the greedy property.
The activity selection problem.

Given a set of activities $S = \{a_1, \ldots, a_n\}$, each with a specified start time, $s_i$, and finish time $f_i$ such that $[s_i, f_i) \neq \phi$, we wish to schedule as many of these activities in the same venue.

That is, all activities are competing for the same machine, or the same classroom, or the same playing field.

Definition: In this context, a subset $A \subset S$ is **compatible** if

$$(i \neq j) \& (a_i, a_j \in A) \Rightarrow [s_i, f_i) \cap [s_j, f_j) = \phi.$$
We note that the search space is of exponential size, as there are $2^n$ subsets of a set with $n$ members.

But, we can discern an optimal substructure and find a DP solution.

First, sort the activities in order of increasing $f_i$ — as in the example.

Consider the candidates for scheduling between $a_i$ and $a_j$:

$$S_{ij} = \{a_k : f_i \leq s_k < f_k \leq s_j\}.$$  

Add two fictitious task $a_0 : [-1,0)$ and $a_{n+1} : [\infty, \infty)$, such that $S_{0,n+1} = \{a_1, a_2, \ldots, a_n\}$ corresponds to the overall problem.

For $i \geq j$, the left sketch above shows that $S_{ij} = \emptyset$, since $[f_i, s_j) = \emptyset$.

For $i < j$, the right sketch shows an optimal $A_{ij} \subseteq S_{ij}$. For any $a_k \in A_{ij}$, the activities in $A_{ij}$ to occurring after $f_i$ but before $s_k$ comprise set $B_1$. Similarly, $B_2$ contains activities in $A_{ij}$ occurring after $f_k$ and before $s_j$.

So, $B_1 \subseteq S_{ik}$ and $B_2 \subseteq S_{kj}$. Since $A_{ij}$ is presumed optimal for $S_{ij}$, a cut-and-paste argument shows that $B_1$ is optimal for $S_{ik}$ and $B_2$ is optimal for $S_{kj}$.

We conclude that the optimal $A_{ij} \subseteq S_{ij}$ is

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}, \text{ for some } i < k < j \text{ with } a_k \in S_{ij}.$$  

Note $i < k$ puts $f_i < f_k$ and $k < j$ puts $f_k < f_j$, so $f_i < f_k < f_j$, but this condition does not guarantee that $a_k \in S_{ij}$. This latter constraint needs $f_i \leq s_k < f_k \leq s_j$, which could still be violated even if $f_i < f_k < f_j$. For example, we could have $s_k < f_i$ or $f_k > s_j$.  

3
With the objective function, \( c[i, j] \), being the size of the chosen set, the recursive formula is

\[
c[i, j] = \begin{cases} 
\max\{c[i, k] + 1 + c[k, j] : i < k < j \text{ and } f_i \leq s_k < f_k \leq s_j\}, & S_{ij} \neq \emptyset \\
0, & S_{ij} = \emptyset.
\end{cases}
\]

function \( c(i, j) \) {
    \( r = 0; \)
    for \( k = i + 1 \) to \( j - 1 \) {
        if \( s_k \geq f_i \) and \( f_k \leq s_j \) {
            \( t = c(i, k) + c(k, j) + 1; \)
            if \( t > r \)
                \( r = t; \)
        }
    }
}

Suppose all tasks are compatible with one another. That is, there are no overlapping \([s_i, f_i]\) time intervals in the entire problem. In this case, the if-statement in the recursive algorithm is always executed, and \( c(i, j) \) makes \( 2[(j - 1) - (i + 1) + 1] = 2(j - i - 1) \) recursive calls while solving a problem having \( j - i - 1 \) activities.

Let \( T(k) \) be the total number of recursive calls for a subproblem with \( k \) activities, all compatible.

\[
T(n) = \sum_{k=0}^{n-1} 2T(k) \geq 2T(n - 1) \geq 2^2T(n - 2) \geq 2^3T(n - 3) \geq \ldots \geq 2^{n-1}T(1) = 2^{n-1}(2) = 2^n.
\]

That is, \( T(n) = \Omega(2^n) \). This recursive solution is worst-case exponential.
Fill DP matrix by rising diagonals, starting with the diagonal just above the main diagonal. The starting diagonal contains zeros, as it corresponds to problems containing a single activity. A cell $c_{ij}$ is filled by maximizing over the competitors (pairs joined by an arrow) for which the common endpoint represents an activity that is actually in the subproblem. That is, $c_{ik}$ pairs with $c_{kj}$ only when activity $a_k \in S_{ij}$. That is, the pair competes for the maximum if $s_k \geq f_i$ and $f_k \leq s_j$.

The following DP program initializes the main diagonal. The actual starting diagonal, one above the main diagonal, is then computed with the same logic that extend the other diagonals.

```plaintext
function c(s, f) {
    for i = 0 to n + 1
        x(i, i) = 0;
    for l = 1 to n + 1
        for i = 0 to n + 1 - l {
            j = i + l;
            x(i, j) = 0;
            y(i, j) = 0;
            for k = i + 1 to j - 1 // no executions when j = i + 1
                if $s_k \geq f_i$ and $f_k \leq s_j$ { // check that $a_k \in S_{ij}$
                    t = x(i, k) + x(k, j) + 1;
                    if t > x(i, j) {
                        x(i, j) = t;
                        y(i, j) = k;
                    }
                }
        }
    return x, y;
}
```
\( T(n) = \Theta(n^3) \). Why? \( O(n^2/2) \) cells to fill each involving at most \( n \) competing pairs. Each competing pair requires \( \Theta(1) \) time to evaluate. This implies \( T(n) = O(n^3) \).

The upper sub-triangle has \( O((n/2)^2) \) cells to fill, each involving at least \( n/2 \) competing pairs. This implies \( T(n) = \Omega(n^3) \).

At conclusion \( x(0, n+1) \) contains the size of the largest compatible subset. We can recover the optimal subset from the \( y(i, j) \) matrix.

\[
\text{OptimalSet}(y, i, j) \{ \\
\text{if } (y(i, j) == 0) \text{ return } \phi; \\
\quad k = y(i, j); \\
\quad \text{return OptimalSet}(y, i, k) \cup \text{OptimalSet}(y, k, j) \cup \{k\}; 
\}
\]

Greedy approaches to start building an optimal set.

1. Choose the “thinnest” activity. That is choose \( a_m \) as a first step for problem \( S_{ij} \), where

\[
m = \arg\min\{f_k - s_k : a_k \in S_{i,j}\}.
\]

\[
\begin{array}{c c c c}
    a_i & & & a_j \\
    \hline
    & & & \\
\end{array}
\]

2. Choose the activity closest to one end of the gap. Recall that we have $f_0 \leq f_1 \leq \ldots \leq f_{n+1}$, and therefore $a_k \in S_{ij}$ if and only if $i < k < j$ and $s_k \geq f_i, f_k \leq s_j$. Hence, for subproblem $S_{ij}$, we are choosing $a_m$ with

$$m = \text{argmin}\{f_k : a_k \in S_{ij}\}.$$ 

That is, we are choosing the task that terminates earliest among the candidates in $S_{ij}$.

Added advantage: only one subproblem remains.

Does it work? Consider an optimal solution that does not include our greedy choice.

Modify solution by ejecting $a_k$, the leftmost task, and replacing it with $a_m$, the greedy choice. Modified solution has the same number of tasks as the optimal solution, and therefore it must be optimal as well. And, the modified solution contains the greedy choice.

**Greedy choice property:** An optimal solution that does not include the greedy choice can be modified to a solution that does include the greedy choice without degrading the objective function.
RecursiveActivitySelector(s, f, i, n) {  // initiate with (s, f, 0, n)
    // find optimal subset for S_{i,n+1}
    m = i + 1;
    while (m ≤ n) and (s_m < f_i))
        m = m + 1;
    if (m > n)
        return φ;
    return \{a_m\} ∪ RecursiveActivitySelector(s, f, m, n);
}

Observations.

1. m = 0 → m − 1 → m = 2 → ... → m = n + 1 monotonically, regardless of recursion depth.
2. constant bound on instruction count while m is at any one of these constant levels.
3. T(n) = Θ(n), assuming that f_0 ≤ f_1 ≤ ... ≤ f_n < f_{n+1} is already sorted; otherwise T(n) = Θ(n lg n).
4. Tail recursion can be converted to an iterative algorithm:

    GreedyActivitySelector(s, f) {
        n = length(s);
        A = \{a_1\};
        i = 1;
        for m = 2 to n
            if s_m ≥ f_i {  // test for a_m ∈ S_{i,n+1}; f_m ≤ ∞ is moot
                A = A ∪ \{a_m\};
                i = m;
            }
        return A;
    }

5. The iterative algorithm is clearly T(n) = Ω(n).
Greedy vs. Dynamic Programming approaches.

1. Both attempt to maximize or minimize some objective function over a typically exponential search space.

2. Both need the optimal substructure property.

3. Optimal solution involves optimal solutions of related \textbf{independent} subproblems.

4. For DP, the optimal solution arises from a competition over the related subproblems.

5. For Greedy, the optimal solution contains a greedy choice, together with its associated single subproblem, for which the greedy property continues to hold.

Examples.

\textbf{Dynamic Programming} \hspace{2cm} \textbf{Greedy}

1. Rod-cutting \hspace{2cm} 1. Activity Selection

2. Matrix Chain Multiplication \hspace{2cm} 2. Fractional Knapsack

3. Polygon Triangulation \hspace{2cm} 3. Huffman Codes

4. CYK Parsing
Fractional Knapsack: Items \(a_1, \ldots, a_n\) have weights \(w_1, \ldots, w_n\) and values \(v_1, \ldots, v_n\). We assume they are sorted such that \(v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n\). The knapsack was weight capacity \(W\).

Problem: Find weights \(x_1, \ldots, x_n\) under constraints \(\sum_{k=1}^{n} x_i \leq W\) and \(0 \leq x_i \leq w_i\) for \(1 \leq i \leq n\), such that \(V = \sum_{k=1}^{n} (x_i/w_i)v_i\) is maximal.

We can assume that all weights, (item weights \(w_i\), chosen weights \(x_i\), and knapsack capacity \(W\) are integer multiples of some quantum \(\delta > 0\). This assumption is equivalent to assuming that the \(w_i, x_i,\) and \(W\) are all nonnegative integers.
Optimal substructure:

Suppose \((x_1, x_2, \ldots, x_n)\) is an optimal solution. Then, for any chosen \(x_k > 0\), we conclude that \((x_1, x_2, \ldots, \hat{x}_k, \ldots, x_n)\) is an optimal for the subproblem with item weights \((w_1, \ldots, w_{k-1}, w_{k+1}, \ldots, w_n)\) and knapsack capacity \(W - x_k\). This follows from a cut-and-paste argument.
Greedy property:

Consider the greedy choice:

\[ x_1 = \begin{cases} 
W, & w_1 \geq W \\
W - w_1, & w_1 < W.
\end{cases} \]

Suppose \((y_1, \ldots, y_n)\) is an optimal solution that does not include the greedy choice. That is, \(y_1 \neq x_1\). There are a number of subcases.

Case 1: If \(x_1 = W\), then \(y_1 \neq x_1\) forces \(y_1 < W\). Why? because \(y_1 = W\) is not possible, given that \(x_1 = W\) and \(y_1 \neq x_1\). Also, \(y_1 > W\) exceeds the knapsack weight limit. Consequently, we must have \(y_1 < W\).

Moreover, \(x_1 = W\) implies \(w_1 \geq W\), so we can replace \(y_1\) with \(W\), thereby including the greedy choice, and decrease each of \(y_2, \ldots, y_n\) to zero. We let \(V\) denote the value of solution \((y_1, \ldots, y_n)\), and we let \(V'\) denote the value of the modified solution \((W, 0, \ldots, 0)\). We have

\[
\sum_{i=1}^{n} y_i \leq W
\]

\[
W - y_1 \geq \sum_{i=2}^{n} y_i
\]

\[
V' = \left( \frac{v_1}{w_1} \right) W = \left( \frac{v_1}{w_1} \right) y_1 + \left( \frac{v_1}{w_1} \right) (W - y_1) \geq \left( \frac{v_1}{w_1} \right) y_1 + \left( \frac{v_1}{w_1} \right) \sum_{i=2}^{n} y_i
\]

\[
= \sum_{i=1}^{n} \left( \frac{v_1}{w_1} \right) y_i \geq \sum_{i=1}^{n} \left( \frac{v_i}{w_i} \right) y_i = V.
\]

The last inequality follows because all of the \(v_i/w_i\) are less than or equal to \(v_1/w_1\). We have shown that the modified optimal solution has increased in value, or at least remained the same as the solution \((y_1, \ldots, y_n)\). Because the latter was optimal, it is not possible that the modified solution increased. Therefore, we must have \(V' = V\). That is, \((W, 0, \ldots, 0)\) is also optimal, and it contains the greedy choice.
Case 2: If \( x_1 \neq W \), then \( x_1 = w_1 < W \). In this case, \( y_1 \neq x_1 \) forces \( y_1 < w_1 \). This follows because \( y_1 = w_1 \) is not possible since \( x_1 = w_1 \) and \( y_i \neq x_1 \). Also, since the optimal solution cannot choose more of item 1 than exists, which is \( w_1 \), we must have \( y_1 < w_1 \).

Now, if \( \sum_{i=1}^{n} y_i < W \), we can increase \( y_1 \) some small \( 0 < \epsilon < w_1 - y_1 \), which will increase the value of the payload but will still keep the payload within the knapsack limit \( W \). As an increase to the optimal payload is not possible, we conclude that

\[
\sum_{i=1}^{n} y_i = \sum_{i=2}^{n} y_i = W - y_1 > w_1 - y_1.
\]

That is, \( \sum_{i=2}^{n} y_i \) is heavy enough to allow compensating decreases in \( (y_2, \ldots, y_n) \) to offset an increase of \( w_1 - y_1 \) in \( y_1 \). Specifically, let \( z_k \) be the weight reduction applied to item \( k \). Then,

\[
z_i \leq y_i, \text{ for } 2 \leq i \leq n
\]

\[
\sum_{i=2}^{n} z_i = w_1 - y_1.
\]

Let \( V \) be the value of solution \( (y_1, \ldots, y_n) \), and let \( V' \) be the value of the modified solution \( (w_1, y_2 - z_2, \ldots, y_n - z_n) \).

\[
V' = \left( \frac{v_1}{w_1} \right) w_1 + \sum_{i=2}^{n} \left( \frac{v_i}{w_i} \right) (y_i - z_i) = \left( \frac{v_1}{w_1} \right) w_1 + \sum_{i=2}^{n} \left( \frac{v_i}{w_i} \right) y_i - \sum_{i=2}^{n} \left( \frac{v_i}{w_i} \right) z_i
\]

\[
\geq \left( \frac{v_1}{w_1} \right) w_1 + \sum_{i=2}^{n} \left( \frac{v_i}{w_i} \right) y_i - \sum_{i=2}^{n} \left( \frac{v_1}{w_1} \right) z_i
\]

\[
= \left( \frac{v_1}{w_1} \right) w_1 + \sum_{i=2}^{n} \left( \frac{v_i}{w_i} \right) y_i - \left( \frac{v_1}{w_1} \right) \sum_{i=2}^{n} z_i
\]

\[
= \left( \frac{v_1}{w_1} \right) w_1 + \sum_{i=2}^{n} \left( \frac{v_i}{w_i} \right) y_i - \left( \frac{v_1}{w_1} \right) (w_1 - y_1) = \sum_{i=1}^{n} \left( \frac{v_i}{w_i} \right) y_i = V
\]

Since the modification cannot exceed the presumed optimal solution, we conclude that \( V' = V \). That, the modification, which now includes the greedy choice, is an optimal solution as well.

13
These properties, optimal substructure plus greedy property, imply that a greedy algorithm will generate an optimal solution. In this case, assuming that the input is sorted in decreasing density order, the following algorithm suffices.

FractionalKnapsack\( (w, v, W) \) \{ // assume integer inputs
\[
\begin{align*}
n &= \text{length}(w); \\
wgt &= 0; \\
\text{for } k = 1 \text{ to } n & \{ \\
x[k] &= 0; \\
k &= 1; \\
\text{while (wgt < } W \text{ and } k \leq n) & \{ \\
\quad \text{if } w[k] \leq W - wgt & \{ \\
\quad \quad x[k] &= w_k; \\
\quad \quad \text{else} & \{ \\
\quad \quad \quad x[k] &= W - wgt; \\
\quad \quad \quad wgt &= wgt + x[k]; \\
\quad \quad \quad k &= k + 1; \\
\quad \}\} \\
\} \text{ return } x;
\end{align*}
\]
\}
\]

The time complexity is clearly \( T(n) = \Theta(n) \).
Discrete (0/1) Knapsack: Indivisible items $a_1, \ldots, a_n$ have weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$. The knapsack was weight capacity $W$.

Problem: Transfer items to the knapsack by specifying $x_i = 0$ or 1. $x_i = 1$ means that $a_i$ is transferred to the knapsack. $x_i = 0$ means that item $a_i$ is not transferred. Find the vector $x_1, \ldots, x_n$, under constraints $\sum_{k=1}^{n} x_i w_i \leq W$, such that $V = \sum_{k=1}^{n} x_i v_i$ is maximal.

A greedy choice involves first taking the object of greatest value per unit weight that will fit in the knapsack’s remaining space.

Failure of the greedy choice. Assume the knapsack capacity is $W = 50$ pounds.

<table>
<thead>
<tr>
<th>item</th>
<th>weight</th>
<th>value</th>
<th>value/pound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>60</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>120</td>
<td>4</td>
</tr>
</tbody>
</table>

The greedy solution is $(x_1 = 1, x_2 = 1, x_3 = 0)$, yielding $V = 160$. The optimal choice is $(x_1 = 0, x_2 = 1, x_3 = 1)$, yielding $V = 220$. Note that the optimal solution cannot be transformed into a solution involving the greedy choice that does not reduce $V$. Specifically, the other eight solutions violate the knapsack weight constraint or reduce $V$.

Observations.

1. If these items were divisible, we could employ the greedy solution: weights $(x_1 = 10, x_2 = 20, x_3 = 20)$ give total value $60 + 100 + 80 = 240$.

2. Dynamic programming still applies to the discrete knapsack (see below).
Let

\[ m[i, j] = \text{largest value from items } 1 \ldots i \text{ for which the total weight } \leq j \]
\[ m[1, j] = \begin{cases} 0, & w_1 > j \\ v_1, & w_1 \leq j \end{cases} \]

Recursion:

\[ m[i + 1, j] = \begin{cases} m[i, j], & w_{i+1} > j \\ \max\{m[i, j], m[i, j - w_{i+1}] + v_{i+1}\}, & w_{i+1} \leq j. \end{cases} \]

<table>
<thead>
<tr>
<th>item</th>
<th>weight</th>
<th>value</th>
<th>value/pound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>60</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>120</td>
<td>4</td>
</tr>
</tbody>
</table>

To recover solution, item \( a_i \) is added when a diagonal arrow leaves a cell in row \( i \) on the path from cell \( (n, W) \). In the example, \((3, 50) \rightarrow (2, 30)\), implying items 3 and 2 were added to the knapsack.

The time complexity is \( T(n) = \Theta(nW) \).
Memoized version skips cells. However, input in which knapsack can hold all items, each of unit weight, forces the entire lower triangle of cells to be filled.
Huffman Codes.

Let $C = \{c_1, c_2, \ldots, c_n\}$ be a finite alphabet, in which the characters appear with relative frequencies $(f_1, f_2, \ldots, f_n)$ respectively. For example, we may characters $(a, b, c, d, e, f)$ with frequencies $(0.45, 0.13, 0.12, 0.16, 0.09, 0.05)$. The frequency represents the probability of occurrence of a given character, and consequently the frequencies sum to one. Let $K = \{0, 1, 00, 01, 10, 11, 100, \ldots\}$ be the collection of bit strings.

Definitions and consequences.

1. A **binary character code** $f$ for $C$ is an injective mapping $f : C \rightarrow K$.

2. A **prefix code** for $C$ is a binary character code for $C$ such that $c_i \neq c_j \Rightarrow f(c_i)$ is not a prefix of $f(c_j)$. That is, no character code is a prefix of another character code.

3. The prefix property means that a tree representation of the code has complete codes only at the leaf nodes. For example, the following **fixed-length code** assigns a three-bit string to each of six characters. Hence the average bits per character is $B(T) = 3.0$.
4. However an internal code without a full set of two children implies that some improvement is possible. The following tree has average bits per character

\[ B(T) = 3(0.45 + 0.13 + 0.12 + 0.16) + 2(0.09 + 0.05) = 2.86. \]

5. A **minimal prefix code** is a binary prefix code for which \( B(T) \), the average bits per character, is minimal.

6. A **full binary tree** is a binary tree in which each internal node has a full set of two children. The example above shows that a minimal prefix code will always have a full binary tree representation.
7. We can create a full binary tree from its leaves as a series of mergers. We first merge two leaves as the leaves of a new node, which we can identify as a **meta-character** corresponding to the occurrence of either of the leaf characters. For a meta-character \( \alpha \), arising from the merger of two leaves \( a \) and \( b \), we define

\[
f(\alpha) = f(a) + f(b).
\]

The meta-character and the remaining leaves are now the field for subsequent mergers.

8. Merging characters, and evolving meta-characters, reduces the alphabet size by one at each step. The first step has \( \binom{n}{2} \) choices to merge. The next step has \( \binom{n-1}{2} \) choices, and so forth. Consequently, the number of full binary trees that we can create in this manner is

\[
T(n) = \binom{n}{2} \binom{n-1}{2} \cdots \binom{2}{2} \geq \left( \frac{n/2}{2} \right)^{n/2} = \left[ \frac{1}{2} \left( \frac{n}{2} \right) \left( \frac{n}{2} - 1 \right) \right]^{n/2}
\]

\[
\geq \left[ \frac{1}{2} \left( \frac{n}{2} \right) \left( \frac{n}{4} \right) \right]^{n/2},
\]

provided \( (n/4) \geq 1 \), which holds for \( n \geq 4 \).

\[
\lg[T(n)] \geq \frac{n}{2} [\lg(n^2) - 4] = n \lg n - 2n \geq \frac{n \lg n}{2},
\]

provided \( \lg n \geq 4 \), which holds for \( n \geq 16 \).

\[
\lg[T^2(n)] = 2 \lg[T(n)] \geq n \lg n
\]

\[
T^2(n) \geq 2^{n \lg n} = n^n
\]

\[
T(n) \geq (n^{n/2})^n = \left( n^{1/2} \right)^n \geq 2^n, \text{ for } n \geq 4,
\]

indicating an exponential search space for the full binary tree that corresponds to a minimal prefix code over an \( n \)-character alphabet.
The greedy choice chooses to merge characters (or meta-characters) with the smallest frequencies at each step of the algorithm.

Huffman($C$) {
    $n = |C|$;
    for $i = 1$ to $n$ {
        $z$ = new node;
        $z$.char = $C[i]$.char;
        $z$.freq = $C[i]$.freq;
        $Q[i] = z$;
    }
    $Q \leftarrow \text{Build-MinHeap}(Q)$; // $O(n)$ to build minHeap on $Q[i].freq$
    for $i = 1$ to $n - 1$ {
        $x = Q.\text{ExtractMin}()$; // $O(\lg n)$ for extraction
        $y = Q.\text{ExtractMin}()$;
        $z$ = new node;
        $z$.left = $x$;
        $z$.right = $y$;
        $z$.freq = $x$.freq + $y$.freq;
        $Q.\text{Insert}(z)$; // $O(\lg n)$ for insertion
    }
    return $Q.\text{ExtractMin}()$; // $O(1)$ as only one node remains in the minHeap
}

The time complexity of Huffman is $T(n) = O(n \lg n)$. But, does it work? It suffices to prove the optimal substructure and greedy properties.
To verify optimal substructure, we must envision subproblem solutions within an optimal solution overall solution and reason that these subproblem solutions must also be optimal.

Lemma 16.3 (Optimal substructure): Let $T$ be the full binary tree representation of an optimal (minimal) prefix code over alphabet $C$. Let $x, y \in C$ be sibling leaves in $T$ with common parent $z$. Considering $z$ as a meta-character with $f(z) = f(x) + f(y)$, we envision a new alphabet $C' = (C\{x, y\}) \cup \{z\}$. Then $T' = T\{x, y\}$ represents an optimal prefix code for $C'$.

Proof: The sketch below illustrates how we prove the optimal substructure property for the optimal prefix code problem. $T$ represents an optimal solution for the alphabet $C$. To its right, is a tree, $T'$, associated with subproblem alphabet $C'$. The lemma claims that $T'$ is optimal for the subproblem. If it is not, we let $T''$ be a superior solution to the subproblem. That is, $B(T'') < B(T')$. We modify $T''$ to form $T'''$, a competing solution for the original problem. Finally, we show that $B(T''') < B(T)$, which contradicts the given fact that $T$ is optimal.

Specifically, we assume that $T'$ does not represent an optimal prefix code for $C'$.

We relate the average bits per character in codes $T$ and $T'$. Let $d_T(c)$ represent the depth (number of links from the root) of character $c$ in tree $T$. We have, for $c \in [C\{x, y\}] = [C'\{z\}]$,
\[d_T(c) = d_{T'}(c)\]
\[f(c)d_T(c) = f(c)d_{T'}(c)\]
\[d_T(x) = d_T(y) = d_{T'}(z) + 1\]
\[f(x)d_T(x) + f(y)d_T(y) = f(x)[d_{T'}(z) + 1] + f(y)[d_{T'}(z) + 1] = [d_{T'}(z) + 1][f(x) + f(y)] = f(z)d_{T'}(z) + f(x) + f(y)\]
\[f(x) + f(y) = f(x)d_T(x) + f(y)d_T(y) - f(z)d_{T'}(z)\]
\[B(T) = \sum_{c \in C} f(c)d_T(c) = \left( \sum_{c \in C \setminus \{x,y\}} f(c)d_T(c) \right) + f(x)d_T(x) + f(y)d_T(y)\]
\[= \left( \sum_{c \in C \setminus \{z\}} f(c)d_T(c) \right) + f(x)d_T(x) + f(y)d_T(y)\]
\[= \left( \sum_{c \in C'} f(c)d_T(c) \right) - f(z)d_{T'}(z) + f(x)d_T(x) + f(y)d_T(y)\]
\[= \left( \sum_{c \in C'} f(c)d_T(c) \right) + f(x) + f(y)\]
\[B(T) = B(T') + f(x) + f(y).\]

As we are assuming the \(T'\) is not optimal for \(C'\), there must exists \(T''\) over \(C'\) with \(B(T'') < B(T')\). We locate leaf \(z\) in \(T''\), convert it to an interior node, and attach \(x, y\) as right and left children. These changes produce a new full binary tree \(T'''\) representing a prefix code for \(C\).

\[B(T''') - B(T'') = f(x)d_{T'''}(x) + f(y)d_{T'''}(y) - f(z)d_{T'''}(z)\]
\[= [f(x) + f(y)][1 + d_{T'''}(z)] - f(z)d_{T'''}(z)\]
\[= f(z)d_{T'''}(z) + f(x) + f(y) - f(z)d_{T'''}(z) = f(x) + f(y)\]
\[B(T''') = B(T'') + f(x) + f(y) < B(T') + f(x) + f(y) = B(T).\]

This inequality is a contradiction because, by hypothesis, \(T\) is optimal for \(C\).
We conclude that the optimal substructure property holds, with subproblems obtained by excising any pair of sibling leaves and treating their common parent as a new meta-character.
The greedy choice first merges the two nodes of lowest frequency, say $x$ and $y$. This choice appears in the full binary tree representation of the eventual code as sibling leaves $x$ and $y$ at the lowest level.

Lemma 16.2 (Greedy property): Let $C$ be an alphabet, and let $f(c)$ denote the relative frequency of $c \in C$. Suppose that $x, y \in C$ have the lowest frequencies. Wolog, $f(x) \leq f(y)$. Then there exists an optimal prefix code for $C$ in which $x$ and $y$ have the longest codes and differ only in the last bit. That is, there exists an optimal prefix code for which the full binary tree representation presents $x$ and $y$ as sibling leaves at the lowest level.

Proof. Let full binary tree $T$ represent an optimal prefix code for $C$. If $x$ and $y$ appear as sibling leaves at the lowest level, we are finished. Otherwise, we transform the tree.

Case 1: $x$ and $y$ appears on the lowest level but are not siblings. Let $z$ be the sibling of $x$ in $T$. We form a new tree $T'$ by swapping $y$ and $z$. In this new tree, $x$ and $y$ are siblings at the lowest level. With $B(T)$ continuing to refer to the average code length of $T$, we have $B(T) = B(T')$, because no characters have changed their code lengths. Consequently, $B(T')$ is an optimal prefix code for $C$ in which $x$ and $y$ have the longest codes and differ only in the last bit.

Case 2: One of $x, y$ does not appear on the lowest level. Wolog, $x$ does not appear on the lowest level. We choose a character $z$ on the lowest level. If $y$ appears on the lowest level, then we choose $z$ to be the sibling of $y$. Otherwise, we choose an arbitrary $z$ on the lowest level. We then swap nodes $x$ and $z$ forming tree $T'$.

$$
B(T) - B(T') = f(x)d_T(x) + f(z)d_T(z) - [f(x)d_{T'}(x) + f(z)d_{T'}(z)]
= f(x)d_T(x) + f(z)d_T(z) - f(x)d_T(z) - f(z)d_T(x)
= [f(z) - f(x)][d_T(z) - d_T(x)] \geq 0,
$$

Since $f(z) - f(x) \geq 0$ and $d_T(z) - d_T(x) \geq 0$. It follows that $B(T') \leq B(T)$. Now, since $B(T)$ was minimal, we have $B(T')$ is minimal, and we have an optimal prefix code with $x$ on the lowest level. If $y$ were on the lowest level in $T$ at the outset, the swap produces an optimal prefix code in which $x$ and $y$ have the longest codes and differ only in the last bit. Otherwise, we proceed with Case 3.
Case 3: Neither $x$ nor $y$ appear on the lowest level of $T$. We proceed as in Case 2 to swap $x$ to the lowest level, producing optimal tree $T'$. We then form tree $T''$ by swapping $y$ with the sibling of $x$, say $z$, in $T'$.

\[
B(T') - B(T'') = f(y)d_{T'}(y) + f(z)d_{T'}(z) - [f(y)d_{T''}(y) + f(z)d_{T''}(z)]
\]
\[
= f(y)d_{T'}(y) + f(z)d_{T'}(z) - f(y)d_{T'}(z) - f(z)d_{T'}(y)
\]
\[
= [f(z) - f(y)][d_{T'}(z) - d_{T'}(y)] \geq 0,
\]

since $f(z) - f(y) \geq 0$ and $d_{T'}(z) - d_{T'}(y) \geq 0$. Consequently, $B(T'') \leq B(T')$. Then $B(T')$ optimal forces $B(T'')$ to be an full binary tree representation of an optimal prefix code for $C$ in which $x$ and $y$ are siblings at the lowest level.