
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 \textbf{compatible} if

\[
(i \neq j) \& (a_i, a_j \in A) \Rightarrow [s_i, f_i) \cap [s_j, f_j) = \phi.
\]

\[
\begin{array}{c|cccccccccccc}
  i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
  s_i & 1 & 3 & 0 & 5 & 3 & 5 & 6 & 8 & 8 & 2 & 12 \\
  f_i & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\end{array}
\]

\{3, 9, 11\} is compatible.

\{1, 4, 8, 11\} is compatible and larger.

Problem: Find a compatible subset of maximal size.
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 \(f_k > s_j\).
With the objective function, \( c[i, j] \), being the size of the chosen set, the recursive formula is

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

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.
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.

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 $j = i + 1$
                // check that $a_k \in S_{ij}$
                if $s_k \geq f_i$ and $f_k \leq s_j$
                   $t = x(i, k) + x(k, j) + 1$;
                   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.

```java
OptimalSet(y, i, j) {
    // for problem \( S_{ij} \); use \( (i, j) = (0, n + 1) \) for initial problem
    if (y(i, j) == 0)
        return \( \phi \);
    k = y(i, j);
    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

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

Problem:

\[\begin{array}{cccc}
a_i & \multicolumn{3}{c}{\text{any activity}} & a_j \\
\end{array}\]
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 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.

**Dynamic Programming**

1. Rod-cutting
2. Matrix Chain Multiplication
3. Polygon Triangulation
4. CYK Parsing

**Greedy**

1. Activity Selection
2. Fractional Knapsack
3. Huffman Codes
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:

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

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

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

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

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.
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.

```plaintext
FractionalKnapsack(w, v, W) {  // assume integer inputs
    n = length(w);
    wgt = 0;
    for k = 1 to n
        x[k] = 0;
    k = 1;
    while (wgt < W and k ≤ n) {
        if w[k] ≤ W − wgt
            x[k] = w[k];
        else
            x[k] = W − wgt;
        wgt = wgt + x[k];
        k = k + 1;
    }
    return x;
}
```

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:

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

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 $B(T) = 3.0$.

![D Dodson(huffman codes)24.png](image)
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

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

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

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 $n \geq 16$.

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

$$T^2(n) \geq 2^n \lg n = n^n$$

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

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 $C' = (C \setminus \{x, y\}) \cup \{z\}$. Then $T' = T \setminus \{x, y\}$ 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.

![Sketch of trees](image)

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 $c \in [C \setminus \{x, y\}] = [C' \setminus \{z\}]$, 


\[d_T(c) = d_{T'}(c)\]
\[f(c)d_T(c) = f(c)d_{T'}(c)\]
\[d_T(x) = d_T(y) = d_{T'}(z) + 1\]
\[f(x)d_T(x) + f(y)d_T(y) = f(x)[d_{T'}(z) + 1] + f(y)[d_{T'}(z) + 1] = [d_{T'}(z) + 1][f(x) + f(y)]\]
\[= f(z)d_{T'}(z) + f(x) + f(y)\]
\[f(x) + f(y) = f(x)d_T(x) + f(y)d_T(y) - f(z)d_{T'}(z)\]
\[B(T) = \sum_{c \in C} f(c)d_T(c) = \left( \sum_{c \in C \setminus \{x,y\}} f(c)d_T(c) \right) + f(x)d_T(x) + f(y)d_T(y)\]
\[= \left( \sum_{c \in C \setminus \{z\}} f(c)d_T(c) \right) + f(x)d_T(x) + f(y)d_T(y)\]
\[= \left( \sum_{c \in C'} f(c)d_T(c) \right) - f(z)d_{T'}(z) + f(x)d_T(x) + f(y)d_T(y)\]
\[= \left( \sum_{c \in C'} f(c)d_T(c) \right) + f(x) + f(y)\]
\[B(T) = B(T') + f(x) + f(y).\]

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.
Matroids.

Definition. A matroid is a pair $\mathcal{M} = (S, \mathcal{L})$, such that

1. $S \neq \emptyset$ is a finite set
2. $\mathcal{L} \neq \emptyset$ is a collection of subsets of $S$ possessing the hereditary and exchange properties.
   (a) hereditary property: $B \in \mathcal{L}, A \subseteq B \Rightarrow A \in \mathcal{L}$
   (b) exchange property: $A, B \in \mathcal{L}, |A| < |B| \Rightarrow (\exists x \in B \setminus A)(A \cup \{x\} \in \mathcal{L})$.
3. The elements of $\mathcal{L}$ are called the admissible (also independent) subsets of $S$. 
Examples.

1. (Theorem 16.5) Let $G = (V, E)$ be an undirected graph with $E \neq \phi$. Let $\mathcal{E}$ be the collection of all subsets of $E$. Define $\mathcal{M} = (E, \mathcal{L})$, where

$$\mathcal{L} = \{X \in \mathcal{E} : G_X = (V, X) \text{ is acyclic (that is, } (V, X) \text{ is a forest)}\}.$$ 

We note that $\phi \in \mathcal{L}$. Specifically, it satisfies the hereditary property because its sole subset is itself, and it satisfies the exchange property by vacuity. Also the singleton $\{e\} \in \mathcal{L}$ for any $e \in E$. A singleton satisfies the hereditary property because its only subsets are itself and the empty set. Also, since the only member of $\mathcal{L}$ of smaller size is the empty set, the exchange property is satisfied with $x = e$.

In general, if $B \in \mathcal{L}$ and $A \subset B$, then $A \in \mathcal{L}$ because removing edges from a forest cannot introduce any cycles.

Finally, suppose $A, B \in \mathcal{L}$ with $|A| < |B|$. The following tabulation lists the number of distinct tree in the forest $(V, B)$ as a function of the size of $B$. It is constructed by exploiting the fact that a tree with $n$ vertices has exactly $n - 1$ edges.

| number of trees in $(V, B)$ | $|B|$                      |
|-----------------------------|---------------------------|
| 1                           | $|V| - 1$                  |
| 2                           | $|V| - 2 = (k_1 - 1) + (k_2 - 1)$ where $k_1 + k_2 = |V|$ |
| 3                           | $|V| - 3 = (k_1 - 1) + (k_2 - 1) + (k_3 - 1)$ where $k_1 + k_2 + k_3 = |V|$ |

We conclude that $(V, B)$ contains $|V| - |B|$ trees and $(V, A)$ contains $|V| - |A|$ trees. Since $|B| > |A|$, $(V, A)$ contains more trees than $(V, B)$. Consequently, some tree in $(V, B)$, say $T_B$ must contain two vertices that appear in distinct trees in $(V, A)$. As there exists a path between these two vertices in $T_B$, we can proceeds from the path ends toward the center to obtain two vertices that appear in distinct trees in $(V, A)$ and have an edge $e$ between them. That is, this edge $e \in B \setminus A$. Since the edge connects two trees in $A$, $A \cup \{e\}$ contains no cycles. That is, $A \cup \{e\} \in \mathcal{L}$. The exchange property is satisfied.

We conclude that $\mathcal{M} = (E, \mathcal{L})$ is a matroid.
2. Let $V$ be a finite subset of a $d$-dimensional real vector space. Let $\mathcal{V}$ be the collection of all subsets of $V$. Define $\mathcal{M} = (V, \mathcal{L})$, where

$$\mathcal{L} = \{ X \in \mathcal{V} : X \text{ is an independent set} \}.$$ 

Recall $X = \{x_1, x_2, \ldots, x_n\}$ is an independent set if

$$\sum_{i=1}^{n} a_i x_i = 0 \Rightarrow a_1 = a_2 = \cdots = a_n = 0.$$ 

As in the previous example, $\phi \in \mathcal{L}$, as if $\{x\}$ for any singleton nonzero vector $x$. Suppose $B \in \mathcal{L}$ and $A \subseteq B$. Wolog, we assume

$$A = \{x_1, x_2, \ldots, x_m\}; B = \{x_1, x_2, \ldots, x_m, x_{m+1}, \ldots, x_n\}.$$ 

Suppose

$$\sum_{i=1}^{m} a_i x_i = 0.$$ 

Define

$$b_i = \begin{cases} a_i, & 1 \leq i \leq m \\ 0, & m + 1 \leq i \leq n. \end{cases}$$ 

Then,

$$\sum_{i=1}^{n} b_i x_i = \sum_{i=1}^{m} a_i x_i = 0$$

$$b_i = 0, \quad \text{for all } 1 \leq i \leq n$$

$$a_i = 0, \quad \text{for all } 1 \leq i \leq m$$

$$A \in \mathcal{L}.$$ 

The hereditary property holds.

Now suppose $A, B \in \mathcal{L}$ with $m = |A| < |B| = n$. The $m$-dimensional subspace $\text{span}(A)$ can contain no more than $m$ independent vectors. As $n > m$, at least one of the vectors in $B$, say $x$, lies outside $\text{span}(A)$. Therefore, $A \cup \{x\}$ remains an independent set. That is, $A \cup \{x\} \in \mathcal{L}$. The exchange property holds.

We conclude that $\mathcal{M} = (V, \mathcal{L})$ is a matroid.
Definitions. Let $\mathcal{M} = (S, \mathcal{L})$ be a matroid.

1. If $A \in \mathcal{L}$, $x \in S \setminus A$, and $A \cup \{x\} \in \mathcal{L}$, then $x$ is an extension of $A$.
2. $A \in \mathcal{L}$ is maximal if $A$ has no extensions.

Theorem 16.6: Let $\mathcal{M} = (S, \mathcal{L})$ be a matroid. Then $A, B$ maximal $\in \mathcal{L}$ implies $|A| = |B|$.

Proof. Suppose $A, B \in \mathcal{L}$ are maximal and $|A| < |B|$. Then, by the exchange property, there exists $x \in B \setminus A$ such that $A \cup \{x\} \in \mathcal{L}$. That is, $x$ is an extension of $A$ — a contradiction. $\blacksquare$
Example.

Let $G = (V, E)$ be an \textbf{connected} undirected graph. Let $\mathcal{E}$ be the collection of all subsets of $E$. Define $\mathcal{M} = (E, \mathcal{L})$, where

$$\mathcal{L} = \{ X \in \mathcal{E} : G_X = (V, X) \text{ is acyclic (that is, } (V, X) \text{ is a forest)} \}.$$

Then $A$ maximal in $\mathcal{L}$ implies $A$ is a spanning tree with $|V| - 1$ edges.

Why? $A \in \mathcal{L}$ means that $(V, A)$ is a forest. If $(V, A)$ contains more than one tree, then the connectedness of $G$ implies there exists an edge $e \in E$ connecting vertices in two distinct trees of $(V, A)$. Consequently, $A \cup \{e\}$ is acyclic, which places $A \cup \{e\} \in \mathcal{L}$. That is, $e$ is an extension of $A$ — a contradiction.

We conclude that $(V, A)$ contains exactly one tree connecting all vertices of $V$. That is, $(V, A)$ is a spanning tree. Moreover, a tree on $|V|$ vertices contains $|V| - 1$ edges.
Definitions.

1. $\mathcal{M} = (S, \mathcal{L})$ is called a **weighted** matroid if there exists $w : S \to \mathbb{R}^+ \setminus \{0\}$. That is, $w(x) > 0$ for all $x \in S$.

2. For $A \in \mathcal{L}$, define
   \[ w(A) = \sum_{x \in A} w(x). \]

3. $A$ is **optimal** for $((S, \mathcal{L}), w)$ if $A \in \mathcal{L}$ and $w(A) = \max\{w(B) : B \in \mathcal{L}\}$.

We note that $A$ optimal implies $A$ maximal. If not, then there exists an optimal $A$ which has an extension, say $x$. Since $w(x) > 0$, we have $A \cup \{x\} \in \mathcal{L}$ with $w(A \cup \{x\}) = w(A) + w(x) > w(A)$ — a contradiction.

**Find-Maximal($\mathcal{M} = (S, \mathcal{L})$) {**

Let $S = (x_1, x_2, \ldots, x_n)$;

$A = \phi$;

for $i = 1$ to $n$

if $A \cup \{x_i\} \in \mathcal{L}$

$A = A \cup \{x_i\};$

return $A$;

}

**Lemma:** The Find-Maximal algorithm returns a maximal element.

**Proof.** For purposes of deriving a contradiction, suppose that the return $A$ is not maximal. Then $A$ has an extension, say $x_j \notin A$. Let $A' = \{x_k \in A : k < j\}$.

When iteration $j$ of the for-loop begins, variable $A$ is equal to $A'$ because $A$ at that point must have accumulated all points in the returned set with indices less than $j$. Therefore, at this point, $A \cup \{x_j\} \in \mathcal{L}$, which forces the algorithm to include $x_j$ into $A$. As the algorithm never removes an element from $A$, we conclude that $x_j$ is in the returned $A$ — a contradiction.  

32
Definition. Given a matroid \( M = (S, \mathcal{L}) \) and a set \( B \in \mathcal{L} \) for which \( B \neq \emptyset \) and \( B \neq S \), we define the **contraction** of \( M \) with respect to \( B \) as \( M\setminus B = (S\setminus B, \mathcal{L}') \) where

\[
\mathcal{L}' = \{ A \subseteq S\setminus B : A \cup B \in \mathcal{L} \}.
\]

Lemma: The contraction of a matroid is itself a matroid.

Proof. As \( B \neq S \), we have \( S\setminus B \neq \emptyset \). Also, \( B \cup \emptyset = B \in \mathcal{L} \), which places \( \{\emptyset\} \in \mathcal{L}' \). That is, \( \mathcal{L}' \neq \emptyset \).

Suppose \( A \subseteq C, \ C \in \mathcal{L}' \). It follows that \( C \subseteq S\setminus B \) and \( C \cup B \in \mathcal{L} \). \( A \subseteq C \) then forces \( A \subseteq S\setminus B \) and the hereditary property of \( \mathcal{L} \), together with \( A \cup B \subseteq C \cup B \), forces \( A \cup B \in \mathcal{L} \). We conclude that \( A \in \mathcal{L}' \), thereby proving the hereditary property for \( (S\setminus B, \mathcal{L}') \). Also, for \( A_1, A_2 \in \mathcal{L}' \) with \( |A_1| > |A_2| \), we have

\[
A_1 \subseteq S'; A_2 \subseteq S\setminus B, \quad A_1 \cup B \in \mathcal{L}; A_2 \cup B \in \mathcal{L}, \quad |A_1 \cup B| > |A_2 \cup B|.
\]

\[
\exists x \in (A_1 \cup B) \setminus (A_2 \cup B) \text{ such that } A_2 \cup B \cup \{x\} \in \mathcal{L}.
\]

\[
x \notin A_2; x \notin B; x \in A_1 \subseteq S\setminus B; x \in A_1 \setminus A_2.
\]

\[
A_2 \cup \{x\} \subseteq S\setminus A_2; A_2 \cup \{x\} \in \mathcal{L}'.
\]

The exchange property for \( (S\setminus B, \mathcal{L}') \) holds. That is, \( (S\setminus B, \mathcal{L}') \) is a matroid.  □
Lemma 16.10 (Optimal substructure): In weighted matroid \( M = (S, \mathcal{L}) \), suppose, for \( n \geq 1 \), \( X = \{x_1, x_2, \ldots, x_n\} \) is an optimal element of \( M \). Then \( Y = \{x_2, \ldots, x_n\} \) is an optimal element of \( M \) contracted via \( B = \{x_1\} \).

Proof. \( X \) optimal implies \( X \in \mathcal{L} \). Therefore the specified contraction is well-defined. That contraction is

\[
M' = (S' = S \setminus \{x_1\}, \mathcal{L}' = \{A \subseteq S' : A \cup \{x_1\} \in \mathcal{L}\}).
\]

We have \( Y \subseteq S' \) and \( X = Y \cup \{x_1\} \in \mathcal{L} \). Therefore \( Y \in \mathcal{L}' \).

Let \( w : S \to \mathcal{R}^+ \setminus \{0\} \) be the weight function. For purposes of deriving a contradiction, suppose there exists \( Z \in \mathcal{L}' \) with \( w(Z) > w(Y) \). Then \( Z \cup \{x_1\} \in \mathcal{L} \), and

\[
w(Z \cup \{x_1\}) = w(x_1) + w(Z) > w(x_1) + w(Y) = \sum_{i=1}^{n} w(x_i) = w(X),
\]

a contradiction. We conclude that \( Y = \{x_2, \ldots, x_n\} \) is optimal in \( M' \). \( \blacksquare \)
The following algorithm calculates an optimal subset in matroid $\mathcal{M} = (S, \mathcal{L})$ by (a) first choosing the heaviest admissible singleton $\{x\} \in \mathcal{L}$, and then (b) continuing by choosing the heaviest admissible singleton in the matroid obtained by contracting $\mathcal{M}$ via the set of elements already chosen. These criteria are actually the same, since $\mathcal{M}$ equals $\mathcal{M}$ contracted via $\phi$.

```
Greedy($\mathcal{M} = (S, \mathcal{L})$) {
    $A = \phi$;
    sort $S$ on decreasing $w(x)$, obtaining $S = (x_1, x_2, \ldots, x_n)$;
    for $i = 1$ to $n$
        if $A \cup \{x_i\} \in \mathcal{L}$
            $A = A \cup \{x_i\}$;
    return $A$;
}
```
Lema 16.7 (Greedy property) In weighted matroid, \( \mathcal{M} = (S, \mathcal{L}) \), suppose that \( S = (x_1, x_2, \ldots, x_n) \), sorted in order of descending \( w(x_i) \). Let

\[
j = \min\{k : \{x_k\} \in \mathcal{L}\}.
\]

Then there exists an optimal \( X \in \mathcal{L} \) such that \( x_j \in X \).

Proof. Because of the sorted order, \( k < j \) implies singleton \( \{x_k\} \not\in \mathcal{L} \).

Let \( Y \) be optimal in \( \mathcal{L} \). If \( x_j \in Y \), we let \( X = Y \). If \( x_j \not\in Y \), we note that \( y \in Y \) implies \( y = x_k \) for some \( k > j \). (Otherwise, the hereditary property forces singletons \( x_k \) with \( k < j \) into \( Y \)). The sorted order then ensures that \( y \in Y \) implies \( w(y) \leq w(x_j) \).

We initialize \( X = \{x_j\} \), and use the exchange property to add elements of \( Y \) to \( X \). Specifically,

\[
X = \{x_j\};
\]
while \( |X| < |Y| \) {
\[
\text{choose } z \in Y \setminus X; \\
X = X \cup \{z\};
\]
}\n
When this process concludes, we have \( |X| = |Y| \). Moreover, the contents of \( X \) are nearly the same as the contents of \( Y \). In particular, \( X \) contains all of the elements of \( Y \), except one. In place of that one missing element, \( X \) contains \( x_j \). Relabeling, we have

\[
Y = (v_1, v_2, \ldots, v_n)
\]
\[
X = (x_j, v_2, \ldots, v_n)
\]
\[
w(X) - w(Y) = w(x_j) - w(v_1) \geq w(x_j) - w(x_j) = 0
\]
\[
w(X) \geq w(Y).
\]

Since \( Y \) is optimal, we must have \( w(X) = w(Y) \), which proves that \( X \) is also optimal. \( \blacksquare \)
We use the greedy and optimal substructure properties to construct an optimal admissible subset in weighted matroid \((S, L)\).

1. Order \(S\) as \((x_1, x_2, \ldots, x_n)\) on descending \(w(x_i)\).

2. The greedy choice is the first \(x_{i_1}\) such that singleton \(\{x_{i_1}\} \in L\). By the greedy choice property, there exists an optimal member of \(L\), say \(C_1\), with \(x_{i_1} \in C_1\). Let \(W = w(C_1)\).

3. By optimal substructure, \(C_1\setminus\{x_{i_1}\}\) is optimal in the contraction via \(\{x_{i_1}\}\). Hence the weight of any optimal set in the contraction is \(w(C_1) - w(x_{i_1})\).

4. The greedy choice in the contraction is the first \(x_{i_2}\) in the sorted order after \(x_{i_1}\) such that singleton \(\{x_{i_2}\}\) is admissible. That is, \(\{x_{i_1}, x_{i_2}\} \in L\). By the greedy choice property, there exists an optimal admissible set in the contraction, say \(D\), with \(x_{i_2} \in D\). Because all optimal admissible sets are the same size, we have \(w(D) = w(C_1) - w(x_{i_1})\).

5. Defining \(C_2 = D \cup \{x_{i_1}\}\), we now have \(w(C_1) = w(D) + w(x_{i_1}) = w(C_2)\), which shows that \(C_2\) is an optimal admissible set in the original matroid, and it contains both greedy choices, \(x_{i_1}\) and \(x_{i_2}\).

6. By optimal substructure, \(C_2\setminus\{x_{i_1}, x_{i_2}\}\) is optimal in the contraction via \(\{x_{i_1}, x_{i_2}\}\). Hence the weight of any optimal set in the contraction is \(w(C_2) - w(x_{i_1}) - w(x_{i_2})\).

7. The greedy choice in the contraction is the first \(x_{i_3}\) in the sorted order after \(x_{i_2}\) such that singleton \(\{x_{i_3}\}\) is admissible. That is, \(\{x_{i_1}, x_{i_2}, x_{i_3}\} \in L\). By the greedy choice property, there exists an optimal admissible set in the contraction, say \(D\), with \(x_{i_3} \in D\). Because all optimal admissible sets are the same size, we have \(w(D) = w(C_2) - w(x_{i_1}) - w(x_{i_2})\).

8. Defining \(C_3 = D \cup \{x_{i_1}, x_{i_2}\}\), we now have \(w(C_3) = w(D) + w(x_{i_1}) + w(x_{i_2}) = w(C_2)\), which shows that \(C_3\) is an optimal admissible set in the original matroid, and it contains the three choices to this point: \(x_{i_1}, x_{i_2}\), and \(x_{i_3}\).

9. And so forth, until there exist no further greedy choices. At that point, we have an optimal admissible subset in the original matroid, constructed via repeated greedy choices.
Greedy($\mathcal{M} = (S, \mathcal{L})$) {
    $A = \emptyset$;
    sort $S$ on decreasing $w(x)$, obtaining $S = (x_1, x_2, \ldots, x_n)$;
    for $i = 1$ to $n$
        if $A \cup \{x_i\} \in \mathcal{L}$
            $A = A \cup \{x_i\}$
    return $A$;
}

The running time for the generic Greedy algorithm depends on the complexity of the test “if $A \cup \{x_i\} \in \mathcal{L}$ . . . .” Given a matroid $(S, \mathcal{L})$ with $|S| = n$, we need $O(n \lg n)$ time for the sort, and assuming $O(f(n))$ for each test, we need $O(nf(n))$ for the loop. The general expression for the time complexity is then $O(n \lg n + nf(n))$.  

Example: **Minimal Spanning Tree.**

**Definition.** Let $G = (V, E)$ be a connected, undirected graph. Let $w : E \to \mathbb{R}^+$ be a weight function on the edges of $G$. A **minimal spanning tree** for $G$ is an acyclic set $T \subseteq E$, such (a) $v \in V$ implies that $v$ is an endpoint of at least one edge in $T$, and (b) $w(T) = \sum_{e \in T} w(e)$ is minimal.

We can obtain a minimal spanning tree with a greedy approach using an appropriate matroid. Specifically, let

$$
\mathcal{L} = \{ F \subseteq E : F \text{ is acyclic} \}
$$

$$
\mathcal{M} = (E, \mathcal{L})
$$

$$
K = 1 + \max_{e \in E} w(e)
$$

$$
w' : E \to \mathbb{R}^+ \setminus \{0\} \text{ via } w'(e) = K - w(e).
$$

Since any tree connecting $V$ vertices contains exactly $V - 1$ edges, we have for any such tree $T = \{e_1, e_2, \ldots, e_{V-1}\}$

$$
w'(T) = \sum_{j=1}^{V-1} (K - w(e_j)) = K(V - 1) - \sum_{j=1}^{V-1} w(e_j).
$$

Hence $w'(T)$ is maximum when $w(T)$ is minimum. In particular, an edge sort on increasing $w(e)$ provides an edge sort on decreasing $w'(e)$. We showed in an earlier example that $\mathcal{M}$ is a matroid. With the addition of $w'$, we have a weighted matroid in which an optimal admissible set constitutes a minimal spanning tree. We can tailor the **Greedy** algorithm to solve for a minimal spanning tree as follows.
Minimal-Spanning-Tree($G = (V, E), w : E \to \mathbb{R}^+$) {
    $T = \phi$;
    sort $E$ on increasing $w(x)$, obtaining $E = (e_1, e_2, \ldots, e_n)$;
    for $i = 1$ to $n$
        if $T \cup \{e_i\}$ is acyclic
            $T = T \cup \{e_i\}$;
    return $T$;
}

As edges are added to the growing set $T$, an original set of $V$ disconnected vertices evolves as a forest, finally coalescing into a single tree. We start with $V$ separate sets, each containing a single vertex. Each edge added to $T$ connects two trees in this forest, thereby decreasing the number of trees by one. Kruskal’s version of this greedy algorithm manipulates the sets in a manner that introduces an amortized $\lg V$ for the test “if $A \cup \{x_i\} \in \mathcal{L}$ . . . ”. (See Chapter 23.) That is, we have $O(E \lg E)$ for the sort and $E \lg V$ for the loop:

\[
E = O(V^2) \\
\lg E = O(\lg V) \\
T(n) = O(E \lg E) + O(E \lg V) = O(E \lg V).
\]
Example: **Unit Task Scheduling.**

A single machine executes all of a sequence of tasks. Each task requires exactly one unit of time. Each task has a deadline and an associated penalty for finishing after that deadline.

\[
S = (a_1, a_2, \ldots, a_n), \quad \text{unit-time tasks}
\]
\[
D = (d_1, d_2, \ldots, d_n), \quad \text{deadlines, } 1 \leq d_i \leq n
\]
\[
W = (w_1, w_2, \ldots, w_n), \quad \text{penalties for missing deadline}
\]

Wolog, we assume \(w_i > 0\) for \(1 \leq i \leq n\), because we can place all zero-penalty tasks at the end with no further increase in the total penalty. We want to order the tasks for sequential entry into the machine such that the total penalty is minimum.

**Definitions.**

1. A **schedule** is a permutation of \((a_1, a_2, \ldots, a_n)\), say \((a_{i_1}, a_{i_2}, \ldots, a_{i_n})\), implying execution order \(a_{i_1}, a_{i_2}, \ldots, a_{i_n}\).
2. Given a schedule \((a_{i_1}, a_{i_2}, \ldots, a_{i_n})\), task \(a_{i_j}\) is **early** if \(d_{i_j} \geq j\).
3. Given a schedule \((a_{i_1}, a_{i_2}, \ldots, a_{i_n})\), task \(a_{i_j}\) is **late** if \(d_{i_j} < j\).
4. Given a schedule \(C = (a_{i_1}, a_{i_2}, \ldots, a_{i_n})\), the penalty for \(C\) is

\[
P_C = \sum_{a_{i_j} \text{ late in } C} w_{i_j}.
\]

We seek a schedule \(C\) such that \(P_C = \min_D \{P_D\}\). Since there are \(n!\) schedules for a problem with \(n\) tasks, we first show that our optimal schedule can be found within a smaller search space.
Definitions.

1. A **canonical schedule** \(\{a_{i_1}, a_{i_2}, \ldots, a_{i_n}\}\) is a schedule that obeys the following constraints.
   
   - \(a_{i_j}\) early and \(a_{i_k}\) late implies \(j < k\)
   - \(a_{i_j}\) early, \(a_{i_k}\) early, and \(j < k\) implies \(d_{i_j} \leq d_{i_k}\).

   That is, a canonical schedule consists of a prefix of early tasks followed by a suffix of late tasks. Moreover, the early tasks appear in order of increasing deadlines.

2. Two schedules are **equivalent** if they have the same penalty. That is, schedules \(A_1\) and \(A_2\) are equivalent if \(P_{A_1} = P_{A_2}\).
Lemma. Every schedule has an equivalent canonical schedule.

Proof. Let \( A = \{a_{i_1}, \ldots, a_{i_n}\} \) be an arbitrary schedule. A late task preceding an early task, as in the sketch below, invites a swap in which the early task remains early and the late task remains late. Hence the penalty remains unchanged. Repeating this procedure as necessary we arrive at an equivalent schedule in which a prefix of early tasks precedes a suffix of late tasks.

In such a task, consider now two adjacent early tasks in which the task closer to the front of the list has the later deadline. The sketch below depicts this situation.

After the indicated swap, \( a_{i_k} \) remains early because it is moved closer to the beginning of the list. As for \( a_{i_j} \), we see that its deadline has \( d_{i_j} > d_{i_k} \geq t \). Therefore, \( a_{i_j} \) remains early when moved one slot farther toward the end of the list. The overall penalty of the schedule remains unchanged, and after continued swaps of this nature, we arrive at an equivalent canonical schedule. 

We conclude that an optimal schedule lies among the canonical schedules.
We also note that minimizing the penalty $P_C$ across schedules $C$ is equivalent to maximizing

$$P'_C = \left( \sum_{i=1}^{n} w_i \right) - \left( \sum_{i : a_i \text{ is late in } C} w_i \right) = \left( \sum_{i : a_i \text{ is early in } C} w_i \right).$$

We now define a relevant matroid.

$$S = \{a_1, a_2, \ldots, a_n\}$$

$$w : S \rightarrow \mathbb{R}^+ \setminus \{0\} \text{ via } w(a_i) = w_i$$

$$\mathcal{L} = \{A \subseteq S : \text{there exists a schedule } C \text{ in which } a \in A \Rightarrow A \text{ is early in } C\}$$

$$\mathcal{M} = (S, \mathcal{L})$$

We will shortly prove that $\mathcal{M}$ is a matroid. However, under that assumption, an optimal $A \in \mathcal{L}$ provides that largest possible $P'_A$ and the corresponding smallest possible $P_A$. We can convert $A$ into a canonical schedule by listing the early tasks in order of increasing deadlines, followed by the late tasks in any order.
Definition. Let \((S, \mathcal{L})\) be the matroid defined above. For \(t = 0, 1, 2, \ldots, n\) and \(A \subseteq S\), define

\[
N_t(A) = \#\{a_i \in A : d_i \leq t\},
\]

where \(\#B\) denotes the number of elements in set \(B\). Because we have assumed all penalties \(w_i > 0\), we have \(N_0(A) = 0\) for all \(A \subseteq S\).
Lemma 16.12. Let \( \mathcal{M} = (S, \mathcal{L}) \) be the matroid defined above, and let \( A \subseteq S \). Then \( A \in \mathcal{L} \) if and only if \( N_t(A) \leq t \) for all \( 0 \leq t \leq n \).

Proof. For an arbitrary \( A \subseteq S \), suppose there exists \( t \) with \( 0 \leq t \leq n \) and \( s = N_t(A) > t \). That is, \( A \) contains \( s > t \) tasks, each needing to complete by time \( t \). Since the interval \([0, t]\) contains only \( t \) unit time slots, it is not possible to schedule \( s \) tasks in this interval. Consequently, at least one task in \( A \) will be late in any schedule. We conclude that \( A \notin \mathcal{L} \). By contrapositive, we have shown \( A \in \mathcal{L} \Rightarrow N_t(A) \leq t \) for all \( 0 \leq t \leq n \).

Conversely, suppose \( N_t(A) \leq t \) for all \( 0 \leq t \leq n \). Order \( A \) by nondecreasing deadlines, followed by \( S \setminus A \) in any order to obtain schedule \( C \):

\[
C = \{ a_{i_1}, a_{i_2}, \ldots, a_{i_k}, a_{i_{k+1}}, a_{i_{k+2}}, \ldots, a_n \} \\
d_{i_1} \leq d_{i_2} \leq \ldots \leq d_{i_k}.
\]

Then,

\[
N_1(A) \leq 1 \Rightarrow \text{at most one member of } A \text{ competes for slot } [0, 1) \Rightarrow d_{i_1} \geq 1, d_{i_2} \geq 2 \\
N_2(A) \leq 2 \Rightarrow \text{at most two members of } A \text{ compete for slots } [0, 2) \Rightarrow d_{i_3} \geq 3 \\
N_3(A) \leq 3 \Rightarrow \text{at most three members of } A \text{ compete for slots } [0, 3) \Rightarrow d_{i_4} \geq 4 \\
\vdots \\
N_{k-1}(A) \leq k - 1 \Rightarrow \text{at most } k - 1 \text{ members of } A \text{ compete for slots } [0, k - 1) \Rightarrow d_{i_k} \geq k.
\]

We conclude that all members of \( A \) are early in \( C \). That is, \( A \in \mathcal{L} \).
Theorem 16.13. Let $\mathcal{M} = (S, \mathcal{L})$ be as defined above. Then $\mathcal{M}$ is a matroid.

Proof. Suppose that $A, B$ are subsets of $S$, $B \in \mathcal{L}$, and $A \subseteq B$. By the definition of $\mathcal{L}$, there exists a schedule in which all tasks in $B$ are early. Consequently, the following canonical schedule also exhibits $b \in B \Rightarrow b$ early.

$$C = \{a_{i_1}, a_{i_2}, \ldots, a_{i_k}, a_{i_{k+1}}, a_{i_{k+2}}, \ldots, a_{i_n}\}$$

Since $A \subseteq B$, the tasks of $A$ lie among the early tasks shown in $B$ above. Consequently, all tasks in $A$ are early. Schedule $C$ then serves to confirm $A \in \mathcal{L}$. We conclude that $\mathcal{M}$ satisfies the hereditary property.

To prove the exchange property, consider $A, B \in \mathcal{L}$ with $|A| < |B|$.

We note that $N_t(A)$ and $N_t(B)$ are both nondecreasing functions of $t$. This follows because increasing $t$ may capture more tasks with deadlines less than or equal to the new $t$, but the increase will certainly not disqualify any tasks with deadlines less than or equal to the old value of $t$.

Moreover, we have

$$N_0(A) = N_0(B) = 0$$
$$N_n(A) = |A| < |B| = N_n(B),$$

the last deriving from the problem statement, in which all deadlines lie in $[1, n]$.

We observe that both $N_t(A)$ and $N_t(B)$ rise with increasing $t$, both start at $N_0(A) = N_0(B) = 0$, and $N_n(B) > N_n(A)$. Rising at different rates, there may be several $t$ values when $N_t(B)$ pulls ahead of $N_t(A)$. But, since $N_n(B)$ is larger, there must be some maximal $t$ for which $N_t(B) \leq N_t(A)$. We define

$$k = \max\{t : N_t(B) \leq N_t(A)\}.$$ 

By construction, we have $0 \leq k < n$ and $N_t(B) > N_t(A)$ for all $k < t \leq n$. 47
\[ \begin{align*}
N_k(B) & \leq N_k(A) \\
N_{k+1}(B) & > N_{k+1}(A) \\
N_{k+1}(B) & = \#\{a_i \in B : d_i \leq k + 1\} \\
& = \#\{a_i \in B : d_i \leq k\} + \#\{a_i \in B : d_i = k + 1\} \\
& = N_k(B) + \#\{a_i \in B : d_i = k + 1\} \\
N_{k+1}(A) & = N_k(A) + \#\{a_i \in A : d_i = k + 1\}, \quad \text{same argument as for } N_{k+1}(B) \\
0 & < N_{k+1}(B) - N_{k+1}(A) \\
& = \underbrace{N_k(B) - N_k(A)}_{\leq 0} + [\#\{a_i \in B : d_i = k + 1\} - \#\{a_i \in A : d_i = k + 1\}] \\
0 & < [\#\{a_i \in B : d_i = k + 1\} - \#\{a_i \in A : d_i = k + 1\}] .
\end{align*} \]

We conclude that there exists some \( a_q \in B \setminus A \) with \( d_q = k + 1 \). Let \( A' = A \cup \{a_q\} \).

We will use Lemma 16.12 above to show that \( A' \in \mathcal{L} \), which will complete the proof of the exchange property.

Via the lemma, we will have \( A' \in \mathcal{L} \) if we can show \( N_t(A') \leq t \) for \( 0 \leq t \leq n \).

For \( t \leq k \), any \( a_i \in A' \) with \( d_i \leq t \) must come from \( A \), because the only entry in \( A' \setminus A \) is \( a_q \) and \( d_q = k + 1 \). That is,

\[ N_t(A') = \#\{a_i \in A' : d_i \leq t\} = \#\{a_i \in A : d_i \leq t\} = N_t(A) \leq t , \]

the last because \( A \in \mathcal{L} \) and Lemma 16.12. Now, for \( t \geq k + 1 \),

\[ \begin{align*}
N_t(A') & = N_t(A) + 1, \quad \text{because only } a_q \text{ has been added to } A \\
N_t(A') & < N_t(B) + 1, \quad \text{because } N_t(B) > N_t(A) \text{ in this } t \text{ range} \\
N_t(A') & \leq N_t(B) \leq t, \quad \text{because } B \in \mathcal{L} \text{ and Lemma 16.12.}
\end{align*} \]

We now have \( N_t(A') \leq t \) for all \( 0 \leq t \leq n \), and a final appeal to Lemma 16.12 places \( A' \in \mathcal{L} \).