# Greedy Algorithms

Adam W. Bargteil

February 10, 2022

A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

**Activity-selection problem:** Suppose we have a set $S = a_1, a_2, \ldots, a_n$ of $n$ proposed **activities** that wish to use a resource, such as a lecture hall, which can serve only one activity at a time. Each activity $a_i$ has a **start time** $s_i$ and a **finish time** $f_i$, where $0 \leq s_i < f_i < \infty$. If selected, activity $a_i$ takes place during the half-open interval $[s_i, f_i)$. Activities $a_i$ and $a_j$ are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. In the activity-selection problem, we wish to select a maximum-size subset of mutually compatible activities. We assume the activities are sorted in monotonically increasing order of finishing time:

$$f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n.$$

| $i$ | 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 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

$a_3, a_9, a_{11}$ are mutually compatible, but $a_1, a_4, a_8, a_{11}$ is larger and is a largest subset. Another largest is $a_2, a_4, a_9, a_{11}$.

**Optimal substructure:** Let $S_{ij}$ denote the set of activities that start after activity $a_i$ finishes and that finish before $a_j$ starts. Suppose that $A_{ij}$ is a maximum set of compatible activities in $S_{ij}$ that include activity $a_k$. By including $a_k$ we are left with two subproblems: finding mutually compatible activities in $S_{ik}$ and $S_{kj}$. Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$. Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$.

$A_{ij}$ must include optimal solutions to the subproblems $S_{ik}$ and $S_{kj}$. If there exists $A'_{kj}$ in $S_{kj}$ where $|A'_{kj}| > |A_{kj}|$, then we could use $A'_{kj}$ rather than $A_{kj}$. We would then have a set of $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$, which contradicts the assumption that $A_{ij}$ is an optimal solution.

**Dynamic programming:** Let $c[i, j]$ be the size of an optimal solution for $S_{ij}$, then we have the recurrence:

$$c[i, j] = c[i, k] + c[k, j] + 1.$$

With the optimum being

$$c[i, j] = \max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\}$$

for $S_{ij} \neq \emptyset$.

**Greedy choice:** Intuition: choose the activity that leaves the resource free as long as possible; choose the compatible activity with earliest finish time. Given the sorted activities, we choose $a_1$. One remaining subproblem to solve: $S_{1n}$. We do not need to consider activities that start before $a_1$ finishes since they will be incompatible with $a_1$ because $s_1 < f_1$ and $f_1$ is the earliest finish time so no activity can have a finish time less than $s_1$.

**Theorem:** Consider any nonempty subproblem $S_k = \{a_i \in S : s_i \geq f_k\}$, and let $a_m$ be an activity in $S_k$ with the earliest finishing time. Then $a_m$ is included in some maximum-size subset of mutually compatible activities of $S_k$.

**Proof:** Let $A_k$ be a maximum-size subset of mutually compatible activities in $S_k$, and let $a_j$ be the activity in $A_k$ with the earliest finishing time. If $a_j = a_m$, we are done, since we have shown that $a_m$ is in some maximum-size subset of mutually compatible activities of $S_k$. If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be $A_k$, but substituting $a_m$ for $a_j$. The activities in $a'_k$ are disjoint because the activities in $A_k$ are disjoint, $a_j$ is the first activity in $A_k$ to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that $A'_k$ is a maximum-size subset of mutually compatible activities of $S_k$, and includes $a_m$.

Note: we have to consider the case that $a_j \neq a_m$ because there may be many optimal solutions to the problem.

Greedy algorithms are often top-down: make a choice and then solve a subproblem, rather than the dynamic programming bottom-up approach of solving subproblems before making a choice.

---

Recursive-Activity-Selector $(s, f, k, n)$

```
1: m = k + 1
2: while m ≤ n and s[m] < f[k] do
3:     m = m + 1
4: end while
5: if m ≤ n then
6:     return {a_m} ∪ Recursive-Activity-Selector (s, f, m, n)
7: else
8:     return ∅
9: end if
```

---

Each activity is considered once, running time $\Theta(n)$.

---

Greedy-Activity-Selector $(s, f)$

```
1: n = s.length
2: A = {a_1}
3: k = 1
4: for m = 2 to n do
5:     if s[m] ≥ f[k] then
6:         A = A ∪ {a_m}
7:         k = m
8:     end if
9: end for
10: return A
```

---

**Our Approach to Activity-Selector**

1. Determine optimal substructure.
2. Develop recursive solution
3. Show that greedy choice results in one subproblem.
4. Prove it is always safe to make the greedy choice.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative one.

**More General Approach**

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

**Greedy-choice property:** we can assemble a globally optimal solution by making locally optimal (greedy) choices. In dynamic programming, the choice depends on the solutions to subproblems. In greedy algorithms we make the choice that seems best, then solve the subproblem that remains. The choice may depend on choices made so far, but does not depend on any future choices or solutions to subproblems.

We must prove that the greedy choice yields a globally optimal solution. Typically, the proof examines a globally optimal solution to some subproblem and then shows how to modify the solution to substitute the greedy for some other choice, resulting in a similar, but smaller, subproblem.

We can often find the greedy choice more efficiently than when we have to consider a wider set of choices. Often we can perform a sort or use a priority queue.

**Optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems. Argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem. This scheme implicitly uses induction on the subproblems to prove that making the greedy choice at every step produces an optimal solution.

$0-1$ **Knapsack Problem:** A thief robbing a store finds $n$ items. The $i$th item is worth $v_i$ dollars and weighs $w_i$ pounds, where $v_i$ and $w_i$ are integers. The thief wants to take as valuable a load as possible, but he can carry at most $W$ pounds in his knapsack, for some integer $W$. Which items should he take?

**Fractional Knapsack Problem:** Thief can take fractions of items instead of making a binary choice for taking an item.

Both problems have optimal substructure. For $0-1$ if we have an optimal solution and remove item $j$ we have a subproblem with max weight $W - w_j$ and $n-1$ items to choose from. For fractional if we remove a weight of $w$ from some item $j$, we have a subproblem with max weight $W - w$.

The problems are similar, but the fractional problem has a greedy solution: compute value per pound $v_i/w_i$. Take as much as possible of the most valuable per pound item, and then solve the remaining subproblem.

The greedy strategy does not work for $0-1$, which requires dynamic programming. (Example: 3 items $v = \{60, 100, 120\}$, $w = \{10, 20, 30\}$, $W = 50$.)

**Huffman Codes** We consider the problem of designing a **binary character code** in which each character is represented by a unique binary string, which we call a **codeword**.

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| Frequency (in thousands) | 45 | 13 | 12 | 16 | 9 | 5 |
| Fixed-length codeword | 000 | 001 | 010 | 011 | 100 | 101 |
| Variable-length codeword | 0 | 101 | 100 | 111 | 1101 | 1100 |

Suppose we have a 100,000-character file. If we use a **fixed-length code**, we need 3 bits to represent 6 characters, requiring 300,000 bits to code the entire file.

A **variable-length code** can do better: $(45 \cdot 1 + 13 \cdot 3|12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000$ bits.

We consider only **prefix codes**—codes in which no codeword is also a prefix of some other codeword. Such codes make decoding simpler and can always achieve optimal compression. In our example, the string 001011101 parses uniquely as $0 \cdot 0 \cdot 101 \cdot 1101$ or aabe.

We can use a binary tree where the leaves contain the characters to represent the code. A '0' means go left, a '1' means go right.

An optimal code for a file always results in a **full** tree, in which every nonleaf node has two children. If $C$ is the alphabet, then the tree has $|C|$ leaves and $|C| - 1$ internal nodes.

**The number of bits to encode a file:** Let $T$ be a tree corresponding to a prefix code. For each character $c \in C$, let $c.freq$ denote the frequency of $c$ in the file and $d_T(c)$ denote the depth of $c$'s leaf (also the length of the codeword). Then

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c),$$

we call this the **cost** of $T$.

---

Huffman-Code $(s, f)$

1: $n = |C|$
2: $Q = C$
3: **for** $i = 1$ **to** $n - 1$ **do**
4: allocate a new node $z$
5: $z.left = x =$ Extract-Min($Q$)
6: $z.right = y =$ Extract-Min($Q$)
7: $z.freq = x.freq + y.freq$
8: Insert($Q, z$)
9: **end for**
10: **return** Extract-Min($Q$)

---

**Huffman code:** $C$ is a set of $n$ characters and each character $c \in C$ has frequency $c.freq$. We build $T$ in a bottom up manner. Starts with $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations. Uses a min-priority queue $Q$ keyed on $freq$ to identify the two least-frequent objects to merge together. $O(n)$ time to build heap. Each iteration of loop takes $O(\lg n)$ time and the loop has $n - 1$ iterations, for a running time of $O(n \lg n)$.

**Greedy-choice Property** Let $C$ be an alphabet in which each character $c \in C$ has frequency $c.freq$. 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:** The idea of the proof is to take the tree $T$ representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters $x$ and $y$ appear as sibling leaves of maximum depth in the new tree. If we can construct such a tree, then the codewords for $x$ and $y$ will have the same length and differ only in the last bit.

Let $a$ and $b$ be two characters that are sibling leaves of maximum depth in $T$. WOLOG, we assume that $a.freq \leq b.freq$ and $x.freq \leq y.freq$. Since $x.freq$ and $y.freq$ are the two lowest leaf frequencies, in order, and $a.freq$ and $b.freq$ are two arbitrary frequencies, in order, we have $x.freq \leq a.freq$ and $y.freq \leq b.freq$.

In the remainder of the proof, it is possible that we could have $x.freq = a.freq$ or $y.freq = b.freq$. However, if we had $x.freq = b.freq$, then we would also have $a.freq = b.freq = x.freq = y.freq$ and the lemma would be trivially true. Thus, we will assume that $x.freq \neq b.freq$, which means that $x \neq b$.

We exchange the positions in $T$ of $a$ and $x$ to produce $T'$, and then we exchange the positions in $T'$ of $b$ and $y$ to produce tree $T''$. in which $x$ and $y$ are sibling leaves of maximum depth. (Note that if $x = b$ but $y \neq a$, then $T''$ does not have $x$ and $y$ as sibling leaves of maximum depth. Because we assume $x \neq b$, this situation cannot occur.) The difference in cost between $T$ and $T'$ is

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\ &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\ &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\ &= (a.freq - x.freq)(d_T(a) - d_T(x)) \geq 0 \end{aligned}$$

because both $a.freq - x.freq$ and $d_T(a) - d_T(x)$ are nonnegative. More specifically, $a.freq - x.freq$ is nonnegative because $x$ is a minimum-frequency leaf, and $d_T(a) - d_T(x)$ is nonnegative because $a$ is a leaf of maximum depth in $T$. similarly, exchanging $y$ and $b$ does not increase the cost, and so $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T)$, and since $T$ is optimal, we have $B(T) \leq B(T'')$, which implies $B(T'') = B(T)$. Thus $T''$ is an optimal tree in which $x$ and $y$ appear as sibling leaves of maximum depth, from which the lemma follows.

We can view the cost of a single merger as being the sum of the frequencies of the two items being merged. Of all possible mergers at each step, Huffman chooses the one that incurs the least cost. (The total cost of the tree constructed equals the sum of the costs of its mergers.)

**Optimal-substructure Property:** Let $C$ be a given alphabet with frequence $c.freq$ defined for each character $c \in C$. Let $x$ and $y$ be two characters in $C$ with minimum frequency. Let $C'$ be the alphabet with the characters $x$ and $y$ removed and a new character $z$ added, so that $C' = C - \{x, y\} \cup \{z\}$. Define $freq$ for $C'$ as for $C$, except that $z.freq = x.freq + y.freq$. Let $T'$ be any tree representing an optimal prefix code for the alphabet $C'$. Then the tree $T$, obtained from $T'$ by replacing the leaf node for $z$ with an internal node having $x$ and $y$ as children, represents an optimal prefix code for the alphabet $C$.

**Proof:** We first show how to express the cost $B(T)$ of tree $T$ in terms of the cost $B(T')$ of tree $T'$, by considering the component costs. For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have $x.freq \cdot d_T(x) + y.freq \cdot d_T(y) = (x.freq + y.freq)(d_{T'}(z) + 1) = z.freq \cdot d_{T'}(z) + (x.freq + y.freq)$, from which we conclude that $B(T) = B(T') + x.freq + y.freq$ or $B(T') = B(T) - x.freq - y.freq$.

We now prove the lemma by contradiction. Suppose that $T$ does not represent an optimal prefix code for $C$. Then there exists an optimal tree $T''$ such that $B(T'') < B(T)$. WOLOG, $T''$ has $x$ and $y$ as siblings. Let $T'''$ be the tree $T''$ with the common parent of $x$ and $y$ replaced by a leaf $z$ with frequency $z.feq - x.freq + y.freq$. Then $B(T''') = B(T'') - x.freq - y.freq < B(T) - x.freq - y.freq = B(T')$. Yielding a contradiction to the assumption that $T'$ represents an optimal prefix code for $C'$. Thus $T$ must represent an optimal prefix code for the alphabet $C$.

**Matroids:** A **matroid** is an ordered pair $M = (S, \mathcal{I})$ satisfying the following conditions:

1. $S$ is a finite set.
2. $\mathcal{I}$ is a nonempty family of subsets of $S$, called the **independent** subsets of $S$, such that if $B \in \mathcal{I}$ and $A \subset B$, then $A \in \mathcal{I}$. We say that $\mathcal{I}$ is **hereditary** if it satisfies this property. Note that the empty set is necessarily a member of $\mathcal{I}$.
3. If $A \in I, B \in I$, and $|A| < |B|$, then there exists some element $x \in B - A$ s.t. $A \cup \{x\} \in \mathcal{I}$. We say that $M$ satisfies the **exchange property**.

"Matroid" is due to Hassler Whitney who was studying matric matroids, in which elements of $S$ are rows of a given matrix and a set of rows are independent if they are linearly independent.

The **graphic matroid** $M_G = (S_G, \mathcal{I}_G)$ is defined in terms of an undirected graph $G = (V, E)$ as follows:

- The set $S_G$ is defined to be $E$.
- If $A$ is a subset of $E$, then $A \in \mathcal{I}_G$ iff $A$ is acyclic. That is, a set of edges $A$ is independent iff the subgraph $G_A = (V, A)$ forms a forest.

**Theorem:** If $G = (V, E)$ is an undirected graph, then $M_G = (S_G, \mathcal{I}_G)$ is a matroid.

**Proof:** Clearly, $S_G = E$ is a finite set. Furthermore, $\mathcal{I}_G$ is hereditary, since a subset of a forest is a forest. Removing edges from an acyclic set of edges cannot create cycles.

Exchange property: Suppose that $G_A = (V, A)$ and $G_B = (V, B)$ are forests of $G$ and that $|B| > |A|$. ($A$ and $B$ are acyclic sets of edges, and $B$ contains more edges than $A$.)

We claim that a forest $F = (V_F.E_F)$ contains exactly $|V_F| - |E_F|$ trees. Suppose $F$ consists of $t$ trees, where the $i$th tree contains $v_i$ vertices and $e_i$ edges. Then,

$$|E_F| = \sum_{i=1}^{t} e_i = \sum_{i=1}^{t} (v_i - 1) = \sum_{i=1}^{t} v_i - t = |V_F| - t,$$

which implies that $t = |V_F| - |E_F|$. The, forest $G_A$ contains $|V| - |A|$ trees, and forest $G_B$ contains $|V| - |B|$ trees.

Since forest $G_B$ has fewer trees than forest $G_A$ does, forest $G_B$ must contain some tree $T$ whose vertices are in two different trees in forest $G_A$. Moreover, since $T$ is connected, it must contain an edge $(u, v)$ s.t. vertices $u$ and $v$ are in different trees in forest $G_A$. Since the edge $(u, v)$ connects vertices in two different trees in forest $G_A$, we can add the edge $(u, v)$ to forest $G_A$ without creating a cycle. Therefore, $M_G$ satisfies the exchange property and $M_G$ is a matroid.

Given a matroid $M - (S, \mathcal{I})$, we call an element $x \notin A$ an **extension** of $A \in \mathcal{I}$ if we can add $x$ to $A$ while preserving independence; formally, $x$ is an extension of $A$ if $A \cup \{x\} \in \mathcal{I}$. Consider a graphic matroid $M_G$. If $A$ is an independent set of edges, then edge $e$ is an extension of $A$ iff $e$ is not in $A$ and the addition of $e$ does not create a cycle. We say that $A$ is **maximal** if it has no extensions. $A$ is maximal if it is not included in any larger independent subset of $M$.

**Theorem:** All maximal independent subsets in a matroid have the same size.

**Proof:** Suppose to the contrary that $A$ is a maximal independent subset of $M$ and there exists another larger maximal independent subset $B$ of $M$. Then, the exchange property implies that for some $x \in B - A$, we can extend $A$ to a larger independent set $A \cup \{x\}$, contradicting the assumption that $A$ is maximal.

Illustration: Consider $M_G$ every maximal independent set (**spanning tree**) has $|V| - 1$ edges and connects all vertices of $G$.

We say that matroid $M$ is **weighted** if it is associated with a weight function $w$ that assigns a strictly positive weight $w(x)$ to each element $x \in S$. The weight function extends to subsets of $S$ by summation:

$$w(A) = \sum_{x \in A} w(x)$$

for any $A \subset S$. For example, if we let $w(e)$ denote the weight of an edge $e$ in a graphic matroid, then $w(A)$ is the total weight of the edges in the edge set $A$.

**Greedy algorithms on a weighted matroid:** Many problems amenable to greedy solution can be formulated as finding a maximum weight independent subset in a weighted matroid. (Given $M = (S, \mathcal{I})$ find an independent set $A \in \mathcal{I}$ such that $w(A)$ is maximized. Independent and maximum weight =¿ **optimal**.) Because weights are positive, an optimal subset is always a maximal independent set.

**Minimum-spanning-tree problem:** Given a graph $G = (V, E)$ and a *length* function $w(e)$ is the positive length of the edge, find a subset of edges that connects all the vertices and has minimum total length. Cast as finding the optimal subset of a matroid, let the weight of the matroid $M_G$ be $w'(e) = w_0 - w(e)$.

$$w'(A) = \sum_{e \in A} w'(e) = \sum_{e \in A} (w_0 - w(e)) = (|V| - 1)w_0 - \sum_{e \in A} w(e) = (|V - 1|)w_0 - w(A).$$

---

Greedy $(M, w)$

1: $A = \emptyset$
2: sort $M.S$ into monotonically decreasing order by weight $w$
3: **for** $x \in M.S$ **do**
4: **if** $A \cup \{x\} \in M.\mathcal{I}$ **then**
5: $A = A \cup \{x\}$
6: **end if**
7: **end for**
8: **return** $A$

---

Empty set is independent. Each iteration of the for loop maintains independence. A is always independent by induction.

Running time: let $n = |S|$, sort takes $n \lg n$, if each execution of line 4 take $O(f(n))$ time, the entire algorithm is $O(n \lg n + nf(n))$.

**Lemma (Matroids exhibit the greedy-choice property):** Suppose that $M = (S, \mathcal{I})$is a weighted matroid with weight function $w$ and that $S$ is sorted into monotonically decreasing order by weight. Let $x$ be the first element of $S$ such that $\{x\}$ is independent, if any such $x$ exists. If $x$ exists, then there exists an optimal subset $A$ of $S$ that contains $x$.

**Proof:** If no $x$ exists, then the only independent subset is the empty set and the lemma is vacuously true. Otherwise, let $B$ be any nonempty optimal subset. Assume that $x \notin B$; otherwise letting $A = B$ gives an optimal subset of $S$ that contains $x$.

No element of $B$ has weight greater than $w(x)$. To see why, observe that $y \in B$ implies that $\{y\}$ is independent, since $B \in \mathcal{I}$ and $\mathcal{I}$ is hereditary. Our choice of $x$ therefore ensures that $w(x) \geq w(y)$ for any $y \in B$.

Construct $A$ as follows, Begin with $A = \{x\}$. By the choice of $x$, set $A$ is independent. Using the exchange property, repeatedly find a new element of $B$ that we can add to $A$ until $|A| = |B|$, while preserving the independence of $A$. At that point $A$ and $B$ are the same except that $A$ has $x$ and $B$ has some other element $y$. That is $A = B - \{y\} \cup \}x\}$ for some $y \in B$, and so

$$w(A) = w(B) - w(y) + w(x) \geq w(B).$$

Because set $B$ is optimal, set $A$, which contains $x$, must also be optimal.

**Lemma:** Let $M = (S, \mathcal{I})$be any matroid. If $x$ is an element of $S$ that is an extension of some independent subset $A$ of $S$, then $x$ is also an extension of $\emptyset$.

**Proof:** Since $x$ is an extension of $A$, we have the $A \cup \{x\}$ is independent. Since $\mathcal{I}$ is hereditary, $\{x\}$ must be independent. Thus $x$ is an extension of $\emptyset$.

**Corollary:** Let $M = (S, \mathcal{I})$be any matroid. If $x$ is an element of $S$ such that $x$ is not an extension of $\emptyset$, then $x$ is not an extension of any independent subset $A$ of $S$.

**Proof:** The corollary is simply the contrapositive of the previous lemma.

The corollary says that any element that cannot immediately be used can never be used. Therefore, Greedy cannot make an error by passing over any initial elements in $S$ that are not an extension of $\emptyset$ because these elements can never be used.

**Lemma (Matroids exhibit the optimal substructure property):** Let $x$ be the first element of $S$ chosen by Greedy for the weighted matroid $M = (S, \mathcal{I})$. The remaining problem of finding a maximum-weight independent subset containing $x$ reduces to finding a maximum-weight independent subset of the weighted matroid $M' = (S', \mathcal{I}')$, where

$$S' = \{y \in S : \{x, y\} \in \mathcal{I}\},$$

$$\mathcal{I}' = \{B \subset S - \{x\} : B \cup \{x\} \in \mathcal{I}\},$$

and the weight function for $M'$ is the weight function for $M$, restricted to $S'$. (We call $M'$ the **contraction** of $M$ by the element $x$.)

**Proof:** If $A$ is any maximum-weight independent subset of $M$ containing $x$, then $A' = A - \{x\}$ is an independent subset of $M'$. Conversely, any independent subset $A'$ of $M'$ yields an independent subset $A = A' \cup \{x\}$ of $M$. Since we have in both cases that $w(A) = w(A') + w(x)$, a maximum-weight solution in $M$ containing $x$ yields a maximum-weight solution in $M'$ and vice versa.

**Theorem:** If $M = (S, \mathcal{I})$is a weighted matroid with weight function $w$, then Greedy$(M, w)$ returns an optimal subset.

**Proof:** By the corollary, any elements that Greedy passes over initially because they are not extensions of $\emptyset$ can be forgotten about, since they can never be useful. Once Greedy selects the first element $x$, the greedy choice property lemma implies that the algorithm does not err by adding $x$ to $A$, since there exists an optimal subset containing $x$. Finally the optimal-substructure lemma implies that the remaining problem is one of finding an optimal subset of matroid $M'$ that is the contraction of $M$ by $x$. After the procedure Greedy sets $A$ to $\{x\}$, we can interpret all of its remaining steps as acting in the matroid $M' = (S', \mathcal{I}')$, because $B$ is independent in $M'$ if and only if $B \cup \{x\}$ is independent in $M$, for all sets $B \in \mathcal{I}'$. Thus, the subsequent operation of Greedy will find a maximum-weight independent subset for $M'$, and the overall operation of Greedy will find a macimum-weight independent subset for $M$.

**Unit Time Task Scheduling** The problem of scheduling unit-time tasks with deadlines and penalties for a single processor has the following inputs:

- a set $S = \{a_1, a_2, \ldots, a_n\}$ of $n$ unit-time tasks
- a set of $n$ integer deadlines $d_1, d_2, \ldots, d_n$, such that $d_i satisfies 1 \leq d_i \leq n$ and task $a_i$ is supposed to finish by time $d_i$
- a set of $n$ nonnegative **penalties** $w_1, w_2 \ldots, w_n$ such that we incur a penalty of $w_i$ if task $a_i$ is not finished by time $d_i$, and we incur no penalty if a task finished by its deadline.

We wish to find a schedule $S$ that minimizes the total penalty incurred for missed deadlines.

We do this by casting the problem as a matroid.

First, we come up with a **canonical form** for an arbitrary schedule where the early tasks come before the late tasks and we schedule the early tasks in order of monotonically increasing deadlines. We can do this because if an early

task is scheduled after a late task the two can be swapped and the early task is still early, the late task will still late (early-first form). If $a_i$ is scheduled before $a_j$ and $d_j < d_i$ we can swap them: moving $a_j$ earlier will not make it late and $a_i$ will not be late if $a_j$ was not late.

We say a set of tasks is independent if there exists a schedule for the tasks such that no task is late.Let $\mathcal{I}$ denote the set of all independent sets of tasks.

How do we tell if a set of tasks is independent? For $t = 0, 1, 2, \ldots, n$, let $N_t(A)$ denote the number of tasks in $A$ whose deadline is $t$ or earlier.

**Lemma** For any set of tasks $A$, the following statements are equivalent.

1. The set $A$ is independent
2. For $t = 0, 1, 2, \ldots, n$, we have $N_t(A) \leq t$
3. If the tasks in $A$ are scheduled in order of monotonically increasing deadlines, then no task it late.

**Proof:** To show that (1) implies (2), we prove the contrapositive: if $N_t(A) > t$ for some $t$, then there is no way to make a schedule with no late tasks for set $A$, because more than $t$ tasks must finish before time $t$. Therefore (1) implies (3). If (2) holds, then (3) must follow: there is no way to "get stuck" when scheduling the tasks in order of monotonically increasing deadlines, since (2) implies that the $i$th largest deadline is at least $i$. Finally, (3) trivially implies (1).

The problem of minimizing the sum of the penalties is the same as the problem of maximizing the penalties of the early tasks.

**Theorem** If $S$ is a set of unit-time tasks with deadlines, and $\mathcal{I}$ si the set of all independent sets of tasks, then the corresponding system $(S, \mathcal{I})$ is a matroid.

**Proof:** Every subset of an independent set of tasks is certainly independent. To prove the exchange property, suppose the $B$ and $A$ are independent sets of tasks and that $|B| > |A|$. Let k be the largest $t$ such that $N_t(B) \leq N_t(A)$. (Such a value of $t$ exists, since $N_0(A) = N_0(B) = 0$.) Since $N_n(B) = |B|$ and $N_n(A) = |A|$, but $|B| > |A|$, we must have that $k < n$ and that $N_j(B) > N_j(A)$ for all $j$ in the range $k+1 \leq j \leq n$. Therefore, $B$ contains more tasks with deadline $k + 1$ than $A$ does. Let $a_i$ be a task in $B - A$ with deadline $k + 1$. Let $A' = A \cup \{a_i\}$.

We now show that $A'$ must be independent by using property 2 of the lemma. For $0 \leq t \leq k$, we have $N_t(A') = N_t(A) \leq t$, since $A$ is independent. For $k < t \leq n$, we have $N_t(A') \leq N_t(B) \leq t$, since $B$ is independent. Therefor, $A'$ is independent, completing our proof that $(S, \mathcal{I})$ is a matroid.