1 Greedy Algorithms

Suppose we want to solve a problem, and we’re able to come up with some recursive formulation of the problem that would give us a nice dynamic programming algorithm. But then, upon further inspection, we notice that any optimal solution only depends on looking up the optimal solution to one other subproblem. A greedy algorithm is an algorithm which exploits such a structure, ignoring other possible choices. Greedy algorithms can be seen as a refinement of dynamic programming; in order to prove that a greedy algorithm is correct, we must prove that to compute an entry in our table, it is sufficient to consider at most one other table entry; that is, at each point in the algorithm, we can make a “greedy”, locally-optimal choice, and guarantee that a globally-optimal solution still exists. Instead of considering multiple choices to solve a subproblem, greedy algorithms only consider a single subproblem, so they run extremely quickly – generally, linear or close-to-linear in the problem size.

Unfortunately, greedy algorithms do not always give the optimal solution, but they frequently give good (approximate) solutions. To give a correct greedy algorithm one must first identify optimal substructure (as in dynamic programming), and then argue that at each step, you only need to consider one subproblem. That is, even though there may be many possible subproblems to recurse on, given our selection of subproblem, there is always an optimal solution that contains the optimal solution to the selected subproblem.

1.1 Activity Selection Problem

One problem, which has a very nice (correct) greedy algorithm, is the Activity Selection Problem. In this problem, we have a number of activities. Your goal is to choose a subset of the activities to participate in. Each activity has a start time and end time, and you can’t participate in multiple activities at once. Thus, given \(n\) activities \(a_1, a_2, ..., a_n\) where \(a_i\) has start time \(s_i\) and finish time \(f_i\), we want to find a maximum set of non-conflicting activities.

The activity selection problem has many applications, most notably in scheduling jobs to run on a single machine.

1.1.1 Optimal Substructure

Let’s start by considering a subset of the activities. In particular, we’ll be interested in considering the set of activities \(S_{i,j}\) that start after activity \(a_i\) finishes and end before activity \(a_j\) starts. That is, \(S_{i,j} = \{a_k \mid f_i \leq s_k, f_k \leq s_j\}\). We can participate in these activities between \(a_i\) and \(a_j\). Let \(A_{i,j}\) be a maximum subset of non-conflicting activities from the subset \(S_{i,j}\). Our first intuition would be to approach this by using dynamic programming. Suppose some \(a_k \in A_{i,j}\), then we can break down the optimal subsolution \(A_{i,j}\) as follows

\[
|A_{i,j}| = 1 + |A_{i,k}| + |A_{k,j}|
\]

where \(A_{i,k}\) is the best set for \(S_{i,k}\) (before \(a_k\)), and \(A_{k,j}\) is the best set for after \(a_k\). Another way of interpreting this expression is to say “once we place \(a_k\) in our optimal set, we can only consider optimal solutions to subproblems that do not conflict with \(a_k\).”

Thus, we can immediately come up with a recurrence that allows us to come up with a dynamic programming algorithm to solve the problem.

\[
|A_{i,j}| = \max_{a_k \in S_{i,j}} (1 + |A_{i,k}| + |A_{k,j}|)
\]
This problem requires us to fill in a table of size $n^2$, so the dynamic programming algorithm will run in $\Omega(n^2)$ time. The actual runtime is $O(n^3)$ since filling in a single entry might take $O(n)$ time.

But we can do better! We will show that we only need to consider the $a_k$ with the smallest finishing time, which immediately allows us to search for the optimal activity selection in linear time.

**Claim 1.** For each $S_{i,j}$, there is an optimal solution $A_{i,j}$ containing $a_k \in S_{i,j}$ of minimum finishing time $f_k$.

Note that if the claim is true, when $f_k$ is minimum, then $A_{i,k}$ is empty, as no activities can finish before $a_k$; thus, our optimal solution only depends on one other subproblem $A_{k,j}$ (giving us a linear time algorithm).

Here, we prove the claim.

**Proof.** Let $a_k$ be the activity of minimum finishing time in $S_{i,j}$. Let $A_{i,j}$ be some maximum set of non-conflicting activities. Consider $A'_{i,j} = A_{i,j} \setminus \{a_l\} \cup \{a_k\}$ where $a_l$ is the activity of minimum finishing time in $A_{i,j}$. It’s clear that $|A'_{i,j}| = |A_{i,j}|$. We need to show that $A'_{i,j}$ does not have conflicting activities. We know $a_l \in A_{i,j} \subset S_{i,j}$. This implies $f_l \geq f_k$, since $a_k$ has the minimum finishing time in $S_{i,j}$.

All $a_t \in A_{i,j} \setminus \{a_l\}$ don’t conflict with $a_l$, which means that $s_t \geq s_l$, which means that $s_t \geq f_k$, so this means that no activity in $A_{i,j} \setminus \{a_l\}$ can conflict with $a_k$. Thus, $A'_{i,j}$ is an optimal solution. □

Due to the above claim, the expression for $A_{i,j}$ from before simplifies to the following expression in terms of $a_k \subseteq S_{i,j}$, the activity with minimum finishing time $f_k$.

\[
|A_{i,j}| = 1 + |A_{k,j}|
A_{i,j} = A_{k,j} \cup \{a_k\}
\]

Algorithm Greedy-AS assumes that the activities are presorted in nondecreasing order of their finishing time, so that if $i < j$, $f_i \leq f_j$.

**Algorithm 1:** Greedy-AS(a)

\[
A \leftarrow \{a_1\} \quad // \text{activity of min } f_i
k \leftarrow 1
\]

for $m = 2 \rightarrow n$ do

\[
\text{if } s_m \geq f_k \text{ then}
\]

\[
// a_m \text{ starts after last activity in } A
A \leftarrow A \cup \{a_m\}
\]

\[
k \leftarrow m
\]

return $A$

By the above claim, this algorithm will produce a legal, optimal solution via a greedy selection of activities. There may be multiple optimal solutions, but there always exists a solution that includes $a_k$ with the minimum finishing time. The algorithm does a single pass over the activities, and thus only requires $O(n)$ time – a dramatic improvement from the trivial dynamic programming solution. If the algorithm also needed to sort the activities by $f_i$, then its runtime would be $O(n \log n)$ which is still better than the original dynamic programming solution.

### 1.2 Scheduling

Consider another problem that can be solved greedily. We are given $n$ jobs which all need a common resource. Let $w_j$ be the weight (or importance) and $l_j$ be the length (time required) of job $j$. Our output is an ordering of jobs. We define the completion time $c_j$ of job $j$ to be the sum of the lengths of jobs in the ordering up to and including $l_j$. Our goal is to output an ordering of jobs that minimizes the weighted sum of completion times $\sum_j w_j c_j$. 
1.2.1 Intuition

Our intuition tells us that if all jobs have the same length, then we prefer larger weighted jobs to appear earlier in the order. If jobs all have equal weights, then we prefer shorter length jobs in the order.

\[
\begin{array}{|c|c|c|}
\hline
1 & 2 & 3 \\
\hline
\end{array}
\]

vs

\[
\begin{array}{|c|c|c|}
\hline
3 & 2 & 1 \\
\hline
\end{array}
\]

In the first case, assuming they all have equal weights of 1, \(\sum_{i=1}^{3} w_i c_i = 1 + 3 + 6 = 10\). In the second case, \(\sum_{i=1}^{3} w_i c_i = 3 + 4 + 6 = 13\).

1.2.2 Optimal Substructure

What do we do in the cases where \(l_i < l_j\) and \(w_i < w_j\)? Consider the optimal ordering of jobs. Suppose we have a job \(i\) that is followed by job \(j\) in the optimal order. Consider swapping jobs \(i\) and \(j\). The example below swaps jobs 1 and 2.

\[
\begin{array}{|c|c|c|}
\hline
l_1 & l_2 & l_2 \\
\hline
1 & 2 & 1 \\
\hline
\end{array}
\]

Note that swapping jobs \(i\) and \(j\) does not alter the completion times for every other job and only changes the completion times for \(i\) and \(j\). \(c_i\) increases by \(l_j\) and \(c_j\) decreases by \(l_i\). This means that our objective function \(\sum_i w_i c_i\) changes by \(w_i l_j - w_j l_i\). Since we assumed we our order was optimal originally, our objective function has to increase because our order is no longer optimal due to the swapping. This means,

\[w_i l_i - w_j l_j \geq 0\]

which implies,

\[\frac{l_i}{w_i} \geq \frac{l_j}{w_j}\]

Therefore, we want to process jobs \(i\) and \(j\) in increasing order of \(\frac{l_i}{w_i}\), the ratio of the length to the weight of each job. The algorithm also does a single pass over jobs, and thus only requires \(O(n)\) time, assuming the jobs were ordered by \(\frac{l_i}{w_i}\). Like previously, if the algorithm also needed to sort the jobs based on the ratio of length to weight, then its runtime would be \(O(n \log n)\).

1.3 Optimal Codes

Our third example comes from the field of information theory. In ASCII, there is a fixed 8 bit code for each character. Suppose we want to incorporate information about frequencies of characters to obtain shorter encodings. What if we want to represent characters by codes of different lengths depending on each character’s frequencies? We explore a greedy solution to find the optimal encoding of characters.

To create optimal codes, we want a way to encode and decode our sequence. To encode the sequence, we would just have to concatenate the code of each character together. How about for decoding? Consider the following codes of characters: \(a \rightarrow 0, b \rightarrow 1, c \rightarrow 01\). However, when decoding, when we encounter 01, this could be decoded as ”ab” or ”c”. Therefore, our codes need to be prefix free: no codeword is a prefix of another.
1.3.1 Tree Representation

It is simple to think of representing our codes in a tree structure, where the codewords represent the leaves of our tree.

The code for each character can be found by concatenating the bits of the path from the root to the leaves. By convention, every left branch is given the bit 0 and every right branch is given the bit 1.

1.3.2 Huffman Codes

In 1951, David A. Huffman, in his MIT information theory class, was given the choice of a term paper or final exam. Huffman chose to do the term paper rather than take the final exam. He quickly proved a method to find the most efficient binary code, which we know today as Huffman codes.

The basic idea is this: build subtrees for subsets of characters and merge them from the bottom up, combining the two trees with the characters of minimum total frequency.

Input: Set of characters $C = \{c_1, c_2, \ldots, c_n\}$ of size $n$, and $F = \{f(c_1), f(c_2), \ldots, f(c_n)\}$, a set of frequencies.

1. Create nodes for each character $c_k$, labeled by their frequencies $f(c_k)$.
2. Find the two nodes $c_i$ and $c_j$ with the minimum frequencies and create a new intermediate node $I$ with $c_i$ and $c_j$ as its children.
3. Label the new node $I$ to have the frequency equal to the sum of frequencies of its children.
4. Repeat steps 2-3 until all characters are part of the tree.

Figure 1: The top level of the Huffman Coding algorithm.

1.3.3 Proof of Correctness

We are given a set of characters $C$ and a set of its associated frequencies $F$ where $f(c)$ is the frequency of character $c$. Let $x$ and $y$ be the characters with the two smallest frequencies.

Claim: There exists an optimal coding tree for $C$ such that $x, y$ are siblings in the tree.
Proof of Claim: Let $T$ be the optimal coding tree for $C$. The optimal coding tree must be a full binary tree, that is, every non-leaf node must have two children. Let $a, b$ be characters that are sibling leaves of maximum depth. We define the number of bits to encode $c$ as $d_T(c)$ and the number of bits needed for the coding tree as $B(T) = \sum_c f(c)d_T(c)$.

We can replace $a, b$ by $x, y$ without increasing the total number of bits needed for the coding tree. If we swap $x$ and $a$, the change in cost becomes

$$f(x)d_T(a) + f(a)d_T(x) - f(x)d_T(x) - f(a)d_T(a) = (f(x) - f(a))(d_T(a) - d_T(x)) \leq 0$$

Therefore, there swapping $a, b$ with $x, y$ will not increase our objective function $B(T)$. Hence, there exists an optimal coding tree where $x, y$ are siblings in the tree.