In the last two lectures, we explored how the divide and conquer technique is useful for designing algorithms for many different problems. Today, we start examining another technique: greedy algorithms.

1. Greedy algorithms

A greedy algorithm is one in which we:

1. Break down a problem into a sequence of decisions that need to be made, then
2. Make the decisions one at a time, each time choosing the option that is optimal at the moment (and not worrying about later decisions).

This is one of the simplest algorithm design techniques around, yet as we will see it yields efficient algorithm for many different problems. But the challenge with this technique in many instances is proving the algorithm’s correctness—or, often, determining whether the algorithm is correct or not in the first place.

You have already seen one classic greedy algorithm in CS 240: Huffman codes are optimal prefix codes obtained by running the greedy algorithm to join trees with different frequencies together. This algorithm is both correct and efficient.

We also use a greedy algorithm in real life when we make change.

**Problem 1** (Making change). Given coins with denominations $d_1 > d_2 > \cdots > d_n = 1$ and a value $v \geq 1$, determine the minimum number of coins whose denominations sum to $v$. (I.e., a valid solution $a_1, \ldots, a_n \in \mathbb{N}$ is one that satisfies $\sum_{i=1}^{n} a_i d_i = v$ and minimizes $\sum_{i=1}^{n} a_i$.)

For example, we have coins worth 200, 100, 25, 10, 5, and 1 cents in current circulation in Canada. How would you make change for 1.43$ using those coins? You would start by taking a coin with maximal denomination that is at most 143 (here the loonie worth 100 cents), subtract that amount from the total (leaving 43 cents in this case), and repeat.

**Algorithm 1:** GreedyChange($d_1 > d_2 > \cdots > d_n; v$)

```plaintext
for i = 1, \ldots, n do
    a_i \leftarrow \lfloor v / d_i \rfloor;
    v \leftarrow v \bmod d_i;
return (a_1, \ldots, a_n);
```

Does this algorithm always return a valid solution to the Making Change problem? It does for every value $v$ when the coins have denominations 200, 100, 25, 10, 5, and 1 but the proof of correctness in this case is not trivial. In general, however, there are denominations and choices of $v$ where the algorithm does not return the minimal number of coins. Take
for example the case where the denominations are 8, 7, and 1 and the value to return is 14. The greedy algorithm will return the solution (1, 0, 6), for a total of 7 coins, when the solution (0, 2, 0) requires only two coins.

Today, we explore another fundamental problem that can be solved using greedy algorithms.

2. INTERVAL SCHEDULING

Many real-world scheduling problems can be formulated in terms of the abstract interval scheduling problem.

**Definition 6.1** (Interval scheduling problem). Given a set of \( n \) pairs of start and finish times \( (s_1, f_1), \ldots, (s_n, f_n) \) where each pair \( (s_i, f_i) \) satisfies \( s_i < f_i \), find a maximum subset \( I \subseteq \{1, 2, \ldots, n\} \) such that no two intervals in \( I \) overlap. (I.e., for every \( i \neq j \in I \), \( s_i > f_j \) or \( s_j > f_i \).)

It is quite natural to try to use the greedy method to solve this problem: to do so, we just need to decide how the algorithm chooses the “best” interval to add to \( I \) given the intervals that have already been added to this set. We have many possibilities on how to define the notion of “best” interval:

- **Earliest starting time:** Pick the interval with the earliest starting time that does not overlap any of the intervals we already added to \( I \).
- **Earliest finish time:** Pick the interval with the earliest finish time that does not overlap any of the intervals we already added to \( I \).
- **Shortest interval:** Pick the interval with the shortest length \( f_i - s_i \) among those that don’t overlap any of the intervals we already added to \( I \).
- **Minimum conflicts:** Let \( J \) be the set of intervals that don’t overlap with any interval in \( I \). Pick the interval in \( J \) that overlaps with the fewest other number of other intervals in \( J \).

All four options sound perfectly reasonable, but only one yields an algorithm that always outputs a valid solution to the interval scheduling problem.

**Proposition 6.2.** The greedy algorithm with earliest starting time does not always find a valid solution to the interval scheduling problem.

*Proof.* Consider the set of intervals

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The valid solution is \( I = \{2, 3, 4\} \) but the greedy algorithm outputs \( I = \{1\} \) instead. \( \Box \)
Proposition 6.3. The greedy algorithm with shortest interval does not always find a valid solution to the interval scheduling problem.

Proof. Consider the set of intervals

The valid solution is \{1, 2, 3\} but the greedy algorithm outputs \(I = \{4, 5\}\) instead. \(\square\)

Proposition 6.4. The greedy algorithm with minimal conflicts does not always find a valid solution to the interval scheduling problem.

Proof. Consider the set of intervals

The valid solution is \{1, 2, 3, 4\} but the greedy algorithm outputs \(I = \{1, 4, 6\}\) instead. \(\square\)

We are now left with a single candidate algorithm: earliest finish time. We can implement this greedy algorithm in a simple way: sort the intervals by finish time, and when we go through the list we add an interval \((s_i, f_i)\) iff its start time is larger than the last (and therefore most recently added) finish time of any interval in \(I\).

Algorithm 2: GreedyScheduler(((s_1, f_1), \ldots , (s_n, f_n))

\[
\begin{align*}
A & \leftarrow \text{indices} \{1, \ldots , n\} \text{ sorted by } f_i; \\
I & \leftarrow A[1]; \\
f^* & \leftarrow f_{A[1]}; \\
\text{for } i = 2, \ldots , n \text{ do} \\
& \quad \text{if } s_{A[i]} > f^* \text{ then} \\
& \quad \quad I \leftarrow I \cup A[i]; \\
& \quad \quad f^* \leftarrow f_{A[i]}; \\
\text{return } I;
\end{align*}
\]

The time complexity of the GreedyIntervalScheduler is \(\Theta(n \log n)\). It remains to show that it always returns a valid solution to the interval scheduling problem. We do so via an always ahead argument: we show that at every point in the execution of our greedy algorithm, we can complete the partial solution we have obtained so far into a valid solution to the original problem.

As a first step, let us show how we can always run the greedy algorithm to obtain the first interval in a maximum set \(I^*\) of non-overlapping intervals.
**Proposition 6.5.** Let $I^* = \{i_1, i_2, \ldots, i_k\} \subseteq \{1, 2, \ldots, n\}$ be the indices of a maximum set of non-overlapping intervals in some instance of the Interval Scheduling problem sorted by finish times so that $f_{i_1} < f_{i_2} < \cdots < f_{i_k}$. Let $j \in [n]$ be the index of the first interval selected by the GreedyIntervalScheduler. Then $I^j = \{j, i_2, i_3, \ldots, i_k\}$ is also a maximum set of non-overlapping intervals.

**Proof.** The set $I^j$ must have the same cardinality as $I^*$ since $f_j \leq f_{i_1} < f_{i_2}$ for each $\ell \in \{2, 3, \ldots, k\}$ and so $j \notin \{i_2, \ldots, i_k\}$. We need to show that $I^j$ is a set of non-overlapping intervals. Since $I^*$ is a set of non-overlapping intervals, the only thing we need to show is that the interval $j$ does not overlap with any of the intervals $i_2, \ldots, i_k$.

The fact that $I^*$ contains non-overlapping intervals and that $f_{i_1}$ is the smallest finish time implies that we must have $f_{i_1} < s_{i_\ell}$ for each $\ell = 2, 3, \ldots, k$. And the definition of the GreedyIntervalScheduler guarantees that $f_j \leq f_{i_1}$, so we also have $f_j < s_{i_\ell}$ for each $\ell = 2, 3, \ldots, k$ and the interval $j$ does not overlap with any of the intervals $i_2, \ldots, i_k$, as we wanted to show. \qed

We can now use a proof by induction to establish the correctness of the GreedyScheduler algorithm.

**Theorem 6.6.** The GreedyScheduler solves the interval scheduler problem.

**Proof.** Let $I^* = \{i_1, i_2, \ldots, i_k\}$ be a maximum set of non-overlapping intervals, and let $I^j = \{j_1, j_2, \ldots, j_m\}$ be the set of non-overlapping intervals returned by the algorithm. We again sort the indices so that $f_{j_1} < f_{j_2} < \cdots < f_{j_m}$ and $f_{j_1} < \cdots < f_{j_m}$. Clearly, $m \leq k$; we want to show that in fact $m = k$.

Let us now use a proof by induction on $c$ to show that for every $c$ in the range $1 \leq c \leq m$, the set $I^{(c)} = \{j_1, j_2, \ldots, j_c, i_{c+1}, \ldots, i_k\}$ is a maximum set of non-overlapping intervals. The base case was established in Proposition 6.5.

For the induction step with $c \geq 2$, the induction hypothesis is that $I^{(c-1)}$ is a maximum set of non-overlapping intervals; we want to show the same is true of $I^{(c)}$. Note that $I^{(c)} = (I^{(c-1)} \setminus \{i_c\}) \cup \{j_c\}$. We must have $f_{j_c} \leq f_{i_c} < f_{i_{c+1}} < \cdots < f_{i_k}$ so $j_c \notin \{j_1, j_2, i_{c+1}, \ldots, i_k\}$ and $|I^{(c)}| = |I^{(c-1)}|$. Furthermore, since $I^j$ and $I^{(c-1)}$ are sets of non-overlapping intervals and since $f_{j_c} \leq f_{i_c} < s_{i_{c+1}} < \cdots < s_{i_k}$, the set $I^{(c)}$ is a maximum set of non-overlapping intervals.

The proof by induction we just completed shows that $I^{(m)} = \{j_1, j_2, \ldots, j_m, i_{m+1}, \ldots, i_k\}$ is a maximum set of non-overlapping intervals. But the greedy algorithm returns $I^j = \{j_1, \ldots, j_m\}$ only if all other intervals with finish times greater than $j_m$ intersect with some interval already in $I^j$; this means that we must have $m = k$ and the algorithm returns a maximum set of non-overlapping intervals. \qed

3. Minimizing lateness

There are a number of other scheduling problems for which the greedy algorithm is effective. Here’s one that may be relevant with respect to juggling coursework for multiple classes at the same time:

**Definition 6.7** (Minimizing lateness problem). Given a set of $n$ tasks with processing times $p_1, \ldots, p_n$ and deadlines $d_1, \ldots, d_n$, find an ordering of the tasks that minimizes the maximum lateness of any task when they are performed one at a time in that order. Formally,
given an ordering $\pi$ of $\{1, 2, \ldots, n\}$, the lateness of the $k$th task in this order is

$$L^{(\pi)}_k := \sum_{i: \pi(i) \leq \pi(k)} p_i - d_k$$

and $\pi$ is a valid solution if $\max_{k \leq n} L^{(\pi)}_k$ is minimal among all permutations.

For example, an input to the problem may be tasks with the following processing times and deadlines:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$d_i$</td>
<td>5</td>
<td>9</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

If we process the tasks in the order above, we obtain lateness values

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_i$</td>
<td>-1</td>
<td>-3</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>13</td>
</tr>
</tbody>
</table>

so that the maximum lateness of this ordering is 14.

Let’s explore how we can solve this problem using greedy algorithms. Here the natural way to break down the problem into individual decisions is to pick which task to do first, then which one to do second, etc. There are a number of different criteria we could use to decide which task to schedule next.

**Shortest processing time first:** Sort the tasks in order of increasing processing times.

**Earliest deadline:** Sort the tasks in order of increasing deadlines.

**Smallest slack:** At each step, pick the task with the smallest “slack” value $d_j - p_j$.

Of those three options, only the earliest deadline criterion yields a greedy algorithm that solves the Minimal Lateness problem.

**Algorithm 3: GreedyLateness**

```plaintext
Algorithm 3: \text{GreedyLateness}(p_1, \ldots, p_n, d_1, \ldots, d_n)

$\pi \leftarrow$ a permutation of $\{1, \ldots, n\}$ for which $d_{\pi(1)} \leq d_{\pi(2)} \leq \cdots \leq d_{\pi(n)}$;

return $\pi$;
```

The permutation $\pi$ can be computed in time $\Theta(n \log n)$, and this is also the time complexity of the algorithm. The more challenging aspect of the analysis of this algorithm is its proof of correctness. We use an exchange argument: we will show that given any valid solution to the lateness problem, we can convert it to the output of the greedy algorithm without increasing its maximum lateness.

We also use the fact that we can sort an out-of-order sequence by swapping pairs of consecutive elements.

**Fact 1 (Bubble sort).** For $\pi$ any ordering of $1, 2, \ldots, n$, if we repeatedly

1. Find a pair of elements $i < j$ where $j$ is right before $i$ in $\pi$;
2. Update $\pi$ by swapping elements $i$ and $j$;

then after at most $\Theta(n^2)$ swaps, we end up with the sorted ordering $1 \ 2 \ 3 \ \cdots \ n$.

---

1Exercise: Prove that statement!
2In class, we first saw the proof of correctness of the greedy algorithm in the special case where $n = 2$, then described how to obtain the general result from the special case. The following proof establishes the result directly; it is a good exercise to rewrite the proof of the special case for $n = 2$. 

For example, starting with the ordering $1 5 3 4 2$, the sorting algorithm described above yields the following sequence of orderings:

\[
\begin{align*}
1 & 5 3 4 2 \\
1 & 3 5 4 2 \\
1 & 3 4 5 2 \\
1 & 3 4 2 5 \\
1 & 3 2 4 5 \\
1 & 2 3 4 5
\end{align*}
\]

We are now ready to prove the correctness of the GreedyLate+ness algorithm.

**Theorem 6.8.** The GreedyLate+ness algorithm solves the Maximal Lateness problem.

**Proof.** For simplicity, let us reorder the tasks by increasing deadline so that the greedy algorithm performs them in order (i.e., 1, then 2, then 3, etc.) We want to prove that this order is a valid solution.

Consider any other ordering $\pi$. Then in that order there must be two consecutive tasks $i, j$ with $d_i \leq d_j$ but $j$ performed right before $i$. Swap those two tasks to obtain a new ordering $\pi'$. Then every task except $i$ and $j$ have the same lateness in $\pi$ and in $\pi'$. The lateness of $i$ satisfies

\[
L_{i}^{(\pi')} \leq L_{i}^{(\pi)}
\]

since we do task $i$ earlier in $\pi'$. And the lateness of $j$ satisfies

\[
L_{j}^{(\pi')} \leq L_{i}^{(\pi)}
\]

because $d_i \leq d_j$. Therefore, the maximum lateness of $\pi'$ is at most that of $\pi$. This means that we can continue swapping in this way until we obtain the ordering $1 2 \cdots n$ of the greedy algorithm, and at every step along the way we never increase the maximum lateness so that the algorithm’s solution is valid. \qed