Administrative Information
Course Information Sheet.
Office hours: W 5-6, R 2-3 in BA3234

Background
Asymptotic notation (big-Oh, Omega, Theta, little-oh, little-omega) [Section 2.2]

Definition Notation for upper bound (Big-O): \( f(n) = O(g(n)) \) if there are a constant \( c > 0 \) and a “threshold” \( n_0 \) so that
\[
\text{for all } n > n_0 : \quad f(n) \leq cg(n)
\]

Definition Notation for upper bound (little-o): \( f(n) = o(g(n)) \) if for all \( c > 0 \), there is a “threshold” \( n_0 \) so that
\[
\text{for all } n > n_0 : \quad f(n) \leq cg(n)
\]

Definition Notation for lower bound (Big-Omega): \( f(n) = \Omega(g(n)) \) if there are a constant \( c > 0 \) and a “threshold” \( n_0 \) so that
\[
\text{for all } n > n_0 : \quad f(n) \geq cg(n)
\]

Definition Notation for lower bound (little-omega): \( f(n) = o(g(n)) \) if for all \( c > 0 \) there is a “threshold” \( n_0 \) so that
\[
\text{for all } n > n_0 : \quad f(n) \geq cg(n)
\]

Definition Notation for exact order (Theta): \( f(n) = \Theta(g(n)) \) if both
\[
f(n) = O(g(n)) \quad \text{and} \quad f(n) = \Omega(g(n))
\]
i.e., there are constants \( c_1, c_2 > 0 \) and a “threshold” \( n_0 \) so that
\[
\text{for all } n > n_0 : \quad c_1g(n) \leq f(n) \leq c_2g(n)
\]

Proof by Induction

<table>
<thead>
<tr>
<th>Prove</th>
<th>( P(n) ) for all ( n \geq 0 )</th>
<th>( P(n) ) for all ( n \geq k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base case</td>
<td>prove ( P(0) )</td>
<td>prove ( P(k) )</td>
</tr>
<tr>
<td>Induction step</td>
<td>I.H: ( P(n) ) for ( n \geq 0 )</td>
<td>I.H: ( P(n) ) for ( n \geq k )</td>
</tr>
<tr>
<td></td>
<td>Prove ( P(n+1) )</td>
<td>Prove ( P(n+1) )</td>
</tr>
</tbody>
</table>

Greedy Algorithms (Chapter 4)

“At each step, make the choice that seems best at the time; never change your mind.”

The local decision is made using some criterion. A challenge is to come up with the criterion, and prove that it works.
Interval Scheduling [Section 4.1]

**Input:** \( n \) requests, the \( i \)-th request has starting time \( s(i) \) and finishing time \( f(i) \) (\( 0 < s(i) < f(i) \)).

**Output:** A compatible subset \( S \) of \( \{1, \ldots, n\} \) of maximal cardinality. ("\( S \) is compatible" means for all \( i \neq j \in S \), the \( i \)-th and \( j \)-th requests do not overlap.)

A. **Brute force:** consider \( 2^n \) subsets of \( \{1, \ldots, n\} \).

B. **Greedy by starting time** (always take the earliest possible request):

1. sort requests so that \( s(1) \leq s(2) \leq \ldots \leq s(n) \)
2. \( S \leftarrow \emptyset \) % partial schedule
3. \( t \leftarrow 0 \) % last finish time of activities in \( S \)
4. for \( i \leftarrow 1 \ldots n \) do
5. if \( t \leq s(i) \) then do % request \( i \) is compatible with \( S \)
6. \( S \leftarrow S \cup \{i\} \)
7. \( t \leftarrow f(i) \)
8. return \( S \)

**Correctness?** Doesn’t work. Counter-example:

\[
\begin{array}{cccccccc}
|     |     |     |     |     |     |     |     | \\
|-----|-----|-----|-----|-----|-----|-----|-----| \\
|---|---|---|---|---|---|---|---| \\
|---|---|---|---|---|---|---|---| \\
\end{array}
\]

C. **Greedy by duration** (always takes the shortest possible request): similar to above except sort by nondecreasing duration, i.e.,

\[ f(1) - s(1) \leq f(2) - s(2) \leq \ldots \leq f(n) - s(n) \]

Fix the counter example in B above, but still not correct: Counter-example:

\[
\begin{array}{cccccccc}
|     |     |     |     |     |     |     |     | \\
|-----|-----|-----|-----|-----|-----|-----|-----| \\
|-----|-----|-----|-----|-----|-----|-----|-----| \\
|---|---|---|---|---|---|---|---| \\
|---|---|---|---|---|---|---|---| \\
\end{array}
\]

D. **Greedy by overlap count** (try to avoid conflicts): similar to above except sort the requests by the "number of conflicts" (the number of conflicts of a request is the number of other requests that overlap with it).

Fix the counter example in C above, but still not correct. Counter-example:

\[
\begin{array}{cccccccc}
|     |     |     |     |     |     |     |     | \\
|---|---|---|---|---|---|---|---| \\
|---|---|---|---|---|---|---|---| \\
|---|---|---|---|---|---|---|---| \\
|---|---|---|---|---|---|---|---| \\
\end{array}
\]

E. **Greedy by finishing time** (try to make the resource free as soon as possible): similar to above except sort by nondecreasing finish time, i.e.,

\[ f(1) \leq f(2) \leq \ldots \leq f(n) \]

**Correctness?**

Let \( S_0, S_1, \ldots, S_n \) be the partial solutions constructed by algo. at the end of each iteration.
**Definition:** $S_i$ is called “promissing” if there is an optimal solution which extends using the requests from \( \{i + 1, \ldots, n\} \). i.e., there is an optimal solution \( OPT \) so that

\[
S_i \subseteq OPT \subseteq S_i \cup \{i + 1, \ldots, n\}
\]

**Note:** \( OPT \) may not be unique (there may be more than one way to achieve optimal).

**Prove by induction** on \( i \) (\# iterations) that \( S_i \) is “promissing”.

**Base case:** \( S_0 = \emptyset \) is promising because any optimal solution extends \( S_0 \) using only requests from \( \{1, \ldots, n\} \).

**Ind. Hyp.:** For some \( i \geq 0 \), assume that \( S_i \) is promising, i.e., there is an optimal \( OPT_i \) that extends \( S_i \) using only requests from \( \{i + 1, \ldots, n\} \).

**Ind. Step:** Prove that \( S_{i+1} \) is promising by showing that there exists an optimal solution \( OPT_{i+1} \) so that

\[
S_{i+1} \subseteq OPT_{i+1} \subseteq S_{i+1} \cup \{i + 2, \ldots, n\}
\]

Consider the following cases:

**Case 1:** \( S_{i+1} = S_i \) This means the request \( i + 1 \) is not compatible with \( S_i \). Take \( OPT_{i+1} = OPT_i \), then the first \( \subseteq \) in (1) holds by the assumption, and the second \( \subseteq \) in (1) holds because \( OPT_i \) does not contain \( i + 1 \) (since \( i + 1 \) is not compatible with \( S \)).

**Case 2:** \( S_{i+1} = S_i \cup \{i+1\} \) Here \( OPT_i \) may or may not include \( i + 1 \). Consider both possibilities.

**Subcase 2a:** \( i + 1 \in OPT_i \) Take \( OPT_{i+1} = OPT_i \), then the first \( \subseteq \) in (1) holds by the I. H., and the second \( \subseteq \) in (1) holds because both \( S_{i+1} \) and \( OPT_{i+1} \) contains \( i + 1 \).

**Subcase 2b:** \( i + 1 \notin OPT_i \) Since \( OPT_i \) is optimal, it there must be some request \( j \) in \( OPT_i \) that overlaps with \( i + 1 \). Let

\[
OPT_{i+1} = OPT_i \setminus \{j\} \cup \{i + 1\}
\]

then (1) holds.

We have to argue that \( OPT_{i+1} \) is an optimal solution. First, \( OPT_{i+1} \) has the same cardinality as \( OPT_i \). So we just have to argue that (the new request) \( i + 1 \) is compatible with all other requests in \( OPT_{i+1} \). This amounts to showing that \( j \) is the only request in \( OPT_i \) that overlaps with \( i + 1 \). In fact, request \( j \) cannot be in \( S_i \) (since \( S_{i+1} = S_i \cup \{i + 1\} \) is compatible), so \( j \geq i + 2 \). If there is another request \( j' \in OPT_i \) that overlaps with \( i + 1 \), then we also have \( j' \geq i + 2 \). Since we sorted the requests in increasing order of finishing time, we have

\[
f(i + 1) \leq f(j), f(j')
\]

So \( j \) and \( j' \) overlap, contradict the fact that \( OPT_i \) is compatible. \( \square \)