Suppose you have a time period (e.g. a day) and a resource that needs to be shared during that period. It could be a room that is available for booking, or a special instrument such as MRI scanner.

Suppose there are a set of intervals \([s(i), f(i)]\) which denote the start and finish times.

Two intervals \([s(i), f(i)]\) and \([s(j), f(j)]\) are compatible if they don't overlap. A set of intervals is compatible if each pair of intervals from that set is compatible.

Exercise:
- Given a set of \(N\) intervals, how do you decide if they are compatible?

Let's look at some "greedy" algorithms for choosing a compatible set of intervals. What is a greedy algorithm?

Kleinberg and Tardos: "... builds up a solution in small steps, choosing a decision at each step myopically (short sighted) to optimize some underlying criterion."

Cormen, Leiserson, Rivest (CLR): "... makes the choice that looks best in the moment... it makes a locally optimal choice in the hope that the choice will lead to a globally optimal solution."

Levitin: "... choice must be (1) feasible i.e. satisfy the problem constraints, (2) the best local choice among all feasible choices available at that step, and (3) irrevocable."

For example, Dijkstra/Prim/Kruskal's algorithms are all greedy, and they happen to work -- they find a global optimum solution.

Ford-Fulkerson is NOT greedy, since it allows you to undo (reverse) flow to find a better solution.
Greedy approaches?

Greedy approach number 1:

- start with an empty set \( S \)
- repeat 
  - choose the **smallest interval** (smallest \( f(i) - s(i) \) that is compatible with all intervals in \( S \), and add this interval to \( S \))
- until there are no remaining intervals that are compatible with \( S \)

Example where this approach fails to find the optimum solution for Problem 1 (number of intervals) and Problem 2 (total duration):

Greedy approach 2:

- start with an empty set \( S \)
- repeat 
  - choose the interval that has the **smallest value of** \( s(i) \), and that is compatible with all intervals in \( S \), and add it to \( S \)
- until there are no remaining intervals that are compatible with \( S \)

Example where this approach fails both for problem 1 (maximize the number of intervals) and problem 2 (maximize the total duration of the intervals):

Greedy approach 3:

- start with an empty set \( S \)
- repeat 
  - find the interval with the **smallest value of** \( f(i) \) and that is compatible with all intervals in \( S \), and add it to \( S \)
- until there are no remaining intervals that are compatible with \( S \)

This works for Problem 1 (number of intervals).

Exercises: does it work for Problem 2 also?

Order the intervals by their finishing time 
\( f(1) \leq f(2) \leq f(3) \leq \ldots \leq f(N) \)
This takes \( O(N \log N) \) eg. mergesort.

```
interval index
1       f(1)
2       f(2)
3       f(3)
4
5
6       f(6)
```

Greedy approach 3 (Algorithm)

// find a maximal set of intervals \( S \)
\( i = 1 \)
\( S = \{ i \} \)
for \( j = 2 \) to \( N \) {
  if \( s(j) > s(i) \) \{  // compatible?
    add \( j \) to \( S \)
    \( i = j \)
  \}
}\n
```
interval index
1       f(1)
2       f(2)
3       f(3)
4       f(4)
5
6       f(6)
```

\( s(i+1) \)
\( s(j) \)
Claim: Greedy approach 3 (choose based on earliest finish) finds a maximum compatible solution to problem 1 (most intervals)

Proof:
Assume intervals are ordered by their finishing times: $f(1) \leq f(2) \leq f(3) \leq \ldots \leq f(n)$

Let $i_1, i_2, i_3, \ldots, i_r$ be the indices of solution found by algorithm

Let $o_1, o_2, o_3, \ldots, o_m$ be the indices of an optimal solution.

We know $r \leq m$. Show that $r = m$.

Induction step

$f(i_k) \leq f(o_k) \Rightarrow f(i_{k+1}) \leq f(o_{k+1})$

Since algorithm's choice of $i_k$ finishes no later than the optimal $O_k$, the algorithm has at least as many intervals to choose from for $i_{k+1}$. In particular, it could choose $O_{k+1}$ since $s(i_k) \leq f(o_k) < s(o_{k+1})$.

Prove then $f(i_n) \leq f(o_n)$ for all $n \leq r$.

Base case:

$f(i_1) \leq f(o_1)$ by definition of algorithm

Induction hypothesis:

$f(i_k) \leq f(o_k)$

In particular, $f(i_r) \leq f(o_r)$.

Next, how do we know $r = m$? If $r < m$ then there would be an interval $[s(o_r), f(o_r)]$ which is impossible since this interval would be chosen by algorithm too.

ASIDE: Another way to think about the compatibility of intervals:
Define a DAG where vertices are intervals and there is an edge from $u$ to $v$ if $f(u) < s(v)$.

Lecture 12

Interval scheduling
- greedy approach
- weighted intervals and dynamic programming approach
What is "dynamic programming"? Term attributed to Bellman (1950's).

CLR: (paraphrase) "Decompose a problem into subproblems. The subproblems are not independent, but rather they share sub-subproblems. The key is to solve each sub-subproblem only once and store these solutions in a table."

We now generalize the interval scheduling problem.

Let interval $i$ have a value $V_i$.

Choose a set $S$ of compatible intervals that maximizes the sum of values:

$$\text{Opt} = \sum_{i \in S} V_i$$

Claim: earlier problems were special cases.

Problem 1: (number) $V_i \equiv 1$

Problem 2: (total duration) $V_i \equiv f(i) - s(i)$

Greedy 3 won't solve this problem.

We introduce another approach, called "dynamic programming".

$$\{1\}$$
$$\{1, 2\}$$
$$\{1, 2, 3\}$$
$$\{1, 2, 3, 4\}$$
$$\vdots$$
$$\{1, 2, 3, 4, \ldots, N-1\}$$
$$\{1, 2, 3, 4, \ldots, N\}$$ \(< original \ problem$$

Consider solving a sequence of smaller versions of the problem. Again assume interval index is the order of finishing time.

Let $S(i)$ be a set of intervals in maximal solution of the problem when we can use only intervals $\{1, 2, \ldots, i\}$.

\[
\begin{array}{c|c|c|c}
 i & V_i & S(i) & \sum_{i \in S(i)} V_i \\
 \hline
 1 & 4 & 1 & 4 = 4 \\
 2 & 8 & 1, 2 & 4 + 8 = 12 \\
 3 & 2 & 1, 2 & 4 + 8 = 12 \\
 4 & 6 & 1, 2, 4 & 4 + 8 + 6 = 18 \\
 5 & 15 & 1, 2, 4 & 4 + 8 + 6 = 18 \\
\end{array}
\]

For each interval $i$, let $p[i]$ be the largest index such that $f(p[i]) < s(i)$.

Note: $p[i] < i$

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
 i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
 \hline
 p[i] & 0 & 1 & 1 & 3 & 1 & 1 & 2 & 3 & 3 & 3 \\
 f(i) & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
 f(p[i]) & 0 & 1 & 1 & 3 & 1 & 1 & 2 & 3 & 3 & 3 \\
 s(i) & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
 f(j) & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]

Claim: If $i \in S(i)$ then $S(i)$ cannot contain $j$ where $p[i] < j < i$.

Proof:

To have $p[i] < j < i$, we would need $f(p[i]) < f(j)$ and $f(j) < s(i)$.

But this would contradict the definition of $p[i]$. 
Let the maximum value using intervals \(S(1) \ldots S(j)\) be:
\[
\text{Opt}(i) = \max \{ \text{opt}(i-1), v_i + \text{Opt}(p[i]) \}
\]

\(S(i)\) does not contain \(i\).

\(S(i)\) contains \(i\).

**Claim:**

\[
\text{Opt}(i) = \max \{ \text{Opt}(i-1), v_i + \text{Opt}(p[i]) \}
\]

\(\text{Opt}(0) = 0\)

for \(i = 1 \to N\),

\[
\text{Opt}(i) = \max \{ \text{Opt}(i-1), v_i + \text{Opt}(p[i]) \}
\]

return \(\text{Opt}(N)\)

**Note:** Assuming \(p[i]\) has been pre-computed, this algorithm takes \(O(N)\).

What about a recursive algorithm?

\[
\text{ComputeOpt}(i) = \begin{cases} 
0 & \text{if } n == 0 \\
0 & \text{return } \\
\text{else} & \text{return } \max \{ \text{ComputeOpt}(i-1), v_i + \text{ComputeOpt}(p[i]) \}
\end{cases}
\]

**Analogy:** Fibonacci

\[
\begin{array}{c|cccccc}
F(0) = 0 \\
F(1) = 1 \\
F(n+1) = F(n) + F(n-1)
\end{array}
\]

<table>
<thead>
<tr>
<th>(n)</th>
<th>0</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>(F(n))</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
</tr>
</tbody>
</table>

Suppose you try to compute \(F(N)\) using recursion, where \(N\) is large.

**Call Tree for recursive Fibonacci**

Note the redundant computation!
We can use recursion for Fibonacci but we must be careful to avoid redundant computation.

Use a global array F[0,...,N]

F[0] = 0, F[1] = 1
F[i] = F[i−2] for all i = 2,...,N

Fibonacci(k) {
  if k == 0 or k == 1 return F[k]
  else {
    if F[k−1] < 0
      F[k−1] = Fibonacci(k−1)
    if F[k−2] < 0
      F[k−2] = Fibonacci(k−2)
    return F[k−1] + F[k−2]
  }
}

Memoization

Save values that have already been computed so that you don’t have to compute them again.

How do we apply this to our weighted interval scheduling problem?

\[
\text{Opt}[i] = \sum \nu_j
\]

\[
\nu_j \in S(i)
\]

\[
\text{Opt}[0] = 0
\]

for all \( i \in \{1,2,\ldots,N\} \)

Opt[i] = \(-1\) // stores maximum value

ComputeOpt(n) {
  if n == 0, return 0
  else if \( \text{Opt}[n] \geq 0 \), return \( \text{Opt}[n] \)
  else \( \text{Opt}[n] = \max \{ \text{ComputeOpt}(n−1), \nu_n + \text{ComputeOpt}(p[n]) \} \)
  return \( \text{Opt}[n] \)

What is \( O(\cdot) \) running time?

Each call to ComputeOpt() either performs a constant number of operations and then returns, or else sets one value of Opt[ ] (and performs two calls to ComputeOpt()).

Since each value of Opt[ ] is set only once, the time required is \( O(N) \), same as iterative solution.

[Don’t forget about \( O(N \log N) \) needed to sort the intervals by finishing time.]

Q: Is this a greedy algorithm?
A: No. Although Opt[i] increases as \( i \) increases, the set \( S(i) \) does not necessarily grow as \( i \) increases.

<table>
<thead>
<tr>
<th>i</th>
<th>( \nu_i )</th>
<th>( S(i) )</th>
<th>( S(C) )</th>
<th>( \nu )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>1,2</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1,2,4</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>1,2,4</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>1,5</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1,5</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

We could compute \( S(n) \) while Opt[ ] is being computed, or do it afterwards as follows.

find \( S(n) \) {
  if \( n > 0 \) {
    if \( \text{Opt}[n−1] = \text{Opt}[n] \)
      // \( S(n) \) doesn’t contain \( n \).
      return \( \text{findS}(n−1) \)
    else {
      return \( S(n) \cup \text{findS}(p[n]) \)
    }
  } else return \{ \} // the empty set
}

my apologies for the notation - curly braces are used in two different ways