Problem – Making Change

- Task – buy a cup of coffee (say it costs 63 cents).
- You are given an unlimited number of coins of all types (neglect 50 cents and 1 dollar).
- Pay exact change.
- What is the combination of coins you’d use?

1 cent  5 cents  10 cents  25 cents
Logically, we want to minimize the number of coins.

The problem is then: Count change using the fewest number of coins – we have 1, 5, 10, 25 unit coins to work with.

The "greedy" part lies in the order: We want to use as many large-value coins to minimize the total number.

When counting 63 cents, use as many 25s as fit, $63 = 2(25) + 13$, then as many 10s as fit in the remainder: $63 = 2(25) + 1(10) + 3$, no 5’s fit, so we have $63 = 2(25) + 1(10) + 3(1)$, 6 coins.
A greedy person grabs everything they can as soon as possible.

Similarly a greedy algorithm makes locally optimized decisions that appear to be the best thing to do at each step.

Example: Change-making greedy algorithm for “change” amount, given many coins of each size:

- Loop until change == 0:
  - Find largest-valued coin less than change, use it.
  - change = change - coin-value;
Greedy Algorithms Conditions

- Used to solve optimization problems.
- For greedy algorithm to be optimal, problems must exhibit optimal substructure.
- A solution contains within it the optimal solutions to subproblems – in this case, the minimum number of coins for smaller change.
- Problems also must exhibit the greedy-choice property.
- When we have a choice to make, make the one that looks best right now.
- Make a locally optimal choice in hope of getting a globally optimal solution.
The choice that seems best at the moment is the one we go with.

Prove that when there is a choice to make, one of the optimal choices is the greedy choice (see above).

Therefore, it’s always safe to make the greedy choice.

Show that all but one of the subproblems resulting from the greedy choice are empty.
The greedy method gives the optimal solution for US coinage. With different coinage, the greedy algorithm doesn’t always find the optimal solution.

Example of a coinage with an additional 21 cent piece. Then $63 = 3(21)$, but greedy says use 2 25s, 1 10, and 3 1’s, a total of 6 coins.

The coin values need to be spread out enough to make greedy work.

But even some spread-out cases don’t work. Consider having pennies, dimes and quarters, but no nickels.

Then 30 by greedy uses 1 quarter and 5 pennies, ignoring the best solution of 3 dimes.
Lemma

Any optimal solution involving US coins cannot have more than two dimes, one nickel and four cents.

Proof.

- If we had three dimes we could replace them by a quarter and a nickel, resulting in one fewer coins.
- Replace two nickles by a dime, resulting in one fewer coins.
- Replace five cents by a nickel, resulting in four fewer coins.

Corollary

The total sum of \{1, 5, 10\} coins cannot exceed 24 cents.
The above property can be shown for values of $n < 25$ (and only $\{1, 5, 10\}$ coins).

Try to do it yourselves.

In this case, the greedy choice is to select, at every step, the largest coin we can use.

In other words: The optimal solution for $n$ always contains the largest coin $c_i$ such that $c_i \leq n$.
The Greedy Choice Property

Proof.

Again, by contradiction

- Assume there is a solution $C$ for $n$ that does not contain $c_i$.
- It means that it contains only smaller coins.
- But $c_i \leq n$ and every bigger coin can be expressed as a combination of smaller coins (see above).
- So we can always substitute $c_i$ for a combination of smaller coins (that includes the next smallest), getting a better solution.
In the case of US coins – yes, but not always. Why?
But wait... Can we always Do That?

- In the case of US coins – yes, but not always. Why?
- Because while the optimal substructure always exists, the greedy choice property does not exist for all coin combinations.

In general, if we have a set of coins \( \{a_1, a_2, \ldots, a_m\} \) such that 
\( a_t < a_{t-1} \) and for each pair \( a_t, a_{t-1} \) define 
\( m_t = \lceil \frac{a_{t-1}}{a_t} \rceil \) and 
\( S_t = a_t \times m_t \), then the greedy solution is optimal only if for every \( t \in 2..m \), 
\( G(S_t) \leq m_t \) where \( G(S_t) \) is the greedy solution for \( S_t \).

- For example – if we add a 7-cent piece, then \( \lceil \frac{10}{7} \rceil = 2 \), and 
\( S_t = 7 \times 2 = 14 \), and \( G(14) = 5 > 2 \).

- Also, for the set \( \{1, 10, 25\} \) we cannot guarantee the greedy choice property for a similar reason: \( \lceil \frac{25}{10} \rceil = 3 \), 
\( S_t = 10 \times 3 = 30 \) and \( G(30) = 6 > 3 \).
Optimal Substructure, More Formally

Lemma

If $C$ is a set of coins that corresponds to optimal change making for an amount $n$, and if $C'$ is a subset of $C$ with a coin $c \in C$ taken out, then $C'$ is an optimal change making for an amount $n - c$. 

Proof.

By contradiction: Assume that $C'$ is not an optimal solution for $n - c$. In other words, there is a solution $C''$ that has fewer coins than $C'$ for $n - c$. So we could combine $C''$ with $c$ to get a better solution than $C'$, contradicting the assumption that $C'$ is optimal.
Lemma

If \( C \) is a set of coins that corresponds to optimal change making for an amount \( n \), and if \( C' \) is a subset of \( C \) with a coin \( c \in C \) taken out, then \( C' \) is an optimal change making for an amount \( n - c \).

Proof.

By contradiction:

- Assume that \( C' \) is not an optimal solution for \( n - c \).
- In other words, there is a solution \( C'' \) that has fewer coins than \( C' \) for \( n - c \).
- So we could combine \( C'' \) with \( c \) to get a better solution than \( C \), contradicting the assumption that \( C \) is optimal.
Input: Set $S$ of $n$ activities $- \{a_1, a_2, \ldots, a_n\}$.

$s_i = \text{start time of activity } i$.

$f_i = \text{finish time of activity } i$.

Output: Subset $A$ of maximum number of compatible activities.

Two activities are compatible, if their intervals do not overlap.

Example (activities in each line are compatible):

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

time
Assume activities are sorted by finishing times – $f_1 \leq f_2 \leq \cdots \leq f_n$.

Suppose an optimal solution includes activity $a_k$.

This generates two subproblems:

- Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before $a_k$ starts (compatible with $a_k$).
- Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after $a_k$ finishes.

The solutions to the two subproblems must be optimal.

Prove using the cut-and-paste approach (exchange argument).
Let $S_{ij}$ = subset of activities in S that start after $a_i$ finishes and finish before $a_j$ starts.

Subproblems: Selecting maximum number of mutually compatible activities from $S_{ij}$.

Let $c[i,j] =$ size of maximum-size subset of mutually compatible activities in $S_{ij}$.

The recursive solution is:

$$c[i, j] = \begin{cases} 
0 & \text{if } S_{ij} = \emptyset \\
\max_{i < k < j} \{c[i, k] + c[k, j] + 1\} & \text{otherwise}
\end{cases}$$
The problem also exhibits the greedy-choice property.

There is an optimal solution to the subproblem $S_{ij}$, that includes the activity with the smallest finish time in set $S_{ij}$.

It can be proved easily (how?).

Hence, there is an optimal solution to $S$ that includes $a_1$.

Therefore, make this greedy choice without solving subproblems first and evaluating them.

Solve the subproblem that ensues as a result of making this greedy choice.

Combine the greedy choice and the solution to the subproblem.
Algorithm 1 Recursive-Activity-Selector (s, f, i, j)

1: \( m \leftarrow i + 1 \)
2: \[ \textbf{while} \ m < j \text{ and } s_m < f_i \text{ \textbf{do}} \]
3: \( m \leftarrow m + 1 \)
4: \[ \textbf{end while} \]
5: \[ \textbf{if} \ m < j \text{ \textbf{then}} \]
6: \( \text{\textbf{return}} \ a_m \cup \text{Recursive} - \text{Activity} - \text{Selector}(s, f, m, j) \)
7: \[ \textbf{else} \]
8: \( \text{\textbf{return}} \ \emptyset \)
9: \[ \textbf{end if} \]

- Top level call: \( \text{Recursive} - \text{Activity} - \text{Selector}(s, f, 0, n + 1) \)
- Complexity??
- See text for iterative version
**Input:** Set S of n lectures – \( \{a_1, a_2, \ldots, a_n\} \).

\( s_i \) = start time of lecture i.

\( f_i \) = finish time of lecture i.

**Goal:** find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

**Example:** Four lecture rooms for 10 lectures.

```
4  _____________  e ___________  j ___________  
3  c _______  d _______  g _______  
2  b ___________  h ___________  
1  a _______  f _______  i _______  
```

<table>
<thead>
<tr>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Input:** Set $S$ of $n$ lectures – $\{a_1, a_2, \ldots, a_n\}$.

$s_i = $ start time of lecture $i$.

$f_i = $ finish time of lecture $i$.

**Goal:** find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

The optimal solution uses three rooms.
Algorithm 2 EARLIEST-START-TIME-FIRST \((n, s, f)\)

1: \(d \leftarrow 0\)
2: for \(j = 1 \ldots n\) do
3: \hspace{1em} if lecture \(j\) compatible with a classroom then
4: \hspace{2em} Put \(j\) in that classroom
5: \hspace{1em} else
6: \hspace{2em} Open a new classroom \(d+1\)
7: \hspace{2em} Schedule lecture \(j\) in classroom \(d+1\)
8: \hspace{1em} \(d \leftarrow d + 1\)
9: \hspace{1em} end if
10: end for
11: return schedule
Interval Partitioning Analysis

- The earliest-start-time-first algorithm can be implemented in $O(n \log n)$ time.
- Store classrooms in a priority queue (key = finish time of its last lecture).
- To determine whether lecture $j$ is compatible with some classroom, compare $s_j$ to key of min classroom $k$ in priority queue.
- To add lecture $j$ to classroom $k$, increase key of classroom $k$ to $f_j$.
- Total number of priority queue operations is $O(n)$.
- Sorting by start times takes $O(n \log n)$ time.
- **Remark:** This implementation chooses a classroom $k$ whose finish time of its last lecture is the earliest.
The **depth** of a set of open intervals is the maximum number of intervals that contain any given point.

In our case – the maximum number of classes held simultaneously at any given point.

Key observation: Number of classrooms needed $\geq$ depth.

Minimum number of classrooms needed always equal depth.

Moreover, earliest-start-time-first algorithm finds a schedule whose number of classrooms equals the depth.
**Observation:** The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

**Theorem:** Earliest-start-time-first algorithm is optimal.

**Proof:** Let \( d \) = number of classrooms that the algorithm allocates.

Classroom \( d \) is opened because we needed to schedule a lecture, say \( j \), that is incompatible with a lecture in each of \( d-1 \) other classrooms.

Thus, these \( d \) lectures each end after \( s_j \).

Since we sorted by start time, each of these incompatible lectures start no later than \( s_j \).

Thus, we have \( d \) lectures overlapping at time \( s_j + \epsilon \).

**Key observation:** all schedules use \( \geq d \) classrooms.
Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there is always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and solve top-down.
- May have to preprocess input to put it into greedy order.
- Example: Sorting activities by finish time.