Greedy Algorithms
Greedy: Example: Coin Change Problem

- **Problem:** Given currency denominations: 1, 5, 10, 25, 100, design an algorithm to pay customers using **fewest** number of coins
  - Example: 34 cents = 25 + 5 + 1 + 1 + 1 + 1

- **Cashier’s Algorithm:**
  - At each iteration, add coin of the **largest** value that does not take us past the amount to be paid
Cashier’s algorithms

- Is cashier’s algorithm optimal for any set of denominations?
- No. Consider U.S. postage: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.
  - Cashier’s algorithm: \(140\)¢ = 100 + 34 + 1 + 1 + 1 + 1 + 1.
  - Optimal: \(140\)¢ = 70 + 70.

Conclusion: Greedy algorithms not always optimal
Greedy Algorithms

- Greedy algorithms are often used for solving **optimization problems**: problems with many possible solutions. Each solution has a value and we wish to find an optimal solution: a solution with the optimal (maximum or minimum) value subject to some constraints:
  - Minimize the number of coins given the set of coins
  - Maximize the number of activities you can go, given that you cannot be at more than one event at a time

- Greedy Algorithms build solutions by choosing locally best optimal option at each step:
  - If the globally best solution is obtained by repeatedly choosing the locally best option the problem can be solved using greedy technique
Greedy Algorithms

● Advantages
  ○ Simple (to design and implement)
  ○ Efficient

● Disadvantages
  ○ It is easy to find a feasible solution, but it is often difficult to find the optimal greedy algorithm
  ○ For each feasible solution, do one of the following:
    ■ use counterexamples to show the algorithm is not optimal
    ■ prove the solution is optimal. Proof techniques:
      ● Greedy Stays ahead: We show that at each step the greedy algorithm makes a better choice compared to the optimal algorithm.
        ○ Greedy never falls behind: in each step, the greedy algorithm does at least as well as any other optimal algorithm
      ● Exchange arguments: An optimal solution can be transformed into a greedy solution without losing its optimality
      ● Structural: The greedy algorithm is optimal based on the output rather than how it operates
Activity Selection/Interval Scheduling
Activity Selection/Interval scheduling

- **Input:** a list of activities \((s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)\)
  - \(s_i\): start time of activity \(i\)
  - \(f_i\): finish time of activity \(i\)

- **Output:** maximize the **number** activities, such that there are no overlapping activities.
Activity Selection: Brute Force

Brute-force technique:

- Generate all subsets of activities
- for each subset of activities
  - Check if the activities are compatible (have overlap or not)
- Pick the subset with maximum number of elements and no overlap

Runtime: $O(2^n)$
Activity Selection: Greedy Alg’m 1

Select the activity that start earliest

- **Sort the activities in order of start time**
- **Repeat**
  - Select the activity with the earliest start time
  - Remove all the activities that overlap with it
- **Counter-example:**
  - Approach 1 does not work
Activity Selection: Greedy Alg’m 2

Select the activity that has the shortest duration

- Sort the activities in order of $f_i - s_i$
- Repeat
  - Select the next activity
  - Remove all the activities that overlap with it

- Counter-example:
  - Approach 2 does not work
Activity Selection: Greedy Alg’m 3

Select the activity with the fewest number of overlaps

- Sort the activities in order of their number of overlaps
- Repeat
  - Select the activity that has the fewest number of overlaps
  - Remove all the activities that overlap with it

- Counter-example:
  - Approach 3 does not work
Activity Selection: Greedy Alg’m 4: Optimal Solution

Select the activity with the earliest finish time

- Sort the activities in order of their finish time
- Repeat
  - Select the activity that has the earliest finish time
  - Remove all the activities that overlap with it

Local decision: Maximize the time left to satisfy other requests
Activity Selection: Greedy Alg’m 4: Runtime Analysis

A = List of all activities

Sort A into ascending order by finishing time $\rightarrow$ O(n log n)

S = []

This loop will iterate through all n intervals, visiting each exactly once for O(n) total iterations.

for j=1 to n

If A[j] does not overlap S:

S = S.append( A[j])

Is done by comparing the start time of activity j and finish time of the last activity added to S. The comparison takes O(1) time.

return S
Activity Selection: Alg’m 4: Proof of optimality

- To prove optimality of algorithm, we use “greedy stays ahead” argument
  - In each step greedy algorithm does better than an optimal solution
    - In each step greedy algorithm chooses an activity that finishes before an activity chosen by an optimal solution
      - Why this is better? Since it means our algorithm cannot choose fewer activities compared to an optimal solution
    - Then use that to prove the algorithm produces an optimal solution
Proof of optimality: Notations

\( \mathcal{G} = \) Greedy Solution (earliest finish time)

\( \mathcal{O} = \) Optimal solution

\( f(\mathcal{G}, i) \) : the finish time of activity \( i \) in schedule \( \mathcal{G} \)

\( f(\mathcal{O}, i) \) : the finish time of activity \( i \) in schedule \( \mathcal{O} \)
Proof of optimality: Greedy stays ahead

- **Theorem (k):** In step \( k \), the greedy algorithm chooses an activity that finishes no later than the activity chosen in step \( K \) of any optimal solution.

- **Proof by induction**
  - **Base case:** \( f(G, 1) \leq f(O, 1) \) : The greedy algorithm selects an activity with minimum finish time
  - **Induction hypothesis:** \( T(i) \) is True. \( f(G, i) \leq f(O, i) \)
  - **Induction step:** Showing \( T(i+1) \) is true: \( f(G, i+1) \leq f(O, i+1) \)
    - \( f(G, i) \leq f(O, i) \rightarrow \) i-th activity in \( G \) finishes before i-th activity in \( O \).
    - \( f(O, i) \leq S(O, i+1) \): The (i+1)-st activity in \( O \) must start after the i-th activity in \( O \) ends
    - \( \rightarrow f(G, i) \leq S(O, i+1) \): (i + 1)st activity in \( O \) must start after the ith activity in \( S \) finishes
    - \( \rightarrow \) Therefore, when the greedy algorithm selects its (i+1)-st activity, the (i+1)-st activity in \( O \) must be there. The greedy algorithm selects the activity with lowest end time, therefore
    - \( f(G, i+1) \leq f(O, i+1) \)
Proof of optimality

Proof by induction

\( O \)

\( g \)
Proof of optimality

- Theorem: The greedy algorithm for activity selection produces an optimal schedule.
  - $|G| \leq |O|$ since $O$ is optimal. To prove greedy algorithm is optimal we must show that $|G| \geq |O|$

- **Proof by contradiction:** Suppose $|G| < |O|$
  - Let $k = |G|$.
  - $|G| < |O| \rightarrow f(G, k) \leq f(O, k) < S(O, k+1)$: there is a $(k + 1)$st activity in $O$ and its start time must be after $f(O, k)$ and therefore after $f(G, k)$.
  - Thus after the greedy algorithm added its $k$-th activity to $G$, the $(k + 1)$st activity from $O$ would still be there. But the greedy algorithm ended after $k$ activities $\rightarrow$ Contradiction

- Greedy keeps selecting jobs until no more compatible jobs left.

- $\rightarrow$ The greedy algorithm must be optimal: $|G| = |O|$
Scheduling to minimize lateness
Scheduling to minimize lateness

- **Input:**
  - n jobs each requiring processing time $t_i$ and has a deadline $d_i$

- **Goal:** Schedule all jobs to minimize maximum lateness: $\min \{ \max_i L_i \}$
  - $s_i$: Start time, $f_i$: finish time
  - Schedule task: assigns to each job start time $s_i$, such that jobs do not overlap: $f_i \rightarrow f_i = s_i + t_i$
  - Lateness for job i: $L_i = f_i - d_i$

<table>
<thead>
<tr>
<th></th>
<th>$t_i$</th>
<th>$d_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>15</td>
</tr>
</tbody>
</table>
Scheduling to minimize lateness

- **Brute-force technique**
  - Compute the maximum lateness for all configurations
  - Pick the one that has the minimum lateness
- **Runtime:** Number of configuration: $n!$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$t_i$</th>
<th>$d_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
Scheduling to minimize lateness

- **Approach 1—Shortest jobs first**
- Sort jobs in order of increasing processing time $t_i$
- Counterexample:

<table>
<thead>
<tr>
<th>i</th>
<th>$t_i$</th>
<th>$d_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Approach 1

- max lateness: 1

Optimal:
- max lateness: 0
Scheduling to minimize lateness

- Approach 2 - shortest slack time first
- Sort jobs in order of increasing slack: $d_i - t_i$
- Counterexample:

<table>
<thead>
<tr>
<th></th>
<th>$t_i$</th>
<th>$d_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Approach 2
max lateness: 9

Optimal
max lateness: 1
Scheduling to minimize lateness: Runtime Analysis

- Approach 3 – Earliest deadline first
- Sort jobs in increasing order of their deadlines \(d_i\) and schedule them in this order

```python
def EARLIEST-DEADLINE-FIRST (n, T, D):
    T: a list containing times (t1, t2, ..., tn)
    D: a list containing deadlines (d1, d2, ..., dn)
    Sort jobs by deadlines
    t = 0
    for i = 1 to n:
        Assign job i to interval \([t, t+t_i]\)
        \(s_i = t\)
        \(f_i \leftarrow t + t_i\)
        \(t \leftarrow t + t_i\)
    return intervals \([s_1, f_1], [s_2, f_2], ..., [s_n, f_n]\).```
Scheduling to minimize lateness

- **Observation 1**: There exists an optimal schedule with no idle time.
- **Observation 2**: The earliest-deadline-first schedule has no idle time.
- **Observation 3**: All schedules with no inversions and no idle time have the same maximum lateness.
Observation 3

All schedules with no inversions and no idle time have the same maximum lateness

- **Distinct deadlines, unique schedule**
- **Non-distinct deadlines:** Consider two jobs with deadline $d$
  - the maximum lateness does not depend on the order in which they are scheduled
  - Suppose two jobs have duration $t_i$ and $t_j$ and the same deadline $d$
    - The second job has the maximum lateness
    - If $i$ is scheduled first at time $s$, the max lateness for the second job is:
      - $\text{Max}\{0, (s + t_i + t_j) - d\}$
    - If $j$ is scheduled first at time $s$, the max lateness for the second job is the same:
      - $\text{Max}\{0, (s + t_i + t_j) - d\}$
Scheduling to minimize lateness: Inversion

**Inversion**: Given a schedule S, an inversion is a pair of jobs i and j such that $d_i < d_j$ but j is scheduled before i.

- **Claim**: Exchanging two adjacent, inverted jobs i and j reduces the number of inversions by 1 and does not increase the maximum lateness.
- **Proof**:
  - Let L be the lateness before the swap and L’ be the lateness after the swap.

  \[
  L'_k = L_k \text{ for all } k \neq i, j
  \]

  \[
  L'_i \leq L_i
  \]

  \[
  L'_j = f'_j - d_j \quad \text{definition}
  \]

  \[
  L'_j = f_i - d_j \quad j \text{ now finishes at time } f_i
  \]

  \[
  L'_j = f_i - d_i \quad i < j \rightarrow d_i \leq d_j
  \]

  \[
  L'_j \leq L_i
  \]
Observation 4: If an idle-free schedule has an inversion, then it has an adjacent inversion

- **Recall**: $i, j$ is an inversion if $j$ is scheduled before $i$ but $d_i < d_j$

**Proof**: Let $i, j$ be any two closest non-adjacent inversion (jobs $j+1$ to $i-1$ are in ascending order of deadline)

- Let $k$ be a job immediately to the right of $j$.
- **Case 1.** $d_j > d_k$: Then $j, k$ is an adjacent and closer inversion → contradiction
- **Case 2.** $d_j < d_k$: Then $i, k$ is a closer inversion since $d_i < d_j$ and $d_j < d_k$ → $d_i < d_k$ → contradiction: $i, j$ assumed to be a closest inversion
The earliest-deadline-first schedule is optimal (has minimum max lateness)

Proof:

- All schedule with no inversions and no idle time have the same maximum lateness: observation3
- Greedy schedule has no inversions and no idle time: observation2
- Consider an optimal schedule $\mathcal{O}$, without loss of generality, we can assume that
  - $\mathcal{O}$ has no idle time
  - $\mathcal{O}$ has no inversions, why?
  - We convert the optimal solution to the greedy solution without losing its optimality
    - [iterate and exchange] If there is an inversion, there must be an adjacent inversion: observation4
    - Exchanging the adjacent inversions decreases number of inversions by 1 without increasing max lateness: claim

$\mathcal{G}$ and $\mathcal{O}$ have same max lateness
Scheduling to minimize lateness: Proof of optimality

- **Proof technique:**
  - **exchange argument:** Transform any optimal solution to the solution found by the greedy algorithm without hurting its quality
  - This means the greedy algorithm must have found a solution that is at least as good as any other solution