Greedy
CSCI 532
Activity Selection

Input:
• \( S = \{a_1, a_2, \ldots, a_n\} \) – set of courses that need rooms.
• \( a_i = (s_i, f_i) \) – start and finish times for each course.

Rules:
• \( a_i \) and \( a_j \) are compatible if \([s_i, f_i]\) and \([s_j, f_j]\) do not overlap.

Goal: Select a maximum sized subset of mutually compatible courses.

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_i )</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>( f_i )</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

Greedy decision?
Earliest compatible finish time first.
Activity Selection

Input:
• \( S = \{a_1, a_2, \ldots, a_n\} \) – set of courses that need rooms.
• \( a_i = (s_i, f_i) \) – start and finish times for each course.

Rules:
• \( a_i \) and \( a_j \) are compatible if \([s_i, f_i]\) and \([s_j, f_j]\) do not overlap.

Goal: Select a maximum sized subset of mutually compatible courses.

Greedy decision?
Smallest conflict first.
Activity Selection

Input:
• \( S = \{a_1, a_2, ..., a_n\} \) – set of courses that need rooms.
• \( a_i = (s_i, f_i) \) – start and finish times for each course.

Rules:
• \( a_i \) and \( a_j \) are compatible if \([s_i, f_i)\) and \([s_j, f_j)\) do not overlap.

Goal: Select a maximum sized subset of mutually compatible courses.

Greedy decision?
Smallest duration first.
Activity Selection

activity_selection(activities A)
    sort_by_finish(A)
    selected = {A[1]}
    last_added = 1
    for i = 2 to A.length
        if A[i].start ≥ A[last_added].finish
            selected = selected ∪ {A[i]}
            last_added = i
    return selected

+---+---+---+---+---+---+
<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>$f_i$</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>
Activity Selection

\[
\text{activity\_selection(activities } A) \\
\quad \text{sort\_by\_finish}(A) \\
\quad \text{selected} = \{A[1]\} \\
\quad \text{last\_added} = 1 \\
\text{for } i = 2 \text{ to } A.\text{length} \\
\quad \text{if } A[i].\text{start} \geq A[\text{last\_added}].\text{finish} \\
\quad \quad \text{selected} = \text{selected} \cup \{A[i]\} \\
\quad \quad \text{last\_added} = i \\
\text{return } \text{selected}
\]

\[
\begin{array}{c|ccccc}
   & 1 & 2 & 3 & 4 & 5 \\
\hline
   i & & & & & \\
   s_i & 1 & 3 & 4 & 5 & 7 \\
   f_i & 3 & 5 & 6 & 7 & 9
\end{array}
\]
Activity Selection

activity_selection(activities A)
    sort_by_finish(A)
    selected = {A[1]}
    last_added = 1
    for i = 2 to A.length
        if A[i].start ≥ A[last_added].finish
            selected = selected ∪ {A[i]}
            last_added = i
    return selected

Running Time? $O(n \log n)$

Validity?

Performance?

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>$f_i$</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>
Activity Selection

activity_selection(activities A)
    sort_by_finish(A)
    selected = {A[1]}
    last_added = 1
    for i = 2 to A.length
        if A[i].start ≥ A[last_added].finish
            selected = selected ∪ {A[i]}
            last_added = i
    return selected

Running Time? $O(n \log n)$

Validity? selected consists of compatible courses.

Performance?

|   |   |   |   |   |   |
|---|---|---|---|---|
| $i$ | 1 | 2 | 3 | 4 | 5 |
| $s_i$ | 1 | 3 | 4 | 5 | 7 |
| $f_i$ | 3 | 5 | 6 | 7 | 9 |
Activity Selection

activity_selection(activities A)
    sort_by_finish(A)
    selected = {A[1]}
    last_added = 1
    for i = 2 to A.length
        if A[i].start ≥ A[last_added].finish
            selected = selected ∪ {A[i]}
            last_added = i
    return selected

Running Time? $O(n \log n)$

Validity? selected consists of compatible courses.

Performance? Is selected the largest possible subset?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>$f_i$</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>
Activity Selection

Greedy decision: Select the next course with the earliest compatible finish time.

Proof of optimality:
Activity Selection

**Greedy decision:** Select the next course with the earliest compatible finish time.

**Proof of optimality:** Let $A$ be the set of courses, $S \subseteq A$ be the greedy algorithm selection, and $OPT \subseteq A$ be an optimal selection, all sorted by increasing finish time.
Activity Selection

Greedy decision: Select the next course with the earliest compatible finish time.

Proof of optimality: Let $A$ be the set of courses, $S \subseteq A$ be the greedy algorithm selection, and $OPT \subseteq A$ be an optimal selection, all sorted by increasing finish time.

Let $A[k], k \neq 1$ be the first course in $OPT$ (i.e. the optimal solution does not start with the greedy choice, $A[1]$).
Activity Selection

Greedy decision: Select the next course with the earliest compatible finish time.

Proof of optimality: Let $A$ be the set of courses, $S \subseteq A$ be the greedy algorithm selection, and $OPT \subseteq A$ be an optimal selection, all sorted by increasing finish time.

Let $A[k], k \neq 1$ be the first course in $OPT$ (i.e. the optimal solution does not start with the greedy choice, $A[1]$).

Activity Selection

Greedy decision: Select the next course with the earliest compatible finish time.

Proof of optimality: Let $A$ be the set of courses, $S \subseteq A$ be the greedy algorithm selection, and $OPT \subseteq A$ be an optimal selection, all sorted by increasing finish time.

Let $A[k], k \neq 1$ be the first course in $OPT$ (i.e. the optimal solution does not start with the greedy choice, $A[1]$).

Swapping $A[k]$ with $A[1]$ in $OPT$ must also be an optimal solution:
Activity Selection

**Greedy decision:** Select the next course with the earliest compatible finish time.

**Proof of optimality:** Let $A$ be the set of courses, $S \subseteq A$ be the greedy algorithm selection, and $OPT \subseteq A$ be an optimal selection, all sorted by increasing finish time.

Let $A[k], k \neq 1$ be the first course in $OPT$ (i.e. the optimal solution does not start with the greedy choice, $A[1]$).

Swapping $A[k]$ with $A[1]$ in $OPT$ must also be an optimal solution:

$$OPT:$$

![Diagram showing the selection process](image-url)
Activity Selection

**Greedy decision:** Select the next course with the earliest compatible finish time.

**Proof of optimality:** Let $A$ be the set of courses, $S \subseteq A$ be the greedy algorithm selection, and $OPT \subseteq A$ be an optimal selection, all sorted by increasing finish time.

Let $A[k], k \neq 1$ be the first course in $OPT$ (i.e. the optimal solution does not start with the greedy choice, $A[1]$).

Swapping $A[k]$ with $A[1]$ in $OPT$ must also be an optimal solution:

$$OPT: \quad A[k]$$
Activity Selection

**Greedy decision:** Select the next course with the earliest compatible finish time.

**Proof of optimality:** Let $A$ be the set of courses, $S \subseteq A$ be the greedy algorithm selection, and $OPT \subseteq A$ be an optimal selection, all sorted by increasing finish time.

Let $A[k], k \neq 1$ be the first course in $OPT$ (i.e. the optimal solution does not start with the greedy choice, $A[1]$).

Swapping $A[k]$ with $A[1]$ in $OPT$ must also be an optimal solution: If every course in $OPT$ is compatible with $A[k]$ (i.e. they all start after $f_k$), they must be also be compatible with $A[1]$ since $f_1 \leq f_k$.

![Diagram of Activity Selection](attachment:image.png)
Activity Selection

**Greedy decision:** Select the next course with the earliest compatible finish time.

**Proof of optimality:** Let \( A \) be the set of courses, \( S \subseteq A \) be the greedy algorithm selection, and \( OPT \subseteq A \) be an optimal selection, all sorted by increasing finish time.

Let \( A[k], k \neq 1 \) be the first course in \( OPT \) (i.e. the optimal solution does not start with the greedy choice, \( A[1] \)).

Swapping \( A[k] \) with \( A[1] \) in \( OPT \) must also be an optimal solution: If every course in \( OPT \) is compatible with \( A[k] \) (i.e. they all start after \( f_k \)), they must be also be compatible with \( A[1] \) since \( f_1 \leq f_k \).
Activity Selection

Greedy decision: Select the next course with the earliest compatible finish time.

Proof of optimality: Let $A$ be the set of courses, $S \subseteq A$ be the greedy algorithm selection, and $OPT \subseteq A$ be an optimal selection, all sorted by increasing finish time.

Let $A[k], k \neq 1$ be the first course in $OPT$ (i.e. the optimal solution does not start with the greedy choice, $A[1]$).

Swapping $A[k]$ with $A[1]$ in $OPT$ must also be an optimal solution: If every course in $OPT$ is compatible with $A[k]$ (i.e. they all start after $f_k$), they must be also be compatible with $A[1]$ since $f_1 \leq f_k$. Thus, $OPT_1 = OPT \setminus A[k] \cup A[1]$ is also optimal.

$OPT$:  

$OPT_1$:  

$A[k]$  

$A[1]$
Activity Selection

Greedy decision: Select the next course with the earliest compatible finish time.

Proof of optimality: Let $A$ be the set of courses, $S \subseteq A$ be the greedy algorithm selection, and $OPT \subseteq A$ be an optimal selection, all sorted by increasing finish time.

Let $A[k], k \neq 1$ be the first course in $OPT$ (i.e. the optimal solution does not start with the greedy choice, $A[1]$).

Swapping $A[k]$ with $A[1]$ in $OPT$ must also be an optimal solution: If every course in $OPT$ is compatible with $A[k]$ (i.e. they all start after $f_k$), they must be also be compatible with $A[1]$ since $f_1 \leq f_k$. Thus, $OPT_1 = OPT \setminus A[k] \cup A[1]$ is also optimal.

What does the solution $OPT_1' = OPT_1 \setminus A[1]$ for the instance $A' = \{A[i] \in A: s_i \geq f_1\}$ look like?

$OPT_1'$: 

\[
\begin{array}{cccccc}
\text{A[1]} & \text{Blue} & \text{Green} & \text{Purple} & \text{Red} & \text{Yellow}
\end{array}
\]
Activity Selection

Greedy decision: Select the next course with the earliest compatible finish time.

Proof of optimality: Let $A$ be the set of courses, $S \subseteq A$ be the greedy algorithm selection, and $OPT \subseteq A$ be an optimal selection, all sorted by increasing finish time.

Let $A[k], k \neq 1$ be the first course in $OPT$ (i.e. the optimal solution does not start with the greedy choice, $A[1]$).

Swapping $A[k]$ with $A[1]$ in $OPT$ must also be an optimal solution: If every course in $OPT$ is compatible with $A[k]$ (i.e. they all start after $f_k$), they must be also be compatible with $A[1]$ since $f_1 \leq f_k$. Thus, $OPT_1 = OPT \setminus A[k] \cup A[1]$ is also optimal.

$OPT_1' = OPT_1 \setminus A[1]$ must be an optimal solution to the instance $A' = \{ A[i] \in A: s_i \geq f_1 \}$.  

$OPT_1'$: 

\[
\begin{array}{c}
A[1] \\
\end{array}
\]
Activity Selection

**Greedy decision:** Select the next course with the earliest compatible finish time.

**Proof of optimality:** Let $A$ be the set of courses, $S \subseteq A$ be the greedy algorithm selection, and $OPT \subseteq A$ be an optimal selection, all sorted by increasing finish time.

Let $A[k], k \neq 1$ be the first course in $OPT$ (i.e. the optimal solution does not start with the greedy choice, $A[1]$).

Swapping $A[k]$ with $A[1]$ in $OPT$ must also be an optimal solution: If every course in $OPT$ is compatible with $A[k]$ (i.e. they all start after $f_k$), they must be also be compatible with $A[1]$ since $f_1 \leq f_k$. Thus, $OPT_1 = OPT \setminus A[k] \cup A[1]$ is also optimal.

$OPT_1' = OPT_1 \setminus A[1]$ must be an optimal solution to the instance $A' = \{A[i] \in A: s_i \geq f_1\}$. If not, then a better solution to $A'$ would lead to a better solution than the optimal $OPT_1$. 

$OPT_1'$: 

```
A[1]  
```

$f_1$
Activity Selection

**Greedy decision:** Select the next course with the earliest compatible finish time.

**Proof of optimality:** Let \( A \) be the set of courses, \( S \subseteq A \) be the greedy algorithm selection, and \( OPT \subseteq A \) be an optimal selection, all sorted by increasing finish time.

Let \( A[k], k \neq 1 \) be the first course in \( OPT \) (i.e. the optimal solution does not start with the greedy choice, \( A[1] \)).

Swapping \( A[k] \) with \( A[1] \) in \( OPT \) must also be an optimal solution: If every course in \( OPT \) is compatible with \( A[k] \) (i.e. they all start after \( f_k \)), they must be also be compatible with \( A[1] \) since \( f_1 \leq f_k \). Thus, \( OPT_1 = OPT \setminus A[k] \cup A[1] \) is also optimal.

\( OPT_1' = OPT_1 \setminus A[1] \) must be an optimal solution to the instance \( A' = \{ A[i] \in A : s_i \geq f_1 \} \). If not, then a better solution to \( A' \) would lead to a better solution than the optimal \( OPT_1 \).

We can then proceed inductively and show that course in OPT can be replaced by the corresponding course in \( S \) without violating compatibility restrictions.
Activity Selection

**Greedy decision:** Select the next course with the earliest compatible finish time.

**Proof of optimality:** Let $A$ be the set of courses, $S \subseteq A$ be the greedy algorithm selection, and $OPT \subseteq A$ be an optimal selection, all sorted by increasing finish time.

Let $A[k], k \neq 1$ be the first course in $OPT$ (i.e. the optimal solution does not start with the greedy choice, $A[1]$).

Swapping $A[k]$ with $A[1]$ in $OPT$ must also be an optimal solution: If every course in $OPT$ is compatible with $A[k]$ (i.e. they all start after $f_k$), they must be also be compatible with $A[1]$ since $f_1 \leq f_k$. Thus, $OPT_1 = OPT \setminus A[k] \cup A[1]$ is also optimal.

$OPT_1' = OPT_1 \setminus A[1]$ must be an optimal solution to the instance $A' = \{ A[i] \in A : s_i \geq f_1 \}$. If not, then a better solution to $A'$ would lead to a better solution than the optimal $OPT_1$.

We can then proceed inductively and show that course in $OPT$ can be replaced by the corresponding course in $S$ without violating compatibility restrictions. Since replacing every course in $OPT$ with the courses in $S$ keeps the solution optimal, $S$ must be optimal. (i.e., we translated $OPT$ into $S$ at no extra cost).
Room Minimization

Input:
• $S = \{a_1, a_2, ..., a_n\}$ – set of courses that need rooms.
• $a_i = (s_i, f_i)$ – start and finish times for each course.

Rules:
• $a_i$ and $a_j$ are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

Goal: Compatibly schedule all courses in the min number of rooms.
Room Minimization

room_minimization1(activities A)
    num_rooms = 0
    while A is not empty
        num_rooms += 1
        A = A \ activity_selection(A)
    return num_rooms
Room Minimization

room_minimization1(activities A)
    num_rooms = 0
    while A is not empty
        num_rooms += 1
        A = A \ activity_selection(A)
    return num_rooms

Assign as many jobs to a single room as possible.
Room Minimization

\[
\text{room\_minimization1(activities } A) \\
\text{ num\_rooms } = 0 \\
\text{ while } A \text{ is not empty } \\
\text{ num\_rooms } += 1 \\
A = A \setminus \text{activity\_selection}(A) \\
\text{return } \text{num\_rooms}
\]

Assign as many jobs to a single room as possible. Is this optimal?
Room Minimization

```
room_minimization1(activities A)
    num_rooms = 0
    while A is not empty
        num_rooms += 1
        A = A \ activity_selection(A)
    return num_rooms
```

Assign as many jobs to a single room as possible.

Optimal? No.

Counterexample:
Room Minimization

```python
room_minimization1(activities A)
    num_rooms = 0
    while A is not empty
        num_rooms += 1
        A = A \ activity_selection(A)
    return num_rooms
```

Assign as many jobs to a single room as possible.

Optimal?
No.

Counterexample: `room_minimization1` will output 3 rooms, but the optimal is 2.
Room Minimization

Input:
• $S = \{a_1, a_2, ..., a_n\}$ – set of courses that need rooms.
• $a_i = (s_i, f_i)$ – start and finish times for each course.

Rules:
• $a_i$ and $a_j$ are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

Goal: Compatibly schedule all courses in the min number of rooms.

Algorithm Outline?
Room Minimization

Input:
- \( S = \{a_1, a_2, ..., a_n\} \) – set of courses that need rooms.
- \( a_i = (s_i, f_i) \) – start and finish times for each course.

Rules:
- \( a_i \) and \( a_j \) are compatible if \([s_i, f_i]\) and \([s_j, f_j]\) do not overlap.

Goal: Compatibly schedule all courses in the min number of rooms.

Algorithm Outline?
Room Minimization

Input:
• $S = \{a_1, a_2, ..., a_n\}$ – set of courses that need rooms.
• $a_i = (s_i, f_i)$ – start and finish times for each course.

Rules:
• $a_i$ and $a_j$ are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

Goal: Compatibly schedule all courses in the min number of rooms.

Algorithm Outline?
Room Minimization

Input:
• $S = \{a_1, a_2, ..., a_n\}$ – set of courses that need rooms.
• $a_i = (s_i, f_i)$ – start and finish times for each course.

Rules:
• $a_i$ and $a_j$ are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

Goal: Compatibly schedule all courses in the min number of rooms.

Algorithm Outline?

At each time:
1. When course ends, move its occupied room to unoccupied rooms.

<table>
<thead>
<tr>
<th>Occupied Rooms</th>
<th>But previously occupied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoccupied Rooms</td>
<td></td>
</tr>
<tr>
<td>Never Used Rooms</td>
<td></td>
</tr>
</tbody>
</table>
Room Minimization

Input:
• \( S = \{a_1, a_2, ..., a_n\} \) – set of courses that need rooms.
• \( a_i = (s_i, f_i) \) – start and finish times for each course.

Rules:
• \( a_i \) and \( a_j \) are compatible if \([s_i, f_i]\) and \([s_j, f_j]\) do not overlap.

Goal: Compatibly schedule all courses in the min number of rooms.

Algorithm Outline?

At each time:
1. When course ends, move its occupied room to unoccupied rooms.
2. When course starts, select unoccupied room. If none exists, selected never used room.
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
  foreach a in A
    if a.finish == t
      F.add(B.getBookedRoom())
  foreach a in A
    if a.start == t
      if F.isEmpty()
        room_num += 1
        F.add(room_num)
      B.add(F.getFreeRoom())
return F.size()
Room Minimization

room_minimization2(activities A)

\[
B = F = \text{empty} \\
\text{room\_num} = 0 \\
\text{foreach} \ \text{timeslot} \ t \\
\quad \text{foreach} \ \ a \ \text{in} \ A \\
\quad \quad \text{if} \ a.\text{finish} == t \\
\quad \quad \quad F.\text{add}(B.\text{getBookedRoom}()) \\
\quad \text{foreach} \ \ a \ \text{in} \ A \\
\quad \quad \text{if} \ a.\text{start} == t \\
\quad \quad \quad \text{if} \ F.\text{isEmpty}() \\
\quad \quad \quad \quad \text{room\_num} += 1 \\
\quad \quad \quad \quad F.\text{add}(\text{room\_num}) \\
\quad \quad \quad B.\text{add}(F.\text{getFreeRoom}()) \\
\text{return} \ F.\text{size}() 
\]

\[F = \{\} \]
\[B = \{\} \]
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
done
    foreach a in A
done
        if a.finish == t
            F.add(B.getBookedRoom())
        endforeach a in A
done
        foreach a in A
done
            if a.start == t
                if F.isEmpty()
                    room_num += 1
                    F.add(room_num)
                    B.add(F.getFreeRoom())
                endif F.isEmpty()
                return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())
    foreach a in A
        if a.start == t
            if F.isEmpty()
                room_num += 1
                F.add(room_num)
            B.add(F.getFreeRoom())

return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0
foreach timeslot t
  foreach a in A
    if a.finish == t
      F.add(B.getBookedRoom())
  foreach a in A
    if a.start == t
      if F.isEmpty()
        room_num += 1
        F.add(room_num)
        B.add(F.getFreeRoom())
  return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())
    foreach a in A
        if a.start == t
            if F.isEmpty()
                room_num += 1
                F.add(room_num)
            B.add(F.getFreeRoom())

return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty

room_num = 0

foreach timeslot t

    foreach a in A

        if a.finish == t

            F.add(B.getBookedRoom())

    foreach a in A

        if a.start == t

            if F.isEmpty()

                room_num += 1

                F.add(room_num)

                B.add(F.getFreeRoom())

return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
  foreach a in A
    if a.finish == t
      F.add(B.getBookedRoom())

foreach a in A
  if a.start == t
    if F.isEmpty()
      room_num += 1
      F.add(room_num)
      B.add(F.getFreeRoom())

return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())
    foreach a in A
        if a.start == t
            if F.isEmpty()
                room_num += 1
                F.add(room_num)
            B.add(F.getFreeRoom())
    return F.size()
Room Minimization

\texttt{room\_minimization2(activities A)}

\begin{verbatim}
B = F = empty
room_num = 0
foreach timeslot t
    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())
    foreach a in A
        if a.start == t
            if F.isEmpty()
                room_num += 1
                F.add(room_num)
            B.add(F.getFreeRoom())
return F.size()
\end{verbatim}

\textbf{Room Minimization}

\begin{align*}
    F &= \{\} \\
    B &= \{1\}
\end{align*}

\textbf{B} = Occupied Rooms \\
\textbf{F} = Unoccupied Rooms \\
\textbf{Never Used Rooms}
room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
  foreach a in A
    if a.finish == t
      F.add(B.getBookedRoom())
  foreach a in A
    if a.start == t
      if F.isEmpty()
        room_num += 1
        F.add(room_num)
        F.add(F.getFreeRoom())
  B.add(F.getFreeRoom())

return F.size()
Room Minimization

\[\text{room\_minimization2(activities } A)\]

\(B = F = \text{empty}\)

\(\text{room\_num} = 0\)

\textbf{foreach} timeslot \(t\)

\textbf{foreach} \(a\) in \(A\)

\textbf{if} \(a.\text{finish} == t\)

\(F.\text{add}(B.\text{getBookedRoom}())\)

\textbf{foreach} \(a\) in \(A\)

\textbf{if} \(a.\text{start} == t\)

\textbf{if} \(F.\text{isEmpty}()\)

\(\text{room\_num} += 1\)

\(F.\text{add}(\text{room\_num})\)

\(B.\text{add}(F.\text{getFreeRoom}())\)

\textbf{return} \(F.\text{size}()\)

---

- \(B = \text{Occupied Rooms}\)
- \(F = \text{Unoccupied Rooms}\)
- \(\text{Never Used Rooms}\)

![Diagram](image)
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())
    endforeach

foreach a in A
    if a.start == t
        if F.isEmpty()
            room_num += 1
            F.add(room_num)
        F.add(B.getFreeRoom())
    endforeach

return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0
foreach timeslot t
  foreach a in A
    if a.finish == t
      F.add(B.getBookedRoom())
  foreach a in A
    if a.start == t
      if F.isEmpty()
        room_num += 1
        F.add(room_num)
        B.add(F.getFreeRoom())
return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0
foreach timeslot t
    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())
    foreach a in A
        if a.start == t
            if F.isEmpty()
                room_num += 1
                F.add(room_num)
            B.add(F.getFreeRoom())
return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
    
    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())

    foreach a in A
        if a.start == t
            if F.isEmpty()
                room_num += 1
                F.add(room_num)
            F.add(F.getFreeRoom())

return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t

    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())

    foreach a in A
        if a.start == t
            if F.isEmpty()
                room_num += 1
                F.add(room_num)
            B.add(F.getFreeRoom())

return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())

    foreach a in A
        if a.start == t
            if F.isEmpty()
                room_num += 1
                F.add(room_num)
            F.add(F.getFreeRoom())
            B.add(F.getFreeRoom())

return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
  foreach a in A
    if a.finish == t
      F.add(B.getBookedRoom())

foreach a in A
  if a.start == t
    if F.isEmpty()
      room_num += 1
      F.add(room_num)
      B.add(F.getFreeRoom())

return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0
foreach timeslot t
    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())

    foreach a in A
        if a.start == t
            if F.isEmpty()
                room_num += 1
                F.add(room_num)
                B.add(F.getFreeRoom())

return F.size()

Room Minimization

\[ F = \{ 2 \} \]

\[ B = \{ 1 \} \]
Room Minimization

```plaintext
room_minimization2(activities A)
    B = F = empty
    room_num = 0
    foreach timeslot t
        foreach a in A
            if a.finish == t
                F.add(B.getBookedRoom())
        foreach a in A
            if a.start == t
                if F.isEmpty()
                    room_num += 1
                    F.add(room_num)
                B.add(F.getFreeRoom())
    return F.size()
```

- $B = \text{Occupied Rooms}$
- $F = \text{Unoccupied Rooms}$
- Never Used Rooms
Room Minimization

room_minimization2(activities A)

\[ B = F = \text{empty} \]

room_num = 0

\[
\begin{align*}
\text{foreach timeslot } t & \\
\quad & \text{foreach } a \text{ in } A \\
\quad & \quad \text{if } a.\text{finish} == t \\
\quad & \quad \quad F.\text{add}(B.\text{getBookedRoom}())
\end{align*}
\]

\[
\begin{align*}
\text{foreach } a \text{ in } A & \\
\quad & \text{if } a.\text{start} == t \\
\quad & \quad \quad \text{if } F.\text{isEmpty}() \\
\quad & \quad \quad \quad \text{room_num} += 1 \\
\quad & \quad \quad \quad F.\text{add}(\text{room_num}) \\
\quad & \quad \quad B.\text{add}(F.\text{getFreeRoom}())
\end{align*}
\]

\[ \text{return } F.\text{size}() \]

\[ B = \text{Occupied Rooms} \]
\[ F = \text{Unoccupied Rooms} \]
\[ \text{Never Used Rooms} \]
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0
foreach timeslot t
  foreach a in A
    if a.finish == t
      F.add(B.getBookedRoom())
  foreach a in A
    if a.start == t
      if F.isEmpty()
        room_num += 1
        F.add(room_num)
        B.add(F.getFreeRoom())
return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0
foreach timeslot t
  foreach a in A
    if a.finish == t
      F.add(B.getBookedRoom())
  foreach a in A
    if a.start == t
      if F.isEmpty()
        room_num += 1
        F.add(room_num)
        B.add(F.getFreeRoom())
  return F.size()
Room Minimization

game_minimization2(activities A)

B = F = empty
room_num = 0

forall timeslot t

forall a in A
    if a.finish == t
        F.add(B.getBookedRoom())

forall a in A
    if a.start == t
        if F.isEmpty()
            room_num += 1
            F.add(room_num)
        B.add(F.getFreeRoom())

return F.size()
Room Minimization

room_minimization2(activities A)

\[
B = F = \text{empty} \\
\text{room\_num} = 0
\]

\text{foreach timeslot } t

\text{foreach } a \text{ in } A

\text{if } a.\text{finish} == t

\hspace{1em} F.\text{add}(B.\text{getBookedRoom}())

\text{foreach } a \text{ in } A

\text{if } a.\text{start} == t

\hspace{2em} \text{if } F.\text{isEmpty}()

\hspace{3em} \text{room\_num} += 1

\hspace{3em} F.\text{add}(\text{room\_num})

\hspace{2em} B.\text{add}(F.\text{getFreeRoom}())

\text{return } F.\text{size}()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
  foreach a in A
    if a.finish == t
      F.add(B.getBookedRoom())

foreach a in A
  if a.start == t
    if F.isEmpty()
      room_num += 1
      F.add(room_num)
      B.add(F.getFreeRoom())

return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t

datascope a in A
    if a.finish == t
        F.add(B.getBookedRoom())

datascope a in A
    if a.start == t
        if F.isEmpty()
            room_num += 1
            F.add(room_num)
        F.add(B.getFreeRoom())

return F.size()
Room Minimization

\[ \text{room\_minimization2(activities A)} \]

\[ B = F = \text{empty} \]

\[ \text{room\_num} = 0 \]

\[ \text{foreach timeslot } t \]

\[ \text{foreach } a \text{ in } A \]

\[ \text{if } a.\text{finish} == t \]

\[ F.\text{add}(B.\text{getBookedRoom}()) \]

\[ \text{foreach } a \text{ in } A \]

\[ \text{if } a.\text{start} == t \]

\[ \text{if } F.\text{isEmpty}() \]

\[ \text{room\_num} += 1 \]

\[ F.\text{add}(\text{room\_num}) \]

\[ B.\text{add}(F.\text{getFreeRoom}()) \]

\[ \text{return } F.\text{size}() \]

Running Time?

\[ O(|A|^2) \] — Don’t need to go through all timeslots. Just start/finish for each activity.

\[ F = \{1,2\} \]

\[ B = \{\} \]
Room Minimization

\text{room\_minimization2(activities A)}

\begin{align*}
B &= F = \text{empty} \\
\text{room\_num} &= 0 \\
\text{foreach} & \text{ timeslot } t \\
\quad & \text{foreach } a \text{ in } A \\
\quad & \quad \text{if } a.\text{finish} == t \\
\quad & \quad \quad F.\text{add}(B.\text{getBookedRoom}()) \\
\quad & \text{foreach } a \text{ in } A \\
\quad & \quad \text{if } a.\text{start} == t \\
\quad & \quad \quad \text{if } F.\text{isEmpty}() \\
\quad & \quad \quad \quad \text{room\_num} += 1 \\
\quad & \quad \quad \quad F.\text{add}(\text{room\_num}) \\
\quad & \quad \quad B.\text{add}(F.\text{getFreeRoom}()) \\
\text{return } F.\text{size}()
\end{align*}

\[ F = \{1, 2\} \]

\[ B = \{\} \]
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())

foreach a in A
    if a.start == t
        if F.isEmpty()
            room_num += 1
            F.add(room_num)
        B.add(F.getFreeRoom())

return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t

    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())

    foreach a in A
        if a.start == t
            if F.isEmpty()
                room_num += 1
                F.add(room_num)
                B.add(F.getFreeRoom())

return F.size()
Room Minimization

room_minimization2(activities A)

B = F = empty
room_num = 0

foreach timeslot t
    foreach a in A
        if a.finish == t
            F.add(B.getBookedRoom())
    foreach a in A
        if a.start == t
            if F.isEmpty()
                room_num += 1
                F.add(room_num)
            B.add(F.getFreeRoom())

return F.size()

Optimal?
Yes. Min # rooms needed ≥ # concurrent courses. New room added ⇔ all other rooms occupied (i.e. concurrent courses).

F = \{1,2\}
B = \{\}
Client Scheduling

• Suppose you are a plumber and you have a list of clients that want help.

• Each client $i$ has a deadline $d_i$ of when they need help by and an amount of time $t_i$ they will need help for.

• You cannot help multiple clients at the same time and cannot pause helping one to help another.

• You need to help all clients, even if their help goes over their deadline (though they will be angry then).

• We need to come up with a schedule that minimizes the lateness of the latest client.
Notation/Example

- $d_i$: deadline for client $i$.
- $t_i$: time required for client $i$.
- $s(i)$: start time for client $i$.
- $f(i) = s(i) + t_i$: finish time for client $i$.
- $l_i = f(i) - d_i$: lateness for client $i$.
- $L = \max_i l_i$: maximum lateness.

Client 1:
- $d_1 = 3$
- $t_1 = 2$

Client 2:
- $d_2 = 7$
- $t_2 = 3$

Client 3:
- $d_3 = 10$
- $t_3 = 5$

Schedule:
Greedy Decision

What are some possible Greedy decisions?
Greedy Decision

What are some possible Greedy decisions?

• Smallest $t_i$ first.

• Smallest slack time $(d_i - t_i)$ first.

• Earliest $d_i$ first.

How to decide which to use?
Greedy Decision

What are some possible Greedy decisions?

• Smallest $t_i$ first.

• Smallest slack time $(d_i - t_i)$ first.

• Earliest $d_i$ first.

How to decide which to use? Hunt for counterexamples.
Greedy Decision

What are some possible Greedy decisions?

• Smallest $t_i$ first.
  
  \[ d_1 = 100, t_1 = 1 \text{ and } d_2 = 10, t_2 = 10 \]
  
  \[ \Rightarrow L_{ALG} = 1 \text{ vs } L_{OPT} = 0 \]

• Smallest slack time $(d_i - t_i)$ first.
  
  \[ d_1 = 2, t_1 = 1 \text{ and } d_2 = 10, t_2 = 10 \]
  
  \[ \Rightarrow L_{ALG} = 9 \text{ vs } L_{OPT} = 1 \]

• Earliest $d_i$ first.
Earliest Deadline First Algorithm

1. Order clients by increasing deadline.
2. Rename so that \( d_1 \leq \cdots \leq d_n \).
3. Let \( s(1) = 0 \Rightarrow f(1) = t_1 \).
4. For each subsequent (in order) client \( c \),
   \[
   s(c) = f(c - 1) \text{ and } f(c) = s(c) + t_c .
   \]
Earliest Deadline First Algorithm

1. Order clients by increasing deadline.
2. Rename so that $d_1 \leq \cdots \leq d_n$.
3. Let $s(1) = 0 \Rightarrow f(1) = t_1$.
4. For each subsequent (in order) client $c$,
   $$s(c) = f(c - 1) \text{ and } f(c) = s(c) + t_c.$$
Earliest Deadline First Algorithm

**Theorem:** The maximum lateness given by a schedule from the EDF algorithm is optimal.

**Plan of attack:**
Earliest Deadline First Algorithm

**Theorem:** The maximum lateness given by a schedule from the EDF algorithm is optimal.

**Plan of attack:** Consider an optimal schedule, modify it in such a way that optimality is preserved until it is the same as our schedule.

How could our schedule differ from optimal?
Earliest Deadline First Algorithm

**Theorem:** The maximum lateness given by a schedule from the EDF algorithm is optimal.

**Plan of attack:** Consider an optimal schedule, modify it in such a way that optimality is preserved until it is the same as our schedule.

How could our schedule differ from optimal?

1. Gaps in schedules.
2. Clients out of order.
Earliest Deadline First Algorithm

1. Order clients by increasing deadline.
2. Rename so that $d_1 \leq \ldots \leq d_n$.
3. Let $s(1) = 0 \Rightarrow f(1) = t_1$.
4. For each subsequent (in order) client $c$,
   \[ s(c) = f(c - 1) \quad \text{and} \quad f(c) = s(c) + t_c. \]

Could there be gaps in our schedule?
Earliest Deadline First Algorithm

1. Order clients by increasing deadline.
2. Rename so that \( d_1 \leq \cdots \leq d_n \).
3. Let \( s(1) = 0 \Rightarrow f(1) = t_1 \).
4. For each subsequent (in order) client \( c \),
   \[
   s(c) = f(c - 1) \quad \text{and} \quad f(c) = s(c) + t_c.
   \]

Could there be gaps in our schedule?
No – as soon as one client is finished, \( f(c) \), the next client starts, \( s(c + 1) = f(c) \).
Earliest Deadline First Algorithm

Lemma: An optimal schedule exists that has no gaps between clients.

Proof: ?
Earliest Deadline First Algorithm

**Lemma:** An optimal schedule exists that has no gaps between clients.

**Proof:** Since all clients are available to start at the same time, they can be scheduled one after the other without gaps. If an optimal schedule exists with gaps, those gaps can be removed by shifting clients forward without increasing the maximum lateness.
Earliest Deadline First Algorithm

**Definition:** A schedule has an *inversion* if some client is scheduled before a client with an earlier deadline.

Client 1: \[d_1 = 5, \quad t_1 = 2\]

Client 2: \[d_2 = 6, \quad t_2 = 3\]

**No Inversion:**

**Inversion:**
Earliest Deadline First Algorithm

**Definition:** A schedule has an *inversion* if some client is scheduled before a client with an earlier deadline.

Client 1:  
\[ d_1 = 5 \quad t_1 = 2 \]

Client 2:  
\[ d_2 = 6 \quad t_2 = 3 \]  

Does our schedule have any inversions?

**No Inversion:**

<table>
<thead>
<tr>
<th></th>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Inversion:**

<table>
<thead>
<tr>
<th></th>
<th>Client 2</th>
<th>Client 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Earliest Deadline First Algorithm

**Definition:** A schedule has an *inversion* if some client is scheduled before a client with an earlier deadline.

Client 1: \(d_1 = 5\) \(t_1 = 2\)

Client 2: \(d_2 = 6\) \(t_2 = 3\)

Does our schedule have any inversions?

No – The algorithm schedules client \(i\) before client \(j\) if \(d_i \leq d_j\)

No Inversion:

<table>
<thead>
<tr>
<th></th>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Inversion:

<table>
<thead>
<tr>
<th></th>
<th>Client 2</th>
<th>Client 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
Earliest Deadline First Algorithm

**Lemma:** An optimal schedule exists that has no inversions.

**Proof:**
Earliest Deadline First Algorithm

Lemma: An optimal schedule exists that has no inversions.

Proof:

1. Any inversion results from two consecutive inverted clients.
2. Swapping an inversion reduces the number of inversions.
3. Swapping an inversion does not increase the maximum lateness of the schedule.
3. Swapping an inversion does not increase the maximum lateness of the schedule.

**Proof:** Whose lateness does swapping consecutive clients affect?
3. Swapping an inversion does not increase the maximum lateness of the schedule.

**Proof:** The lateness is less for the client swapped earlier, later for the client swapped later, and the same for all other clients (because inverted $f(i) = \text{swapped } f(j)$).

<table>
<thead>
<tr>
<th>Inverted:</th>
<th>Client j</th>
<th>Client i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Swapped:</th>
<th>Client i</th>
<th>Client j</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. Swapping an inversion does not increase the maximum lateness of the schedule.

**Proof:** The only client we need to consider is the one swapped to be later (because all other clients remain the same or have smaller lateness).
Earliest Deadline First Algorithm

3. Swapping an inversion does not increase the maximum lateness of the schedule.

**Proof:** The only client we need to consider is the one swapped to be later (because all other clients remain the same or have smaller lateness).

Let $s(i), f(i), l_i$ be for inverted schedule.
Let $s'(i), f'(i), l_i'$ be for swapped schedule.
Earliest Deadline First Algorithm

3. Swapping an inversion does not increase the maximum lateness of the schedule.

Proof: The only client we need to consider is the one swapped to be later (because all other clients remain the same or have smaller lateness).

Let $s(i), f(i), l_i$ be for inverted schedule.
Let $s'(i), f''(i), l_i'$ be for swapped schedule.

$$l_i' \geq l_i$$

Remember:
- $l_j = f(j) - d_j$
- $d_i \leq d_j$
Earliest Deadline First Algorithm

3. Swapping an inversion does not increase the maximum lateness of the schedule.

**Proof:** The only client we need to consider is the one swapped to be later (because all other clients remain the same or have smaller lateness).

Let \( s(i), f(i), l_i \) be for inverted schedule.
Let \( s'(i), f'(i), l'_i \) be for swapped schedule.

\[
l'_j = f'(j) - d_j = f(i) - d_j \leq f(i) - d_i = l_i
\]

<table>
<thead>
<tr>
<th>Inverted:</th>
<th>Client j</th>
<th>Client i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swapped:</td>
<td>Client i</td>
<td>Client j</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remember:
- \( l_j = f(j) - d_j \)
- \( d_i \leq d_j \)
Earliest Deadline First Algorithm

3. Swapping an inversion does not increase the maximum lateness of the schedule.

**Proof:** The only client we need to consider is the one swapped to be later (because all other clients remain the same or have smaller lateness).

Let $s(i), f(i), l_i$ be for inverted schedule.
Let $s'(i), f'(i), l_i'$ be for swapped schedule.

$$l_j' = f'(j) - d_j = f(i) - d_j \leq f(i) - d_i = l_i$$

Since the only lateness that was changed does not increase, the maximum lateness was not affected.
Theorem: The maximum lateness given by a schedule from the EDF algorithm is optimal.

Proof: The EDF schedule can only differ from an optimal schedule by the order of clients with identical deadlines (since both have no gaps or inversions).

Does the ordering of these clients lead to different maximal lateness?
Theorem: The maximum lateness given by a schedule from the EDF algorithm is optimal.

Proof: The EDF schedule can only differ from an optimal schedule by the order of clients with identical deadlines (since both have no gaps or inversions).

Clients with identical deadlines $d$ are all scheduled consecutively. The client with the largest lateness ($l_i = f(i) - d$) is the one with the latest finish time (regardless of order).
Earliest Deadline First Algorithm

**Theorem**: The maximum lateness given by a schedule from the EDF algorithm is optimal.

**Proof**: The EDF schedule can only differ from an optimal schedule by the order of clients with identical deadlines (since both have no gaps or inversions).

Clients with identical deadlines $d$ are all scheduled consecutively. The client with the largest lateness ($l_i = f(i) - d$) is the one with the latest finish time (regardless of order).

Thus, all schedules with no inversions or gaps have the same maximal lateness.

Therefore, the EDF schedule is optimal.