Lemma: It is safe to schedule the job \( j \) with the earliest finish time: There is an optimum solution where the job \( j \) with the earliest finish time is scheduled.

Proof:
- Take an arbitrary optimum solution \( S \)
- If it contains \( j \), done
- Otherwise, replace the first job in \( S \) with \( j \) to obtain another optimum schedule \( S' \).
**Lemma** It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

- What is the remaining task after we decided to schedule $j$?
- Is it another instance of interval scheduling problem?
Lemma  It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

- What is the remaining task after we decided to schedule $j$?
- Is it another instance of interval scheduling problem?  Yes!
**Lemma** It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

- What is the remaining task after we decided to schedule $j$?
- Is it another instance of interval scheduling problem? Yes!
Greedy Algorithm for Interval Scheduling

**Lemma** It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

- What is the remaining task after we decided to schedule $j$?
- Is it another instance of interval scheduling problem? Yes!

![Interval scheduling diagram](image-url)
Greedy Algorithm for Interval Scheduling

**Schedule**\( (s, f, n) \)

1. \( A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset \)
2. \textbf{while} \( A \neq \emptyset \) \textbf{do}
3. \quad \( j \leftarrow \arg \min_{j' \in A} f_{j'} \)
4. \quad \( S \leftarrow S \cup \{j\}; A \leftarrow \{j' \in A : s_{j'} \geq f_j\} \)
5. \textbf{return} \( S \)
Greedy Algorithm for Interval Scheduling

Schedule\((s, f, n)\)

1. \(A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset\)
2. \textbf{while} \(A \neq \emptyset\) \textbf{do}
3. \quad \(j \leftarrow \arg\min_{j' \in A} f_{j'}\)
4. \quad \(S \leftarrow S \cup \{j\}; A \leftarrow \{j' \in A : s_{j'} \geq f_j\}\)
5. \textbf{return} \(S\)
Greedy Algorithm for Interval Scheduling

**Schedule**$(s, f, n)$

1: $A \leftarrow \{1, 2, \cdots, n\}, \; S \leftarrow \emptyset$
2: while $A \neq \emptyset$ do
3: $\; j \leftarrow \text{arg min}_{j' \in A} f_{j'}$
4: $\; S \leftarrow S \cup \{j\}; \; A \leftarrow \{j' \in A : s_{j'} \geq f_j\}$
5: return $S$
Greedy Algorithm for Interval Scheduling

\textbf{Schedule}(s, f, n)

1: \(A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset\)
2: \textbf{while} \(A \neq \emptyset\) \textbf{do}
3: \(j \leftarrow \arg \min_{j' \in A} f_{j'}\)
4: \(S \leftarrow S \cup \{j\}; A \leftarrow \{j' \in A : s_{j'} \geq f_j\}\)
5: \textbf{return} \(S\)
Greedy Algorithm for Interval Scheduling

Schedule \((s, f, n)\)

1. \(A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset\)
2. while \(A \neq \emptyset\) do
3. \(j \leftarrow \arg \min_{j' \in A} f_{j'}\)
4. \(S \leftarrow S \cup \{j\}; \ A \leftarrow \{j' \in A : s_{j'} \geq f_j\}\)
5. return \(S\)
Greedy Algorithm for Interval Scheduling

\[\text{Schedule}(s, f, n)\]

1. \(A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset\)
2. \textbf{while} \(A \neq \emptyset\) \textbf{do}
3. \(j \leftarrow \arg\min_{j' \in A} f_{j'}\)
4. \(S \leftarrow S \cup \{j\}; A \leftarrow \{j' \in A : s_{j'} \geq f_j\}\)
5. \textbf{return} \(S\)
Greedy Algorithm for Interval Scheduling

**Schedule**\((s, f, n)\)

1: \(A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset\)
2: while \(A \neq \emptyset\) do
3: \(j \leftarrow \text{arg min}_{j' \in A} f_{j'}\)
4: \(S \leftarrow S \cup \{j\}; A \leftarrow \{j' \in A : s_{j'} \geq f_j\}\)
5: return \(S\)

Running time of algorithm?
Greedy Algorithm for Interval Scheduling

Schedule($s, f, n$)

1: $A \leftarrow \{1, 2, \cdots, n\}$, $S \leftarrow \emptyset$
2: while $A \neq \emptyset$ do
3: $j \leftarrow \text{arg min}_{j' \in A} f_{j'}$
4: $S \leftarrow S \cup \{j\}$; $A \leftarrow \{j' \in A : s_{j'} \geq f_j\}$
5: return $S$

Running time of algorithm?

- Naive implementation: $O(n^2)$ time
Greedy Algorithm for Interval Scheduling

Schedule($s$, $f$, $n$)

1: $A \leftarrow \{1, 2, \ldots, n\}$, $S \leftarrow \emptyset$
2: while $A \neq \emptyset$ do
3: \hspace{1em} $j \leftarrow \arg\min_{j' \in A} f_{j'}$
4: \hspace{1em} $S \leftarrow S \cup \{j\}$; $A \leftarrow \{j' \in A : s_{j'} \geq f_j\}$
5: return $S$

Running time of algorithm?

- Naive implementation: $O(n^2)$ time
- Clever implementation: $O(n \log n)$ time
Clever Implementation of Greedy Algorithm

**Schedule**$(s, f, n)$

1. sort jobs according to $f$ values
2. $t \leftarrow 0$, $S \leftarrow \emptyset$
3. for every $j \in [n]$ according to non-decreasing order of $f_j$ do
   4. if $s_j \geq t$ then
   5. $S \leftarrow S \cup \{j\}$
   6. $t \leftarrow f_j$
7. return $S$
Clever Implementation of Greedy Algorithm

Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
4: \quad if \(s_j \geq t\) then
5: \quad \quad \(S \leftarrow S \cup \{j\}\)
6: \quad \quad \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
4: \hspace{1em} if \(s_j \geq t\) then
5: \hspace{2em} \(S \leftarrow S \cup \{j\}\)
6: \hspace{2em} \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

Schedule$(s, f, n)$

1: sort jobs according to $f$ values
2: $t \leftarrow 0$, $S \leftarrow \emptyset$
3: for every $j \in [n]$ according to non-decreasing order of $f_j$ do
4:  \hspace{1em} if $s_j \geq t$ then
5:  \hspace{2em} $S \leftarrow S \cup \{j\}$
6:  \hspace{2em} $t \leftarrow f_j$
7: return $S$
Clever Implementation of Greedy Algorithm

**Schedule** \((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
   4:     if \(s_j \geq t\) then
   5:         \(S \leftarrow S \cup \{j\}\)
   6:         \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

Schedule($s, f, n$)

1: sort jobs according to $f$ values
2: $t \leftarrow 0$, $S \leftarrow \emptyset$
3: for every $j \in [n]$ according to non-decreasing order of $f_j$ do
   4: if $s_j \geq t$ then
      5: $S \leftarrow S \cup \{j\}$
      6: $t \leftarrow f_j$
7: return $S$
Clever Implementation of Greedy Algorithm

Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, \ S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
   4:     if \(s_j \geq t\) then
   5:         \(S \leftarrow S \cup \{j\}\)
   6:         \(t \leftarrow f_j\)
4: return \(S\)
Clever Implementation of Greedy Algorithm

\[
\text{Schedule}(s, f, n)
\]

1. sort jobs according to \( f \) values
2. \( t \leftarrow 0, S \leftarrow \emptyset \)
3. for every \( j \in [n] \) according to non-decreasing order of \( f_j \) do
4. \hspace{1em} if \( s_j \geq t \) then
5. \hspace{2em} \( S \leftarrow S \cup \{j\} \)
6. \hspace{2em} \( t \leftarrow f_j \)
7. return \( S \)
Clever Implementation of Greedy Algorithm

**Schedule**\((s, f, n)\)

1. sort jobs according to \(f\) values
2. \(t \leftarrow 0, S \leftarrow \emptyset\)
3. **for** every \(j \in [n]\) according to non-decreasing order of \(f_j\) **do**
   4. **if** \(s_j \geq t\) **then**
   5. \(S \leftarrow S \cup \{j\}\)
   6. \(t \leftarrow f_j\)
7. **return** \(S\)
Clever Implementation of Greedy Algorithm

Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
4: \(\text{if } s_j \geq t \text{ then}\)
5: \(S \leftarrow S \cup \{j\}\)
6: \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
4: \hspace{1em} if \(s_j \geq t\) then
5: \hspace{2em} \(S \leftarrow S \cup \{j\}\)
6: \hspace{2em} \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

\[ \text{Schedule}(s, f, n) \]

1: sort jobs according to \( f \) values
2: \( t \leftarrow 0, S \leftarrow \emptyset \)
3: for every \( j \in [n] \) according to non-decreasing order of \( f_j \) do
4: \hspace{0.5cm} if \( s_j \geq t \) then
5: \hspace{1cm} \( S \leftarrow S \cup \{j\} \)
6: \hspace{1cm} \( t \leftarrow f_j \)
7: return \( S \)
Clever Implementation of Greedy Algorithm

Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
4: \hspace{1em} if \(s_j \geq t\) then
5: \hspace{2em} \(S \leftarrow S \cup \{j\}\)
6: \hspace{2em} \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

**Schedule**\((s, f, n)\)

1: sort jobs according to \(f\) values  
2: \(t \leftarrow 0, S \leftarrow \emptyset\)  
3: **for** every \(j \in [n]\) according to non-decreasing order of \(f_j\) **do**  
4: \hspace{1em} **if** \(s_j \geq t\) **then**  
5: \hspace{2em} \(S \leftarrow S \cup \{j\}\)  
6: \hspace{2em} \(t \leftarrow f_j\)  
7: **return** \(S\)
Clever Implementation of Greedy Algorithm

Schedule(\(s, f, n\))

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0\), \(S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
4: \hspace{1em} if \(s_j \geq t\) then
5: \hspace{2em} \(S \leftarrow S \cup \{j\}\)
6: \hspace{1em} \(t \leftarrow f_j\)
7: return \(S\)
Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
   4: if \(s_j \geq t\) then
   5: \(S \leftarrow S \cup \{j\}\)
   6: \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
4: \(\text{if } s_j \geq t \text{ then}\)
5: \(S \leftarrow S \cup \{j\}\)
6: \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

**Schedule**$(s, f, n)$

1: sort jobs according to $f$ values
2: $t \leftarrow 0$, $S \leftarrow \emptyset$
3: **for** every $j \in [n]$ according to non-decreasing order of $f_j$ **do**
4:  **if** $s_j \geq t$ **then**
5:     $S \leftarrow S \cup \{j\}$
6:     $t \leftarrow f_j$
7: **return** $S$
Outline

1. Toy Example: Box Packing
2. Interval Scheduling
   - Interval Partitioning
3. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue
4. Data Compression and Huffman Code
5. Summary
6. Exercise Problems
Interval Partitioning

**Input:** $n$ jobs, job $i$ with start time $s_i$ and finish time $f_i$.

$i$ and $j$ are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint.

**Output:** A minimum number of machines to schedule all jobs so that all jobs on a single machine are compatible.
Interval Partitioning

**Input:** \( n \) jobs, job \( i \) with start time \( s_i \) and finish time \( f_i \)

\( i \) and \( j \) are compatible if \([s_i, f_i)\) and \([s_j, f_j)\) are disjoint

**Output:** A minimum number of machines to schedule all jobs so that all jobs on a single machine are compatible.
Interval Partitioning

**Input:** \( n \) jobs, job \( i \) with start time \( s_i \) and finish time \( f_i \).

\( i \) and \( j \) are compatible if \([s_i, f_i)\) and \([s_j, f_j)\) are disjoint.

**Output:** A minimum number of machines to schedule all jobs so that all jobs on a single machine are compatible.

![Interval Partitioning Diagram](image)
Interval Partitioning

**Input:** \( n \) jobs, job \( i \) with start time \( s_i \) and finish time \( f_i \)

\( i \) and \( j \) are compatible if \([s_i, f_i)\) and \([s_j, f_j)\) are disjoint

**Output:** A minimum number of machines to schedule all jobs so that all jobs on a single machine are compatible.
Interval Partitioning

**Input:** \( n \) jobs, job \( i \) with start time \( s_i \) and finish time \( f_i \)

\( i \) and \( j \) are compatible if \([s_i, f_i)\) and \([s_j, f_j)\) are disjoint

**Output:** A minimum number of machines to schedule all jobs so that all jobs on a single machine are compatible.
Interval Partitioning

**Input:** \( n \) jobs, job \( i \) with start time \( s_i \) and finish time \( f_i \)

\( i \) and \( j \) are compatible if \([s_i, f_i)\) and \([s_j, f_j)\) are disjoint

**Output:** A minimum number of machines to schedule all jobs so that all jobs on a single machine are compatible.
Interval Partitioning

**Input:** \( n \) jobs, job \( i \) with start time \( s_i \) and finish time \( f_i \)

\( i \) and \( j \) are compatible if \([s_i, f_i)\) and \([s_j, f_j)\) are disjoint

**Output:** A minimum number of machines to schedule all jobs so that all jobs on a single machine are compatible.
Interval Partitioning

**Input:** \( n \) jobs, job \( i \) with start time \( s_i \) and finish time \( f_i \). \( i \) and \( j \) are compatible if \( [s_i, f_i) \) and \( [s_j, f_j) \) are disjoint.

**Output:** A minimum number of machines to schedule all jobs so that all jobs on a single machine are compatible.
Interval Partitioning

**Input:** $n$ jobs, job $i$ with start time $s_i$ and finish time $f_i$

$i$ and $j$ are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

**Output:** A minimum number of machines to schedule all jobs so that all jobs on a single machine are compatible.
Lemma  It is safe to schedule the job \( j \) with the earliest starting time to a feasible machine: There exists an optimum solution where job \( j \) with the earliest starting time is scheduled first on a machine that is compatible with all jobs in that machine if applicable; otherwise, it can be scheduled by opening a new machine.

Proof.
**Lemma**  It is safe to schedule the job $j$ with the earliest starting time to a feasible machine: There exists an optimum solution where job $j$ with the earliest starting time is scheduled first on a machine that is compatible with all jobs in that machine if applicable; otherwise, it can be scheduled by opening a new machine.

**Proof.**

- Take an arbitrary optimum solution $S'$
**Lemma**  It is safe to schedule the job \( j \) with the earliest starting time to a feasible machine: There exists an optimum solution where job \( j \) with the earliest starting time is scheduled first on a machine that is compatible with all jobs in that machine if applicable; otherwise, it can be scheduled by opening a new machine.

**Proof.**
- Take an arbitrary optimum solution \( S' \)
- If it schedules \( j \) to the chosen feasible machine \( i \), done
**Lemma**  It is safe to schedule the job $j$ with the earliest starting time to a feasible machine: There exists an optimum solution where job $j$ with the earliest starting time is scheduled first on a machine that is compatible with all jobs in that machine if applicable; otherwise, it can be scheduled by opening a new machine.

**Proof.**

- Take an arbitrary optimum solution $S'$
- If it schedules $j$ to the chosen feasible machine $i$, done
Lemma  It is safe to schedule the job $j$ with the earliest starting time to a feasible machine: There exists an optimum solution where job $j$ with the earliest starting time is scheduled first on a machine that is compatible with all jobs in that machine if applicable; otherwise, it can be scheduled by opening a new machine.

Proof.

- Take an arbitrary optimum solution $S'$
- If it schedules $j$ to the chosen feasible machine $i$, done
- Otherwise, replace all the jobs scheduled to the machine $i$ in $S$ with $j$ and its subsequent jobs to obtain another optimum schedule $S'$. 

Greedy Algorithm for Interval Partitioning

- What is the remaining task after we decided to schedule $j$?
- Is it another instance of interval partitioning problem?
Greedy Algorithm for Interval Partitioning

- What is the remaining task after we decided to schedule $j$?
- Is it another instance of interval partitioning problem? Yes!

![Diagram of interval partitioning]

- $j$
Greedy Algorithm for Interval Partitioning

- What is the remaining task after we decided to schedule $j$?
- Is it another instance of interval partitioning problem? Yes!

![Diagram showing intervals and $j$]
Greedy Algorithm for Interval Partitioning

Partition($s, f, n$)

1: $A \leftarrow \{1, 2, \cdots, n\}$, $S \leftarrow \{1\}$, $t_1 = 0$
2: while $A \neq \emptyset$ do
3: $j \leftarrow \arg\min_{j' \in A} s_{j'}$, $S_j \leftarrow \{i'\} \forall i' \in S, t_{i'} \leq s_j$
4: If $S_j \neq \emptyset$, then schedule $j$ to a machine $i \in S_j$ and $t_i = f_j$
5: Otherwise, schedule $j$ to machine $|S| + 1$, $S \leftarrow S \cup \{|S| + 1\}$ and $t_{|S|} = f_j$
6: return $S$
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning

![Diagram showing intervals on a number line with a specific interval marked by an arrow labeled j.]
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning

The diagram illustrates the greedy algorithm for interval partitioning. Each interval is represented by a horizontal bar on the timeline. The algorithm makes a series of greedy choices to partition the intervals, aiming to minimize the number of partitions.

For example, the interval marked with the letter "j" is split at point 4, creating two new intervals. This process is repeated until all intervals are assigned to a partition.
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning

The diagram illustrates a greedy algorithm for interval partitioning. The intervals are represented as bars on a timeline. The algorithm selects intervals in a greedy manner, trying to minimize the number of intervals used to cover all the events.

The notation $j$ indicates a specific interval being considered in the algorithm.
Greedy Algorithm for Interval Partitioning

![Interval Partitioning Diagram]
Greedy Algorithm for Interval Partitioning

![Diagram showing intervals partitioned into subintervals]

- Intervals are represented on a number line.
- The algorithm selects intervals to be partitioned based on a greedy strategy.
- The figure illustrates the partitioning process for an interval marked as $j$. 
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning

Diagram showing intervals from 0 to 9, with some intervals shaded in blue and others in orange. The interval $j$ is indicated on the left side of the diagram.
Greedy Algorithm for Interval Partitioning
Def. The **depth** of a set of jobs is the maximum number of overlapping jobs at any point within the given set.
**Def.** The **depth** of a set of jobs is the maximum number of overlapping jobs at any point within the given set.

**Obs.** The number of machines $\geq$ the depth of the jobs.
Greedy Algorithm for Interval Partitioning

**Def.** The depth of a set of jobs is the maximum number of overlapping jobs at any point within the given set.

**Obs.** The number of machines $\geq$ the depth of the jobs.

**Obs.** Greedy algorithm never schedules two incompatible jobs in the same machine.
Why “Greedy algorithm” is optimal?

**Theorem**  Greedy algorithm is optimal.

**Proof.**
- Let $d$ be the number of machines that greedy algorithm used.
Why “Greedy algorithm” is optimal?

**Theorem** Greedy algorithm is optimal.

**Proof.**
- Let \( d \) be the number of machines that greedy algorithm used.
- \( d \)-th machine is opened because the greedy algorithm need to schedule a job, wlog, say job \( j \), such that job \( j \) is incompatible with all the last scheduled jobs in the \( d - 1 \) other machines. In other words, these \( d - 1 \) job each ends after \( s_j \).
Why “Greedy algorithm” is optimal?

**Theorem** Greedy algorithm is optimal.

**Proof.**

- Let \( d \) be the number of machines that greedy algorithm used.
- \( d \)-th machine is opened because the greedy algorithm need to schedule a job, wlog, say job \( j \), such that job \( j \) is incompatible with all the last scheduled jobs in the \( d - 1 \) other machines. In other words, these \( d - 1 \) job each ends after \( s_j \).
- Observation: all these \( d - 1 \) jobs starts earlier than \( s_j \) because we schedule the jobs in order of starting time. Thus, we have \( d \) jobs overlapping at time \( s_j + \epsilon \). The jobs depth \( \geq d \).
Why “Greedy algorithm” is optimal?

**Theorem** Greedy algorithm is optimal.

**Proof.**
- Let \( d \) be the number of machines that greedy algorithm used.
- \( d \)-th machine is opened because the greedy algorithm need to schedule a job, wlog, say job \( j \), such that job \( j \) is incompatible with all the last scheduled jobs in the \( d - 1 \) other machines. In other words, these \( d - 1 \) job each ends after \( s_j \).
- Observation: all these \( d - 1 \) jobs starts earlier than \( s_j \) because we schedule the jobs in order of starting time. Thus, we have \( d \) jobs overlapping at time \( s_j + \epsilon \). The jobs **depth** \( \geq d \).
- By the Observation in the previous slide, an optimal solution \( \geq d \). Thus the greedy algorithm is optimal.
Partition \( (s, f, n) \)

1: \( A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \{1\}, t_1 = 0 \)
2: \textbf{while} \( A \neq \emptyset \) \textbf{do}
3: \( j \leftarrow \arg\min_{j' \in A} s_{j'}, S_j \leftarrow \{i'\}_{i' \in S, t_{i'} \leq s_j} \)
4: \textbf{If} \( S_j \neq \emptyset \), \textbf{then schedule} \( j \) to \textbf{a machine} \( i \in S_j \) \textbf{and} \( t_i = f_j \)
5: \textbf{Otherwise}, \textbf{schedule} \( j \) to \textbf{machine} \( |S| + 1 \), \( S \leftarrow S \cup \{|S| + 1\} \) \textbf{and} \( t_{|S|} = f_j \)
6: \textbf{return} \( S \)

Running time of algorithm?
Greedy Algorithm for Interval Partitioning

\[ \text{Partition}(s, f, n) \]

1: \( A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \{1\}, t_1 = 0 \)
2: \textbf{while} \( A \neq \emptyset \) \textbf{do}
3: \( j \leftarrow \text{arg min}_{j' \in A} s_{j'}, S_j \leftarrow \{i' \mid i' \in S, t_i \leq s_j\} \)
4: \text{If} \( S_j \neq \emptyset \), \text{then schedule} \( j \) \text{to a machine} \( i \in S_j \) \text{and} \( t_i = f_j \)
5: \text{Otherwise, schedule} \( j \) \text{to machine} \( |S| + 1 \), \( S \leftarrow S \cup \{|S| + 1\} \)
   \text{and} \( t_{|S|} = f_j \)
6: \textbf{return} \( S \)

Running time of algorithm?

- Naive implementation: \( O(n^2) \) time
Greedy Algorithm for Interval Partitioning

**Partition**($s$, $f$, $n$)

1. $A \leftarrow \{1, 2, \cdots, n\}$, $S \leftarrow \{1\}$, $t_1 = 0$
2. **while** $A \neq \emptyset$ **do**
3. 
   $j \leftarrow \arg \min_{j' \in A} s_{j'}$, $S_j \leftarrow \{i'\}_{i' \in S, t_{i'} \leq s_j}$
4. 
   If $S_j \neq \emptyset$, then schedule $j$ to a machine $i \in S_j$ and $t_i = f_j$
5. 
   Otherwise, schedule $j$ to machine $|S| + 1$, $S \leftarrow S \cup \{|S| + 1\}$
   and $t_{|S|} = f_j$
6. **return** $S$

---

Running time of algorithm?

- Naive implementation: $O(n^2)$ time
- Clever implementation: $O(n \log n)$ time with Priority Queue.
Outline

1. Toy Example: Box Packing
2. Interval Scheduling
   - Interval Partitioning
3. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue
4. Data Compression and Huffman Code
5. Summary
6. Exercise Problems
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests

Cache miss happens if requested page not in cache. We need to bring the page into cache, and evict some existing page if necessary. Cache hit happens if requested page already in cache. Goal: minimize the number of cache misses.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests

```
page sequence
1
5
4
2
5
3
2
1
```

```
cache
[ ] [ ] [ ]
```
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.

<table>
<thead>
<tr>
<th>page sequence</th>
<th>cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td>×</td>
</tr>
<tr>
<td>4</td>
<td>×</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
### Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.

<table>
<thead>
<tr>
<th>Page Sequence</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.

<table>
<thead>
<tr>
<th>page sequence</th>
<th>cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✔</td>
</tr>
<tr>
<td>5</td>
<td>✗ 1 ✔</td>
</tr>
<tr>
<td>4</td>
<td>✗ 1 ✔ 5</td>
</tr>
<tr>
<td>2</td>
<td>✗ 1 ✔ 5 4</td>
</tr>
<tr>
<td>3</td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td>✔</td>
</tr>
<tr>
<td>1</td>
<td>✔</td>
</tr>
</tbody>
</table>

Goal: minimize the number of cache misses.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.

<table>
<thead>
<tr>
<th>page sequence</th>
<th>cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x 1</td>
</tr>
<tr>
<td>5</td>
<td>x 1 5</td>
</tr>
<tr>
<td>4</td>
<td>x 1 5 4</td>
</tr>
<tr>
<td>2</td>
<td>x 1 2 4</td>
</tr>
<tr>
<td>5</td>
<td>x 1 2 5</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.

<table>
<thead>
<tr>
<th>page sequence</th>
<th>cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✗ 1</td>
</tr>
<tr>
<td>5</td>
<td>✗ 1</td>
</tr>
<tr>
<td>4</td>
<td>✗ 1</td>
</tr>
<tr>
<td>2</td>
<td>✗ 1</td>
</tr>
<tr>
<td>5</td>
<td>✗ 1</td>
</tr>
<tr>
<td>3</td>
<td>✗ 1</td>
</tr>
<tr>
<td>2</td>
<td>✗ 1</td>
</tr>
<tr>
<td>1</td>
<td>✓ 1</td>
</tr>
</tbody>
</table>
**Offline Caching**

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.

<table>
<thead>
<tr>
<th>page sequence</th>
<th>cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X 1</td>
</tr>
<tr>
<td>5</td>
<td>X 1</td>
</tr>
<tr>
<td>4</td>
<td>X 1</td>
</tr>
<tr>
<td>2</td>
<td>X 1</td>
</tr>
<tr>
<td>5</td>
<td>X 1</td>
</tr>
<tr>
<td>3</td>
<td>X 1</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

misses = 6
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.
- Goal: minimize the number of cache misses.

<table>
<thead>
<tr>
<th>Page sequence</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td>✗</td>
</tr>
<tr>
<td>4</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td>✗</td>
</tr>
<tr>
<td>3</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
</tr>
</tbody>
</table>

misses = 6
A Better Solution for Example

<table>
<thead>
<tr>
<th>page sequence</th>
<th>cache</th>
<th>cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✗ 1</td>
<td>✗ 1</td>
</tr>
<tr>
<td>5</td>
<td>✗ 1 5</td>
<td>✗ 1 5</td>
</tr>
<tr>
<td>4</td>
<td>✗ 1 5 4</td>
<td>✗ 1 5 4</td>
</tr>
<tr>
<td>2</td>
<td>✗ 1 2 4</td>
<td>✗ 1 5 2</td>
</tr>
<tr>
<td>5</td>
<td>✗ 1 2 5</td>
<td>✓ 1 5 2</td>
</tr>
<tr>
<td>3</td>
<td>✗ 1 2 3</td>
<td>✓ 1 3 2</td>
</tr>
<tr>
<td>2</td>
<td>✓ 1 2 3</td>
<td>✓ 1 3 2</td>
</tr>
<tr>
<td>1</td>
<td>✓ 1 2 3</td>
<td>✓ 1 3 2</td>
</tr>
</tbody>
</table>

misses = 6

misses = 5
Offline Caching Problem

**Input:**
- $k$: the size of cache
- $n$: number of pages
- $\rho_1, \rho_2, \rho_3, \ldots, \rho_T \in [n]$: sequence of requests

**Output:**
- $i_1, i_2, i_3, \ldots, i_T \in \{\text{hit, empty}\} \cup [n]$: indices of pages to evict ("hit" means evicting no page, "empty" means evicting empty page)

We use $[n]$ for $\{1, 2, 3, \ldots, n\}$.

Online Caching: we know the whole sequence ahead of time.

Online Caching: we have to make decisions on the fly, before seeing future requests.

Q: Which one is more realistic?
A: Online caching
## Offline Caching Problem

**Input:**
- \(k\): the size of cache
- \(n\): number of pages

\(\rho_1, \rho_2, \rho_3, \ldots, \rho_T \in [n]\): sequence of requests

**Output:**
- \(i_1, i_2, i_3, \ldots, i_T \in \{\text{hit}, \text{empty}\} \cup [n]\): indices of pages to evict ("hit" means evicting no page, "empty" means evicting empty page)

- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

We use \([n]\) for \(\{1, 2, 3, \ldots, n\}\).
**Offline Caching Problem**

**Input:** $k$ : the size of cache  
$n$ : number of pages  

$\rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n]$: sequence of requests

**Output:** $i_1, i_2, i_3, \cdots, i_T \in \{\text{hit, empty}\} \cup [n]$: indices of pages to evict (“hit” means evicting no page, “empty” means evicting empty page)

- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

Q: Which one is more realistic?
## Offline Caching Problem

**Input:**
- $k$: the size of cache
- $n$: number of pages

We use $[n]$ for $\{1, 2, 3, \cdots, n\}$.

**Output:**
- $\rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n]$: sequence of requests
- $i_1, i_2, i_3, \cdots, i_T \in \{\text{hit, empty}\} \cup [n]$: indices of pages to evict (“hit” means evicting no page, “empty” means evicting empty page)

### Offline Caching
- we know the whole sequence ahead of time.

### Online Caching
- we have to make decisions on the fly, before seeing future requests.

**Q:** Which one is more realistic?

**A:** Online caching
- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

**Q:** Which one is more realistic?

**A:** Online caching

**Q:** Why do we study the offline caching problem?
• Offline Caching: we know the whole sequence ahead of time.
• Online Caching: we have to make decisions on the fly, before seeing future requests.

Q: Which one is more realistic?

A: Online caching

Q: Why do we study the offline caching problem?

A: Use the offline solution as a benchmark to measure the “competitive ratio” of online algorithms
Offline Caching: Potential Greedy Algorithms

- **FIFO (First-In-First-Out):** Evict the first-in page in cache

Indeed all the algorithms are not optimum! Indeed all the algorithms are "online," i.e., the decisions can be made without knowing future requests. Online algorithms can not be optimum.
Offline Caching: Potential Greedy Algorithms

- **FIFO (First-In-First-Out):** Evict the first-in page in cache
- **LRU (Least-Recently-Used):** Evict page whose most recent access was earliest

All the above algorithms are not optimum! Indeed all the algorithms are "online", i.e., the decisions can be made without knowing future requests. Online algorithms cannot be optimum.
Offline Caching: Potential Greedy Algorithms

- **FIFO (First-In-First-Out):** Evict the first-in page in cache
- **LRU (Least-Recently-Used):** Evict page whose most recent access was earliest
- **LFU (Least-Frequently-Used):** Evict page that was least frequently requested

All the above algorithms are not optimum! Indeed all the algorithms are "online," i.e., the decisions can be made without knowing future requests. Online algorithms cannot be optimum.
Offline Caching: Potential Greedy Algorithms

- **FIFO (First-In-First-Out)**: Evict the first-in page in cache
- **LRU (Least-Recently-Used)**: Evict page whose most recent access was earliest
- **LFU (Least-Frequently-Used)**: Evict page that was least frequently requested
- **LIFO (Last In First Out)**: Evict the last-in page in cache

All the above algorithms are not optimum! Indeed all the algorithms are “online”, i.e., the decisions can be made without knowing future requests. Online algorithms cannot be optimum.
Offline Caching: Potential Greedy Algorithms

- FIFO (First-In-First-Out): Evict the first-in page in cache
- LRU (Least-Recently-Used): Evict page whose most recent access was earliest
- LFU (Least-Frequently-Used): Evict page that was least frequently requested
- LIFO (Last In First Out): Evict the last-in page in cache

All the above algorithms are not optimum!
Indeed all the algorithms are “online”, i.e, the decisions can be made without knowing future requests. Online algorithms can not be optimum.
FIFO is not optimum
FIFO is not optimum
FIFO is not optimum

<table>
<thead>
<tr>
<th>requests</th>
<th>FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
FIFO is not optimum

FIFO

requests

1
2
3
4

1

1

1

1

1

1

1

1
FIFO is not optimum

<table>
<thead>
<tr>
<th>requests</th>
<th>FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x 1</td>
</tr>
<tr>
<td>2</td>
<td>x 1</td>
</tr>
<tr>
<td>3</td>
<td>x 1</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
FIFO is not optimum

requests

1
2
3
4
1

FIFO

1
1
1
2

FIFO is not optimum

requests

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIFO

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
FIFO is not optimum

<table>
<thead>
<tr>
<th>requests</th>
<th>FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
FIFO is not optimum

requests

\[
\begin{array}{c|cccc}
\text{requests} & 1 & 2 & 3 & 4 \\
\hline
1 & \times & 1 & & \\
2 & \times & 1 & 2 & \\
3 & \times & 1 & 2 & 3 \\
4 & \times & 4 & 2 & 3 \\
1 & & & \\
\end{array}
\]
FIFO is not optimum

requests

1
2
3
4

FIFO

1
2
3
4
FIFO is not optimum

<table>
<thead>
<tr>
<th>requests</th>
<th>FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x 1</td>
</tr>
<tr>
<td>2</td>
<td>x 1 2</td>
</tr>
<tr>
<td>3</td>
<td>x 1 2 3</td>
</tr>
<tr>
<td>4</td>
<td>x 4 2 3</td>
</tr>
<tr>
<td>1</td>
<td>x 4 1 3</td>
</tr>
</tbody>
</table>
FIFO is not optimum

requests

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>

FIFO

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

misses = 5
FIFO is not optimum

<table>
<thead>
<tr>
<th>requests</th>
<th>FIFO</th>
<th>Furthest-in-Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[ ] [ ] [ ]</td>
<td>[ ] [ ] [ ]</td>
</tr>
<tr>
<td>2</td>
<td>[ ] [ ] [ ]</td>
<td>[ ] [ ] [ ]</td>
</tr>
<tr>
<td>3</td>
<td>[ ] [ ] [ ]</td>
<td>[ ] [ ] [ ]</td>
</tr>
<tr>
<td>4</td>
<td>[ ] [ ] [ ] [ ]</td>
<td>[ ] [ ] [ ] [ ]</td>
</tr>
<tr>
<td>1</td>
<td>[ ] [ ] [ ]</td>
<td>[ ] [ ] [ ] [ ]</td>
</tr>
</tbody>
</table>

misses = 5

misses = 4
Furthest-in-Future (FF)

- Algorithm: every time, evict the page that is not requested until furthest in the future, if we need to evict one.
- The algorithm is not an online algorithm, since the decision at a step depends on the request sequence in the future.