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 \(S\)

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: \(j \leftarrow \arg\min_{j' \in A} f_{j'}\)
4: \(S \leftarrow S \cup \{j\}; A \leftarrow \{j' \in A : s_{j'} \geq f_j\}\)
5: return \(S\)

Running time of algorithm?

- Naive implementation: \(O(n^2)\) time
- Clever implementation: \(O(n \lg n)\) time
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: \(t \leftarrow f_j\)
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: \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: \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: \(\text{if } s_j \geq t \text{ then}\)
5: \(S \leftarrow S \cup \{j\}\)
6: \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

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

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

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

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

Diagram:

- Jobs 1, 2, 3, 4, 5, 6, 7, 8, 9
- Schedule starts at time 0
- Jobs scheduled from left to right
- Time stamp \(t\) moves rightward with each job completion
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{10pt} if \(s_j \geq t\) then
5: \hspace{20pt} \(S \leftarrow S \cup \{j\}\)
6: \hspace{10pt} \(t \leftarrow f_j\)
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
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.
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 earliest-finished machine: There exists an optimum solution where job $j$ with the earliest starting time is scheduled first on the earliest-finished machine that is compatible with all jobs in that machine if applicable; otherwise, it can be scheduled by opening a new machine.

**Proof.**
Greedy Algorithm for Interval Partitioning

**Lemma**  It is safe to schedule the job $j$ with the earliest starting time to an earliest-finished machine: There exists an optimum solution where job $j$ with the earliest starting time is scheduled first on the earliest-finished 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 earliest-finished machine: There exists an optimum solution where job $j$ with the earliest starting time is scheduled first on the earliest-finished 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 earliest-finished machine $i$, done.
Greedy Algorithm for Interval Partitioning

**Lemma**  It is safe to schedule the job $j$ with the earliest starting time to a earliest-finished machine: There exists an optimum solution where job $j$ with the earliest starting time is scheduled first on the earliest-finished 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 earliest-finished machine $i$, done
**Lemma** It is safe to schedule the job $j$ with the earliest starting time to an earliest-finished machine: There exists an optimum solution where job $j$ with the earliest starting time is scheduled first on the earliest-finished 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 earliest-finished machine $i$, done
- Otherwise, replace all the jobs scheduled to the earliest-finished 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!
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: \textbf{while} $A \neq \emptyset$ \textbf{do}
3: \hspace{1em} $j \leftarrow \arg \min_{j' \in A} s_{j'}$, $S_j \leftarrow \{i'\}_{i' \in S, t_{i'} \leq s_j}$
4: \hspace{1em} \textbf{If} $S_j \neq \emptyset$, \textbf{then} schedule $j$ to machine $i \leftarrow \arg \min_{i' \in S_j} t_{i'}$
5: \hspace{1em} \hspace{1em} and $t_i = f_j$
6: \hspace{1em} \textbf{Otherwise}, schedule $j$ to machine $|S| + 1$, $S \leftarrow S \cup \{|S| + 1\}$
7: \hspace{1em} \hspace{1em} and $t_{|S|} = f_j$
8: \textbf{return} $S$
Greedy Algorithm for Interval Partitioning

The diagram illustrates intervals on a number line. Each interval is represented by a horizontal bar, and the intervals are partitioned at a certain point indicated by the index $j$. The intervals are ordered from left to right, with each interval starting at a specific index and ending at another index on the number line.
Greedy Algorithm for Interval Partitioning

![Diagram of Greedy Algorithm for Interval Partitioning]

- A line representing time from 0 to 9.
- Rectangles indicating intervals: one large grey rectangle at position 0, two medium grey rectangles at positions 3 and 5, and two small grey rectangles at positions 4 and 6.
- An orange rectangle at position 1, representing the interval for which we are optimizing.
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning

\[ \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} b_i \]

\[ j \]

\[ 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \]
Greedy Algorithm for Interval Partitioning

![Diagram of intervals]

- Interval at position 4 overlaps with intervals at positions 5, 6, 7, and 8.
- Interval at position 0 overlaps with intervals at positions 1 and 2.
- Interval at position 3 overlaps with intervals at positions 5 and 6.

The greedy algorithm selects intervals in a specific order to minimize overlap.

Interval at position $j$ is highlighted in orange.
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning

![Diagram of Greedy Algorithm for Interval Partitioning]
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning

![Diagram showing intervals and a decision point for interval j]
Greedy Algorithm for Interval Partitioning

![Diagram of intervals]

The diagram illustrates the greedy algorithm for interval partitioning. Each interval is represented on the horizontal axis, and the algorithm selects intervals in a greedy manner, ensuring that no intervals overlap. The selected intervals are highlighted in blue, and the diagram shows the optimal partitioning that minimizes overlap.
Greedy Algorithm for Interval Partitioning

The diagram illustrates the partitioning of intervals on a timeline. Each interval is represented as a colored bar, with different colors indicating different sets to which the intervals are assigned.

The variable $j$ is marked to indicate a specific interval or position on the timeline.
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning

![Diagram of Interval Partitioning](image)
**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.
**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.
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. \hspace{1em} $j \leftarrow \arg \min_{j' \in A} s_{j'}, S_j \leftarrow \{i'\}_{i' \in S, t_i \leq s_j}$
4. \hspace{1em} If $S_j \neq \emptyset$, then schedule $j$ to machine $i \leftarrow \arg \min_{i' \in S_j} t_{i'}$
5. \hspace{1em} and $t_i = f_j$
6. \hspace{1em} Otherwise, schedule $j$ to machine $|S| + 1$, $S \leftarrow S \cup \{|S| + 1\}$
7. \hspace{1em} and $t_{|S|} = f_j$
8. return $S$

Running time of algorithm?
Greedy Algorithm for Interval Partitioning

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: \text{If} \(S_j \neq \emptyset\), then schedule \(j\) to machine \(i \leftarrow \arg \min_{i' \in S_j} t_{i'}\) and \(t_i = f_j\)
5: \text{Otherwise, schedule} \(j\) to machine \(|S| + 1\), \(S \leftarrow S \cup \{|S| + 1\}\) 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: \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} \(S_j \neq \emptyset\), then schedule \(j\) to machine \(i \leftarrow \arg\min_{i' \in S_j} t_{i'}\) and \(t_i = f_j\)
5: \textbf{Otherwise}, schedule \(j\) to machine \(|S| + 1\), \(S \leftarrow S \cup \{|S| + 1\}\) and \(t_{|S|} = f_j\)
6: \textbf{return} \(S\)

Running time of algorithm?

- Naive implementation: \(O(n^2)\) time
- Clever implementation: \(O(n \lg n)\) 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
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

<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>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</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.
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 to bring the page into cache, and evict some existing page if necessary.

```
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.

<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.

```
page sequence  | cache
-------------|-----------
[2]           | x        |
[5]           |          |
[3]           |          |
[2]           |          |
[1]           |          |
```
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>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.

<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>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>x</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.
- 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</td>
</tr>
<tr>
<td>5</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>x</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 to 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>✗</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
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>✗ 1</td>
</tr>
<tr>
<td>4</td>
<td>✗ 1 5</td>
</tr>
<tr>
<td>2</td>
<td>✗ 1 5 4</td>
</tr>
<tr>
<td>5</td>
<td>✗ 1 2 4</td>
</tr>
<tr>
<td>3</td>
<td>✗ 1 2 5</td>
</tr>
<tr>
<td>2</td>
<td>✔ 1 2 3</td>
</tr>
<tr>
<td>1</td>
<td>✔ 1 2 3</td>
</tr>
</tbody>
</table>

misses = 6