Greedy Algorithm for Interval Scheduling

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

1: \(A \leftarrow \{1, 2, \ldots, n\}, S \leftarrow \emptyset\)
2: \textbf{while} \(A \neq \emptyset\) \textbf{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: \textbf{return} \(S\)

Running time of algorithm?

- Naive implementation: \(O(n^2)\) time
- Clever implementation: \(O(n \lg n)\) 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: \(\text{if } s_j \geq t \text{ then}\)
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

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

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

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

![Diagram](image-url)
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: \[\text{if } s_j \geq t \text{ then}\]
5: \(S \leftarrow S \cup \{j\}\)
6: \(t \leftarrow f_j\)
7: **return** \(S\)

![Diagram of Schedule algorithm with job durations and timestamps]
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

**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 $S'$. 

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!
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

Partition($s, f, n$)

1: $A \leftarrow \{1, 2, \ldots, n\}, S \leftarrow \{1\}, t_1 = 0$
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 a machine $i \in S_j$ and $t_i = f_j$
5: \hspace{1em} 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

![Diagram of Greedy Algorithm for Interval Partitioning]

The diagram illustrates a greedy algorithm for interval partitioning. Each interval is represented by a horizontal bar on the time axis.

- **0**: Start of the algorithm.
- **1**: First interval is selected.
- **2**: Second interval is selected.
- **3**: Third interval is selected.
- **4**: Fourth interval is selected.
- **5**: Fifth interval is selected.
- **6**: Sixth interval is selected.
- **7**: Seventh interval is selected.
- **8**: Eighth interval is selected.
- **9**: Ninth interval is selected.

The algorithm selects intervals in a greedy manner, aiming to minimize the number of intervals used while covering all intervals.
Greedy Algorithm for Interval Partitioning

![Diagram of Greedy Algorithm for Interval Partitioning](image)
Greedy Algorithm for Interval Partitioning

![Diagram showing intervals and the greedy algorithm selection strategy]

- Intervals are represented by horizontal bars.
- The greedy algorithm selects intervals in a specific order, indicated by the orange bar labeled with "j".
- The selection strategy aims to cover all intervals while minimizing overlaps.

This diagram illustrates how the greedy algorithm works in interval partitioning, emphasizing the step-by-step selection process.
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning

![Diagram showing interval partitioning](image)
Greedy Algorithm for Interval Partitioning

![Diagram showing intervals on a number line with a greedy algorithm for partitioning intervals.]
Greedy Algorithm for Interval Partitioning

![Diagram showing the greedy algorithm for interval partitioning. The intervals are represented on a timeline.]

The diagram illustrates the greedy algorithm for interval partitioning, where intervals are selected in a sequential manner to minimize overlap.
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning
Greedy Algorithm for Interval Partitioning

![Diagram of interval partitioning with intervals represented by bars at different positions on a timeline.]
Greedy Algorithm for Interval Partitioning

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

The diagram illustrates the process of partitioning intervals. Each interval is represented by a colored bar on the x-axis, which ranges from 0 to 9. The intervals are optimized for efficient partitioning, demonstrating how the algorithm selects intervals to maximize the partition's efficiency. The notation $j$ indicates a specific point of interest in the partitioning process.
Def. The **depth** of a set of jobs is the maximum number of overlapping jobs at any point within the given set.
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.
**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.
- 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. \( j \leftarrow \arg\min_{j'} s_{j'}, S_j \leftarrow \{i'\}_{i' \in S, t_{i'} \leq s_j} \)
   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?
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: \( j \leftarrow \arg \min_{j' \in A} s_{j'}, S_j \leftarrow \{i'\} \mid i' \in S, t_{i'} \leq s_j \)
4: \text{If} \( S_j \neq \emptyset \), \text{then schedule} \( j \) to a machine \( i \in S_j \) 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. 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: $O(n \lg n)$ 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
6. Exercise Problems
Offline Caching

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

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

Page sequence: 1 5 4 2 5 3 2 1
Cache:
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>1</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.
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>× 1</td>
</tr>
<tr>
<td>5</td>
<td>× 1 5</td>
</tr>
<tr>
<td>4</td>
<td>× 1 5 4</td>
</tr>
<tr>
<td>2</td>
<td>× 1 2 4</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</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>
```
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.

<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>

![Cache Diagram](cache_diagram.png)
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 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>1</td>
</tr>
<tr>
<td>4</td>
<td>1, 5</td>
</tr>
<tr>
<td>2</td>
<td>1, 5</td>
</tr>
<tr>
<td>2</td>
<td>1, 5</td>
</tr>
<tr>
<td>5</td>
<td>1, 2</td>
</tr>
<tr>
<td>3</td>
<td>1, 2</td>
</tr>
<tr>
<td>2</td>
<td>1, 2</td>
</tr>
<tr>
<td>1</td>
<td>1, 2</td>
</tr>
<tr>
<td>1</td>
<td>1, 2</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.

<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>✗</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>Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✗ 1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>✗ 1 5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>✗ 1 5 4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>✗ 1 2 4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>✗ 1 2 5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>✗ 1 2 3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>✓ 1 2 3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>✓ 1 2 3</td>
<td></td>
</tr>
</tbody>
</table>

misses = 6

<table>
<thead>
<tr>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗ 1</td>
</tr>
<tr>
<td>✗ 1</td>
</tr>
<tr>
<td>✓ 1</td>
</tr>
<tr>
<td>✗ 1</td>
</tr>
<tr>
<td>✓ 1</td>
</tr>
</tbody>
</table>

misses = 5
<table>
<thead>
<tr>
<th><strong>Offline Caching Problem</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> ( k ) : the size of cache</td>
</tr>
<tr>
<td>( n ) : number of pages</td>
</tr>
<tr>
<td>( \rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n] ): sequence of requests</td>
</tr>
<tr>
<td><strong>Output:</strong> ( i_1, i_2, i_3, \cdots, i_T \in { \text{hit}, \text{empty} } \cup [n] ): indices of pages to evict (&quot;hit&quot; means evicting no page, &quot;empty&quot; means evicting empty page)</td>
</tr>
</tbody>
</table>

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

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

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

$\rho_1, \rho_2, \rho_3, \cdots , \rho_T \in [n]$: 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.
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

**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\}.

- **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

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.
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.

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
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

requests

1
2
3
4
1

FIFO


FIFO is not optimum

requests

1
2
3
4
1

FIFO

[Diagram showing requests in order, with a mark indicating an incorrect order]
FIFO is not optimum

FIFO

requests

1
2
3
4
1

× 1
FIFO is not optimum

requests

1 2 3 4

FIFO

1 1 1
FIFO is not optimum

requests

1
2
3
4

FIFO

1
2
1
2
1
FIFO is not optimum

requests

1
2
3
4
1

FIFO

[Diagram showing FIFO requests with x marks for non-optimum sequence]
FIFO is not optimum

requests

1
2
3
4
1

FIFO

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

requests

1
2
3
4
1
2
3
4

FIFO

<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>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
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>✗</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

<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></td>
<td>4</td>
<td>1</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>XX 1 2 3</td>
<td>XX 1 4 3</td>
</tr>
<tr>
<td>2</td>
<td>XX 1 2 3</td>
<td>XX 1 4 3</td>
</tr>
<tr>
<td>3</td>
<td>XX 1 2 3</td>
<td>XX 1 4 3</td>
</tr>
<tr>
<td>4</td>
<td>XX 4 2 3</td>
<td>XX 1 4 3</td>
</tr>
<tr>
<td>1</td>
<td>XX 4 1 3</td>
<td>✓ 1 4 3</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.
## Furthest-in-Future (FF)

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

misses = 5

misses = 4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

×  ×  ×

1  1  1

5  5

4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

×  ×  ×

1  1  1

5  5

4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

1  1  1  1  2

5  5  5

4  4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

×  ×  ×  ×  ×

☐  1  1  1  2

☐  ☐  5  5  5

☐  ☐  ☐  4  4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

\[\times \times \times \times \checkmark\]

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 2 & 2 & \\
5 & 5 & 5 & 5 & \\
4 & 4 & 4 & \\
\end{array}
\]
Example

requests

1 5 4 2 5 3 2 4 3 1 5 3

X X X X √

1 1 1 2 2

5 5 5 5

4 4 4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

✔️ ✗ ✗ ✗ ✗ ✔️ ✗
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

\[\times\times\times\times\checkmark\times\]

1  1  1  2  2  2

5  5  5  5  5  3

4  4  4  4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

- - - - ✓ - - ✓ ✓

  1  1  1  2  2  2  2

  5  5  5  5  3  3

  4  4  4  4  4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

[Red X] [Red X] [Red X] [Red X] [Green √] [Red X] [Green √] [Green √]

[ ]  1  1  1  2  2  2  2  2

[ ]  [ ]  5  5  5  5  3  3  3

[ ]  [ ]  [ ]  4  4  4  4  4  4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

X  X  X  X  ✓  X  ✓  ✓  ✓  ✓

1  1  1  2  2  2  2  2  2

5  5  5  5  3  3  3  3

4  4  4  4  4  4  4  4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

X X X X √ X X √ √ √

☐  ☐  1  1  1  2  2  2  2  2  2

☐  ☐  5  5  5  5  3  3  3  3

☐  ☐  ☐  4  4  4  4  4  4  4
Example

requests

\[\begin{array}{cccccccccccc}
1 & 5 & 4 & 2 & 5 & 3 & 2 & 4 & 3 & 1 & 5 & 3 \\
\text{X} & \text{X} & \text{X} & \text{X} & \checkmark & \text{X} & \checkmark & \checkmark & \checkmark & \text{X} \\
\hline
\text{□} & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 1 \\
\text{□} & \text{□} & 5 & 5 & 5 & 5 & 3 & 3 & 3 & 3 & 3 \\
\text{□} & \text{□} & \text{□} & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
\end{array}\]
Example

requests

- 1
- 5
- 4
- 2
- 5
- 3
- 2
- 4
- 3
- 1
- 5
- 3

- x
- x
- x
- x
- x
- ✓
- x
- ✓
- ✓
- ✓
- x
- x

- □
- 1
- 1
- 1
- 2
- 2
- 2
- 2
- 2
- 2
- 1
- 5
- □
- □
- 5
- 5
- 5
- 5
- 3
- 3
- 3
- 3
- 3
- 3
- □
- □
- □
- 4
- 4
- 4
- 4
- 4
- 4
- 4
- 4
- 4
- 4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

x  x  x  x  x  v  x  v  v  v  x  x  v

1  1  1  2  2  2  2  2  2  1  5  5

5  5  5  5  3  3  3  3  3  3  3  3

4  4  4  4  4  4  4  4  4  4  4  4
Recall: Designing and Analyzing Greedy Algorithms

Greedy Algorithm
- Build up the solutions in steps
- At each step, make an irrevocable decision using a “reasonable” strategy

Analysis of Greedy Algorithm
- Safety: Prove that the reasonable strategy is “safe” (key)
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)
Recall: Designing and Analyzing Greedy Algorithms

**Greedy Algorithm**
- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

**Analysis of Greedy Algorithm**
- Safety: Prove that the reasonable strategy is “safe” (key)
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)
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:** $i_1, i_2, i_3, \cdots, i_t \in \{\text{hit, empty}\} \cup [n]$  
- empty stands for an empty page  
- “hit” means evicting no pages
# 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
- $p_1, p_2, \ldots, p_k \in \{\text{empty}\} \cup [n]$: initial set of pages in cache

**Output:**
- $i_1, i_2, i_3, \ldots, i_t \in \{\text{hit}, \text{empty}\} \cup [n]$
  - empty stands for an empty page
  - “hit” means evicting no pages