Greedy Algorithm for Interval Scheduling

Schedule(s, f, n)

```
A ← {1, 2, ⋯ , n}, S ← ∅
while A ≠ ∅ do
    j ← arg min_{j'∈A} f_{j'}
    S ← S ∪ {j}; A ← {j' ∈ A : s_{j'} ≥ f_j}
return S
```

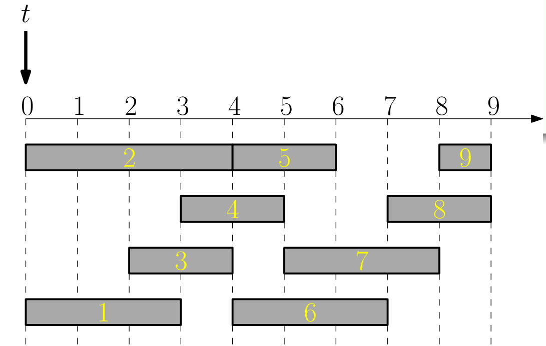
Running time of algorithm?

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

Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

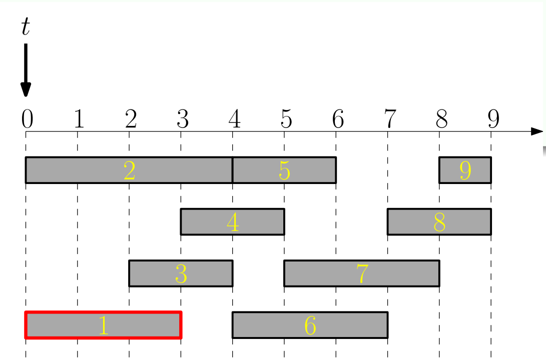
sort jobs according to f values
$t \leftarrow 0$, $S \leftarrow \emptyset$
for every $j \in [n]$ according to non-decreasing order of f_j **do**
    if $s_j \geq t$ **then**
        $S \leftarrow S \cup \{j\}$
        $t \leftarrow f_j$
return S



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

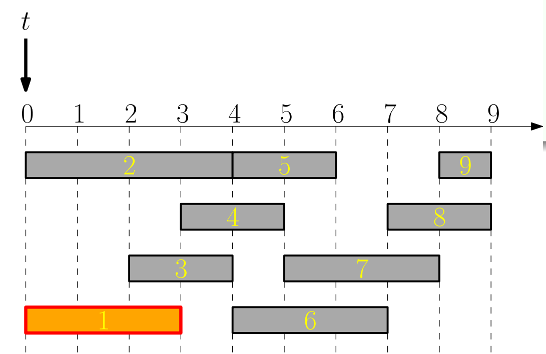
```
sort jobs according to f values
t ← 0, S ← ∅
for every j ∈ [n] according to non-decreasing order of f_j do
    if s_j ≥ t then
        S ← S ∪ {j}
        t ← f_j
return S
```



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

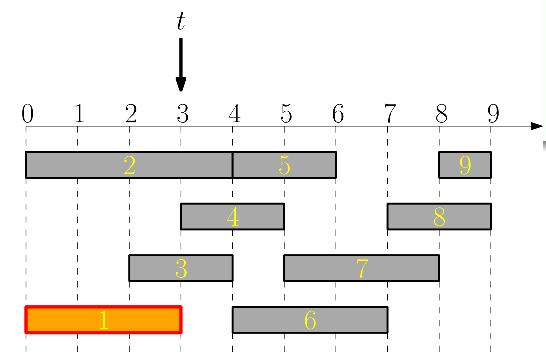
```
sort jobs according to f values
t ← 0, S ← ∅
for every j ∈ [n] according to non-decreasing order of f_j do
    if s_j ≥ t then
        S ← S ∪ {j}
        t ← f_j
return S
```



Clever Implementation of Greedy Algorithm

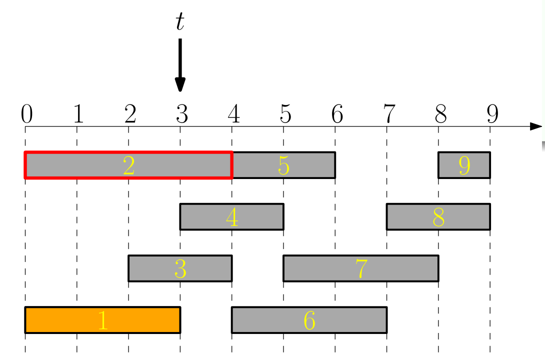
Schedule(s, f, n)

```
sort jobs according to f values
t ← 0, S ← ∅
for every j ∈ [n] according to non-decreasing order of f_j do
    if s_j ≥ t then
        S ← S ∪ {j}
        t ← f_j
return S
```



Schedule(s, f, n)

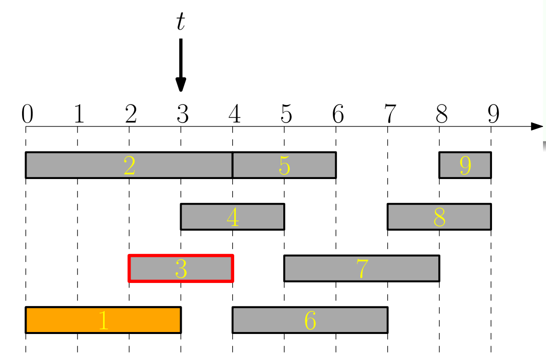
1: sort jobs according to f values
2: $t \leftarrow 0$, $S \leftarrow \emptyset$
3: **for** every $j \in [n]$ according to non-decreasing order of f_j **do**
4: **if** $s_j \geq t$ **then**
5: $S \leftarrow S \cup \{j\}$
6: $t \leftarrow f_j$
7: **return** S



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

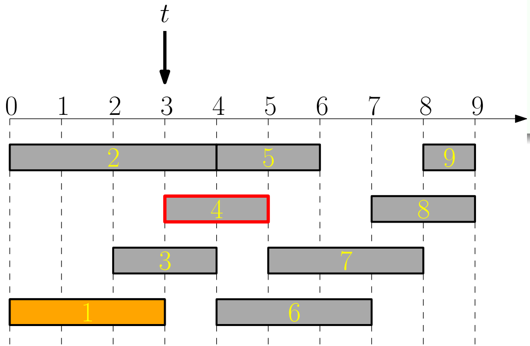
sort jobs according to f values
$t \leftarrow 0$, $S \leftarrow \emptyset$
for every $j \in [n]$ according to non-decreasing order of f_j **do**
  if $s_j \geq t$ **then**
    $S \leftarrow S \cup \{j\}$
    $t \leftarrow f_j$
return S



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

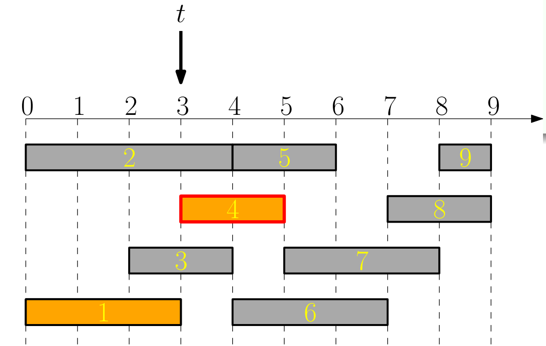
sort jobs according to f values
$t \leftarrow 0$, $S \leftarrow \emptyset$
for every $j \in [n]$ according to non-decreasing order of f_j **do**
    if $s_j \geq t$ **then**
        $S \leftarrow S \cup \{j\}$
        $t \leftarrow f_j$
return S



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

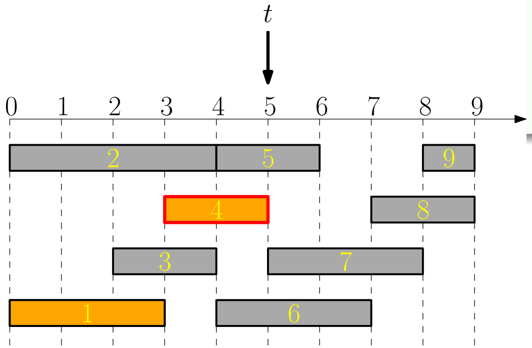
```
sort jobs according to f values
t ← 0, S ← ∅
for every j ∈ [n] according to non-decreasing order of f_j do
    if s_j ≥ t then
        S ← S ∪ {j}
        t ← f_j
return S
```



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

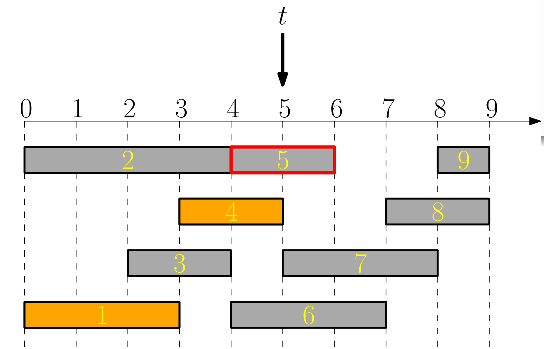
1: sort jobs according to f values
2: $t \leftarrow 0$, $S \leftarrow \emptyset$
3: **for** every $j \in [n]$ according to non-decreasing order of f_j **do**
4:     **if** $s_j \geq t$ **then**
5:         $S \leftarrow S \cup \{j\}$
6:         $t \leftarrow f_j$
7: **return** S



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

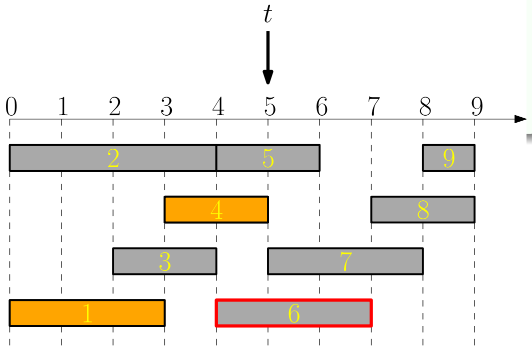
sort jobs according to f values
$t \leftarrow 0$, $S \leftarrow \emptyset$
for every $j \in [n]$ according to non-decreasing order of f_j **do**
  if $s_j \geq t$ **then**
    $S \leftarrow S \cup \{j\}$
    $t \leftarrow f_j$
return S



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

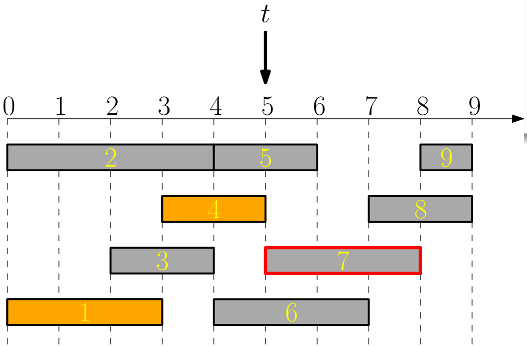
sort jobs according to f values
$t \leftarrow 0$, $S \leftarrow \emptyset$
for every $j \in [n]$ according to non-decreasing order of f_j **do**
    if $s_j \geq t$ **then**
        $S \leftarrow S \cup \{j\}$
        $t \leftarrow f_j$
return S



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

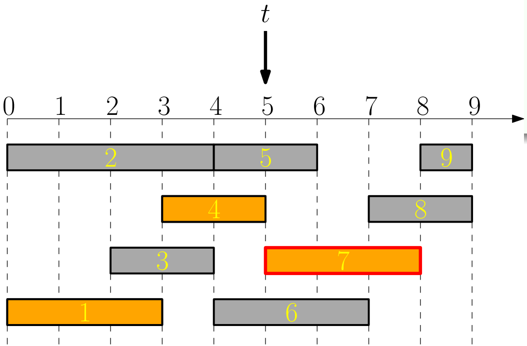
1: sort jobs according to f values
2: $t \leftarrow 0$, $S \leftarrow \emptyset$
3: **for** every $j \in [n]$ according to non-decreasing order of f_j **do**
4: **if** $s_j \geq t$ **then**
5: $S \leftarrow S \cup \{j\}$
6: $t \leftarrow f_j$
7: **return** S



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

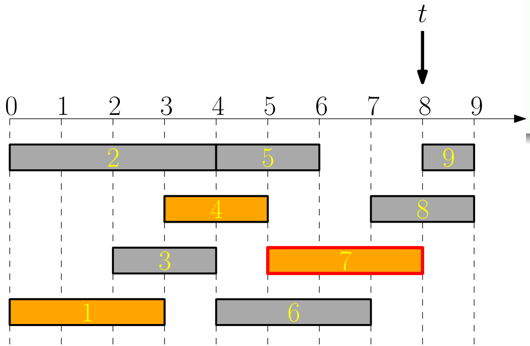
sort jobs according to f values
$t \leftarrow 0$, $S \leftarrow \emptyset$
for every $j \in [n]$ according to non-decreasing order of f_j **do**
    if $s_j \geq t$ **then**
        $S \leftarrow S \cup \{j\}$
        $t \leftarrow f_j$
return S



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

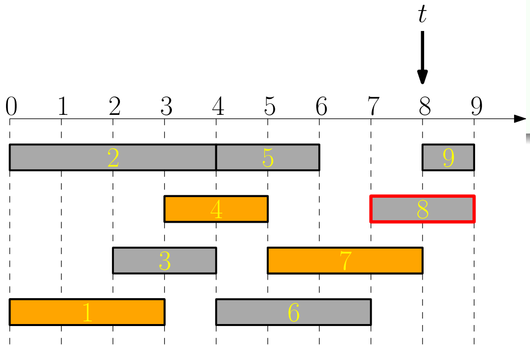
```
sort jobs according to f values
t ← 0, S ← ∅
for every j ∈ [n] according to non-decreasing order of f_j do
    if s_j ≥ t then
        S ← S ∪ {j}
        t ← f_j
return S
```



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

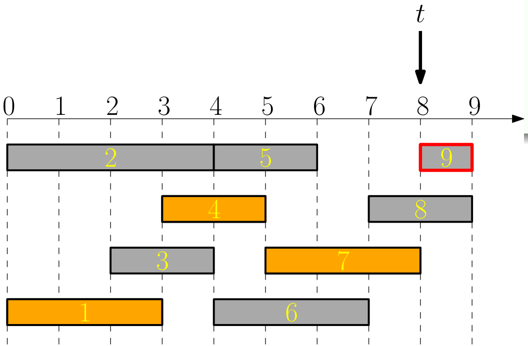
sort jobs according to f values
$t \leftarrow 0$, $S \leftarrow \emptyset$
for every $j \in [n]$ according to non-decreasing order of f_j **do**
    if $s_j \geq t$ **then**
        $S \leftarrow S \cup \{j\}$
        $t \leftarrow f_j$
return S



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

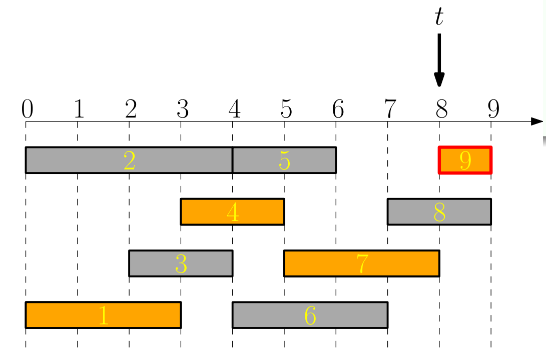
```
sort jobs according to f values
t ← 0, S ← ∅
for every j ∈ [n] according to non-decreasing order of f_j do
    if s_j ≥ t then
        S ← S ∪ {j}
        t ← f_j
return S
```



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

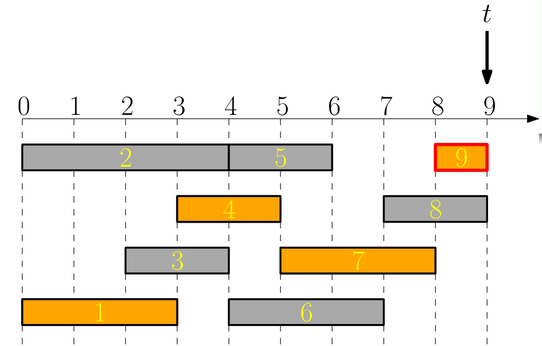
```
sort jobs according to f values
t ← 0, S ← ∅
for every j ∈ [n] according to non-decreasing order of f_j do
    if s_j ≥ t then
        S ← S ∪ {j}
        t ← f_j
return S
```



Clever Implementation of Greedy Algorithm

Schedule(s, f, n)

```
sort jobs according to f values
t ← 0, S ← ∅
for every j ∈ [n] according to non-decreasing order of f_j do
    if s_j ≥ t then
        S ← S ∪ {j}
        t ← f_j
return S
```



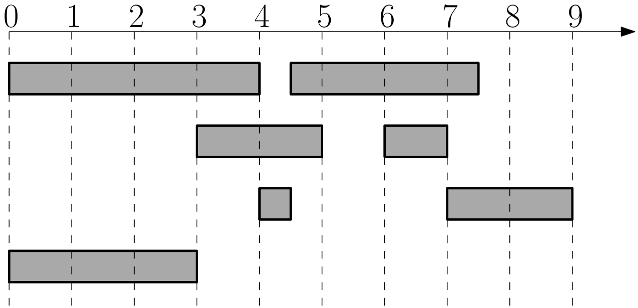
Outline

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.

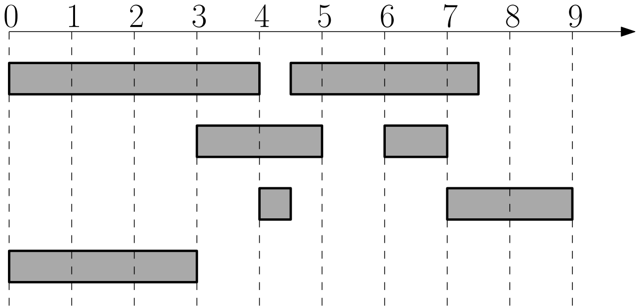


Interval Partitioning

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.

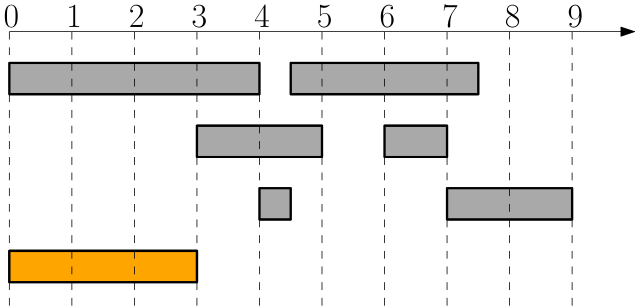


Interval Partitioning

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.

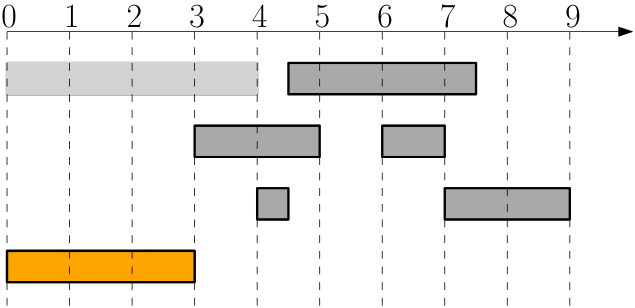


Interval Partitioning

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.

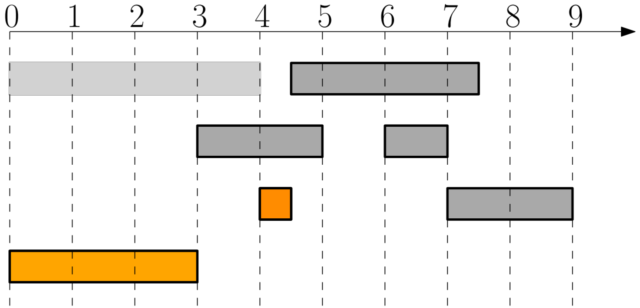


Interval Partitioning

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.

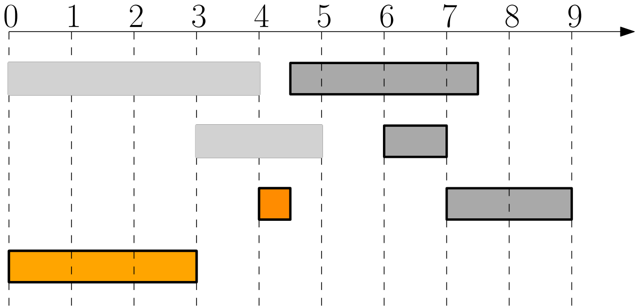


Interval Partitioning

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.

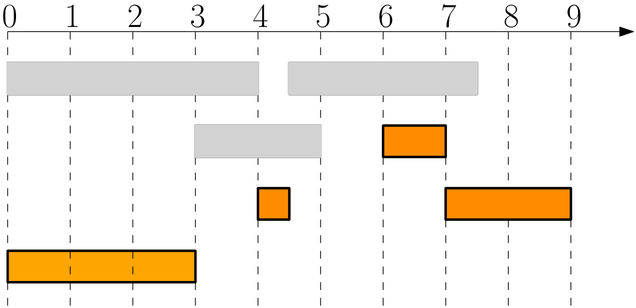


Interval Partitioning

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.

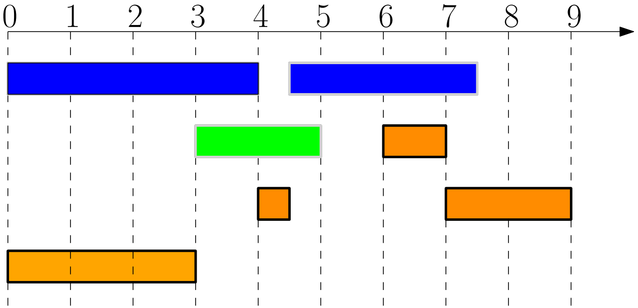


Interval Partitioning

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.

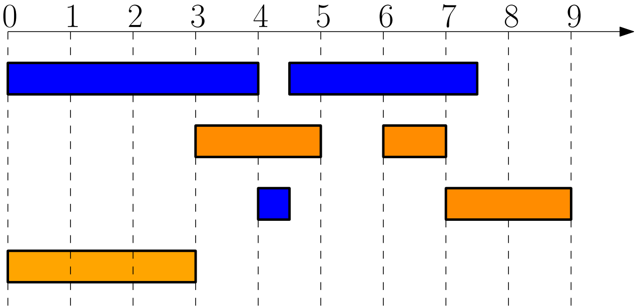


Interval Partitioning

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.



Greedy Algorithm for Interval Partitioning

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.

Greedy Algorithm for Interval Partitioning

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

Greedy Algorithm for Interval Partitioning

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

Greedy Algorithm for Interval Partitioning

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

Greedy Algorithm for Interval Partitioning

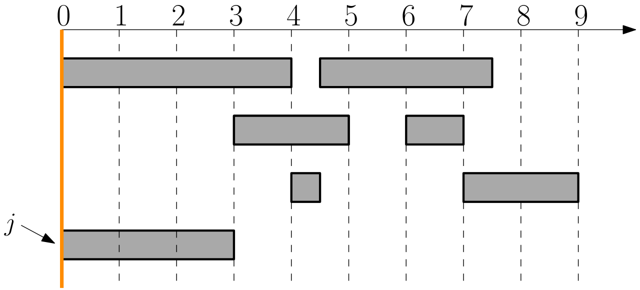
Lemma It is safe to schedule the job j with the earliest starting time to a feasible machine: There exists an optimum solution where job j with the earliest starting time is scheduled first on a machine that is compatible with all jobs in that machine if applicable; otherwise, it can be scheduled by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

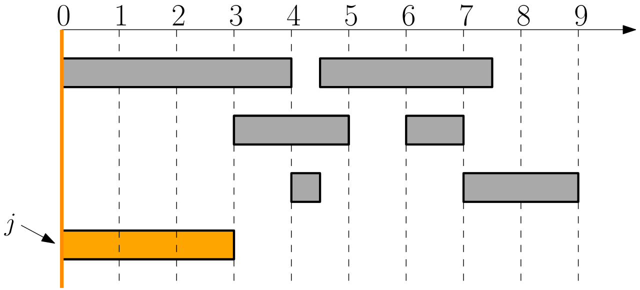
Greedy Algorithm for Interval Partitioning

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



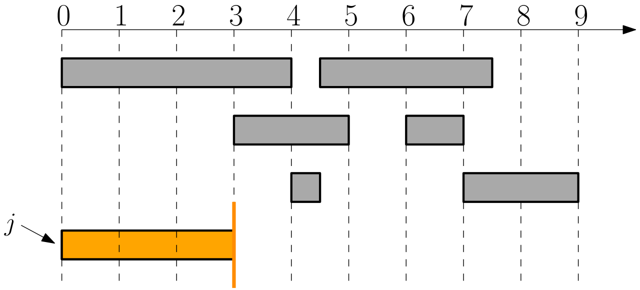
Greedy Algorithm for Interval Partitioning

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



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

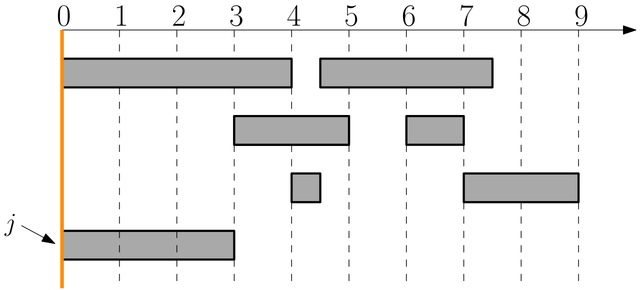
Partition(s, f, n)

```
A ← {1, 2, ⋯, n}, S ← {1}, t_1 = 0
while A ≠ ∅ do
    j ← argmin_{j'∈A} s_{j'}, S_j ← {i'}_{i'∈S, t_{i'} ≤ s_j}
    If S_j ≠ ∅, then schedule j to a machine i ∈ S_j and t_i = f_j
    Otherwise, schedule j to machine |S| + 1, S ← S ∪ {|S| + 1} and t_{|S|} = f_j
return S
```

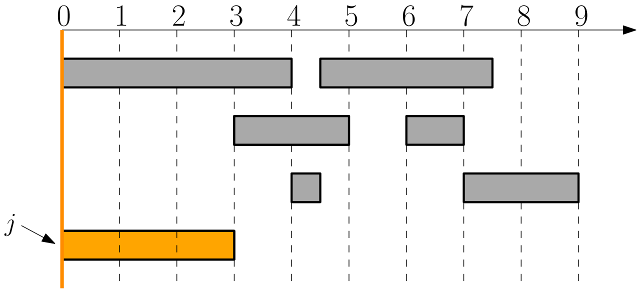
In math notation:

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

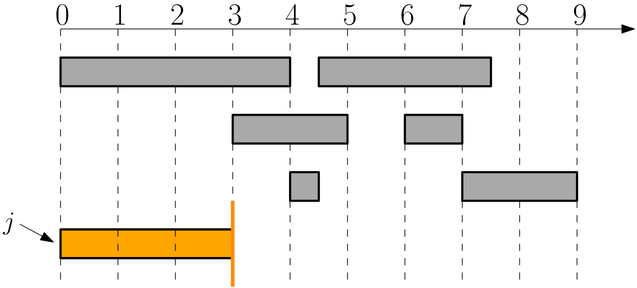
Greedy Algorithm for Interval Partitioning



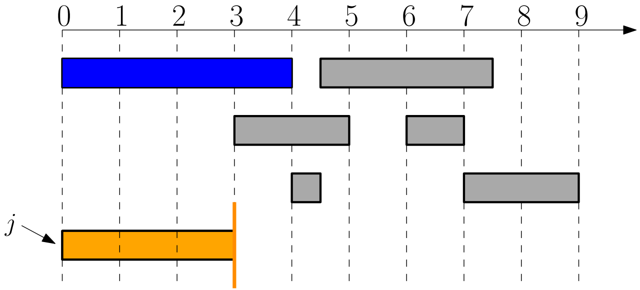
Greedy Algorithm for Interval Partitioning



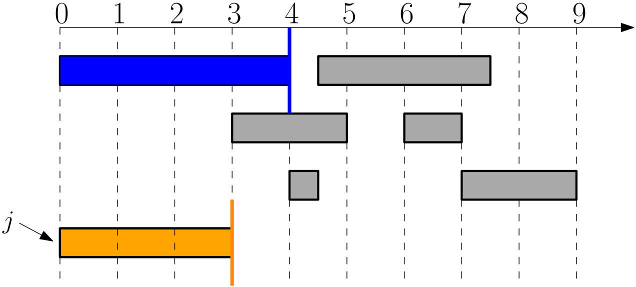
Greedy Algorithm for Interval Partitioning



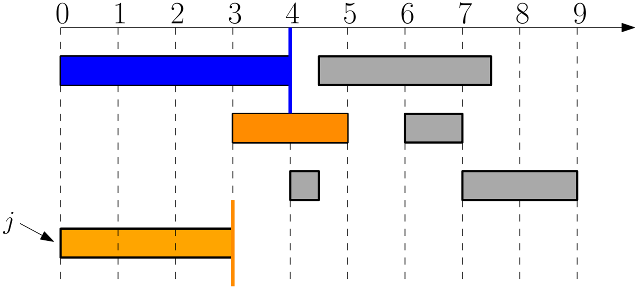
Greedy Algorithm for Interval Partitioning



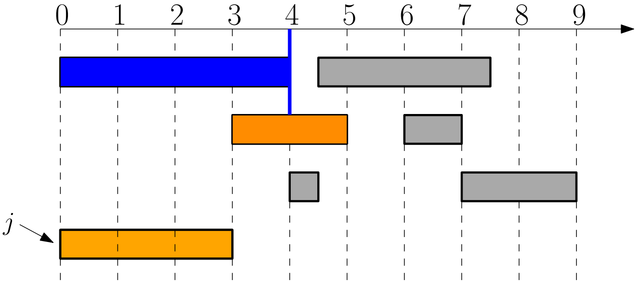
Greedy Algorithm for Interval Partitioning



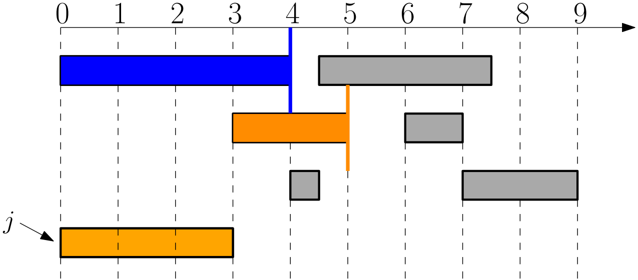
Greedy Algorithm for Interval Partitioning



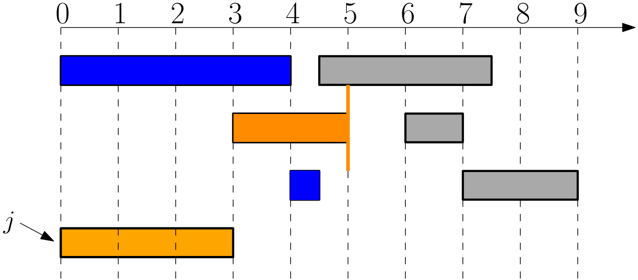
Greedy Algorithm for Interval Partitioning



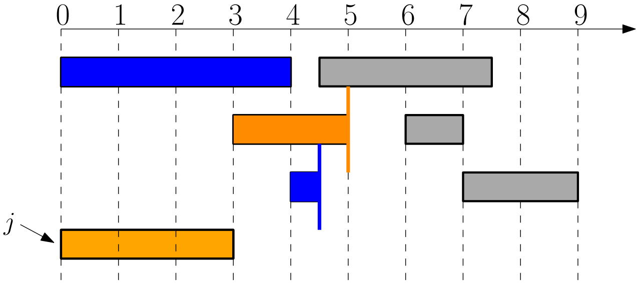
Greedy Algorithm for Interval Partitioning



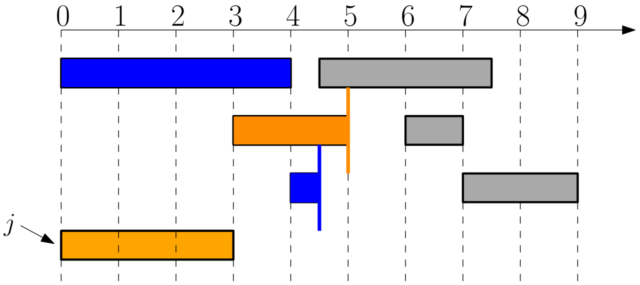
Greedy Algorithm for Interval Partitioning



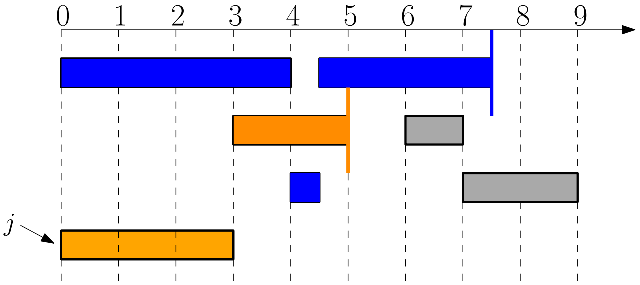
Greedy Algorithm for Interval Partitioning



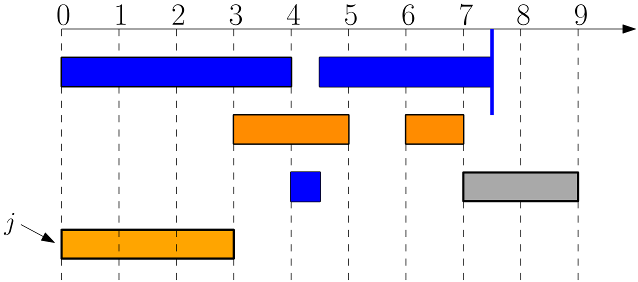
Greedy Algorithm for Interval Partitioning



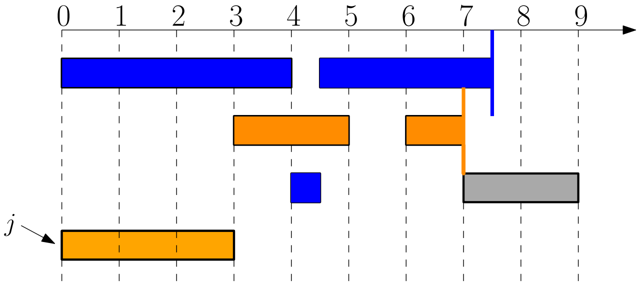
Greedy Algorithm for Interval Partitioning



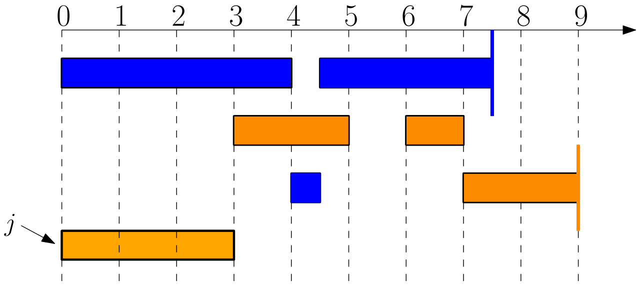
Greedy Algorithm for Interval Partitioning



Greedy Algorithm for Interval Partitioning



Greedy Algorithm for Interval Partitioning



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.

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.

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.

□

Greedy Algorithm for Interval Partitioning

Partition(s, f, n)

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

Running time of algorithm?

Greedy Algorithm for Interval Partitioning

Partition(s, f, n)

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

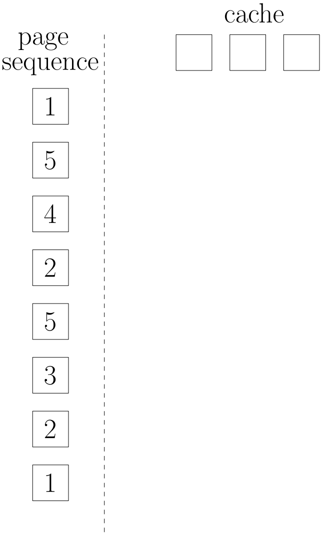
Outline

Offline Caching

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

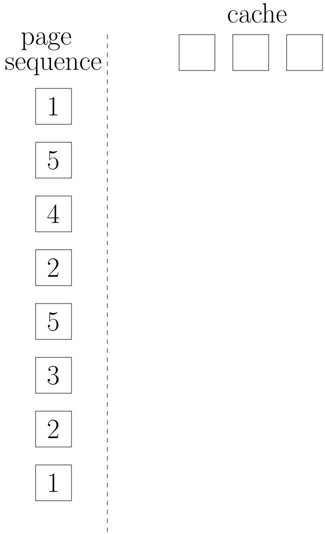
Offline Caching

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



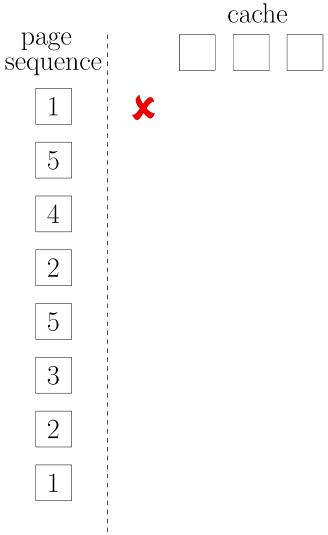
Offline Caching

- 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.



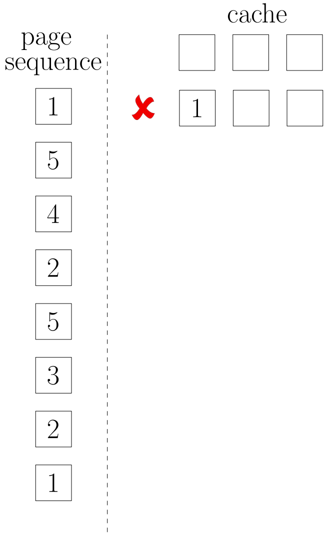
Offline Caching

- 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.



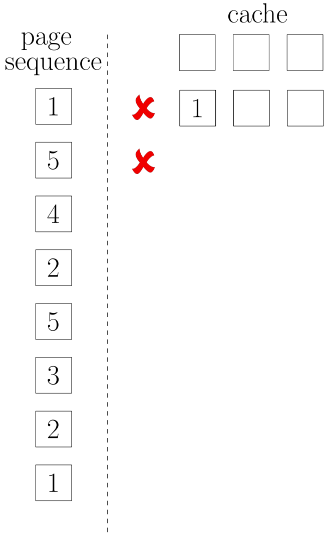
Offline Caching

- 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.



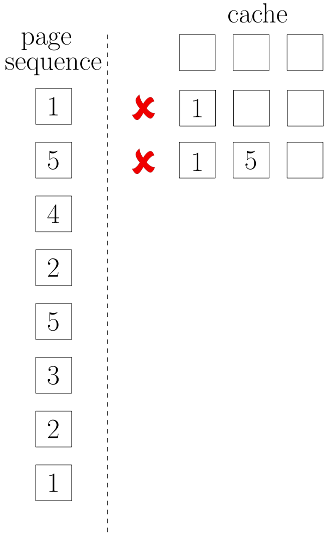
Offline Caching

- 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.



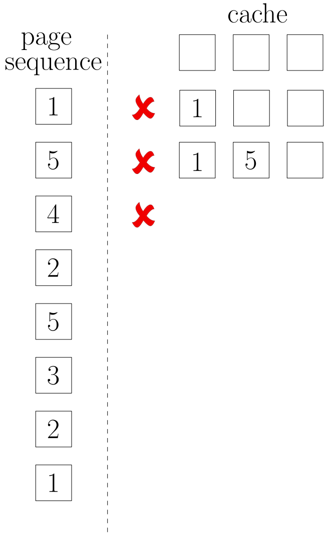
Offline Caching

- 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.



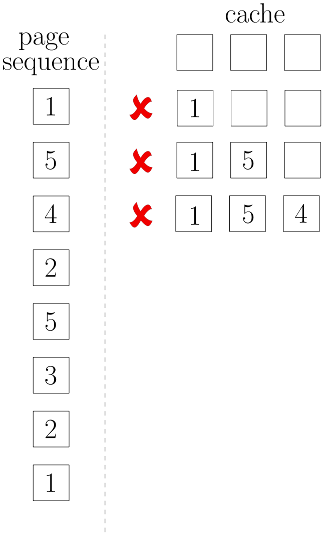
Offline Caching

- 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.



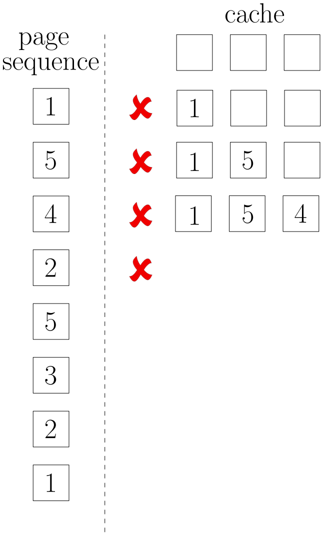
Offline Caching

- 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.



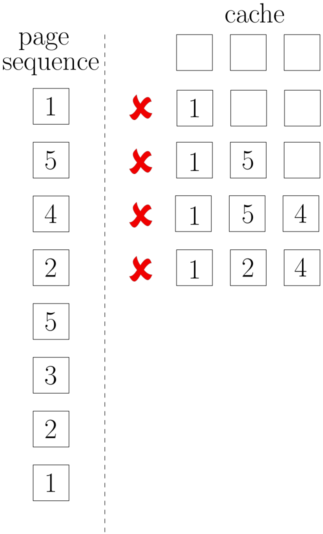
Offline Caching

- 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.



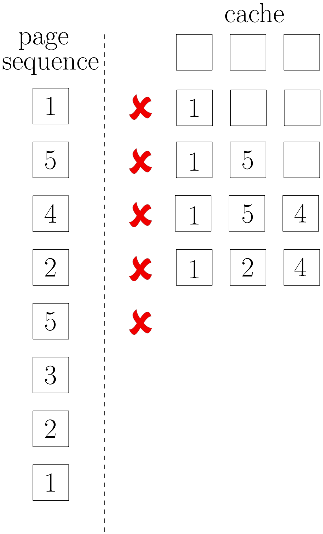
Offline Caching

- 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.



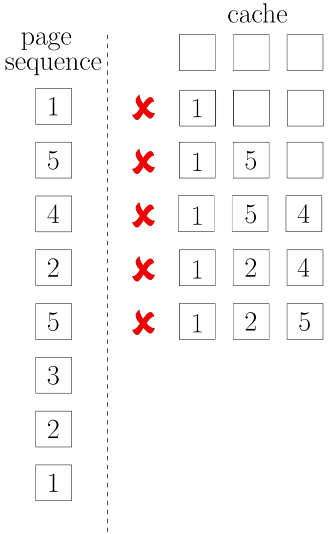
Offline Caching

- 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.



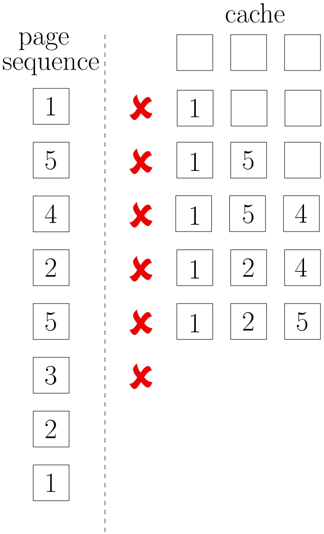
Offline Caching

- 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.



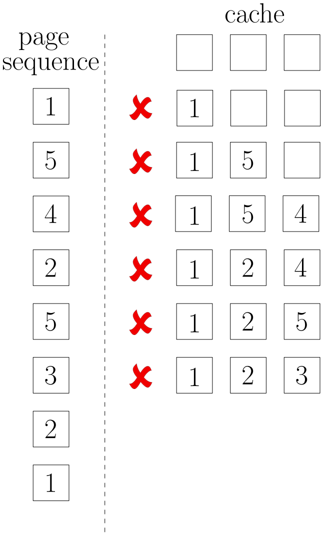
Offline Caching

- 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.



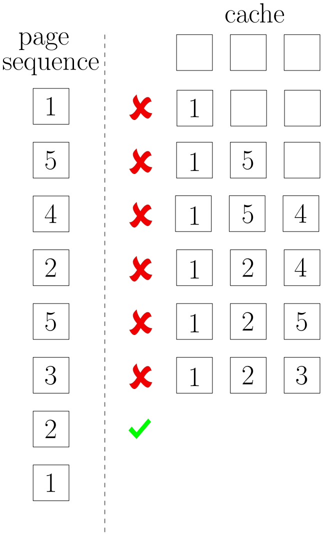
Offline Caching

- 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.



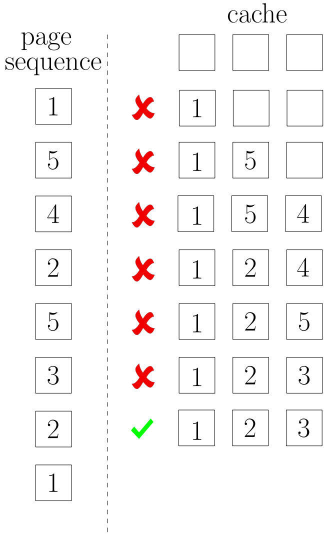
Offline Caching

- 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.
- Cache hit happens if requested page already in cache.



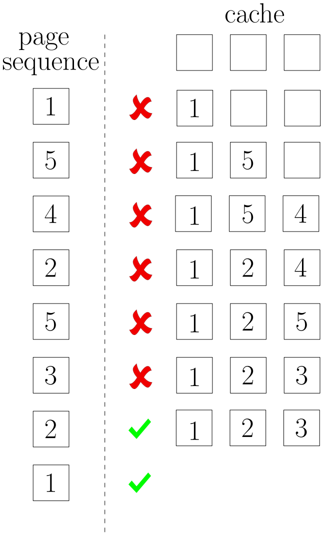
Offline Caching

- 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.



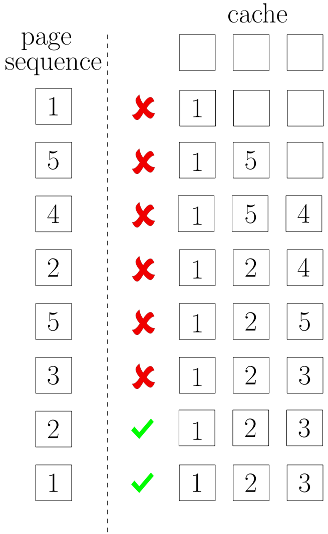
Offline Caching

- 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.



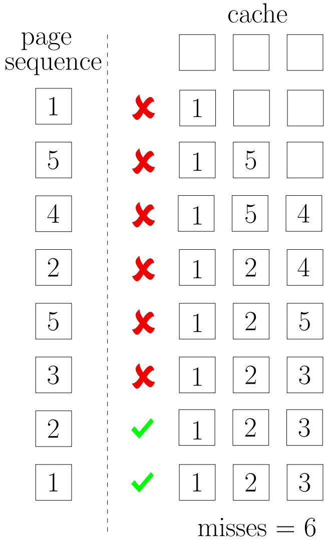
Offline Caching

- 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.



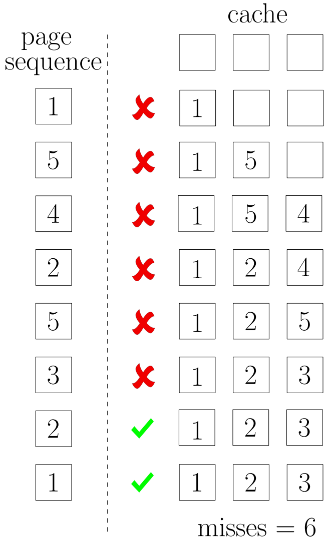
Offline Caching

- 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.
- Goal: minimize the number of cache misses.



A Better Solution for Example

page sequence		cache				cache		
1	✗	1			✗	1		
5	✗	1	5		✗	1	5	
4	✗	1	5	4	✗	1	5	4
2	✗	1	2	4	✗	1	5	2
5	✗	1	2	5	✓	1	5	2
3	✗	1	2	3	✗	1	3	2
2	✓	1	2	3	✓	1	3	2
1	✓	1	2	3	✓	1	3	2
		misses = 6				misses = 5		

Offline Caching Problem

Input: k : the size of cache

n : number of pages

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

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

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

Offline Caching Problem

Input: k : the size of cache

n : number of pages

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

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

Output: $i_1, i_2, i_3, \cdots, i_T \in \{\text{hit}, \text{empty}\} \cup [n]$: 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.

Offline Caching Problem

Input: k : the size of cache

n : number of pages

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

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

Output: $i_1, i_2, i_3, \cdots, i_T \in \{\text{hit}, \text{empty}\} \cup [n]$: 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.

Q: Which one is more realistic?

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

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

Output: $i_1, i_2, i_3, \cdots, i_T \in \{\mathrm{hit}, \mathrm{empty}\} \cup [n]$: 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

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

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

Offline Caching: Potential Greedy Algorithms

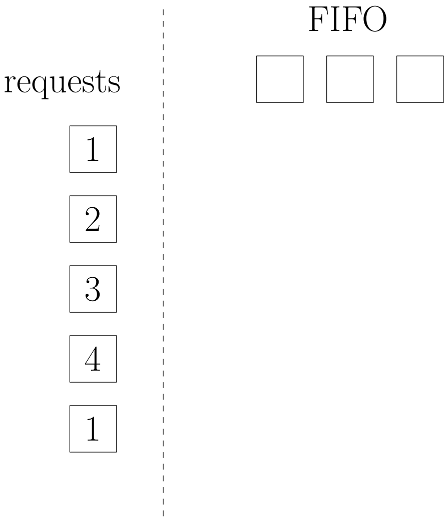
- FIFO(First-In-First-Out): Evict the first-in page in cache
- LRU(Least-Recently-Used): Evict page whose most recent access was earliest
- LFU(Least-Frequently-Used): Evict page that was least frequently requested
- LIFO (Last In First Out): Evict the last-in page in cache

Offline Caching: Potential Greedy Algorithms

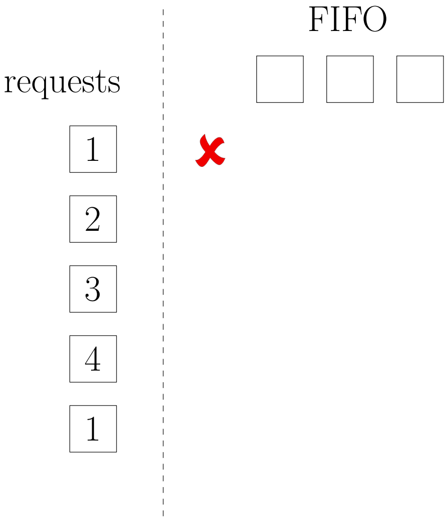
- FIFO(First-In-First-Out): Evict the first-in page in cache
- LRU(Least-Recently-Used): Evict page whose most recent access was earliest
- LFU(Least-Frequently-Used): Evict page that was least frequently requested
- LIFO (Last In First Out): Evict the last-in page in cache

- 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.

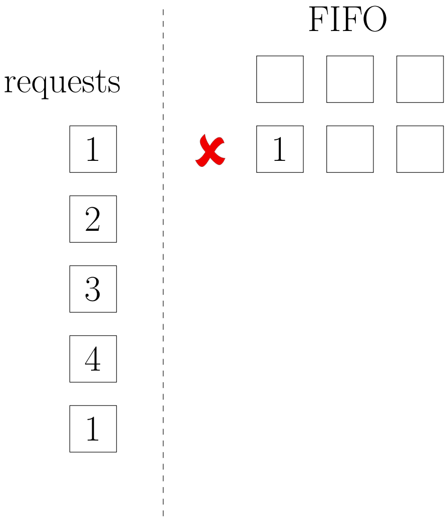
FIFO is not optimum



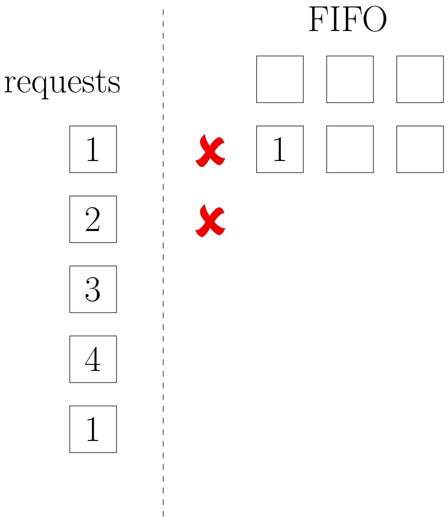
FIFO is not optimum



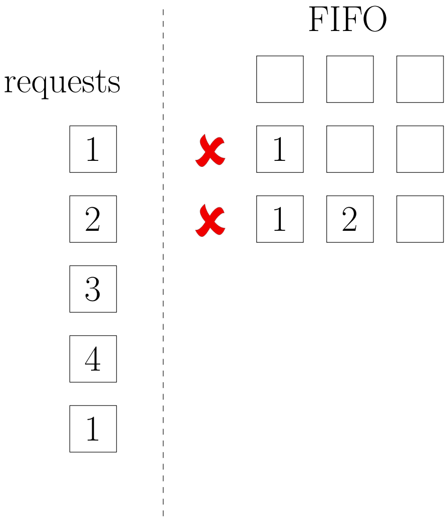
FIFO is not optimum



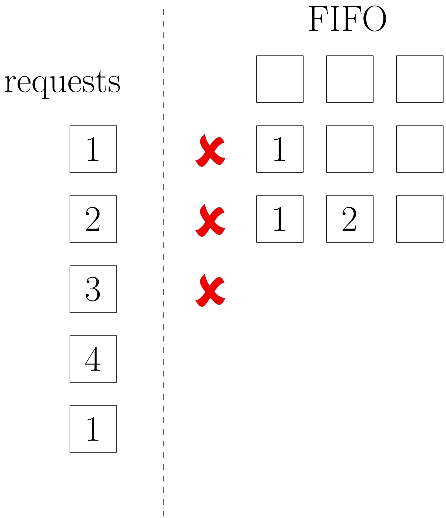
FIFO is not optimum



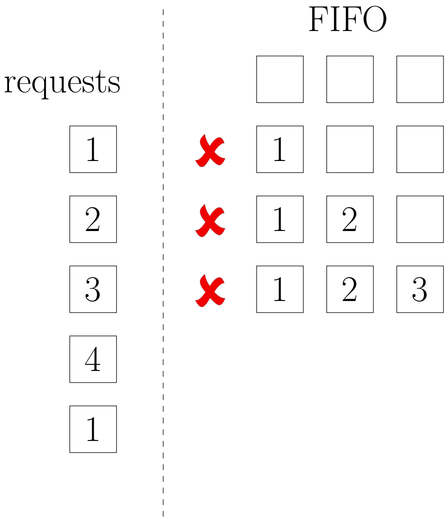
FIFO is not optimum



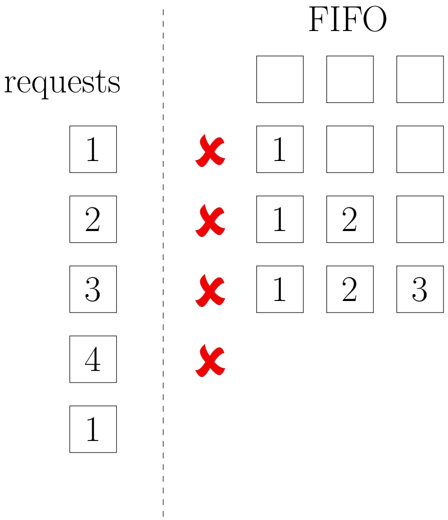
FIFO is not optimum



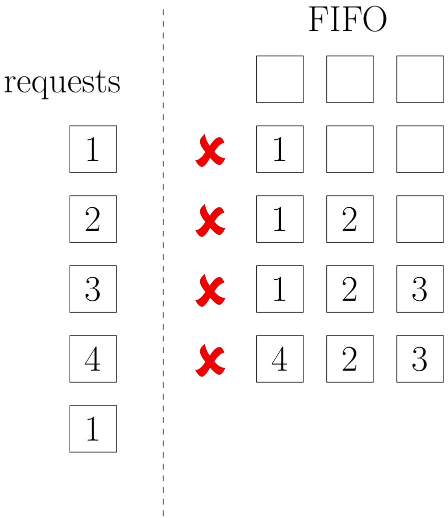
FIFO is not optimum



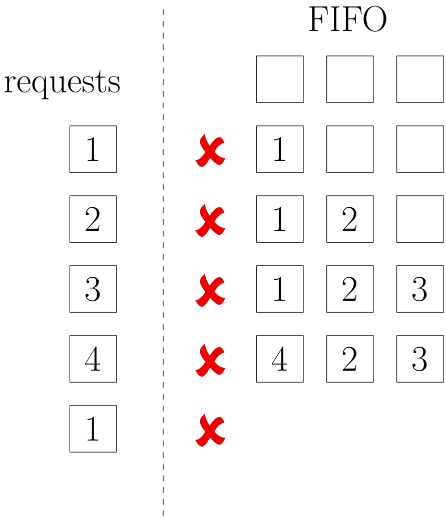
FIFO is not optimum



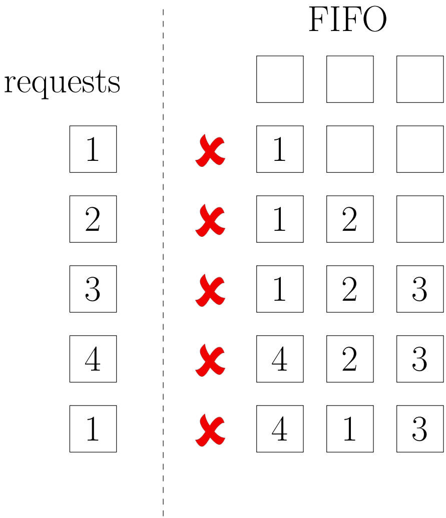
FIFO is not optimum



FIFO is not optimum



FIFO is not optimum



FIFO is not optimum

requests		FIFO		
1	✘	1		
2	✘	1	2	
3	✘	1	2	3
4	✘	4	2	3
1	✘	4	1	3

$\text{misses} = 5$

FIFO is not optimum

requests		FIFO				Furthest-in-Future		
1	✗	1			✗	1		
2	✗	1	2		✗	1	2	
3	✗	1	2	3	✗	1	2	3
4	✗	4	2	3	✗	1	4	3
1	✗	4	1	3	✓	1	4	3

FIFO: misses $= 5$

Furthest-in-Future: misses $= 4$

Optimum Offline Caching

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)

requests		FIFO				Furthest-in-Future		
1	✗	1			✗	1		
2	✗	1	2		✗	1	2	
3	✗	1	2	3	✗	1	2	3
4	✗	4	2	3	✗	1	4	3
1	✗	4	1	3	✓	1	4	3

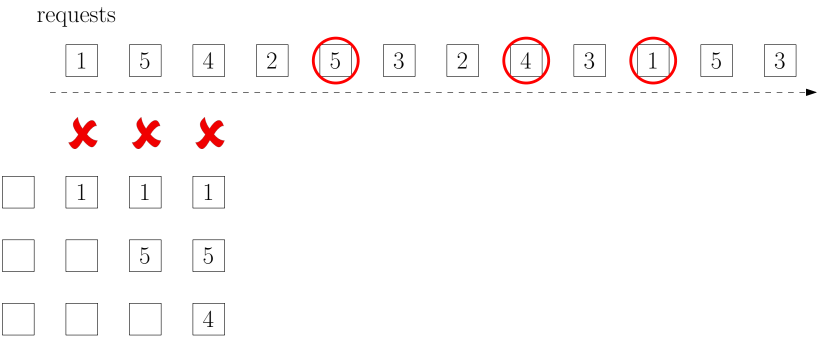
FIFO: misses = 5

Furthest-in-Future: misses = 4

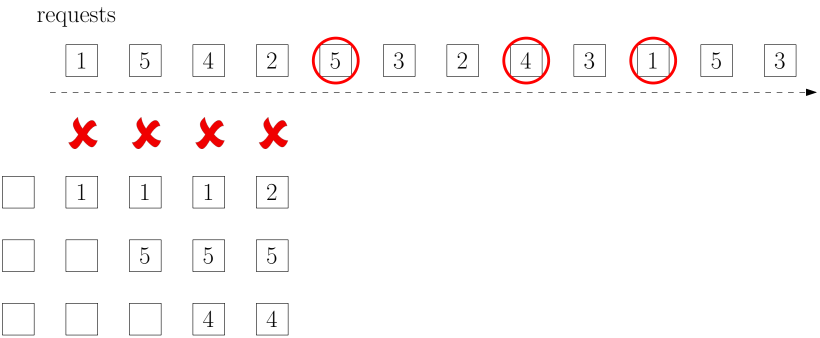
Example

Example

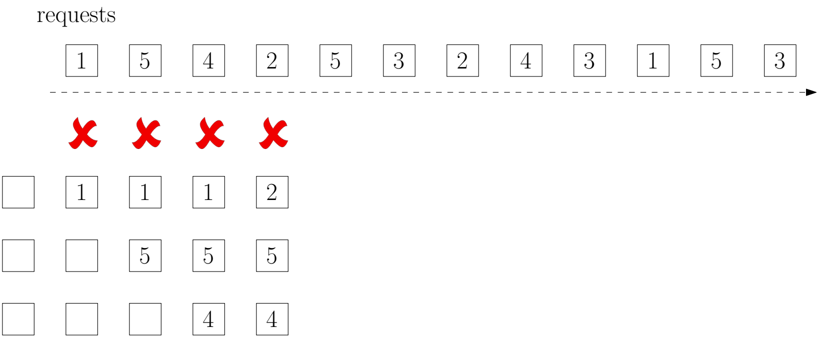
Example



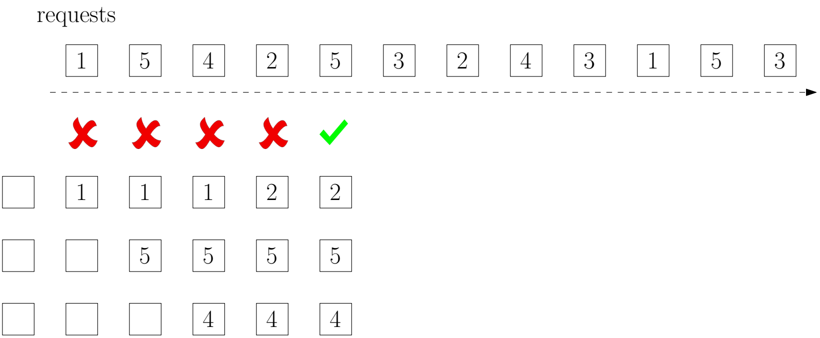
Example



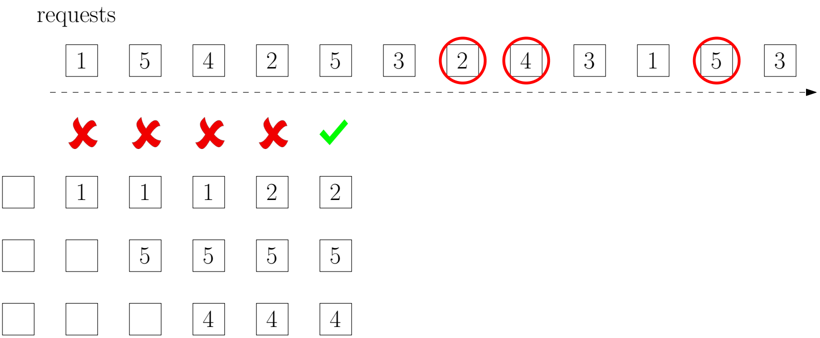
Example



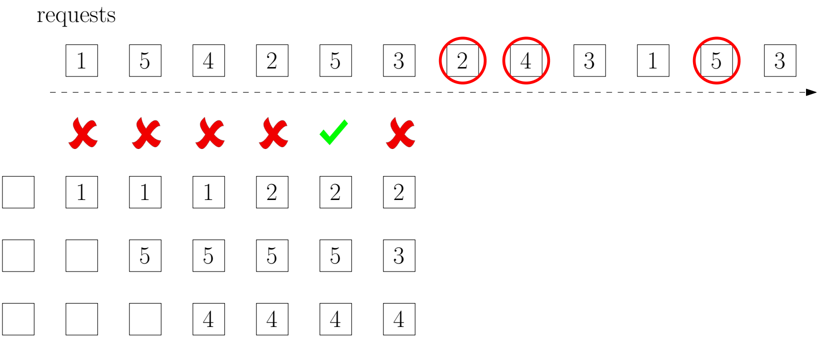
Example



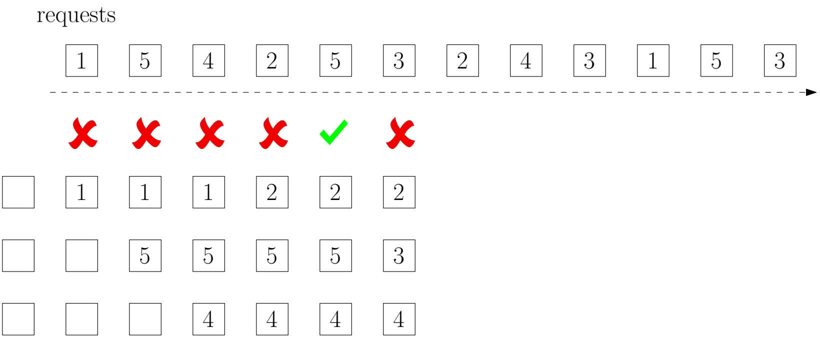
Example

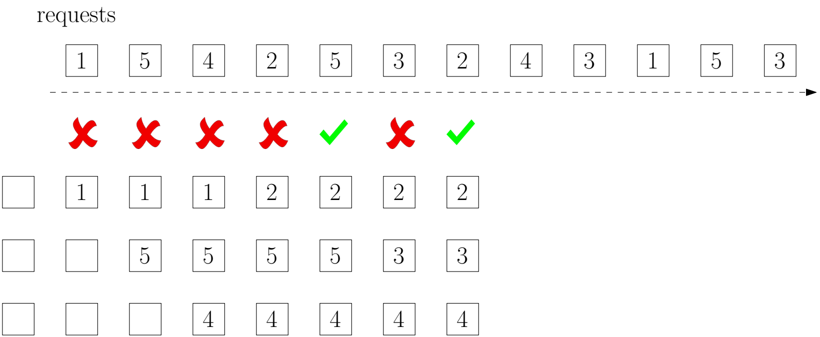


Example



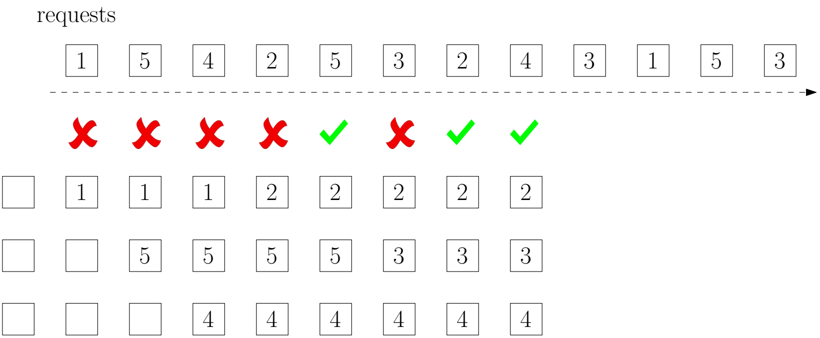
Example



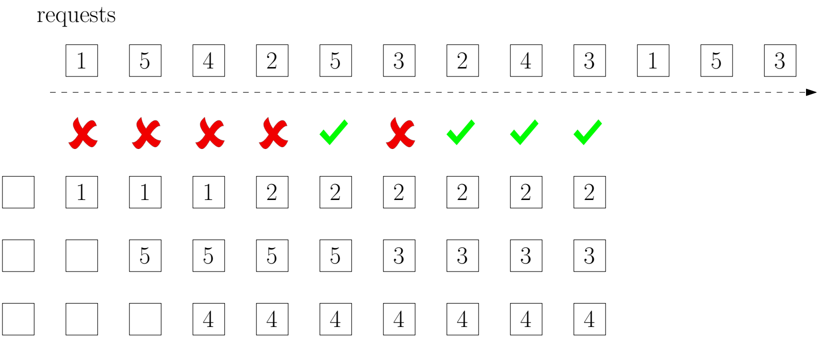


requests
1 5 4 2 5 3 2 4 3 1 5 3
1 1 1 2 2 2 2
5 5 5 5 3 3
4 4 4 4 4

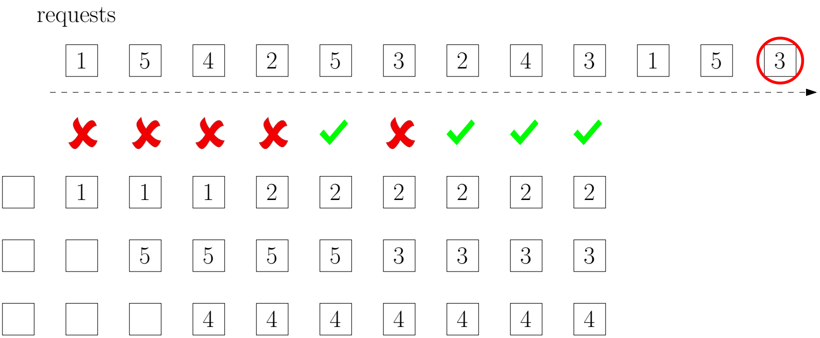
Example



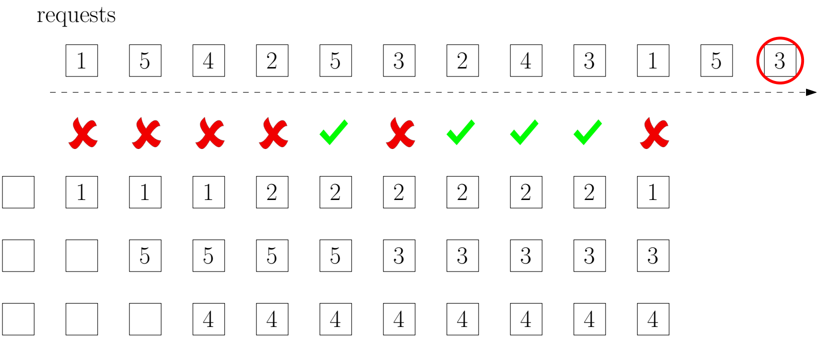
Example



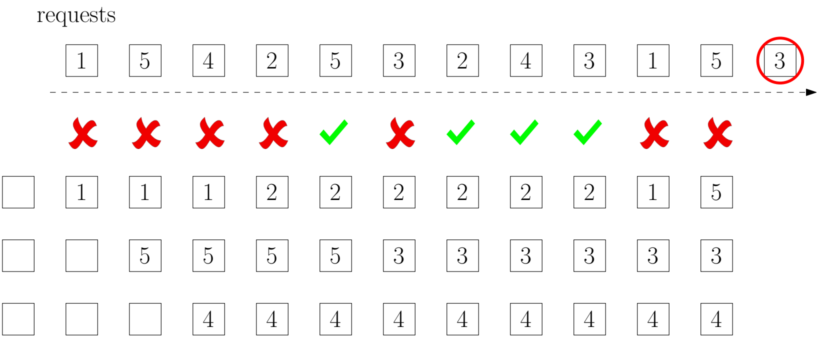
Example



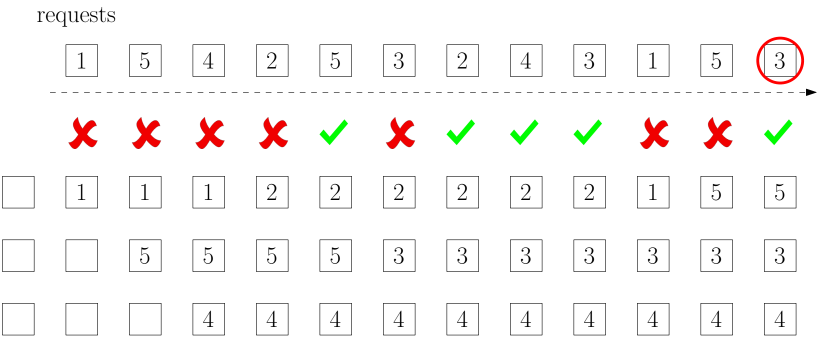
Example



Example


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

Example


requests
1 5 4 2 5 3 2 4 3 1 5 3
1 1 1 2 2 2 2 2 2 1 5 5
5 5 5 5 3 3 3 3 3 3 3
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: $i_1, i_2, i_3, \cdots, i_t \in \{\mathrm{hit}, \text{empty}\} \cup [n]$

- empty stands for an empty page
- "hit" means evicting no pages

Offline Caching Problem

Input: k : the size of cache

n : number of pages

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

$p_1, p_2, \cdots, p_k \in \{\text{empty}\} \cup [n]$: initial set of pages in cache

Output: $i_1, i_2, i_3, \cdots, i_t \in \{\text{hit}, \text{empty}\} \cup [n]$

- empty stands for an empty page
- "hit" means evicting no pages