CSE 431/531: Algorithm Analysis and Design (Fall 2024)
Greedy Algorithms

Lecturer: Kelin Luo

Department of Computer Science and Engineering
University at Buffalo

Outline

1. Interval Scheduling
   - Interval Partitioning

2. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue

Outline

1. Interval Scheduling
   - Interval Partitioning

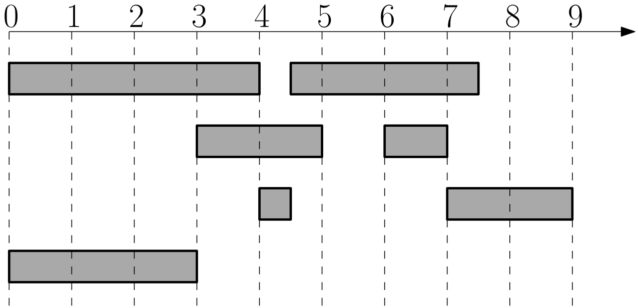
2. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue

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.

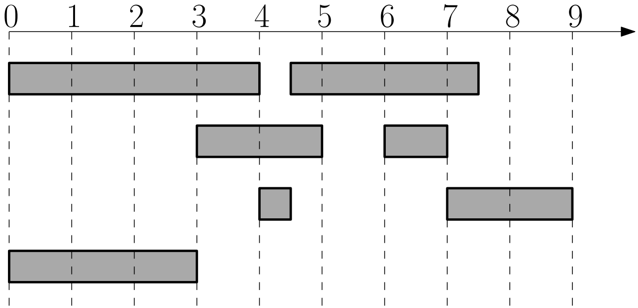


Interval Partitioning

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.

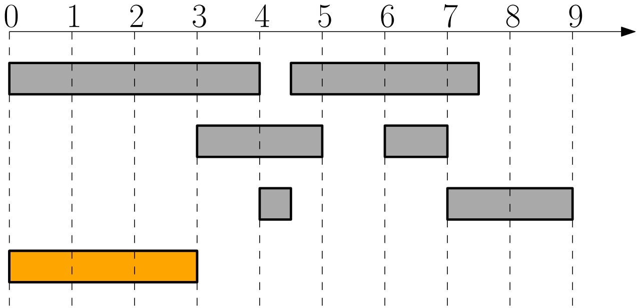


Interval Partitioning

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.

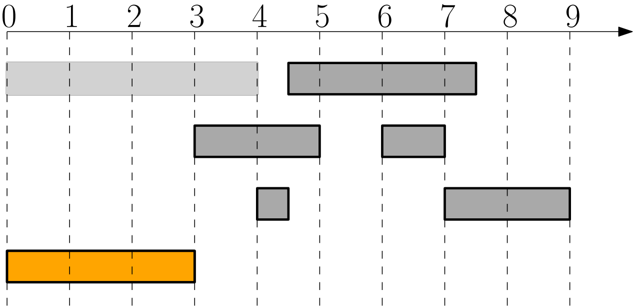


Interval Partitioning

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.

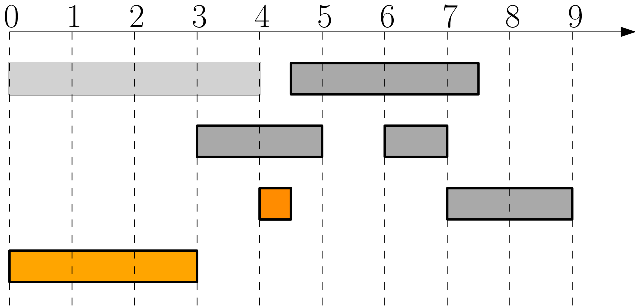


Interval Partitioning

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.

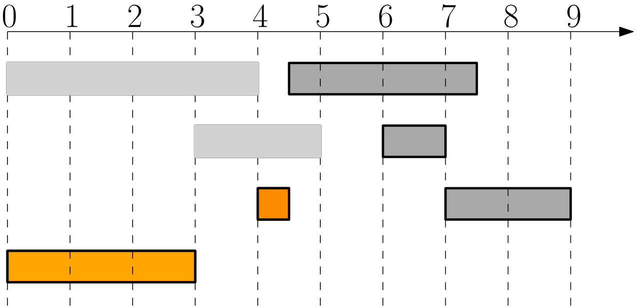


Interval Partitioning

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.

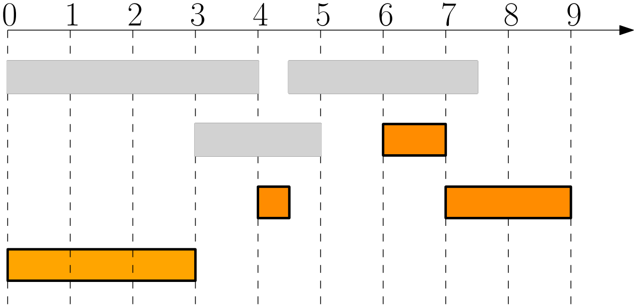


Interval Partitioning

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.

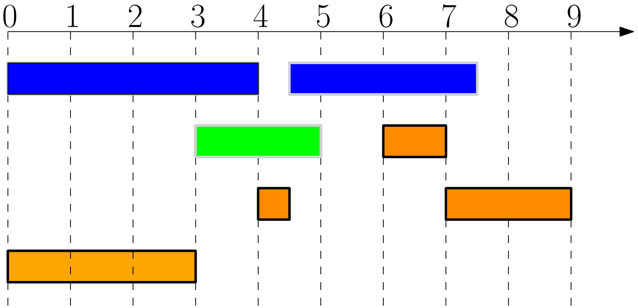


Interval Partitioning

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.

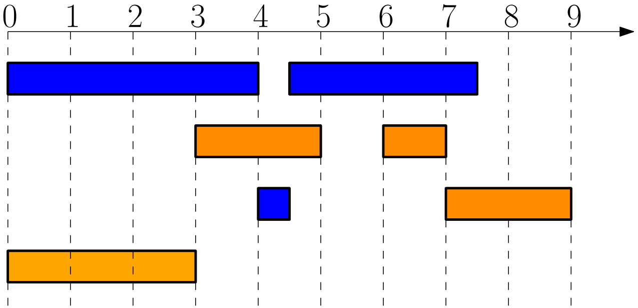


Interval Partitioning

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

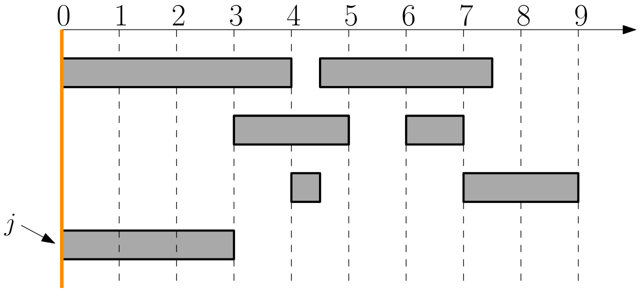
Lemma It is safe to schedule the job j with the earliest starting time to a feasible machine: There exists an optimum solution where job j with the earliest starting time is scheduled first on a machine that is compatible with all jobs in that machine if applicable; otherwise, it can be scheduled by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

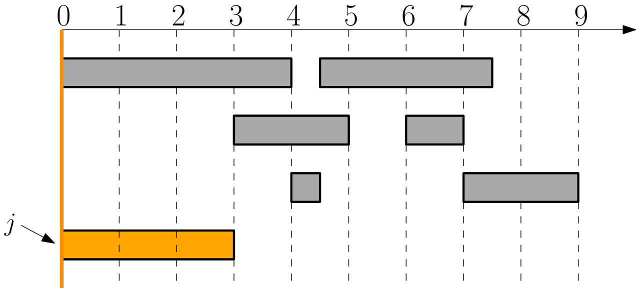
Greedy Algorithm for Interval Partitioning

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



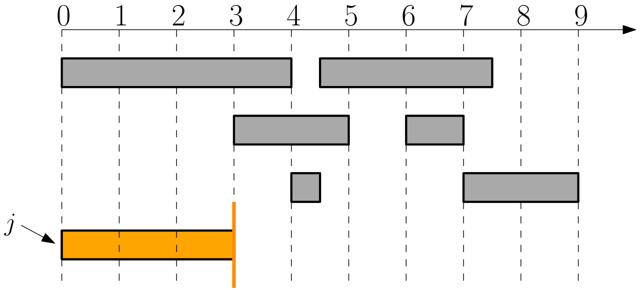
Greedy Algorithm for Interval Partitioning

- 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!

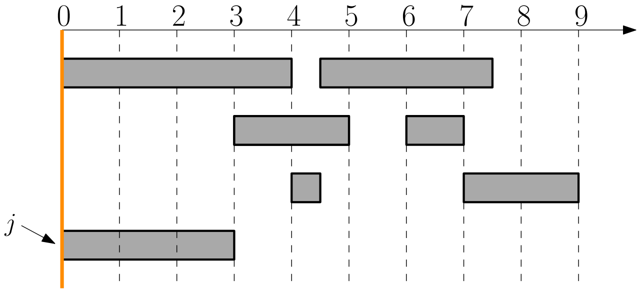


Greedy Algorithm for Interval Partitioning

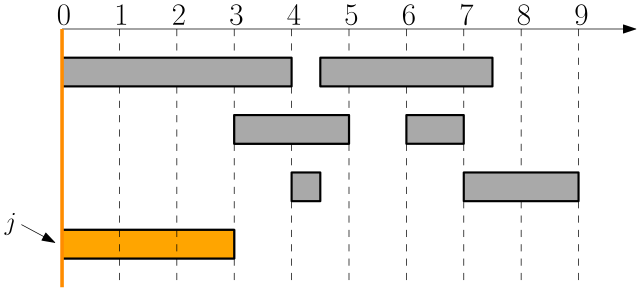
Partition(s, f, n)

$A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \{1\}$, $t_1 = 0$
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

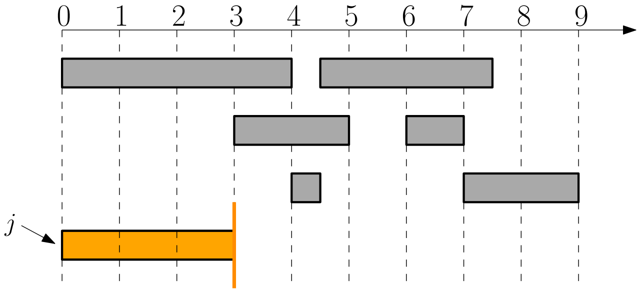
Greedy Algorithm for Interval Partitioning



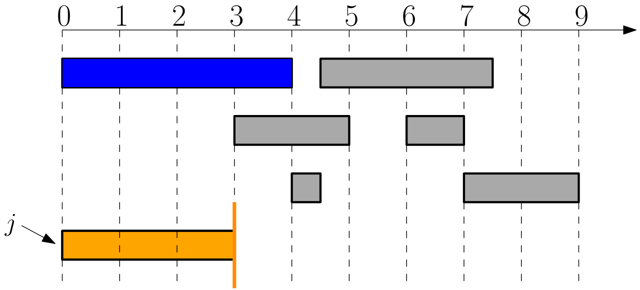
Greedy Algorithm for Interval Partitioning



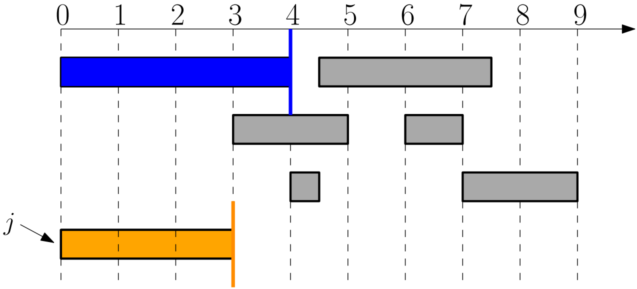
Greedy Algorithm for Interval Partitioning



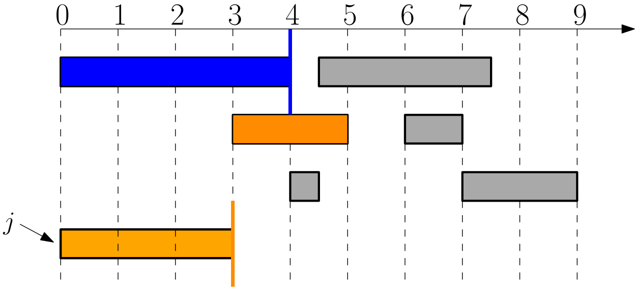
Greedy Algorithm for Interval Partitioning



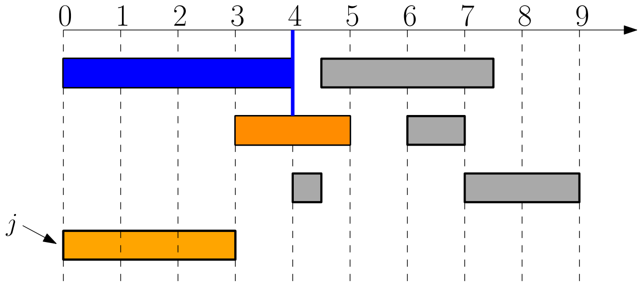
Greedy Algorithm for Interval Partitioning



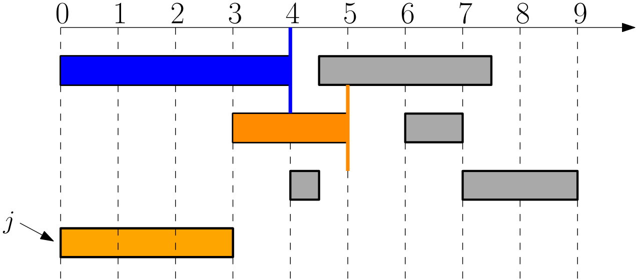
Greedy Algorithm for Interval Partitioning



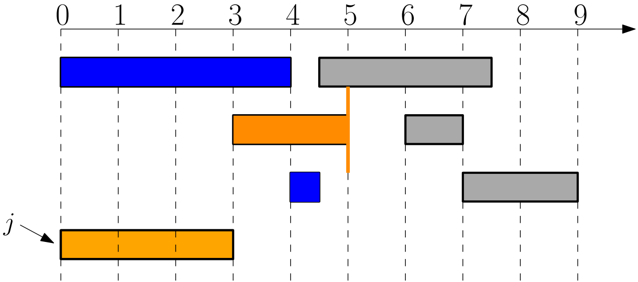
Greedy Algorithm for Interval Partitioning



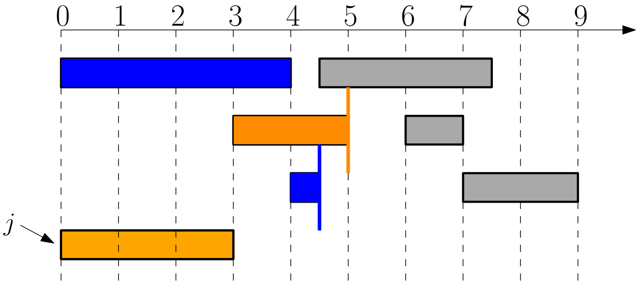
Greedy Algorithm for Interval Partitioning



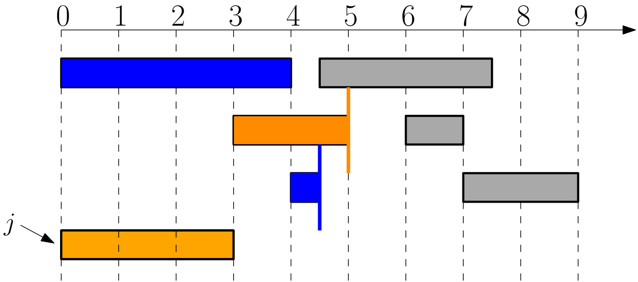
Greedy Algorithm for Interval Partitioning



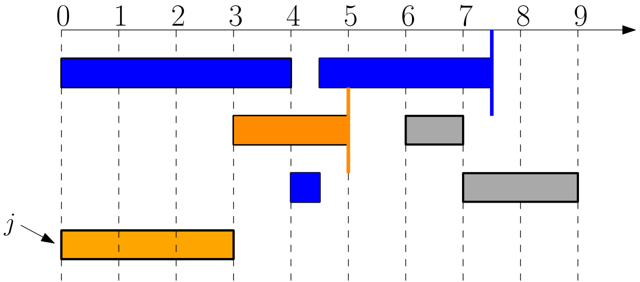
Greedy Algorithm for Interval Partitioning



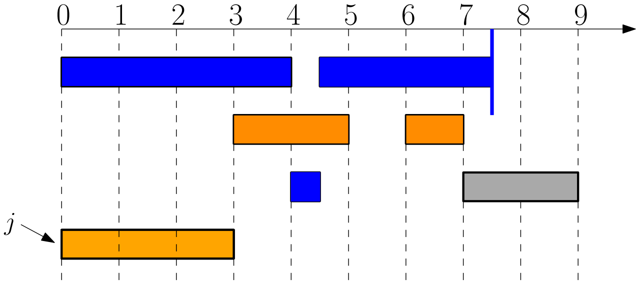
Greedy Algorithm for Interval Partitioning



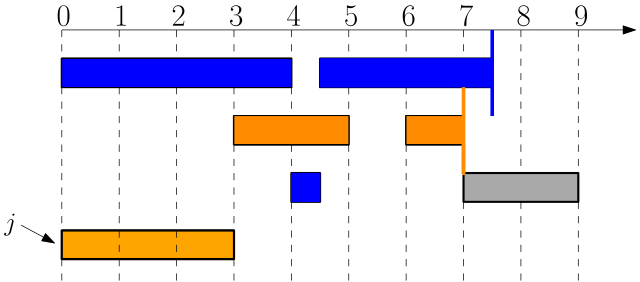
Greedy Algorithm for Interval Partitioning



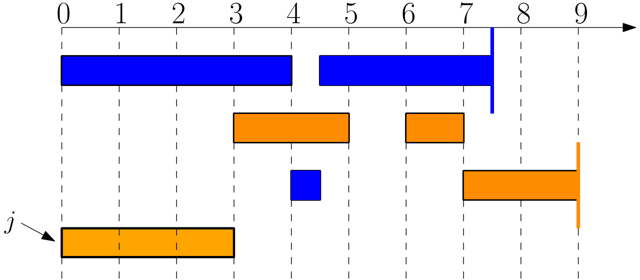
Greedy Algorithm for Interval Partitioning



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.

$\square$

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.

$\square$

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 ← {1, 2, ··· , n}, S ← {1}, t_1 = 0
while A ≠ ∅ do
    j ← arg min_{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
```

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.

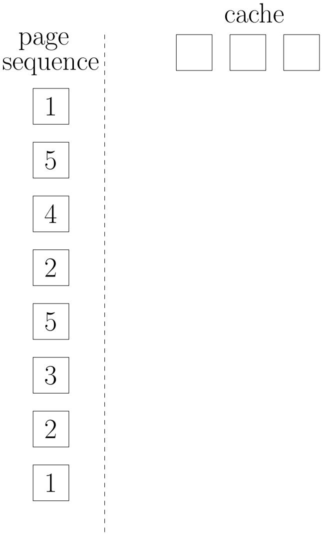
Outline

Offline Caching

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

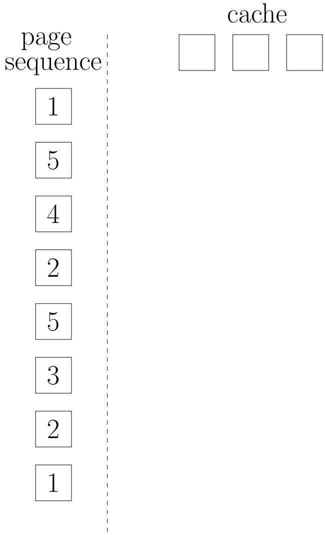
Offline Caching

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



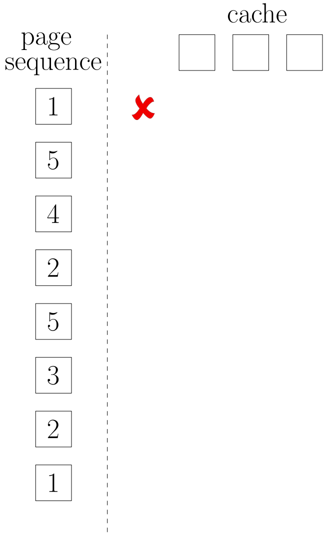
Offline Caching

- 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.



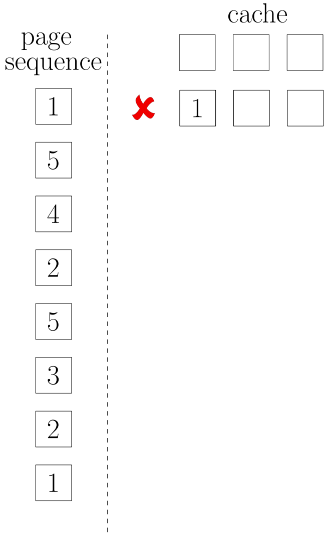
Offline Caching

- 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.



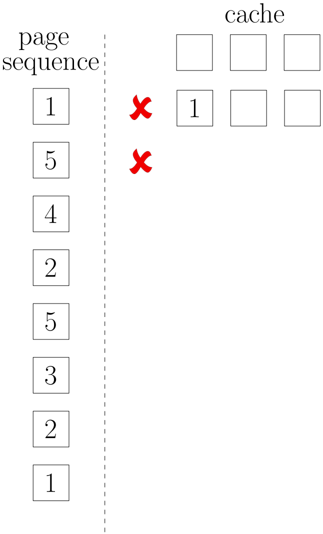
Offline Caching

- 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.



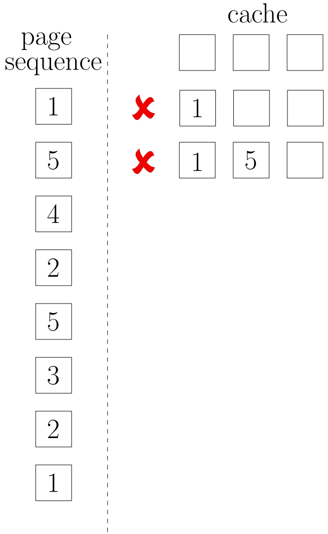
Offline Caching

- 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.



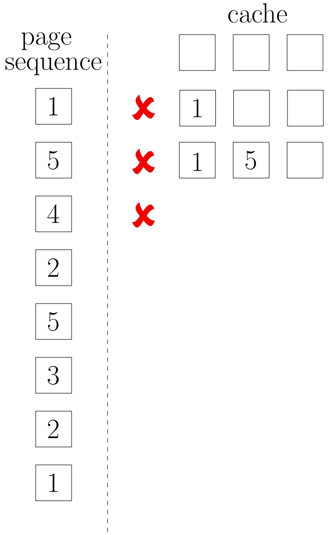
Offline Caching

- 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.



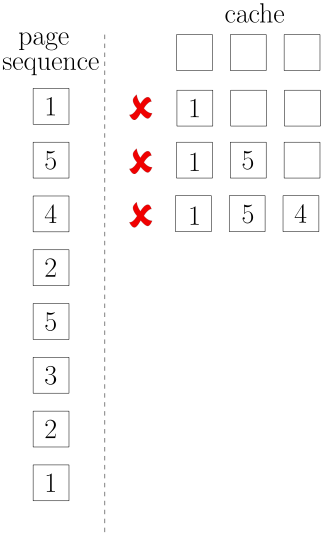
Offline Caching

- 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.



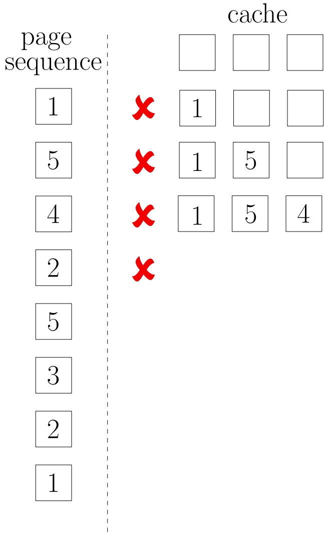
Offline Caching

- 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.



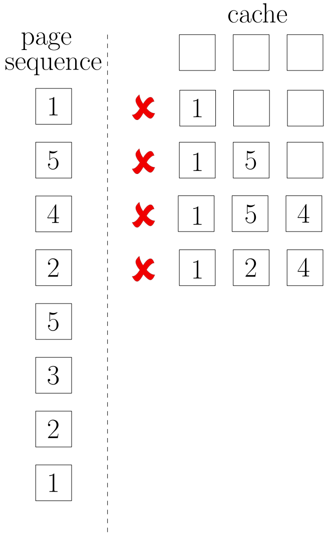
Offline Caching

- 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.



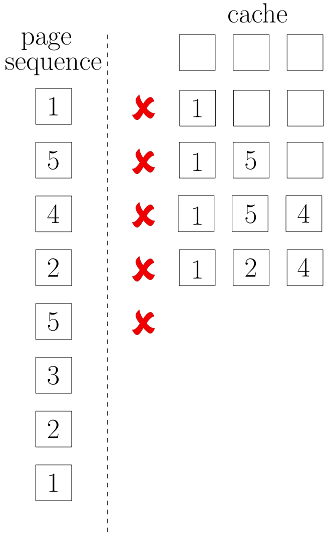
Offline Caching

- 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.



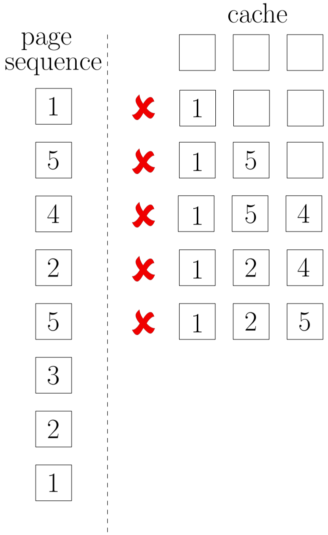
Offline Caching

- 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.



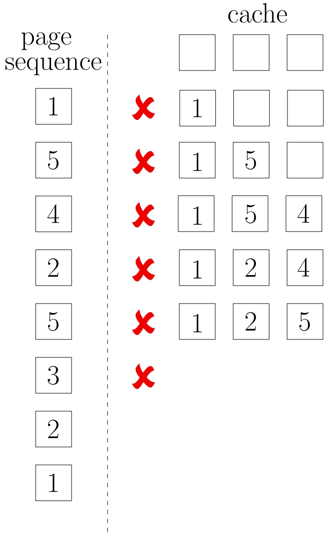
Offline Caching

- 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.



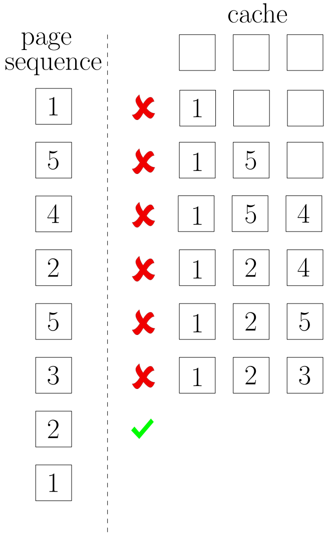
Offline Caching

- 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.

page sequence		cache		
1	✘	1		
5	✘	1	5	
4	✘	1	5	4
2	✘	1	2	4
5	✘	1	2	5
3	✘	1	2	3
2				
1				

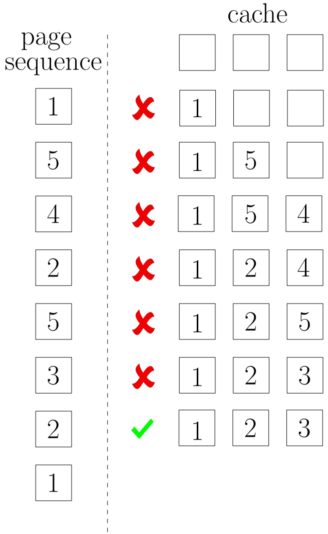
Offline Caching

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



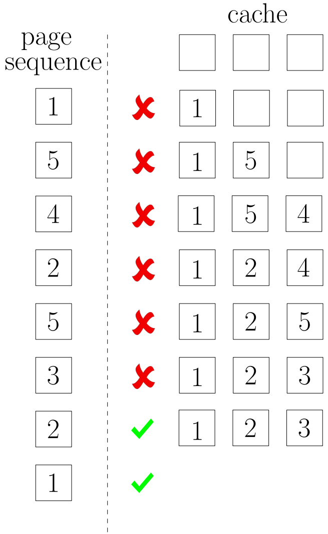
Offline Caching

- 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.



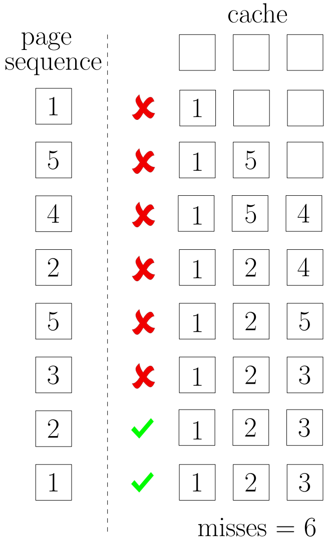
Offline Caching

- 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.

page sequence		cache		
1	✘	1		
5	✘	1	5	
4	✘	1	5	4
2	✘	1	2	4
5	✘	1	2	5
3	✘	1	2	3
2	✓	1	2	3
1	✓	1	2	3

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.

page sequence		cache		
1	✗	1		
5	✗	1	5	
4	✗	1	5	4
2	✗	1	2	4
5	✗	1	2	5
3	✗	1	2	3
2	✓	1	2	3
1	✓	1	2	3

misses $= 6$

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)