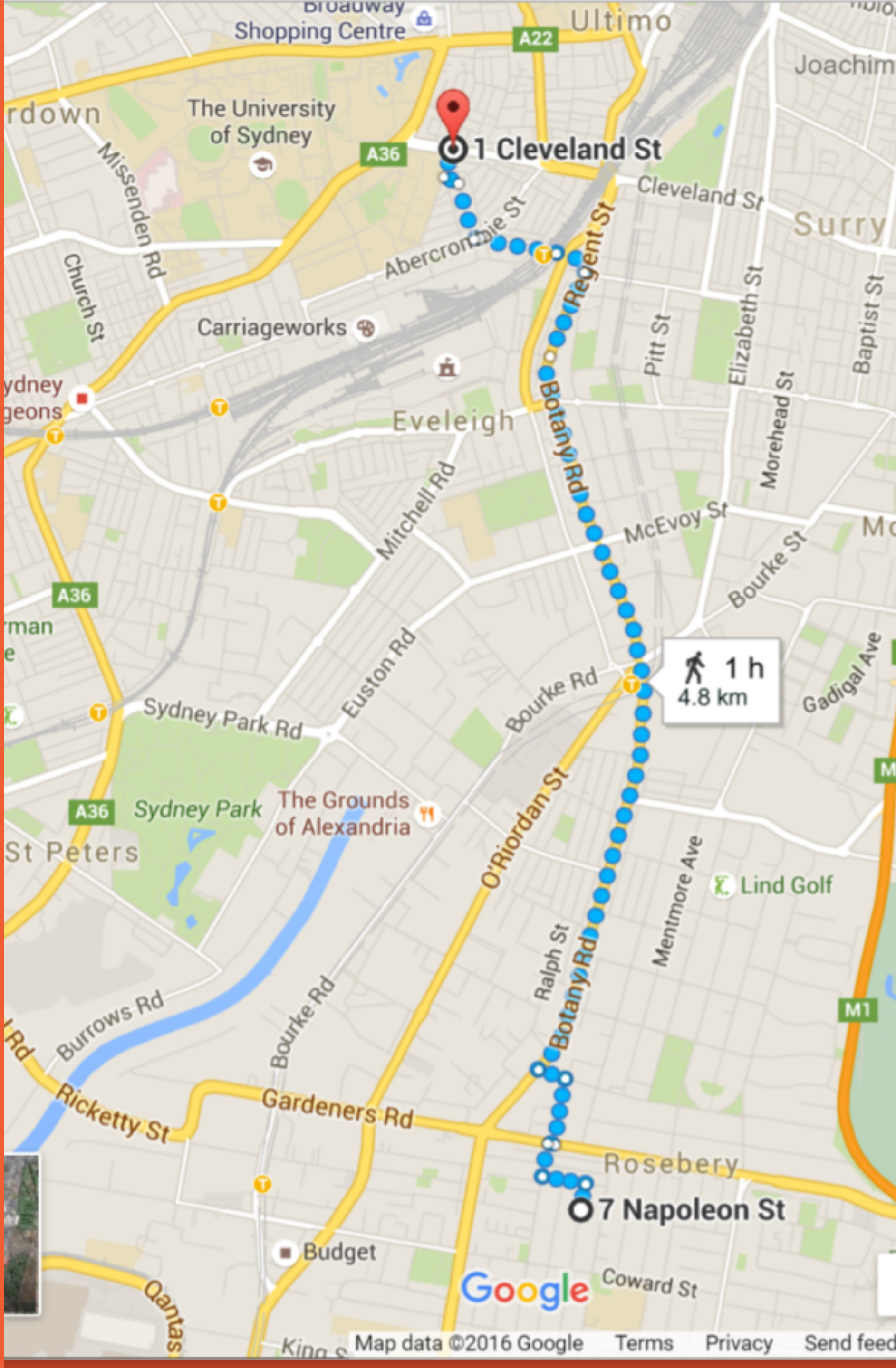
Lecture 2: Greedy algorithms [Ch 4 KT]

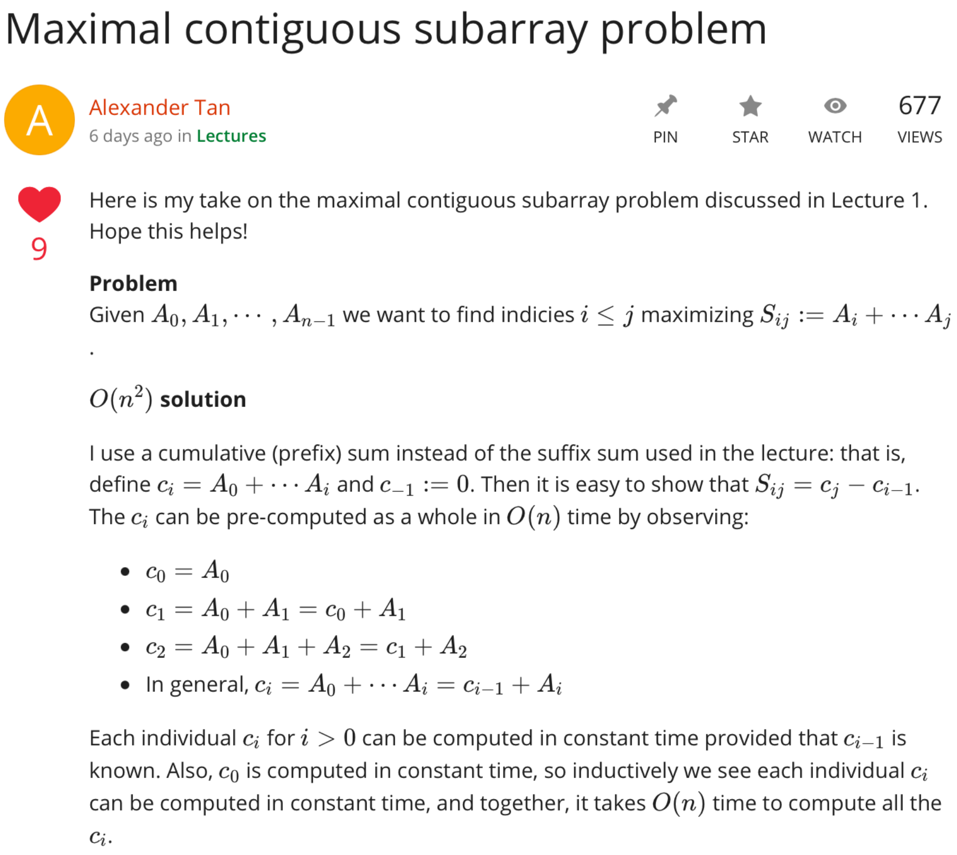
William Umboh

School of Computer Science

THE UNIVERSITY OF SYDNEY



Ed Participation



Maximal contiguous subarray problem

Alexander Tan
6 days ago in Lectures

Here is my take on the maximal contiguous subarray problem discussed in Lecture 1. Hope this helps!

Problem
Given $A_0, A_1, \cdots, A_{n-1}$ we want to find indicies $i \leq j$ maximizing $S_{ij} := A_i + \cdots A_j$
.

$O(n^2)$ solution

I use a cumulative (prefix) sum instead of the suffix sum used in the lecture: that is, define $c_i = A_0 + \cdots A_i$ and $c_{-1} := 0$. Then it is easy to show that $S_{ij} = c_j - c_{i-1}$. The c_i can be pre-computed as a whole in $O(n)$ time by observing:

- $c_0 = A_0$
- $c_1 = A_0 + A_1 = c_0 + A_1$
- $c_2 = A_0 + A_1 + A_2 = c_1 + A_2$
- In general, $c_i = A_0 + \cdots A_i = c_{i-1} + A_i$

Each individual c_i for $i > 0$ can be computed in constant time provided that c_{i-1} is known. Also, c_0 is computed in constant time, so inductively we see each individual c_i can be computed in constant time, and together, it takes $O(n)$ time to compute all the c_i.

- Feel free to discuss your ideas on Ed
- Improvements/different take on lecture materials, attempts to exercises from textbooks, are welcome
- **DO NOT POST SOLUTIONS TO ASSIGNMENTS. DUH**

General techniques in this course

- Greedy algorithms [today]
- Divide & Conquer algorithms [W3]
- Dynamic programming algorithms [W4-5]
- Network flow algorithms [W6-7]

Greedy algorithms

A greedy algorithm is an algorithm that follows the problem solving approach of making a locally optimal choice at each stage with the hope of finding a global optimum.

Greedy algorithms can be some of the simplest algorithms to implement, but they're often among the hardest algorithms to design and analyse.

Greedy: Overview

Consider problems that can be solved using a greedy approach:

- Interval scheduling/partitioning
- Scheduling to minimize lateness
- Paging
- Shortest path [COMP2123]
- Minimum spanning trees [COMP2123]

How to design algorithms

Step 1: Understand problem

Step 4: Better understanding of problem

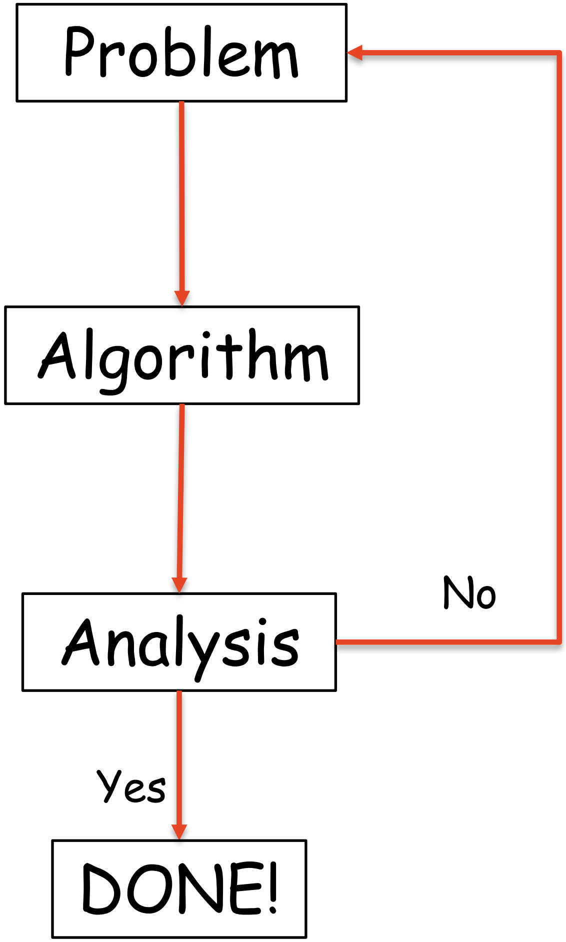
Step 2: Start with simple alg.

Step 5: Improved alg.

Step 3: Does it work? Is it fast?

Useful strategy:

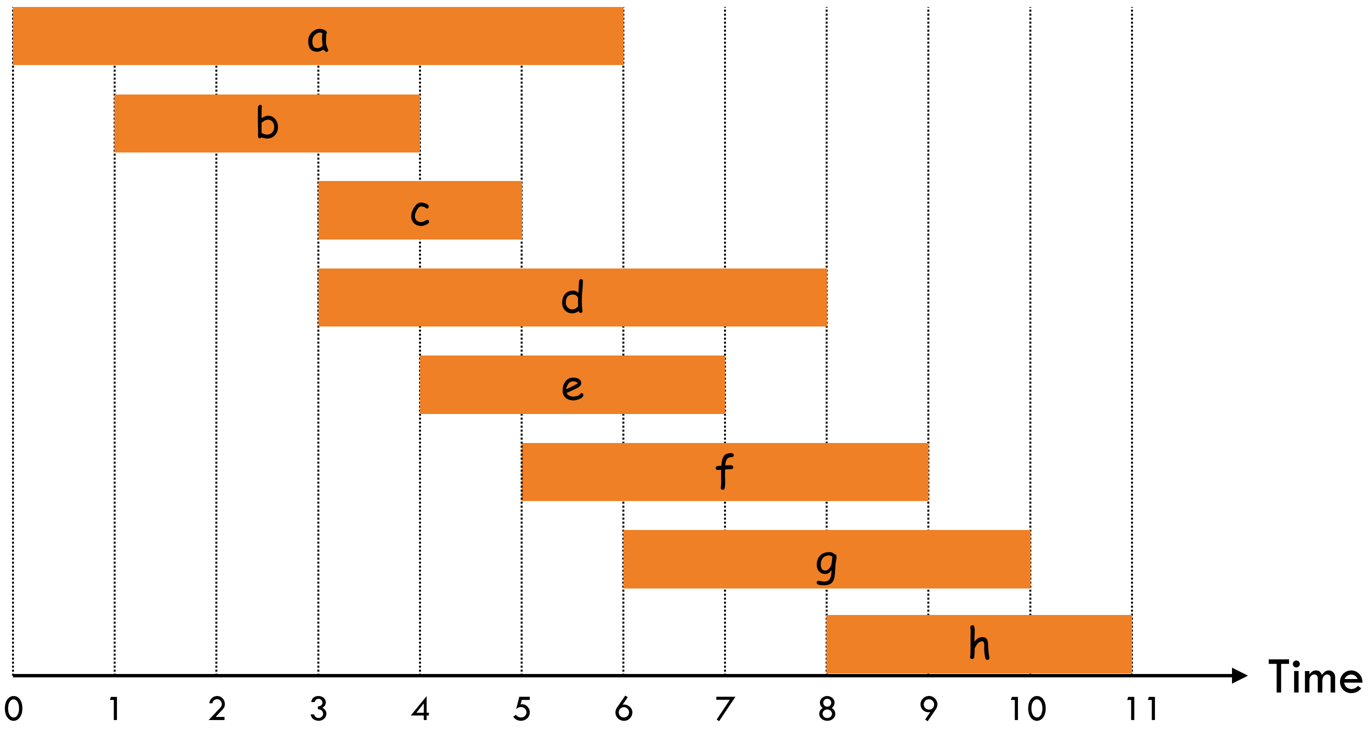
- Try some simple examples to get feel for algorithm.
- If none of them break algorithm, see if there's underlying structural property we can use to prove correctness.



Interval Scheduling

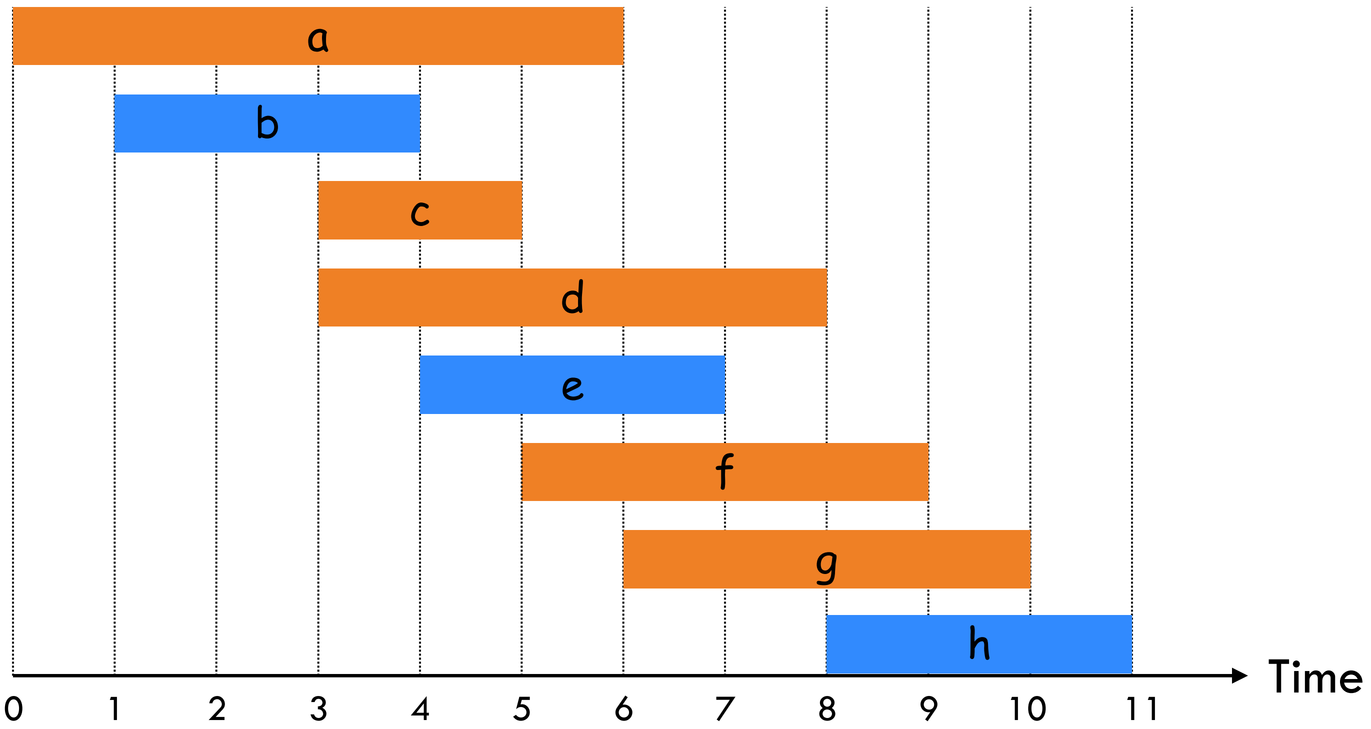
Interval Scheduling

- Interval scheduling.
  - **Input:** Set of n jobs. Each job i starts at time s_i and finishes at time f_i.
  - Two jobs are compatible if they don't overlap in time.
  - **Goal:** find maximum subset of mutually compatible jobs.



Interval Scheduling

- Interval scheduling.
  - **Input:** Set of n jobs. Each job i starts at time s_i and finishes at time f_i.
  - Two jobs are compatible if they don't overlap in time.
  - **Goal:** find maximum subset of mutually compatible jobs.

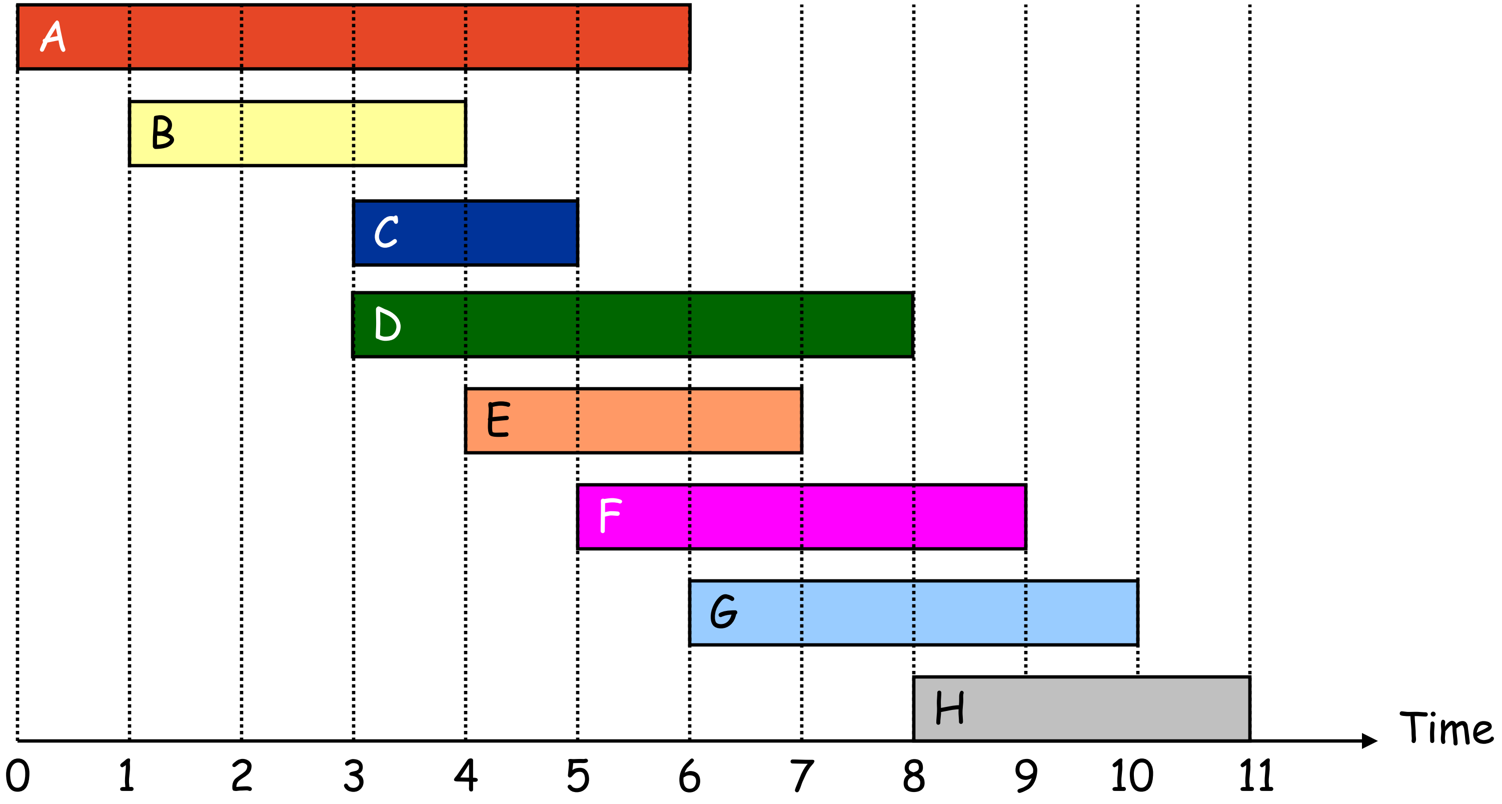


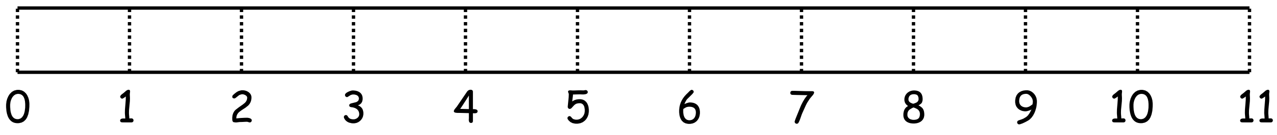
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.

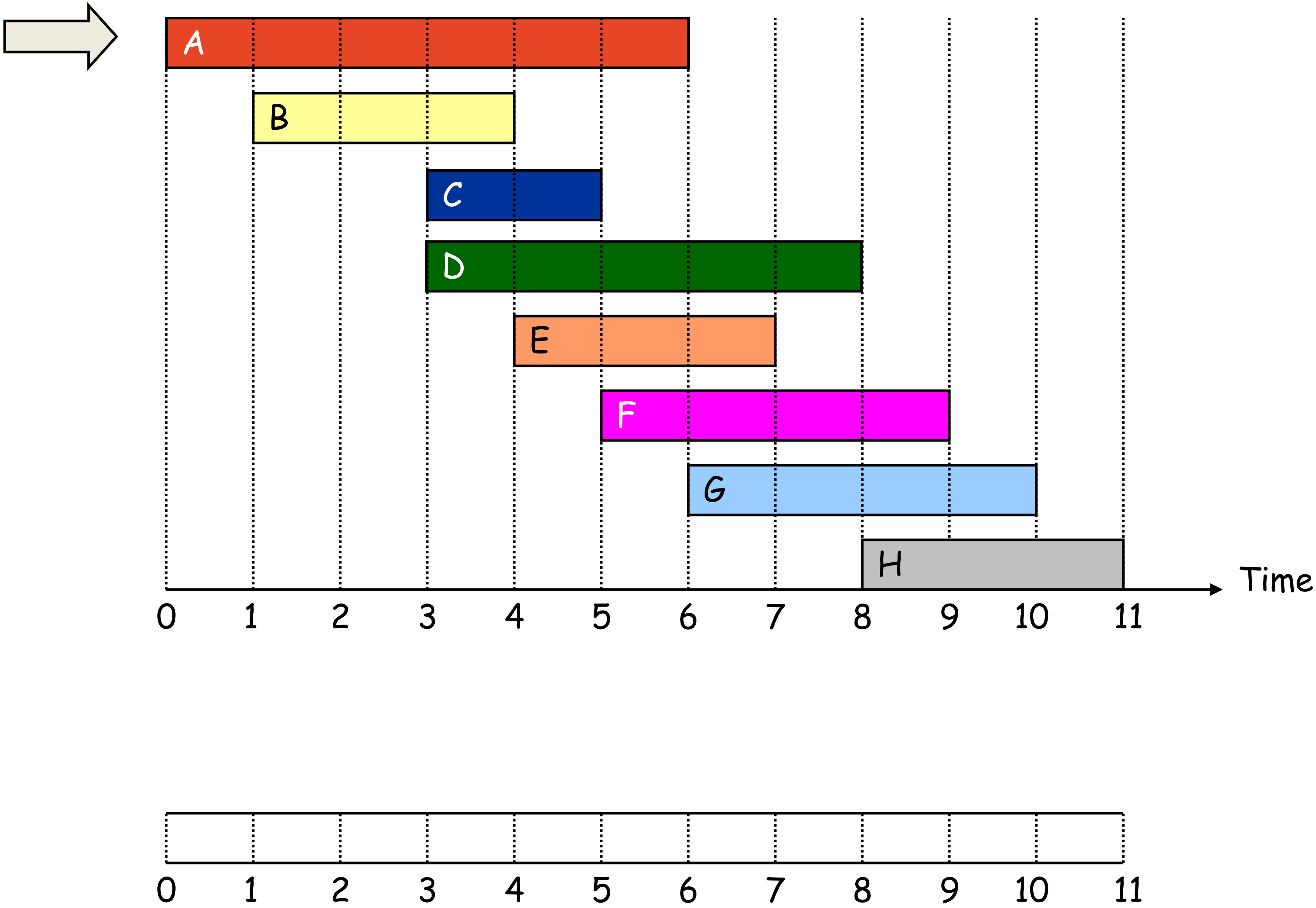
- [Earliest start time] Consider jobs in ascending order of start time s_j.

Interval Scheduling - [Earliest start time]

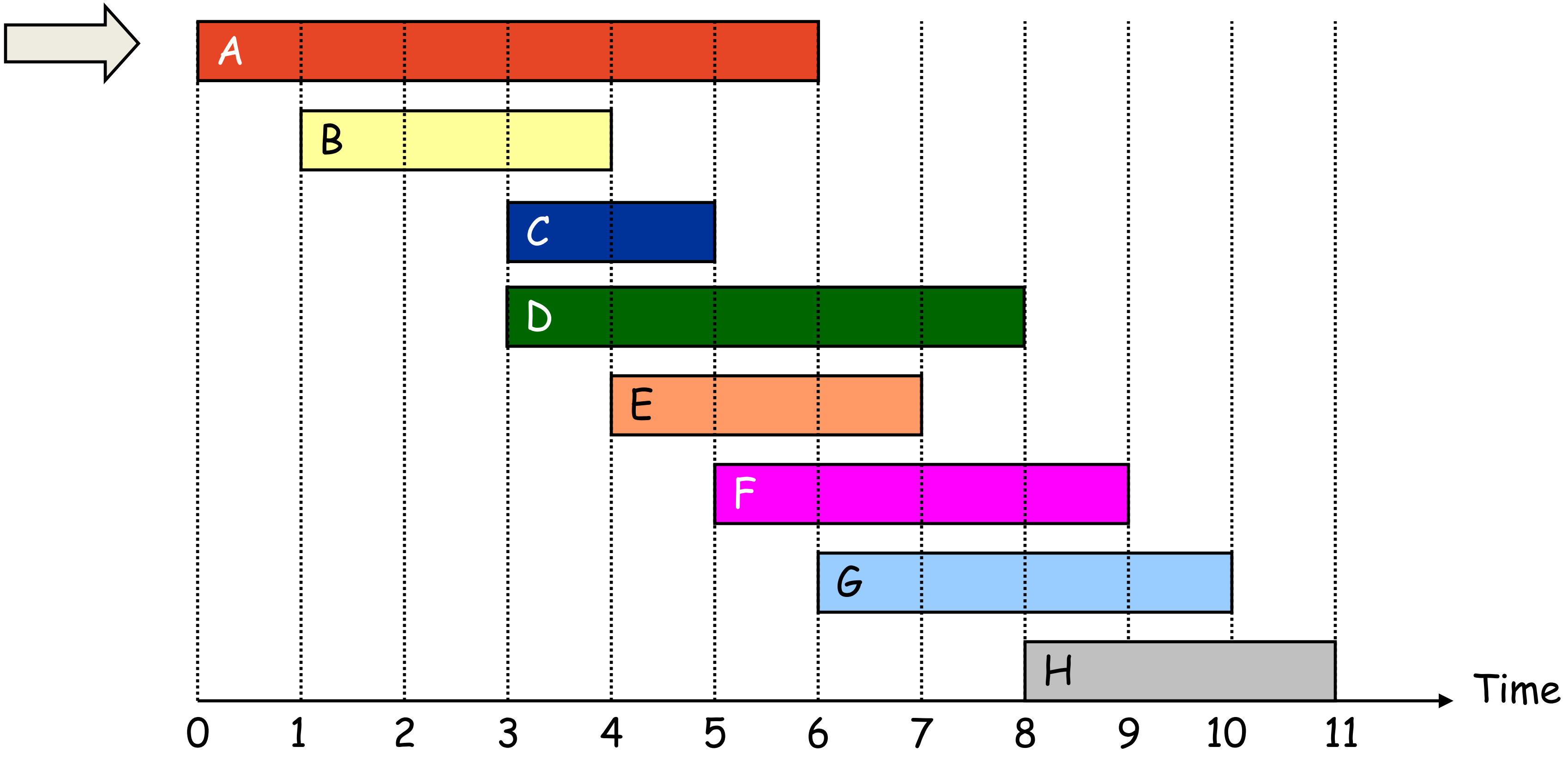


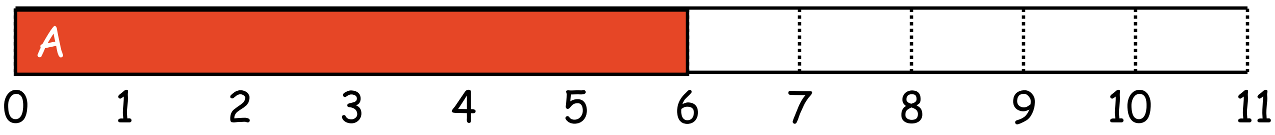


Interval Scheduling - [Earliest start time]

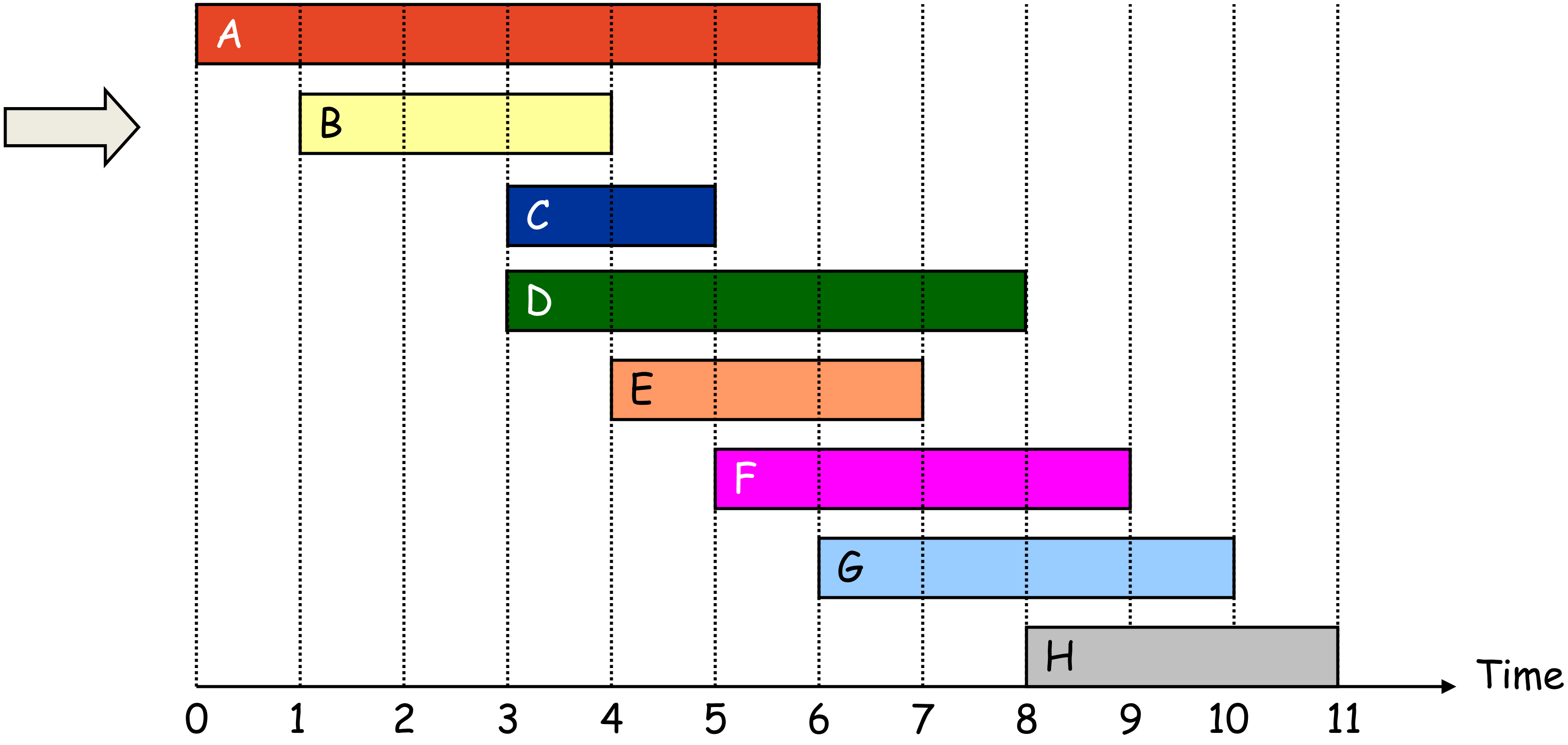


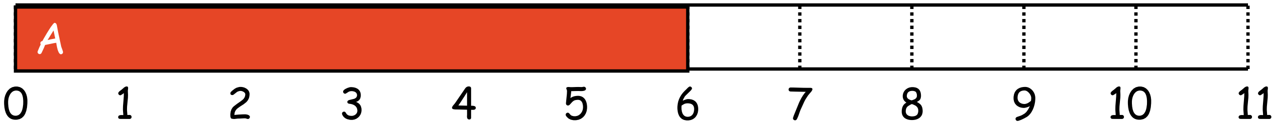
Interval Scheduling - [Earliest start time]



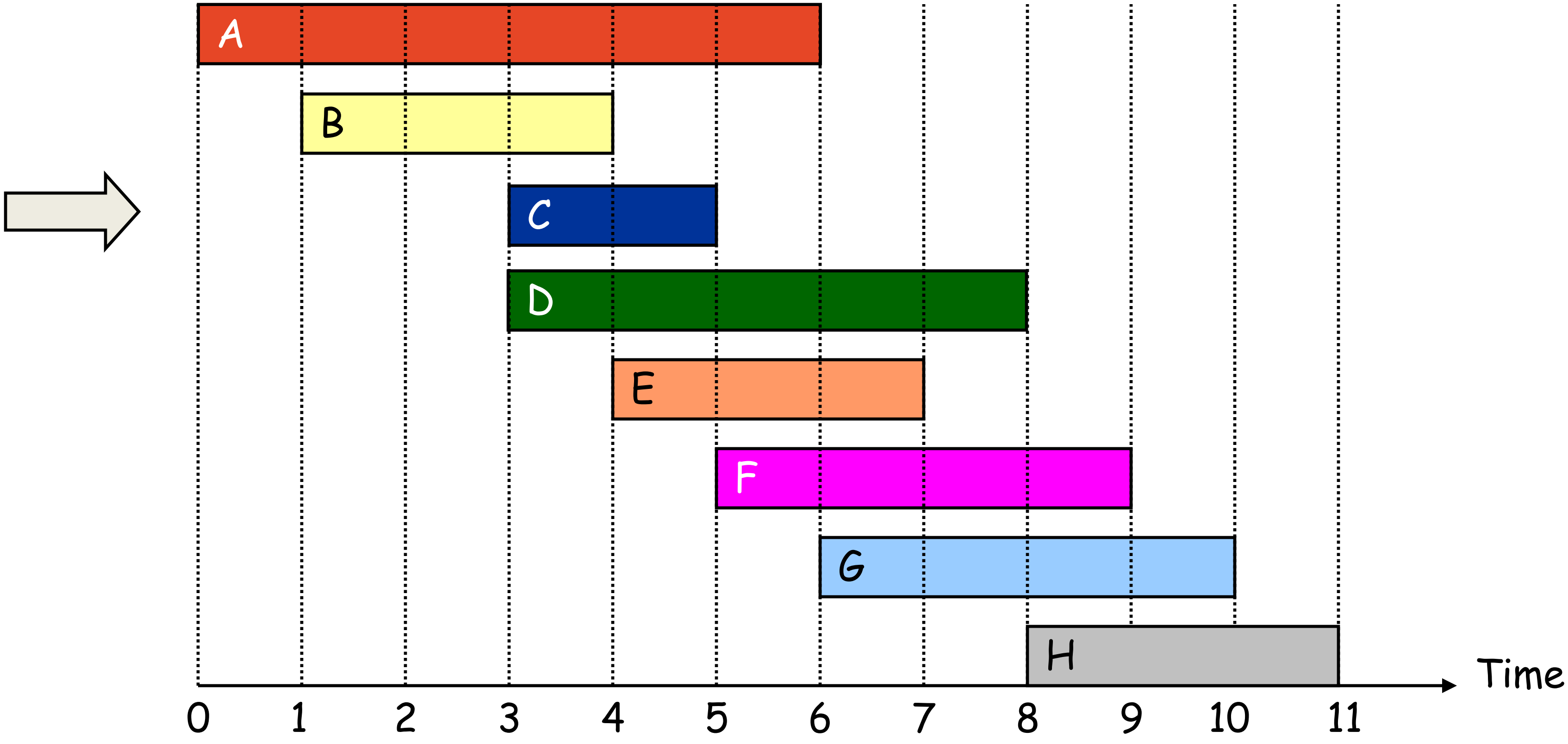


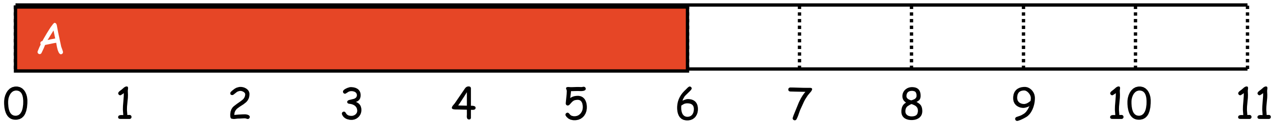
Interval Scheduling - [Earliest start time]



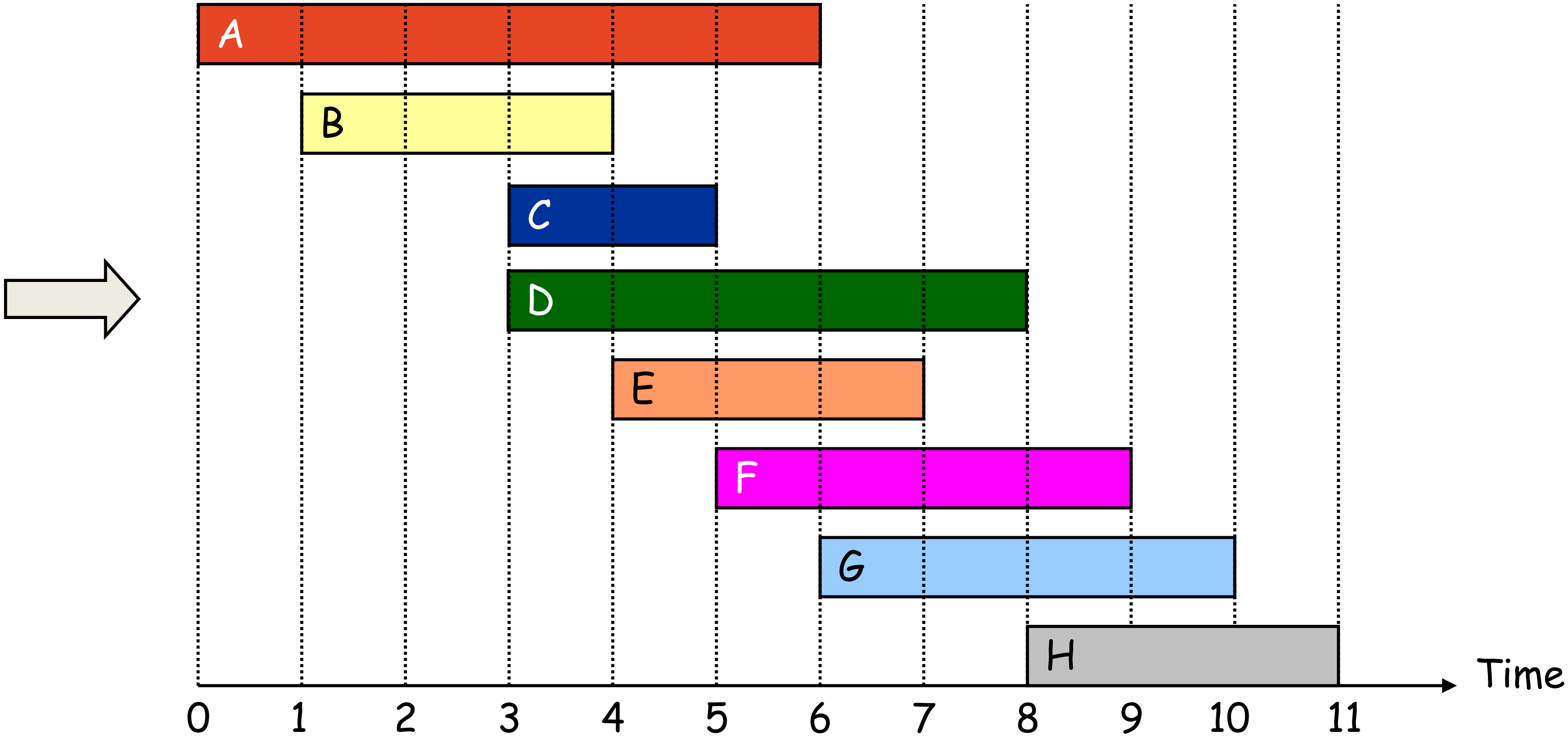


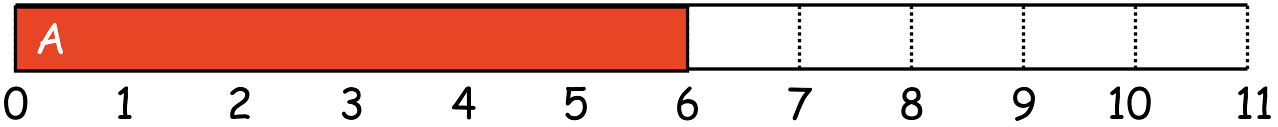
Interval Scheduling - [Earliest start time]



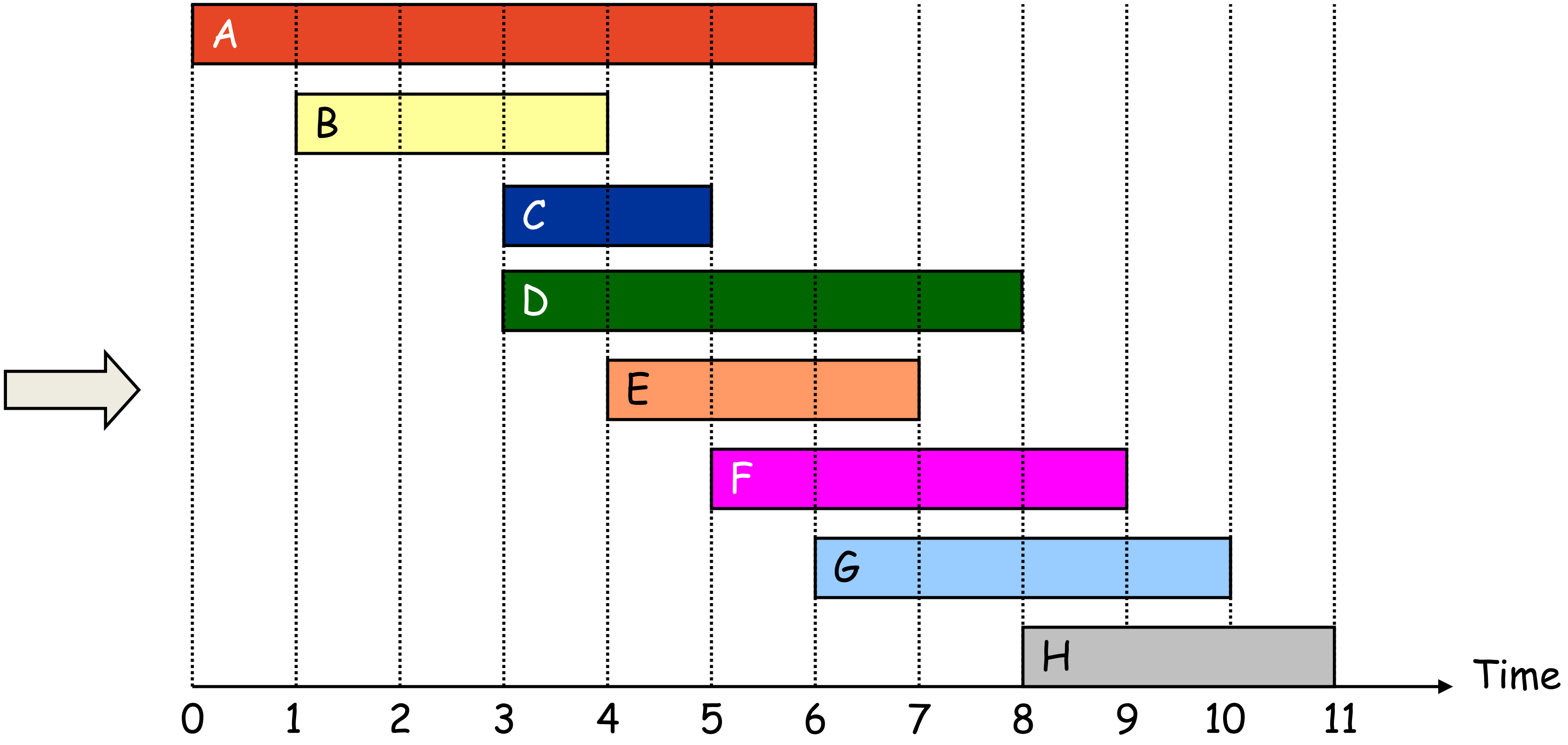


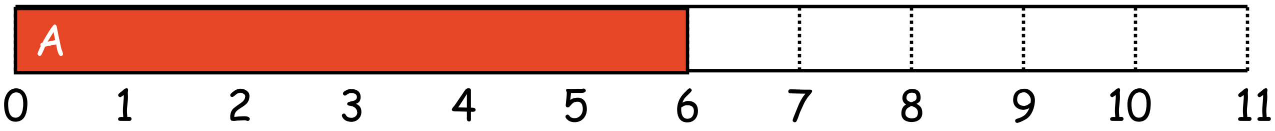
Interval Scheduling - [Earliest start time]



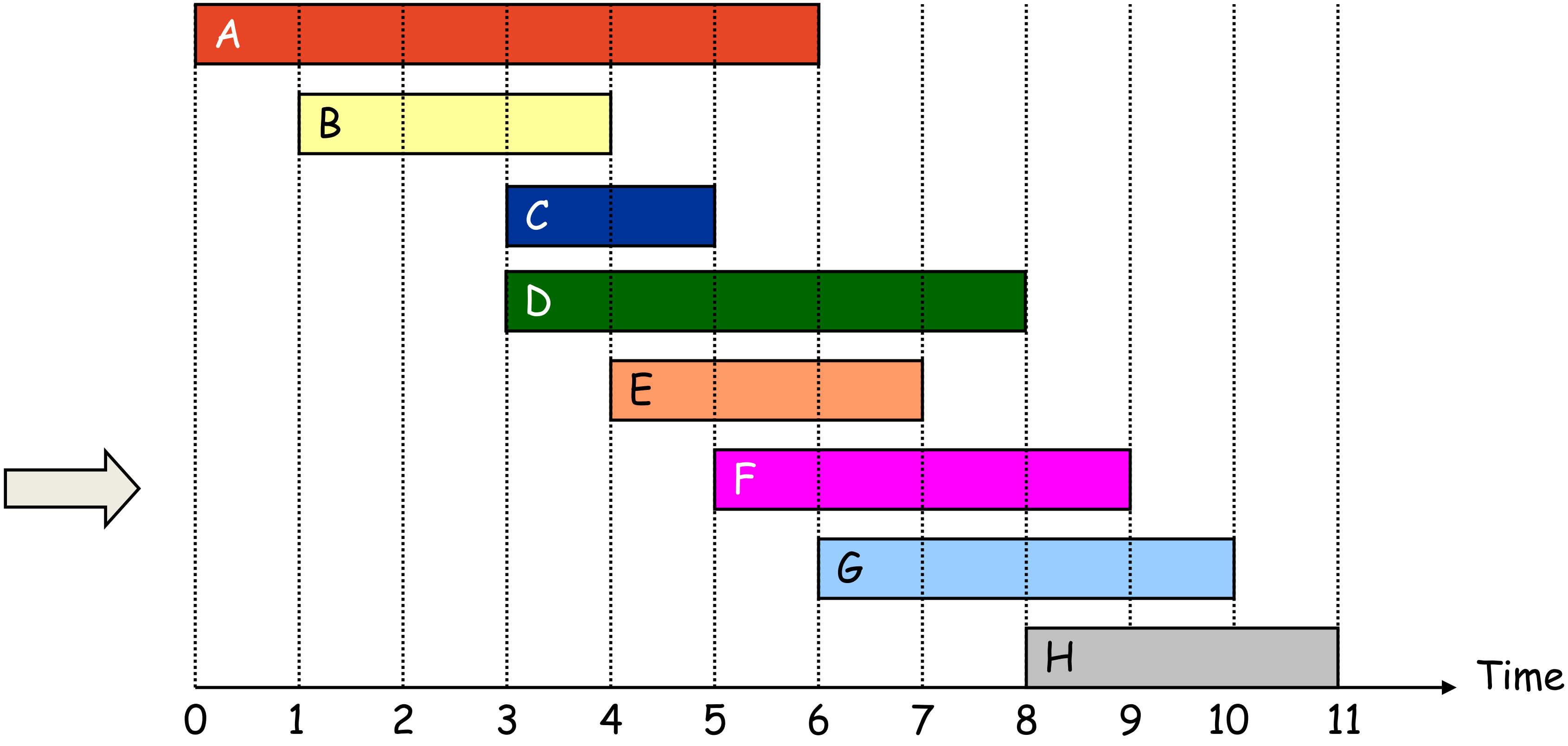


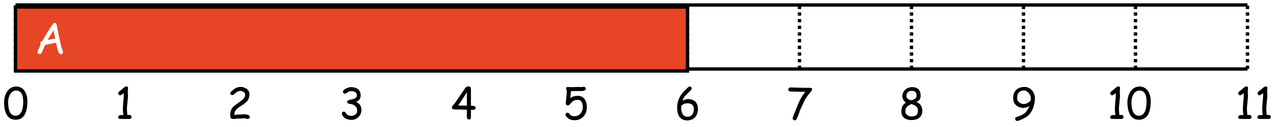
Interval Scheduling - [Earliest start time]



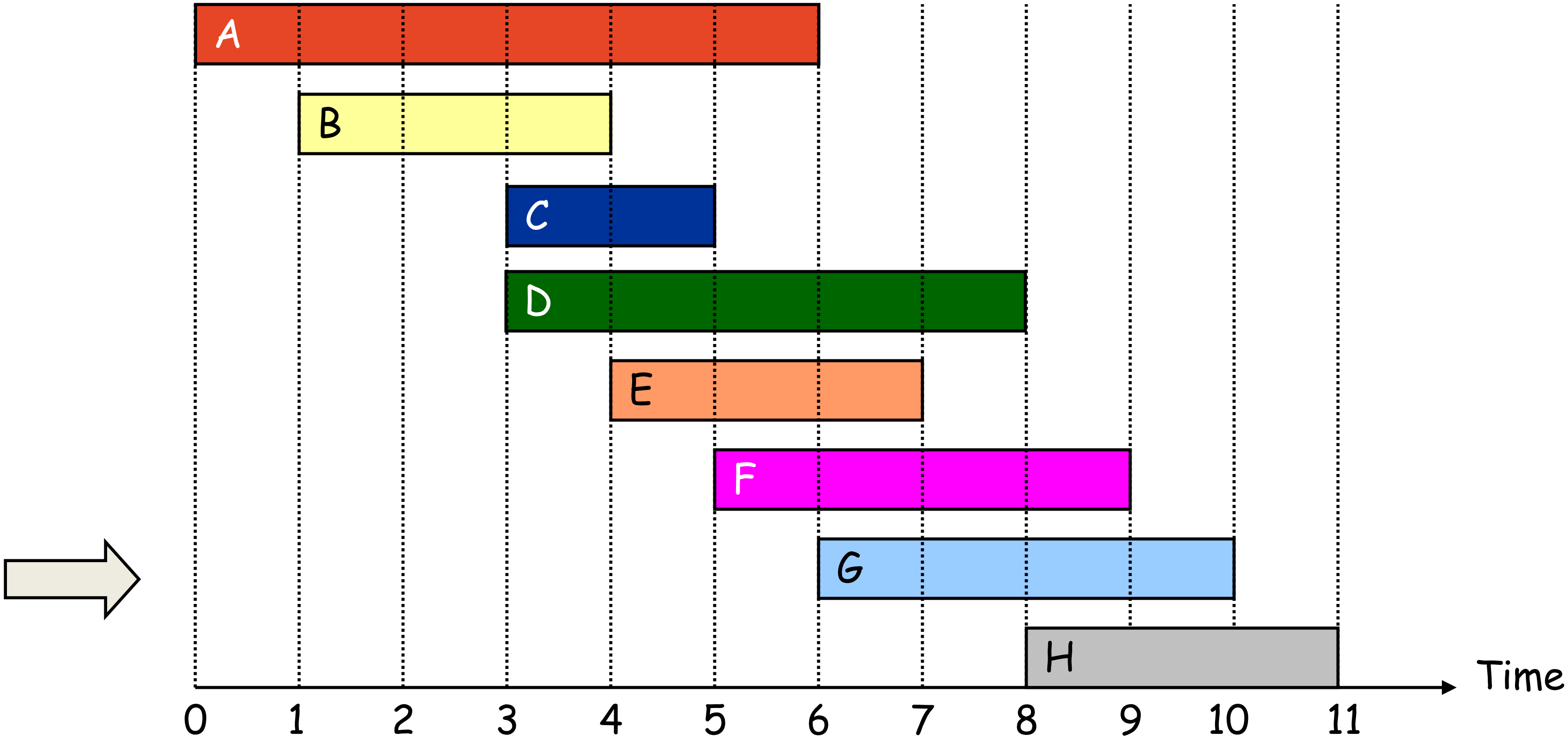


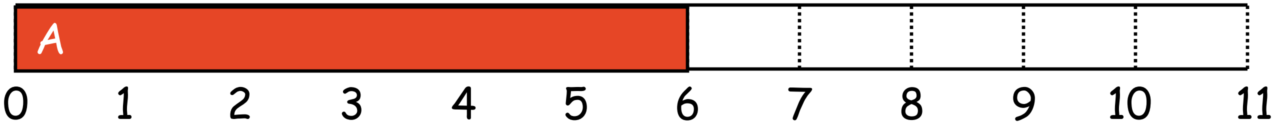
Interval Scheduling - [Earliest start time]



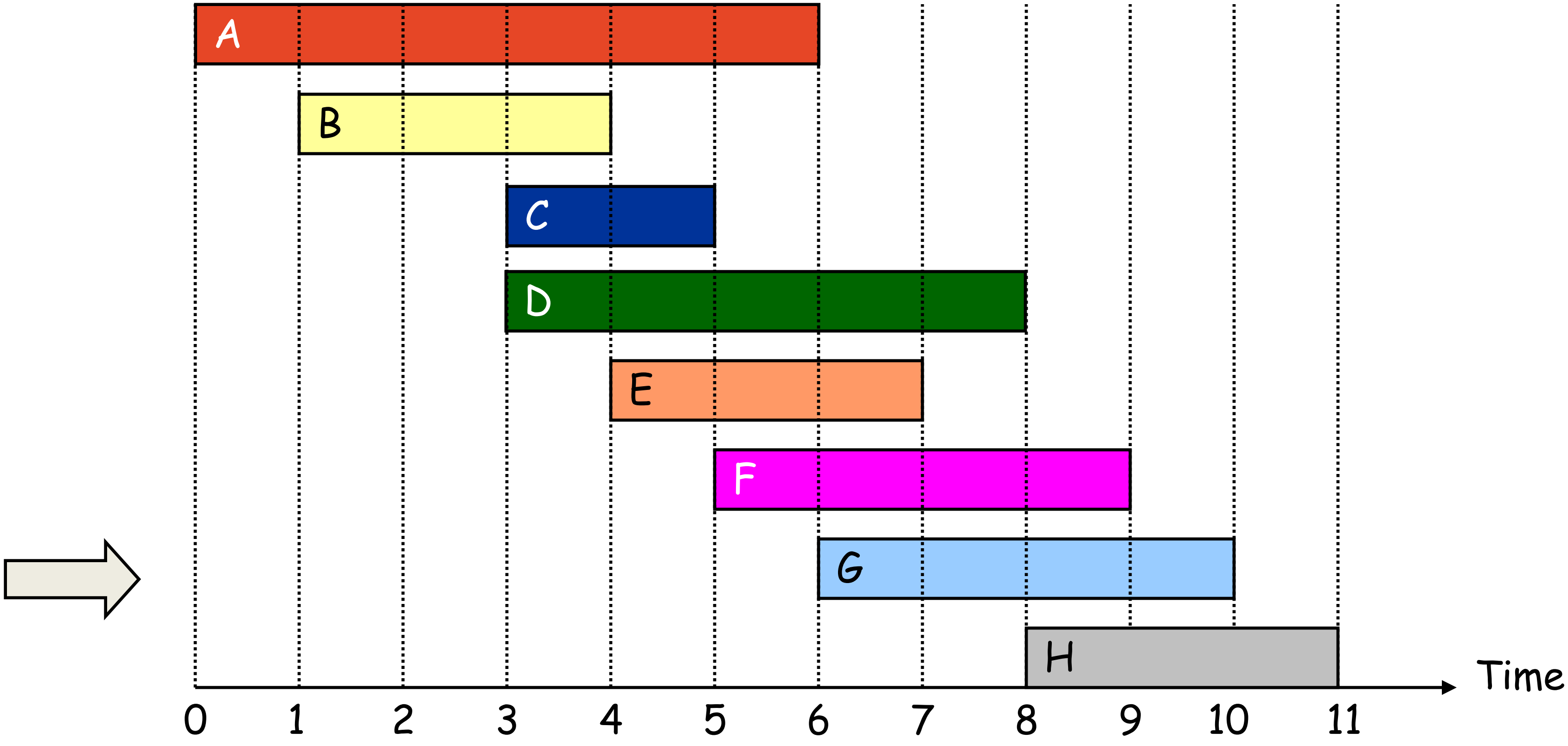


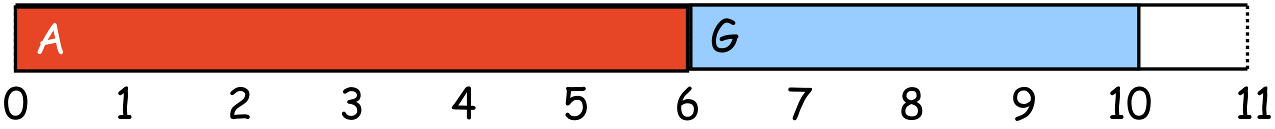
Interval Scheduling - [Earliest start time]



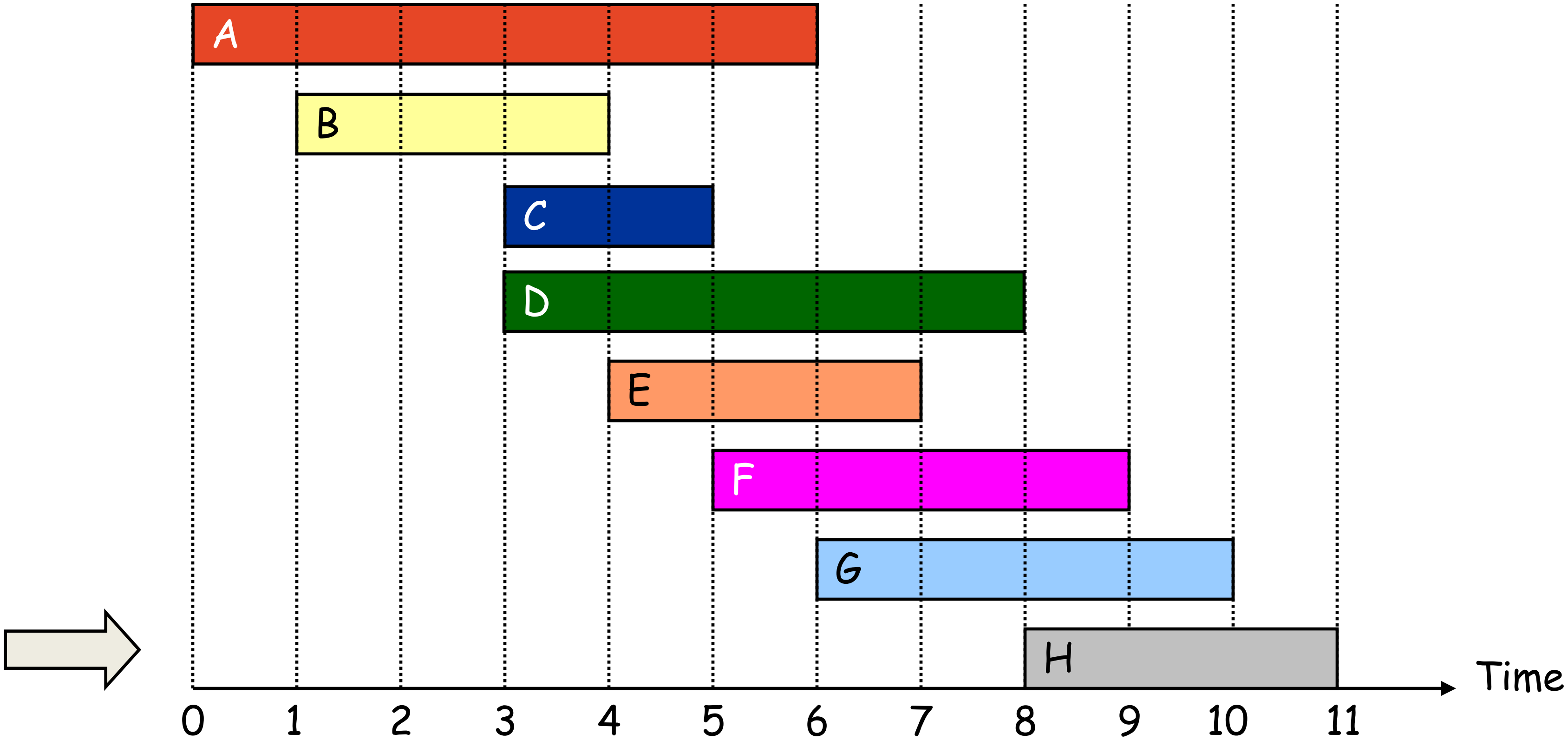


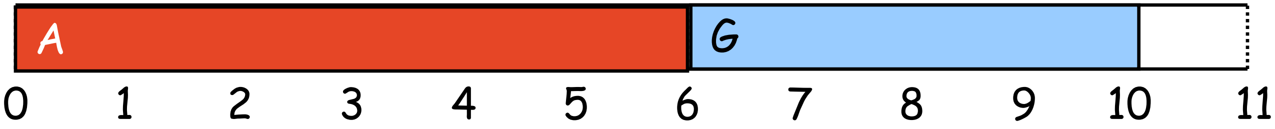
Interval Scheduling - [Earliest start time]





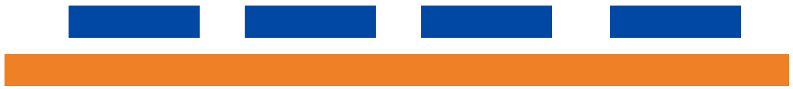
Interval Scheduling - [Earliest start time]





Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.



breaks [Earliest start time]

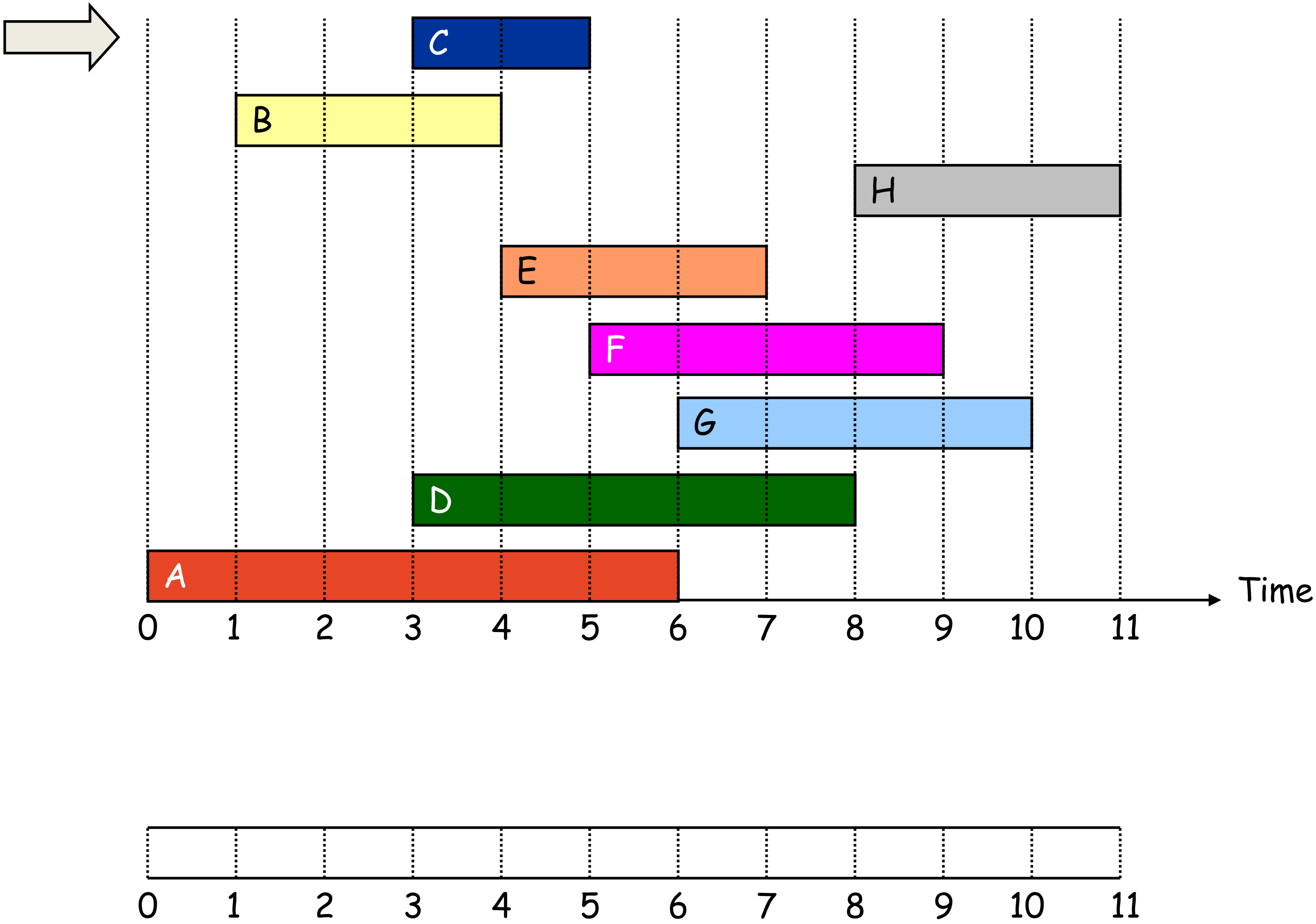
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.

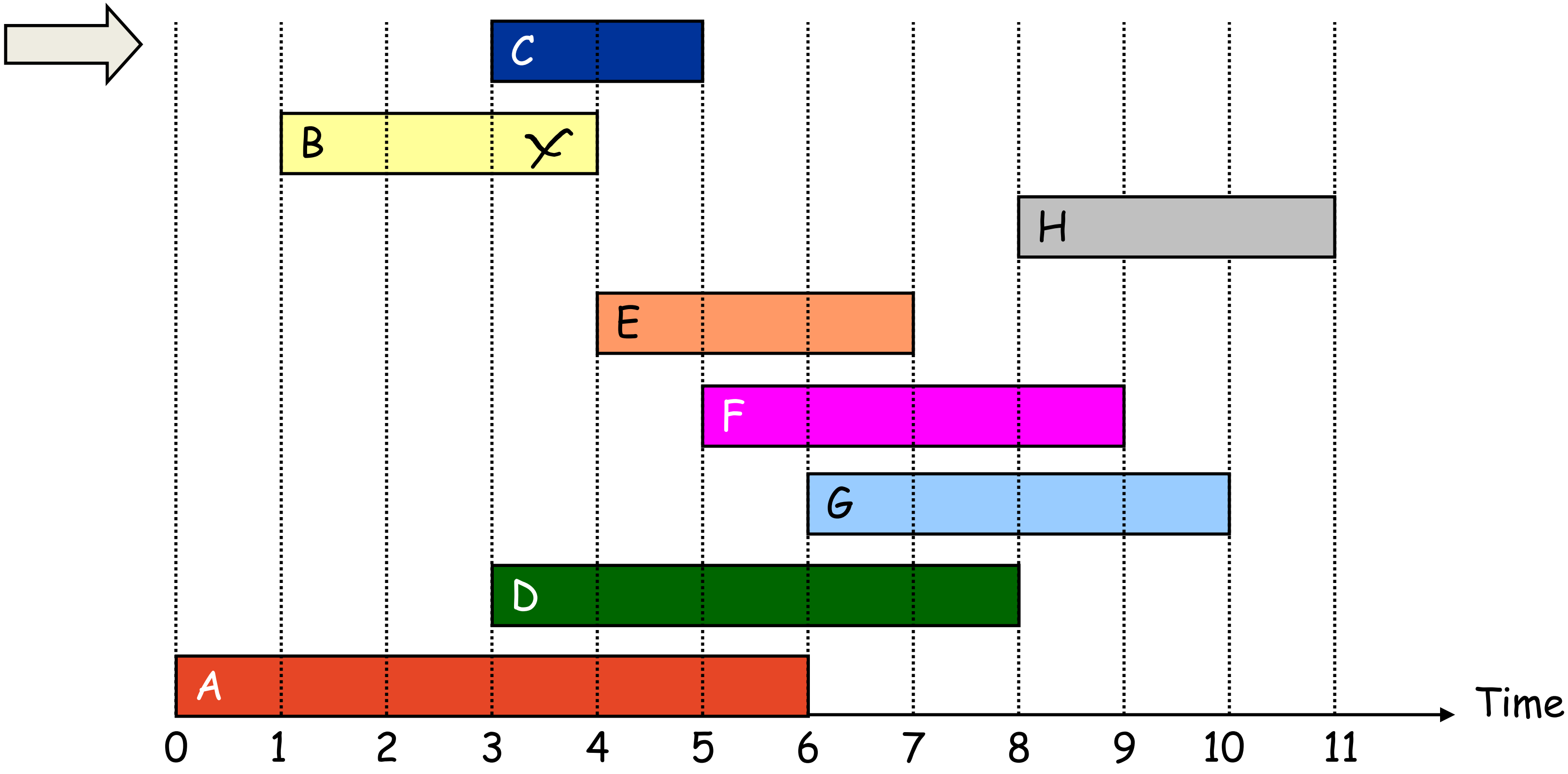
- [Earliest start time] Consider jobs in ascending order of start time s_j.

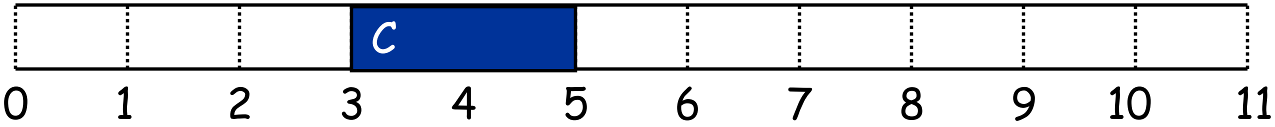
- [Shortest interval] Consider jobs in ascending order of interval length $f_j - s_j$.

Interval Scheduling - [Shortest interval]

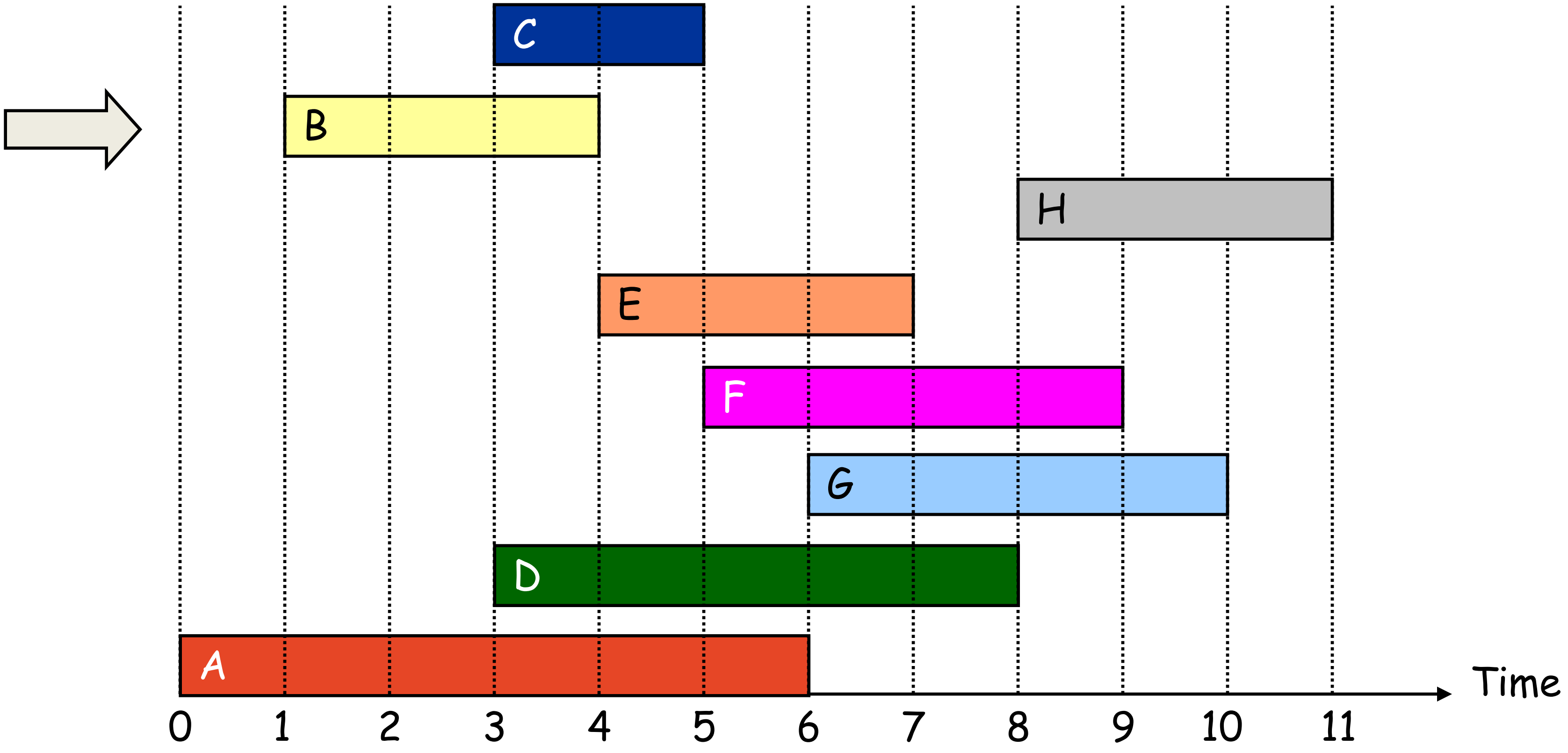


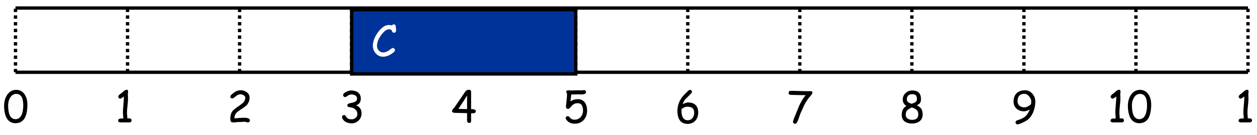
Interval Scheduling - [Shortest interval]



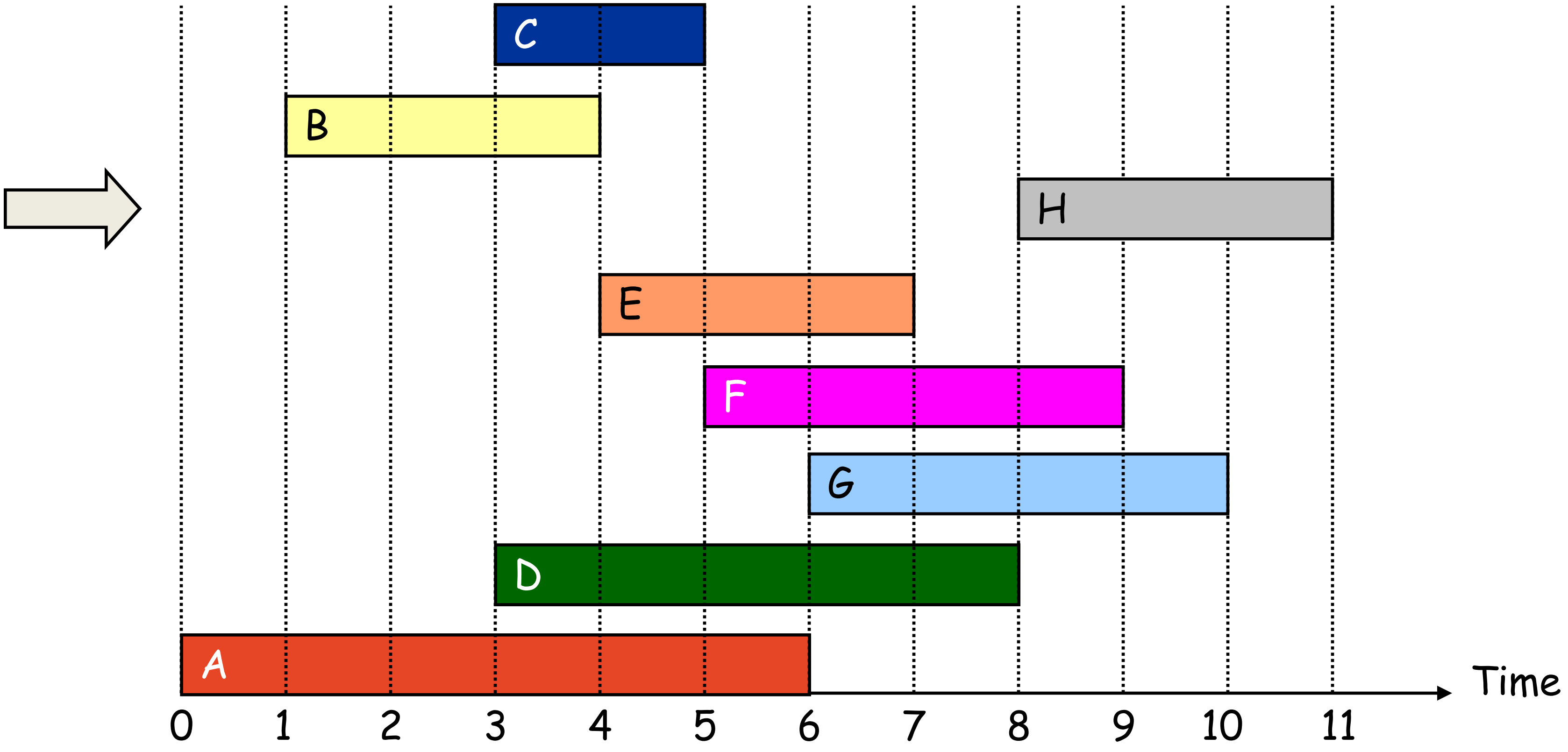


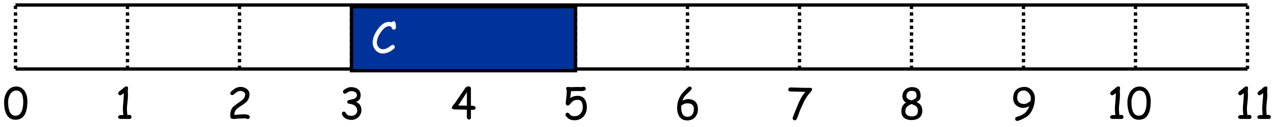
Interval Scheduling - [Shortest interval]



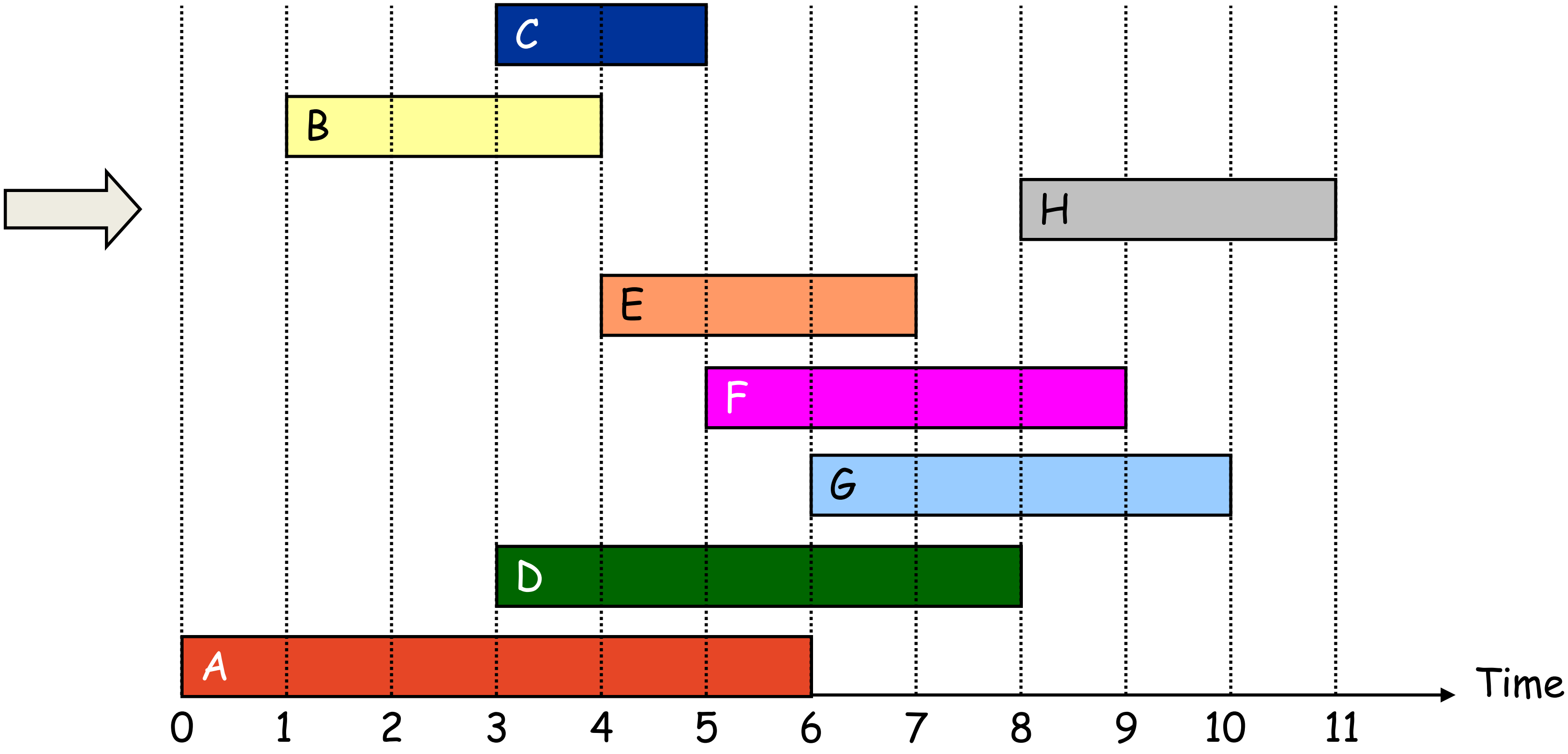


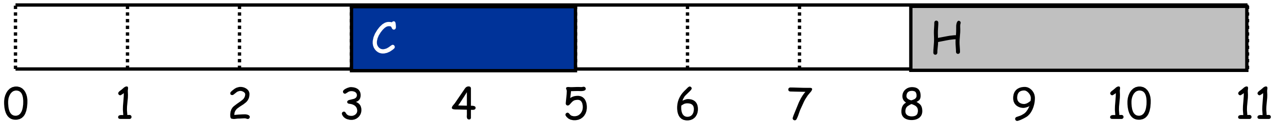
Interval Scheduling - [Shortest interval]



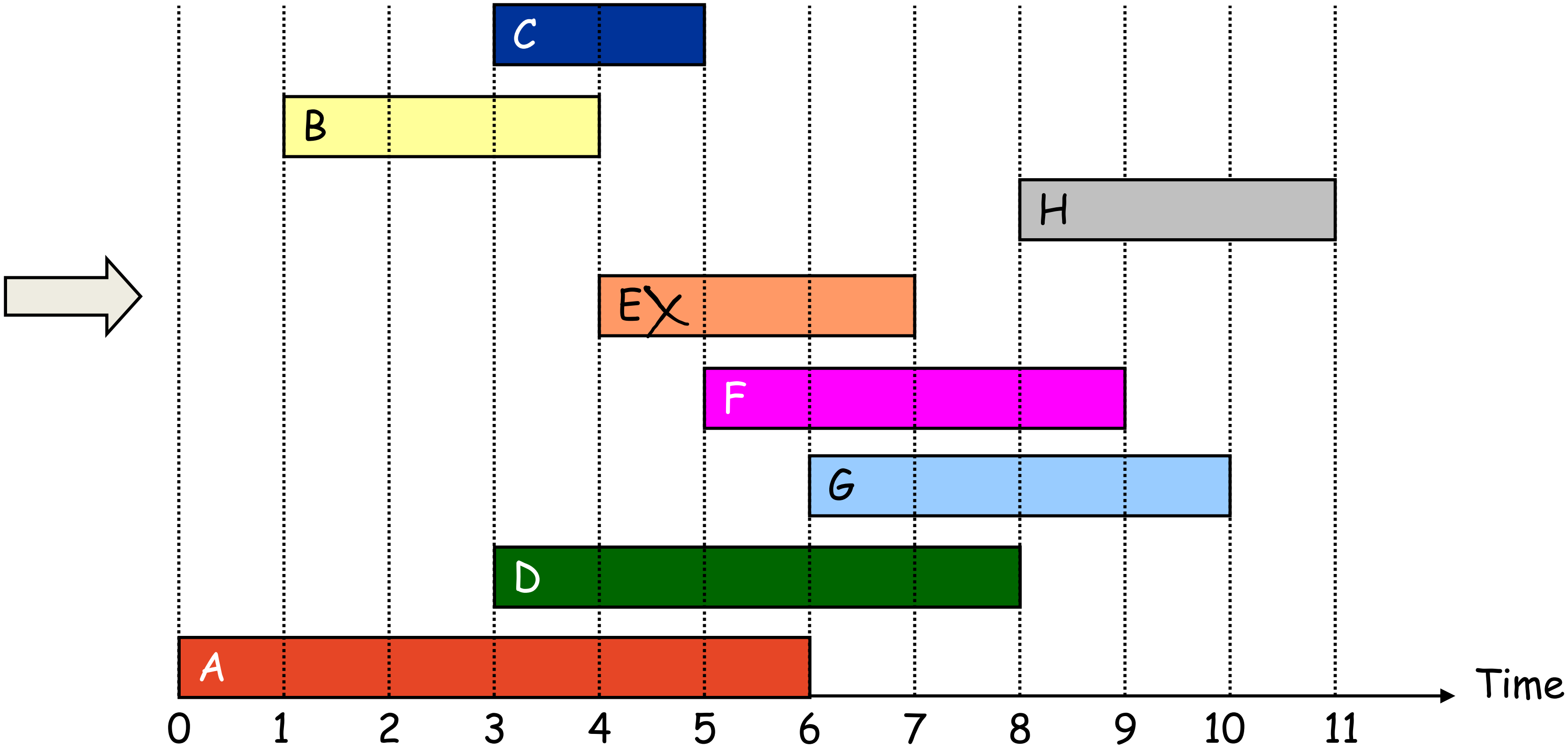


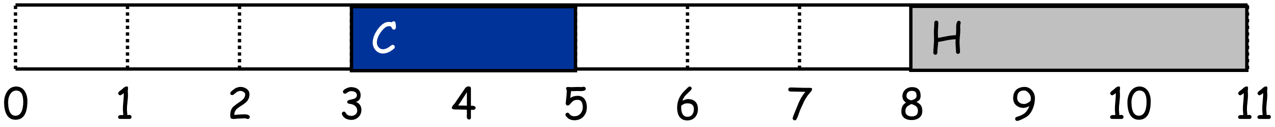
Interval Scheduling - [Shortest interval]



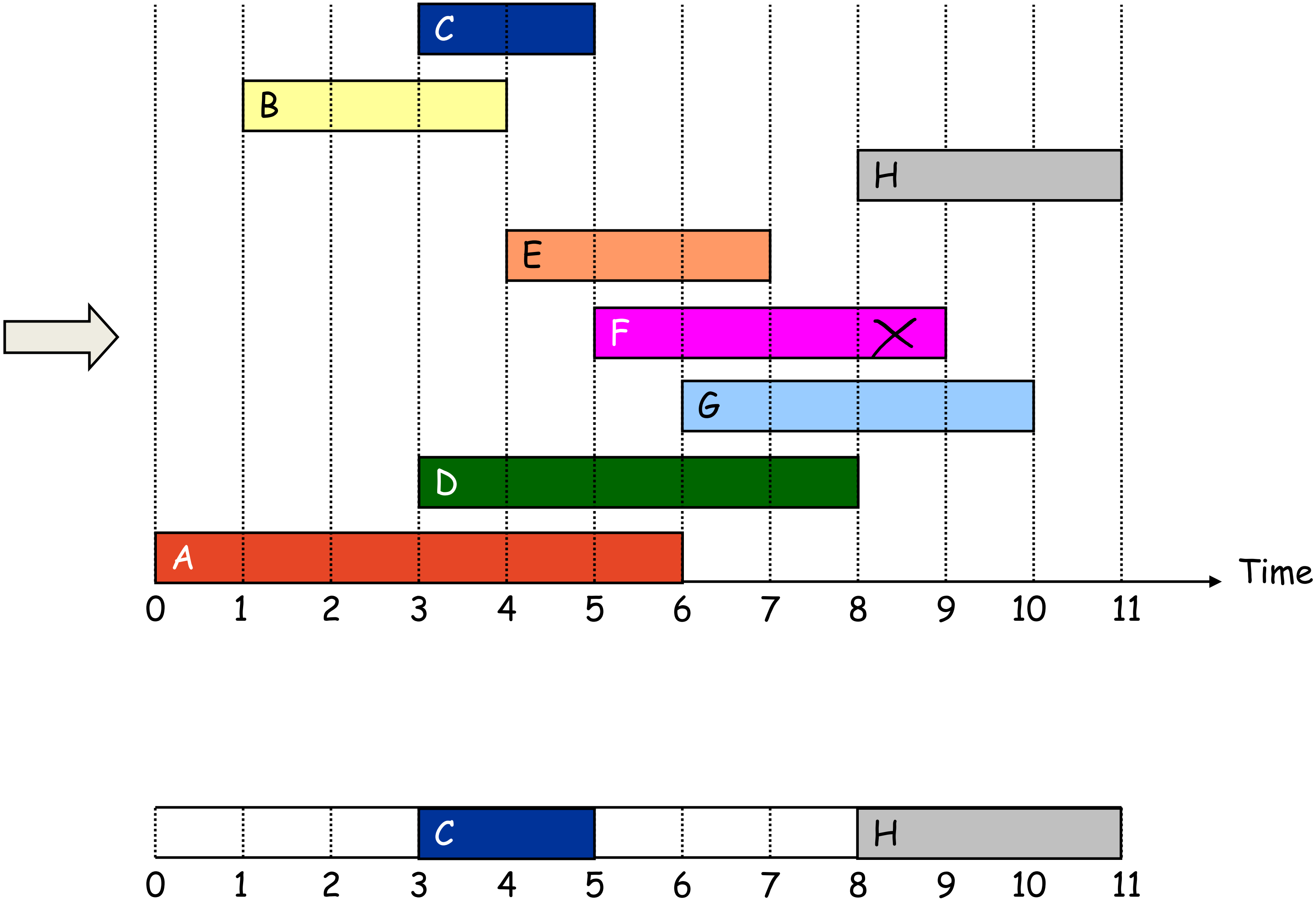


Interval Scheduling - [Shortest interval]

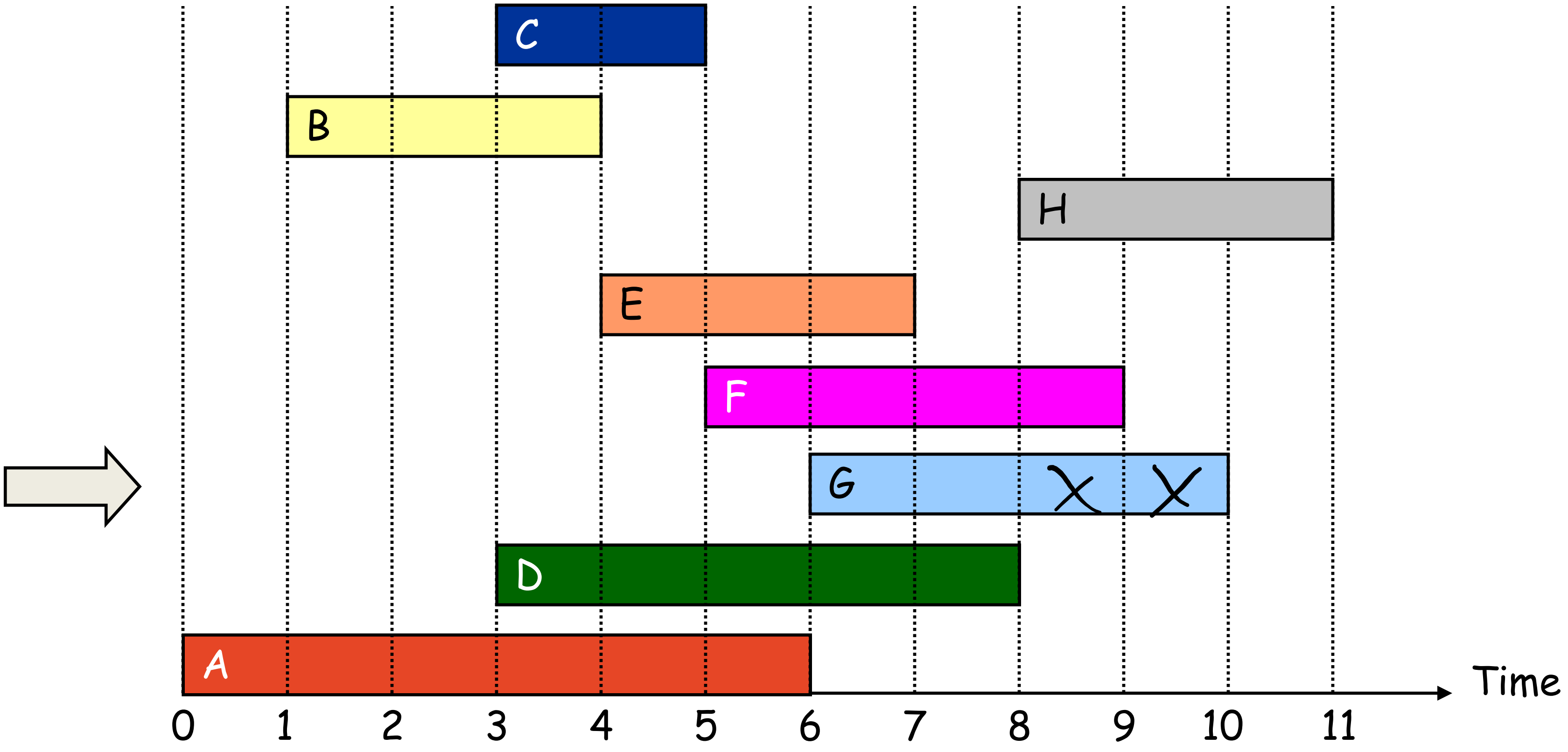


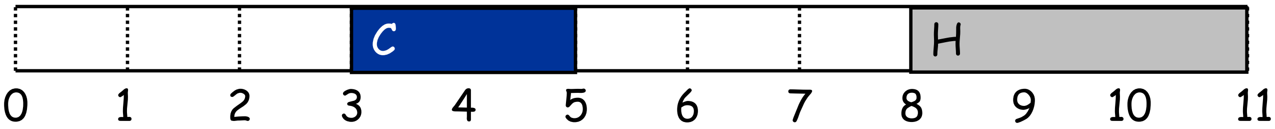


Interval Scheduling - [Shortest interval]

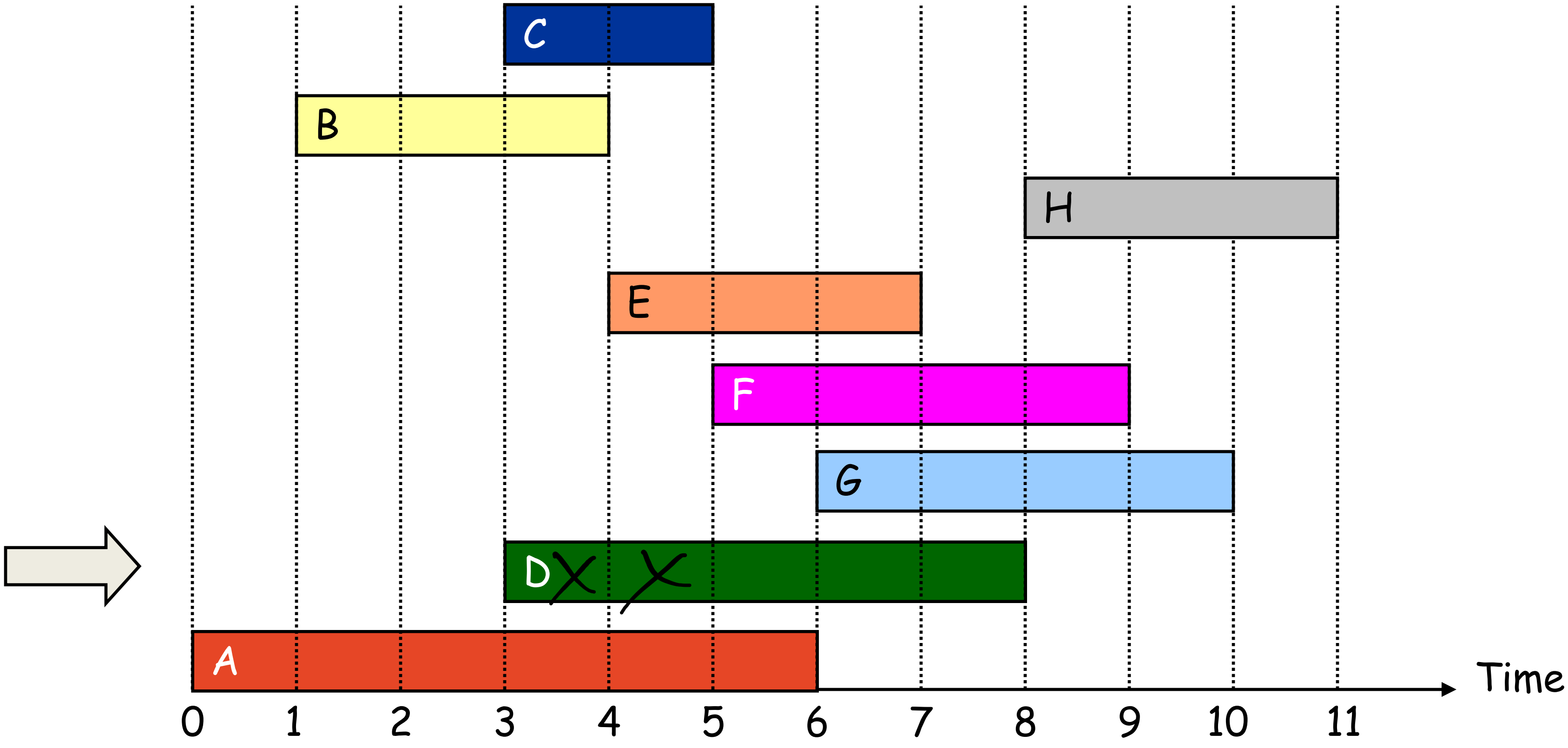


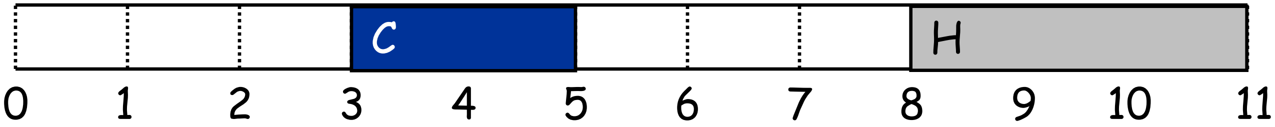
Interval Scheduling - [Shortest interval]



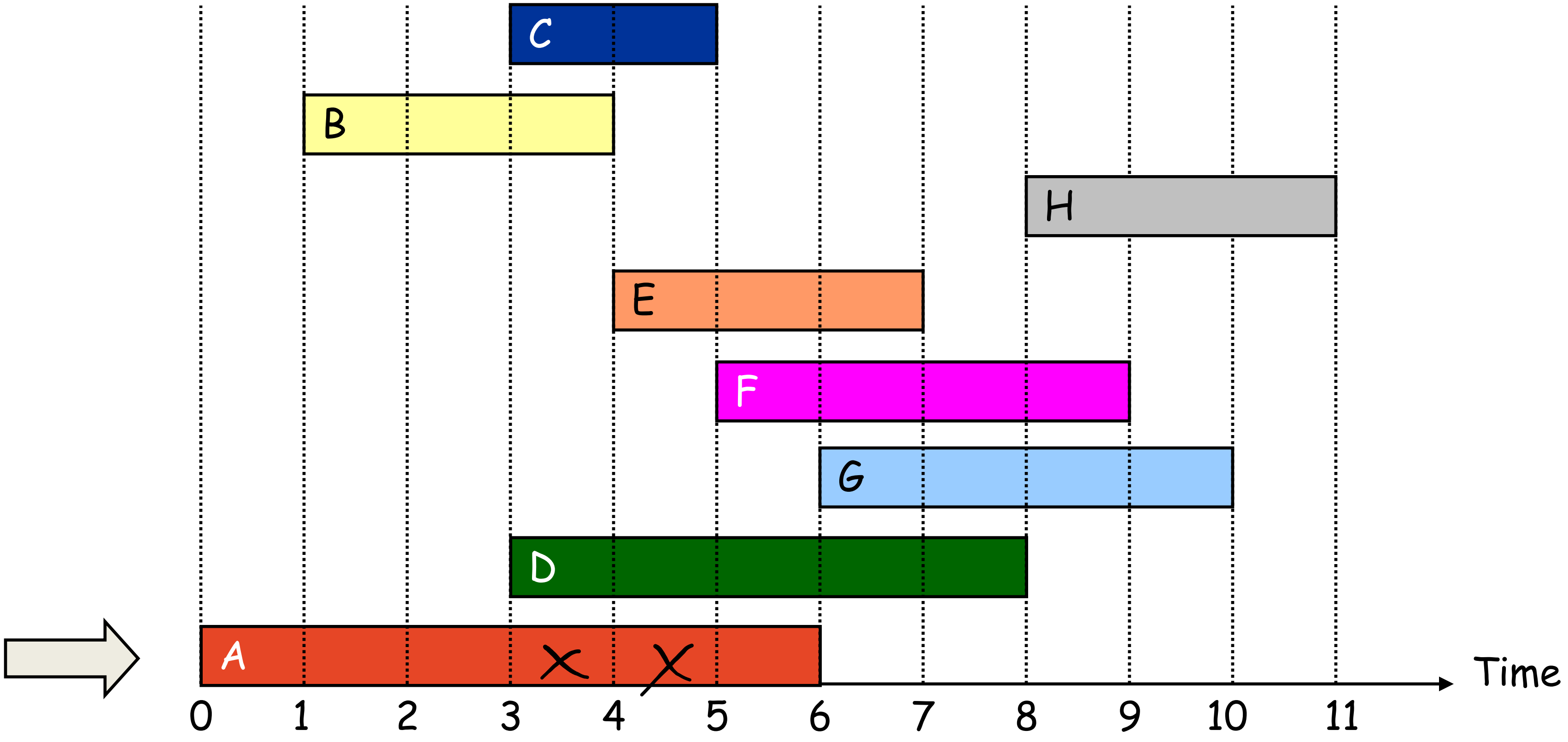


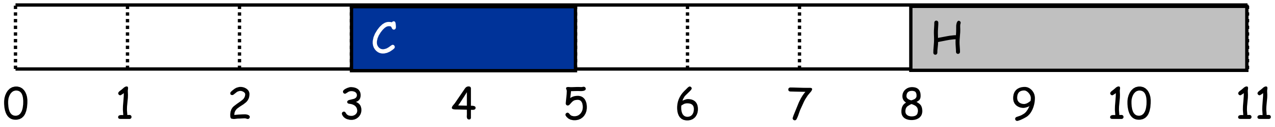
Interval Scheduling - [Shortest interval]





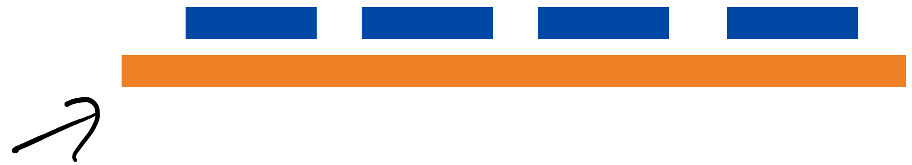
Interval Scheduling - [Shortest interval]



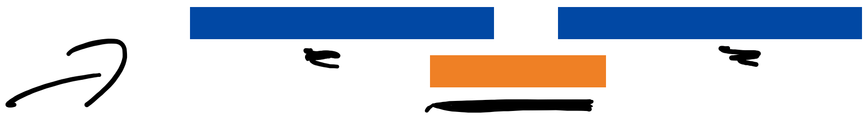


Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.



breaks [Earliest start time]



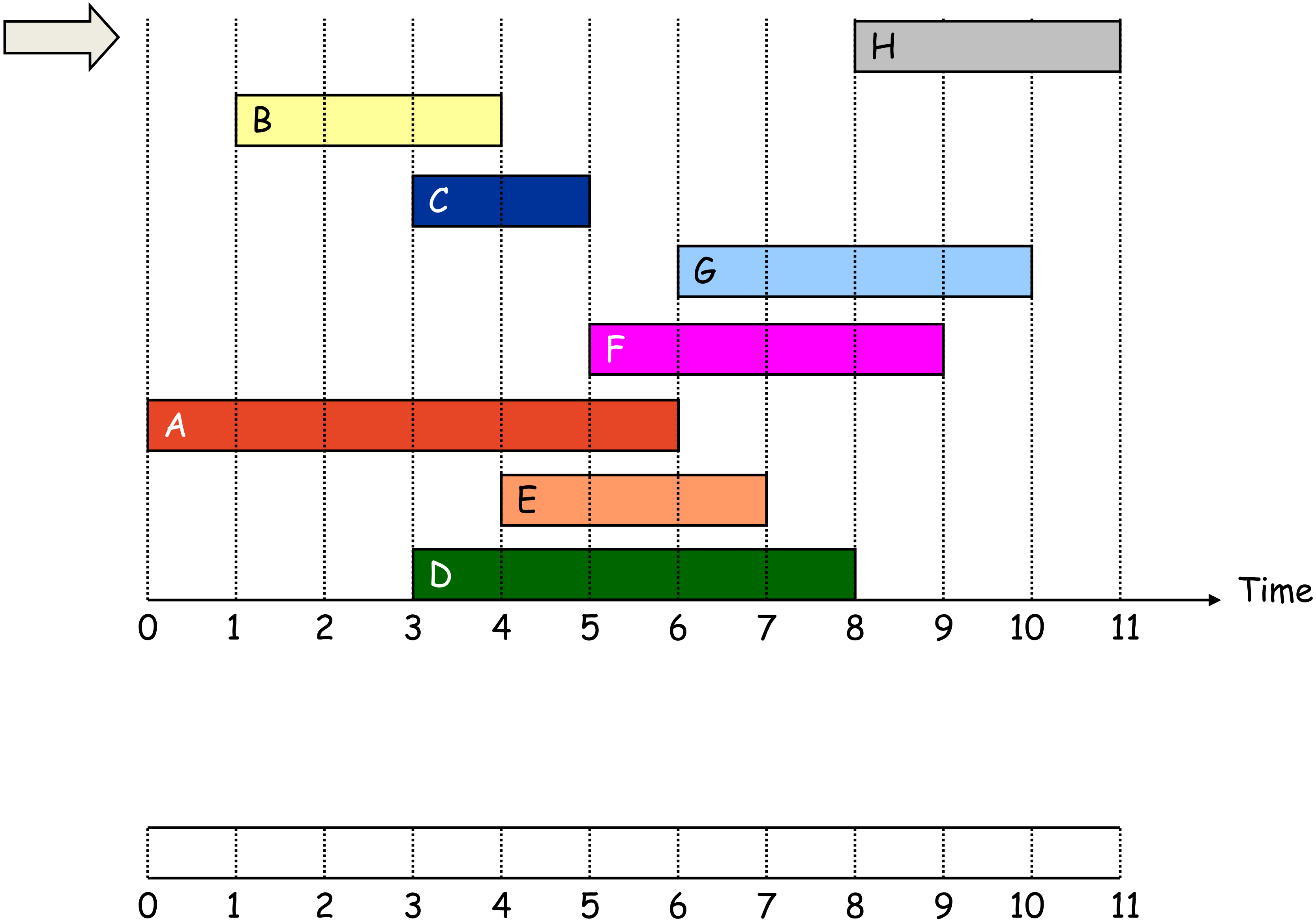
breaks [Shortest interval]

Interval Scheduling: Greedy Algorithms

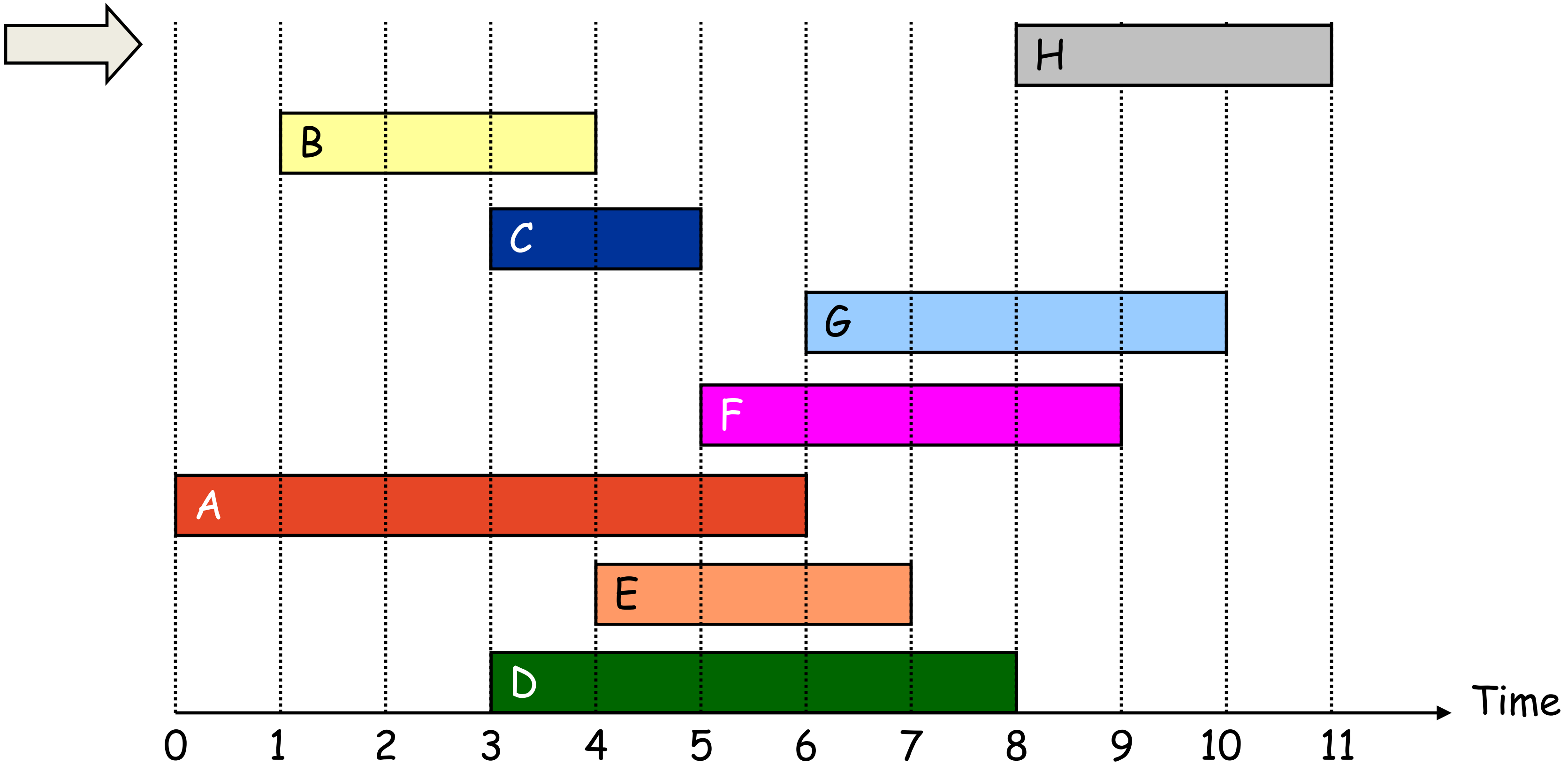
Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.

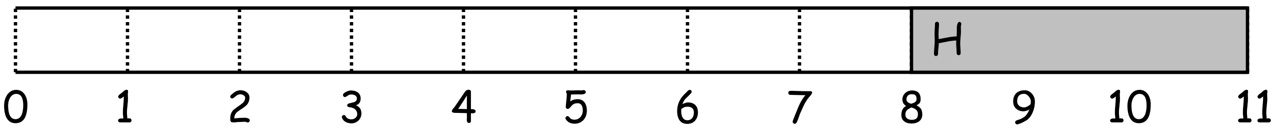
- [Earliest start time] Consider jobs in ascending order of start time s_j.
- [Shortest interval] Consider jobs in ascending order of interval length $f_j - s_j$.
- [Fewest conflicts] For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

Interval Scheduling - [Fewest Conflicts]

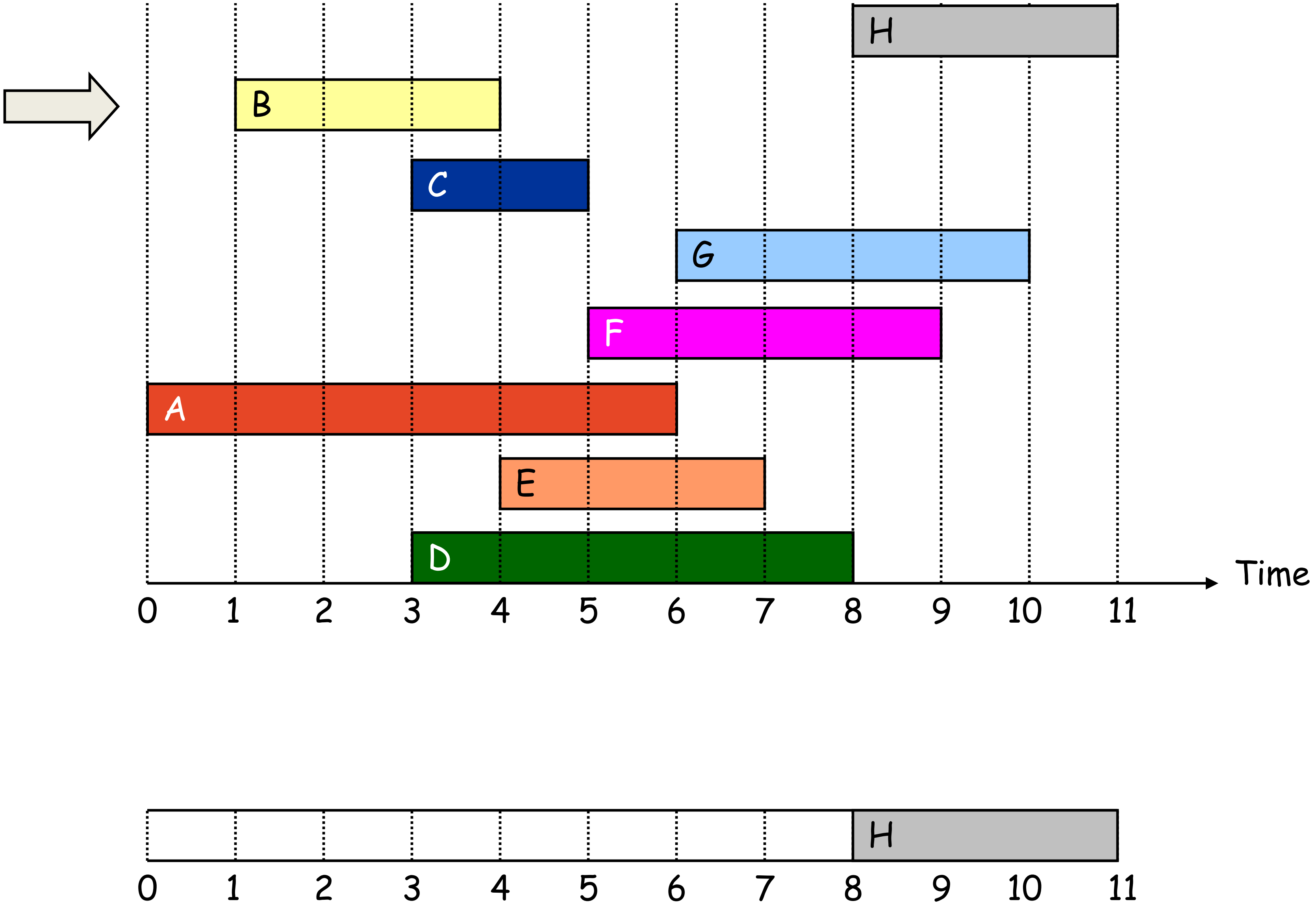


Interval Scheduling - [Fewest Conflicts]

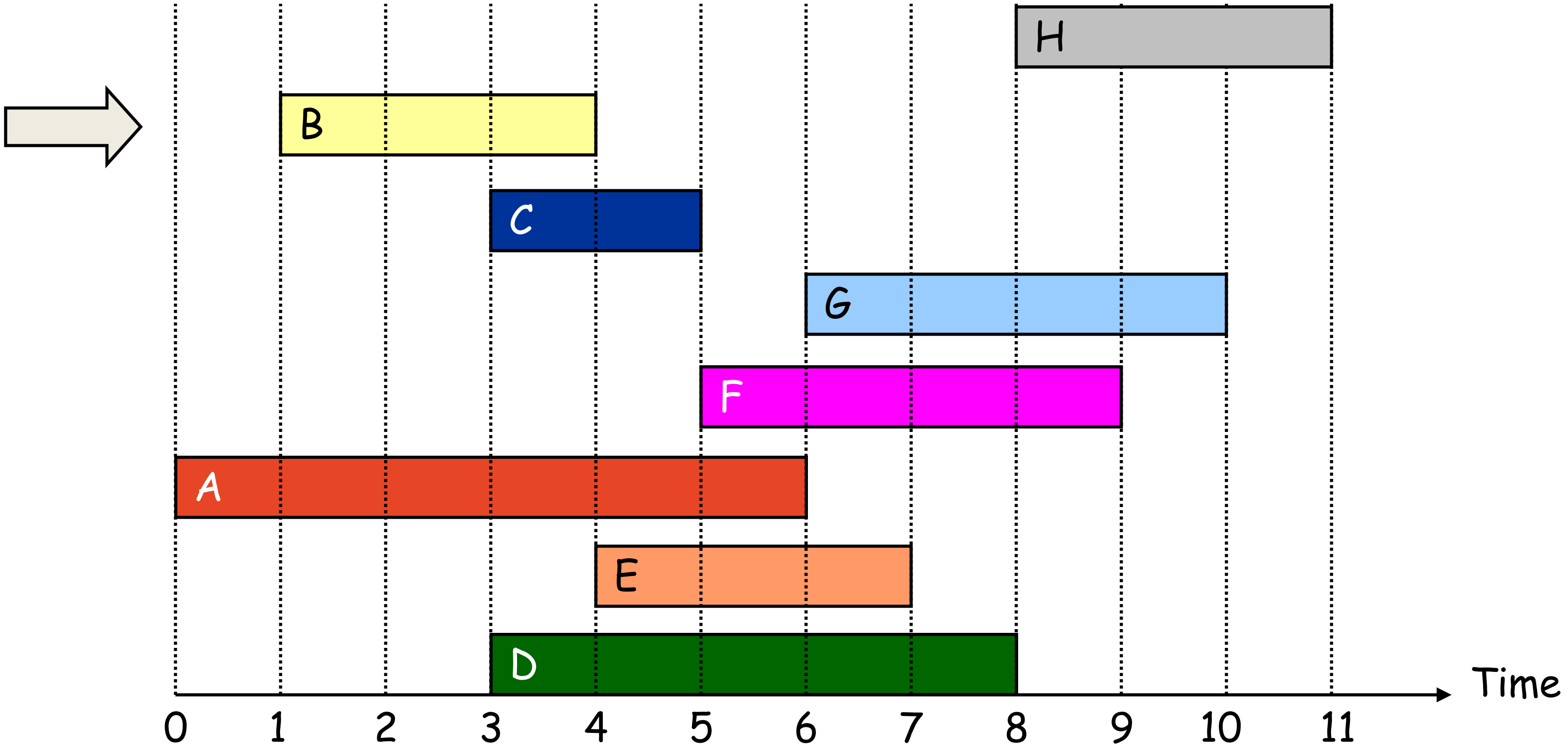


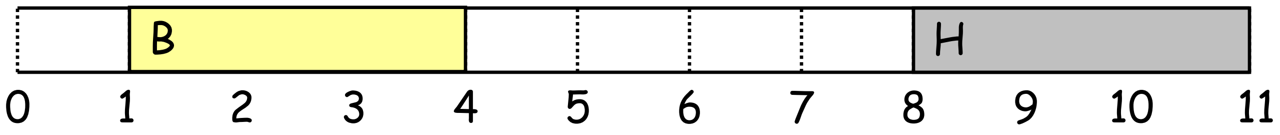


Interval Scheduling - [Fewest Conflicts]

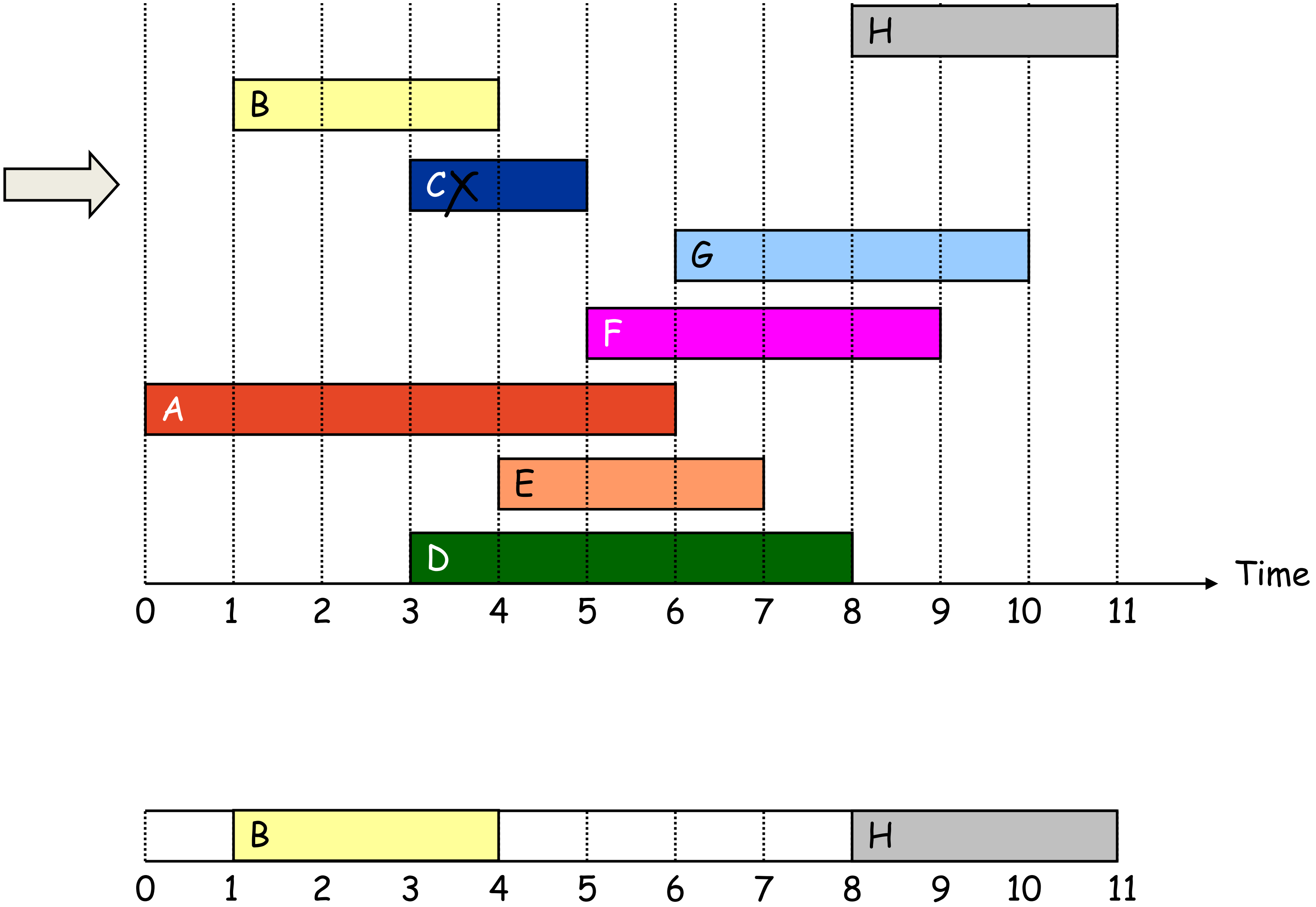


Interval Scheduling - [Fewest Conflicts]

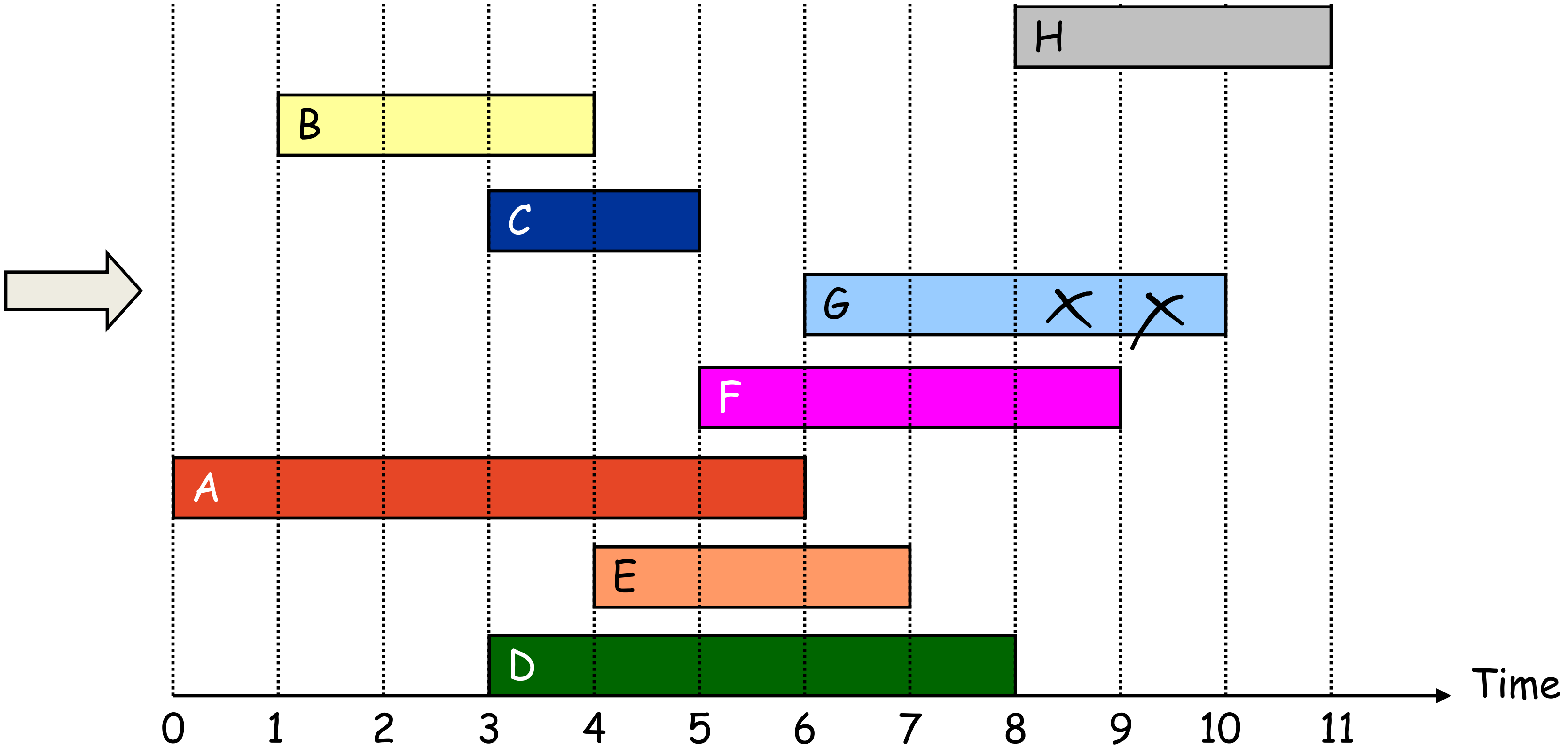


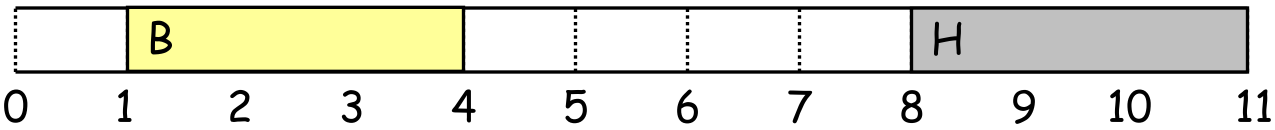


Interval Scheduling - [Fewest Conflicts]

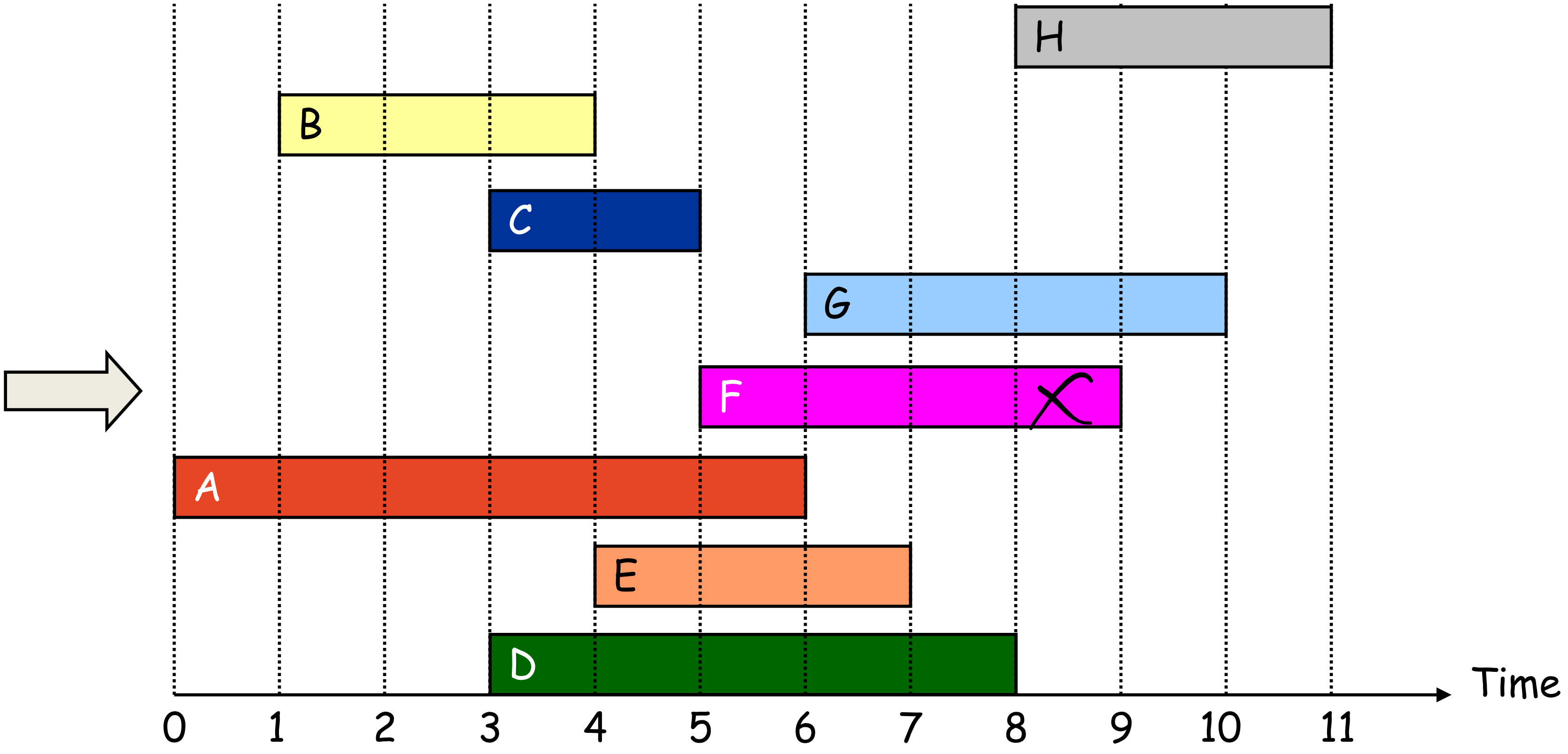


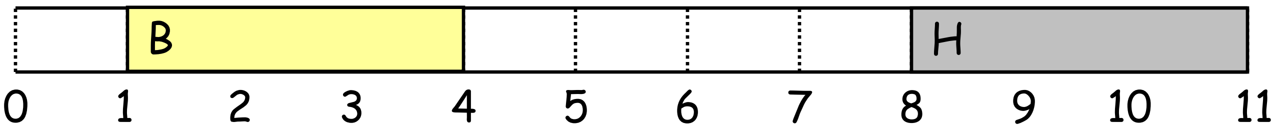
Interval Scheduling - [Fewest Conflicts]



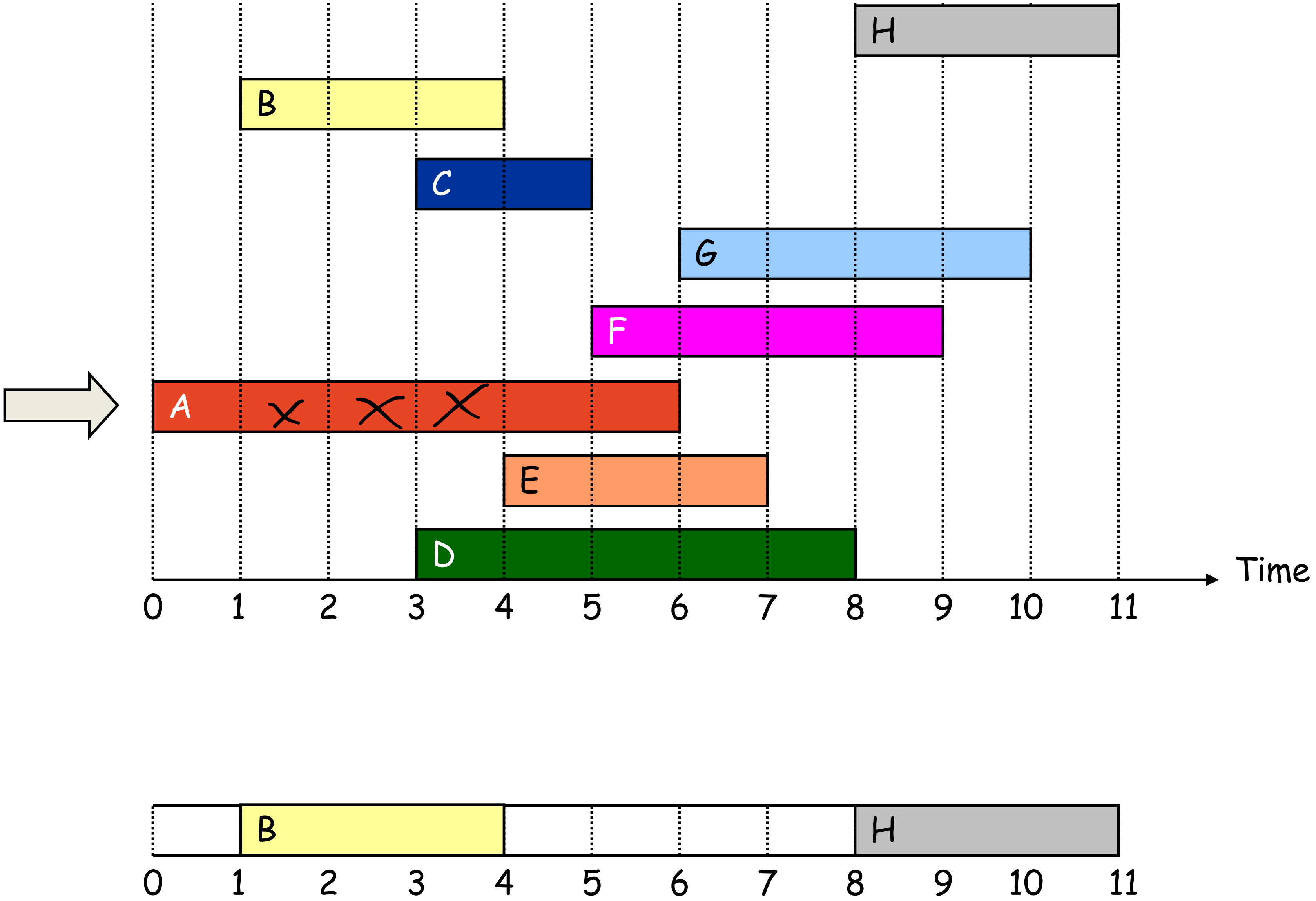


Interval Scheduling - [Fewest Conflicts]

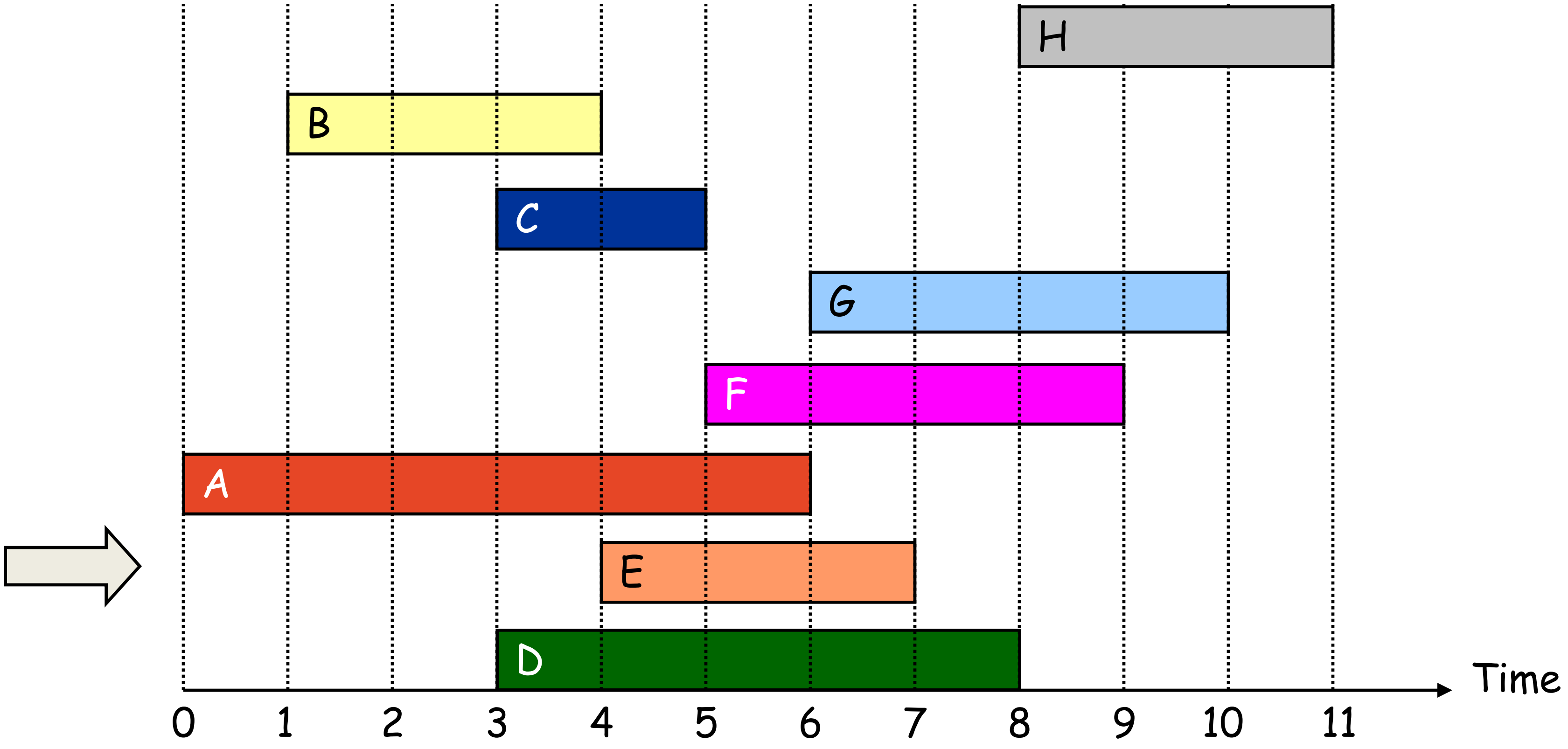


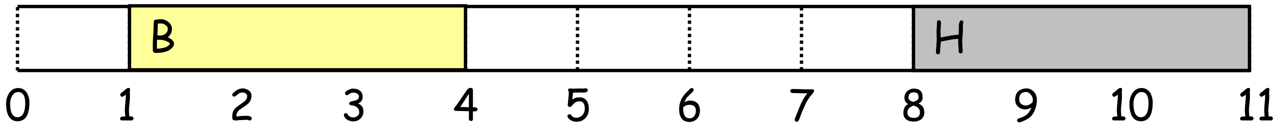


Interval Scheduling - [Fewest Conflicts]

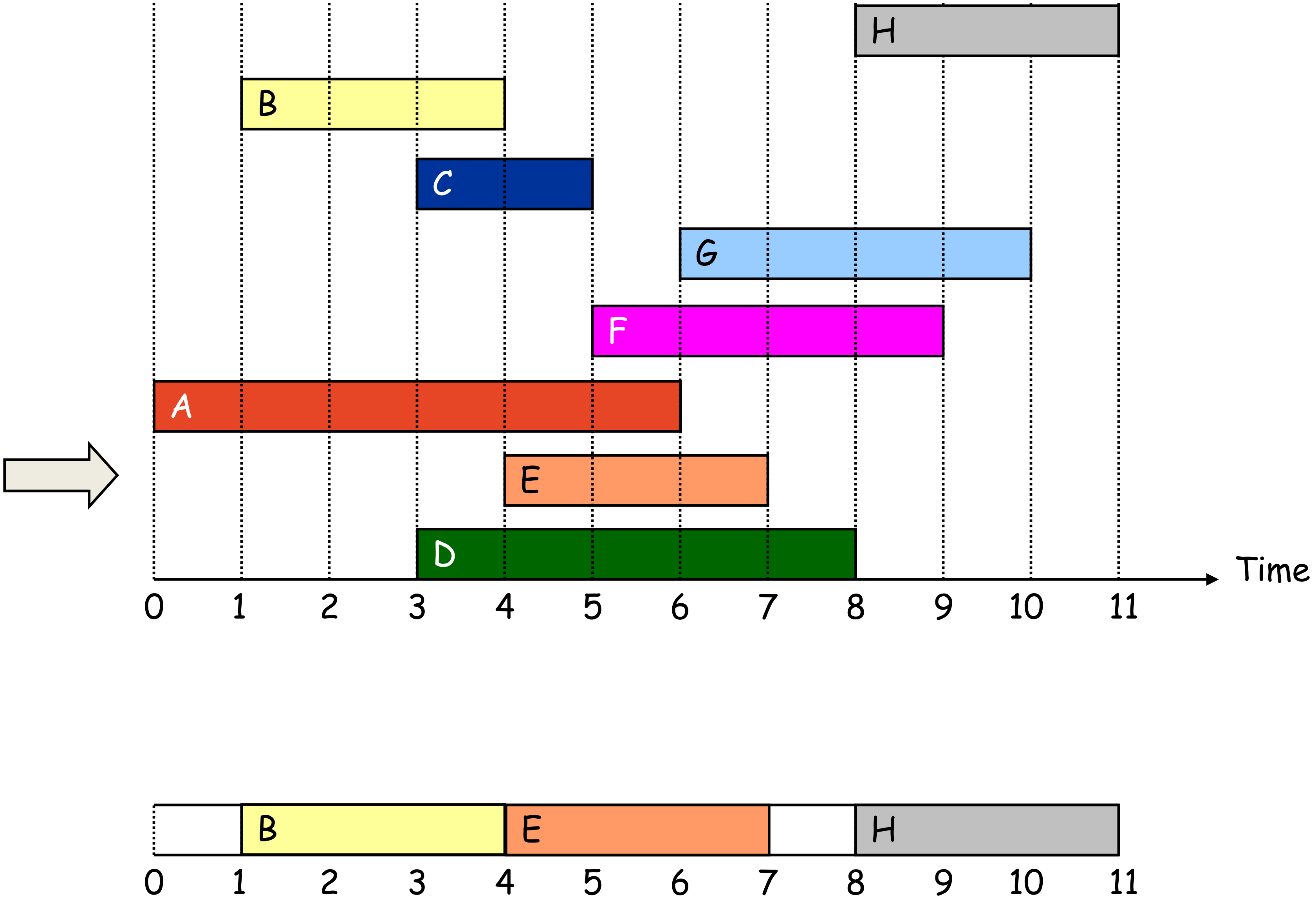


Interval Scheduling - [Fewest Conflicts]

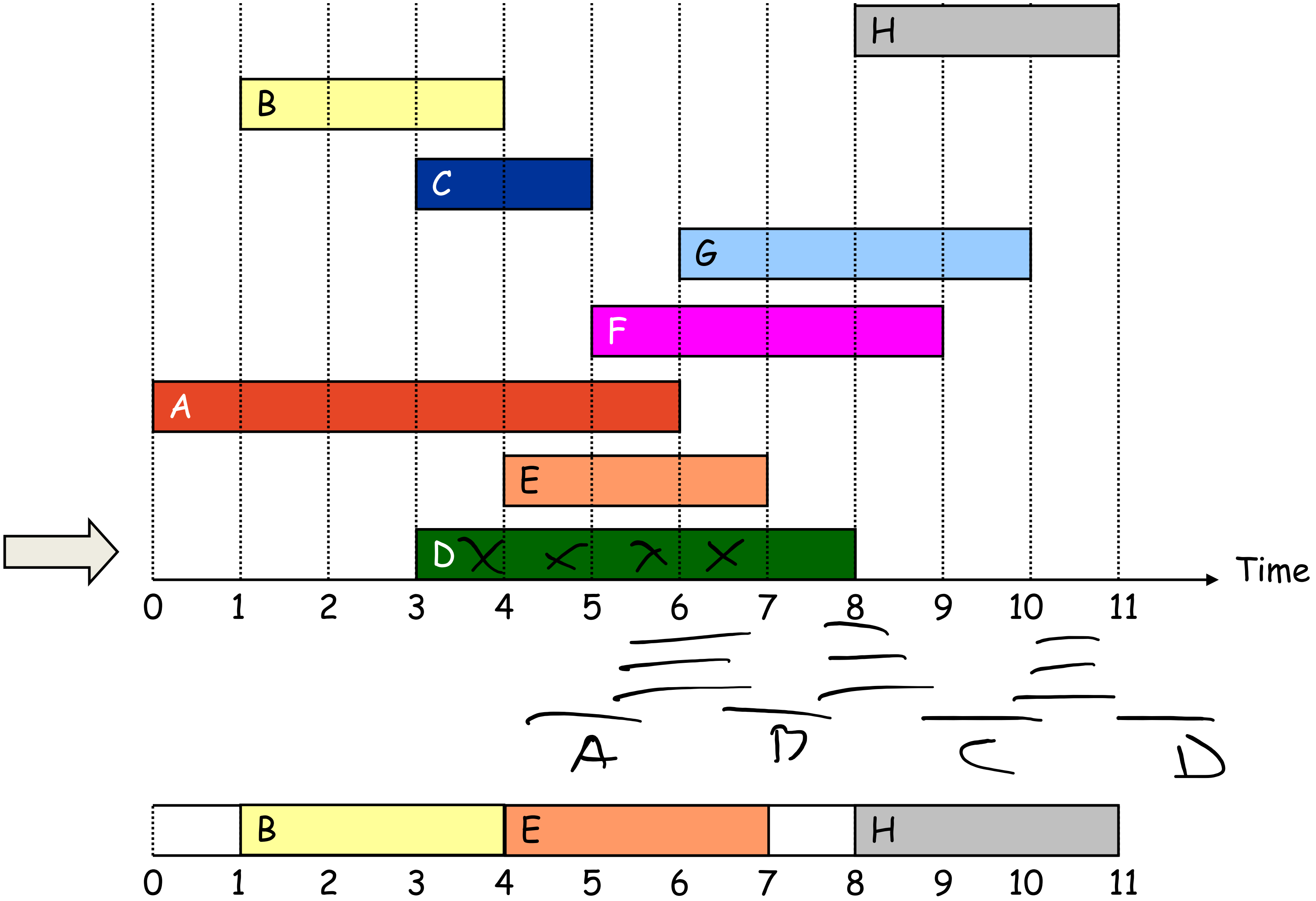




Interval Scheduling - [Fewest Conflicts]

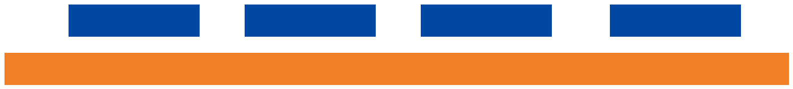


Interval Scheduling - [Fewest Conflicts]

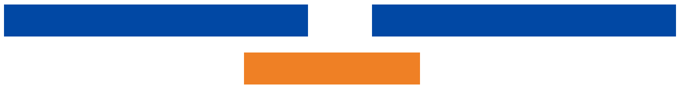


Interval Scheduling: Greedy Algorithms

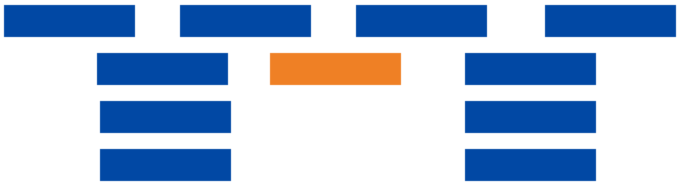
Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.



breaks [Earliest start time]



breaks [Shortest interval]



breaks [Fewest conflicts]

3 jobs

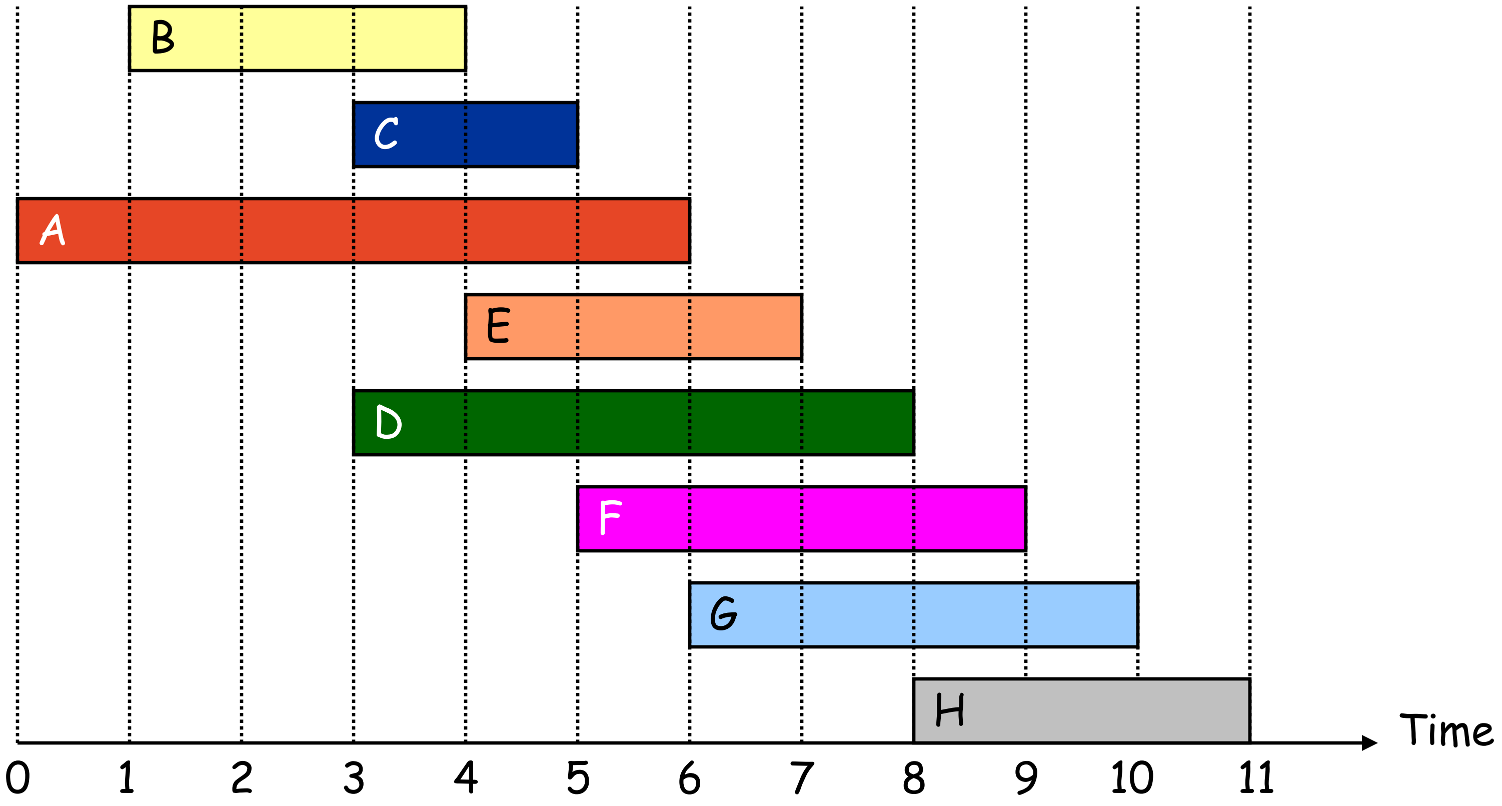
OPT = 4 jobs

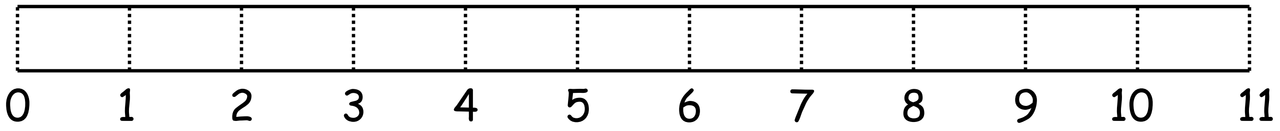
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.

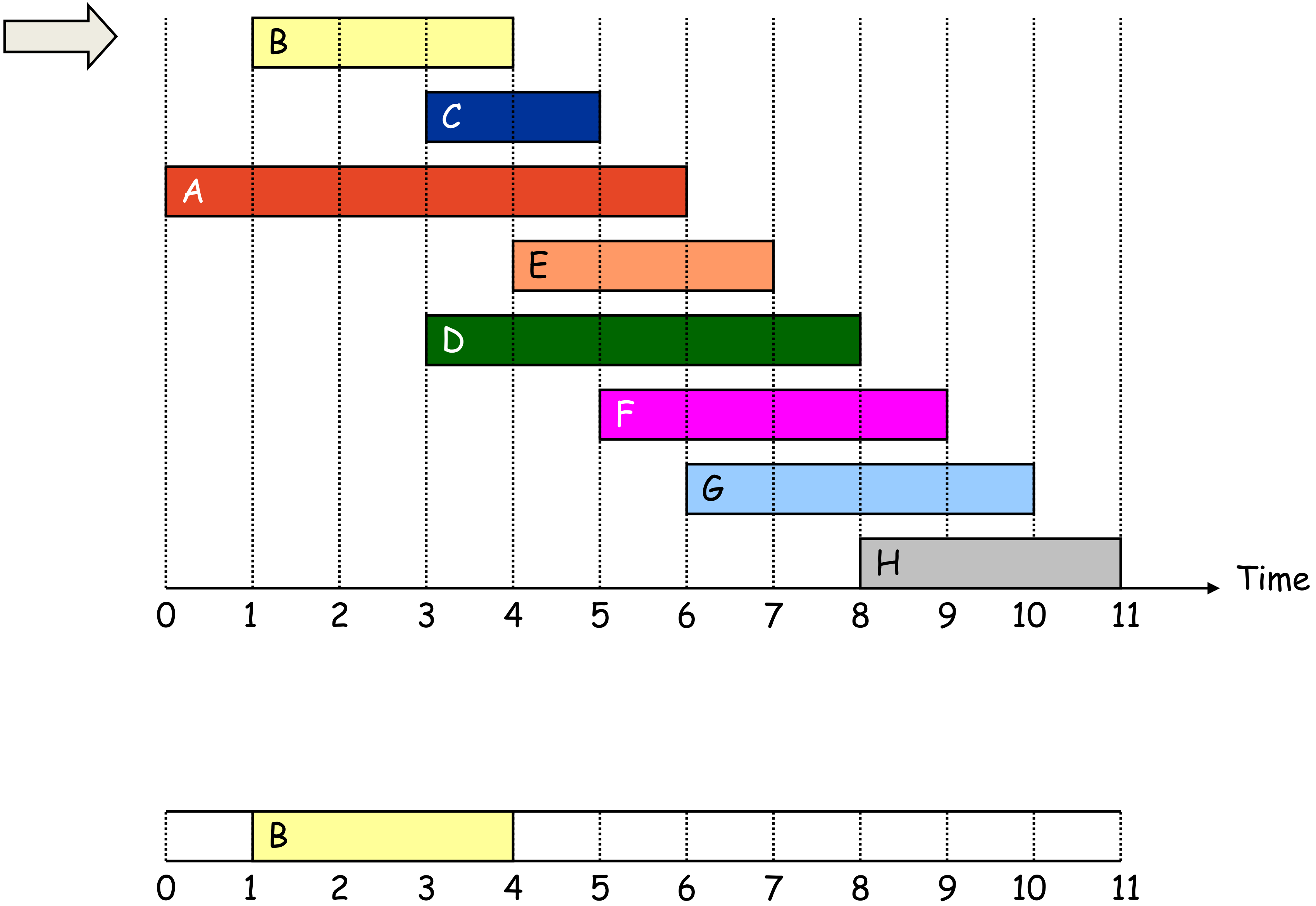
- [Earliest start time] Consider jobs in ascending order of start time s_i.
- [Shortest interval] Consider jobs in ascending order of interval length f_i - s_i.
- [Fewest conflicts] For each job, count the number of conflicting jobs c_i. Schedule in ascending order of conflicts c_i.
- [Earliest finish time] Consider jobs in ascending order of finish time f_i.

Interval Scheduling

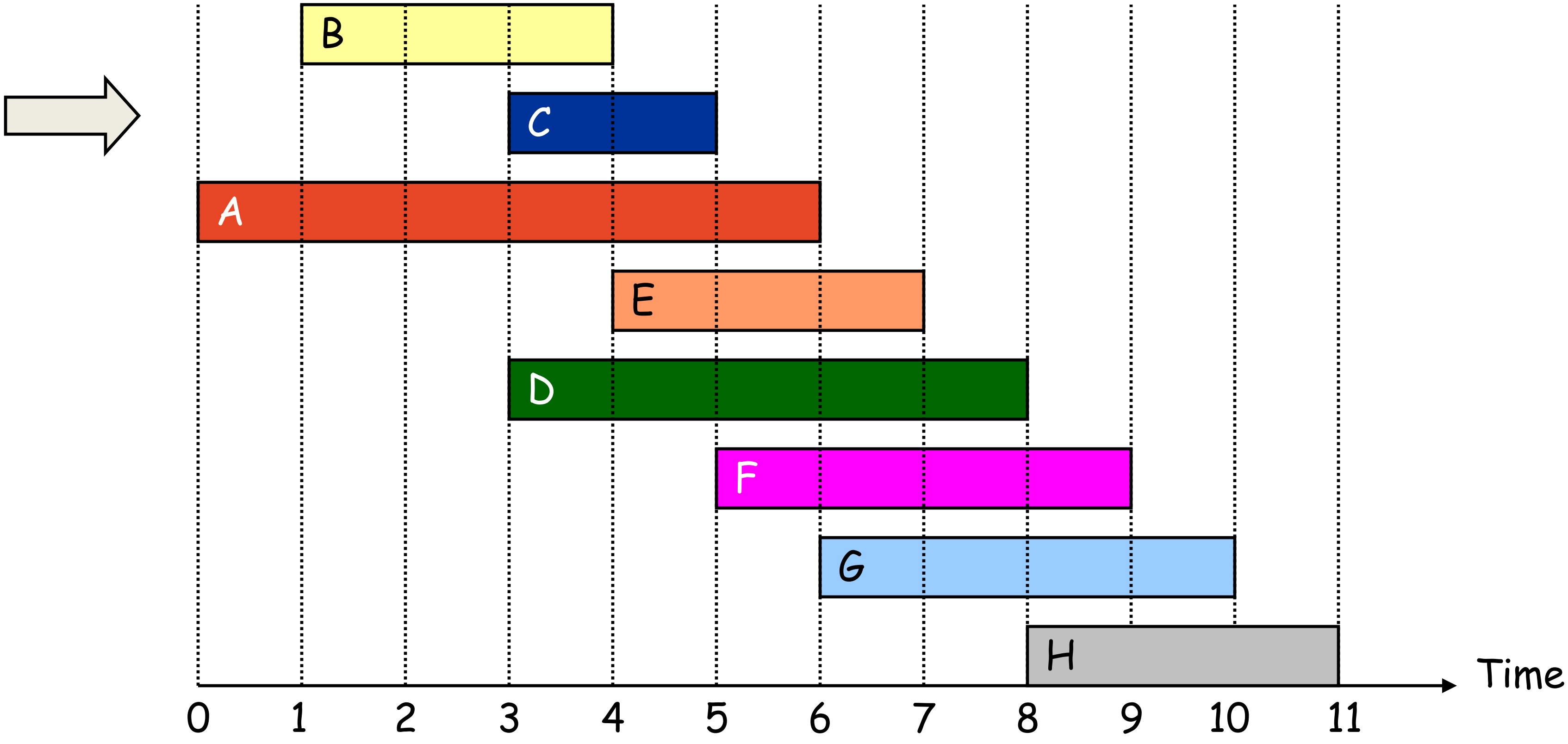


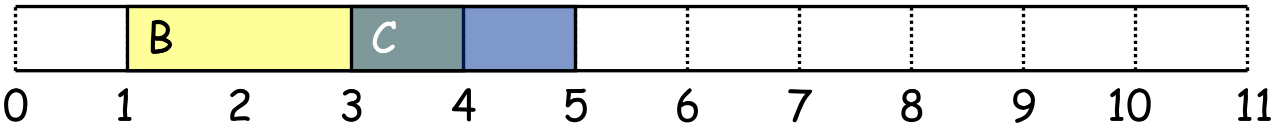


Interval Scheduling

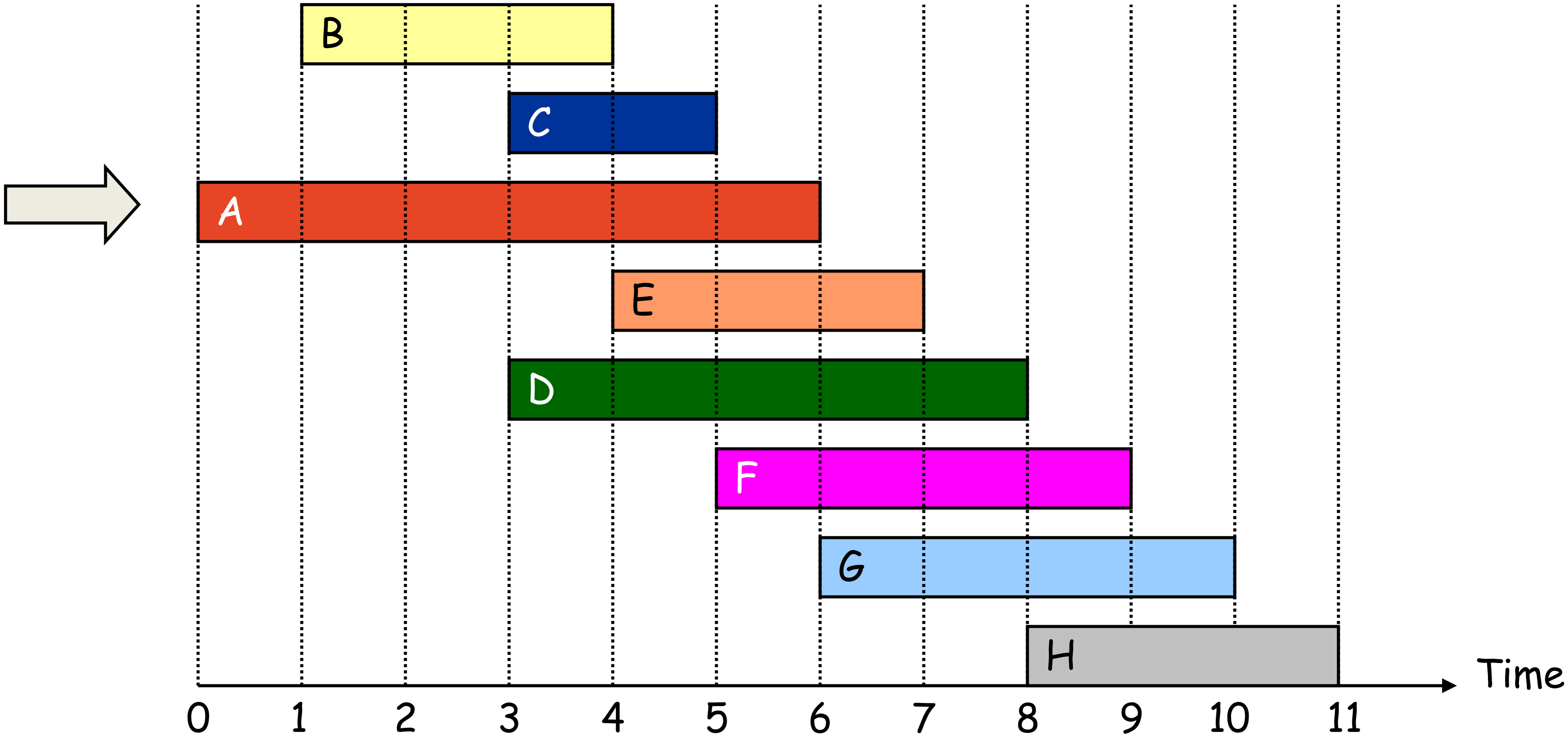


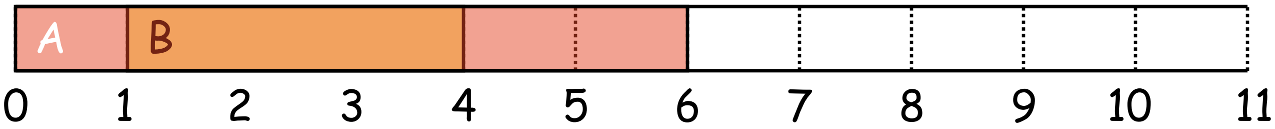
Interval Scheduling



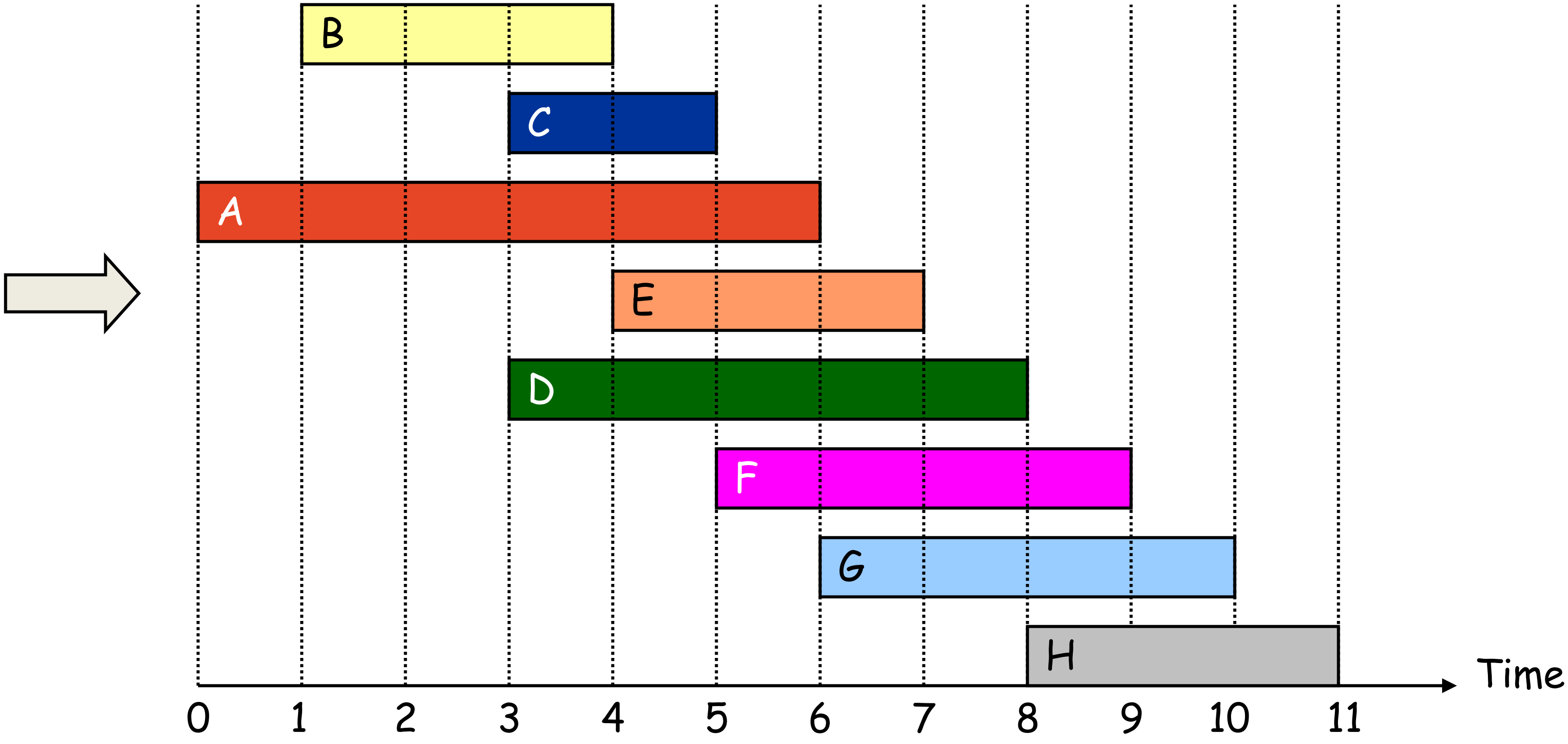


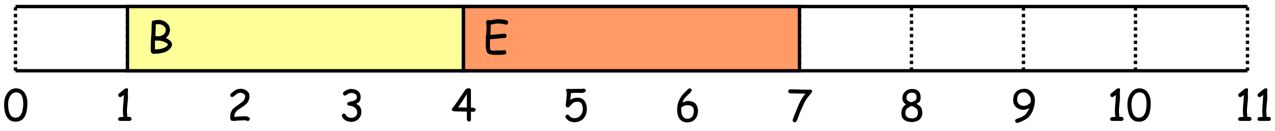
Interval Scheduling



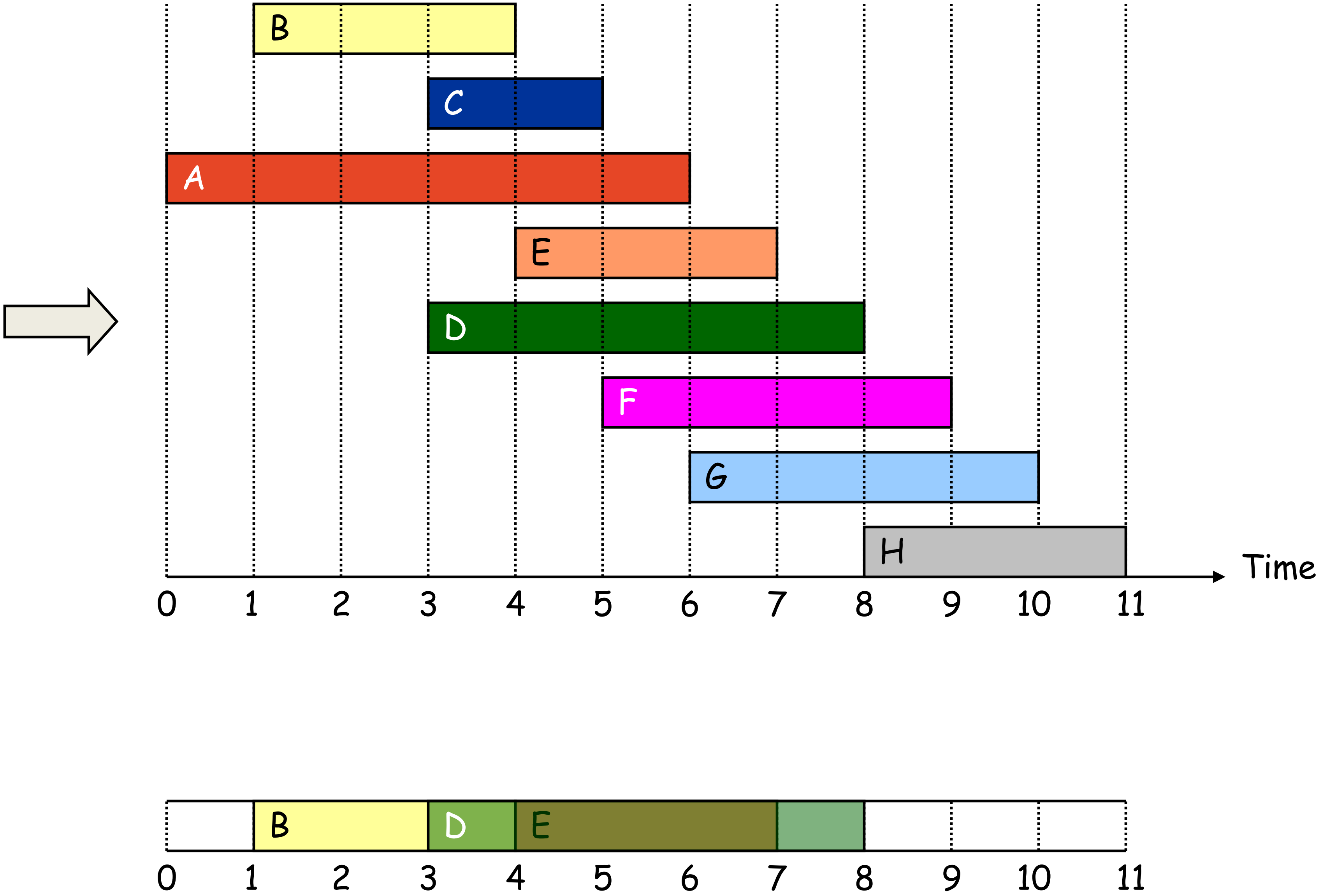


Interval Scheduling

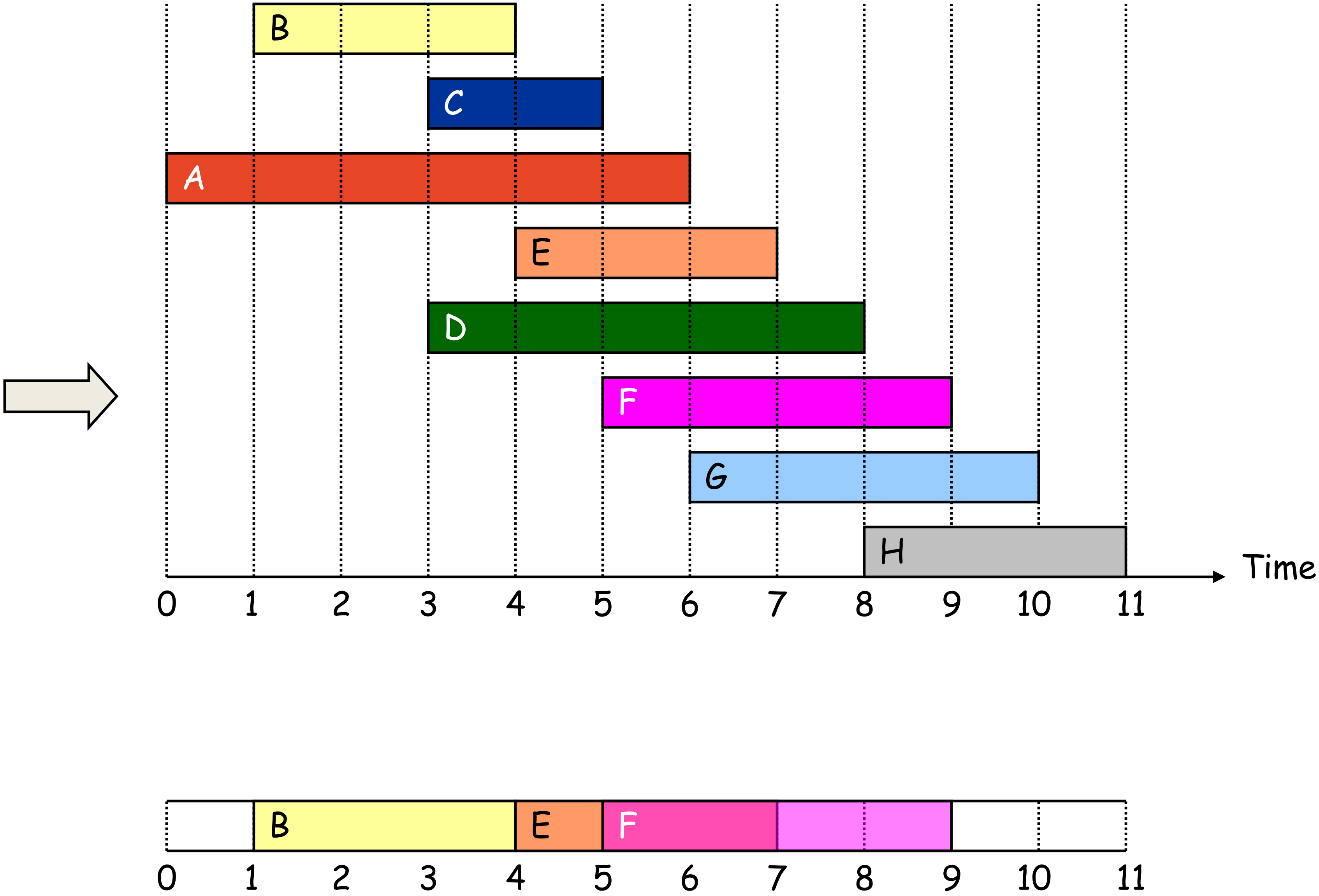




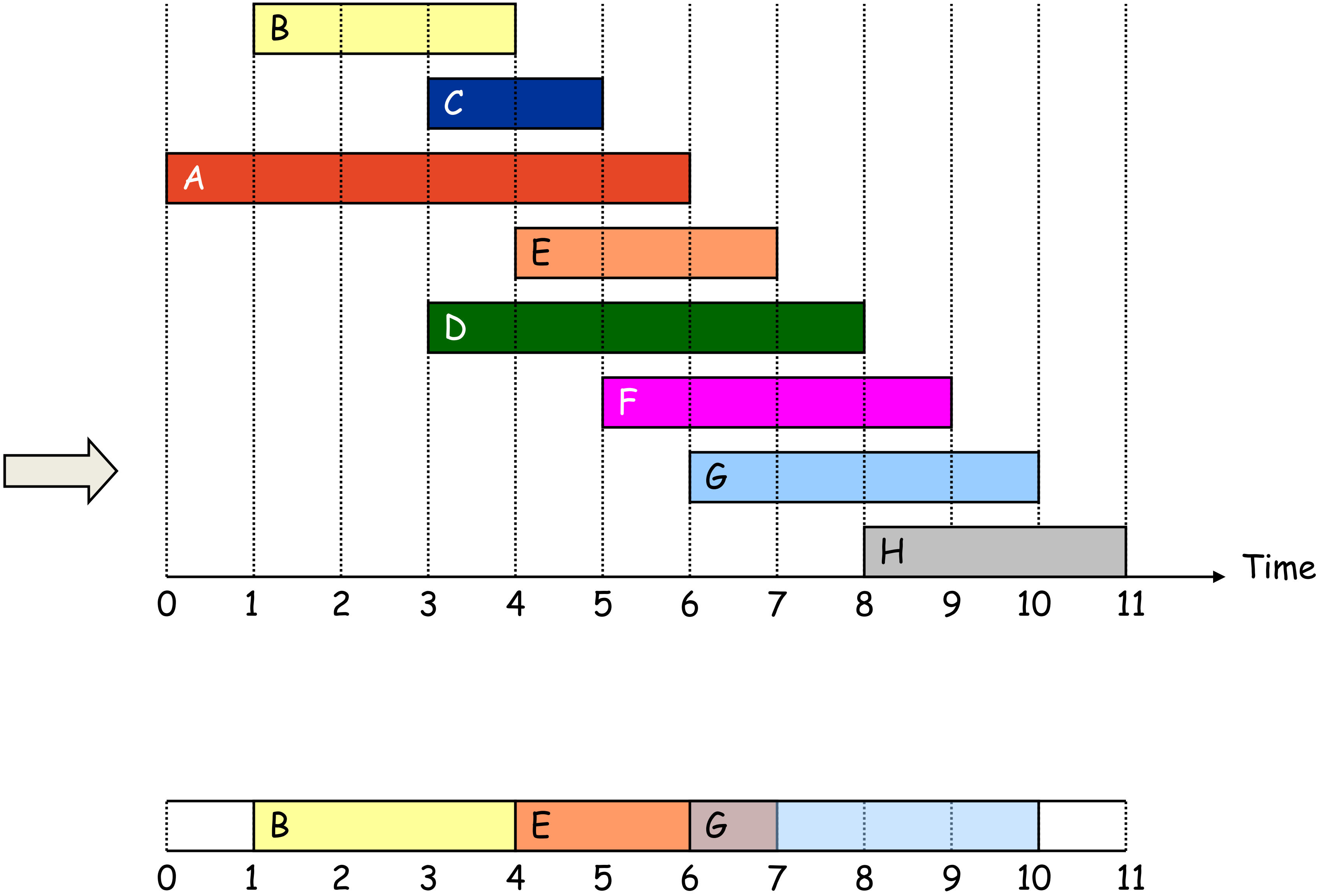
Interval Scheduling



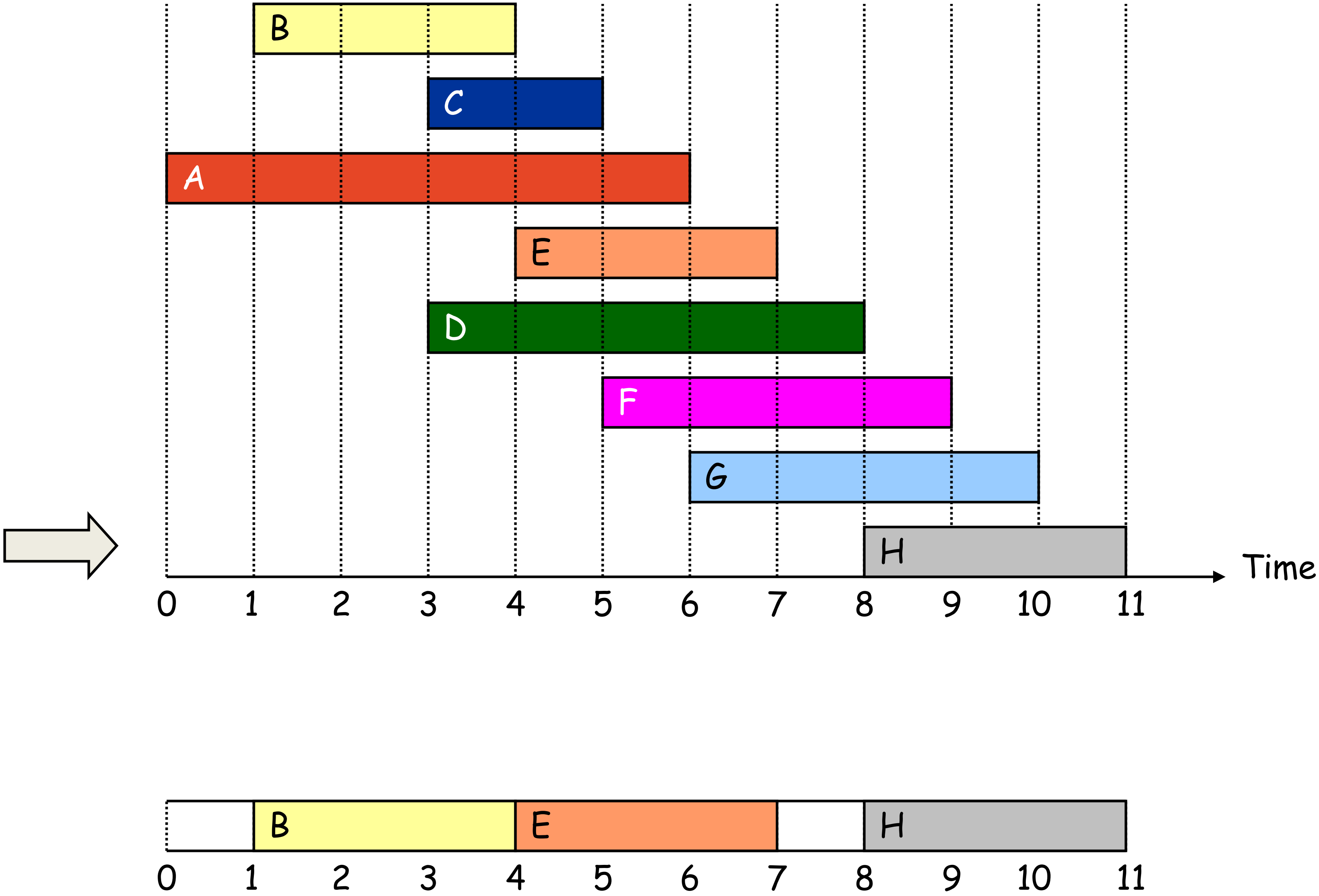
Interval Scheduling



Interval Scheduling



Interval Scheduling



Interval Scheduling: Greedy Algorithm

Only [Earliest finish time] remains to be tested.

- Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it is compatible with the ones already taken.

```
Sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$.   } O(n log n)

 jobs selected
A ← ∅
for j = 1 to n {                                   how to do this ?
    if (job j compatible with A)                 } O(1) time.
        A ← A ∪ {j}
}
return A
```

- Implementation. O(n log n).
  - Remember job j* that was added last to A.
  - Job j is compatible with A if $s_j \geq f_{j^*}$.

Interval Scheduling: Analysis

$X_{opt1} \rightarrow X_{opt2} \rightarrow X_{opt3} \rightarrow \cdots \rightarrow X_{greedy}$

High-Level General Idea:

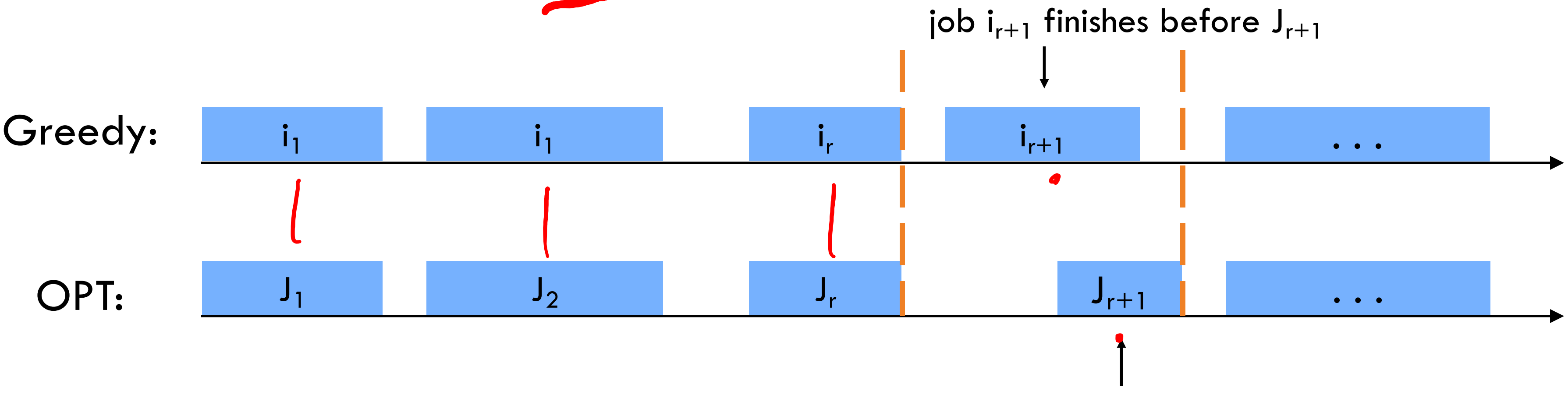
At each step of Greedy, there is an optimal solution consistent with Greedy's choices so far

One way to do this is by using an exchange argument.

1. Define your greedy solution.
2. Compare solutions. If $X_{greedy} \neq X_{opt}$, then they must differ in some specific way.
3. Exchange Pieces. Transform X_{opt} to a solution that is "closer" to X_{greedy} and prove cost doesn't increase.
4. Iterate. By iteratively exchanging pieces one can turn X_{opt} into X_{greedy} without impacting the quality of the solution.

Interval Scheduling: Analysis

- **Theorem:** Greedy algorithm [Earliest finish time] is optimal.
- **Proof:** (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $i_1, i_2, \ldots i_k$ denote the set of jobs selected by greedy.
  - Let $J_1, J_2, \ldots J_m$ denote the set of jobs in an optimal solution with $i_1 = J_1, i_2 = J_2, \ldots, i_r = J_r$ for the largest possible value of r.

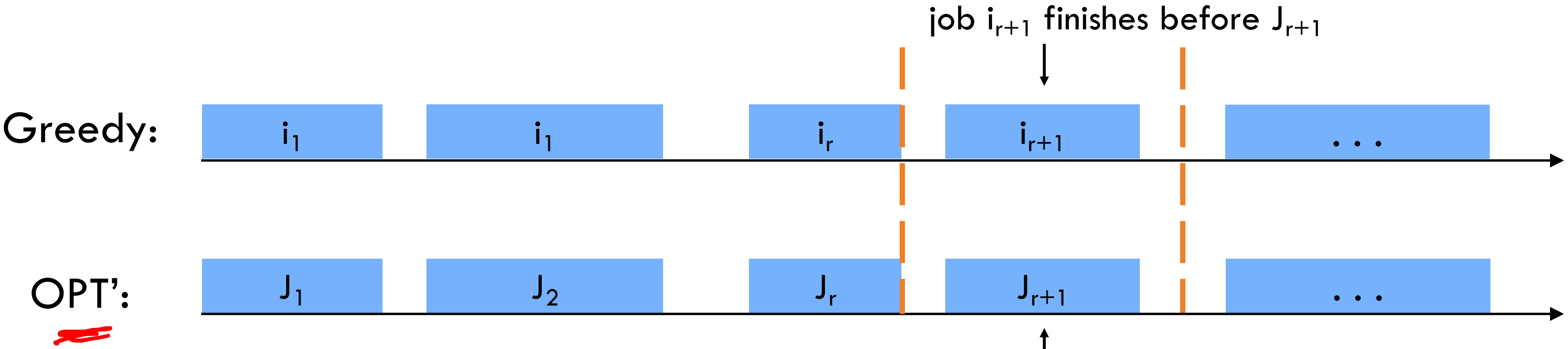


Why not replace job J_{r+1} with job i_{r+1}?

Interval Scheduling: Analysis

- **Theorem:** Greedy algorithm [Earliest finish time] is optimal.

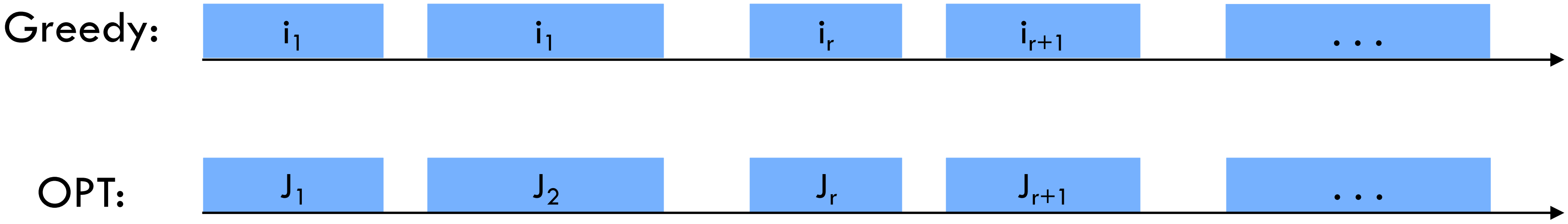
- **Proof:** (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $i_1, i_2, \ldots i_k$ denote the set of jobs selected by greedy.
  - Let $J_1, J_2, \ldots J_m$ denote the set of jobs in an optimal solution with $i_1 = J_1, i_2 = J_2, \ldots, i_r = J_r$ for the largest possible value of r.



Exchange argument!

solution still feasible and optimal and agrees with larger prefix of greedy's solution, contradicting definition of r

Interval Scheduling: Recap of Exchange Argument



Greedy: i_1 i_1 i_r i_{r+1} . . .

OPT: J_1 J_2 J_r J_{r+1} . . .

- We have an optimal solution that is “closer” to the greedy solution.

- Start the argument over again, but now the first (r+1) elements of the greedy solution and the optimal solution are identical.

- Continue iteratively until the optimal solution is transformed into the greedy solution without ~~increasing~~ the ~~cost~~.
  decreasing number of jobs

Interval Scheduling

There exists a greedy algorithm [Earliest finish time] that computes an optimal solution in O(n log n) time.

what about Latest start time?

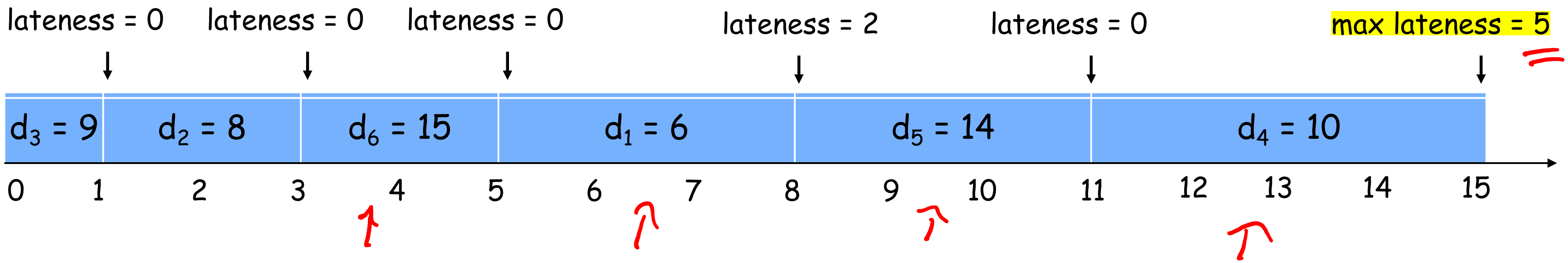
Scheduling to Minimize Lateness

Scheduling to Minimizing Lateness

- Minimizing lateness problem. [No fix start time]
  - Single resource processes one job at a time.
  - Job i requires t_i units of processing time and is due at time d_i.
  - Due times are unique
  - If i starts at time s_i, it finishes at time $f_i = s_i + t_i$.
  - Lateness: $\ell_i = \max \{ 0, f_i - d_i \}$.
  - **Goal:** schedule all jobs to minimize maximum lateness $L = \max \ell_i$.

- Ex:

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	10	14	15	due time



lateness = 0 lateness = 0 lateness = 0 lateness = 2 lateness = 0 max lateness = 5

$d_3 = 9$ $d_2 = 8$ $d_6 = 15$ $d_1 = 6$ $d_5 = 14$ $d_4 = 10$

Minimizing Lateness: Greedy Algorithms

→ max lateness $= \max_i l_i$

X total lateness $= \sum_i l_i$

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_i.

	1	2
t_i	1	10
d_i	100	10

99 slack

0 slack

counterexample

$l_1 = 0$

$l_2 = 1$

optimal lateness $= 0$

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

- [Smallest slack] Consider jobs in ascending order of slack d_j - t_j.

	1	2
t_j	1	10
d_j	2	10

slack 1 slack 0

counterexample

$\ell_2 = 0$

$\ell_1 = 11 - 2 = 9$

optimal lateness $= 1$

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

- [Smallest slack] Consider jobs in ascending order of slack d_j - t_j.

- [Earliest deadline first] Consider jobs in ascending order of deadline d_j.

Minimizing Lateness: Greedy Algorithm

– Greedy algorithm. [Earliest deadline first]

```
Sort n jobs by deadline so that $d_1 \le d_2 \le \ldots \le d_n$

$t \leftarrow 0$
for j = 1 to n
   Assign job j to interval $[t, t + t_j]$
   $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
   $t \leftarrow t + t_j$
output intervals $[s_j, f_j]$
```

O(n log n)

start next job when previous job finishes. O(1)

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	10	14	15	due time

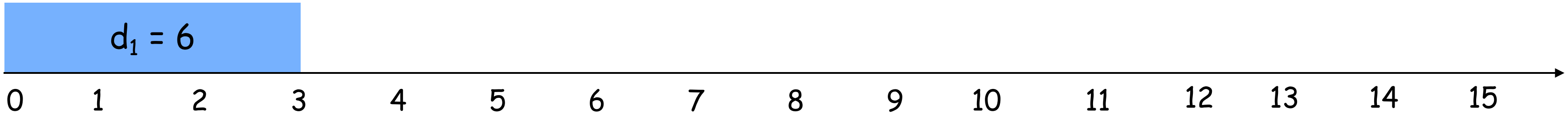
Minimizing Lateness: Greedy Algorithm

- Greedy algorithm. [Earliest deadline first]

```
Sort n jobs by deadline so that $d_1 \leq d_2 \leq \ldots \leq d_n$

$t \leftarrow 0$
for j = 1 to n
    Assign job j to interval $[t, t + t_j]$
    $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
    $t \leftarrow t + t_j$
output intervals $[s_j, f_j]$
```

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	10	14	15	due time



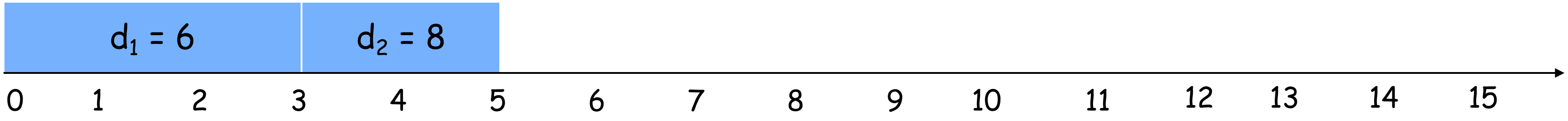
Minimizing Lateness: Greedy Algorithm

– Greedy algorithm. [Earliest deadline first]

```
Sort n jobs by deadline so that $d_1 \leq d_2 \leq \ldots \leq d_n$

$t \leftarrow 0$
for j = 1 to n
    Assign job j to interval $[t, t + t_j]$
    $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
    $t \leftarrow t + t_j$
output intervals $[s_j, f_j]$
```

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	10	14	15	due time



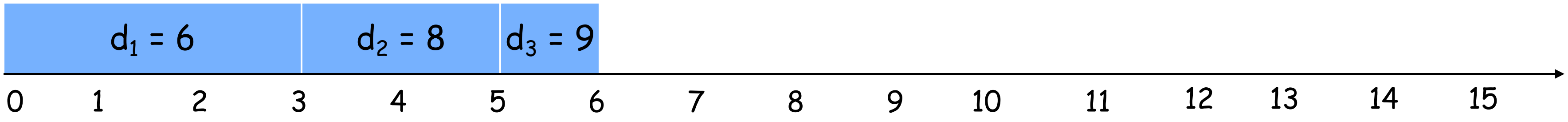
Minimizing Lateness: Greedy Algorithm

- Greedy algorithm. [Earliest deadline first]

```
Sort n jobs by deadline so that $d_1 \le d_2 \le \dots \le d_n$

$t \leftarrow 0$
for j = 1 to n
   Assign job j to interval $[t, t + t_j]$
   $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
   $t \leftarrow t + t_j$
output intervals $[s_j, f_j]$
```

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	10	14	15	due time



Minimizing Lateness: Greedy Algorithm

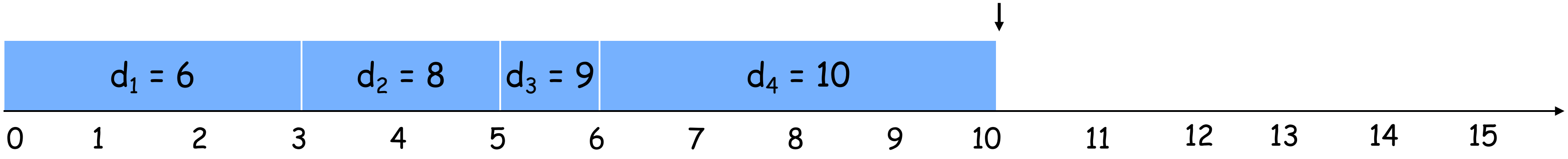
- Greedy algorithm. [Earliest deadline first]

```
Sort n jobs by deadline so that $d_1 \le d_2 \le … \le d_n$

$t \leftarrow 0$
for j = 1 to n
   Assign job j to interval $[t, t + t_j]$
   $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
   $t \leftarrow t + t_j$
output intervals $[s_j, f_j]$
```

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	10	14	15	due time

max lateness = ~~1~~ 0



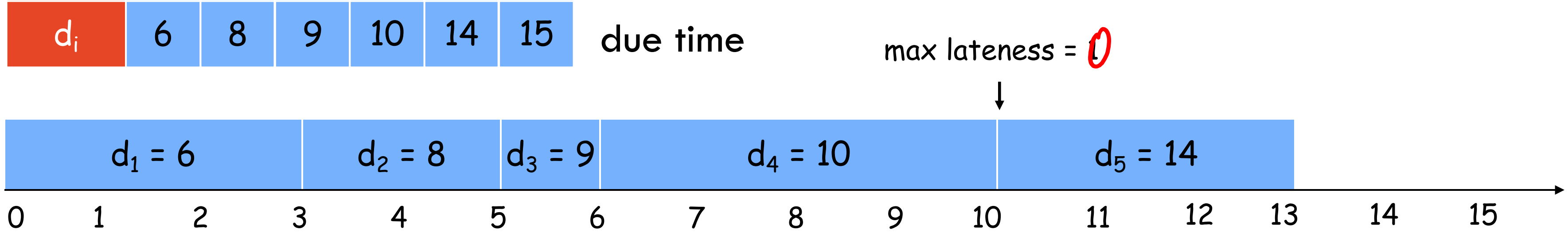
Minimizing Lateness: Greedy Algorithm

- Greedy algorithm. [Earliest deadline first]

```
Sort n jobs by deadline so that $d_1 \le d_2 \le \ldots \le d_n$

$t \leftarrow 0$
for j = 1 to n
   Assign job j to interval $[t, t + t_j]$
   $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
   $t \leftarrow t + t_j$
output intervals $[s_j, f_j]$
```

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	10	14	15	due time



Minimizing Lateness: Greedy Algorithm

- Greedy algorithm. [Earliest deadline first]

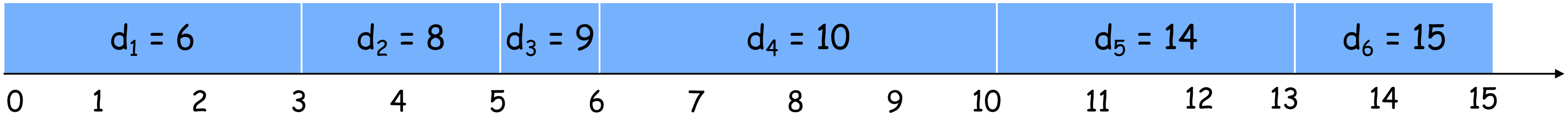
```
Sort n jobs by deadline so that $d_1 \le d_2 \le … \le d_n$

$t \leftarrow 0$
for j = 1 to n
   Assign job j to interval $[t, t + t_j]$
   $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
   $t \leftarrow t + t_j$
output intervals $[s_j, f_j]$
```

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	10	14	15	due time

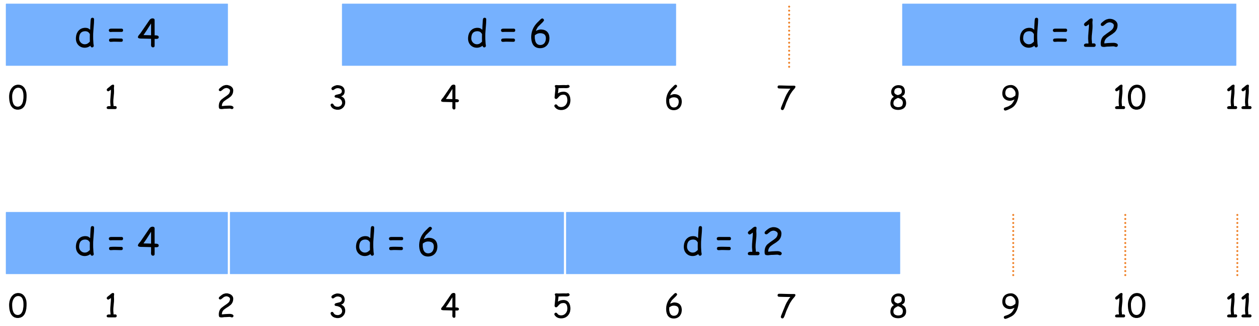
Algorithm ignores processing time!

max lateness = 0



Minimizing Lateness: No Idle Time

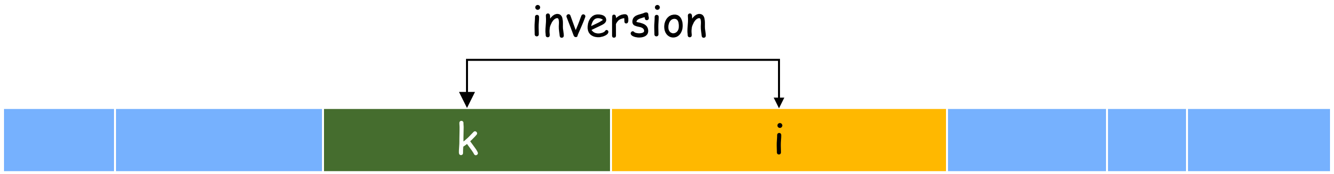
- **Observation:** There exists an optimal schedule with no idle time.



- **Observation:** The greedy schedule has no idle time.

Minimizing Lateness: Inversions

- **Definition:** An inversion in schedule S is a pair of jobs i and k such that i < k (by deadline) but k is scheduled before i.



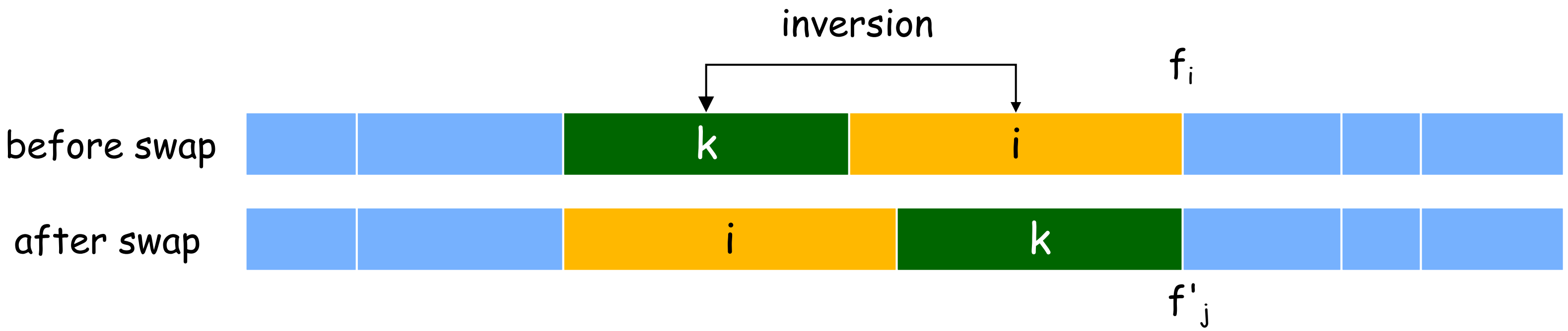
- **Observation:** Greedy schedule has no inversions. Moreover, Greedy is only such schedule (by uniqueness of deadlines).

- **Observation:** If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

WHY?

Minimizing Lateness: Inversions

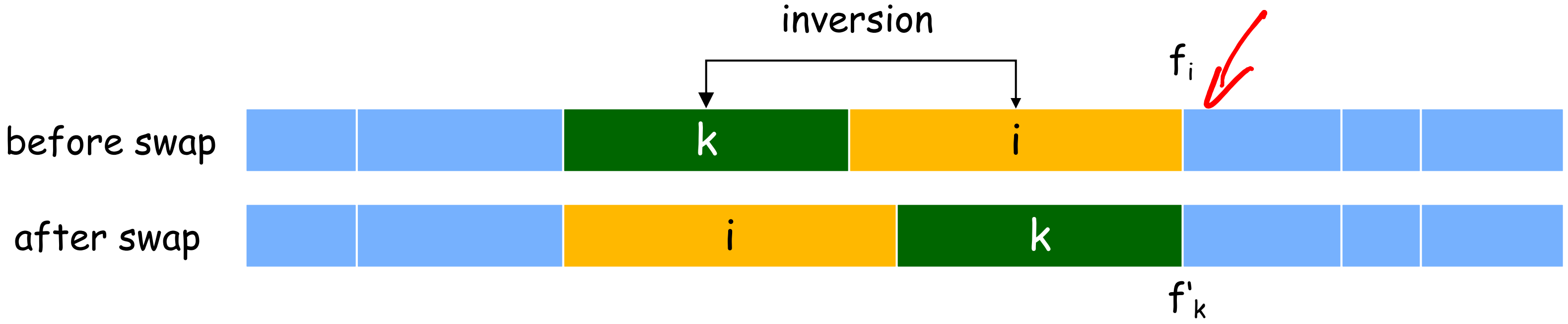
- **Definition:** An inversion in schedule S is a pair of jobs i and k such that i < k (by deadline) but k is scheduled before i.



- **Claim:** Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

Minimizing Lateness: Inversions

- **Definition:** An inversion in schedule S is a pair of jobs i and k such that i < k (by deadline) but k is scheduled before i.



- **Claim:** Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

- **Proof:** Let ℓ be the lateness before the swap, and let ℓ' be the lateness after the swap.
  - → $\ell'_x = \ell_x$ for all $x \neq i, k$
  - → $\ell'_i \leq \ell_i$
  - If job k is late:

$$\underline{\ell'_k = f'_k - d_k} \qquad \text{definition}$$

$$= f_i - d_k \qquad (f'_k = f_i) \qquad (i \text{ finishes at time } f_i)$$

$$\rightarrow \leq f_i - d_i \qquad (d_i < d_k) \qquad (i < k)$$

$$= \ell_i \qquad \text{(definition)}$$

→ lateness of job i before the swap

max lateness ≤ before swap

Minimizing Lateness: Analysis of Greedy Algorithm

- **Theorem:** Greedy schedule S is optimal.

- **Proof:** Define S* to be an optimal schedule that has the fewest number of inversions, and let's see what happens.
  - Can assume S* has no idle time.
  - If S* has no inversions, then S = S*.
  - If S* has an inversion, let i-k be an adjacent inversion.
    - swapping i and k does not increase the maximum lateness and strictly decreases the number of inversions
    - this contradicts definition of S* ■

Minimizing Lateness

There exists a greedy algorithm [Earliest deadline first] that computes the optimal solution in O(n log n) time.

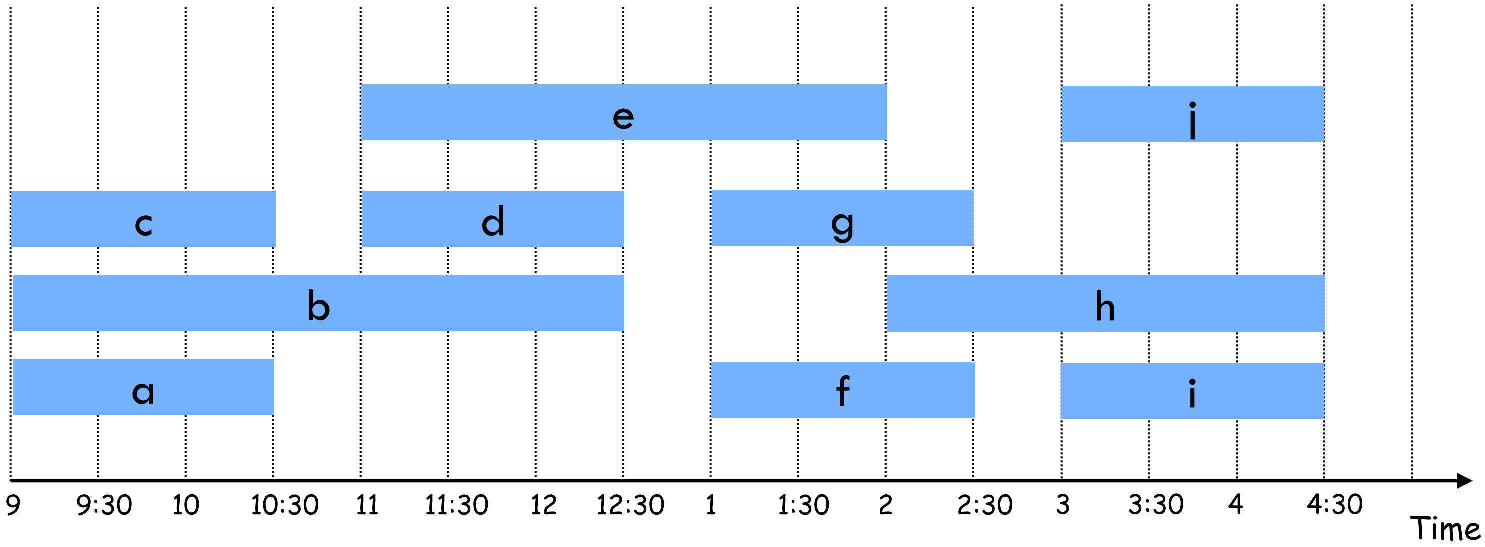
What if deadlines are not unique?
Can show that all schedules with no idle time and no inversions have the same lateness (p.g. 128 of textbook)

Interval Partitioning

Interval Partitioning

- Interval partitioning.
  - Lecture i starts at s_i and finishes at f_i. Assume integers.
  - Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

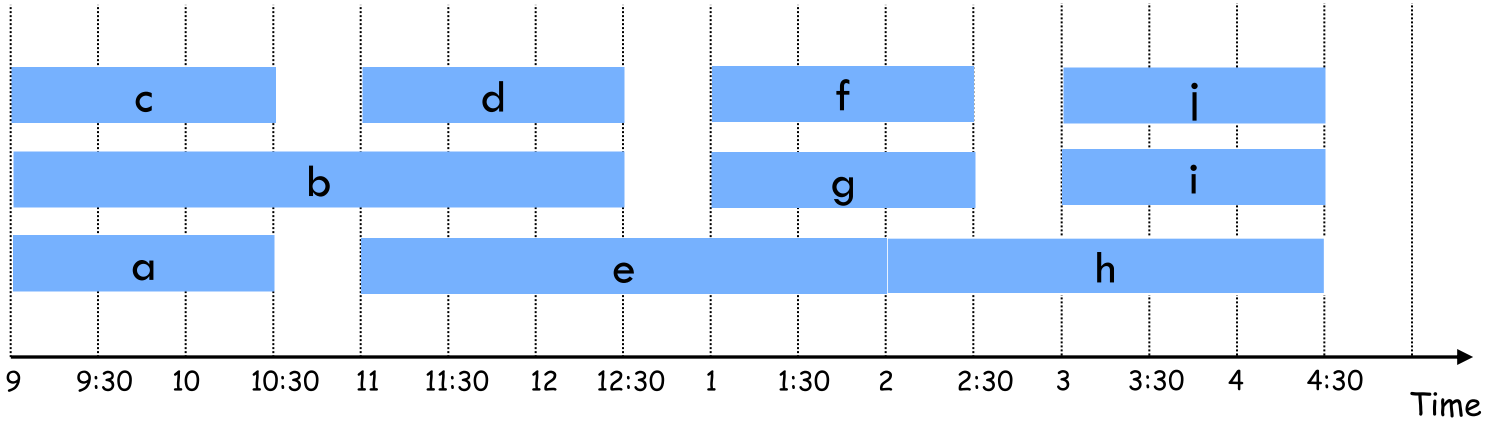
- Ex: This schedule uses 4 classrooms to schedule 10 lectures.



Interval Partitioning

- Interval partitioning.
  - Lecture i starts at s_i and finishes at f_i.
  - Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

- Ex: This schedule uses only 3.

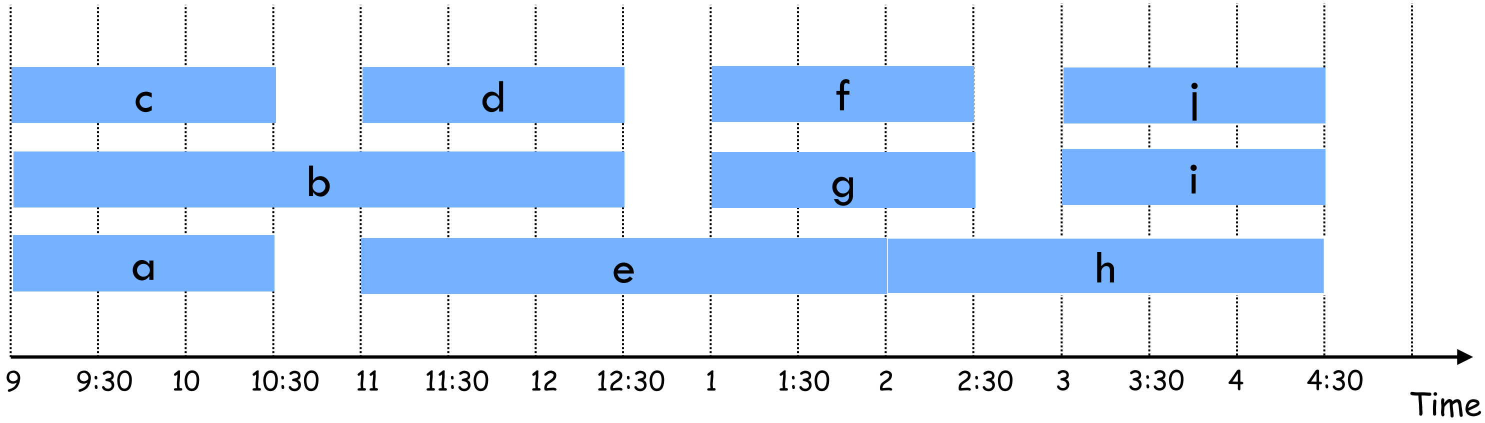


Interval Partitioning: Lower bound

- **Definition:** The depth of a set of open intervals is the maximum number that contain any given time.

- **Observation:** Number of classrooms needed ≥ depth.

- Example: Depth of schedule below is 3 (a, b, c all contain 9:30)
  ⇒ schedule below is optimal.
- **Question:** Does there always exist a schedule equal to depth of intervals?



Interval Partitioning: Greedy Algorithm

- **Greedy algorithm.** Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that $s_1 \le s_2 \le \ldots \le s_n$.
d ← 0    ← number of allocated classrooms

for i = 1 to n {
   if (lecture i is compatible with some classroom k)
      schedule lecture i in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture i in classroom d + 1
      d ← d + 1
}
```

Interval Partitioning: Greedy Algorithm

- **Greedy algorithm.** Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that $s_1 \leq s_2 \leq ... \leq s_n$.
d ← 0    number of allocated classrooms

for i = 1 to n {
   if (lecture i is compatible with some classroom k)
      schedule lecture i in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture i in classroom d + 1
      d ← d + 1
}
```

- **Implementation.** O(n log n).
  - For each classroom k, maintain the finish time of the last job added.
  - Keep the classrooms in a priority queue.

Interval Partitioning: Greedy Analysis

- **Observation:** Greedy algorithm never schedules two incompatible lectures in the same classroom so it is feasible.

- **Theorem:** Greedy algorithm is optimal.
- **Proof:**
  - d = number of classrooms that the greedy algorithm allocates.
  - Classroom d is opened because we needed to schedule a job, say i, that is incompatible with all d-1 other classrooms.
  - Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_i.
  - Thus, we have d lectures overlapping at time $[s_i, s_i + 1]$. ← just after time s_i
  - Key observation $\Rightarrow$ all schedules use $\geq$ d classrooms.

■

Interval Partitioning

There exists a greedy algorithm [Earliest starting time] that computes the optimal solution in O(n log n) time.

Greedy Analysis Strategies

- Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

- Structural. Discover a simple "structural" bound asserting that every possible solution must have at least (or at most) a certain value. Then show that your algorithm always achieves this bound.

4.3 Optimal Caching

Optimal Offline Caching

Caching.

- Cache with capacity to store k items.
- Sequence of m item requests $d_1, d_2, \ldots, d_m$.
- Cache hit: item already in cache when requested.
- Cache miss: item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

Goal. Eviction schedule that minimizes number of cache misses.

Optimal Offline Caching

Caching.

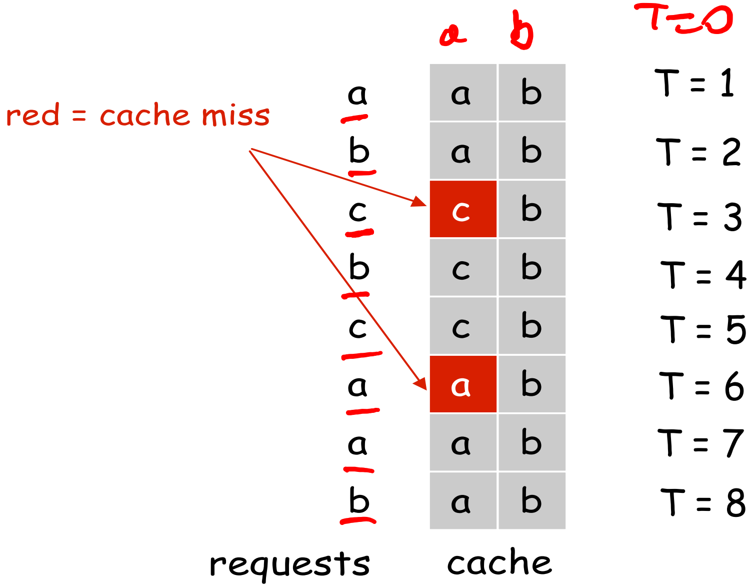
- Cache with capacity to store k items.
- Sequence of m item requests $d_1, d_2, \ldots, d_m$.
- Cache hit: item already in cache when requested.
- Cache miss: item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

Goal. Eviction schedule that minimizes number of cache misses.

Ex: k = 2, initial cache = ab,

requests: a, b, c, b, c, a, a, b.

Optimal eviction schedule: 2 cache misses.



red = cache miss

requests	cache		
	a	b	T=0
a	a	b	T = 1
b	a	b	T = 2
c	c	b	T = 3
b	c	b	T = 4
c	c	b	T = 5
a	a	b	T = 6
a	a	b	T = 7
b	a	b	T = 8

Optimal Offline Caching

Caching.

- Cache with capacity to store k items.
- Sequence of m item requests $d_1, d_2, \ldots, d_m$.
- Cache hit: item already in cache when requested.
- Cache miss: item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

Goal. Eviction schedule that minimizes number of cache misses.

Ex: k = 2, initial cache = ab,
requests: a, b, c, b, c, a, a, b.

Optimal eviction schedule: 2 cache misses.

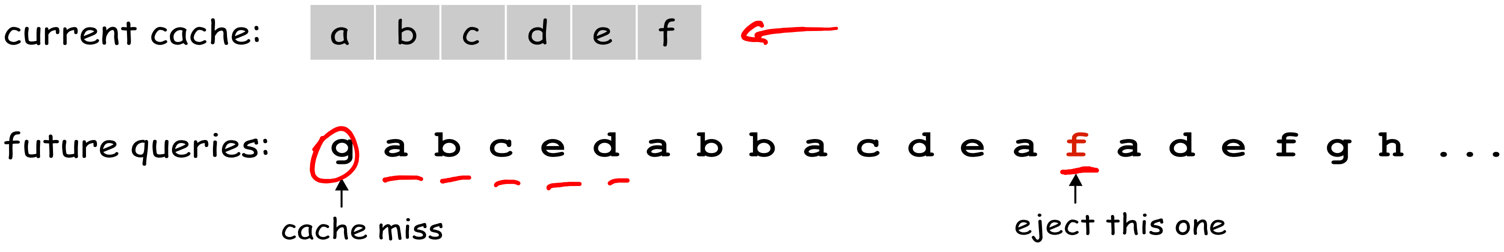
Least recently used?

Not optimal

	a	b	T=0
a	a	b	T = 1
b	a	b	T = 2
c	c	b	T = 3
b	c	b	T = 4
c	c	b	T = 5
c	c	b	T = 6
a	c	a	T = 7
b	b	a	T = 8
requests	cache		

Optimal Offline Caching: Farthest-In-Future

Farthest-in-future. Evict item in the cache that is not requested until farthest in the future.



current cache: a b c d e f

future queries: g a b c e d a b b a c d e a f a d e f g h . . .

cache miss

eject this one

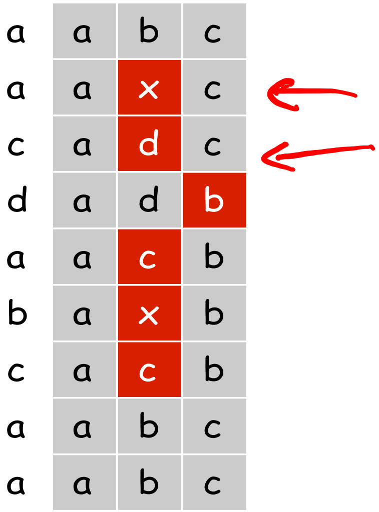
Theorem. [Bellady, 1960s] FF is optimal eviction schedule.

Pf. Algorithm and theorem are intuitive; proof is subtle.

Reduced Eviction Schedules

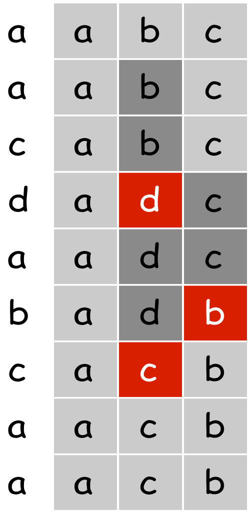
Def. A reduced schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.

Intuition. Can transform an unreduced schedule into a reduced one with no more cache misses.



a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	c	b
b	a	x	b
c	a	c	b
a	a	b	c
a	a	b	c

an unreduced schedule



a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
a	a	c	b
a	a	c	b

a reduced schedule

Be lazy, sloth is good

Reduced Eviction Schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more cache misses.

Pf. (by induction on number of unreduced items) ← doesn't enter cache at requested time

Reduced Eviction Schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more cache misses.

Pf. (by induction on number of unreduced items)

doesn't enter cache at requested time

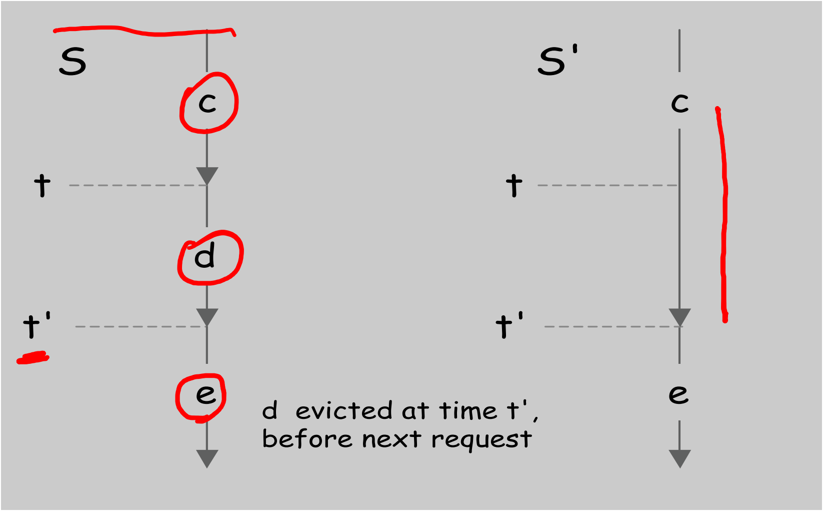
- Suppose S brings d into the cache at time t, without a request.
- Let c be the item S evicts when it brings d into the cache.

Reduced Eviction Schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more cache misses.

Pf. (by induction on number of unreduced items) ← doesn't enter cache at requested time

- Suppose S brings d into the cache at time t, without a request.
- Let c be the item S evicts when it brings d into the cache.
- Case 1: d evicted at time t', before next request for d.



d evicted at time t', before next request

Case 1

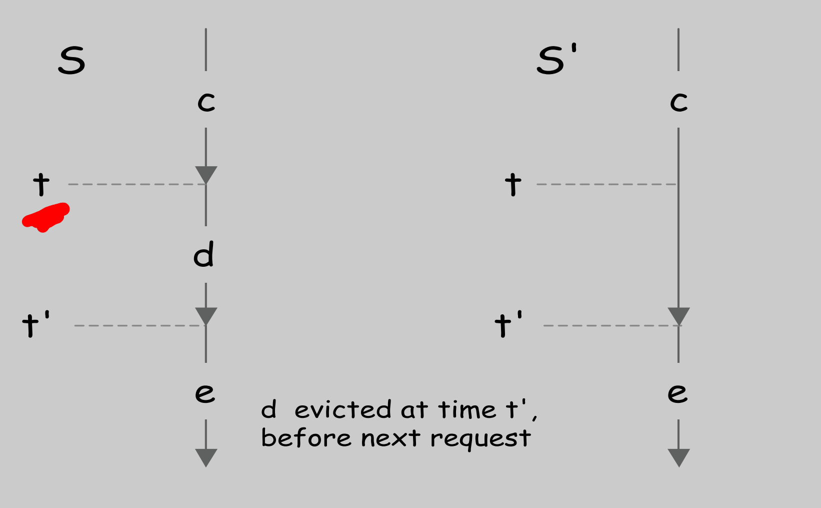
Reduced Eviction Schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more cache misses.

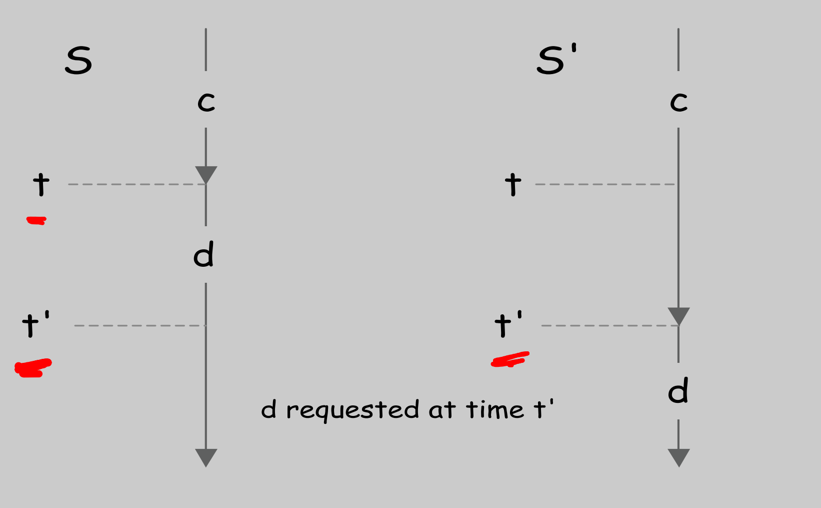
Pf. (by induction on number of unreduced items)

doesn't enter cache at requested time

- Suppose S brings d into the cache at time t, without a request.
- Let c be the item S evicts when it brings d into the cache.
- Case 1: d evicted at time t', before next request for d.
- Case 2: d requested at time t' before d is evicted. ▪



Case 1



Case 2

no difference

more misses

Farthest-In-Future: Analysis

Theorem. FF is optimal eviction algorithm.

Pf. (by induction on number or requests j)

Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as S_{FF} through the first j+1 requests.

Farthest-In-Future: Analysis

Theorem. FF is optimal eviction algorithm.
Pf. (by induction on number or requests j)

> Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as S_{FF} through the first j+1 requests.

Let S be reduced schedule that satisfies invariant through j requests.
We produce S' that satisfies invariant after j+1 requests.

- Consider $(j+1)^{st}$ request $d = d_{j+1}$.
- Since S and S_{FF} have agreed up until now, they have the same cache contents before request j+1.

Farthest-In-Future: Analysis

Theorem. FF is optimal eviction algorithm.

Pf. (by induction on number or requests j)

> Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as S_{FF} through the first j+1 requests.

Let S be reduced schedule that satisfies invariant through j requests. We produce S' that satisfies invariant after j+1 requests.

- Consider $(j+1)^{st}$ request $d = d_{j+1}$.
- Since S and S_{FF} have agreed up until now, they have the same cache contents before request j+1.
- Case 1: (d is already in the cache). S' = S satisfies invariant.

Farthest-In-Future: Analysis

Let S be a reduced schedule that is the same as SFF for the first j requests, then there is a reduced schedule S' that is the same as SFF for the first j+1 requests, and incurs no more misses than S

Theorem. FF is optimal eviction algorithm.

Pf. (by induction on number or requests j)

Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as S_{FF} through the first j+1 requests.

Let S be optimal reduced schedule that satisfies invariant through j requests.
We produce S' that satisfies invariant after j+1 requests.

- Consider (j+1)st request $d = d_{j+1}$.
- Since S and S_{FF} have agreed up until now, they have the same cache contents before request j+1.
- Case 1: (d is already in the cache). S' = S satisfies invariant.
- Case 2: (d is not in the cache and S and S_{FF} evict the same element). S' = S satisfies invariant.

Farthest-In-Future: Analysis

Pf. (continued)

- Case 3: (d is not in the cache; S_{FF} evicts e; S evicts f $\neq$ e).
  - begin construction of S' from S by evicting e instead of f

j	same	e	f	same	e	f
		S			S'	
j+1	same	e	d	same	d	f
		S			S'	

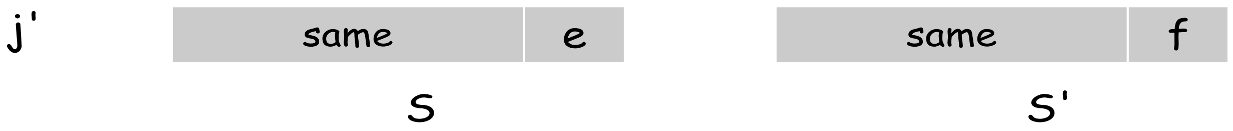
  - now S' agrees with S_{FF} on first j+1 requests; we show that having element f in cache is no worse than having element e

Farthest-In-Future: Analysis

- e is requested
- f is requested
- S evicts e
- S' evicts f

Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

must involve e or f (or both)

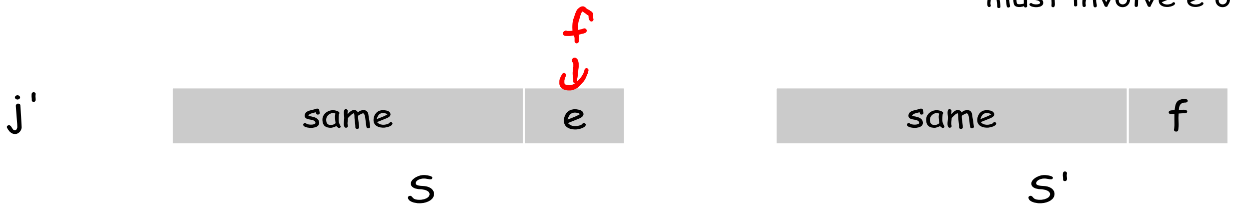


- Case 3a: g = e. Can't happen with Farthest-In-Future since there must be a request for f before e.

Farthest-In-Future: Analysis

Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

↑ must involve e or f (or both)



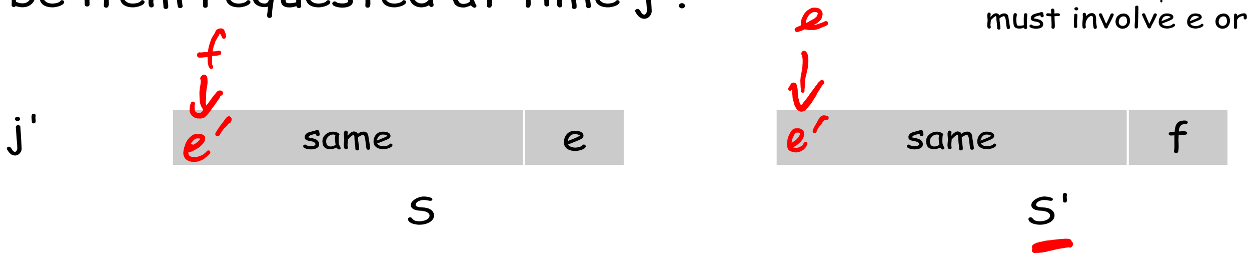
- Case 3a: g = e. Can't happen with Farthest-In-Future since there must be a request for f before e.

- Case 3b: g = f. Element f can't be in cache of S, so let e' be the element that S evicts.
  - if e' = e, S' accesses f from cache; now S and S' have same cache

Farthest-In-Future: Analysis

Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

↑ must involve e or f (or both)



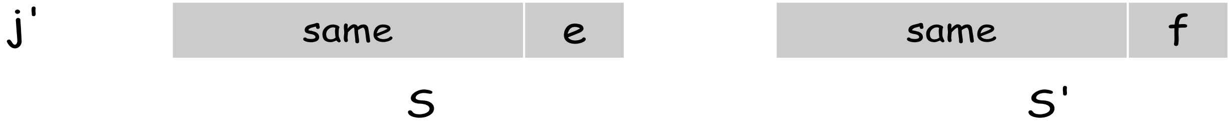
- Case 3a: g = e. Can't happen with Farthest-In-Future since there must be a request for f before e.

- Case 3b: g = f. Element f can't be in cache of S, so let e' be the element that S evicts.
  - if e' = e, S' accesses f from cache; now S and S' have same cache
  - if e' ≠ e, S' evicts e' and brings e into the cache; now S and S' have the same cache

Farthest-In-Future: Analysis

Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

↑ must involve e or f (or both)



j'

same | e

S

same | f

S'

- Case 3a: g = e. Can't happen with Farthest-In-Future since there must be a request for f before e. ?

- Case 3b: g = f. Element f can't be in cache of S, so let e' be the element that S evicts.
  - if e' = e, S' accesses f from cache; now S and S' have same cache
  - if e' ≠ e, then S' evicts e' and brings e into the cache; now S and S' have the same cache

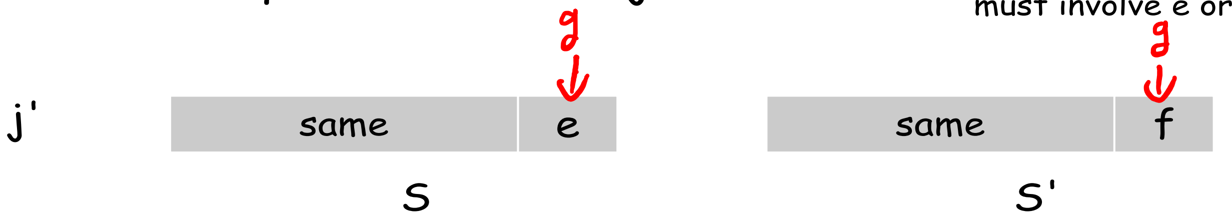
↑ Note: S' is no longer reduced, but can be transformed into a reduced schedule that agrees with S_{FF} through step j+1

j' is first time S' has unreduced item.

Farthest-In-Future: Analysis

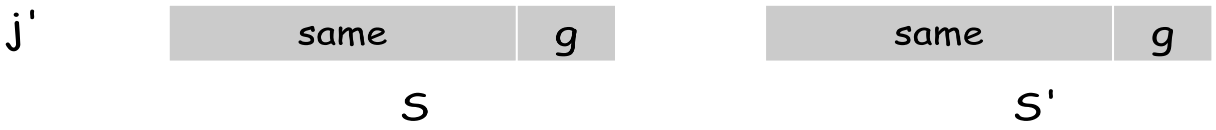
Let j' be the **first time** after j+1 that S and S' take a **different action**, and let g be item requested at time j'.

must involve e or f (or both)



otherwise S' would take the same action

- Case 3c: $g \neq e, f$. **S must evict e.**
  Make S' evict f; now S and S' have the same cache. ▪



Caching Perspective

Online vs. offline algorithms.

- Offline: full sequence of requests is known a priori.
- Online (reality): requests are not known in advance.
- Caching is among most fundamental online problems in CS.

LIFO. Evict page brought in most recently.

LRU. Evict page whose most recent access was earliest.

FF with direction of time reversed!

Theorem. FF is optimal offline eviction algorithm.

- Provides basis for understanding and analyzing online algorithms.
- LRU is k-competitive. i.e. at most k times worse than optimal
- LIFO is arbitrarily bad.

Summary: Greedy algorithms

A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

Problems

- Interval scheduling/partitioning
- Scheduling: minimize lateness
- Caching
- Shortest path in graphs (Dijkstra's algorithm)
- Minimum spanning tree (Prim's/Kruskal's algorithms)
- …

This Week

- **Quiz 1:**
  - Posted tonight
  - Due Sunday 14 March 23:59:00
    - One try
    - 20 minutes from the time quiz is opened
    - No late submissions accepted