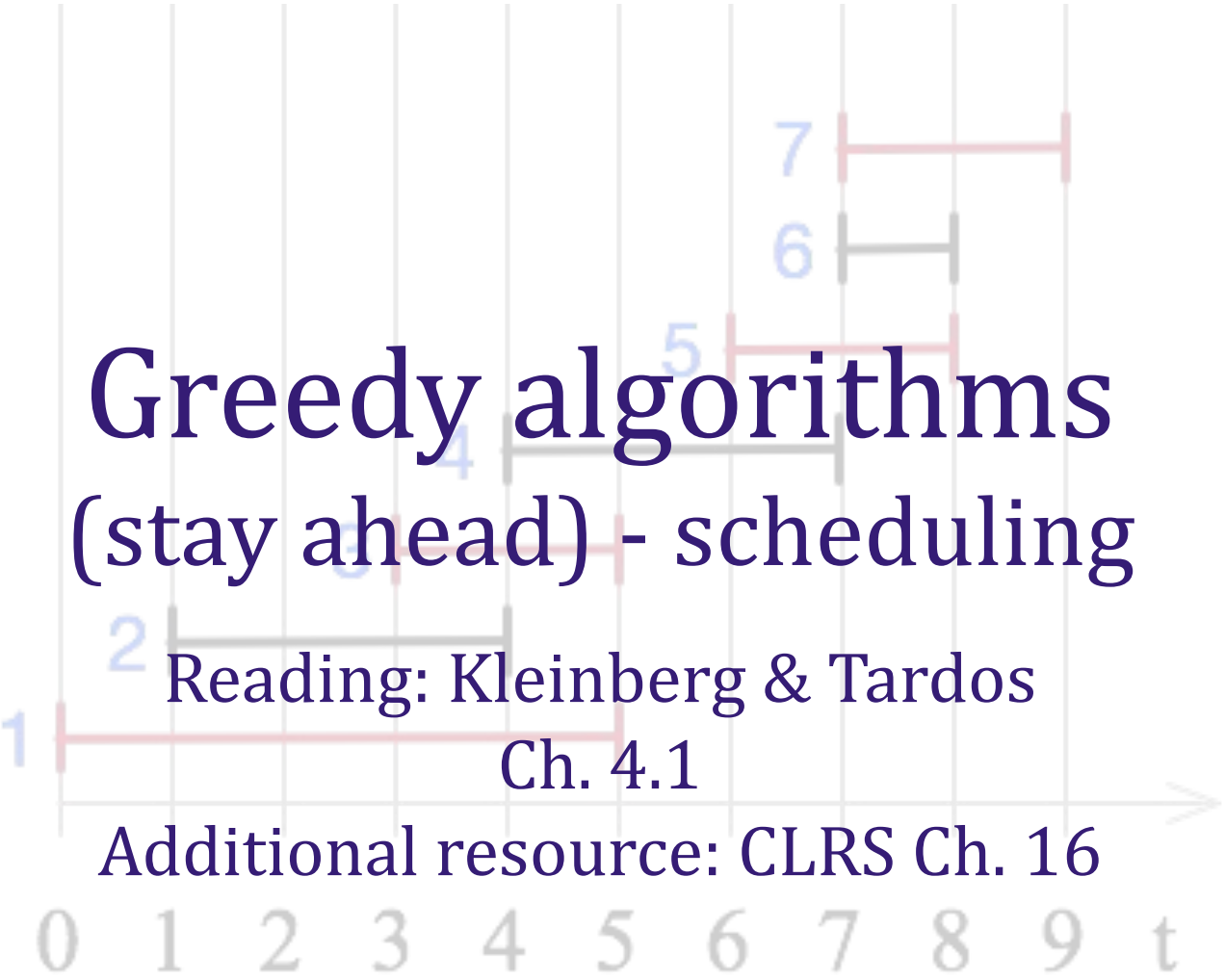


Greedy algorithms

(stay ahead) - scheduling

Reading: Kleinberg & Tardos
Ch. 4.1
Additional resource: CLRS Ch. 16

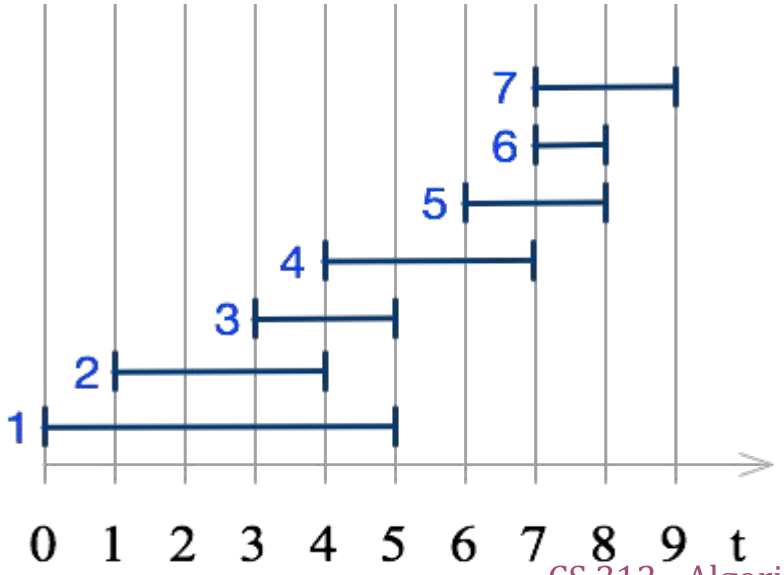
So much to do, so little time…

Senior symposium is only a couple months away! Your plan is to look at the schedule and mark down sessions and talks that you want to attend. Unfortunately, some will overlap, and you'll be forced to make a choice.

How do you choose sessions and talks so that you maximize the total number that you attend?

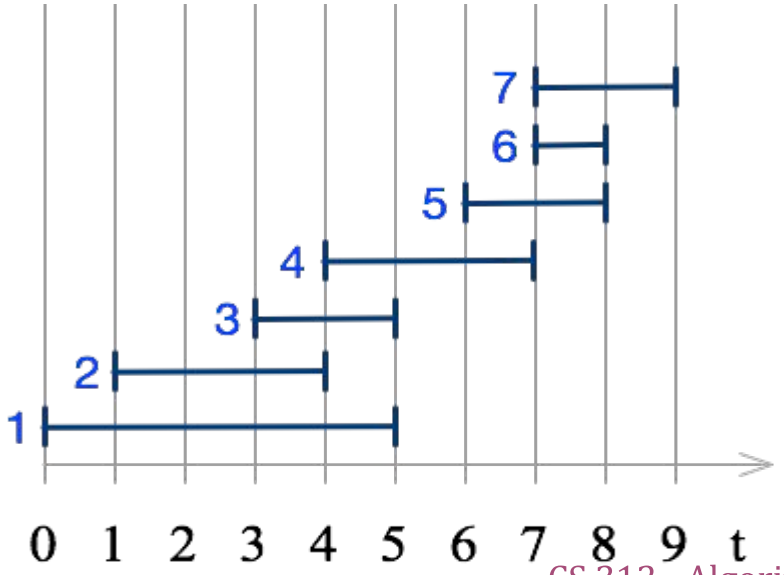
Formalizing the problem

- Set of 1, …, *n* sessions/talks



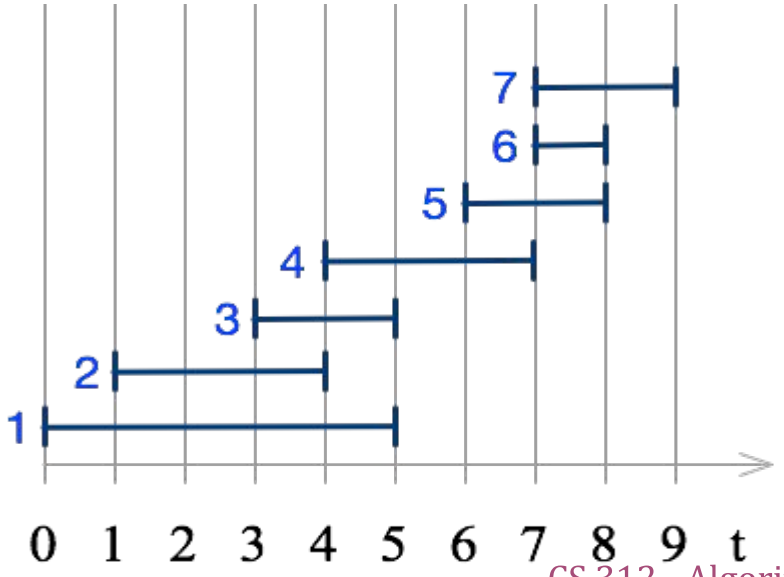
Formalizing the problem

- Set of 1, …, *n* requests
- *s(i)*, *f(i)* start and finish times



Problem 1: interval scheduling

- Set of 1, …, *n* requests
- *s(i)*, *f(i)* start and finish times
- i and j **compatible**: no overlap
- **compatible set**: no pair overlaps
- **optimal**: maximum-sized compatible set



i	*s*(*i*)	*f*(*i*)
1	0	5
2	1	4
3	3	5
4	4	7
5	6	8
6	7	8
7	7	9

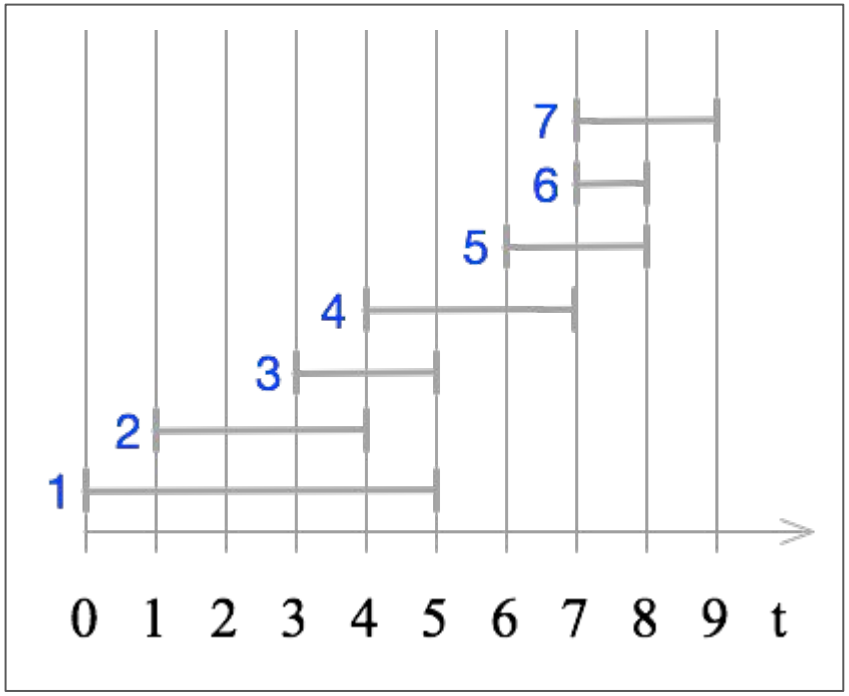
How to pick?

- Build up set by adding one request at a time
- Choose the locally "best" request
- Will it lead to an optimal solution?

R = all requests sorted by some property
S = {} // selected requests
while (*R* is not empty)
    remove next request *i* from *R*
    add *i* to *S*
    remove all overlapping requests from *R*

Next request please!

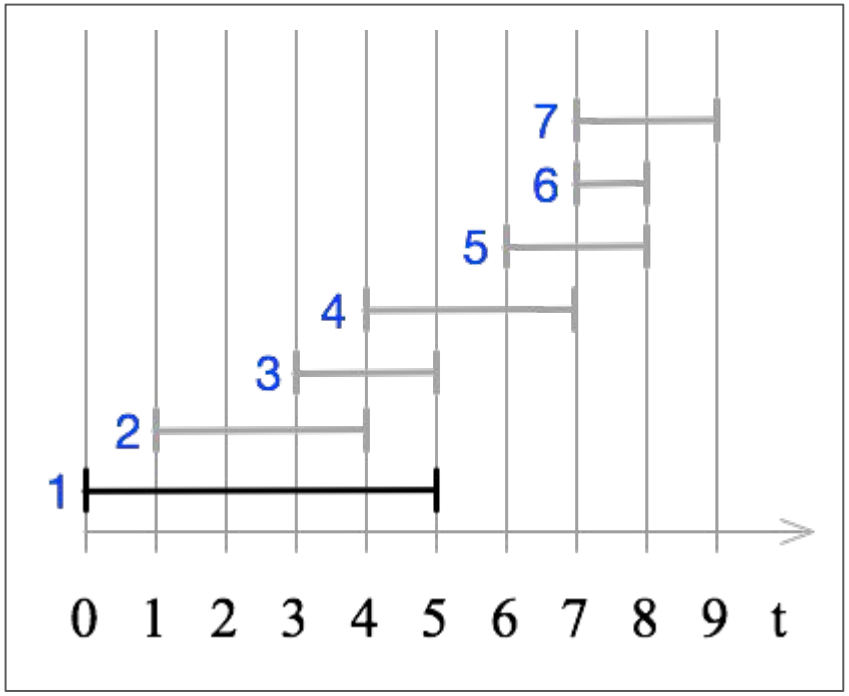
Let's try taking them in order of next starting time:



i	*s*(*i*)	*f*(*i*)
1	0	5
2	1	4
3	3	5
4	4	7
5	6	8
6	7	8
7	7	9

Next request please!

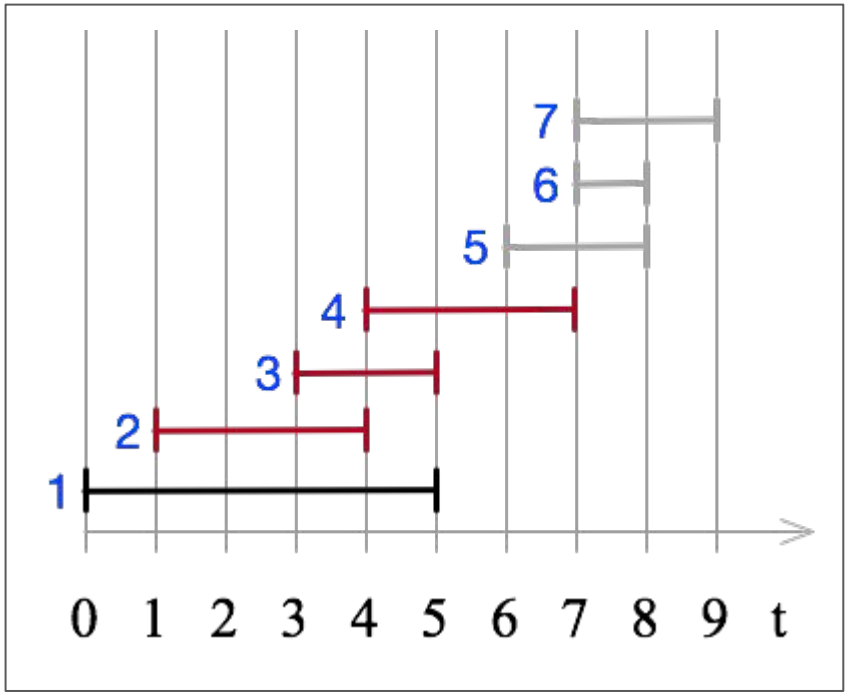
Let's try taking them in order of next starting time:



i	*s*(*i*)	*f*(*i*)
1	**0**	**5**
2	1	4
3	3	5
4	4	7
5	6	8
6	7	8
7	7	9

Next request please!

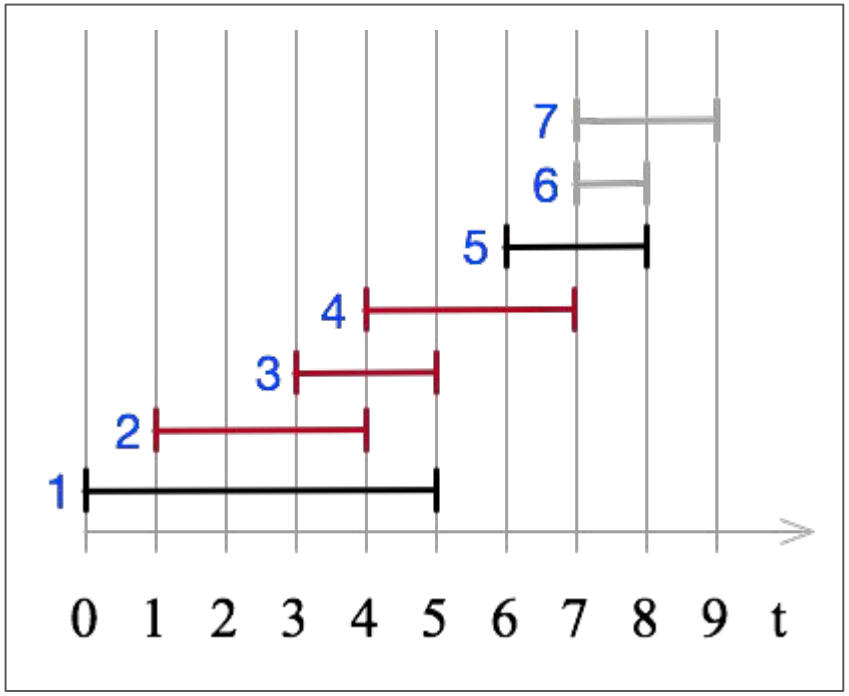
Let's try taking them in order of next starting time:



i	*s*(*i*)	*f*(*i*)
1	**0**	**5**
2	1	4
3	3	5
4	4	7
5	6	8
6	7	8
7	7	9

Next request please!

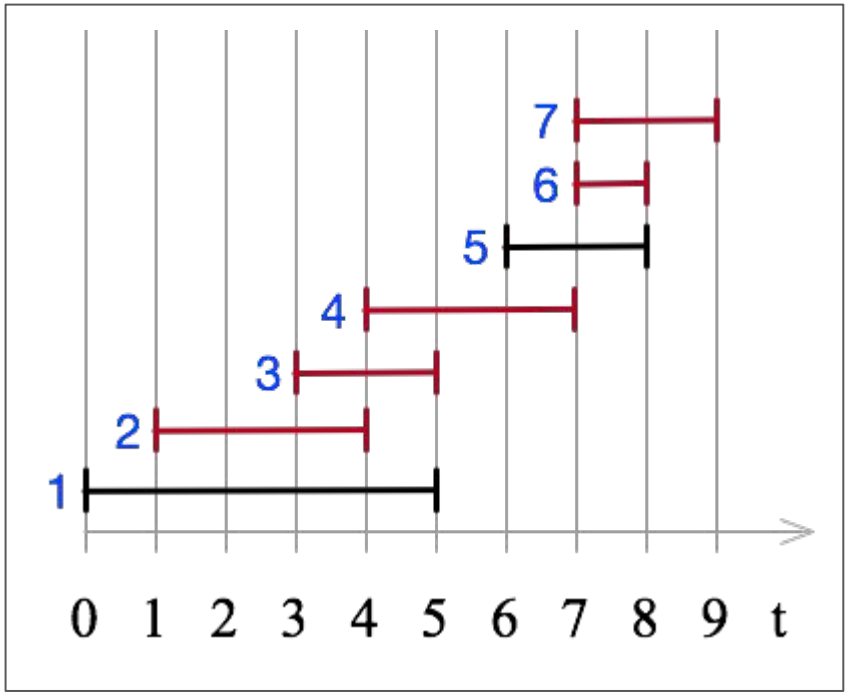
Let's try taking them in order of next starting time:



i	*s*(*i*)	*f*(*i*)
1	**0**	**5**
2	1	4
3	3	5
4	4	7
5	**6**	**8**
6	7	8
7	7	9

Next request please!

Let's try taking them in order of next starting time:

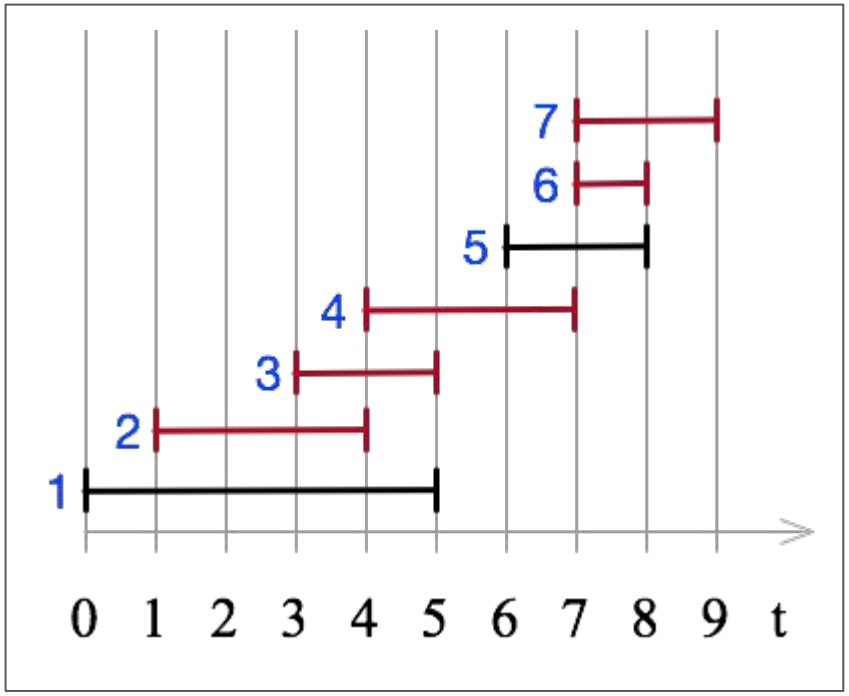


i	*s*(*i*)	*f*(*i*)
1	**0**	**5**
2	1	4
3	3	5
4	4	7
5	**6**	**8**
6	7	8
7	7	9

Next request please!

Let's try taking them in order of next starting time:

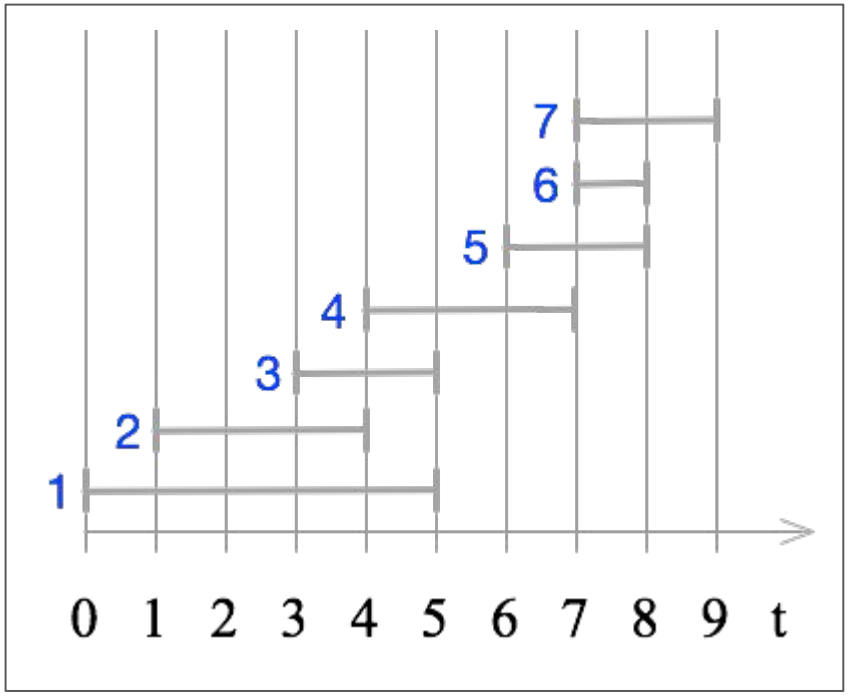
2 requests can be fulfilled



i	*s*(*i*)	*f*(*i*)
1	**0**	**5**
2	1	4
3	3	5
4	4	7
5	**6**	**8**
6	7	8
7	7	9

Less is more?

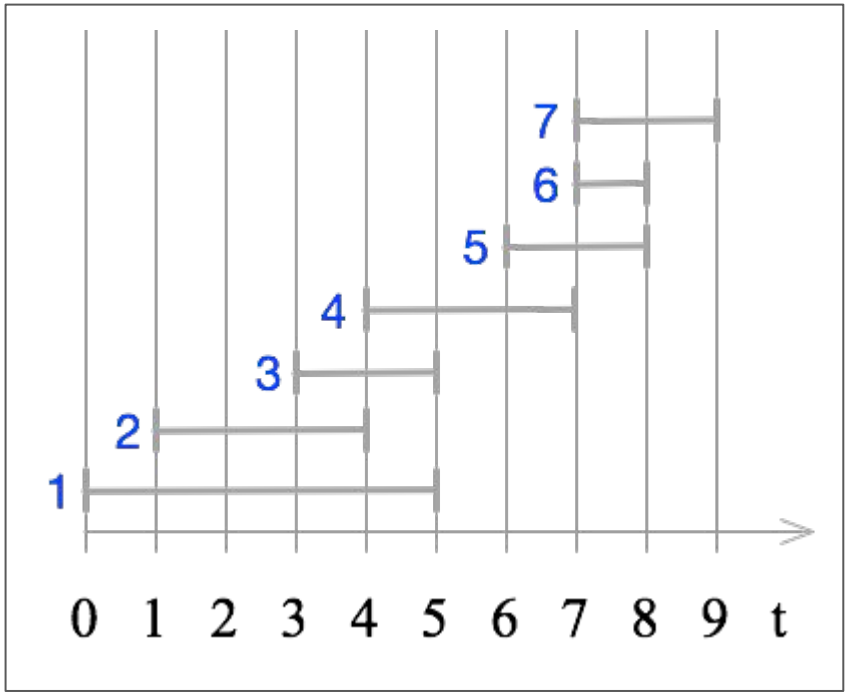
Let's try taking them in order of duration:



i	*s*(*i*)	*f*(*i*)
1	0	5
2	1	4
3	3	5
4	4	7
5	6	8
6	7	8
7	7	9

Less is more?

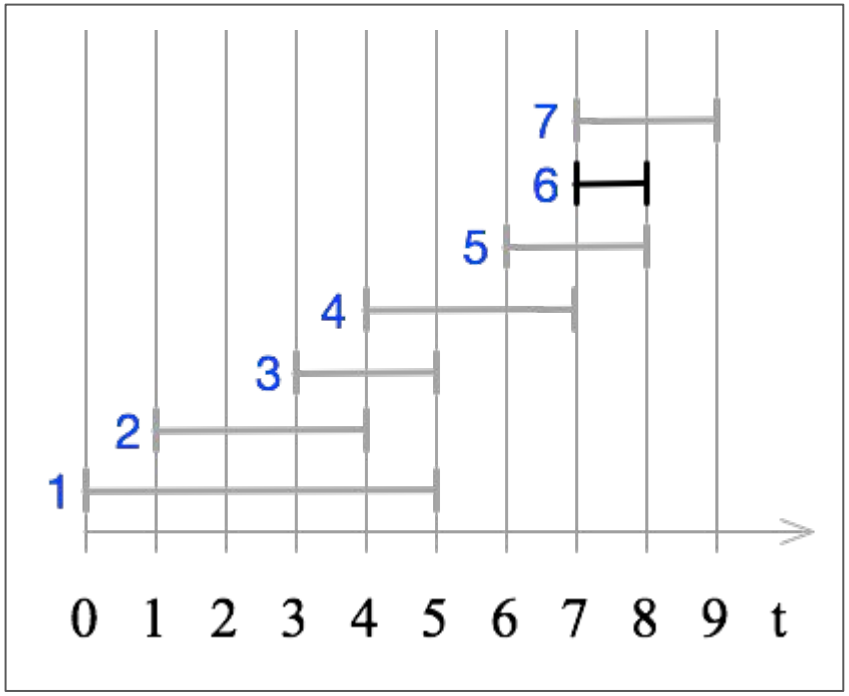
Let's try taking them in order of duration:



i	*s*(*i*)	*f*(*i*)	*dur*
6	7	8	1
3	3	5	2
5	6	8	2
7	7	9	2
2	1	4	3
4	4	7	3
1	0	5	5

Less is more?

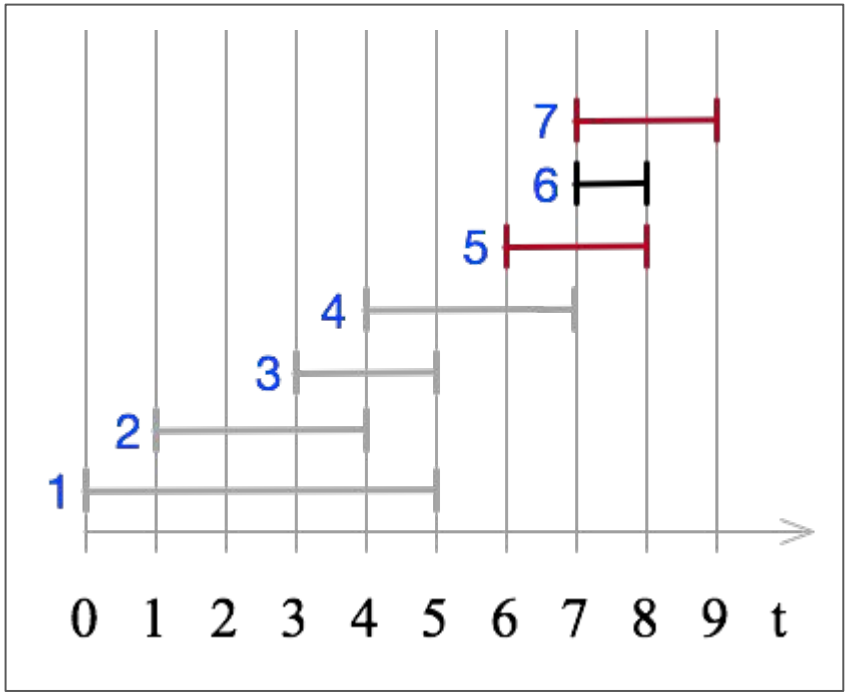
Let's try taking them in order of duration:



i	*s*(*i*)	*f*(*i*)	*dur*
6	**7**	**8**	**1**
3	3	5	2
5	6	8	2
7	7	9	2
2	1	4	3
4	4	7	3
1	0	5	5

Less is more?

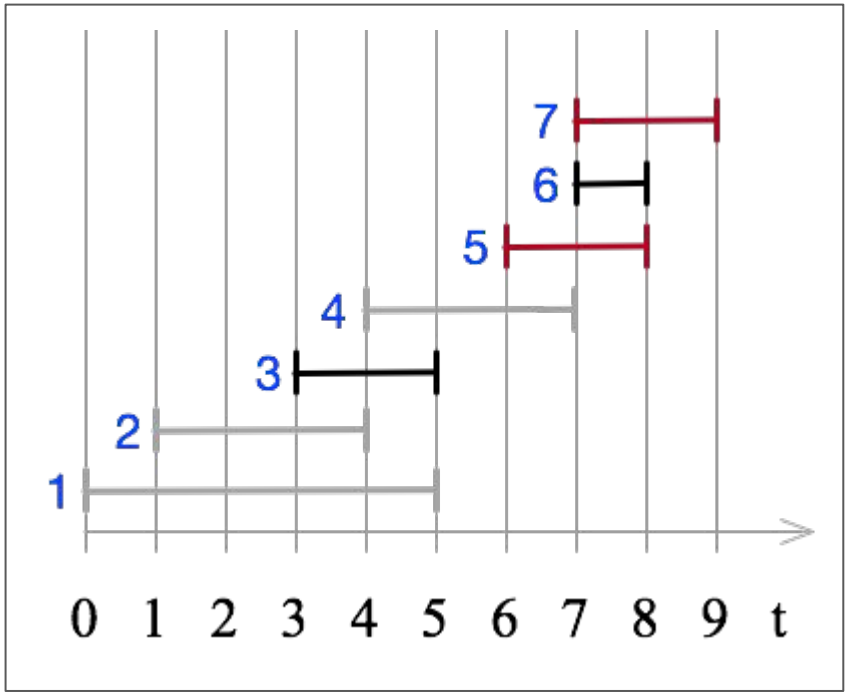
Let's try taking them in order of duration:



i	*s*(*i*)	*f*(*i*)	*dur*
6	**7**	**8**	**1**
3	3	5	2
5	6	8	2
7	7	9	2
2	1	4	3
4	4	7	3
1	0	5	5

Less is more?

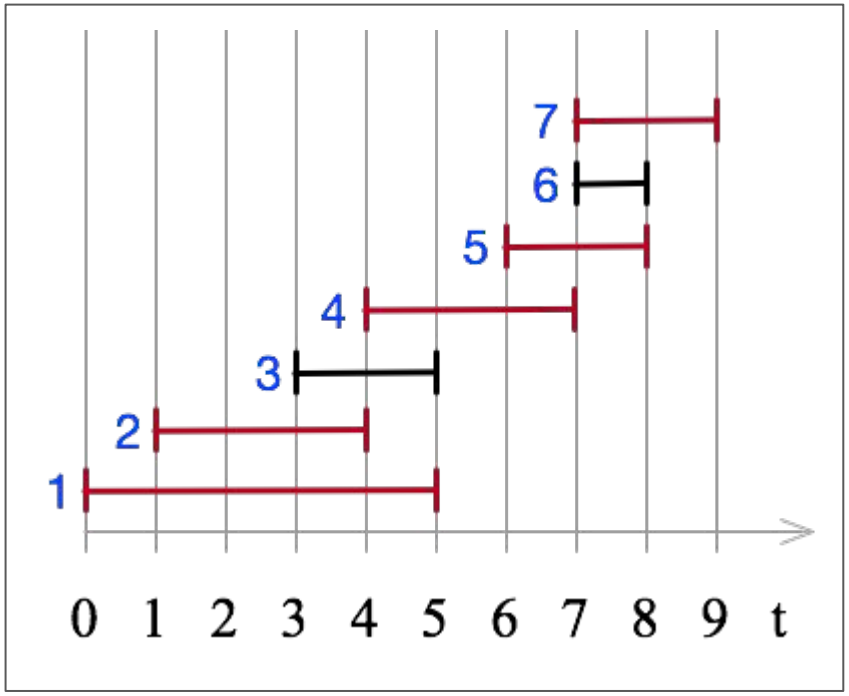
Let's try taking them in order of duration:



i	*s*(*i*)	*f*(*i*)	*dur*
6	7	8	1
3	3	5	2
5	6	8	2
7	7	9	2
2	1	4	3
4	4	7	3
1	0	5	5

Less is more?

Let's try taking them in order of duration:

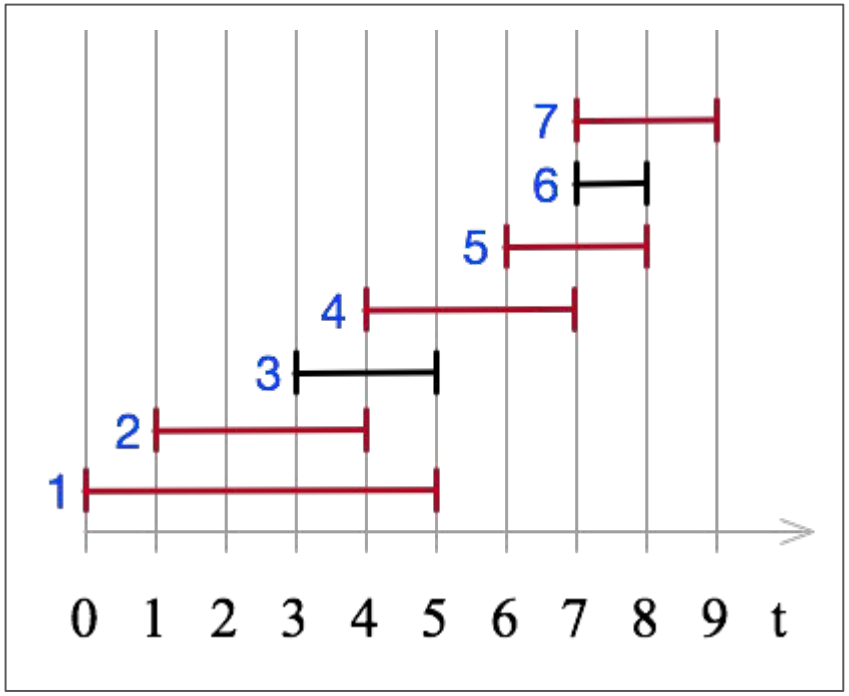


i	*s*(*i*)	*f*(*i*)	*dur*
6	**7**	**8**	**1**
3	**3**	**5**	**2**
5	6	8	2
7	7	9	2
2	1	4	3
4	4	7	3
1	0	5	5

Less is more?

Let's try taking them in order of duration:

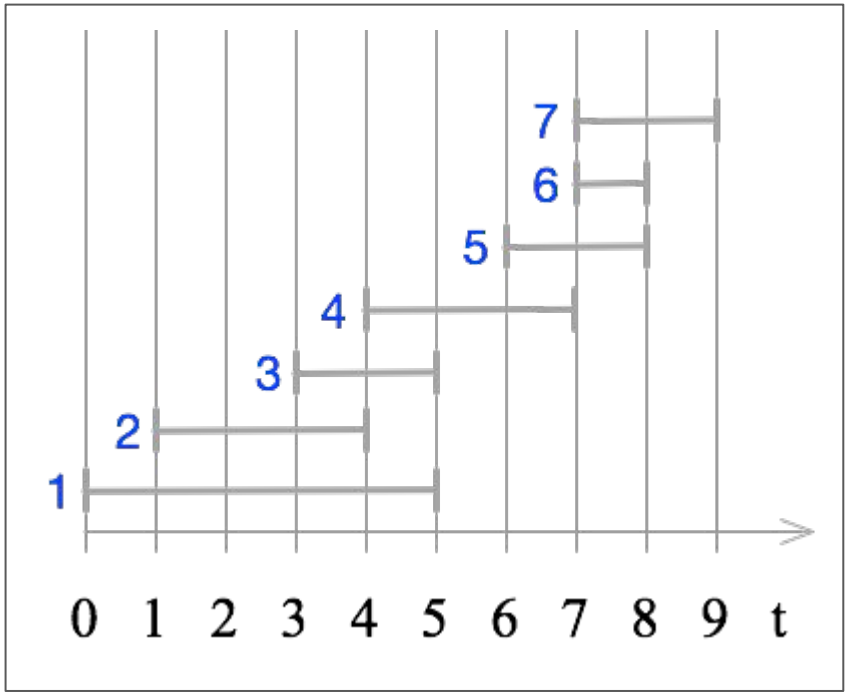
2 requests can be fulfilled



i	*s*(*i*)	*f*(*i*)	*dur*
6	**7**	**8**	**1**
3	**3**	**5**	**2**
5	6	8	2
7	7	9	2
2	1	4	3
4	4	7	3
1	0	5	5

Not so pushy…

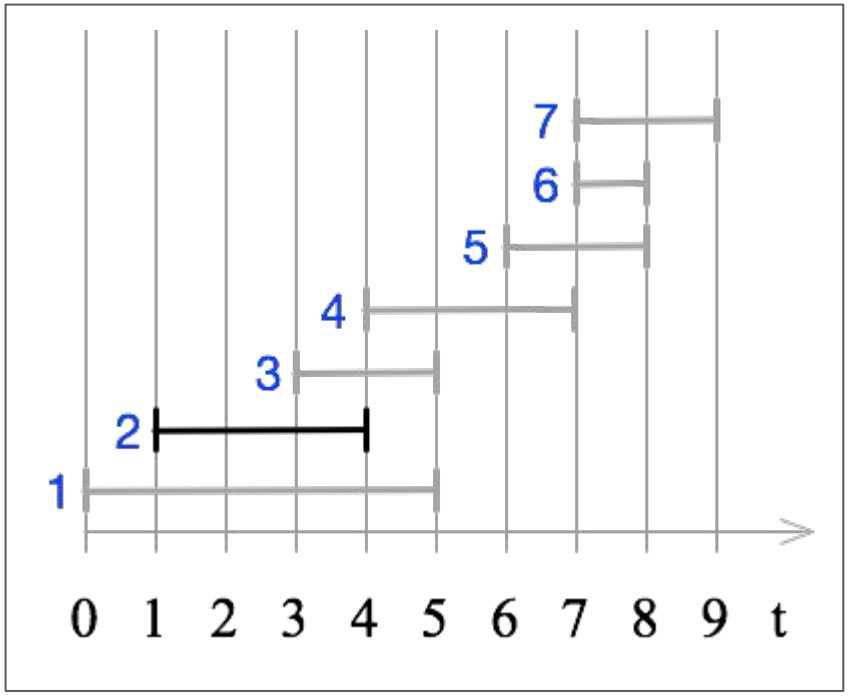
Let's try taking them in order of conflicts/overlaps:



i	*s*(*i*)	*f*(*i*)	*conflicts*
2	1	4	2
6	7	8	2
7	7	9	2
1	0	5	3
3	3	5	3
4	4	7	3
5	6	8	2

Not so pushy…

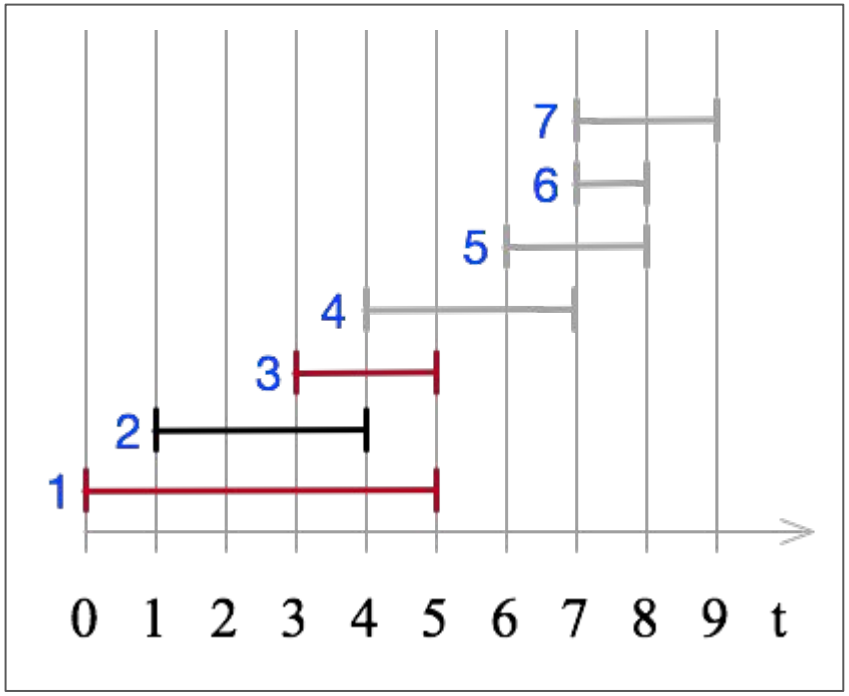
Let's try taking them in order of conflicts/overlaps:



i	*s*(*i*)	*f*(*i*)	*conflicts*
2	**1**	**4**	**2**
6	7	8	2
7	7	9	2
1	0	5	3
3	3	5	3
4	4	7	3
5	6	8	2

Not so pushy…

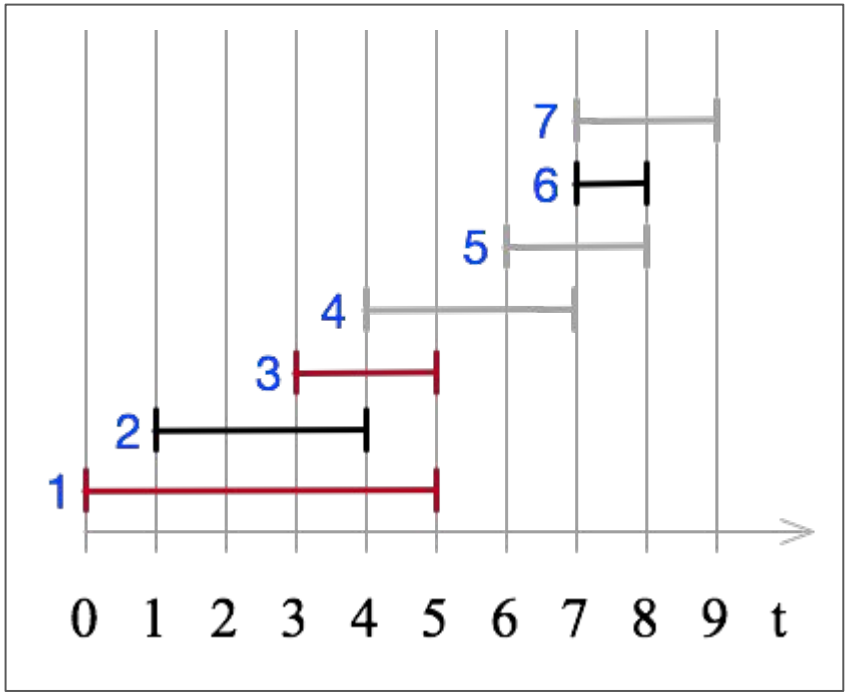
Let's try taking them in order of conflicts/overlaps:



i	*s*(*i*)	*f*(*i*)	*conflicts*
2	**1**	**4**	**2**
6	7	8	2
7	7	9	2
1	0	5	3
3	3	5	3
4	4	7	3
5	6	8	2

Not so pushy…

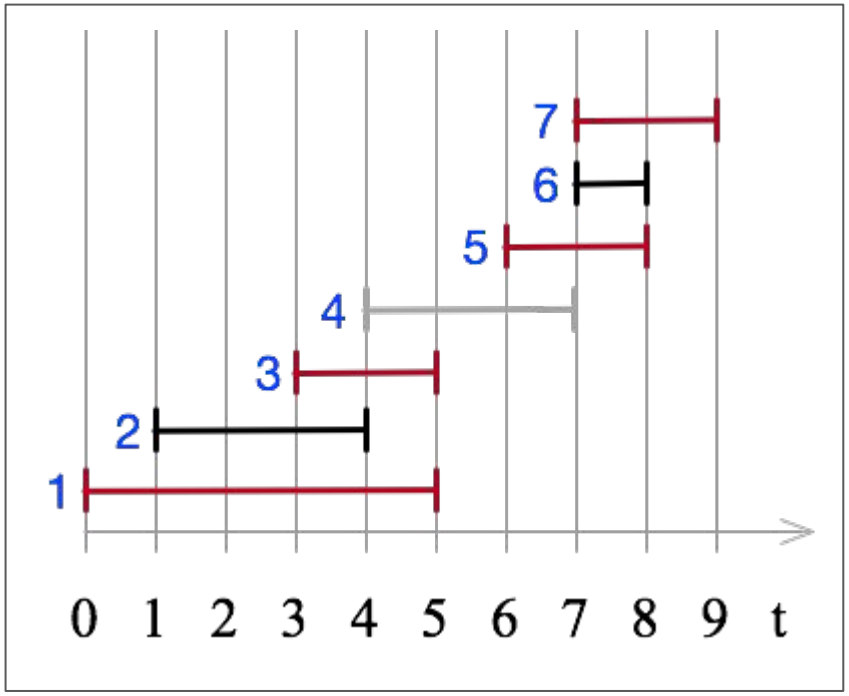
Let's try taking them in order of conflicts/overlaps:



i	*s*(*i*)	*f*(*i*)	*conflicts*
2	**1**	**4**	**2**
6	**7**	**8**	**2**
7	7	9	2
1	0	5	3
3	3	5	3
4	4	7	3
5	6	8	3

Not so pushy…

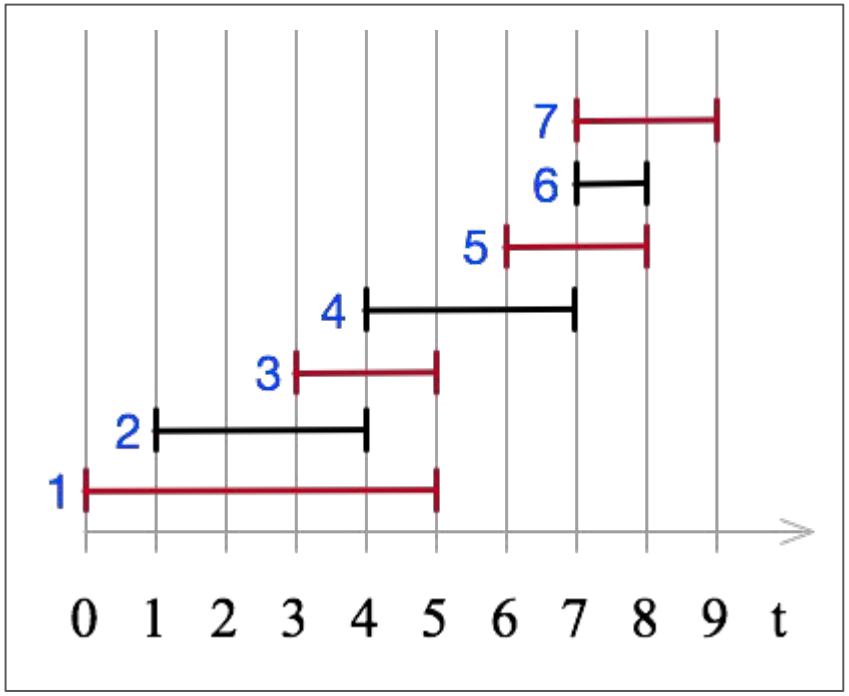
Let's try taking them in order of conflicts/overlaps:



i	*s*(*i*)	*f*(*i*)	*conflicts*
2	1	4	2
6	7	8	2
7	7	9	2
1	0	5	3
3	3	5	3
4	4	7	3
5	6	8	3

Not so pushy…

Let's try taking them in order of conflicts/overlaps:

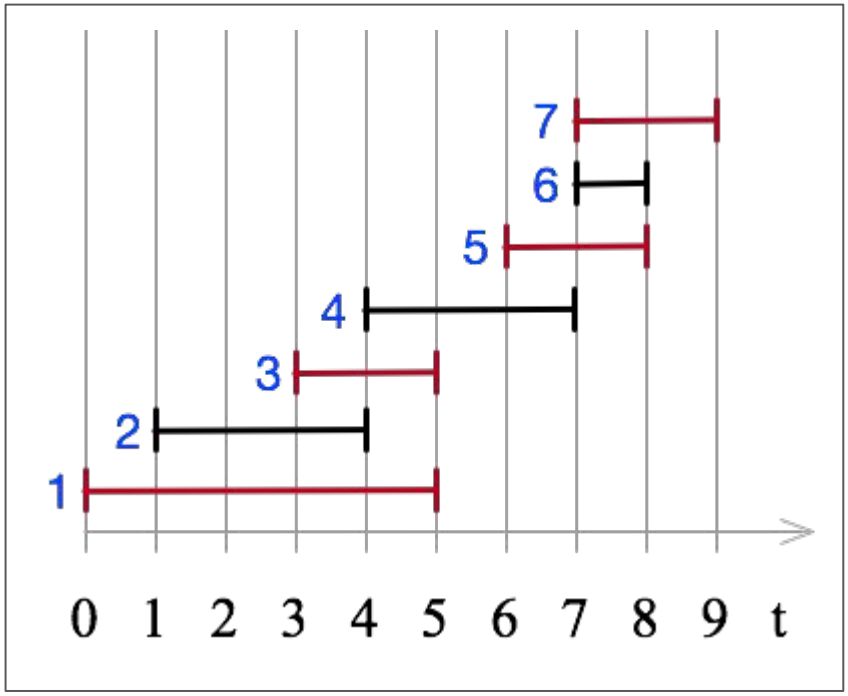


i	*s*(*i*)	*f*(*i*)	*conflicts*
2	**1**	**4**	**2**
6	**7**	**8**	**2**
7	7	9	2
1	0	5	3
3	3	5	3
4	**4**	**7**	**3**
5	6	8	3

Not so pushy…

Let's try taking them in order of conflicts/overlaps:

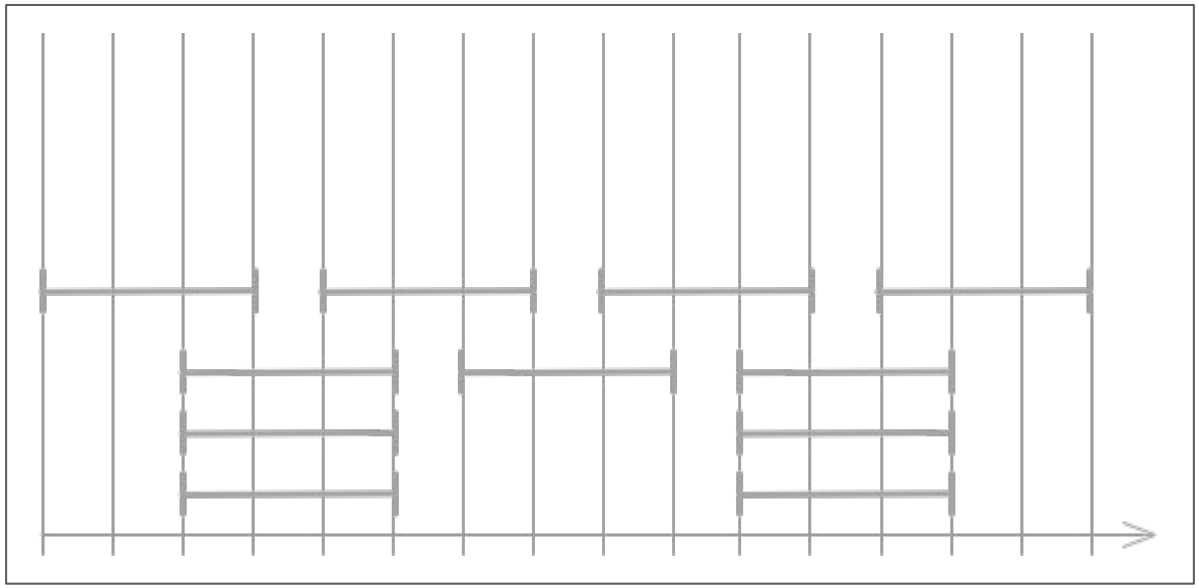
3 requests fulfilled!



i	*s*(*i*)	*f*(*i*)	*conflicts*
2	**1**	**4**	**2**
6	**7**	**8**	**2**
7	7	9	2
1	0	5	3
3	3	5	3
4	**4**	**7**	**3**
5	6	8	3

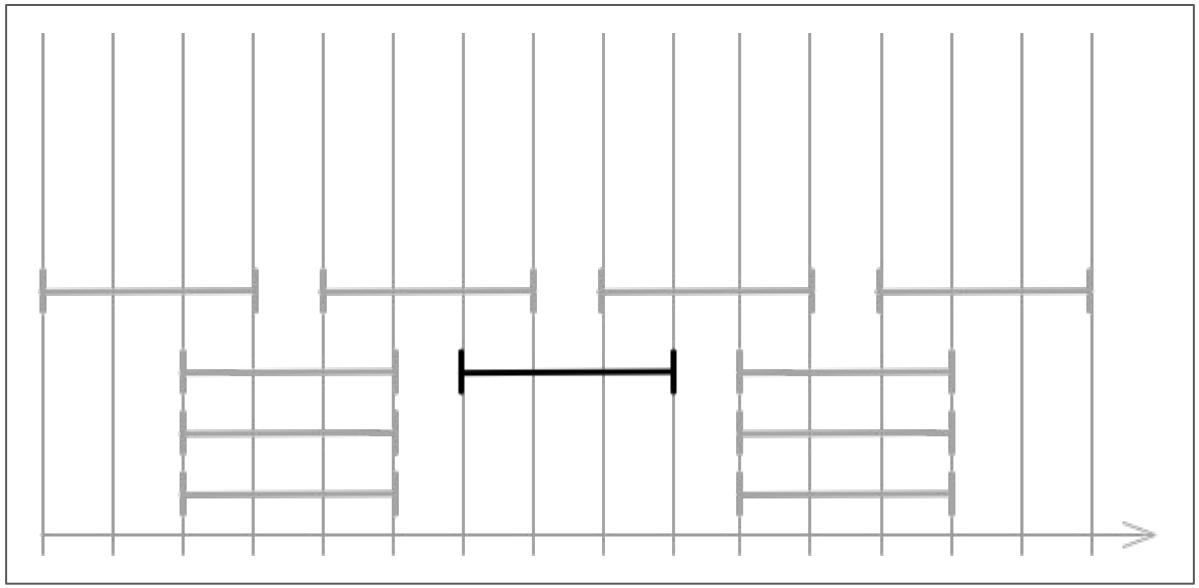
Will this always work?

Let's try taking them in order of conflicts/overlaps:



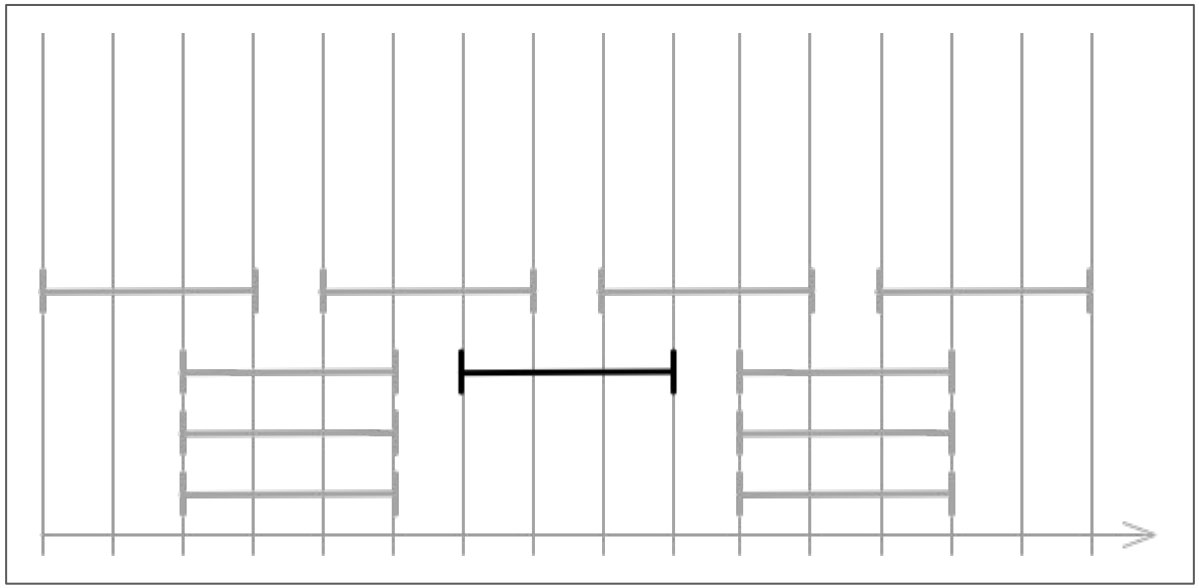
Will this always work?

Let's try taking them in order of conflicts/overlaps:



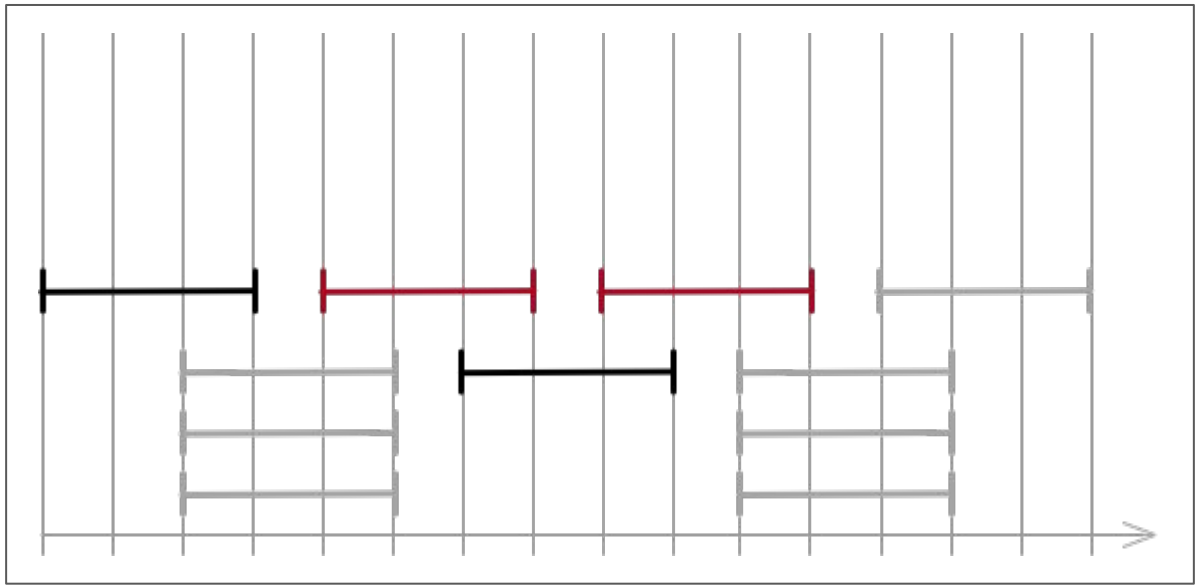
Will this always work?

Let's try taking them in order of conflicts/overlaps:



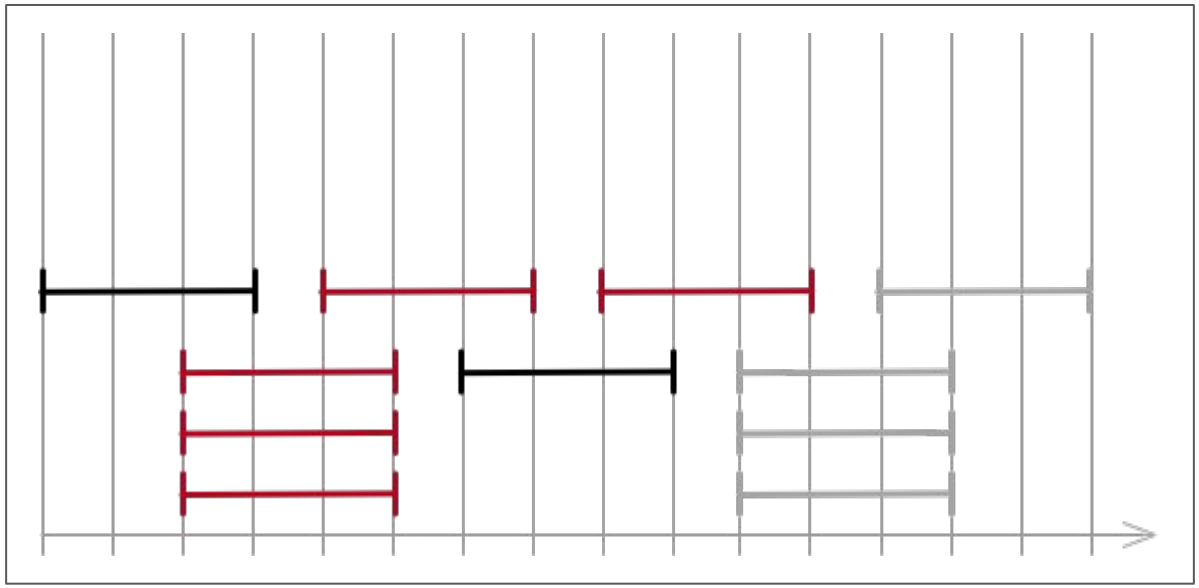
Will this always work?

Let's try taking them in order of conflicts/overlaps:



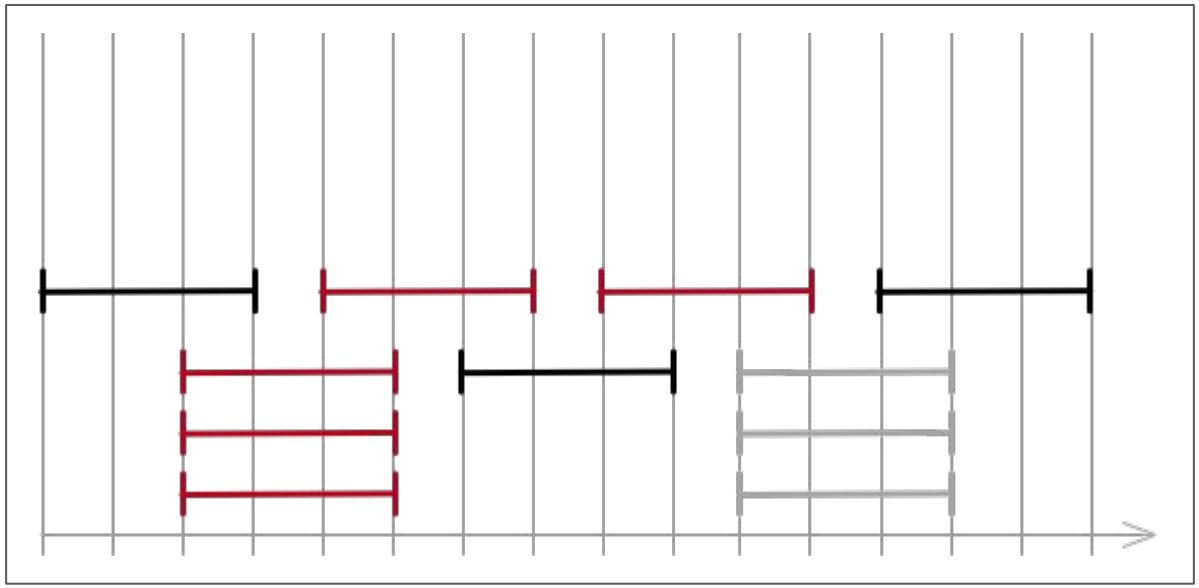
Will this always work?

Let's try taking them in order of conflicts/overlaps:



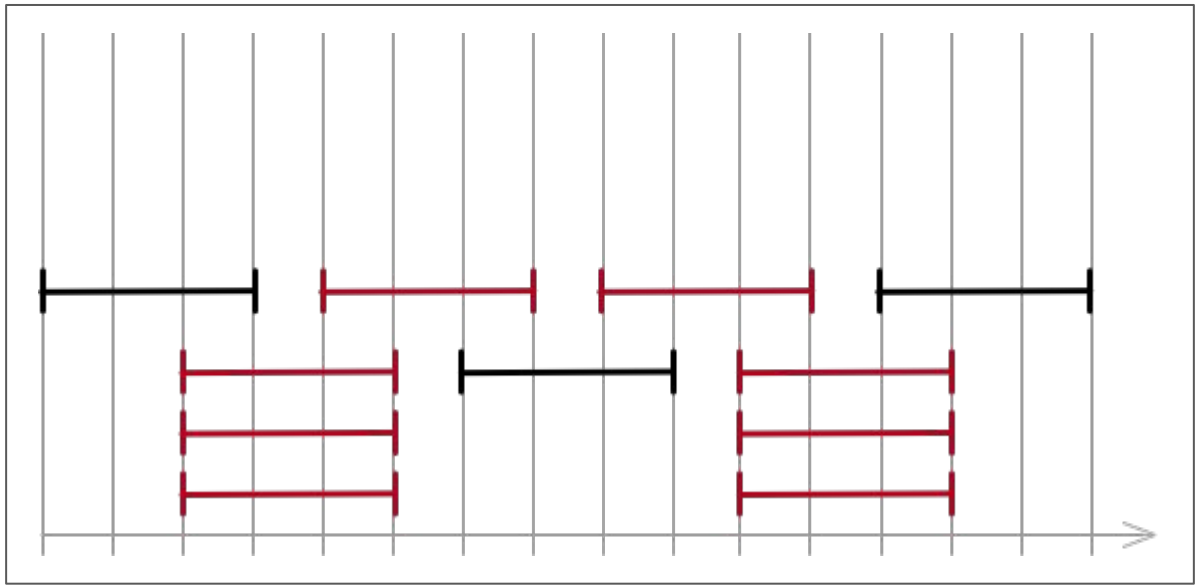
Will this always work?

Let's try taking them in order of conflicts/overlaps:



Will this always work?

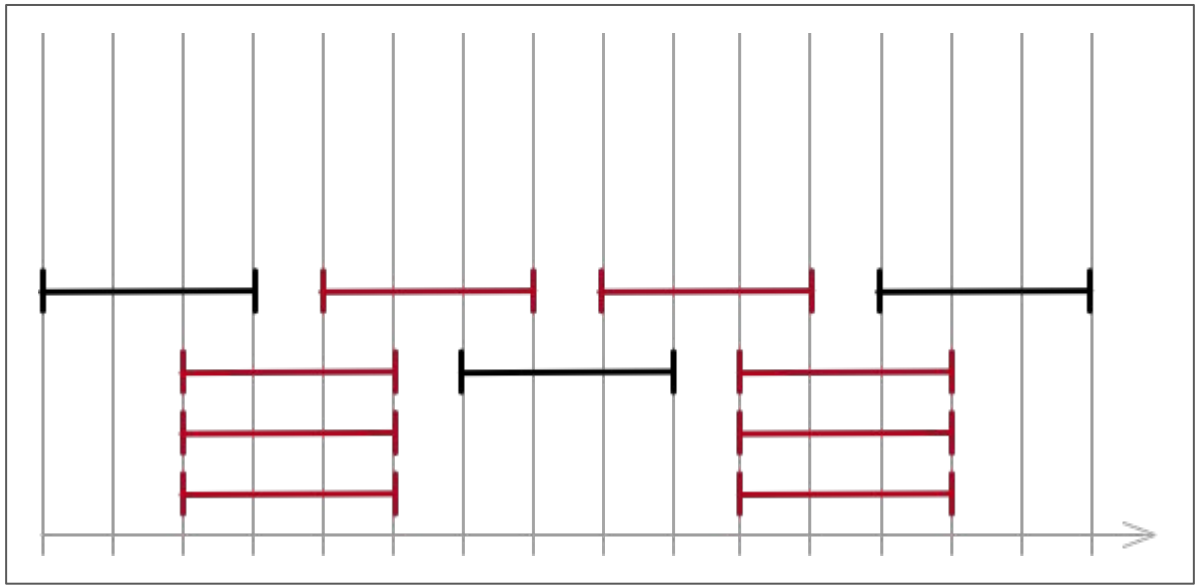
Let's try taking them in order of conflicts/overlaps:



Will this always work?

Let's try taking them in order of conflicts/overlaps:

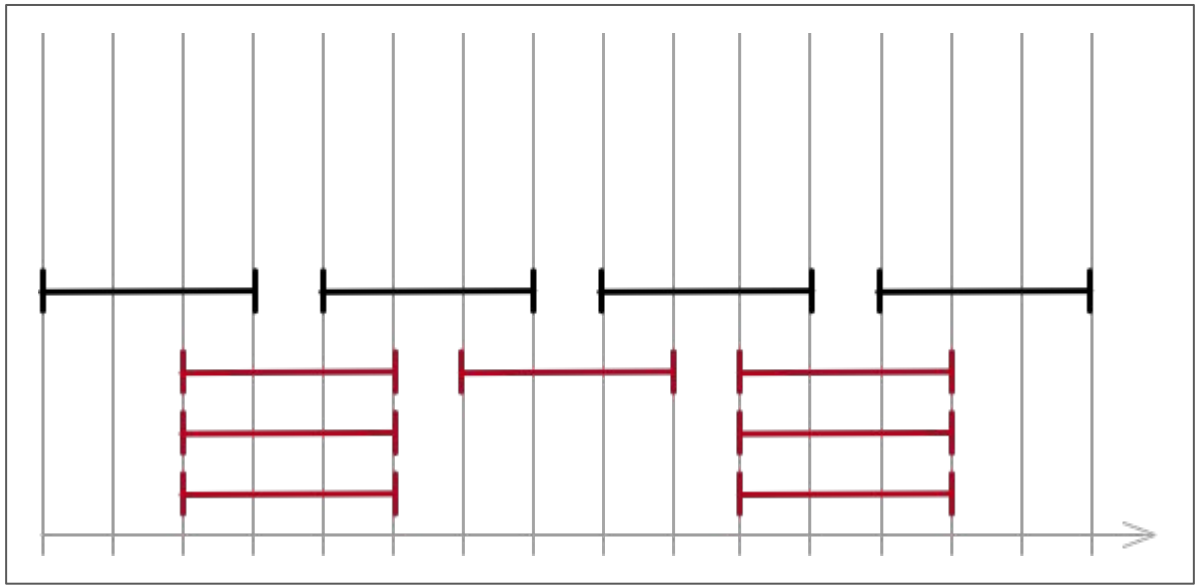
3 requests fulfilled



Will this always work?

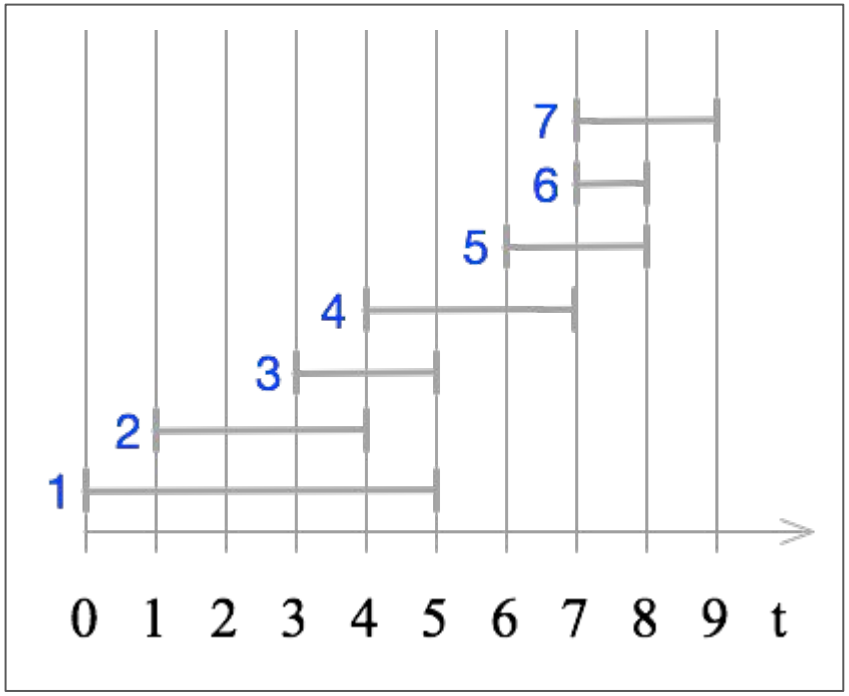
Let's try taking them in order of conflicts/overlaps:

… but **4 requests fulfilled is optimal!**



Okay, the sooner this ends, the better!

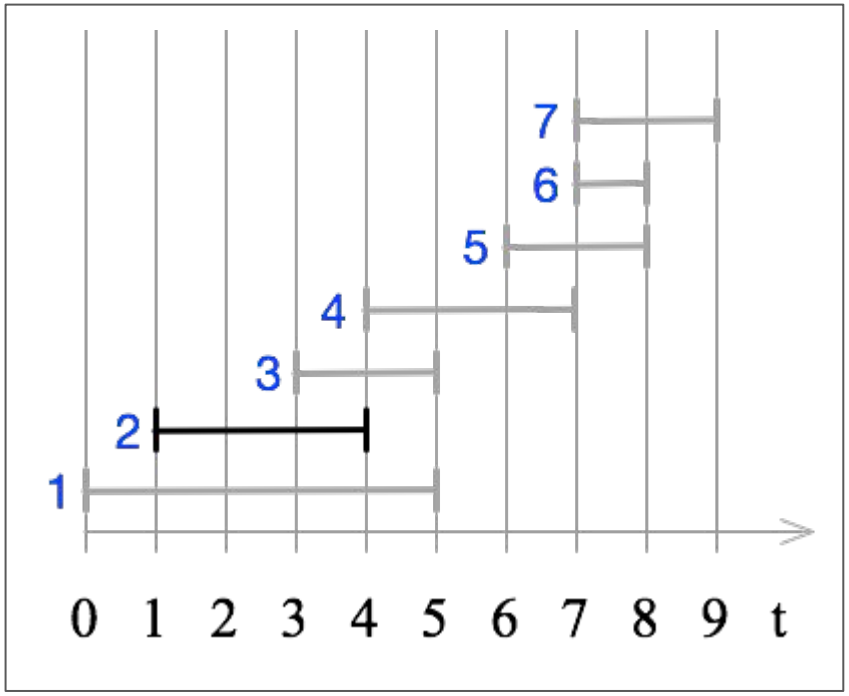
Let's try taking them in order of finish time:



i	*s*(*i*)	*f*(*i*)
2	1	4
1	0	5
3	3	5
4	4	7
5	6	8
6	7	8
7	7	9

Okay, the sooner this ends, the better!

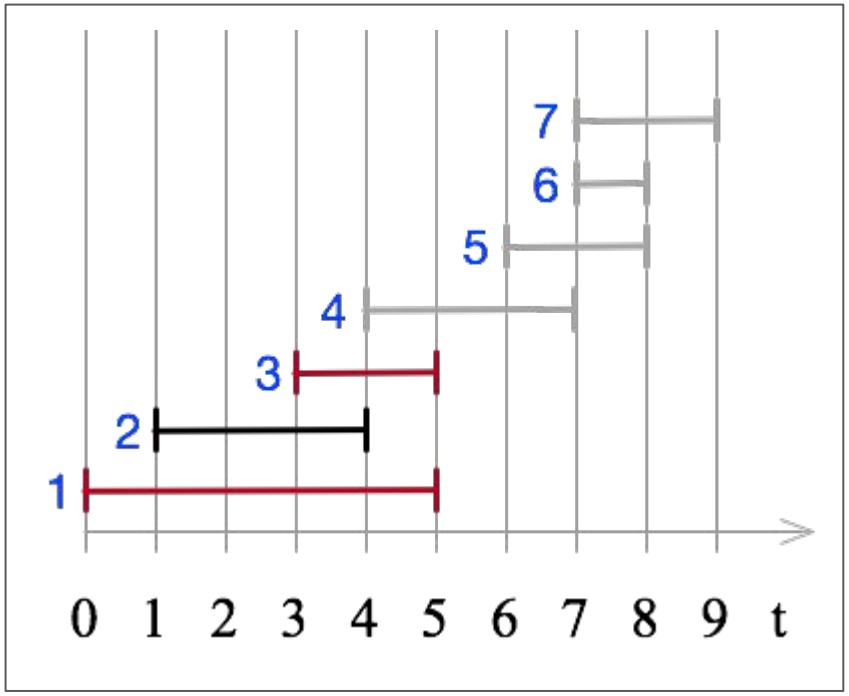
Let's try taking them in order of finish time:



i	*s*(*i*)	*f*(*i*)
2	**1**	**4**
1	0	5
3	3	5
4	4	7
5	6	8
6	7	8
7	7	9

Okay, the sooner this ends, the better!

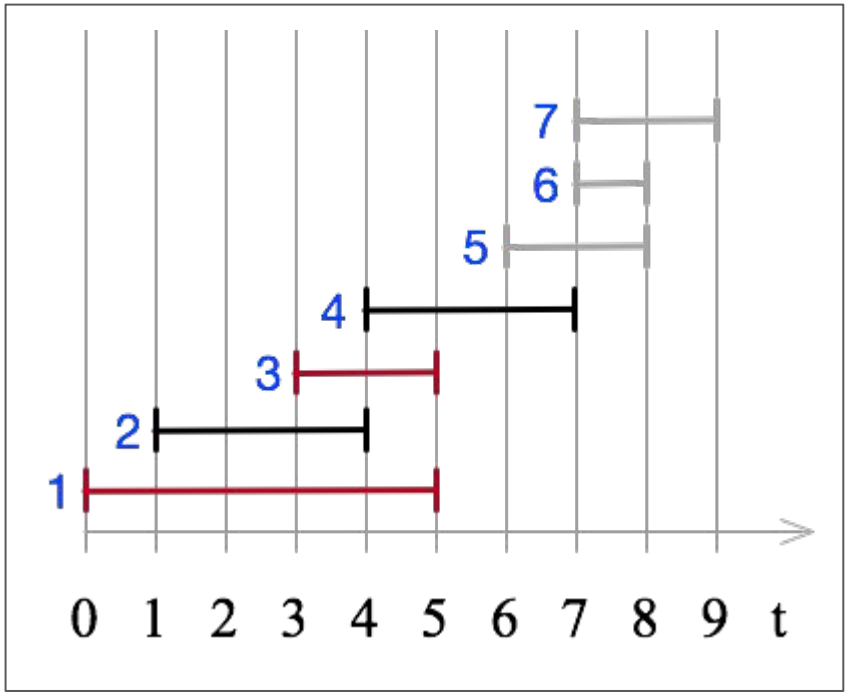
Let's try taking them in order of finish time:



i	*s*(*i*)	*f*(*i*)
2	**1**	**4**
1	0	5
3	3	5
4	4	7
5	6	8
6	7	8
7	7	9

Okay, the sooner this ends, the better!

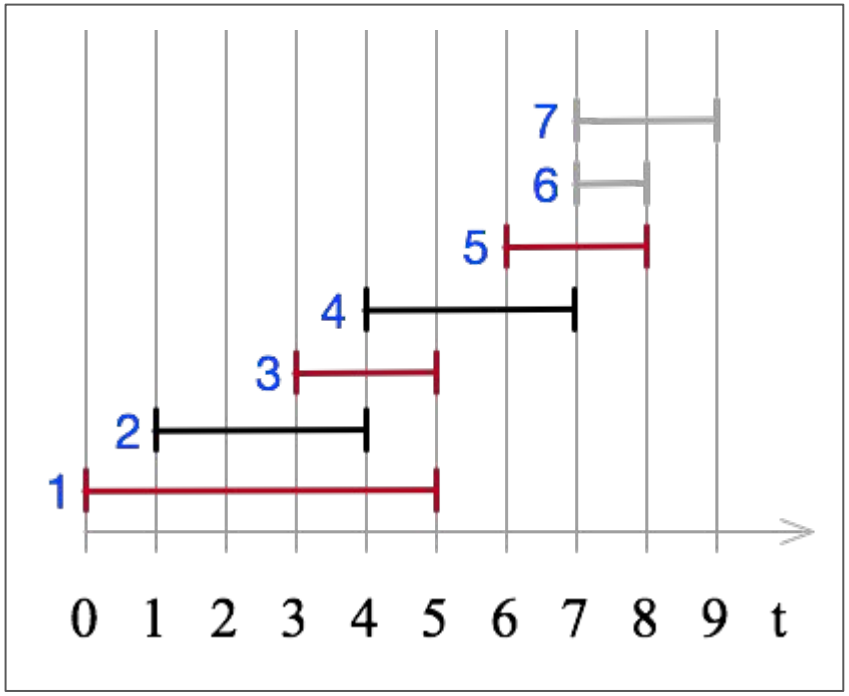
Let's try taking them in order of finish time:



i	*s*(*i*)	*f*(*i*)
2	**1**	**4**
1	0	5
3	3	5
4	**4**	**7**
5	6	8
6	7	8
7	7	9

Okay, the sooner this ends, the better!

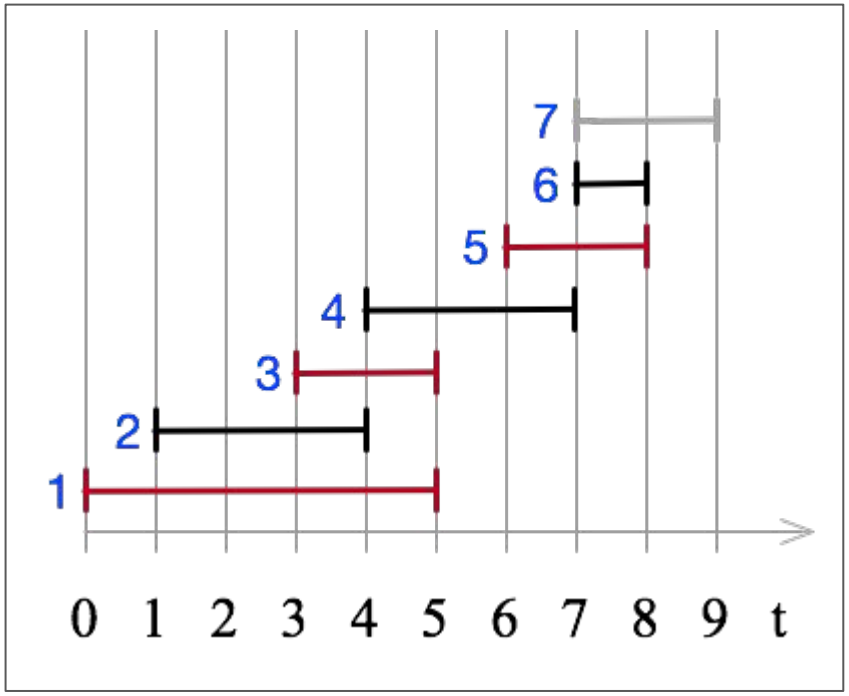
Let's try taking them in order of finish time:



i	*s*(*i*)	*f*(*i*)
2	**1**	**4**
1	0	5
3	3	5
4	**4**	**7**
5	6	8
6	7	8
7	7	9

Okay, the sooner this ends, the better!

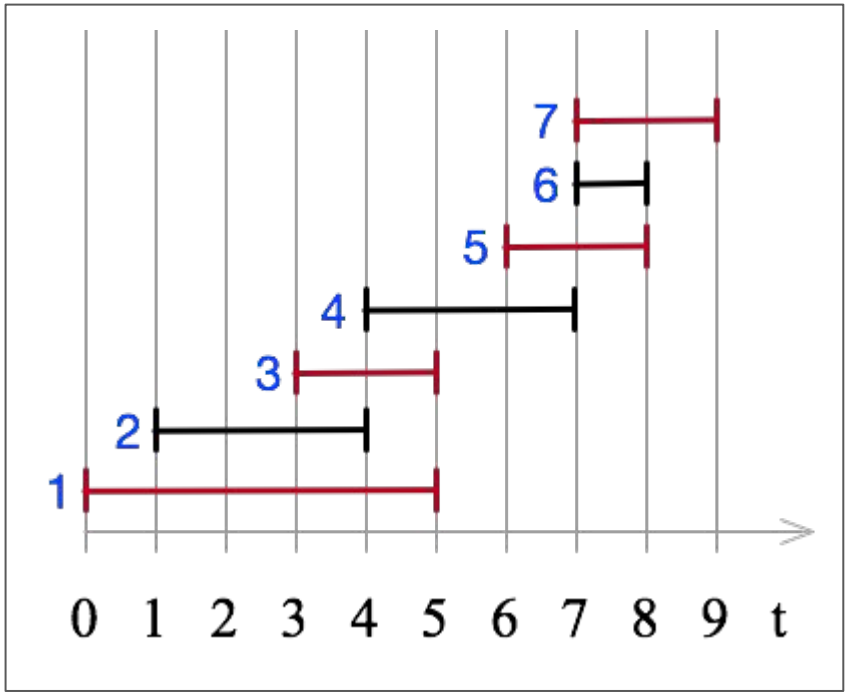
Let's try taking them in order of finish time:



i	*s*(*i*)	*f*(*i*)
2	**1**	**4**
1	0	5
3	3	5
4	**4**	**7**
5	6	8
6	**7**	**8**
7	7	9

Okay, the sooner this ends, the better!

Let's try taking them in order of finish time:

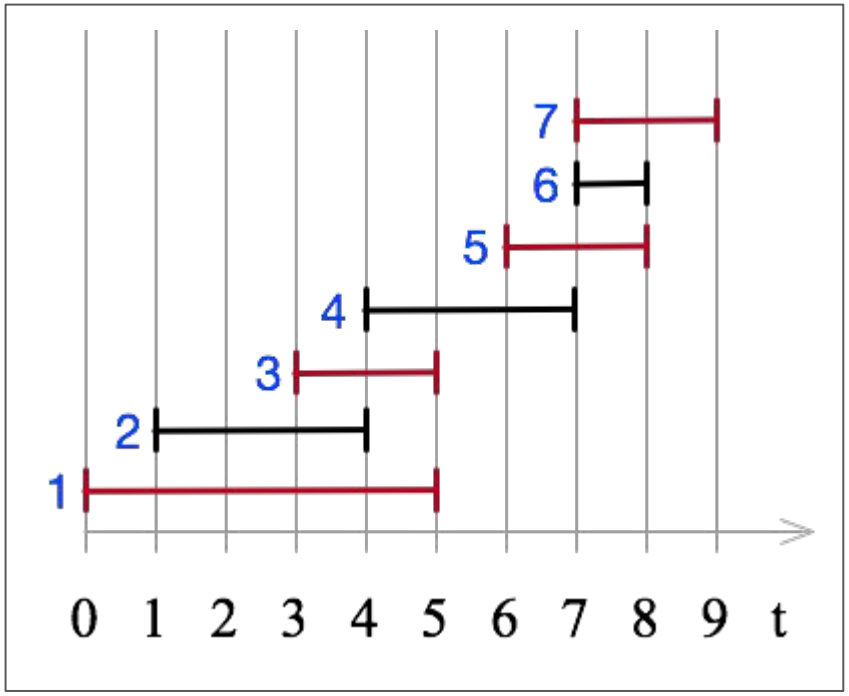


i	*s*(*i*)	*f*(*i*)
2	**1**	**4**
1	0	5
3	3	5
4	**4**	**7**
5	6	8
6	**7**	**8**
7	7	9

Okay, the sooner this ends, the better!

Let's try taking them in order of finish time:

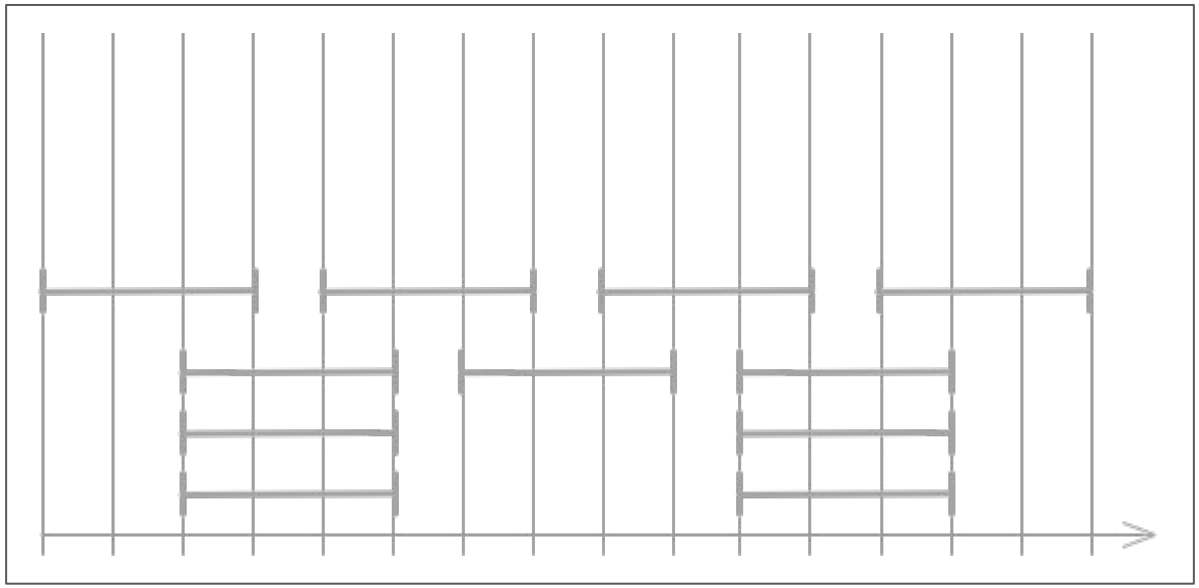
3 requests fulfilled!



i	*s*(*i*)	*f*(*i*)
2	**1**	**4**
1	0	5
3	3	5
4	**4**	**7**
5	6	8
6	**7**	**8**
7	7	9

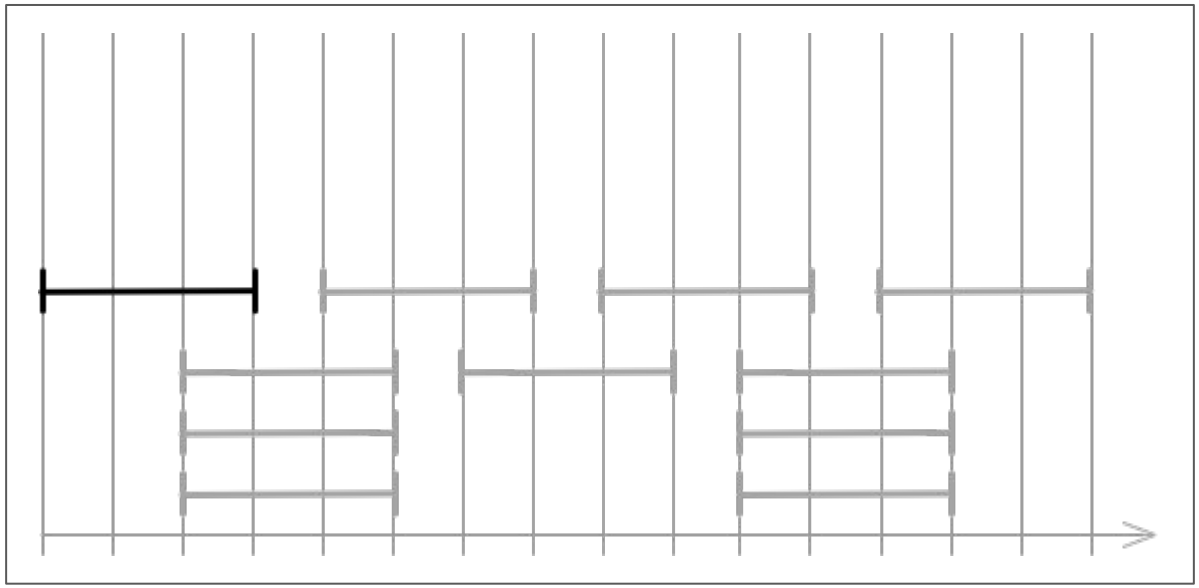
Will this always work?

Let's try taking them in order of finish time:



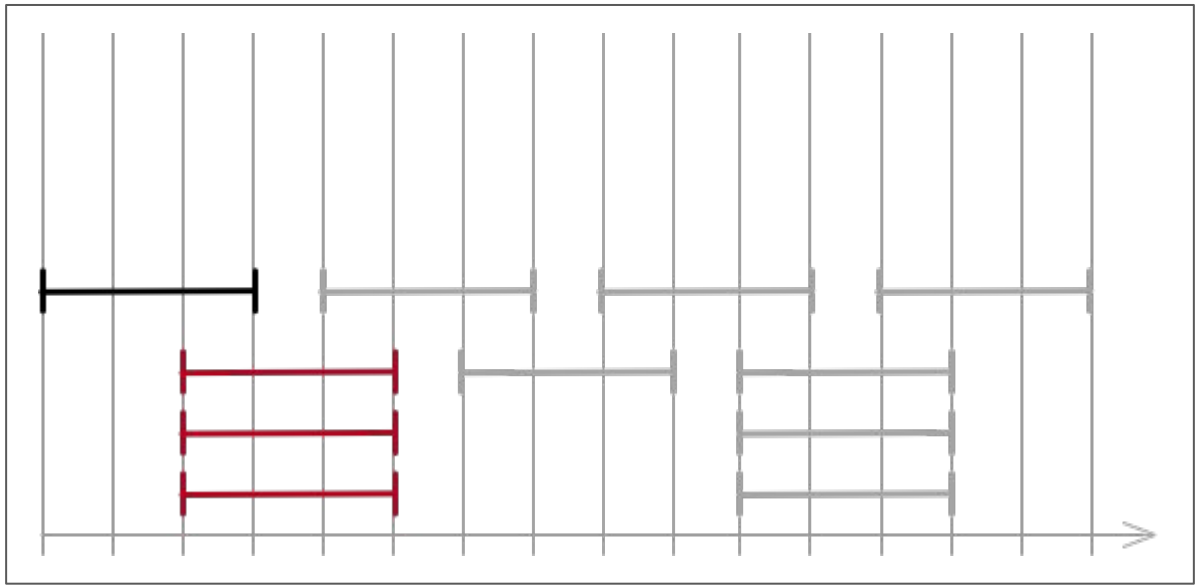
Will this always work?

Let's try taking them in order of finish time:



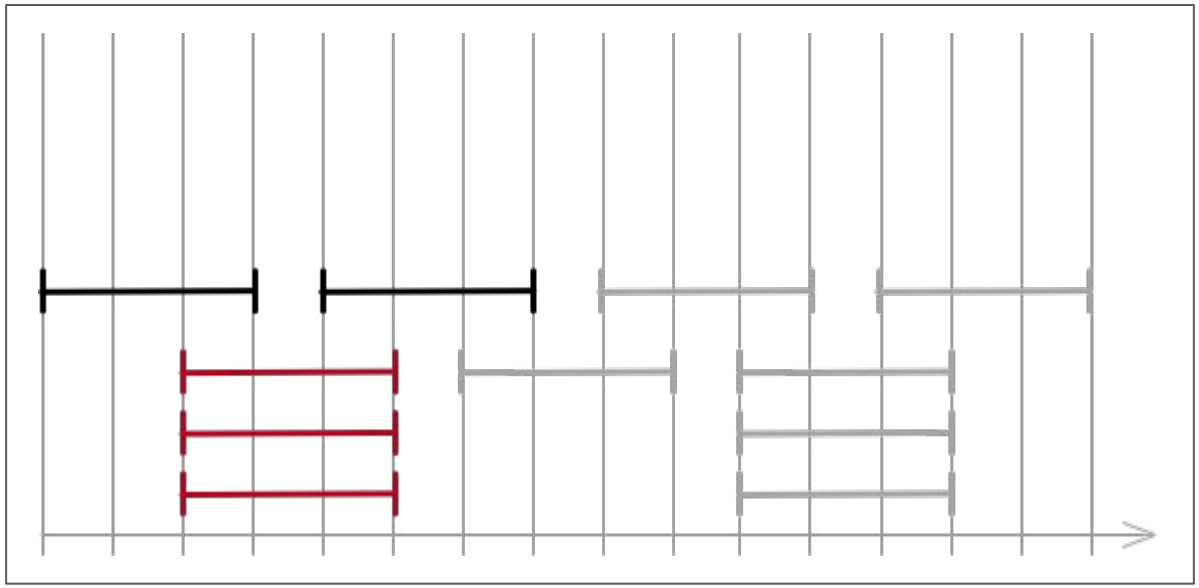
Will this always work?

Let's try taking them in order of finish time:



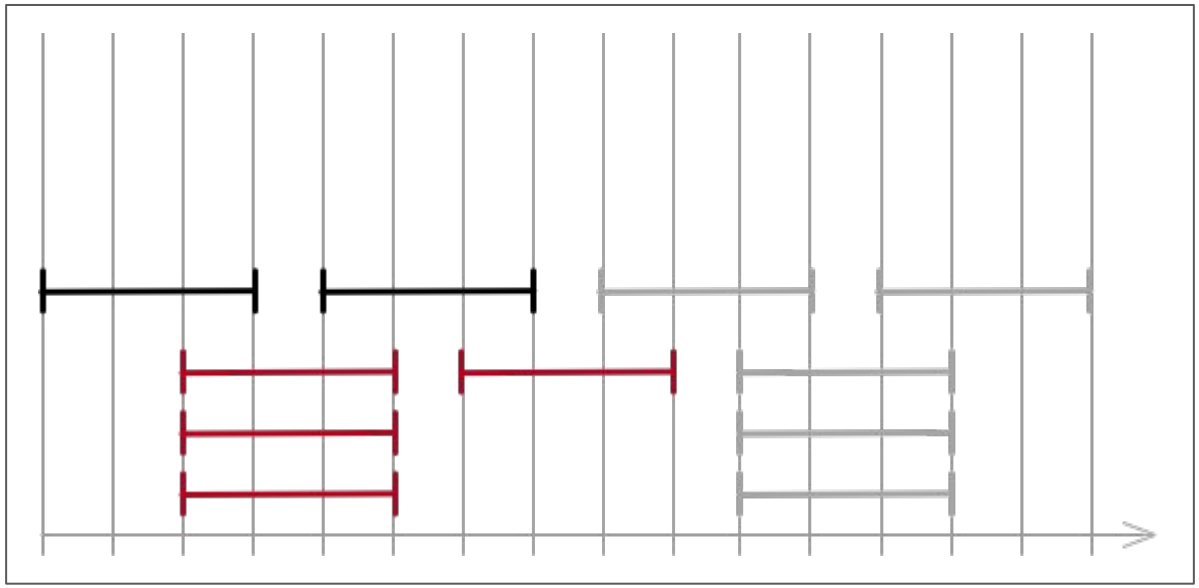
Will this always work?

Let's try taking them in order of finish time:



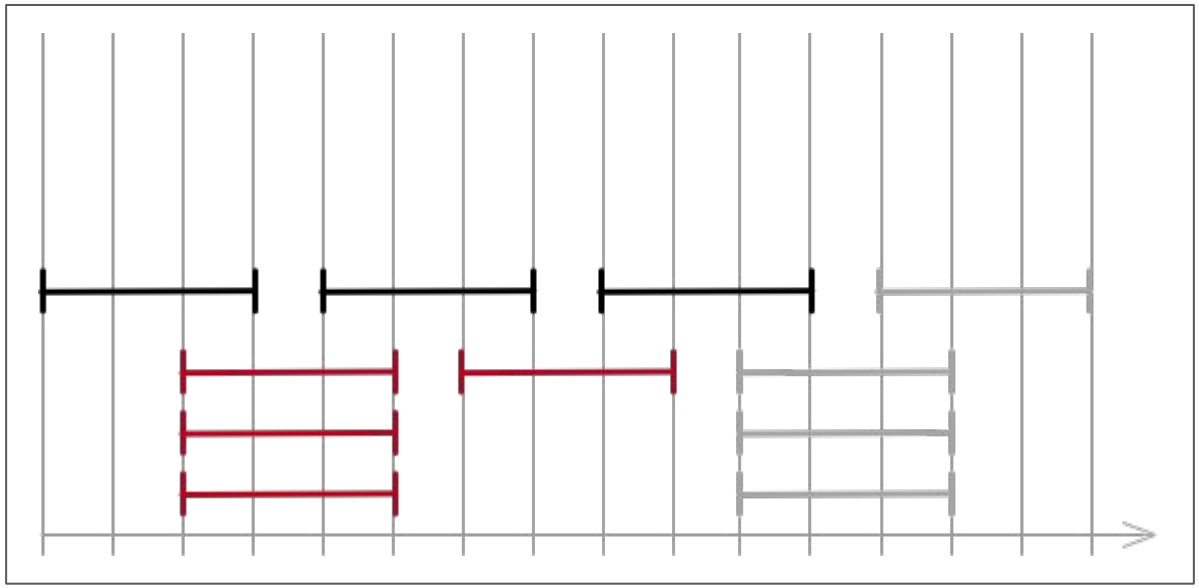
Will this always work?

Let's try taking them in order of finish time:



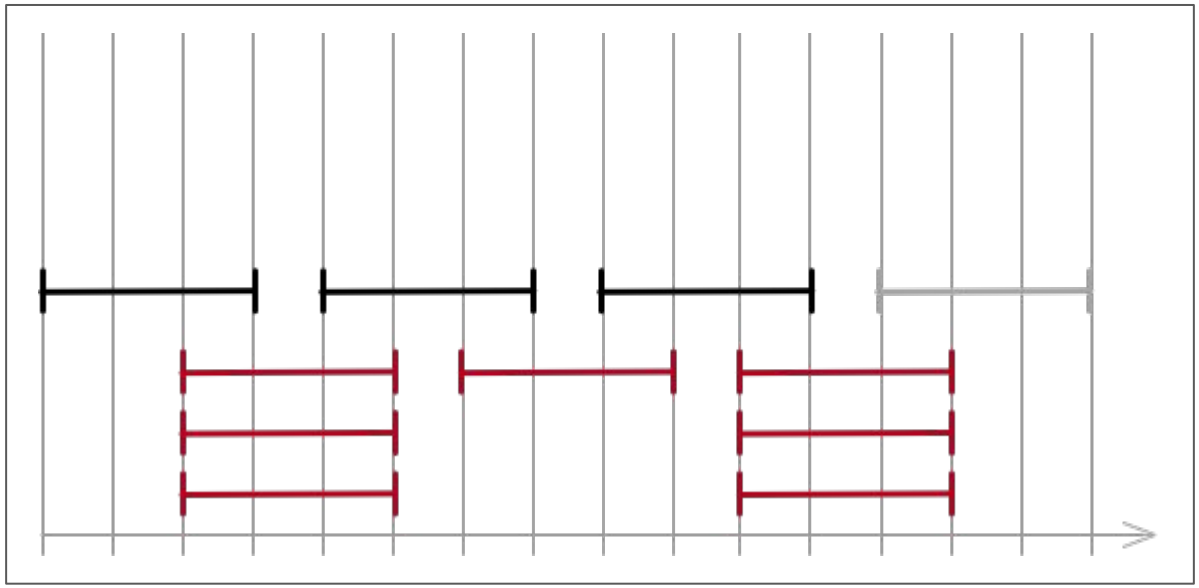
Will this always work?

Let's try taking them in order of finish time:



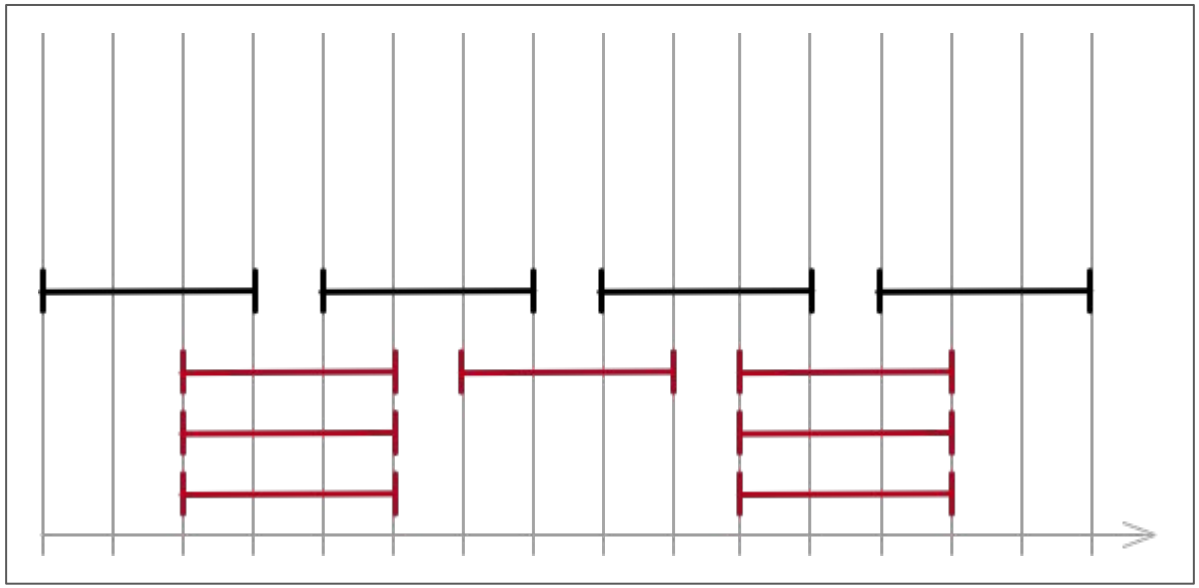
Will this always work?

Let's try taking them in order of finish time:



Will this always work?

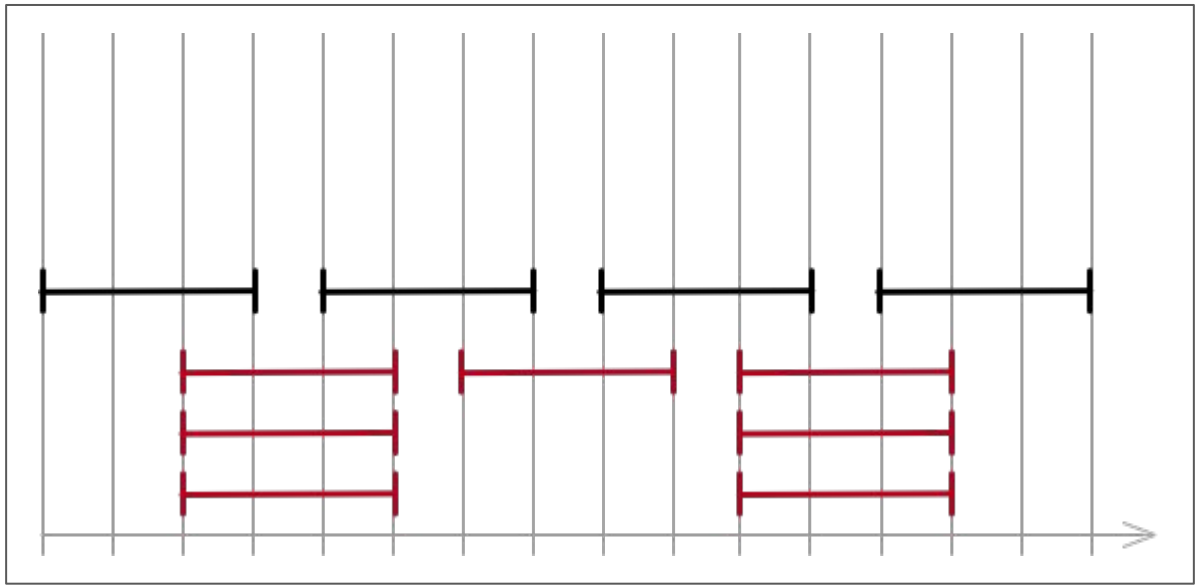
Let's try taking them in order of finish time:



Will this always work?

Let's try taking them in order of finish time:

4 requests fulfilled!



Finish time sorting is optimal

- Recall: **optimal** set
  - Compatible (no overlap)
  - Maximum-sized
- Proof
  - Let *S* be set produced by algorithm using finish time to sort
  - Observe *S* is **compatible** by construction

```
R = all requests sorted by finish time
S = {} // selected requests
while (R is not empty)
    remove next request i from R
    add i to S
    remove all overlapping requests from R
```

How do we prove maximum-sized?

- Let k = # requests selected by algorithm; $|S| = k$
- Let $i_1, i_2, \ldots, i_k$ be indices of selected requests, ordered by start/finish time

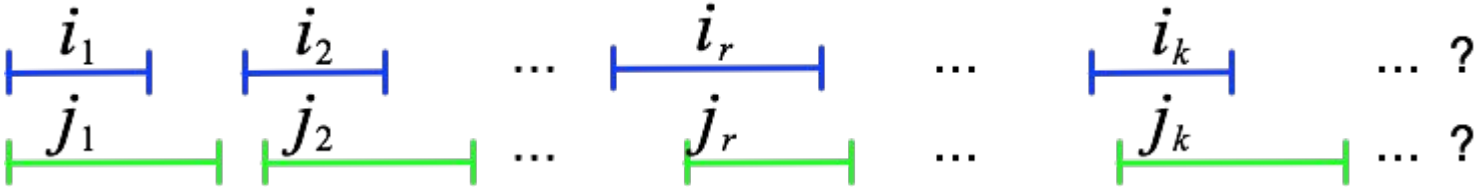
requests are compatible =>
$s(i_r) < f(i_r) \leq s(i_{r+1})$
sorting by start = sorting by finish

How do we prove maximum-sized?

- Let k = # requests selected by algorithm; $|S| = k$
- Let $i_1, i_2, \ldots, i_k$ be indices of selected requests, ordered by start/finish time
- Let O be other compatible set of m requests; $|O| = m$
- Let $j_1, j_2, \ldots, j_m$ be indices of selected requests, ordered by start/finish time

Claim "greedy stays ahead"

$f(i_r) \leq f(j_r)$ for all $r = 1, 2, \ldots$



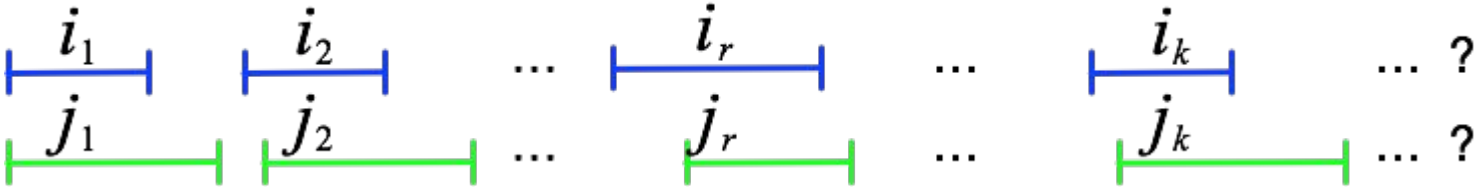
How do we prove maximum-sized?

- Let k = # requests selected by algorithm; $|S| = k$
- Let $i_1, i_2, \ldots, i_k$ be indices of selected requests,
- Let $J_1, J_2, \ldots, J_m$ dices of selected requests,
  ordered by star ish time

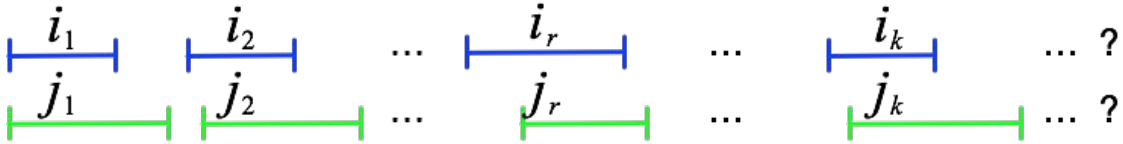
r^{th} request in S finishes no later than r^{th} request in O

Claim "greedy stays ahead"

$f(i_r) \le f(j_r)$ for all $r = 1, 2, \ldots$



Greedy stays ahead



Claim $f(i_r) \leq f(j_r)$ for all $r = 1, 2, \ldots$

Proof by induction on index r

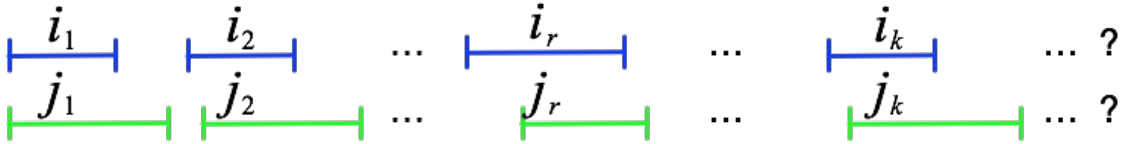
Base case: $r = 1$

- Algorithm chose i_1 to be request with smallest finish => $f(i_1) \leq f(j_1)$

Inductive hypothesis: assume $f(i_{r-1}) \leq f(j_{r-1})$ for index r-1 (>1)

Inductive step: show for index r

Greedy stays ahead



Claim $f(i_r) \leq f(j_r)$ for all $r = 1, 2, \ldots$

Proof by induction on index r

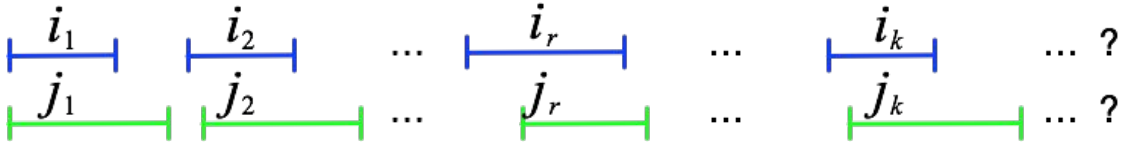
Base case: $r = 1$

- Algorithm chose i_1 to be request with smallest finish => $f(i_1) \leq f(j_1)$

Inductive hypothesis: assume $f(i_{r-1}) \leq f(j_{r-1})$ for index r-1 (>1)

Inductive step: show for index r

Greedy stays ahead



Claim $f(i_r) \leq f(j_r)$ for all $r = 1, 2, \ldots$

Proof by induction on index r

Inductive hypothesis (IH): assume $f(i_{r-1}) \leq f(j_{r-1})$ for index r-1 (>1)

Inductive step: show for index r

- O compatible, so $f(j_{r-1}) \leq s(j_r)$
- By (IH), $f(i_{r-1}) \leq f(j_{r-1})$, so $f(i_{r-1}) \leq s(j_r) \leq f(j_r)$
- Then request j_r compatible with S, so j_r in set R when choosing r^{th} request for S
- Algorithm chooses next request with smallest finish (greedily), so $f(i_r) \leq f(j_r)$

Completing the proof: finish time sorting is optimal

- Claim algorithm outputs an **optimal** set
  - Compatible (no overlap)
  - Maximum-sized

```
R = all requests sorted by finish time
S = {} // selected requests
while (R is not empty)
    remove next request i from R
    add i to S
    remove all overlapping requests from R
```

Completing the proof: finish time sorting is optimal

- Claim algorithm outputs an **optimal** set
  - Compatible (no overlap)
  - Maximum-sized
- Proof
  - S is **compatible** by construction
  - Suppose, for contradiction, there exists O with $|O| > |S|$; $m > k$
    - Since $m > k$, O has request j_{k+1}
    - O compatible => $f(j_k) \leq s(j_{k+1})$
    - Just proved Claim: $f(i_r) \leq f(j_r)$ for all $r = 1, 2, \ldots, k$
      => $f(i_k) \leq f(j_k) \leq s(j_{k+1})$
    - Then requests R included j_{k+1} => algorithm would add at least 1 more request, contradicting size k

Running time analysis

R = all requests sorted by finish time	$O(n \log n)$
S = {} // selected requests	$O(1)$
while (R is not empty)	$O(n)$
remove next request i from R	$O(1)$
add i to S	$O(1)$
remove all overlapping requests from R	$O(n)$

	$O(n^2)$

Can we do better?

R = all requests sorted by finish time	$O(n \log n)$
S = {} // selected requests	$O(1)$
while (R is not empty)	$O(n)$
remove next request i from R	$O(1)$
add i to S	$O(1)$
remove all overlapping requests from R	**$O(n)$**

	$O(n^2)$

Can we do better?

R = all requests sorted by finish time	$O(n \log n)$
S = {} // selected requests	$O(1)$
while (R is not empty)	$O(n)$
remove next request i from R	$O(1)$
add i to S	$O(1)$
~~remove all overlapping requests from R~~	~~$O(n)$~~

	$O(n^2)$

Can we do better?

R = all requests sorted by finish time	$O(n \log n)$
S = {} // selected requests	$O(1)$
current_finish_time = 0	**$O(1)$**
while (R is not empty)	$O(n)$
remove next request i from R	$O(1)$
if $s(i)$ >= current_finish_time	**$O(1)$**
add i to S	$O(1)$
current_finish_time = $f(i)$	**$O(1)$**

	$O(n \log n)$

Can we do even better?

```
R = all requests sorted by finish time          O(n log n)
S = {} // selected requests                     O(1)
current_finish_time = 0                         O(1)
while (R is not empty)                          O(n)
    remove next request i from R                    O(1)
    if s(i) >= current_finish_time                  O(1)
        add i to S                                      O(1)
        current_finish_time = f(i)                      O(1)
                                                ----------
                                                O(n log n)
```

Can we do even better?

R = all requests sorted by finish time	**$\Theta(n \log n)$**
S = {} // selected requests	$O(1)$
current_finish_time = 0	$O(1)$
while (R is not empty)	$O(n)$
remove next request i from R	$O(1)$
if $s(i)$ >= current_finish_time	$O(1)$
add i to S	$O(1)$
current_finish_time = $f(i)$	$O(1)$

	$\Theta(n \log n)$

Greedy algorithm design

- Build up set one item at a time
- Choose next item “greedily”
- No going back

How do we know when it works?

- Proof techniques
  - Greedy stays ahead
  - Exchange argument

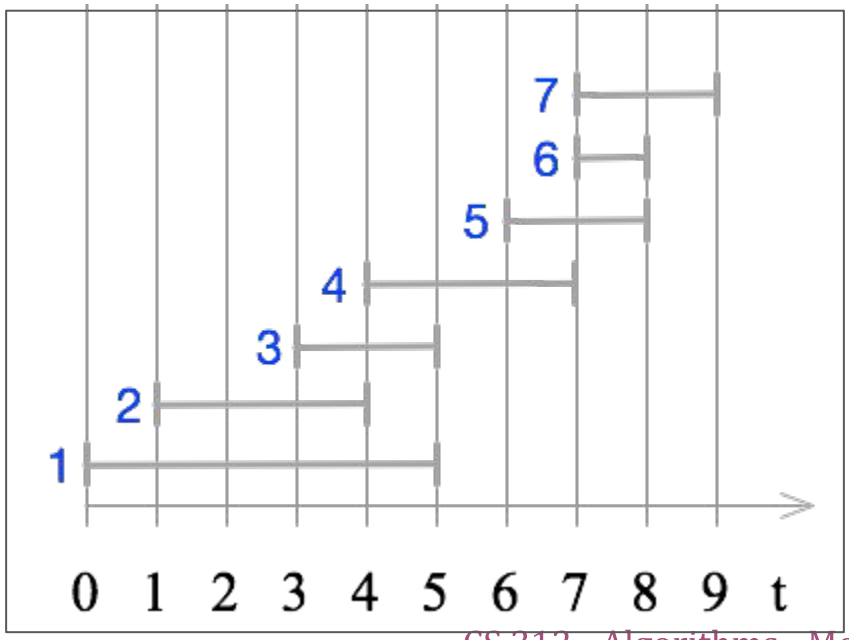
Where to put the talks?

Senior symposium is coming up! As the organizer, you must assign rooms to the talks. Of course, you can't schedule two talks in the same room at one time!

How do you assign talks to rooms so that you minimize the total number of rooms used for the event?

Problem 2: interval partitioning

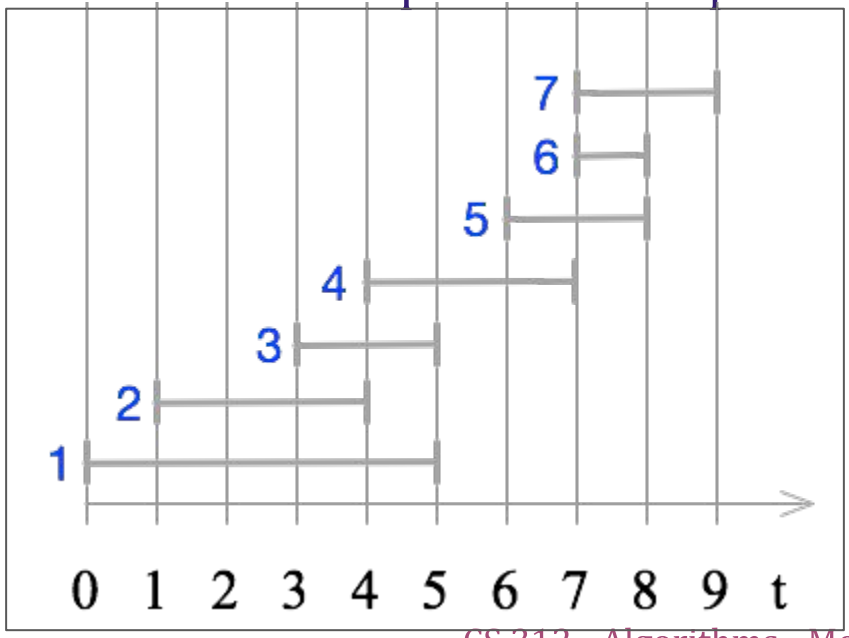
- Set of 1, …, *n* requests
- *s(i)*, *f(i)* start and finish times



i	*s*(*i*)	*f*(*i*)
1	0	5
2	1	4
3	3	5
4	4	7
5	6	8
6	7	8
7	7	9

Greedy algorithm for interval partitioning

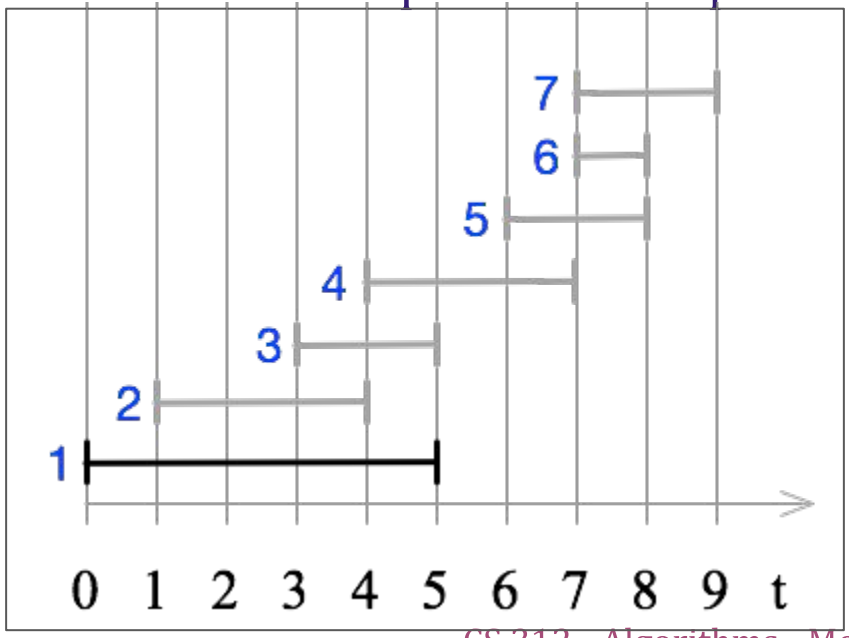
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - reuse previously allocated resource if its last request has completed



i	*s*(*i*)	*f*(*i*)	
1	0	5	
2	1	4	
3	3	5	
4	4	7	
5	6	8	
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

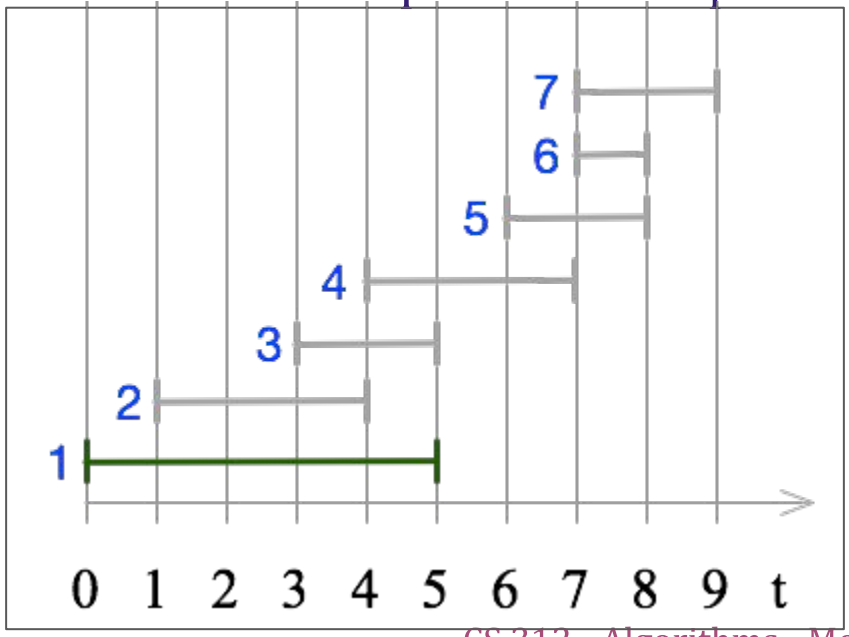
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - reuse previously allocated resource if its last request has completed



i	*s*(*i*)	*f*(*i*)	
1	**0**	**5**	
2	1	4	
3	3	5	
4	4	7	
5	6	8	
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

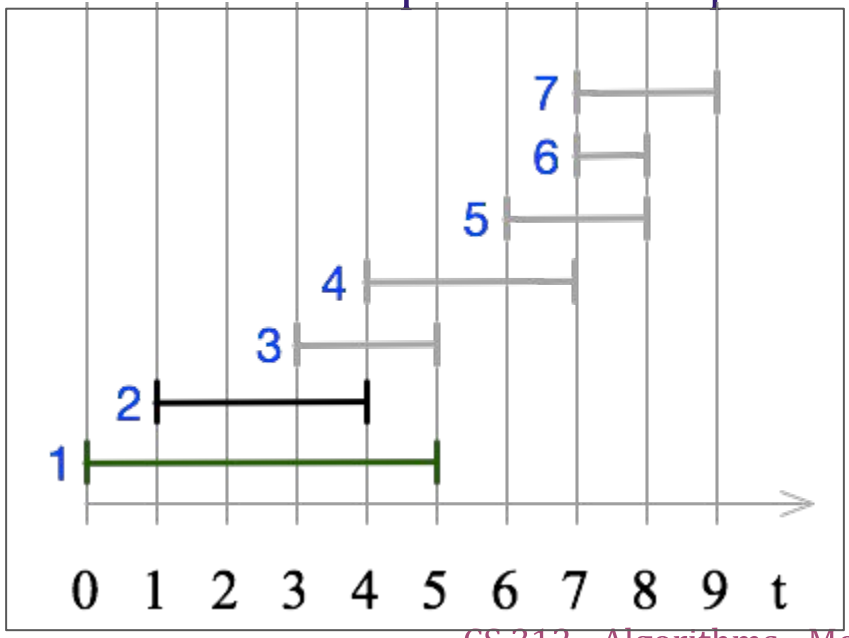
- Process requests ordered by start time
- For each request
  - **allocate a new resource**, or
  - reuse previously allocated resource if its last request has completed



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	
3	3	5	
4	4	7	
5	6	8	
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

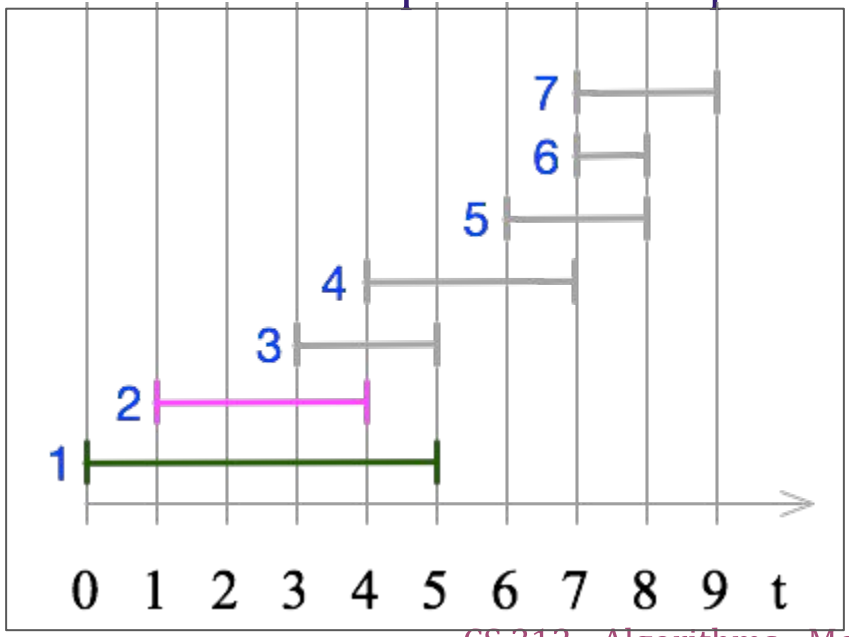
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - reuse previously allocated resource if its last request has completed



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	**1**	**4**	
3	3	5	
4	4	7	
5	6	8	
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

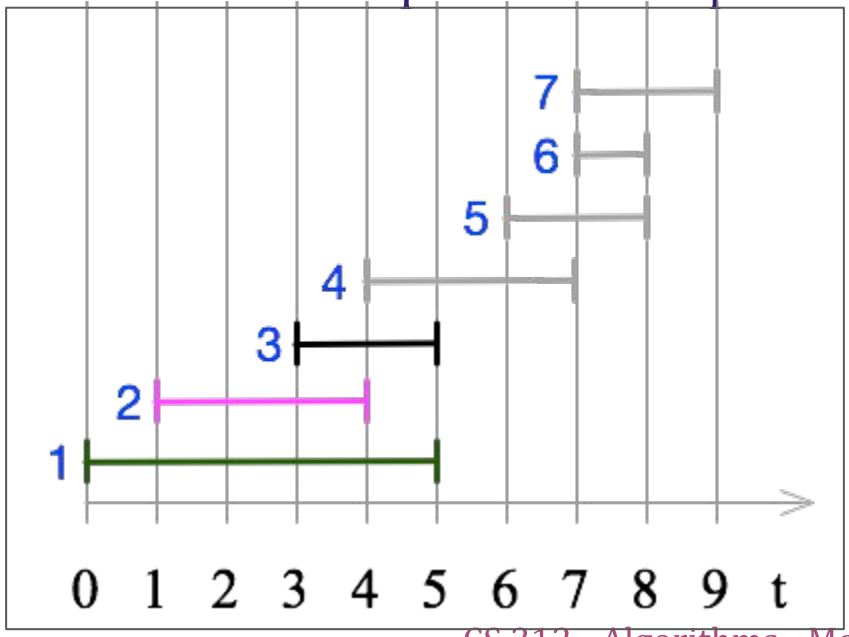
- Process requests ordered by start time
- For each request
  - **allocate a new resource**, or
  - reuse previously allocated resource if its last request has completed



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	
4	4	7	
5	6	8	
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

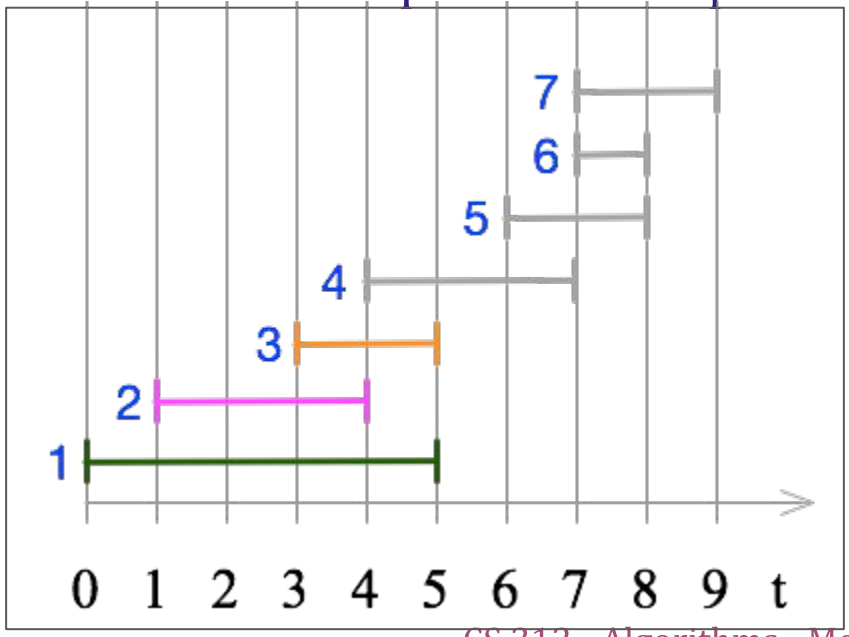
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - reuse previously allocated resource if its last request has completed



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	**3**	**5**	
4	4	7	
5	6	8	
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

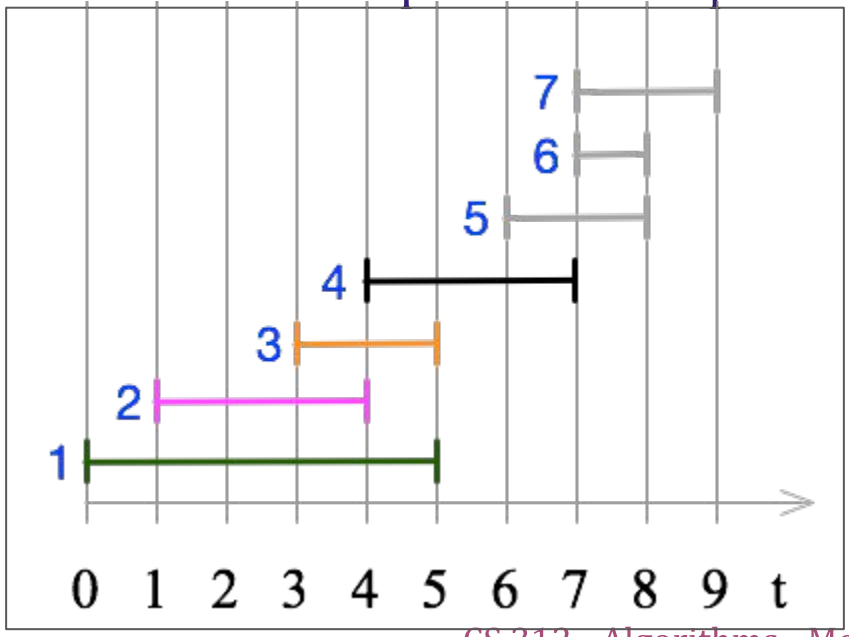
- Process requests ordered by start time
- For each request
  - **allocate a new resource**, or
  - reuse previously allocated resource if its last request has completed



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	4	7	
5	6	8	
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

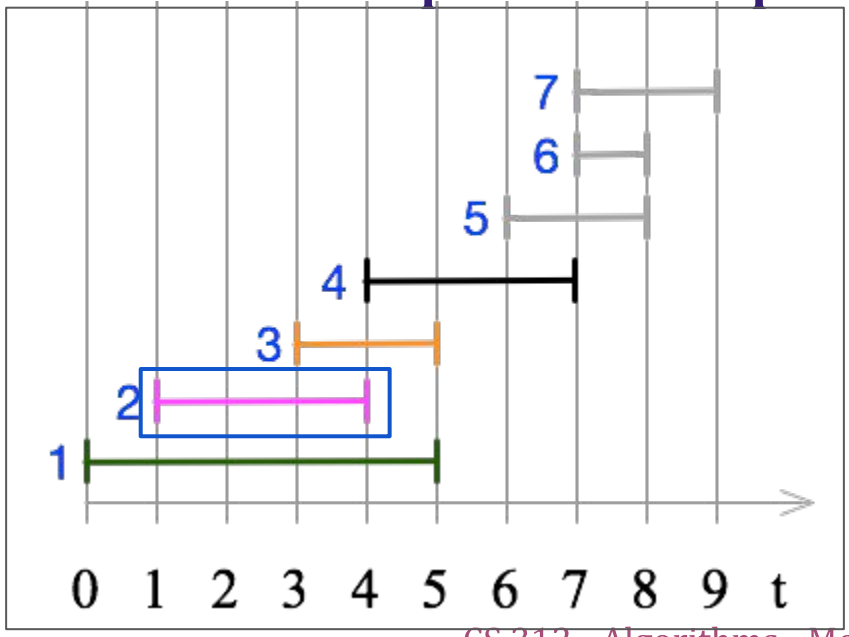
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - reuse previously allocated resource if its last request has completed



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	**4**	**7**	
5	6	8	
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

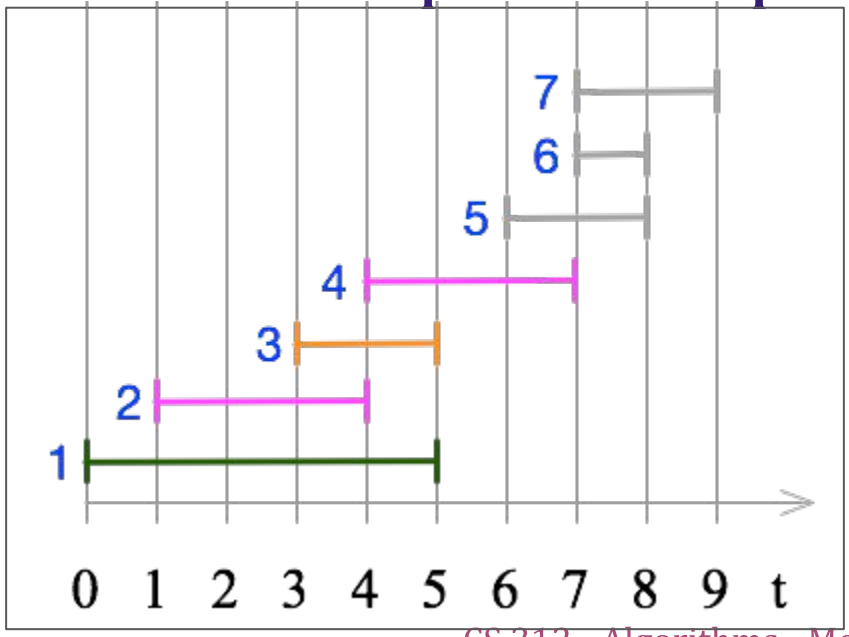
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - **reuse previously allocated resource if its last request has completed**



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	**4**	**7**	
5	6	8	
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

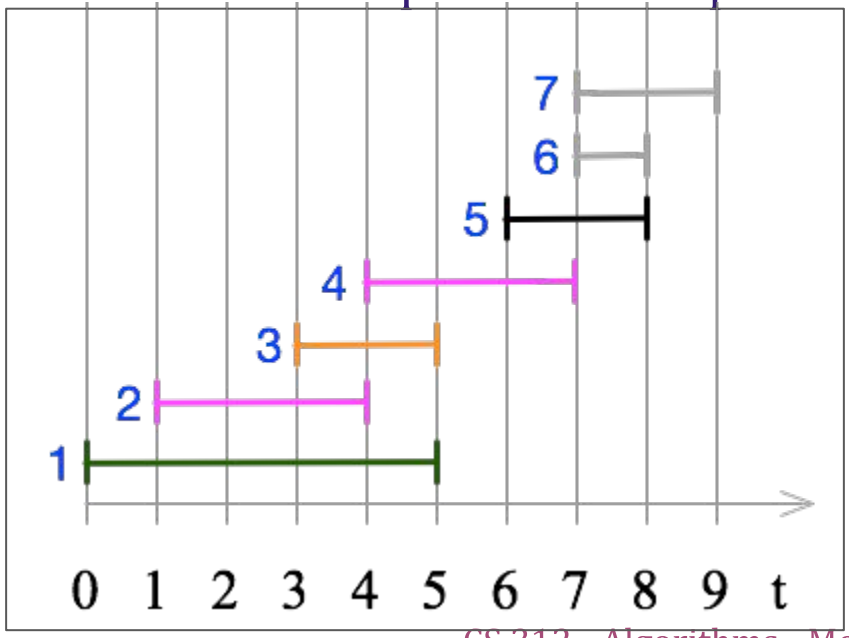
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - **reuse previously allocated resource if its last request has completed**



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	4	7	B
5	6	8	
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

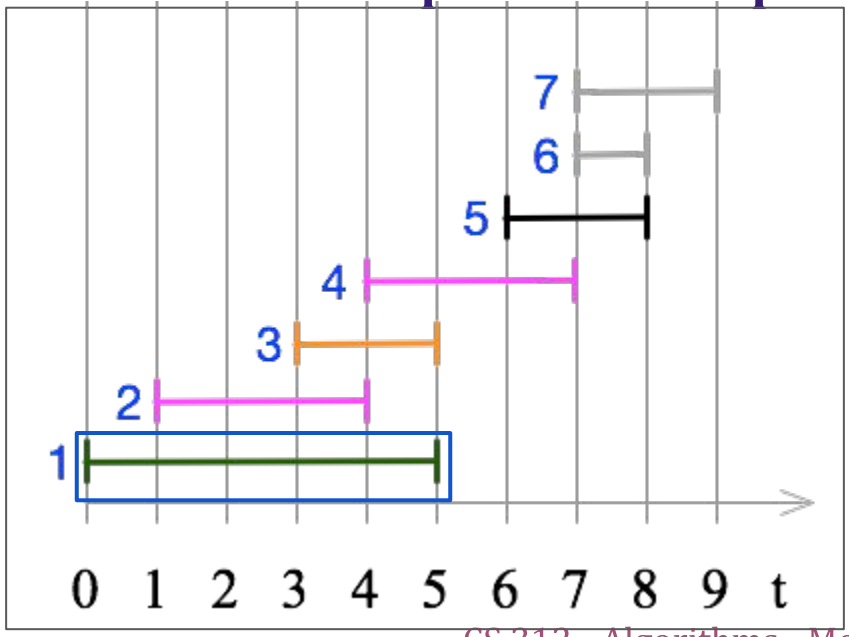
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - reuse previously allocated resource if its last request has completed



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	4	7	B
5	**6**	**8**	
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

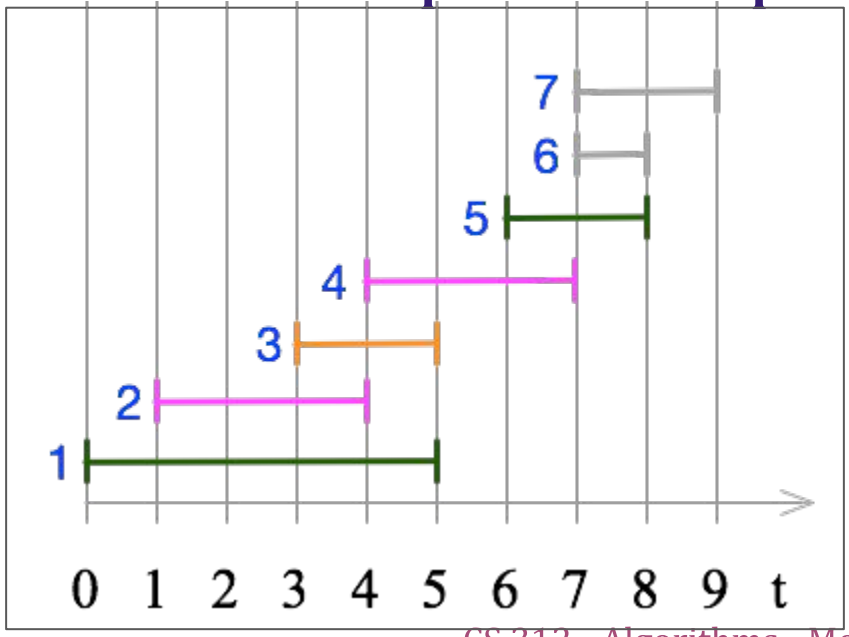
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - **reuse previously allocated resource if its last request has completed**



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	4	7	B
5	**6**	**8**	
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

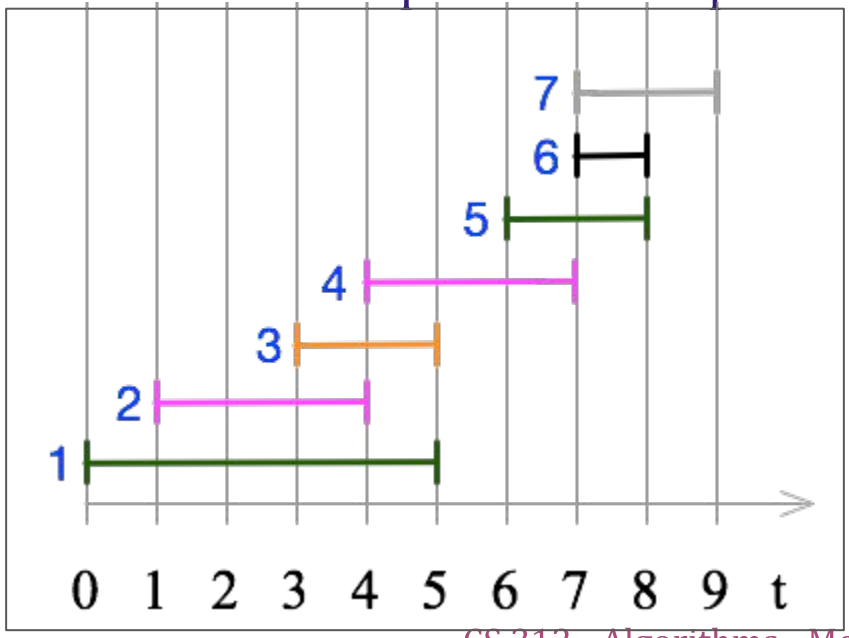
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - **reuse previously allocated resource if its last request has completed**



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	4	7	B
5	6	8	A
6	7	8	
7	7	9	

Greedy algorithm for interval partitioning

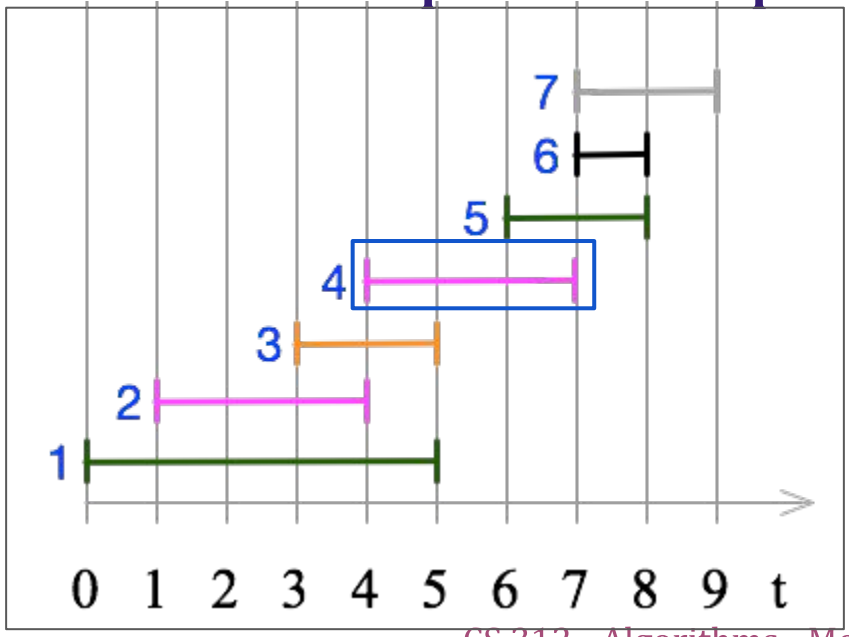
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - reuse previously allocated resource if its last request has completed



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	4	7	B
5	6	8	A
6	**7**	**8**	
7	7	9	

Greedy algorithm for interval partitioning

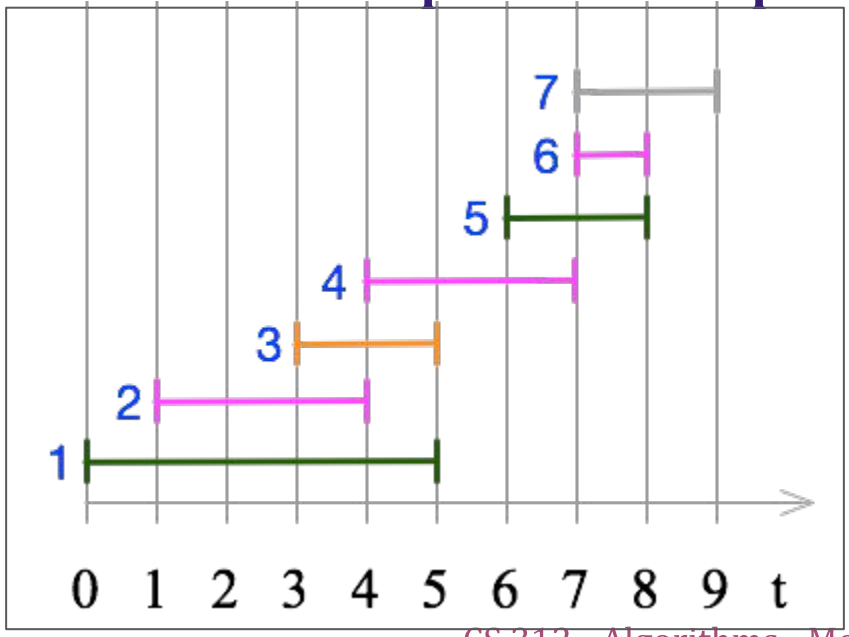
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - **reuse previously allocated resource if its last request has completed**



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	4	7	B
5	6	8	A
6	**7**	**8**	
7	7	9	

Greedy algorithm for interval partitioning

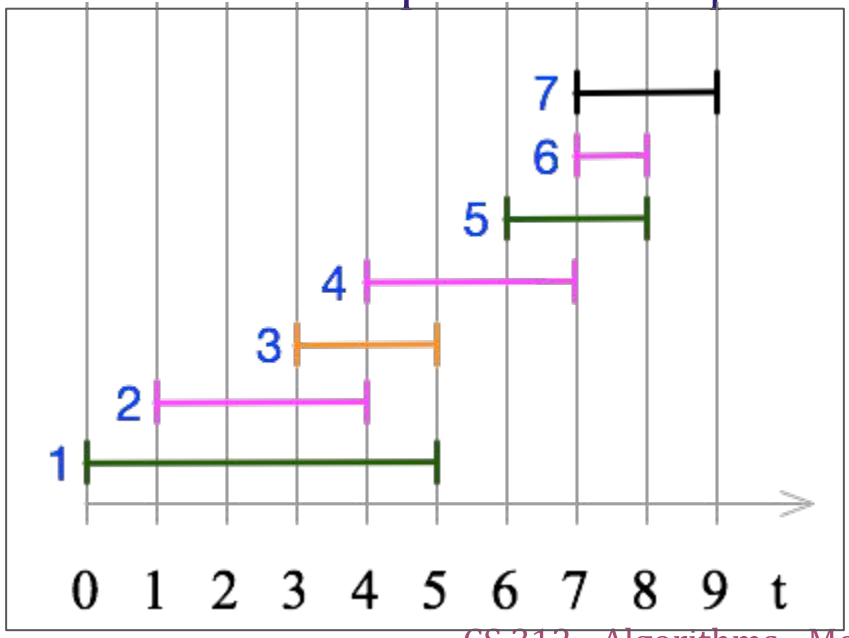
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - **reuse previously allocated resource if its last request has completed**



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	4	7	B
5	6	8	A
6	7	8	B
7	7	9	

Greedy algorithm for interval partitioning

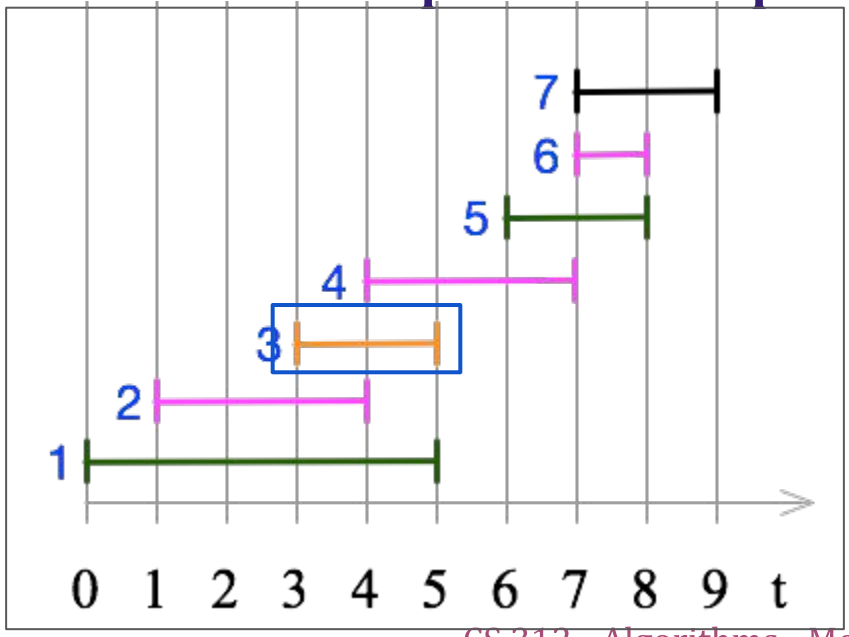
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - reuse previously allocated resource if its last request has completed



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	4	7	B
5	6	8	A
6	7	8	B
7	**7**	**9**	

Greedy algorithm for interval partitioning

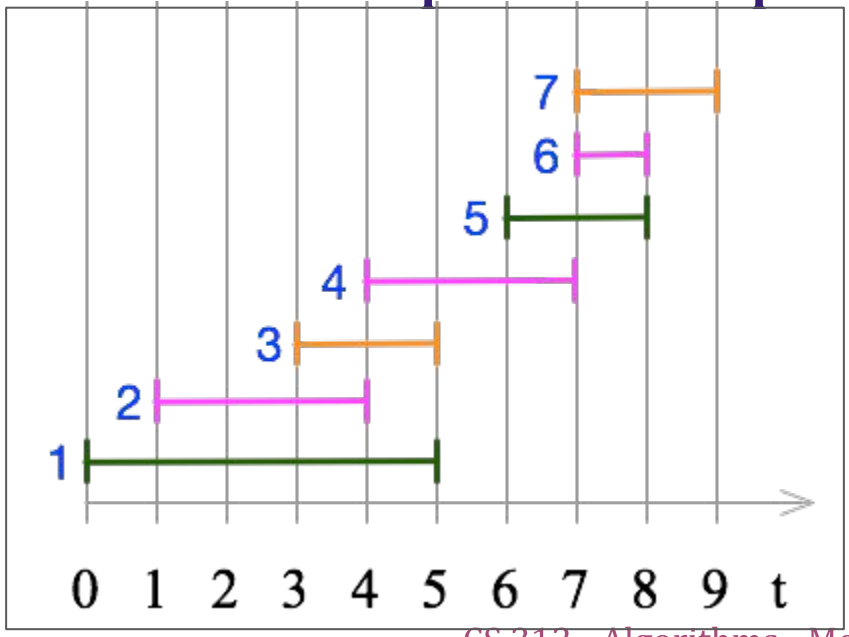
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - **reuse previously allocated resource if its last request has completed**



i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	4	7	B
5	6	8	A
6	7	8	B
7	**7**	**9**	

Greedy algorithm for interval partitioning

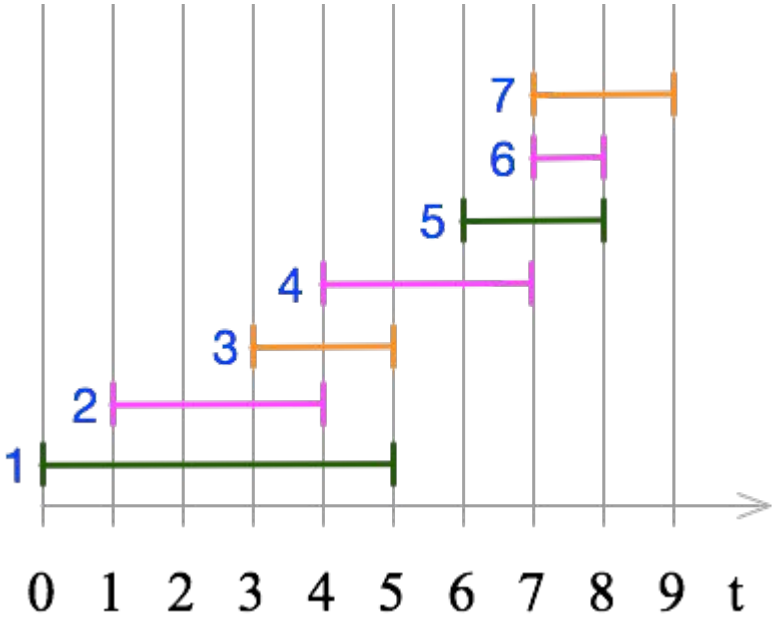
- Process requests ordered by start time
- For each request
  - allocate a new resource, or
  - **reuse previously allocated resource if its last request has completed**

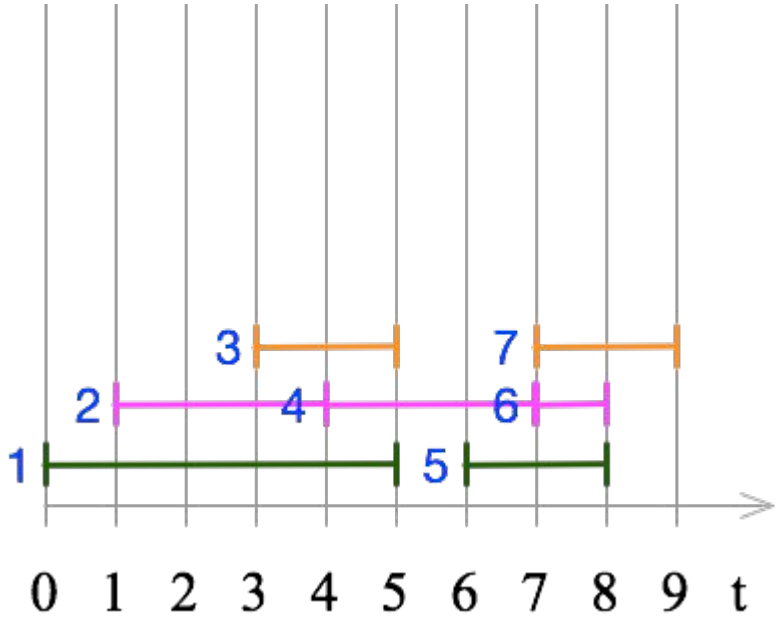


i	*s*(*i*)	*f*(*i*)	
1	0	5	A
2	1	4	B
3	3	5	C
4	4	7	B
5	6	8	A
6	7	8	B
7	7	9	C

Final schedule

3 rooms used





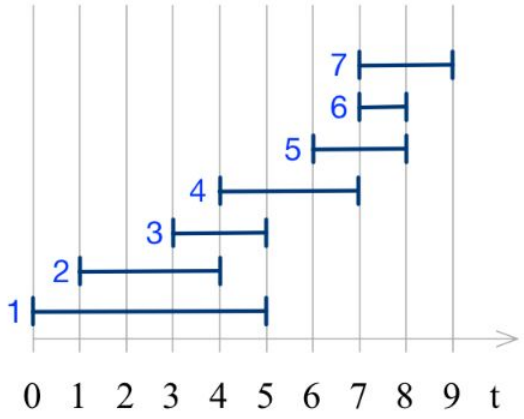
Analysis: do we get an optimal schedule?

- Let *d* be **depth**: maximum number of requests that overlap
- Any solution requires at least this many resources
- Claim greedy algorithm uses exactly *d* resources
  and is therefore optimal.
- Proof: By contradiction.
  - Suppose greedy uses more than *d* resources
  - There must have been *d* requests running when $(d+1)^{st}$ resource is allocated
  - This set of *d*+1 requests overlap => depth > *d*
  - Contradiction!

Running time analysis

- Exercise

Which of the following is an optimal set?



1,4,6

2,6

1,7

2,4,6

I don't know.

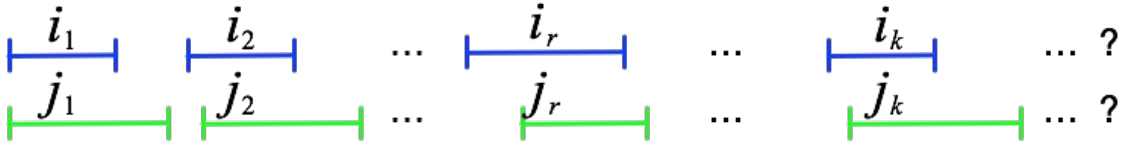
Because the given algorithm includes sorting, its running time must be:

```
R = all requests sorted by some property
S = {} // selected requests
while (R is not empty)
    remove next show i from R
    add i to S
    remove all overlapping shows from R
```

- $O(n \log n)$
- $\Omega(n \log n)$
- $\Theta(n \log n)$
- None of the above
- I don't know

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

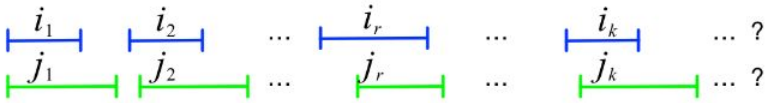
Greedy stays ahead



Claim $f(i_r) \leq f(j_r)$ for all $r = 1, 2, \ldots$

Parsons activity

Recall that k is the number of requests in the greedy solution and m is the number of requests in another solution. What have we just proven?



$f(i_r) \leq f(j_r)$ for $r = 1, 2, \ldots, k$

$f(i_r) \leq f(j_r)$ for $r = 1, 2, \ldots, m$

$f(i_r) \leq f(j_r)$ for $r = 1, 2, \ldots, \min(k, m)$

Correctness of the algorithm (the output is optimal).

All of the above.

I don't know.

Completing the proof: finish time sorting is optimal

- Claim algorithm outputs an **optimal** set
  - Compatible (no overlap)
  - Maximum-sized
- Proof
  - *S* is **compatible** by construction

Parsons activity

If the request with the next starting time is compatible with several resources, what should we choose to minimize the total number of resources?

Use the resource with the earliest finishing time (unused the longest)

Use the resource with the latest finishing time (most recently used)

Use a new resource

A or B

I don't know