Greedy algorithms and interval scheduling
COMS20010 (Algorithms II)

John Lapinskas, University of Bristol

Satellite scheduling

Suppose you're running a satellite imaging service.

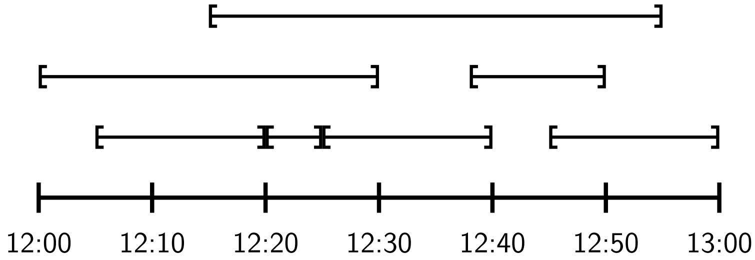
Taking a satellite picture of an area isn't instant — it takes time proportional to its latitude, and it can only be done at a specific time of day (when your satellite's orbit is lined up correctly).

You have a set of requested images, each of which can only be taken at specific times, and you can only take one picture at once.

How can you satisfy as many requests as possible?

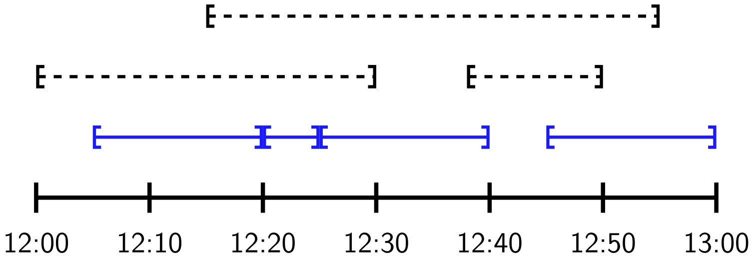
Satellite scheduling: An example

Requested satellite times: 12:00–12:30, 12:05–12:20, 12:15–12:55, 12:20-12:25, 12:25–12:40, 12:38–12:50, and 12:45-13:00.



Satellite scheduling: An example

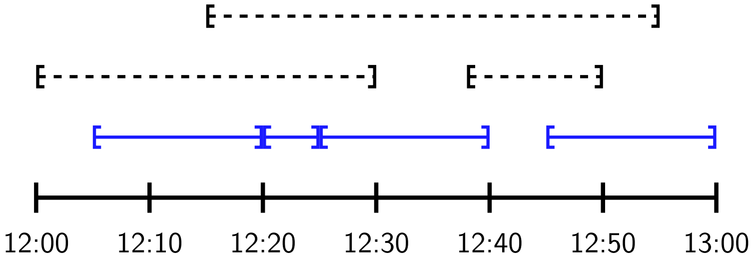
Requested satellite times: 12:10–12:30, **12:05–12:20**, 12:15–12:55, **12:20-12:25**, **12:25–12:40**, 12:38–12:50, and **12:45-13:00**.



Here, we can satisfy four requests.

Satellite scheduling: An example

Requested satellite times: 12:10–12:30, **12:05–12:20**, 12:15–12:55, **12:20-12:25**, **12:25–12:40**, 12:38–12:50, and **12:45-13:00**.

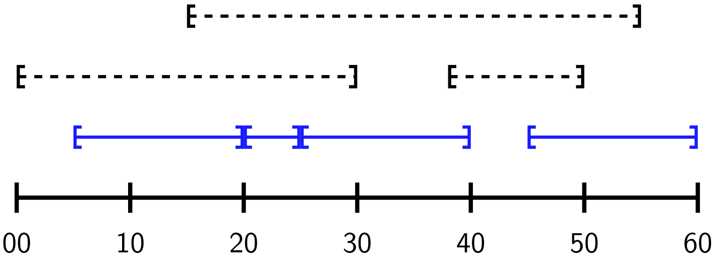


Here, we can satisfy four requests.

We might as well give our times integer labels for simplicity, though.

Satellite scheduling: An example

Requested satellite times: 0–30, **5–20**, 15–55, **20–25**, **25–40**, 38–50, and **45-60**.



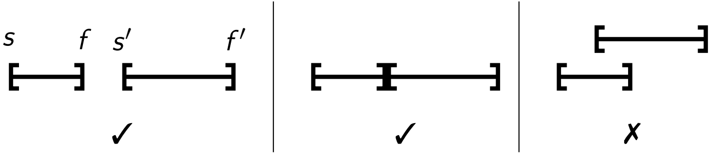
Here, we can satisfy four requests.

We might as well give our times integer labels for simplicity, though.

Interval scheduling: a formal definition

A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
We call s the **start time** and f the **finish time**.

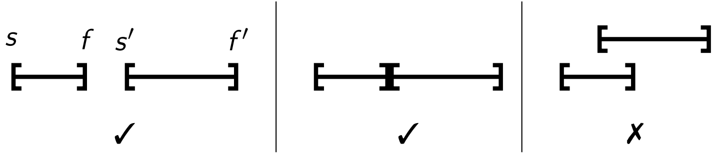
A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.



Interval scheduling: a formal definition

A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
We call s the **start time** and f the **finish time**.

A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.



Interval Scheduling Problem
Input: An array $\mathcal{R}$ of n requests $(s_1, f_1), \ldots, (s_n, f_n)$.
Desired Output: A compatible subset of $\mathcal{R}$ of maximum possible size.

Our satellite problem above is an example of this — the maximum compatible subset is the list of image requests we accept.

Solving interval scheduling

A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.

Input: An array $\mathcal{R}$ of n requests $(s_1, f_1), \ldots, (s_n, f_n)$.
Desired Output: A compatible subset of $\mathcal{R}$ of maximum possible size.

Solving interval scheduling

A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.

Input: An array $\mathcal{R}$ of n requests $(s_1, f_1), \dots, (s_n, f_n)$.
Desired Output: A compatible subset of $\mathcal{R}$ of maximum possible size.

Idea: Use a **greedy** approach: start with an empty output and slowly build it up, making choices that look good in the moment, without worrying about future implications. These algorithms are very common and useful!

Solving interval scheduling

A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.

Input: An array $\mathcal{R}$ of n requests $(s_1, f_1), \ldots, (s_n, f_n)$.
Desired Output: A compatible subset of $\mathcal{R}$ of maximum possible size.

Idea: Use a **greedy** approach: start with an empty output and slowly build it up, making choices that look good in the moment, without worrying about future implications. These algorithms are very common and useful!

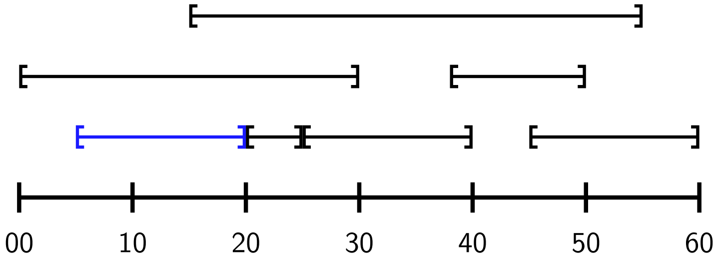
Heuristic: Tie up the satellite for as little time as possible before the next request. So at each stage, we accept the request which finishes earliest (out of those we haven't already missed the start time for).

A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.

Heuristic: Tie up the satellite for as little time as possible before the next request. So at each stage, we accept the request which finishes earliest (out of those we haven't already missed the start time for).

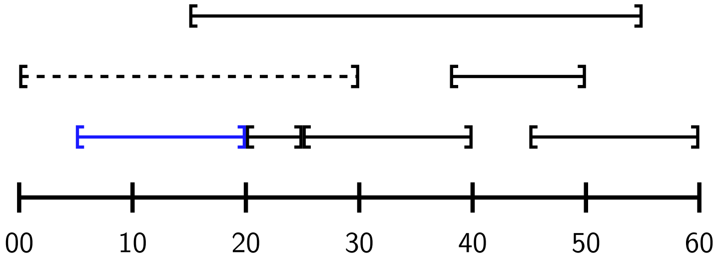
A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.

Heuristic: Tie up the satellite for as little time as possible before the next request. So at each stage, we accept the request which finishes earliest (out of those we haven't already missed the start time for).



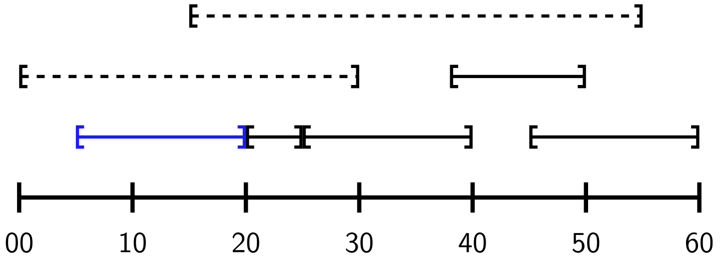
A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.

Heuristic: Tie up the satellite for as little time as possible before the next request. So at each stage, we accept the request which finishes earliest (out of those we haven't already missed the start time for).



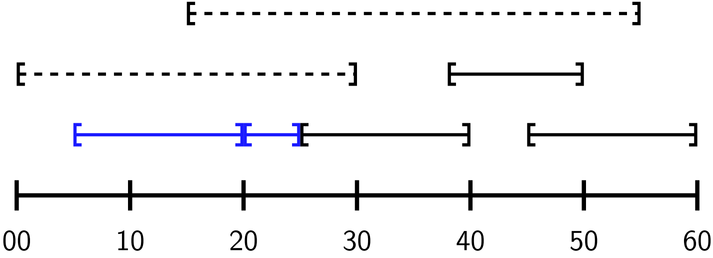
A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.

Heuristic: Tie up the satellite for as little time as possible before the next request. So at each stage, we accept the request which finishes earliest (out of those we haven't already missed the start time for).



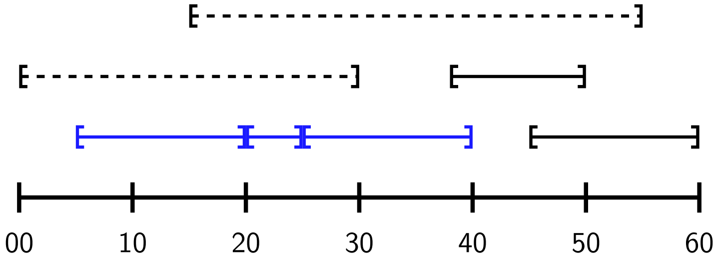
A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.

Heuristic: Tie up the satellite for as little time as possible before the next request. So at each stage, we accept the request which finishes earliest (out of those we haven't already missed the start time for).



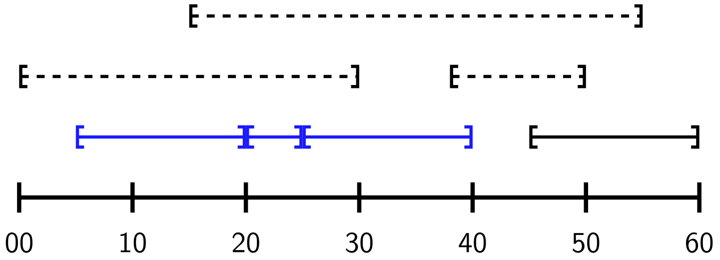
A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.

Heuristic: Tie up the satellite for as little time as possible before the next request. So at each stage, we accept the request which finishes earliest (out of those we haven't already missed the start time for).



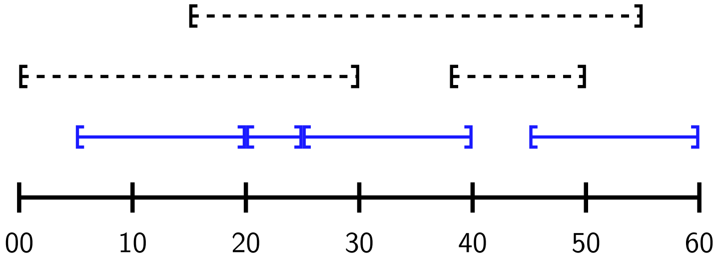
A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.

Heuristic: Tie up the satellite for as little time as possible before the next request. So at each stage, we accept the request which finishes earliest (out of those we haven't already missed the start time for).



A **request** is a pair of integers (s, f) with $0 \leq s \leq f$.
A set A of requests is **compatible** if for all distinct (s, f), $(s', f') \in A$, either $s' \geq f$ or $s \geq f'$ — that is, the requests' time intervals don't overlap.

Heuristic: Tie up the satellite for as little time as possible before the next request. So at each stage, we accept the request which finishes earliest (out of those we haven't already missed the start time for).



Algorithm: GreedySchedule

Input: An array $\mathcal{R}$ of n requests.

Output: A maximum compatible subset of $\mathcal{R}$.

begin
- Sort $\mathcal{R}$'s entries so that $\mathcal{R} \leftarrow [(s_1, f_1), \dots, (s_n, f_n)]$ where $f_1 \leq \cdots \leq f_n$.
- Initialise $A \leftarrow []$, `lastf` $\leftarrow 0$.
- **foreach** $i \in \{1, \dots, n\}$ **do**
  - **if** $s_i \geq$ `lastf` **then**
    - Append (s_i, f_i) to A and update `lastf` $\leftarrow f_i$.
- Return A.

Algorithm: GreedySchedule

Input: An array $\mathcal{R}$ of n requests.

Output: A maximum compatible subset of $\mathcal{R}$.

1 **begin**

2 Sort $\mathcal{R}$'s entries so that $\mathcal{R} \leftarrow [(s_1, f_1), \dots, (s_n, f_n)]$ where $f_1 \leq \cdots \leq f_n$.

3 Initialise $A \leftarrow [\,]$, `lastf` $\leftarrow 0$.

4 **foreach** $i \in \{1, \dots, n\}$ **do**

5 **if** $s_i \geq$ `lastf` **then**

6 Append (s_i, f_i) to A and update `lastf` $\leftarrow f_i$.

7 Return A.

Time analysis:

Step 2 takes $O(n \log n)$ time, as covered exhaustively in COMS10007.

Algorithm: GreedySchedule

Input: An array $\mathcal{R}$ of n requests.

Output: A maximum compatible subset of $\mathcal{R}$.

1 **begin**

2 Sort $\mathcal{R}$'s entries so that $\mathcal{R} \leftarrow [(s_1, f_1), \ldots, (s_n, f_n)]$ where $f_1 \leq \cdots \leq f_n$.

3 Initialise $A \leftarrow [\,]$, `lastf` $\leftarrow 0$.

4 **foreach** $i \in \{1, \ldots, n\}$ **do**

5 **if** $s_i \geq$ `lastf` **then**

6 Append (s_i, f_i) to A and update `lastf` $\leftarrow f_i$.

7 Return A.

Time analysis:

Step 2 takes $O(n \log n)$ time, as covered exhaustively in COMS10007.

Steps 3–6 all take O(1) time and are executed at most n times.

Algorithm: GREEDYSCHEDULE

Input: An array $\mathcal{R}$ of n requests.

Output: A maximum compatible subset of $\mathcal{R}$.

1 **begin**

2   Sort $\mathcal{R}$'s entries so that $\mathcal{R} \leftarrow [(s_1, f_1), \dots, (s_n, f_n)]$ where $f_1 \leq \dots \leq f_n$.

3   Initialise $A \leftarrow []$, `lastf` $\leftarrow 0$.

4   **foreach** $i \in \{1, \dots, n\}$ **do**

5   **if** $s_i \geq$ `lastf` **then**

6   Append (s_i, f_i) to A and update `lastf` $\leftarrow f_i$.

7   Return A.

Time analysis:

Step 2 takes $O(n \log n)$ time, as covered exhaustively in COMS10007.

Steps 3–6 all take O(1) time and are executed at most n times.

So overall the running time is $O(n \log n) + O(n) \cdot O(1) = O(n \log n)$. ✓

Algorithm: GreedySchedule

Input: An array $\mathcal{R}$ of n requests.

Output: A maximum compatible subset of $\mathcal{R}$.

```
1 begin
2     Sort R's entries so that R ← [(s_1, f_1), ..., (s_n, f_n)] where f_1 ≤ ··· ≤ f_n.
3     Initialise A ← [], lastf ← 0.
4     foreach i ∈ {1, ..., n} do
5         if s_i ≥ lastf then
6             Append (s_i, f_i) to A and update lastf ← f_i.
7     Return A.
```

Time analysis:

Step 2 takes $O(n \log n)$ time, as covered exhaustively in COMS10007.

Steps 3–6 all take O(1) time and are executed at most n times.

So overall the running time is $O(n \log n) + O(n) \cdot O(1) = O(n \log n)$. ✓

Proof of correctness: Next video!

Greedy algorithms in general

Greedy algorithm is a very informal term, and different people have incompatible definitions. My definition (see e.g. KT's book) is:

- they start with a sub-optimal (often trivial) solution, e.g. $A = []$, and gradually turn it into the output.
- they look over all possible improvements and pick the one that "looks best at the time", e.g. adding the fastest-ending compatible request.
- they never backtrack in "quality", e.g. the size of A never goes down.

Some people (see e.g. CLRS' book) have stricter definitions.
But this will never actually matter to you!

What matters is being able to **use** and **design** algorithms like this one.

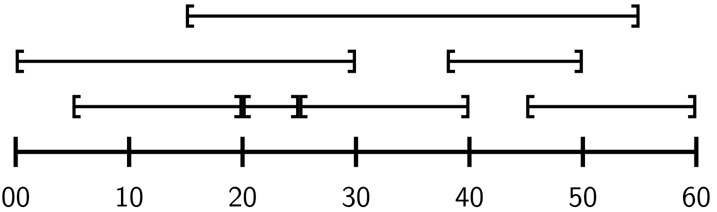
Greed is... not always good?

Sometimes the locally-optimal choices are the ones we regret the most...

Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** “just do the obvious thing at each stage”. Sometimes there are multiple obvious things, and only a few will work!

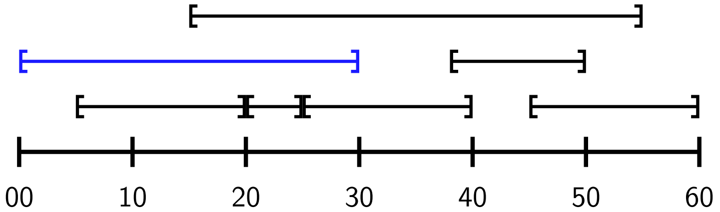
E.g. what if instead of choosing the fastest-finishing request to add at each stage, we chose the fastest-starting request?



Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** "just do the obvious thing at each stage". Sometimes there are multiple obvious things, and only a few will work!

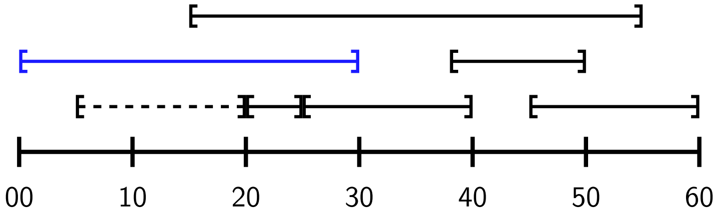
E.g. what if instead of choosing the fastest-finishing request to add at each stage, we chose the fastest-starting request?



Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** "just do the obvious thing at each stage". Sometimes there are multiple obvious things, and only a few will work!

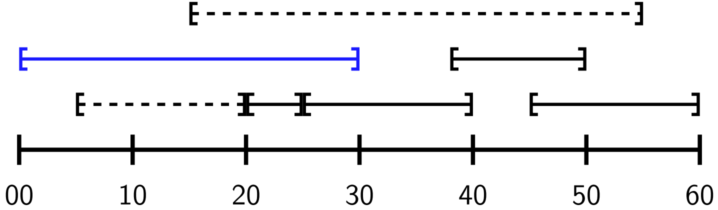
E.g. what if instead of choosing the fastest-finishing request to add at each stage, we chose the fastest-starting request?



Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** "just do the obvious thing at each stage". Sometimes there are multiple obvious things, and only a few will work!

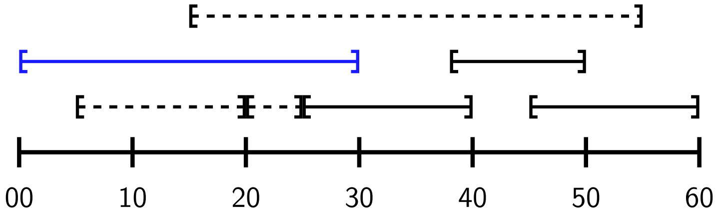
E.g. what if instead of choosing the fastest-finishing request to add at each stage, we chose the fastest-starting request?



Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** "just do the obvious thing at each stage". Sometimes there are multiple obvious things, and only a few will work!

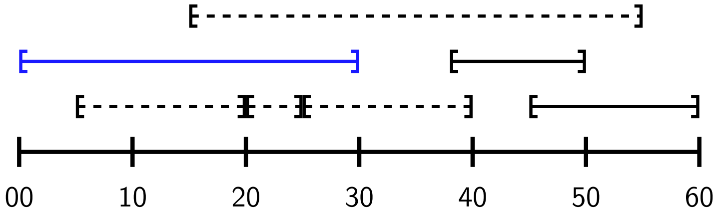
E.g. what if instead of choosing the fastest-finishing request to add at each stage, we chose the fastest-starting request?



Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** "just do the obvious thing at each stage". Sometimes there are multiple obvious things, and only a few will work!

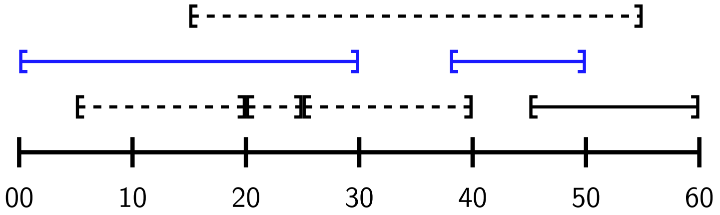
E.g. what if instead of choosing the fastest-finishing request to add at each stage, we chose the fastest-starting request?



Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** “just do the obvious thing at each stage”. Sometimes there are multiple obvious things, and only a few will work!

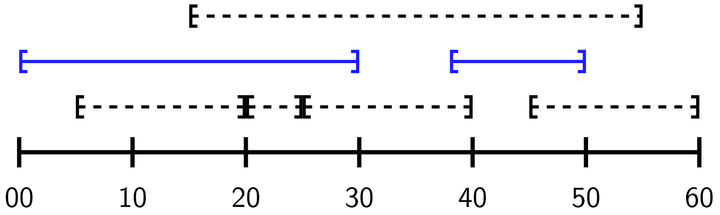
E.g. what if instead of choosing the fastest-finishing request to add at each stage, we chose the fastest-starting request?



Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** "just do the obvious thing at each stage". Sometimes there are multiple obvious things, and only a few will work!

E.g. what if instead of choosing the fastest-finishing request to add at each stage, we chose the fastest-starting request?

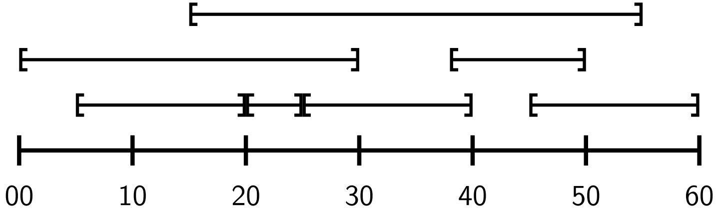


It doesn't work!

Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** "just do the obvious thing at each stage". Sometimes there are multiple obvious things, and only a few will work!

Or we chose the shortest interval?

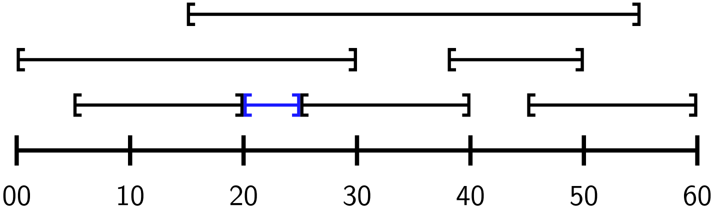


Still no luck. The lesson is: don't give up trying if "the" greedy algorithm doesn't work, because there are lots of possible greedy algorithms!

Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** "just do the obvious thing at each stage". Sometimes there are multiple obvious things, and only a few will work!

Or we chose the shortest interval?

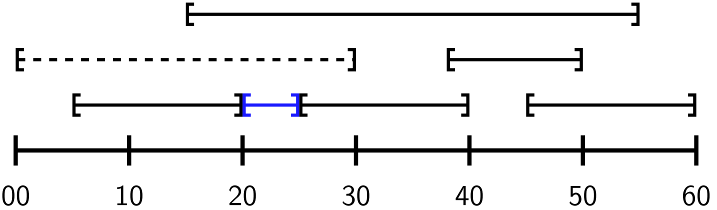


Still no luck. The lesson is: don't give up trying if "the" greedy algorithm doesn't work, because there are lots of possible greedy algorithms!

Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** "just do the obvious thing at each stage". Sometimes there are multiple obvious things, and only a few will work!

Or we chose the shortest interval?

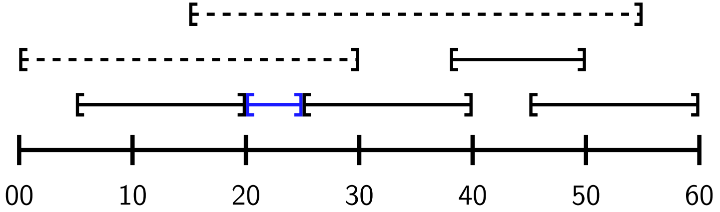


Still no luck. The lesson is: don't give up trying if "the" greedy algorithm doesn't work, because there are lots of possible greedy algorithms!

Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** "just do the obvious thing at each stage". Sometimes there are multiple obvious things, and only a few will work!

Or we chose the shortest interval?

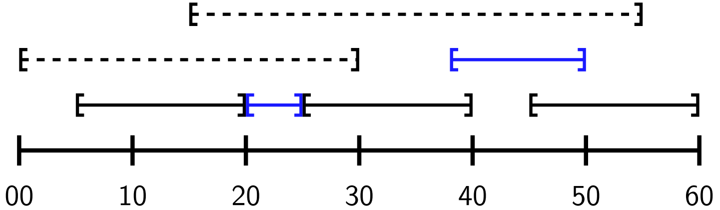


Still no luck. The lesson is: don't give up trying if "the" greedy algorithm doesn't work, because there are lots of possible greedy algorithms!

Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** "just do the obvious thing at each stage". Sometimes there are multiple obvious things, and only a few will work!

Or we chose the shortest interval?

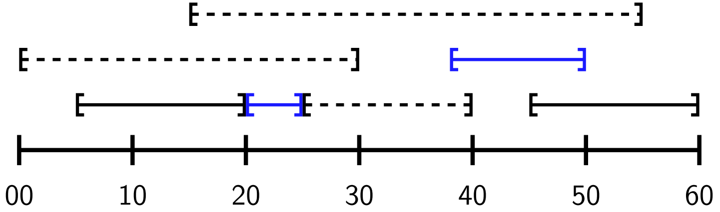


Still no luck. The lesson is: don't give up trying if "the" greedy algorithm doesn't work, because there are lots of possible greedy algorithms!

Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** “just do the obvious thing at each stage”. Sometimes there are multiple obvious things, and only a few will work!

Or we chose the shortest interval?

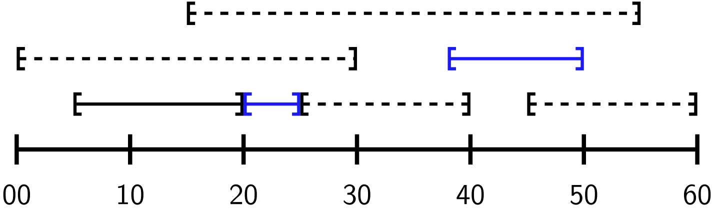


Still no luck. The lesson is: don't give up trying if “the” greedy algorithm doesn't work, because there are lots of possible greedy algorithms!

Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** “just do the obvious thing at each stage”. Sometimes there are multiple obvious things, and only a few will work!

Or we chose the shortest interval?

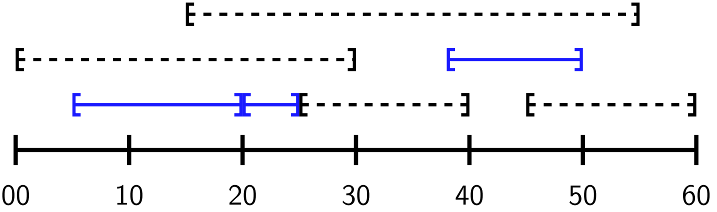


Still no luck. The lesson is: don’t give up trying if “the” greedy algorithm doesn’t work, because there are lots of possible greedy algorithms!

Greedy algorithms are not unique, and may fail!

A greedy algorithm is **not** “just do the obvious thing at each stage”. Sometimes there are multiple obvious things, and only a few will work!

Or we chose the shortest interval?



Still no luck. The lesson is: don’t give up trying if “the” greedy algorithm doesn’t work, because there are lots of possible greedy algorithms!