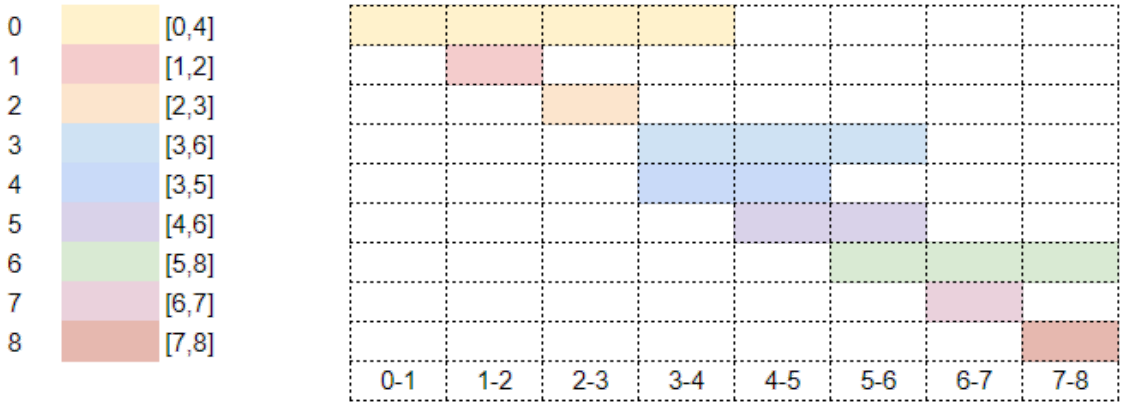
16.8 Application: Scheduling Events

Over the course of this chapter, we've studied several different sorting algorithms, learning about their design and implementation, and analysing their running time. We mentioned in the introduction of this chapter that sorting is commonly used as a step in more complex algorithms, and in this section we'll introduce one such algorithm to solve a real-world problem. While the code we'll develop here will use the built-in `sorted` function, we could easily swap it out with any of the sorting functions we've implemented in this chapter.

The problem description

Suppose you are the owner of a restaurant with a private dining room that can be booked by appointment only. Each day, you receive a list of requests to book the room for a certain period of time, and have to decide which requests to approve to book the room. Obviously, you can't book the room for more than one request at a time, and so can't say yes to every request if there are conflicting times. Here's an example of what these requests might look like:



There are many way you could decide which requests to accept. Perhaps you want to maximize the number of requests accepted, or the maximum amount of time that the room is booked for. You might see if you can get additional information, like the number of people expected to use the room or how much they're willing to bribe you.[1]

[1] That's a joke!!

For the purpose of this section, our goal will be to maximize the *number of different requests* that we accept. In our above example, we could take requests 1, 2, 3, 7, and 8, giving us 5 non-conflicting booking requests in total. But how do we know that 5 is the maximum number of requests? And more importantly, how can we design an *algorithm* to compute a schedule for us?

The interval scheduling maximization problem

Our above restaurant example is an instance of a more abstract problem in computer science known as **interval scheduling maximization** (shortened to *interval scheduling*).

Interval scheduling maximization problem

Input: a set of tuples, where each tuple is of the form (b, e), where $b, e \in \mathbb{R}$ and $b \leq e$. - We call each tuple (b, e) an *interval*.

Output: the largest possible subset of the input intervals where every pair of intervals are disjoint. We say that two intervals (b_1, e_1) and (b_2, e_2) are disjoint when they don't overlap except possibly at their endpoints; or equivalently, when $e_1 \leq b_2$ or $e_2 \leq b_1$.

- *Note*: If there is more than one largest possible subset, we can return any of them.

Here is the specification for a Python function to solve this problem:[2]

[2] We're using `int`s here to keep things simple, but the same function will work on `float`s as well.

```
def schedule_intervals(intervals: set[tuple[int, int]]) -> set[tuple[int,
        int]]:
    """Return a maximum-size subset of intervals that contains only disjoint
        intervals.

    If there is a tie in the maximum size, return any subset of disjoint
        intervals of that size.

    Preconditions:
        - all(interval[0] <= interval[1] for interval in intervals)
    """
```

The algorithm

While there are various algorithms for solving the interval scheduling maximization problem, the one we'll focus on is a algorithm that processes the intervals one at a time, at each iteration choosing whether to select it or not.[3] Here is the skeleton of such an

[3] We say that this algorithm is *greedy* because it makes choices by processing each interval one at a time, without taking into account what the remaining interval might be.

algorithm:

```
def schedule_intervals(intervals: set[tuple[int, int]]) -> set[tuple[int,
        int]]:
    """..."""
    selected_so_far = set()

    for interval in intervals:
        if ...:
            selected_so_far.add(interval)

    return selected_so_far
```

You might wonder how an algorithm like this could possibly be correct in computing a subset of maximum size. After all, it wouldn't be too hard to fill in the `...` with a helper function call that checks whether the current `interval` is disjoint with all current intervals in `selected_so_far`. But depending on the iteration order through `intervals`, we'd get different subsets, and quite possibly ones of different sizes as well.

It turns out that this algorithm structure does work, as long as we peform a preprocessing step: *sorting the intervals by their end time*. We can achieve this in Python by using the optional `key` argument for `list.sort`. In the code below, we define an additional function `end_time`. We also could have used an anonymous function (with the keyword `lambda`), which we discussed in lecture.

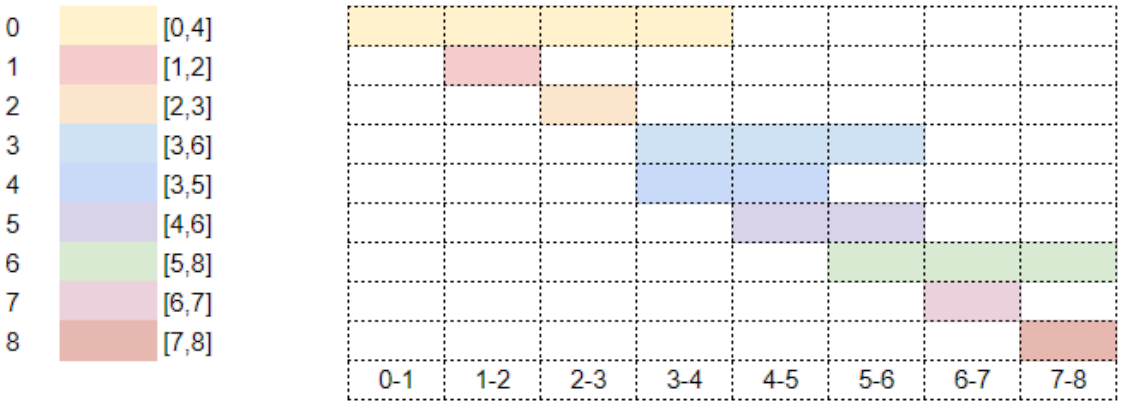
```
def schedule_intervals(intervals: set[tuple[int, int]]) -> set[tuple[int,
        int]]:
    """..."""
    sorted_intervals = sorted(intervals, key=end_time)

    selected_so_far = set()

    for interval in sorted_intervals:  # Loop over sorted_intervals
        if ...:
            selected_so_far.add(interval)

    return selected_so_far


def end_time(interval: tuple[int, int]) -> int:
    """Return the end time of the given interval."""
    return interval[1]
```

Now to complete the implementation, we need to decide when to add a new interval to our accumulator. The second key insight of this algorithm is that we can keep track of the latest end time of the intervals we've added so far, and then check whether the current `interval` starts after that or not. If it does, then `interval` doesn't conflict with the other intervals in `selected_so_far`, and we can add it to the accumulator and update the latest end time.

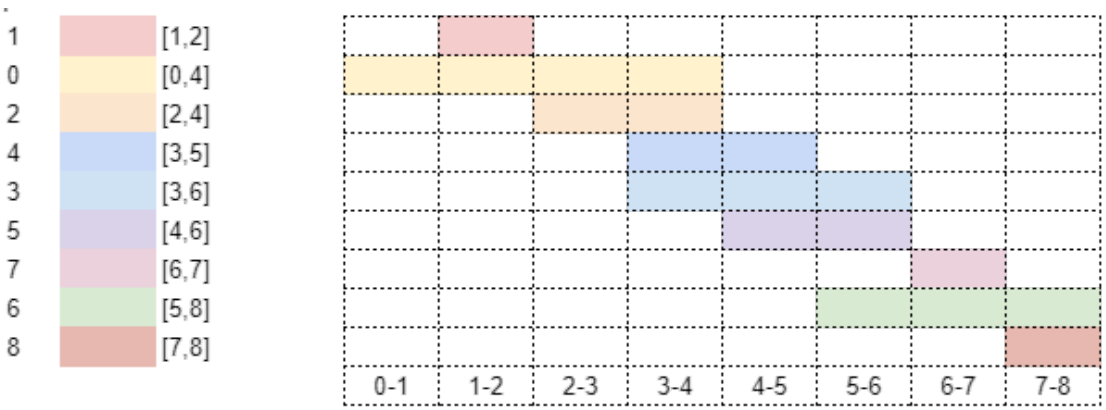
Using this idea, here is our complete implementation of `schedule_intervals`:

```
def schedule_intervals(intervals: set[tuple[int, int]]) -> set[tuple[int,
        int]]:
    """..."""
    sorted_intervals = sorted(intervals, key=end_time)

    selected_so_far = set()
    latest_end_time = -math.inf

    for interval in sorted_intervals:  # Loop over sorted_intervals
        if interval[0] >= latest_end_time:
            selected_so_far.add(interval)
            latest_end_time = end_time(interval)

    return selected_so_far


def end_time(interval: tuple[int, int]) -> int:
    """Return the end time of the given interval."""
    return interval[1]
```

Let's trace through this algorithm on a concrete example. Recall our room booking request example from earlier:



As a first step, our algorithm sorts these intervals by their end time, producing a list `sorted_intervals` that looks like:



Our loop (`for interval in sorted_intervals`) iterates through these intervals one at a time.

- At the first iteration, `interval = (1, 2)` and `latest_end_time = -math.inf`, and so this interval gets added to the accumulator. `latest_end_time` gets updated to `2`.
- At the second iteration, `interval = (0, 4)` and `latest_end_time = 2`. But then `interval[0] < latest_end_time`, and so it doesn't get added to the set.
- At the third iteration, `interval = (2, 4)` and `latest_end_time = 2`. Then `interval[0] >= latest_end_time`, and so this iterval is added to the accumulator. `latest_end_time` gets updated to `4`.

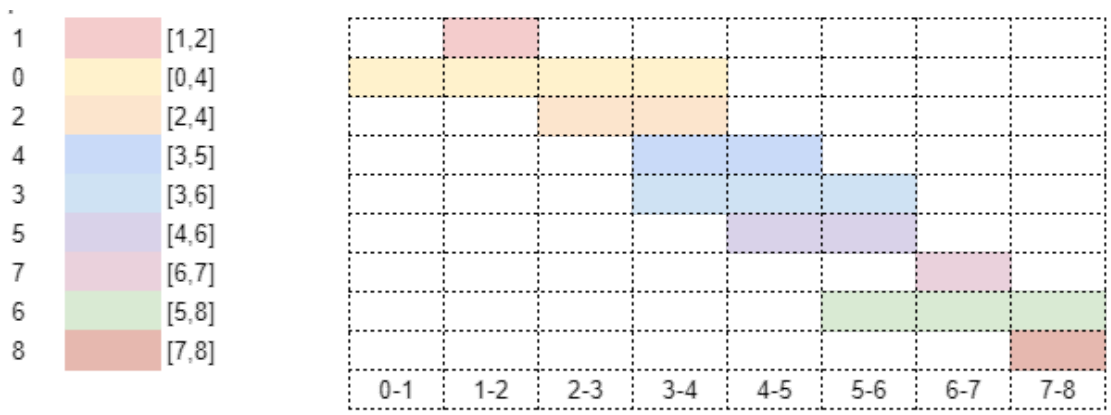
This process repeats for every interval; here is the full loop accumulation table for this algorithm.[4]

[4] Remember, the `latest_end_time` and `selected_so_far` columns show the values of these variables at the *end* of the row's corresponding loop iteration.

Iteration	interval	latest_end_time	selected_so_far
0		-math.inf	set()
1	(1, 2)	2	{(1, 2)}
2	(0, 4)	2	{(1, 2)}
3	(2, 4)	4	{(1, 2), (2, 4)}
4	(3, 5)	4	{(1, 2), (2, 4)}
5	(3, 6)	4	{(1, 2), (2, 4)}
6	(4, 6)	6	{(1, 2), (2, 4), (4, 6)}
7	(6, 7)	7	{(1, 2), (2, 4), (4, 6), (6, 7)}
8	(5, 8)	7	{(1, 2), (2, 4), (4, 6), (6, 7)}
9	(7, 8)	8	{(1, 2), (2, 4), (4, 6), (6, 7), (7, 8)}

So at the end, we obtain the final set of disjoint intervals `{(1, 2), (2, 4), (4, 6), (6, 7), (7, 8)}`. Notice that this set has the same size as the one we discussed in the opening

example, but is not identical to it. This is a good reminder that there can be more than one set with the maximum size.



But why does this algorithm work?

It might seem surprising to you that this algorithm works at all, given how short it seems to be! There are two subtle ideas at play in this algorithm: the sorting order (why end time?) and how we use the second accumulator variable `latest_end_time` to determine whether `interval` is disjoint to all of the elements of `selected_so_far`.

The latter is the easiest to explain, and we'll use a familiar approach: encoding the key property of `latest_end_time` as a loop invariant.

```
def schedule_intervals(intervals: set[tuple[int, int]]) -> set[tuple[int,
        int]]:
    """..."""
    sorted_intervals = sorted(intervals, key=end_time)

    selected_so_far = set()
    latest_end_time = -math.inf

    for interval in sorted_intervals:  # Loop over sorted_intervals
        # Loop invariant: latest_end_time is the maximum end time of the
         selected intervals
        assert (selected_so_far == set() or
                latest_end_time == max(end_time(i) for i in selected_so_far))

        if interval[0] >= latest_end_time:
            selected_so_far.add(interval)
            latest_end_time = end_time(interval)

    return selected_so_far
```

So our variable `latest_end_time` represents the *maximum end time* of the intervals in `selected_so_far`. This justifies why we can always just compare `interval[0] >= latest_end_time` to determine whether `interval` is disjoint to all of the selected intervals: if

`interval[0] >= latest_end_time`, then `interval[0]` is `>=` *all* of the interval end times in `selected_sof_far`, and so is disjoint with them. And because we're iterating through the intervals in order of end time, we know that we can assign `latest_end_time = end_time(interval)` every time we add the current `interval` to the selected set.

So that's how we're using `latest_end_time`, and why it's helpful to have sorted the intervals based on their end time. But this doesn't explain how we know that `selected_so_far` has maximum size! And in fact, if we had instead sorted the intervals by some other "key" functions, like their start time or size, this algorithm wouldn't always return an interval set with the maximum possible size.

Intuitively, it is the *maximum end time* of the intervals selected so far that determines which of the remaining intervals might conflict or not, rather than the start times or interval lengths. But how do we formalize this intuition? While it is possible to prove that sorting by end time does indeed result in the maximum set size, doing so is beyond the scope of this course.[5]

[5] But don't worry, you'll discuss interval scheduling and this algorithm formally in CSC373!

For now, you'll have to take our word for it that this algorithm is indeed correct, and makes use of sorting in a vital way. In fact, if you analyse the running time of this algorithm, the for loop takes $\Theta(n)$ time (where n is the number of intervals), so in fact the *slowest* part of this algorithm is the sorting step, even though the algorithm's logical complexity lies not in the sorting itself but what we're sorting by.