A

In plain terms, we want to check if there is an intersection between the segments $[a, b)$ and $[c, d)$, where it is guaranteed that $a < b$ and $c < d$.

We can derive a simple criterion, through the following chain of logic:

- If $b \leq c$, then clearly the answer is NO.
- If $d \leq a$, then clearly the answer is NO.
- On the other hand, if $c < b$ and $a < d$, then you can actually check that the answer is YES!
    - To convince yourself of this "intuitively", just draw a picture, and you'll see that these two constraints force an intersection.
    - Slightly more formally, you can check that:
        - If $a \leq c$, then $[c, b)$ belongs to both intervals.
        - If $c < a$, then $[a, d)$ belongs to both intervals.
    - Checking that these intervals have nonzero length and satisfy the above claims is left as a simple exercise in inequalities for a reader who wants to practice wrangling symbols. Working formally is an important skill to practice, especially since drawings can occasionally be misleading; we just won't dwell on it here.

The idea is to gain some footing by identifying *necessary* conditions, and then showing that what we have is (hopefully) *sufficient*.

Thus, the problem is *exactly equivalent* to just checking that $c \leq b$ and $a \leq d$ both hold!

```
a, b, c, d = map(int, input().split())
print(
    "YES"
    if c < b and a < d
    else "NO"
)

```

B

Let's consider the most straightforward brute force approach. For each integer t in $[420, 1140)$, check one-by-one if there exists *any* interval $[e_i, l_i)$ such that $t \in [e_i, l_i)$; if so, then someone is in the room, and if not, then the room is empty. Count all such t for which no one is in the room.

This is definitely correct, but is it efficient enough? Well...

- There are $1140 - 420 = 720$ minutes t to check
- For each such t, we (in the worst case) have to go over all n intervals to check if any of them contain t, totalling $\sim 720n$ operations
- Since $n \leq 1000$, that means our program never has to do more than $\sim 7.2 \times 10^5$ operations
- Because $7.2 \times 10^5 \ll 10^8$, our program is indeed more than fast enough, even in a slow language like Python.

So yes, problem solved! The bounds are generous enough that we don't need to overcomplicate things.

```
n = int(input())
intervals = [
    [int(x) for x in input().split()]
    for _ in range(n)
]

total = 0
for t in range(420, 1140): # [420, 1140)
    if not any(
        e <= t < l
        for e, l in intervals
    ):
        total += 1

print(total)

```

For an explanation of what the `any` function is doing at a more primitive level, you can refer to Problem B in this tutorial.

K

This is basically the same as Problem B, except the bounds *are* big enough to prohibit our previous solution: $(68400 - 25200) \times (3 \times 10^5) \gg 10^8$, so the brute force solution would perform too many operations and be too slow.

However, we note that $68400 - 25200 = 43200$ is still manageably small. For each integer time t in $[25200, 68400)$, is there a "fast" way for us determine if the room is empty?

Yes! In fact, more strongly, we can determine the exact number of people in the room at each second using a very natural idea. Simulate the state of the room as time passes. Let's *keep track of* the number of people in the room over time, and *update* this number whenever someone exits or leaves.

Initially, let there be 0 people in the room. Now, for each integer $t \in [25200, 68400)$, in order, do the following:

- Each interval with $e_i = t$ corresponds to someone entering the room at time t; add 1 to the number of people in the room, for each.
- Each interval with $l_i = t$ corresponds to someone leaving the room at time t; subtract 1 from the number of people in the room, for each
- Now, if there are currently 0 people in the room, add +1 to the total number of seconds when the room is empty.

To quickly count the number of intervals with an $e_i = t$ (and similarly for counting $l_i = t$), we can pre-process and use a data structure `dict` (in Python) or `map` (in C++).

```python
from collections import defaultdict
import io, os
input = io.BytesIO(os.read(0, os.fstat(0).st_size)).readline

n = int(input())
intervals = [
    [int(x) for x in input().split()]
    for _ in range(n)
]

START = 25200
END = 68400

# defaultdict --- if a key is not found, make a new entry for it whose value is 0
just_entered_count = defaultdict(int)
just_exited_count = defaultdict(int)

for e, l in intervals:
    just_entered_count[e] += 1
    just_exited_count[l] += 1

total = 0
curr_count = 0
for t in range(START, END): # [START, END)
    curr_count += just_entered_count[t]
    curr_count -= just_exited_count[t]
    if curr_count == 0:
        total += 1

print(total)

```

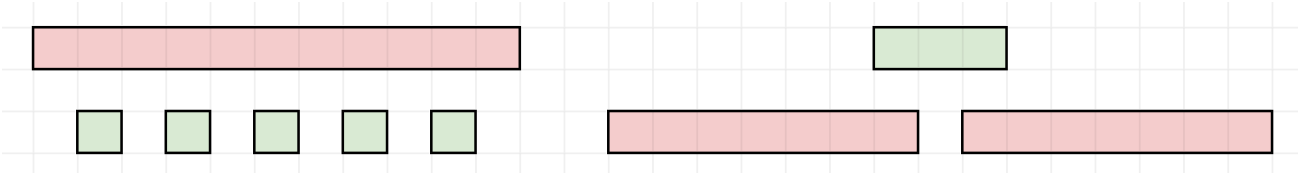
D

This problem can be solved using a family of techniques called *greedy algorithms*, which just means there is a "simple rule" for determining who to prioritize. Once you solve more problems, you will become more adept at identifying the "smell" of a problem that is amenable to greedy techniques.

Before we delve into the answer, let's take a moment to appreciate the non-trivial-ness of the fact that this problem can be solved by a simple rule, by first looking at some **wrong** "simple rules" and a counterexample for each one.

- Prioritize the people with the earlier start times, i.e. sort by e_i
- Prioritize the people with the shortest use times, i.e. sort by $|s_i - e_i|$

In the first bullet, we can imagine someone who starts at the beginning of the day, and then hogs the book all day. For the second bullet, we can imagine that it's possible for someone to only use the book for a brief amount of time, but with very unfortunate timing that causes them to conflict with *two* other possible borrowers.



Seems like any simple rule is too simple, right? Surely we could always construct some sort of wacky counterexample to stop any naive ideas... and yet a simple rule *does* exist that lets us solve the problem!

Prioritize the people with the earlier **end** times, i.e. sort by e_i. To elaborate:

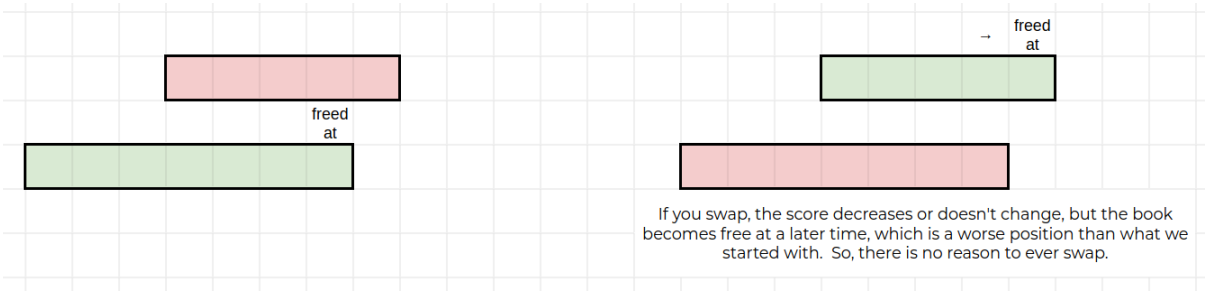
- **Sort** the intervals according to non-decreasing order of the e_i values
- Then, process each interval in that order:
  - If it does not conflict with any previously accepted intervals, accept it.
  - Otherwise, reject it.

Intuitively, an earlier end time is always better, because then the book is free again *at a sooner time*.

- Let `freed_at` be the e value of the last interval that we picked.
- We can see that the next interval (s, e) causes a conflict if and only if `s < freed_at`; in words, if we try to start a new interval before the book has been freed by the last borrower, we get a conflict.
- From here, we see that having a smaller `freed_at` value is **always more preferable**.

From this, we can conclude that deviating from the greedy plan is *never worth it*.

- Suppose there is an interval that conflicts with our currently-accepted choices.
- If we want to pick it, we would have to *give up* all of the previously-accepted choices that conflict with it, so our "score" can never improve, it only stays the same or gets worse.
- But why would you do that? Is it ever worth it? The only thing we've achieved is making `freed_at` even bigger... which leaves us in a worse state that before.
- Therefore, there is no good reason for us to deviate from the greedy plan.




```python
n = int(input())
# This time, also remember the index of each interval
intervals = [
    (i, [int(x) for x in input().split()])
    for i in range(1, n+1)
]

# Sort, according to "smaller l[i] first"
def smaller_l_first(interval):
    i, (e, l) = interval
    return l
intervals.sort(key=smaller_l_first)

accepted = []
freed_at = 420
for i, (e, l) in intervals:
    if freed_at <= e:
        accepted.append(i)
        freed_at = l

print(len(accepted))
print(*accepted)
```