<https://leetcode.com/problems/my-calendar-i/description/>

You are implementing a program to use as your calendar. We can add a new event if adding the event will not cause a **double booking**.

A **double booking** happens when two events have some non-empty intersection (i.e., some moment is common to both events.).

The event can be represented as a pair of integers start and end that represents a booking on the half-open interval [start, end), the range of real numbers x such that start <= x < end.

Implement the MyCalendar class:

* MyCalendar() Initializes the calendar object.
* boolean book(int start, int end) Returns true if the event can be added to the calendar successfully without causing a **double booking**. Otherwise, return false and do not add the event to the calendar.

**Example 1:**

Input

["MyCalendar", "book", "book", "book"]

[[], [10, 20], [15, 25], [20, 30]]

Output

[null, true, false, true]

Explanation

MyCalendar myCalendar = new MyCalendar();

myCalendar.book(10, 20); // return True

myCalendar.book(15, 25); // return False, It can not be booked because time 15 is already booked by another event.

myCalendar.book(20, 30); // return True, The event can be booked, as the first event takes every time less than 20, but not including 20.

**Constraints:**

* 0 <= start < end <= 109
* At most 1000 calls will be made to book.

**Attempt 1: 2023-11-30**

**Solution 1: Brute Force (10 min)**

class MyCalendar {

// s1 e1

// s2 e2 s2 e2

// no conflict condition: e2 <= s1 or e1 <= s2

// Based on De Morgan's law

// conflict condition: e2 > s1 and e1 > s2

List<int[]> list;

public MyCalendar() {

list = new ArrayList<>();

}

public boolean book(int start, int end) {

for(int[] range : list) {

if(end > range[0] && range[1] > start) {

return false;

}

}

list.add(new int[]{start, end});

return true;

}

}

/\*\*

\* Your MyCalendar object will be instantiated and called as such:

\* MyCalendar obj = new MyCalendar();

\* boolean param\_1 = obj.book(start,end);

\*/

Time Complexity: O(N^2). For each new event, we process every previous event to decide whether the new event can be booked.

Space Complexity: O(N), the size of the calendar.

**Refer to**

<https://leetcode.com/problems/my-calendar-i/editorial/>

#### **Approach #1: Brute Force**

**Intuition**

When booking a new event [start, end), check if every current event conflicts with the new event. If none of them do, we can book the event.

**Algorithm**

We will maintain a list of interval *events* (not necessarily sorted). Evidently, two events [s1, e1) and [s2, e2) do *not* conflict if and only if one of them starts after the other one ends: either e1 <= s2 OR e2 <= s1. By De Morgan's laws, this means the events conflict when s1 < e2 AND s2 < e1.

**Implementation**

public class MyCalendar {

List<int[]> calendar;

MyCalendar() {

calendar = new ArrayList();

}

public boolean book(int start, int end) {

for (int[] iv: calendar) {

if (iv[0] < end && start < iv[1]) return false;

}

calendar.add(new int[]{start, end});

return true;

}

}

**Complexity Analysis**

Let N be the number of events booked.

Time Complexity: O(N^2). For each new event, we process every previous event to decide whether the new event can be booked. This leads to



complexity.

Space Complexity: O(N), the size of the calendar.

**Solution 2: TreeMap sort based on interval start and only check 2 books start right before & after the new book starts (60 min)**

class MyCalendar {

// floor[fs,fe] ceiling[cs,ce]

// fs fe cs ce

// s e (new interval)

TreeMap<Integer, Integer> map;

public MyCalendar() {

map = new TreeMap<>();

}

public boolean book(int start, int end) {

// Retrieves the maximum entry whose key is less than or equal to start.

Map.Entry<Integer, Integer> floor = map.floorEntry(start);

// Retrieves the minimum entry whose key is greater than or equal to start.

Map.Entry<Integer, Integer> ceiling = map.ceilingEntry(start);

// If there is an overlap with the previous interval, return false.

if(floor != null && floor.getValue() > start) {

return false;

}

// If there is an overlap with the next interval, return false.

if(ceiling != null && ceiling.getKey() < end) {

return false;

}

map.put(start, end);

return true;

}

}

/\*\*

\* Your MyCalendar object will be instantiated and called as such:

\* MyCalendar obj = new MyCalendar();

\* boolean param\_1 = obj.book(start,end);

\*/

Time Complexity: O(N\*log N)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/my-calendar-i/solutions/109475/java-c-clean-code-with-explanation/>

**Instead of iterating through all the booked slots, we can just maintain all the booked slots in sorted order of their start. Then, we just need to check two cases:** **Keep existing books sorted and only check 2 books start right before & after the new book starts**

Another way to check overlap of 2 intervals is a started with b, or, b started within a.

Keep the intervals sorted, if the interval started right before the new interval contains the start, or if the interval started right after the new interval started within the new interval.

floor ceiling

... |----| ... |----| ...

|---------|

s e (the new interval)

if s < floor.end or e > ceiling.start, there is an overlap.

Another way to think of it:

If there is an interval start within the new book (must be the ceilingEntry) at all, or

books: |----| |--|

s |------| e

books: |----| |----|

s |----| e

If the new book start within an interval (must be the floorEntry) at all

books: |-------| |--|

s |---| e

books: |----| |----|

s |----| e

There is a overlap

**Java**

**TreeSet**

class MyCalendar {

TreeSet<int[]> books = new TreeSet<int[]>((int[] a, int[] b) -> a[0] - b[0]);

public boolean book(int s, int e) {

int[] book = new int[] { s, e }, floor = books.floor(book), ceiling = books.ceiling(book);

if (floor != null && s < floor[1]) return false; // (s, e) start within floor

if (ceiling != null && ceiling[0] < e) return false; // ceiling start within (s, e)

books.add(book);

return true;

}

}

**TreeMap**

class MyCalendar {

TreeMap<Integer, Integer> books = new TreeMap<>();

public boolean book(int s, int e) {

java.util.Map.Entry<Integer, Integer> floor = books.floorEntry(s), ceiling = books.ceilingEntry(s);

if (floor != null && s < floor.getValue()) return false; // (s, e) start within floor

if (ceiling != null && ceiling.getKey() < e) return false; // ceiling start within (s, e)

books.put(s, e);

return true;

}

}

**Refer to**

<https://algo.monster/liteproblems/729>

## Problem Description

In this problem, you are tasked with creating a program that acts as a calendar. The primary function of this calendar is to ensure that when a new event is added, it doesn't clash with any existing events (a situation referred to as a "double booking"). An event is defined by a start time and an end time, which are represented by a pair of integers. These times create a half-open interval [start, end), meaning it includes the start time up to, but not including, the end time.

Your job is to implement a MyCalendar class that will hold the events and has the following capabilities:

* MyCalendar() is a constructor that initializes the calendar object.
* book(int start, int end) is a method that adds an event to the calendar if it does not conflict with any existing events. If the event can be added without causing a double booking, it returns true; otherwise, it returns false and does not add the event.

The goal is to ensure that no two events overlap in time.

## Intuition

The key to solving this problem is to efficiently determine whether the newly requested booking overlaps with any existing bookings. One way to approach this is by maintaining a sorted list of events, allowing for quick searches and insertions.

The intuition is to search for the correct location to insert a new event such that the list remains sorted. We must check two things:

* That the new event's start time does not conflict with the end time of the previous event.
* That the new event's end time does not conflict with the start time of the next event.

We can accomplish this by:

1. Using the sortedcontainers.SortedDict class, which keeps keys in a sorted order. This allows us to quickly find the position where the new event could be inserted.
2. Applying the bisect\_right method to find the index of the smallest event end that is greater than the new event's start time.
3. Checking if the new event's end time conflicts with the next event's start time in the sorted dictionary.
4. If there is no conflict, we insert the new event into the "sorted" dictionary, with the end time as the key and the start time as the value.
5. By keeping the dictionary sorted by the end time, this ensures that we can always quickly check for potential overlap with the immediately adjacent events in the sorted list of bookings.

The implementation of the book method in our solution proceeds with this intuition, allowing for efficient booking operations. Each booking can be processed in logarithmic time with respect to the number of existing bookings, therefore making the solution scalable for a large number of events.

## Solution Approach

The implementation of MyCalendar relies on the sortedcontainers Python library, which offers a SortedDict data structure to maintain the events sorted by their end times. Here's a walkthrough of how the solution is implemented using this SortedDict:

**1.Initialization:** When the MyCalendar class is instantiated, it initializes a SortedDict in the constructor. This SortedDict is stored in the [self.sd](http://self.sd) attribute of the class instances, ready to keep track of the booked events.

def \_\_init\_\_(self):

self.sd = SortedDict()

**2.Booking an Event:** The book method is where the logic to check for double bookings and add events takes place.

First, we find the index (position) where the new event's end time would be inserted into the SortedDict. We use the bisect\_right method, which returns an index pointing to the first element in the SortedDict's values that is greater than the start time.

idx = self.sd.bisect\_right(start)

Now, we need to ensure that the new event does not conflict with the next event in the SortedDict. We check if the found index is within the bounds of the SortedDict and if the new event's end time is greater than the start time of the event at that index.

if idx < len(self.sd) and end > self.sd.values()[idx]:

return False

If there is no conflict, it means the new event does not cause a double booking, and we insert it into the dictionary. Here, the event's end time is used as the key and the start time as the value. This ensures the events are sorted by their end times.

self.sd[end] = start

After successfully adding the event without conflicts, the method returns True.

If at any point we detect an overlap (a potential double booking), we return False without adding the event.

def book(self, start: int, end: int) -> bool:

idx = self.sd.bisect\_right(start)

if idx < len(self.sd) and end > self.sd.values()[idx]:

return False

self.sd[end] = start

return True

The book method performs at most two key operations: finding where to insert and actually inserting the event. Both operations are efficient due to the nature of the SortedDict, which maintains the order of keys and allows for binary search insertions and lookups. This is how the provided solution ensures no double bookings occur while adding events to the MyCalendar.

### Example Walkthrough

Let's go through a small example to illustrate the solution approach.

Imagine we have a MyCalendar instance and we want to book two events. The first event is from time 10 to 20, and the second event is from time 15 to 25.

When we try to book the first event:

* We initialize our MyCalendar and its underlying SortedDict, currently empty.
* We attempt to book an event with start=10 and end=20.
* [self.sd.bisect\_right(10)](http://self.sd.bisect_right(10)) will return 0 since there are no keys greater than 10 (as the dictionary is empty).
* Since the index 0 is within bounds and there are no events, there are no conflicts.
* We add the event to the SortedDict with key 20 and value 10.

Now, MyCalendar looks like this:

[self.sd](http://self.sd) contains {20: 10}

When we try to book the second event:

* We attempt to book the second event with start=15 and end=25.
* [self.sd.bisect\_right(15)](http://self.sd.bisect_right(15)) will return index 0 since 20 is the first key greater than 15.
* We check if idx is within bounds and if end (25) is greater than the start time of the event at index 0 (which is 10). Since 25 is indeed greater than 10, there is a potential overlap with the existing event.
* Since end time 25 of the new event is greater than start time 10 of the existing event, which would result in a double booking, we return False.

Thus, the attempt to book an event from time 15 to 25 fails, preserving the non-overlapping constraint of the calendar. The SortedDict remains unchanged with the single event {20: 10} and no double booking occurs.

import java.util.Map;

import java.util.TreeMap;

// The class MyCalendar is designed to store bookings as intervals.

// It uses a TreeMap to keep the intervals sorted by start time.

class MyCalendar {

// Using TreeMap to maintain the intervals sorted by the start key.

private final TreeMap<Integer, Integer> calendar;

// Constructor initializes the TreeMap.

public MyCalendar() {

calendar = new TreeMap<>();

}

/\*\*

\* Tries to book an interval from start to end.

\* @param start the starting time of the interval

\* @param end the ending time of the interval

\* @return true if the booking does not conflict with existing bookings, false otherwise

\*/

public boolean book(int start, int end) {

// Retrieves the maximum entry whose key is less than or equal to start.

Map.Entry<Integer, Integer> floorEntry = calendar.floorEntry(start);

// If there is an overlap with the previous interval, return false.

if (floorEntry != null && floorEntry.getValue() > start) {

return false;

}

// Retrieves the minimum entry whose key is greater than or equal to start.

Map.Entry<Integer, Integer> ceilingEntry = calendar.ceilingEntry(start);

// If there is an overlap with the next interval, return false.

if (ceilingEntry != null && ceilingEntry.getKey() < end) {

return false;

}

// If there is no overlap, add the interval to the TreeMap and return true.

calendar.put(start, end);

return true;

}

}

// Usage example:

// MyCalendar obj = new MyCalendar();

// boolean isBooked = obj.book(start, end);

## Time and Space Complexity

The provided code implements a class MyCalendar that stores the start time of the events as values and the end time as keys in a SortedDict. When a new event is booked, it checks if there is any overlap with existing events and then stores the event in the SortedDict if there is no overlap.

### Time Complexity

* \_\_init\_\_: The constructor simply creates a new SortedDict, which is an operation taking O(1) time.
* book: This method involves two main actions. First, it performs a binary search to find the right position where the new event should be inserted. The bisect\_right method in SortedDict runs in O(log n) time where n is the number of keys in the dictionary. Secondly, the code inserts the new (end, start) pair into the dictionary. Inserting into a SortedDict also takes O(log n) time. Therefore, the overall time complexity for each book operation is O(log n).

### Space Complexity

The space complexity of the code is mainly dictated by the storage requirements of the SortedDict.

* With no events booked, the space complexity is O(1) as only an empty SortedDict is maintained.
* As events are added, the space complexity grows linearly with the number of non-overlapping events stored. Therefore, in the worst-case scenario, where the calendar has n non-overlapping events, the space complexity would be O(n).

Overall, the space complexity of the MyCalendar data structure is O(n) where n is the number of non-overlapping events booked in the calendar.