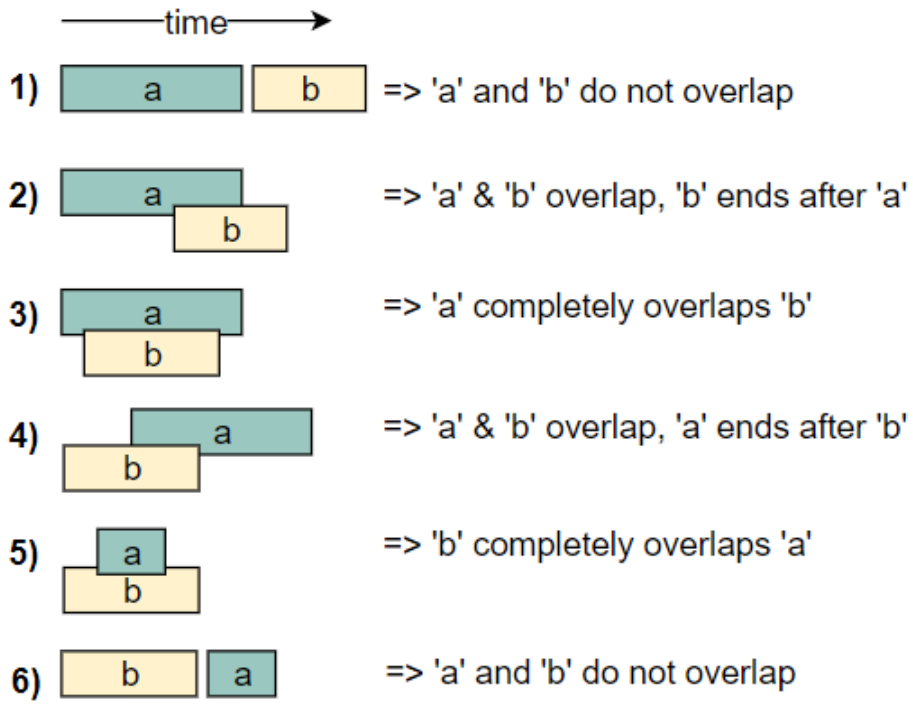
区间问题

笔记本:	0_leetcode		
创建时间:	2020/8/24 14:47	**更新时间:**	2020/8/25 10:30
作者:	415669592@qq.com		
URL:	https://www.educative.io/courses/grokking-the-coding-interview/3jyVPKRA8yx		

区间问题

0. 介绍

首先要明白俩个区间之间的6种关系。



1. 实战

1.1 合并区间

> 给定一系列区间，将相交的区间合并成一个大的区间。区间之间没有顺序。

e.g.1

```
Intervals: [[1,4], [2,5], [7,9]]
Output: [[1,5], [7,9]]
Explanation: Since the first two intervals [1,4] and [2,5] overlap, we merged
them into one [1,5].
```

e.g.2

```
Intervals: [[6,7], [2,4], [5,9]]
Output: [[2,4], [5,9]]
```

```
Explanation: Since the intervals [6,7] and [5,9] overlap, we merged them into
one [5,9].
```

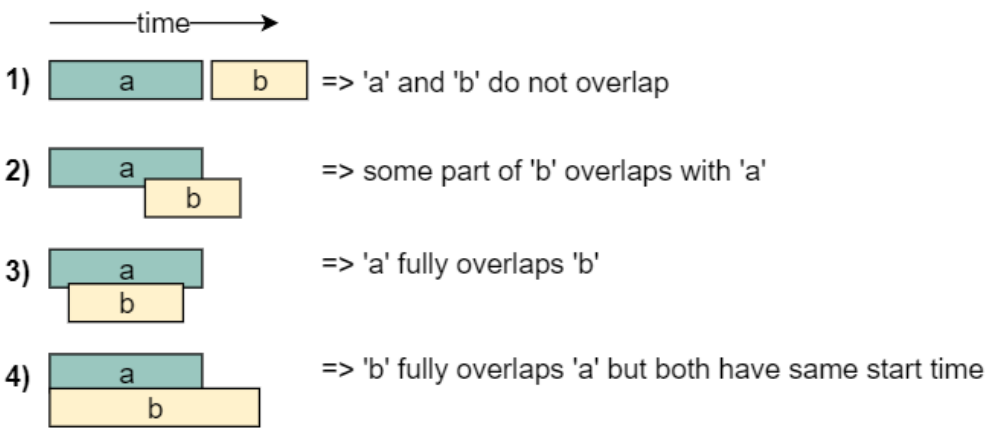
e.g.3

```
Intervals: [[1,4], [2,6], [3,5]]
Output: [[1,6]]
Explanation: Since all the given intervals overlap, we merged them into one.
```

思路

1. 区间是无序的，第一想到的是排序。很明显，应该直接使用区间start作为比较值。
2. 判断相交。因为排序了，所以保证a.start <= b.start 。

那么只会出现如下四种情况。



判断相交就是b.start <= a.end。合并后的区间就是c
c.start = a.start
c.end = max(a.end, b.end)
3. 很明显，类似双指针，我们应该持有一个指向当前区间的interval，和一个指向下一个区间的指针。
如果相交，当前interval被merged。当区间不相交时，更新当前区间，并将之前的区间放到结果集合中。

```cpp
using namespace std;

#include <algorithm>
#include <iostream>
#include <vector>

class Interval {
```

```
  public:
    int start = 0;
    int end = 0;

    Interval( int start, int end ) {
        this->start = start;
        this->end = end;
    }
};

class MergeIntervals {
  public:
    static vector<Interval> merge( vector<Interval> &intervals ) {
        vector<Interval> mergedIntervals;
        // TODO: Write your code here
        sort( intervals.begin(), intervals.end(), []( Interval &i1, Interval
&i2 ) {
            if ( i1.start == i2.start )
                return i1.end > i2.end;
            return i1.start <= i2.start;
        } );
        Interval curr_interval = intervals[0];
        Interval *next_interval;
        for ( int i = 1; i < intervals.size(); i++ ) {
            next_interval = &intervals[i];
            if ( next_interval->start <= curr_interval.end ) {
                curr_interval.end = max( curr_interval.end, next_interval-
>end );
            } else {
                mergedIntervals.push_back( curr_interval );
                curr_interval = *next_interval;
            }
        }
        mergedIntervals.push_back( curr_interval );
        return mergedIntervals;
    }
};

int main( int argc, char *argv[] ) {
    vector<Interval> input = {{1, 3}, {2, 5}, {7, 9}};
    cout << "Merged intervals: ";
    for ( auto interval : MergeIntervals::merge( input ) ) {
        cout << "[" << interval.start << "," << interval.end << "] ";
    }
```

```
    cout << endl;

    input = {{6, 7}, {2, 4}, {5, 9}};
    cout << "Merged intervals: ";
    for ( auto interval : MergeIntervals::merge( input ) ) {
        cout << "[" << interval.start << "," << interval.end << "] ";
    }
    cout << endl;

    input = {{1, 4}, {2, 6}, {3, 5}};
    cout << "Merged intervals: ";
    for ( auto interval : MergeIntervals::merge( input ) ) {
        cout << "[" << interval.start << "," << interval.end << "] ";
    }
    cout << endl;
    system( "pause" );
}
```

1.2 插入区间

> 给定一系列不相交的区间，且使用start排序。插入一个新的区间，求合并后的区间。

e.g.1

```
Input: Intervals=[[1,3], [5,7], [8,12]], New Interval=[4,6]
Output: [[1,3], [4,7], [8,12]]
Explanation: After insertion, since [4,6] overlaps with [5,7], we merged them
into one [4,7].
```

e.g.2

```
Input: Intervals=[[1,3], [5,7], [8,12]], New Interval=[4,10]
Output: [[1,3], [4,12]]
Explanation: After insertion, since [4,10] overlaps with [5,7] & [8,12], we
merged them into [4,12].Example 3:
```

e.g.3

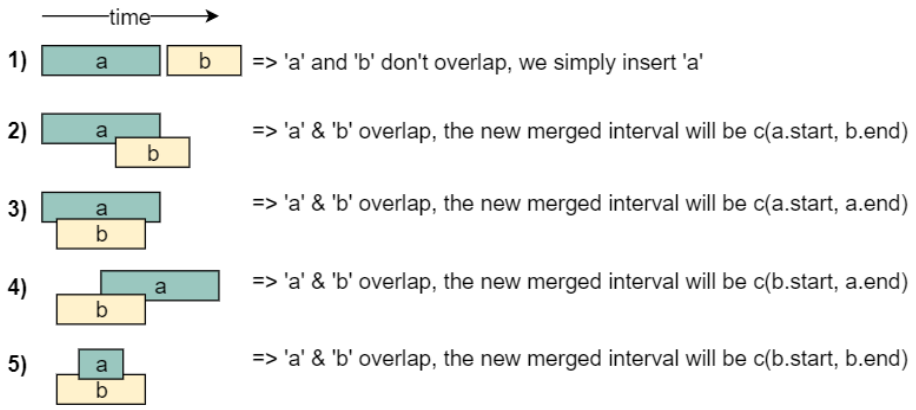
```
Input: Intervals=[[2,3],[5,7]], New Interval=[1,4]
Output: [[1,4], [5,7]]
Explanation: After insertion, since [1,4] overlaps with [2,3], we merged them
into one [1,4].
```

思路

区间已经排好序，我们只需要将新的区间插入到即可。自己可以构造几种情况来模拟，就知道如何插入了。

1. 如何跳过区间。**很明显，新区间出现在intervals[i]右边，且不相交。就可以跳过。intervals[i].end < newInterval.start**。明白这个条件是这道题的关键。

直接画图。一目了然。



a为newInterval

```
if(intervals[i].end < newInterval.start)
    skip;
else{
    可能发生相交assert(newInterval.start <= intervals.end)
    if(newInterval.end < intervals[i].end)
        不相交
    else
        相交
        curr_interval.start = max(....)
        curr_interval.end = max(...)
}
```

```cpp
using namespace std;

#include <iostream>
#include <vector>
#include <algorithm>

class Interval {
  public:
    int start = 0;
    int end = 0;

    Interval() = default;
    Interval( int start, int end ) {
        this->start = start;
        this->end = end;
    }
};

class InsertInterval {
  public:
    static vector<Interval> insert( const vector<Interval> &intervals,
Interval newInterval ) {
        vector<Interval> mergedIntervals;
        // TODO: Write your code here
        Interval curr_interval;
        int record = 0;
        for ( int i = 0; i < intervals.size(); i++ ) {
            if ( intervals[i].end < newInterval.start ) {
                mergedIntervals.push_back( intervals[i] );
                continue;
            } else {
                // 五种情况，实际上就是相交不相交俩种情况
                if ( newInterval.end < intervals[i].start ) {
                    mergedIntervals.push_back( newInterval );
                    for( int k = i; k < intervals.size(); k++ )
                        mergedIntervals.push_back( intervals[k] );
                    return mergedIntervals;
                } else {
                    curr_interval.start = min( intervals[i].start,
newInterval.start );
                    curr_interval.end = max( intervals[i].end,
newInterval.end );
                    record = i;
```

```
                    break;
                }
            }
        }
        // 开始合并
        const Interval *next_interval;
        for ( int i = record + 1; i < intervals.size(); i++ ) {
            next_interval = &intervals[i];
            if ( next_interval->start >= curr_interval.start &&
next_interval->start <= curr_interval.end ) {
                curr_interval.end = max( curr_interval.end, next_interval-
>end );
            } else {
                mergedIntervals.push_back( curr_interval );
                curr_interval = *next_interval;
            }
        }
        mergedIntervals.push_back( curr_interval );
        return mergedIntervals;
    }
};

int main( int argc, char *argv[] ) {
    vector<Interval> input = {{1, 3}, {5, 7}, {8, 12}};
    cout << "Intervals after inserting the new interval: ";
    for ( auto interval : InsertInterval::insert( input, {4, 6} ) ) {
        cout << "[" << interval.start << "," << interval.end << "] ";
    }
    cout << endl;

    cout << "Intervals after inserting the new interval: ";
    for ( auto interval : InsertInterval::insert( input, {4, 10} ) ) {
        cout << "[" << interval.start << "," << interval.end << "] ";
    }
    cout << endl;

    input = {{2, 3}, {5, 7}};
    cout << "Intervals after inserting the new interval: ";
    for ( auto interval : InsertInterval::insert( input, {1, 4} ) ) {
        cout << "[" << interval.start << "," << interval.end << "] ";
    }
    cout << endl;

    input = {{2, 3}, {6, 7}};
```

```
    cout << "Intervals after inserting the new interval: ";
    for ( auto interval : InsertInterval::insert( input, {4, 5} ) ) {
        cout << "[" << interval.start << "," << interval.end << "] ";
    }
    cout << endl;
    system( "pause" );
}
```

1.3 合并俩个区间链表 （经历过的面试题）

> 合并俩个有序的区间数组， 找交集

e.g.1

```
Input: arr1=[[1, 3], [5, 6], [7, 9]], arr2=[[2, 3], [5, 7]]
Output: [2, 3], [5, 6], [7, 7]
Explanation: The output list contains the common intervals between the two
lists.
```

e.g.2

```
Input: arr1=[[1, 3], [5, 7], [9, 12]], arr2=[[5, 10]]
Output: [5, 7], [9, 10]
Explanation: The output list contains the common intervals between the two
lists.
```

思路

这里有点像二路归并。
因为已经按照start排好序。
所以思路如下

1. 比较俩个数组的第一个区间。是否相交。
2. 如果相交。合并成一个大的区间。
3. 然后将相交的的区间插入即可。

```
using namespace std;

#include <iostream>
```

```cpp
#include <vector>
#include <algorithm>

class Interval {
  public:
    int start = 0;
    int end = 0;

    Interval( int start  = 0, int end  = 0 ) {
        this->start = start;
        this->end = end;
    }
    bool operator<(  const Interval &rhs )const {
        if ( start == rhs.start )
            return end < rhs.end;
        return start < rhs.start;
    }
};

class IntervalsIntersection {
  public:
    static vector<Interval> merge( const vector<Interval> &arr1, const
vector<Interval> &arr2 ) {
        vector<Interval> result;
        // TODO: Write your code here
        int x = 0;
        int y = 0;
        for ( ; x < arr1.size() && y < arr2.size(); ) {
            if ( arr1[x].start >= arr2[y].start && arr1[x].start <=
arr2[y].end  ||
                    arr2[x].start >= arr1[y].start && arr2[x].start <=
arr1[y].end ) {
                Interval cur_interval;
                cur_interval.start = max( arr1[x].start, arr2[y].start );
                cur_interval.end = min( arr1[x].end, arr2[y].end );
                result.push_back( cur_interval );
            }
            if ( arr1[x] < arr2[y] ) {
                x++;
            } else {
                y++;
            }
        }
        return result;
```

```
    }
};
int main( int argc, char *argv[] ) {
    vector<Interval> input1 = {{1, 3}, {5, 6}, {7, 9}};
    vector<Interval> input2 = {{2, 3}, {5, 7}};
    vector<Interval> result = IntervalsIntersection::merge( input1, input2 );
    cout << "Intervals Intersection: ";
    for ( auto interval : result ) {
        cout << "[" << interval.start << "," << interval.end << "] ";
    }
    cout << endl;

    input1 = {{1, 3}, {5, 7}, {9, 12}};
    input2 = {{5, 10}};
    result = IntervalsIntersection::merge( input1, input2 );
    cout << "Intervals Intersection: ";
    for ( auto interval : result ) {
        cout << "[" << interval.start << "," << interval.end << "] ";
    }
    system( "pause" );
}
```

1.4 合并俩个区间链表 （经历过的面试题）

> 合并俩个有序的区间数组， 找并集

e.g.1

```
Input: arr1=[[1, 3], [5, 6], [7, 9]], arr2=[[2, 3], [5, 7]]
Output: [1, 3], [5, 9]
Explanation: The output list contains the common intervals between the two
lists.
```

e.g.2

```
Input: arr1=[[1, 3], [5, 7], [9, 12]], arr2=[[5, 10]]
Output: [1, 3], [5, 12]
Explanation: The output list contains the common intervals between the two
lists.
```

思路

这里有点像二路归并。
因为已经按照start排好序。
所以思路如下

1. 比较俩个数组的第一个区间。是否相交。
2. 如果相交。合并成一个大的区间。

这里的思路是二路归并。因此需要重载<，来定义区间比较。即获得较小的区间。
最后需要处理如果余下一路的问题。

```
using namespace std;

#include <iostream>
#include <vector>
#include <algorithm>

class Interval {
  public:
    int start = 0;
    int end = 0;

    Interval( int start  = 0, int end  = 0 ) {
        this->start = start;
        this->end = end;
    }
    bool operator<(  const Interval &rhs )const {
        if ( start == rhs.start )
            return end < rhs.end;
        return start < rhs.start;
    }
};

class IntervalsIntersection {
  public:
    static vector<Interval> merge( const vector<Interval> &arr1, const
vector<Interval> &arr2 ) {
        vector<Interval> result;
        // TODO: Write your code here
        int x = 0;
        int y = 0;
        Interval curr_interval( 0, 0 );
        // 处理第一次
        if ( arr1[0] < arr2[0] ) {
```

```
            curr_interval = arr1[0];
            x++;
        } else {
            curr_interval = arr2[0];
            y++;
        }
        for ( ; x < arr1.size() && y < arr2.size(); ) {
            // 先选出一个小的
            Interval to_compare;
            if ( arr1[x] < arr2[y] ) {
                to_compare = arr1[x];
                x++;
            } else {
                to_compare = arr2[y];
                y++;
            }
            if ( to_compare.start <= curr_interval.end ) {
                curr_interval.end = std::max( to_compare.end,
curr_interval.end );
            } else {
                result.push_back( curr_interval );
                curr_interval = to_compare;
            }
        }
        // 找到剩余所有重合
        for ( ; x != arr1.size(); x++ ) {
            if ( arr1[x].start <= curr_interval.end ) {
                curr_interval.end = std::max( arr1[x].end, curr_interval.end
);
            }
        }
        for ( ; y != arr2.size(); y++ ) {
            if ( arr2[y].start <= curr_interval.end ) {
                curr_interval.end = std::max( arr2[y].end, curr_interval.end
);
            }
        }
        result.push_back( curr_interval );
        for ( ; x != arr1.size(); x++ ) {
            result.push_back( arr1[x] );
        }
        for ( ; y != arr2.size(); y++ ) {
            result.push_back( arr2[y] );
        };
```

```
        return result;
    }
};
```

1-5 冲突的会议

> 给定一系列区间，代表会议时间。判断一个人是否能参加所有的会议。

e.g.1

```
Appointments: [[1,4], [2,5], [7,9]]
Output: false
Explanation: Since [1,4] and [2,5] overlap, a person cannot attend both of
these appointments.
```

e.g.2

```
Appointments: [[6,7], [2,4], [8,12]]
Output: true
Explanation: None of the appointments overlap, therefore a person can attend
all of them.
```

e.g.3

```
Appointments: [[4,5], [2,3], [3,6]]
Output: false
Explanation: Since [4,5] and [3,6] overlap, a person cannot attend both of
these appointments.
```

思路

俩个思路

1. 按照start排序，然后遍历查看是否存在冲突。时间复杂度是O(logN)
2. 如果将所有时间点排在一个时间线上，可以发现，同一时间，只能经历一个start。

这里实现思路2。但是思路2存在局限。就是需要给定最大的结束时间。

```cpp
using namespace std;

#include <algorithm>
#include <iostream>
#include <vector>

class Interval {
  public:
    int start;
    int end;

    Interval( int start, int end ) {
        this->start = start;
        this->end = end;
    }
};

class ConflictingAppointments {
  public:
    static bool canAttendAllAppointments( vector<Interval> &intervals ) {
        // TODO: Write your code here
        vector<int> arr( 1000, 0 );
        int count = 0;
        for ( int i = 0; i < intervals.size(); i++ ) {
            int start = intervals[i].start;
            int end = intervals[i].end;
            if ( !arr[start] )
                arr[start] = 1;
            else
                return false;
            if ( !arr[end] )
                arr[end] = -1;
            else
                return false;
        }
        for ( int i = 0; i < arr.size(); i++ ) {
            count += arr[i];
            if ( count > 1 )
                return false;
        }
        return true;
    }
};
```

```
int main( int argc, char *argv[] ) {
    vector<Interval> intervals = {{1, 4}, {2, 5}, {7, 9}};
    bool result = ConflictingAppointments::canAttendAllAppointments(
intervals );
    cout << "Can attend all appointments: " << result << endl;

    intervals = {{6, 7}, {2, 4}, {8, 12}};
    result = ConflictingAppointments::canAttendAllAppointments( intervals );
    cout << "Can attend all appointments: " << result << endl;

    intervals = {{4, 5}, {2, 3}, {3, 6}};
    result = ConflictingAppointments::canAttendAllAppointments( intervals );
    cout << "Can attend all appointments: " << result << endl;

    system( "pause" );
}
```

1-6 会议安排

> 这算是比较经典的问题了。给定一些列区间。代表会议时间。问需要的最小会议室数目。

e.g.1

```
Meetings: [[1,4], [2,5], [7,9]]
Output: 2
Explanation: Since [1,4] and [2,5] overlap, we need two rooms to hold these
two meetings. [7,9] can occur in any of the two rooms later
```

e.g.2

```
Meetings: [[6,7], [2,4], [8,12]]
Output: 1
Explanation: None of the meetings overlap, therefore we only need one room to
hold all meetings.
```

e.g.3

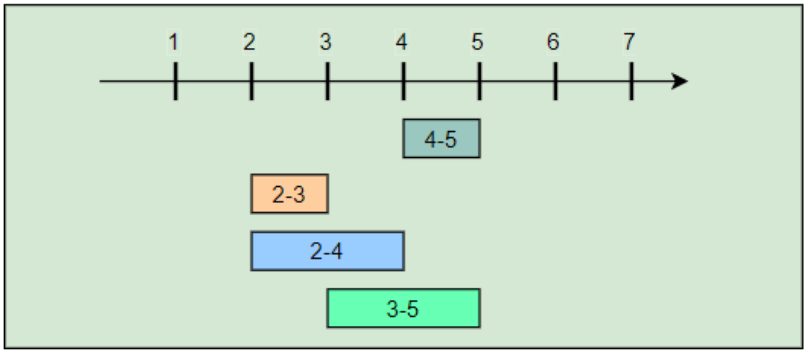
```
Meetings: [[1,4], [2,3], [3,6]]
Output:2
```

```
Explanation: Since [1,4] overlaps with the other two meetings [2,3] and
[3,6], we need two rooms to hold all the meetings.
```

e.g.4

```
Meetings: [[4,5], [2,3], [2,4], [3,5]]
Output: 2
Explanation: We will need one room for [2,3] and [3,5], and another room for
[2,4] and [4,5]. Here is a visual representation of Example 4:
```

思路



从这幅图，我们可以直接将这个建模成为在同一时间，重叠的最多区间数目。

假设我们从时间1开始遍历。
遇到2。cnt = 1。即遇到时间的起点。
遇到2。cnt = 2
遇到3。 cnt = 1。约到结束点了
遇到3。cnt = 2。从这里我们可以判断结束点的优先级高于起始点。（这里将会影响排序的优先级）
依次类推。发现我们的最大cnt = 2。

```
using namespace std;

#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>

class Meeting {
  public:
    int start = 0;
```

```
    int end = 0;

    Meeting( int start, int end ) {
        this->start = start;
        this->end = end;
    }
};

struct Timetype {
    int time;
    int type;
};
class MinimumMeetingRooms {
  public:
    static int findMinimumMeetingRooms( vector<Meeting> &meetings ) {
        // TODO: Write your code here
        vector<Timetype> time_vec;
        time_vec.reserve( meetings.size() * 2 );
        for ( int i = 0; i < meetings.size(); i++ ) {
            time_vec.push_back( { meetings[i].start, 0 } );
            time_vec.push_back( { meetings[i].end, 1 } );
        }
        std::sort( time_vec.begin(), time_vec.end(), []( const Timetype &x1,
const Timetype& x2 ) {
            if ( x1.time == x2.time )
                return x1.type > x2.type;
            return x1.time < x2.time;
        } );
        int max_meeting_rooms = 0;
        int cnt = 0;
        for ( int i = 0; i < time_vec.size(); i++ ) {
            if ( time_vec[i].type == 0 )
                cnt++;
            else
                cnt--;
            max_meeting_rooms = std::max( max_meeting_rooms, cnt );
        }
        return max_meeting_rooms;
    }
};

int main( int argc, char *argv[] ) {

    vector<Meeting> input;
```

```
    int result;

    input = {{1, 4}, {2, 5}, {7, 9}};
    result = MinimumMeetingRooms::findMinimumMeetingRooms( input );
    cout << "Minimum meeting rooms required: " << result << endl;

    input = {{6, 7}, {2, 4}, {8, 12}};
    result = MinimumMeetingRooms::findMinimumMeetingRooms( input );
    cout << "Minimum meeting rooms required: " << result << endl;

    input = {{1, 4}, {2, 3}, {3, 6}};
    result = MinimumMeetingRooms::findMinimumMeetingRooms( input );
    cout << "Minimum meeting rooms required: " << result << endl;

    input = {{4, 5}, {2, 3}, {2, 4}, {3, 5}};
    result = MinimumMeetingRooms::findMinimumMeetingRooms( input );
    cout << "Minimum meeting rooms required: " << result << endl;

    system( "pause" );
}
```

思路2。
对于[2 3] [2 4][3 5][4 5] 来说。对于这种题，我们一般喜欢构造一个很长的例子，来寻找规律。

1. 将[2, 3] 安排到第一个会议室
2. [2,4]和[2,3]冲突，所以需要安排到一个新的会议室。怎么判断冲突呢，发现start:2 < active_room::end:3
3. 安排[3, 5]的时候，我们发现我们需要查看已经安排会议室的是否有空闲出来。从这里开始，思路就出来了。怎么判断空闲呢? start:3 >= active_room::min_end:3。很明显，这里应该while循环，将所有空闲的全部腾出去。

```
active_list.push_back(meeting[0]);
for(int i = 1; i < size; i++)
{
    if(active_list.is_expire(meeting[0].start))
        ; // 查看是否有会议过期
    if(当前会议是否和active_list中的冲突)
       push(meeting[i]) ;//这里只要start<活动会议中的最小的end就有冲突
    update(min_room);

}
```

```
using namespace std;

#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>

class Meeting {
  public:
    int start = 0;
    int end = 0;

    Meeting( int start, int end ) {
        this->start = start;
        this->end = end;
    }
};

class MinimumMeetingRooms {
  public:
    struct endCompare {
        bool operator()( const Meeting &x, const Meeting &y ) {
            return x.end > y.end;
        }
    };

    static int findMinimumMeetingRooms( vector<Meeting> &meetings ) {
        if ( meetings.empty() ) {
            return 0;
        }

        // sort the meetings by start time
        sort( meetings.begin(), meetings.end(),
        []( const Meeting &x, const Meeting &y ) {
            return x.start < y.start;
        } );

        int minRooms = 0;
        priority_queue<Meeting, vector<Meeting>, endCompare> minHeap;
        for ( auto meeting : meetings ) {
            // remove all meetings that have ended
            while ( !minHeap.empty() && meeting.start >= minHeap.top().end )
{
                minHeap.pop();
```

```
            }
            // add the current meeting into the minHeap
            minHeap.push( meeting );
            // all active meeting are in the minHeap, so we need rooms for
all of them.
            minRooms = max( minRooms, ( int )minHeap.size() );
        }

        return minRooms;
    }
};

int main( int argc, char *argv[] ) {
    vector<Meeting> input = {{4, 5}, {2, 3}, {2, 4}, {3, 5}};
    int result = MinimumMeetingRooms::findMinimumMeetingRooms( input );
    cout << "Minimum meeting rooms required: " << result << endl;

    input = {{1, 4}, {2, 5}, {7, 9}};
    result = MinimumMeetingRooms::findMinimumMeetingRooms( input );
    cout << "Minimum meeting rooms required: " << result << endl;

    input = {{6, 7}, {2, 4}, {8, 12}};
    result = MinimumMeetingRooms::findMinimumMeetingRooms( input );
    cout << "Minimum meeting rooms required: " << result << endl;

    input = {{1, 4}, {2, 3}, {3, 6}};
    result = MinimumMeetingRooms::findMinimumMeetingRooms( input );
    cout << "Minimum meeting rooms required: " << result << endl;

    input = {{4, 5}, {2, 3}, {2, 4}, {3, 5}};
    result = MinimumMeetingRooms::findMinimumMeetingRooms( input );
    cout << "Minimum meeting rooms required: " << result << endl;
}
```

1-7 max CPU load

> 给定一系列job。有3个field——start，end，CPU load。寻找最大的CPU，load。多个任务可以同时工作。

e.g.1

```
Jobs: [[1,4,3], [2,5,4], [7,9,6]]
Output: 7
Explanation: Since [1,4,3] and [2,5,4] overlap, their maximum CPU load
(3+4=7) will be when both the jobs are running at the same time i.e., during
the time interval (2,4).
```

e.g.2

```
Jobs: [[6,7,10], [2,4,11], [8,12,15]]
Output: 15
Explanation: None of the jobs overlap, therefore we will take the maximum
load of any job which is 15.
```

e.g.3

```
Jobs: [[1,4,2], [2,4,1], [3,6,5]]
Output: 8
Explanation: Maximum CPU load will be 8 as all jobs overlap during the time
interval [3,4].
```

思路

和上道题完全一模一样，只不过上道题的代价是1，这里新增加了代价。

```
using namespace std;

#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>

class Job {
  public:
    int start = 0;
    int end = 0;
    int cpuLoad = 0;

    Job( int start, int end, int cpuLoad ) {
```

```
        this->start = start;
        this->end = end;
        this->cpuLoad = cpuLoad;
    }
};

struct JobCompare {
    bool operator()( const Job &x1, const Job &x2 ) {
        return x1.start < x2.start;
    }
};
class MaximumCPULoad {
  public:
    static int findMaxCPULoad( vector<Job> &jobs ) {
        int maxCPULoad = 0;
        // TODO: Write your code here
        sort( jobs.begin(), jobs.end(), []( const Job &x1, const Job &x2 ) {
            return x1.start < x2.start;
        } );
        priority_queue<Job, vector<Job>, JobCompare> min_heap;
        int min_heap_load = 0;
        min_heap.push( jobs[0] );
        min_heap_load += jobs[0].cpuLoad;
        for ( int i = 1; i < jobs.size(); i++ ) {
            while ( !min_heap.empty() && jobs[i].start > min_heap.top().end )
{
                min_heap_load -= min_heap.top().cpuLoad;
                min_heap.pop();
            }
            min_heap.push( jobs[i] );
            min_heap_load += jobs[i].cpuLoad;

            maxCPULoad = std::max( maxCPULoad, min_heap_load );
        }
        return maxCPULoad;
    }
};

int main( int argc, char *argv[] ) {
    vector<Job> input = {{1, 4, 3}, {7, 9, 6}, {2, 5, 4}};
    cout << "Maximum CPU load at any time: " <<
MaximumCPULoad::findMaxCPULoad( input ) << endl;

    input = {{6, 7, 10}, {8, 12, 15}, {2, 4, 11}};
```

```
    cout << "Maximum CPU load at any time: " <<
MaximumCPULoad::findMaxCPULoad( input ) << endl;

    input = {{1, 4, 2}, {3, 6, 5}, {2, 4, 1}};
    cout << "Maximum CPU load at any time: " <<
MaximumCPULoad::findMaxCPULoad( input ) << endl;
    system( "pause" );
}
```

1-8 员工的空闲时间

> 给K个员工的**工作区间**。找到所有员工都**空闲的时间段**。每个员工的工作时间都是经过start排序后的。

e.g.1

```
Input: Employee Working Hours=[[[1,3], [5,6]], [[2,3], [6,8]]]
Output: [3,5]
Explanation: Both the employess are free between [3,5].
```

e.g.2

```
Input: Employee Working Hours=[[[1,3], [9,12]], [[2,4]], [[6,8]]]
Output: [4,6], [8,9]
Explanation: All employess are free between [4,6] and [8,9].
```

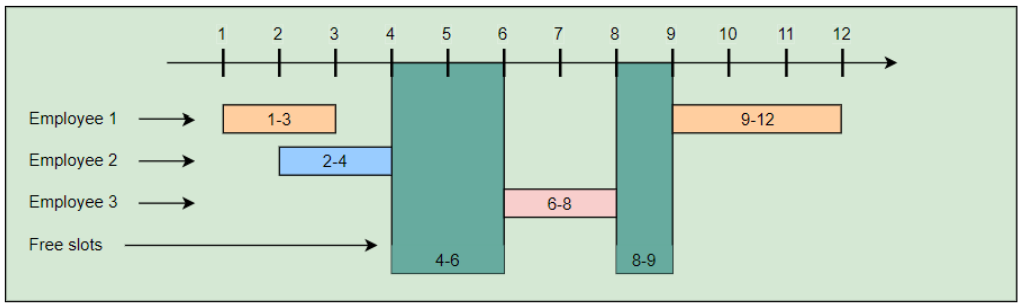
e.g.3

```
Input: Employee Working Hours=[[[1,3]], [[2,4]], [[3,5], [7,9]]]
Output: [5,7]
Explanation: All employess are free between [5,7].
```

思路

注意，题目给的是工作时间。去找空闲时间。

我们将每个员工的区间画在时间点上。



不难看出，实际上我们就是在找没有区间覆盖的地方，和之前的做法刚好相反。

1. 直接将所有区间放到一个数组中，然后去排序。再按之前的思路去做。
2. 但实际上，这里实际上是K个排好序的区间。一般K个有序数组直接会让我们思考到K路归并——最小堆。

提供一个经典自作聪明解法。这个解法虽然能解出来。但是相比于上面俩个思路的时间复杂度要高。
如果你数据结构学的好的话，一眼看出来这里的时间复杂度是O(N * logN * logN)
因为每次插入一次数据，都要进行一次调整。共调整N次。每次调整i次，i = 0-N-1。显然是logN * logN的复杂度。

```
using namespace std;

#include <iostream>
#include <queue>
#include <vector>

class Interval {
  public:
    int start = 0;
    int end = 0;

    Interval( int start, int end, int employee = 0, int n_work_interval = 0 )
{
        this->start = start;
        this->end = end;
        this->employee = employee;
        this->n_work_interval = n_work_interval;
    }
    bool is_overlap( const Interval &rhs ) {
        return rhs.start <= this->end;
    }
    int employee;
    int n_work_interval;
};
```

```
struct IntervalCompare {
    bool operator()( const Interval &x1, const Interval &x2 ) {
        return x1.start > x2.start;
    }
};

class EmployeeFreeTime {
  public:
    static vector<Interval> findEmployeeFreeTime( const
vector<vector<Interval>> &schedule ) {
        vector<Interval> result;
        // TODO: Write your code here
        priority_queue<Interval, vector<Interval>, IntervalCompare> min_heap;
        for ( int i = 0; i < schedule.size(); i++ ) {
            for ( int k = 0; k < schedule[i].size(); k++ ) {
                min_heap.push( Interval( schedule[i][k].start, schedule[i]
[k].end, i, k ) );
            }
        }

        Interval curr_interval = min_heap.top();
        min_heap.pop();
        while ( !min_heap.empty() ) {
            Interval next_interval = min_heap.top();
            min_heap.pop();

            if ( curr_interval.is_overlap( next_interval ) ) {
                if( next_interval.end > curr_interval.end )
                    curr_interval = next_interval;
            } else {
                Interval res( curr_interval.end, next_interval.start );
                result.push_back( res );
                curr_interval = next_interval;
            }
        }
        return result;
    }
};

int main( int argc, char *argv[] ) {
    vector<vector<Interval>> input = {{{1, 3}, {5, 6}}, {{2, 3}, {6, 8}}};
    vector<Interval> result = EmployeeFreeTime::findEmployeeFreeTime( input
);
    cout << "Free intervals: ";
```

```
    for ( auto interval : result ) {
        cout << "[" << interval.start << ", " << interval.end << "] ";
    }
    cout << endl;

    input = {{{1, 3}, {9, 12}}, {{2, 4}}, {{6, 8}}};
    result = EmployeeFreeTime::findEmployeeFreeTime( input );
    cout << "Free intervals: ";
    for ( auto interval : result ) {
        cout << "[" << interval.start << ", " << interval.end << "] ";
    }
    cout << endl;

    input = {{{1, 3}}, {{2, 4}}, {{3, 5}, {7, 9}}};
    result = EmployeeFreeTime::findEmployeeFreeTime( input );
    cout << "Free intervals: ";
    for ( auto interval : result ) {
        cout << "[" << interval.start << ", " << interval.end << "] ";
    }
    system( "pause" );
}
```

真正的K路归并堆只做K大小。其时间复杂度只有O(N * logK)
对比使用排序的O(N * logN)。非常值得使用了。

```
using namespace std;

#include <iostream>
#include <queue>
#include <vector>

class Interval {
  public:
    int start = 0;
    int end = 0;

    Interval( int start, int end, int employee = 0, int n_work_interval = 0 )
{
        this->start = start;
        this->end = end;
        this->employee = employee;
        this->n_work_interval = n_work_interval;
    }
```

```
    bool is_overlap( const Interval &rhs ) {
        return rhs.start <= this->end;
    }
    int employee;
    int n_work_interval;
};
struct IntervalCompare {
    bool operator()( const Interval &x1, const Interval &x2 ) {
        return x1.start > x2.start;
    }
};

class EmployeeFreeTime {
  public:
    static vector<Interval> findEmployeeFreeTime( const
vector<vector<Interval>> &schedule ) {
        vector<Interval> result;
        // TODO: Write your code here
        priority_queue<Interval, vector<Interval>, IntervalCompare> min_heap;
        for ( int i = 0; i < schedule.size(); i++ ) {
            min_heap.push( Interval( schedule[i][0].start, schedule[i]
[0].end, i, 0 ) );
        }

        Interval curr_interval = min_heap.top();
        min_heap.pop();
        // 补充
        if( curr_interval.n_work_interval + 1 <
schedule[curr_interval.employee].size() )
            min_heap.push(
                Interval(
                    schedule[curr_interval.employee]
[curr_interval.n_work_interval + 1].start,
                    schedule[curr_interval.employee]
[curr_interval.n_work_interval + 1].end,
                    curr_interval.employee,
                    curr_interval.n_work_interval + 1
                )
            );
        while ( !min_heap.empty() ) {
            // 选取下一个最小
            Interval next_interval = min_heap.top();
            min_heap.pop();
```

```
            // 比较
            if ( curr_interval.is_overlap( next_interval ) ) {
                if( next_interval.end > curr_interval.end )
                    curr_interval = next_interval;
            } else {
                Interval res( curr_interval.end, next_interval.start );
                result.push_back( res );
                curr_interval = next_interval;
            }

            // 补充
            if( curr_interval.n_work_interval + 1 <
schedule[curr_interval.employee].size() )
                min_heap.push(
                    Interval(
                        schedule[curr_interval.employee]
[curr_interval.n_work_interval + 1].start,
                        schedule[curr_interval.employee]
[curr_interval.n_work_interval + 1].end,
                        curr_interval.employee,
                        curr_interval.n_work_interval + 1
                    )
                );

        }
        return result;
    }
};

int main( int argc, char *argv[] ) {
    vector<vector<Interval>> input = { {{1, 3}, {5, 6}}, {{2, 3}, {6, 8}} };
    vector<Interval> result = EmployeeFreeTime::findEmployeeFreeTime( input
);
    cout << "Free intervals: ";
    for ( auto interval : result ) {
        cout << "[" << interval.start << ", " << interval.end << "] ";
    }
    cout << endl;

    input = { {{1, 3}, {9, 12}}, {{2, 4}}, {{6, 8}} };
    result = EmployeeFreeTime::findEmployeeFreeTime( input );
    cout << "Free intervals: ";
    for ( auto interval : result ) {
        cout << "[" << interval.start << ", " << interval.end << "] ";
```

```
    }
    cout << endl;

    input = { {{1, 3}}, {{2, 4}}, {{3, 5}, {7, 9}} };
    result = EmployeeFreeTime::findEmployeeFreeTime( input );
    cout << "Free intervals: ";
    for ( auto interval : result ) {
        cout << "[" << interval.start << ", " << interval.end << "] ";
    }
    cout << endl;
    input = { {{1, 3}}, {{2, 4}}, {{3, 5}, {7, 9}}, {{6, 7}, {11, 12}} };
    result = EmployeeFreeTime::findEmployeeFreeTime( input );
    cout << "Free intervals: ";
    for ( auto interval : result ) {
        cout << "[" << interval.start << ", " << interval.end << "] ";
    }
    cout << endl;
    system( "pause" );
}
```

2. 总结

1. 明白区间之间的关系。6种小情况。重点掌握相交的4种情况，以及当按照start排序时候，相交的情况判断。只需要比较arr[i+1].start <= arr[i].end就存在交集。
2. 学会使用自定义的sort排序，在很多时候，我们需要定义这种排序，经常以start作为排序。有时候还会附加其他东西。当按start排序时，情况一般就会比较明了。
3. 学会自定义区间的比较。
4. 学会使用优先级队列。注意优先级队列的插入，和删除都是logN的操作复杂度。
5. 在多数组问题上，区间排好序后，可以等价为K路归并问题。