【DP/贪心】-观看文艺汇演 题目描

述与示例

某公园将举行多场文艺表演，很多演出都是同时进行，一个人只能同时观看一场演出，且不能迟到早退，由于演出分布在不同的演出场地，所以连续观看的演出最少有 `15` 分钟的时间间隔，小明是一个狂热的文艺迷，想观看尽可能多的演出。现给出演出时间表，请帮小明计算他最多能观看几场演出。

输入

第一行为一个数 `N`，表示演出场数，`1 <= N <= 1000`。

接下来 `N` 行，每行两个空格分割的整数，第一个整数 `T` 表示演出的开始时间，第二个整数 `L` 表示演出的持续时间，`T` 和 `L` 的单位为分钟，`0 <= T <= 1440, 0 < L <= 100`。

输出

最多能观看的演出场数。

示例一

输入

```
2
720 120
840 120
```

输出

```
1
```

说明

第一场演出开始时间是第 `720` 分钟，经过 `120` 分钟演出结束，即第 `840` 分钟结束，此时还需要 `15` 分钟的间隔时间，即要等到第 `855` 分钟才可以看下一场演出，故来不及看第二场在第 `840` 分钟开始的演出。最多只能看 `1` 场演出。

示例二

输入

```
2
20 60
100 60
```

输出

```
2
```

说明

第一场演出开始时间是第 `20` 分钟，经过 `60` 分钟演出结束，即第 `80` 分钟结束，此时还需要 `15` 分钟的间隔时间，即要等到第 `95` 分钟才可以看下一场演出，第二场演出在第 `100` 分钟开始的演出，赶得上观看第二场演出。最多可以观看 `2` 场演出。

示例三

输入

```
4
10 20
100 20
150 60
80 40
```

输出

```
3
```

解题思路

> 注意，本题和LC435. 无重叠区间几乎完全一致。

原始数据处理

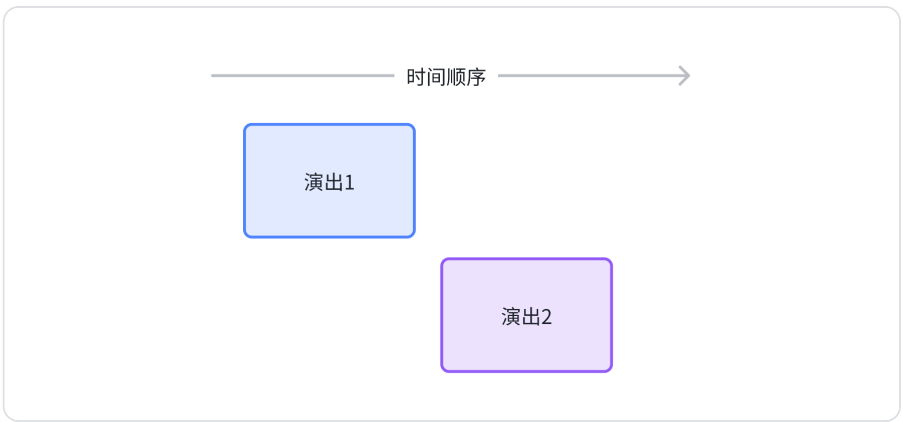
我们可以储存每一场演出的开始和结束时间，即按照 `[start, end]` 的方式进行储存，储存在列表 `intervals` 中。由于题目要求每间隔 `15` 分钟才能够看下一场演出，所以我们可以把每一场演出的结束时间再加上 `15` 分钟，这样题目就转变为：考虑所有不重叠的 `[start, end]` 区间的**最大数目**。处理输入的代码如下

```python
N = int(input())
intervals = list()
for _ in range(N):
    start, during = map(int, input().split())
    end = start + during + 15
    intervals.append((start, end))
```

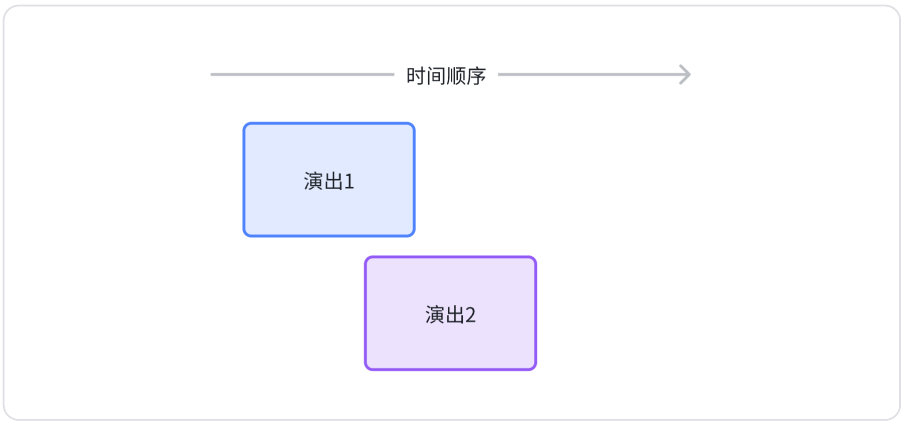
贪心思想求解问题

为了方便我们**贪心地思考问题**，我们先按照开始时间 `start` 从小到大对间隔列表 `intervals` 进行排序。然后我们考虑相邻的两场演出，`[start1, end1]` 和 `[start2, end2]`，由于 `intervals` 已经排序，必然存在 `start1 <= start2` 成立，故这两场演出之间的关系存只有以下**三种可能性**：

1. `start1 < end1 <= start2 < end2`，即演出 `2` 的开始时间在演出 `1` 的结束时间之后。故看完演出 `1`，可以继续看演出 `2`。



2. `start1 <= start2 <= end1 <= end2`，即演出 `2` 的开始时间在演出 `1` 的结束时间之前，但演出 `2` 的结束时间在演出 `1` 的结束时间之后。故看完演出 `1` 之后，没办法观看演出 `2` 。



3. `start1 <= start2 <= end2 <= end1`，即演出 `2` 的开始时间和结束时间均在演出 `1` 的结束时间之前。故看完演出 `1` 之后，没办法观看演出 `2` 。为了尽可能多地看更多的演出，选择演出 `2` 来观看会比选择演出 `1` 更好，因为演出 `2` 的结束时间更早，有充裕的时间去观看后续的演出。

理解了相邻两场演出的三种可能性之后，我们发现解决问题的关键实际上在于考虑演出 `1` 的结束时间 `end1` 和演出 `2` 的间隔 `[start2, end2]` 之间的关系：

1. `end1` 在 `[start2, end2]` 之前
2. `end1` 在 `[start2, end2]` 之间
3. `end1` 在 `[start2, end2]` 之后

由于我们需要遍历排序后的间隔列表 `intervals` 中的每一个间隔 `[start, end]` ，因此可以维护变量 `pre_end` ，表示上一场演出的结束时间。初始化 `pre_end = -inf` ，表示第一场演出始终可以观看。

考虑当前间隔 `[start, end]` 和上一场演出结束时间 `pre_end` 之间的关系，我们可以得到以下逻辑：

1. 当 `pre_end` 在 `[start, end]` 之前，我们**可以**选择当前的演出 `[start, end]` 进行观看

- 能观看的演出场次 `ans += 1`。
- 由于选择了 `[start, end]` 进行观看，下一场演出的观看时间应该由当前的 `end` 决定，即对于下一场演出而言，当前结束时间 `end` 是上一场演出的结束时间，故更新 `pre_end = end`。

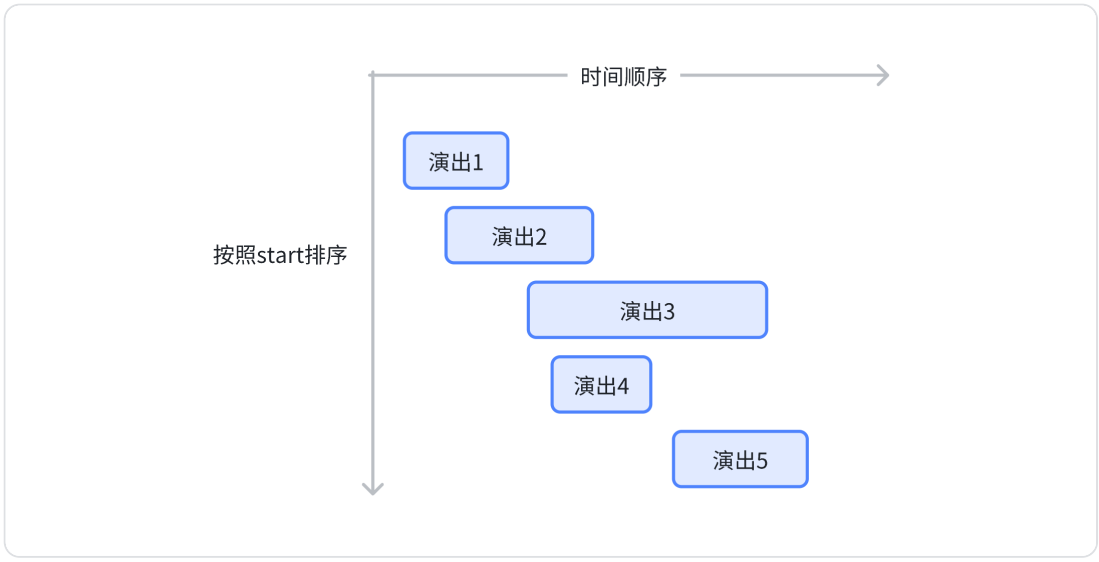
2. 当 `pre_end` 在 `[start, end]` 之间，我们**不能**选择当前的演出 `[start, end]` 进行观看
   - 由于 `pre_end ≤ end`，因此我们保留之前的 `pre_end`，作为判断下一场演出是否能观看的依据。故无需做任何事情。
3. 当 `pre_end` 在 `[start, end]` 之后，我们**不能**选择当前的演出 `[start, end]` 进行观看
   - 由于 `pre_end > end`，选择 `end` 作为判断下一场演出是否能观看的依据是更佳的选择，即我们不去选择观看 `pre_end` 所对应的之前某场演出，而选择观看当前的演出 `[start, end]`，这样的选择有利于后面留出充裕的时间来尽可能地观看更多演出，故更新 `pre_end = end`。

整理上述逻辑后，代码为

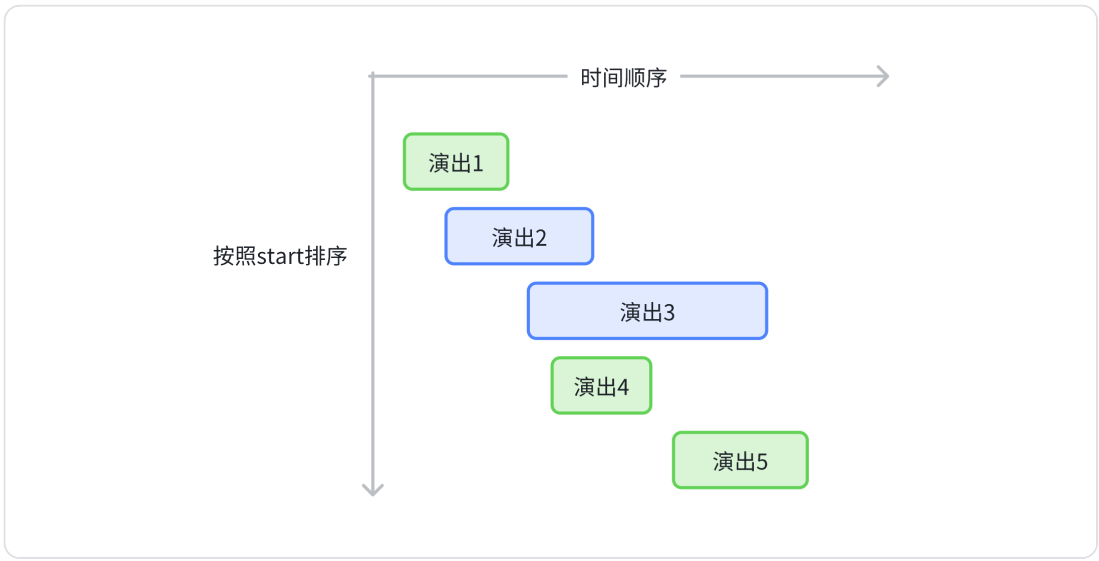
```python
for start, end in intervals:
    if start >= pre_end:
        ans += 1
        pre_end = end
    elif start < pre_end <= end:
        continue
    elif pre_end > end:
        pre_end = end
```

动态规划求解问题

实际上，当我们对间隔列表 `intervals` 排序之后，我们也可以把这个问题当作经典的LIS问题（LC300. 最长递增子序列）进行处理。



譬如对于上述例子，我们所选的演出应该为演出 `1` 、 `4` 、 `5`



换句话说，我们需要找到尽可能多的演出区间，所有演出区间均需要满足 `start ≥ pre_end` ，其中 `start` 为第 `i` 个区间的开始时间， `pre_end` 为上一个区间即第 `i-1` 个区间的结束时间。这是一个非常自然的LIS问题，故也可以用dp来解决问题。

我们考虑动态规划三部曲：

1. **`dp` 数组的含义是什么？**

- `dp` 数组是一个长度为 `n` 的一维列表， `dp[i]` 表示包含了第 `i` 场演出 `intervals[i]` 的最长无重叠演出数目。

2. **动态转移方程是什么？**

- 包含了第 `i` 场演出 `intervals[i]` 的最长无重叠演出数目，由前面的 `i-1` 场演出中（用索引 `j` 表示），结束时间 `intervals[j][1]` 小于当前演出开始时间 `intervals[i][0]` 且 `dp[j]` 最大的那场演出决定。

```python
for i in range(1, N):
    temp = 0
    for j in range(i):
        if intervals[j][1] <= intervals[i][0]:
            temp = max(dp[j], temp)
    dp[i] = temp + 1
```

3. `dp` **数组如何初始化?**

- 包含第 `1` 场演出的最长无重叠演出数目为 `1` 。

```python
dp[0] = 1
```

代码

解法一：贪心

Python

```python
# 题目：2024E-观看文艺汇演
# 分值：200
# 作者：许老师-闭着眼睛学数理化
# 算法：贪心
# 代码看不懂的地方，请直接在群上提问


from math import inf

# 输入演出的数目
N = int(input())

# 初始化间隔列表
intervals = list()
for _ in range(N):
    start, during = map(int, input().split())
    # 对于每一个结束时间都+15后再储存，方便后续进行比较
    end = start + during + 15
    intervals.append((start, end))

# 对intervals进行排序
```

```python
intervals.sort()
ans = 0

# 初始化【上个区间结束时间】为pre_end = -inf
pre_end = -inf

# 遍历所有区间的起始时间和结束时间
for start, end in intervals:
    # 如果【当前起始时间】大于等于【上次结束时间】
    # 可以选择【当前区间】进行观看，接在【上个区间】后面
    # 同时 pre_end 应该修改为【当前结束时间】
    # 作为下一个区间的【上次结束时间】
    if start >= pre_end:
        ans += 1
        pre_end = end
    # 如果【上次结束时间】正好落在【当前区间】内
    # 则不能选择【当前区间】进行观看，保留【上个区间】
    # 无需做任何事情
    elif start < pre_end <= end:
        continue
    # 如果【上次结束时间】大于【当前结束时间】
    # 则应该选择【当前区间】进行观看，而不应该选择【上个区间】
    # 故 pre_end 应该修改为【当前结束时间】
    # 作为下一个区间的【上次结束时间】
    elif pre_end > end:
        pre_end = end

print(ans)
```

Java

```java
import java.util.*;

class Interval implements Comparable<Interval> {
    int start;
    int end;

    public Interval(int start, int end) {
        this.start = start;
        this.end = end;
    }

    public int compareTo(Interval other) {
        return this.start - other.start;
    }
```

```java
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int N = scanner.nextInt();

        List<Interval> intervals = new ArrayList<>();
        for (int i = 0; i < N; i++) {
            int start = scanner.nextInt();
            int during = scanner.nextInt();
            int end = start + during + 15;
            intervals.add(new Interval(start, end));
        }

        Collections.sort(intervals);

        int ans = 0;
        int preEnd = Integer.MIN_VALUE;

        for (Interval interval : intervals) {
            int start = interval.start;
            int end = interval.end;

            if (start >= preEnd) {
                ans++;
                preEnd = end;
            } else if (start < preEnd && preEnd <= end) {
                continue;
            } else if (preEnd > end) {
                preEnd = end;
            }
        }

        System.out.println(ans);
    }
}

```

C++

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

```

```cpp
using namespace std;

struct Interval {
    int start;
    int end;

    Interval(int s, int e) : start(s), end(e) {}
};

bool compareIntervals(const Interval& a, const Interval& b) {
    return a.start < b.start;
}

int main() {
    int N;
    cin >> N;

    vector<Interval> intervals;
    for (int i = 0; i < N; ++i) {
        int start, during;
        cin >> start >> during;
        int end = start + during + 15;
        intervals.push_back(Interval(start, end));
    }

    sort(intervals.begin(), intervals.end(), compareIntervals);

    int ans = 0;
    int preEnd = -1e9;

    for (const auto& interval : intervals) {
        int start = interval.start;
        int end = interval.end;

        if (start >= preEnd) {
            ans++;
            preEnd = end;
        } else if (start < preEnd && preEnd <= end) {
            continue;
        } else if (preEnd > end) {
            preEnd = end;
        }
    }

    cout << ans << endl;

    return 0;
```

```
}

```

时空复杂度

时间复杂度：`O(NlogN)`。排序时间复杂度。

空间复杂度：`O(1)`。仅需要用到若干常数变量。

解法二：DP

Python

```python
# 题目：2024E-观看文艺汇演
# 分值：200
# 作者：许老师-闭着眼睛学数理化
# 算法：dp(LIS问题)
# 代码看不懂的地方，请直接在群上提问


# 输入演出的数目
N = int(input())

# 初始化间隔列表
intervals = list()
for _ in range(N):
    start, during = map(int, input().split())
    # 对于每一个结束时间都+15后再储存，方便后续进行比较
    end = start + during + 15
    intervals.append((start, end))

# 对intervals进行排序
intervals.sort()
ans = 0

# 初始化长度为N的dp数组
dp = N * [0]
# 包含第1场演出的最长无重叠演出数目为1
dp[0] = 1

```

```python
# 遍历所有演出
for i in range(1, N):
    # 初始化变量temp，用于找到前面的i-1场演出中，最长无重叠的演出场次
    temp = 0
    # 对于每一场演出i，遍历其前面的i-1场演出
    for j in range(i):
        # 如果演出j的结束时间，小于等于当前演出i的开始时间
        if intervals[j][1] <= intervals[i][0]:
            # 则更新temp
            temp = max(dp[j], temp)
    # 结束上述循环后，还需要考虑本场演出本身
    dp[i] = temp + 1


# dp数组中的最大值，即为最长无重叠的演出场次
# 也就是能够观看的最多的演出场次
print(max(dp))
```

Java

```java
import java.util.*;

class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int N = scanner.nextInt();

        List<int[]> intervals = new ArrayList<>();
        for (int i = 0; i < N; i++) {
            int start = scanner.nextInt();
            int during = scanner.nextInt();
            int end = start + during + 15;
            intervals.add(new int[]{start, end});
        }

        intervals.sort(Comparator.comparingInt(a -> a[0]));

        int[] dp = new int[N];
        dp[0] = 1;

        for (int i = 1; i < N; i++) {
            int temp = 0;
            for (int j = 0; j < i; j++) {
                if (intervals.get(j)[1] <= intervals.get(i)[0]) {
                    temp = Math.max(dp[j], temp);
```

```
                }
            }
            dp[i] = temp + 1;
        }

        int maxWatch = Arrays.stream(dp).max().orElse(0);
        System.out.println(maxWatch);
    }
}

```

C++

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

using namespace std;

int main() {
    int N;
    cin >> N;

    vector<pair<int, int>> intervals;
    for (int i = 0; i < N; ++i) {
        int start, during;
        cin >> start >> during;
        int end = start + during + 15;
        intervals.push_back({start, end});
    }

    sort(intervals.begin(), intervals.end());

    vector<int> dp(N, 0);
    dp[0] = 1;

    for (int i = 1; i < N; ++i) {
        int temp = 0;
        for (int j = 0; j < i; ++j) {
            if (intervals[j].second <= intervals[i].first) {
                temp = max(dp[j], temp);
            }
        }
        dp[i] = temp + 1;
    }
```

```
33
34      int maxWatch = *max_element(dp.begin(), dp.end());
35      cout << maxWatch << endl;
36
37      return 0;
38  }
39
```

时空复杂度

时间复杂度：`O(N^2)`。dp过程需要进行双重循环。

空间复杂度：`O(N)`。dp数组所占空间。