09 - Weighted Interval Scheduling

动态规划（Dynamic Programming）

动态规划的范式是：

- 给定一个问题 P，定义一系列子问题，具有以下属性：
  - 子问题按照从小到大的顺序排列
  - 最大的子问题就是我们的原始问题 P
  - 子问题的最优解可以由子子问题的最优解构建（最优子结构）
- 从最小的子问题开始求解，解决一个子问题时，存储其解，并使用它来解决更大的子问题

加权区间调度（Weighted Interval Scheduling）

我们有一组请求 $\{1, 2, ..., n\}$
请求 i 有一个开始时间 $s(i)$、一个结束时间 $f(i)$、一个值 $v(i)$

- 另一种看法：每个请求都是一个区间 $[s(i), j(i)]$ 并且带有一个关联的值 $v(i)$

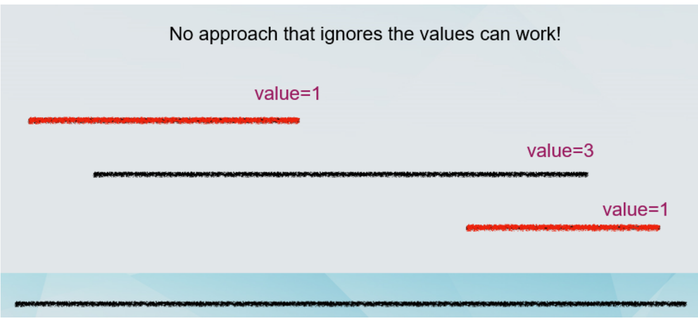
如果两个请求 i 和 j 各自的区间不重叠，则他们是兼容的
我们的目标是输出一个调度，使兼容区间的总值最大化

贪心算法

我们之前学习过基本的区间调度贪心算法：

- 最早开始时间
- 最短区间
- 最少冲突数
- 最早结束时间（最优）
- 最大值（新增）

但是在加权区间调度问题上，最早结束时间就不奏效了

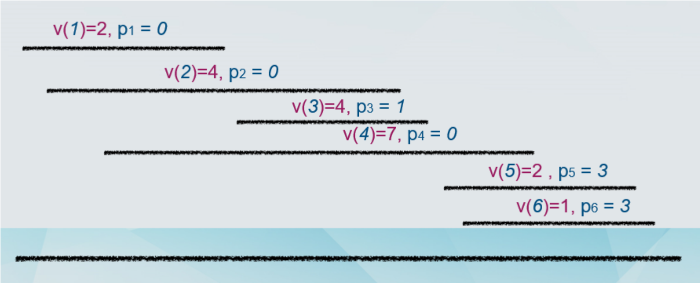


最大值

考虑按非递减结束时间排序的区间 $f(1) \leq f(2) \leq \cdots \leq f(n)$
对于一个区间 $j = [s(j), f(j)]$，令 p_j 为满足区间 i 和 j 不相交的最大索引 $i < j$，即

- i 是在 j 开始之前结束的排序中的最后一个区间

- 如果不存在这样的区间，则定义 $p_i = 0$



我们令 O 为最优调度：O 要么包含区间 n，要么不包含

$$O = \begin{cases} O(1, \ldots, p_n) \cup \{n\} \\ O(1, \ldots, n-1) \end{cases}$$

$n \in O$

这意味着

- 任何与 n 重叠的区间都不能在 O 中
- 任何索引大于 p_n 的区间 j 都不能在 O 中
- O 包含子问题 $\{1, 2, \ldots, p_n\}$ 的一个最优解 O'
    - 这里，p_n 是结束时间早于 n 开始时间的最大索引（上图中最后两个区间有解释），因此，这个子问题是不包括 n 的
    - 因为否则我们可以用 $O' \cup \{n\}$ 替换 O，从而获得更优的解

让我们用 $O(i, \ldots, j)$ 表示排序后的区间 $i, \ldots, j$ 上的最优解
则，$O = O(1, \ldots, p_n) + n$

$n \notin O$

那么 $O = O(1, \ldots, n-1)$
既然 n 没有被选择，那么 $1, \ldots, n-1$ 都可以被选择
如果 O 不是这些区间的最优调度，会发生了冲突。这是因为 n 不能被选择，我们无法通过并入 n 来使 O 最优

构建一个解决方案

- 因此，为了找到 O，只需查看更小的问题并找到某个 j 的 $O(1, \ldots, j)$
- 令 O_j 表示 $O(1, \ldots, j)$ 的简写，并令 $\mathrm{OPT}(j)$ 表示其总值
- 定义 $\mathrm{OPT}(0) = 0$
- 然后，$O = O_n$ 的值为 $\mathrm{OPT}(n)$

判断j 是否在 O 中

我们进行如下判断：

- 如果 j 在 O 中：
    - 则 $\mathrm{OPT}(j) = \mathrm{OPT}(p_j) + v(j)$，即最优解包含区间 j 的值 $v(j)$ 加上不与之重叠的最大前缀区间的最优解 $\mathrm{OPT}(p_j)$
- 如果 j 不在 O 中：

    - 则 $\mathrm{OPT}(j) = \mathrm{OPT}(j-1)$，即最优解不包含区间 j，直接等于前 $j-1$ 个区间的最优解

最终，$\mathrm{OPT}(j)$ 的计算公式为

$$\mathrm{OPT}(j) = \max\left(\mathrm{OPT}(p_j) + v(j), \mathrm{OPT}(j-1)\right) \tag{1}$$

换句话说，区间 j 在最优解 O 中，当且仅当

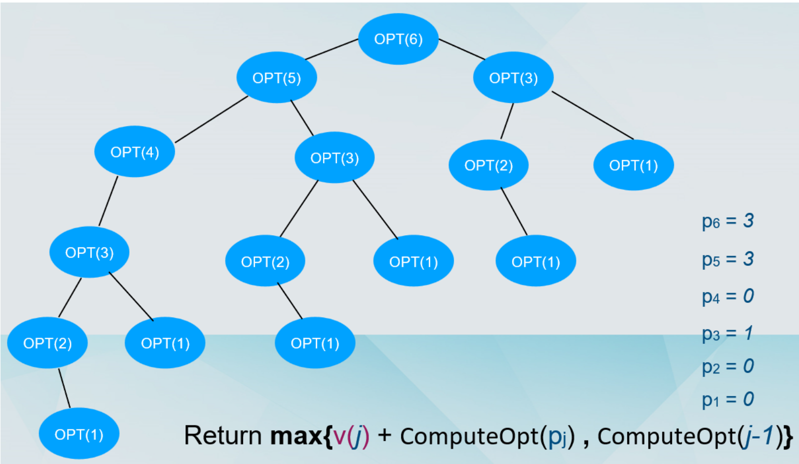
$$\mathrm{OPT}(p_j) + v(j) \geq \mathrm{OPT}(j-1)$$

递推关系

而对于 1 式，这就像：一个算法输入 $\{1, \ldots, j\}$，输出 $\mathrm{OPT}(j)$，这是一个递推关系

```
ComputeOpt(j)
        If j = 0 then
            Return 0
        Else
            Return max(v(j) + ComputeOpt(p_j), ComputeOpt(j-1))
        EndIf
```

举例来说，对于 p 列 $\{0,0,1,0,3,3\}$（升序）



运行时间

朴素算法

- 解决大小为 j 的问题需要解决大小为 $j-1$ 和 $j-2$ 的问题（其实不是）
    - 这是一个斐波那契数列！
- 第 n 个斐波那契数大约是 $\phi^n\sqrt{5}$（其中 ϕ 是黄金分割比，约等于 1.618）
- 因此，我们的算法运行时间是 $\Omega(2^n)$

其中，每个子问题不直接等同于 $j-1$ 和 $j-2$，但是递归树的展开方式和斐波那契数列的递归树相似。每个递归调用会导致两个进一步的递归调用，这在时间复杂度上其实没有区别，而递归调用的增长方式使得时间复杂度达到指数级

我们并非每次都要计算重复出现的项目（空间换时间）

- 对于每个 j，只计算一次 `ComputeOpt(j)`
- 将计算结果存储在一个可以访问的地方，以便将来再次使用

- 维护一个数组 $M[0,...,n]$
  - 初始时，所有 $M[j]$ 都是“空”的
- 当 `ComputeOpt(j)` 被计算时，$M[j]$ = `ComputeOpt(j)`

```
M-ComputeOpt(j)
        If j = 0 then
            Return 0
        Else if M[j] is not empty then
            Return M[j]
        Else
            M[j] = max{v(j) + M-ComputeOpt(p_j), M-ComputeOpt(j-1)}
            Return M[j]
        EndIf
```

这个算法能找到最优调度的值，但是我们需要的是最优调度序列

算法的运行时间:
显然，这个算法的运行时间取决于递归调用的次数
由于每个 j 只会进行一次递归调用，且总共有 n 个区间，因此总的递归调用次数为 $O(n)$

假设输入的区间已经按照结束时间排序，那么 `M-ComputeOpt` 的运行时间是 $O(n)$；而排序时间为 $O(n \log n)$

从计算出的最优值中追溯回具体的调度方案

在之前的内容中，我们有

$$\mathrm{OPT}(p_j) + v(j) \geq \mathrm{OPT}(j-1)$$

因此我们可以从计算出的**最优值**中追溯回具体的调度方案

```
        FindSolution(j)
        If j = 0
                Output "no solution"
        Else
                If v(j) + M(p_j) ≥ M(j-1) then
                    Output j together with FindSolution(p_j)
                Else
                    Output FindSolution(j-1)
                EndIf
        End If
```

其中 `M()` 是之前的 `M-ComputeOpt()` 中计算出的
这是一个 $O(n)$ 算法

动态规划 vs. 分治法

动态规划（DP）

1. **适用问题**:
   - 适用于具有最优子结构和重叠子问题的问题
   - 典型的应用包括斐波那契数列、最长公共子序列、背包问题等

2. 问题分解：
    - 将问题分解成若干规模较小但重叠的子问题
    - 通过解决这些子问题并合并其结果来构建原问题的解
3. 子问题依赖关系：
    - 子问题之间的依赖关系通常可以表示为有向无环图（DAG）
    - 通过记忆化（Memoization）或自底向上的方法（如填表法）来避免重复计算

分治法（DQ）

1. 适用问题：
    - 适用于可以分解成互不重叠的子问题的问题
    - 典型的应用包括归并排序、快速排序、二分搜索等
2. 问题分解：
    - 将问题递归地分解成若干规模显著减小且互不重叠的子问题
    - 通过解决这些子问题并合并其结果来构建原问题的解
3. 子问题依赖关系：
    - 子问题之间的依赖关系通常可以表示为树结构
    - 由于子问题互不重叠，因此没有重复计算的问题