Dilworth's theorem

Updated 2151 GMT+8 Nov 3 2024

2020 fall, Complied by Hongfei Yan

Dilworth 定理，最小的链覆盖数等于最长反链长度。https://oi-wiki.org/math/order-theory/

> Dilworth 定理是组合数学中的一个重要定理，主要应用于部分有序集（Partially Ordered Set, poset）的研究。该定理描述了部分有序集中最小链覆盖与最大反链的关系。具体来说，Dilworth 定理指出：
>
> **Dilworth** 定理
>
> 在一个部分有序集 $(P, \leq) $中，最小链覆盖的大小等于最大反链的大小。
>
> **定义**
>
> 1. **链（Chain）：**
>   - 一个链是指部分有序集中的一个子集，其中任意两个元素都是可比较的。换句话说，对于链中的任意两个元素 a 和 b，要么 $a \leq b$，要么 $ b \leq a $。
> 2. **反链（Antichain）：**
>   - 一个反链是指部分有序集中的一个子集，其中任意两个不同的元素都是不可比较的。换句话说，对于反链中的任意两个不同的元素 a 和 b，既不满足 $a \leq b$，也不满足 $ b \leq a $。
> 3. **链覆盖（Chain Cover）：**
>   - 一个链覆盖是指一组链，它们的并集覆盖了整个部分有序集 P。换句话说，每个元素都至少属于一个链。
> 4. **最小链覆盖：**
>   - 最小链覆盖是指覆盖整个部分有序集所需的最少链的数量。
> 5. **最大反链：**
>   - 最大反链是指部分有序集中最大的反链的大小，即包含最多元素的反链。
>
> **定理陈述**
>
> Dilworth 定理可以形式化为：
>
> $ \text{最小链覆盖的大小} = \text{最大反链的大小} $
>
> **应用**
>
> Dilworth 定理在许多领域都有广泛的应用，特别是在算法设计和组合优化中。以下是一些常见的应用场景：

6. **最长单调子序列问题：**
    - 在寻找最长单调递增子序列（LIS）时，可以将问题转化为部分有序集中的链覆盖问题。具体来说，将数组中的元素视为部分有序集中的元素，定义 $ a_i \leq a_j $ 当且仅当 $ i < j $ 且 $ a_i \leq a_j $。Dilworth 定理告诉我们，最长递增子序列的长度等于最小链覆盖的大小，而最小链覆盖的大小又等于最大反链的大小。
7. **任务调度问题：**
    - 在任务调度问题中，可以将任务视为部分有序集中的元素，定义任务之间的依赖关系。Dilworth 定理可以帮助我们找到最小的并行任务集，使得所有任务都能在最短时间内完成。

总结

Dilworth 定理是一个强大的工具，它将部分有序集中的最小链覆盖问题与最大反链问题联系起来。通过理解和应用 Dilworth 定理，可以在许多组合优化问题中找到高效的解决方案。

最长不增子序列：Longest Non-Increasing Subsequence

最长单调递减子序列：Longest Decreasing Subsequence

最长不降子序列：Longest Non-Decreasing Subsequence

最长单调递增子序列：Longest Increasing Subsequence

编程题目

28389: 跳高

http://cs101.openjudge.cn/practice/28389

Dilworth 定理表明，任何一个有限偏序集的最长反链（即最长下降子序列）的长度，等于将该偏序集划分为尽量少的链（即上升子序列）的最小数量。

因此，计算序列的最长下降子序列长度，即可得出最少需要多少台测试仪。

```
"""
Dilworth 定理:
Dilworth 定理表明，任何一个有限偏序集的最长反链(即最长下降子序列)的长度，
等于将该偏序集划分为尽量少的链(即上升子序列)的最小数量。
因此，计算序列的最长下降子序列长度，即可得出最少需要多少台测试仪。
"""

from bisect import bisect_left

def min_testers_needed(scores):
    scores.reverse()  # 反转序列以找到最长下降子序列的长度
    lis = []  # 用于存储最长上升子序列

    for score in scores:
        pos = bisect_left(lis, score)
```

```
        if pos < len(lis):
            lis[pos] = score
        else:
            lis.append(score)

    return len(lis)


N = int(input())
scores = list(map(int, input().split()))

result = min_testers_needed(scores)
print(result)
```

01065: Wooden Sticks

http://cs101.openjudge.cn/practice/01065/

也是 Dilworth's theorem。容易发现，所需时间为上升子序列的个数，根据 Dilworth 定理，最少的 chain 个数等于最大的 antichain 的大小，即最少上升子序列的个数等于最长递减子序列的长度。最长严格递减子序列有经典的 nlogn 的算法（https://en.wikipedia.org/wiki/Longest_increasing_subsequence）。

```
#dilworth 和最长单调子序列
# 答案就是对 l 排序后求 w 的最长严格递减子序列（用 Dilworth's theorem 不难证明），
# 最长严格递减子序列有经典的 nlogn 的算法
# （https://en.wikipedia.org/wiki/Longest_increasing_subsequence）。
#一般有一样的都不是大问题，因为可以把(3,5) (3,6) 直接看作(3.1, 5) (3.2, 6)

import bisect

def doit():
    n = int(input())
    data = list(map(int, input().split()))
    sticks = [(data[i], data[i + 1]) for i in range(0, 2 * n, 2)]
    sticks.sort()
    f = [sticks[i][1] for i in range(n)]
    f.reverse()
    stk = []

    for i in range(n):
        t = bisect.bisect_left(stk, f[i])
        if t == len(stk):
            stk.append(f[i])
        else:
            stk[t] = f[i]

    print(len(stk))

T = int(input())
for _ in range(T):
    doit()
```

02945: 拦截导弹

dp, http://cs101.openjudge.cn/practice/02945/

bisect_right 二分查找可以高效地求出最长不降子序列的长度

```
"""
与这个题目思路相同：
28389: 跳高，http://cs101.openjudge.cn/practice/28389

拦截导弹 求最长不升 LNIS，可以相等所以用 bisect right。如果求最长上升 LIS，用
bisect_left
"""

from bisect import bisect_right

def min_testers_needed(scores):
    scores.reverse()  # 反转序列以找到最长下降子序列的长度
    lnis = []  # 用于存储最长上升子序列

    for score in scores:
        pos = bisect_right(lis, score)
        if pos < len(lis):
            lnis[pos] = score
        else:
            lnis.append(score)

    return len(lis)


N = int(input())
scores = list(map(int, input().split()))

result = min_testers_needed(scores)
print(result)
```

P1020 [NOIP1999 提高组] 导弹拦截

https://www.luogu.com.cn/problem/P1020

某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹。

输入导弹依次飞来的高度，计算这套系统最多能拦截多少导弹，如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。

输入格式

一行，若干个整数，中间由空格隔开。

输出格式

两行，每行一个整数，第一个数字表示这套系统最多能拦截多少导弹，第二个数字表示如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。

样例输入 #1

```
389 207 155 300 299 170 158 65
```

样例输出 #1

```
6
2
```

提示

对于前 50% 数据（NOIP 原题数据），满足导弹的个数不超过 10^4 个。该部分数据总分共 100 分。可使用$\mathcal O(n^2)$ 做法通过。
对于后 50% 的数据，满足导弹的个数不超过 10^5 个。该部分数据总分也为 100 分。请使用 $\mathcal O(n\log n)$ 做法通过。

对于全部数据，满足导弹的高度为正整数，且不超过 5×10^4。

此外本题开启 spj，每点两问，按问给分。

NOIP1999 提高组 第一题

---

upd 2022.8.24：新增加一组 Hack 数据。

第一问，求最长链，偏序取反，bisect_right。

第二问，Dilworth 定理，偏序取反，bisect_left。

```
from bisect import bisect_left

def min_testers_needed(scores):
    scores.reverse()  # 反转序列以找到最长下降子序列的长度
    lis = []  # 用于存储最长上升子序列

    for score in scores:
        pos = bisect_left(lis, score)
        if pos < len(lis):
            lis[pos] = score
        else:
            lis.append(score)

    return len(lis)


N = int(input())
scores = list(map(int, input().split()))

result = min_testers_needed(scores)
print(result)
```

求最长单调子序列

Dilworth 定理在组合数学中非常重要，尤其是在处理部分有序集（poset）的问题时。Dilworth 定理的一个经典应用是寻找最长单调子序列（LIS）。虽然二分查找可以高效地求出最长单调子序列的长度，但它并不直接提供具体的子序列。如果你还需要知道具体的子序列，可以使用动态规划（DP）方法，并在 DP 过程中记录路径信息。

Dilworth 定理与最长单调子序列

Dilworth 定理指出，在一个部分有序集中，最小链覆盖的大小等于最大反链的大小。在最长单调子序列问题中，链对应于单调递增子序列，反链对应于单调递减子序列。

二分查找与最长单调子序列

二分查找可以高效地求出最长单调子序列的长度，但不直接提供具体的子序列。具体步骤如下：

1. **初始化：**
    - 创建一个数组 `dp`，用于存储当前最长递增子序列的末尾元素。
    - 创建一个数组 `prev`，用于记录每个元素在最长递增子序列中的前驱元素。
2. **遍历数组：**
    - 对于每个元素 `nums[i]`，使用二分查找在 `dp` 中找到合适的位置 `pos`，使得 `dp[pos-1] < nums[i] <= dp[pos]`。
    - 更新 `dp[pos]` 为 `nums[i]`。
    - 更新 `prev[i]` 为 `dp[pos-1]` 的索引。
3. **构造最长单调子序列：**
    - 从 `dp` 数组的最后一个元素开始，通过 `prev` 数组回溯，构造最长单调子序列。

示例代码

以下是一个 Python 代码示例，展示了如何使用二分查找和动态规划来求出最长单调子序列及其具体内容：

```
# 用动态规划结合二分查找来找到最长单调递增子序列（LIS）的长度和具体序列
"""
维护一个数组，用于记录当前找到的最长子序列的末尾元素，并且在此基础上再维护一个额外的数组或列表，
用来记录每个元素在最长单调子序列中的前驱信息。这样，当你完成处理之后，可以通过前驱信息追踪回去，
得到具体的子序列。

"""
from bisect import bisect_left
```

```
def longest_increasing_subsequence(nums):
    if not nums:
        return []

    # dp 数组，用于存储当前找到的最长子序列的末尾元素
    dp = []
    # 存储每个元素的前驱索引
    prev_indices = [-1] * len(nums)
    # 存储每个位置的索引
    indices = []

    for i, num in enumerate(nums):
        # 使用二分查找找到插入位置
        pos = bisect_left(dp, num)

        # 如果 pos 等于 dp 的长度，说明 num 比 dp 中的所有元素都大
        if pos == len(dp):
            dp.append(num)
            if pos > 0:
                prev_indices[i] = indices[-1]  # 更新前驱索引
            indices.append(i)
        else:
            dp[pos] = num
            indices[pos] = i  # 更新当前 pos 位置的索引
            if pos > 0:
                prev_indices[i] = indices[pos - 1]  # 更新前驱索引

    # 重建最长递增子序列
    lis = []
    k = indices[-1]
    while k >= 0:
        lis.append(nums[k])
        k = prev_indices[k]

    # 反转序列，得到从小到大的顺序
    lis.reverse()

    return lis


# 示例
#nums = [10, 9, 2, 5, 3, 7, 101, 18]
nums = [1, 7, 3, 5, 2]
result = longest_increasing_subsequence(nums)
print("最长单调递增子序列:", result)
```

解释

4. **初始化**：
    - `dp` 用于存储当前最长递增子序列的末尾元素。
    - `prev` 用于记录每个元素在最长递增子序列中的前驱元素。

5. **遍历数组：**
    - 对于每个元素 `nums[i]`，使用 `bisect_left` 在 `dp` 中找到合适的位置 `pos`。
    - 更新 `dp[pos]` 为 `nums[i]`。
    - 更新 `prev[i]` 为 `dp[pos-1]` 的索引。
6. **构造最长单调子序列：**
    - 从 `dp` 数组的最后一个元素开始，通过 `prev` 数组回溯，构造最长单调子序列。
    - 最后将结果反转，得到从前往后的顺序。

总结

虽然二分查找可以高效地求出最长单调子序列的长度，但为了得到具体的子序列，需要在 DP 过程中记录路径信息。通过这种方式，可以在保持高效的同时，获得完整的最长单调子序列。