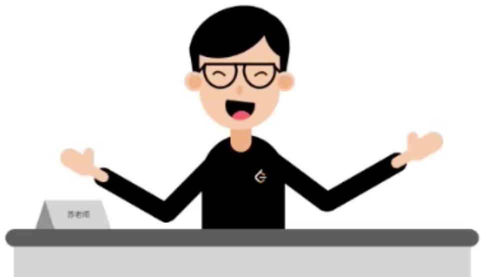
动态规划

- 基本属性
- 题目分类
- 解题思想
- 巩固与加深：算法复杂度

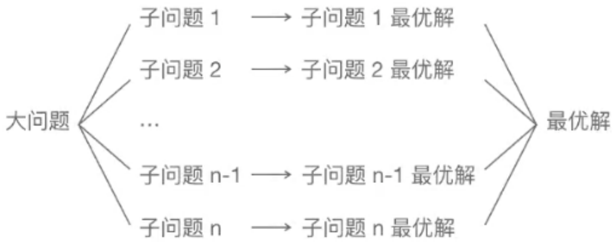


动态规划的定义

- 一种数学优化的方法，同时也是编程的方法。

重要属性

- 最优子结构 Optimal Substructure

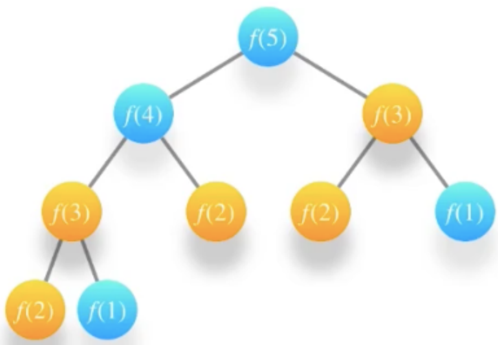


动态规划的定义

- 一种数学优化的方法，同时也是编程的方法。

重要属性

- 最优子结构 Optimal Substructure
  - 状态转移方程 $f(n)$
- 重叠子问题 Overlapping Sub-problems



300. 最长子序列的长度

给定一个无序的整数数组，找到其中最长子序列长度。

说明：
可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
你算法的时间复杂度应该为 $O(n^2)$。

注意：
子序列和子数组不同，它并不要求元素是连续的。

示例：
输入：[10, 9, 2, 5, 3, 7, 101, 4]
输出：4

解释：
最长的上升子序列是[2, 3, 7, 101]
它的长度是4

300. 最长子序列的长度

[10, 9, 2, 5, 3, 7, 101, 4]

非空子数组：$\frac{n(n+1)}{2}$ 复杂度 $O(n^2)$

非空子序列：$2^n - 1$ 复杂度 $O(2^n)$

将问题规模减少，推导出状态转移方程式

$f(n)$ 表示数组 nums[0, 1, 2, …, n-1] 中最长的子序列

$f(n-1)$ 表示数组 nums[0, 1, 2, … n-2] 中最长的子序列

$f(1)$ 表示数组 nums[0] 的最长子序列

nums[n-2] **nums[n-1]**

解决动态规划问题最难的两个地方：

- 如何定义 $f(n)$

对于这道题而言，$f(n)$ 是以 nums[n-1] 结尾的最长的上升子序列的长度

- 如何通过 $f(1), f(2), \ldots, f(n-1)$ 推导出 $f(n)$，即状态转移方程

- 拿nums[n-1]与比它小的每一个值 nums[i] 作比较，其中 $1 \leq i < n$，然后加 1 即可
- 因此状态转移方程为：

$$f(n) = max\{f(i)\} + 1 \quad (1 \leq i < n - 1，并且\ nums[i-1] < nums[n-1])$$

```java
class LISRecursion {
  static int max;

  public int f(int[] nums, int n) {
    if (n <= 1) {
      return n;
    }

    int result = 0, maxEndingHere = 1;

    for (int i = 1; i < n; i++) {
      result = f(nums, i);

      if (nums[i - 1] < nums[n - 1] && result + 1 > maxEndingHere) {
        maxEndingHere = result + 1;
      }
    }

    if (max < maxEndingHere) {
      max = maxEndingHere;
    }

    return maxEndingHere;
  }

  public int LIS(int[] nums) {
    max = 1;
    f(nums, nums.length);
    return max;
  }
}
```

- 首先最基本的情况是：
  当数组的长度为 0 时，没有上升子序列，
  当数组长度为 1 时，最长的上升子序列长度是 1。
- maxEndingHere 变量的含义是：
  包含当前最后一个元素的情况下，最长的上升子序列长度。
- 从头遍历数组，
  递归求出以每个点为结尾的子数组中最长上升序列的长度。
- 判断一下，如果该数比目前最后一个数要小，
  就能构成一个新的上升子序列。
  这个新的子序列有可能成为最终的答案。
- 最后返回以当前数结尾的上升子序列的最长长度。

```java
class LISRecursion {
  static int max;

  public int f(int[] nums, int n) {
    if (n <= 1) {
      return n;
    }

    int result = 0, maxEndingHere = 1;

    for (int i = 1; i < n; i++) {
      result = f(nums, i);

      if (nums[i - 1] < nums[n - 1] && result + 1 > maxEndingHere) {
        maxEndingHere = result + 1;
      }
    }

    if (max < maxEndingHere) {
      max = maxEndingHere;
    }

    return maxEndingHere;
  }

  public int LIS(int[] nums) {
    max = 1;
    f(nums, nums.length);
    return max;
  }
}
```

时间复杂度分析

- $T(n-1) = T(1) + T(2) + ... + T(n-2)$
- $T(n) = T(1) + T(2) + ... + T(n-1)$

$$T(n) = 2 \times T(n-1) \quad \neq \quad T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

$$O(n) = O(2^n)$$

```java
class LISMemoization {
  static int max;
  static HashMap<Integer, Integer> cache;

  public int f(int[] nums, int n) {
    if (cache.containsKey(n)) {
      return cache.get(n);
    }

    if (n <= 1) {
      return n;
    }

    int result = 0, maxEndingHere = 1;

    for (int i = 1; i < n; i++) {
      ...
    }

    if (max < maxEndingHere) {
      max = maxEndingHere;
    }

    cache.put(n, maxEndingHere);
    return maxEndingHere;
  }
}
```

- 首先，定义一个哈希表 cache，用来保存我们的计算结果。
- 在每次调用递归函数的时候，
  判断一下对于这个输入，我们是否已经计算过了，
  也就是 cache 里是否已经保留了这个值，
  是的话，立即返回，如果不是，再继续递归调用。
- 在返回当前结果前，保存到 cache 里，
  这样下次遇到了同样的输入时，就不用再浪费时间计算了。

动态规划解题难点

- 应当采用什么样的数据结构来保存什么样的计算结果
- 如何利用保存下来的计算结果推导出状态转移方程