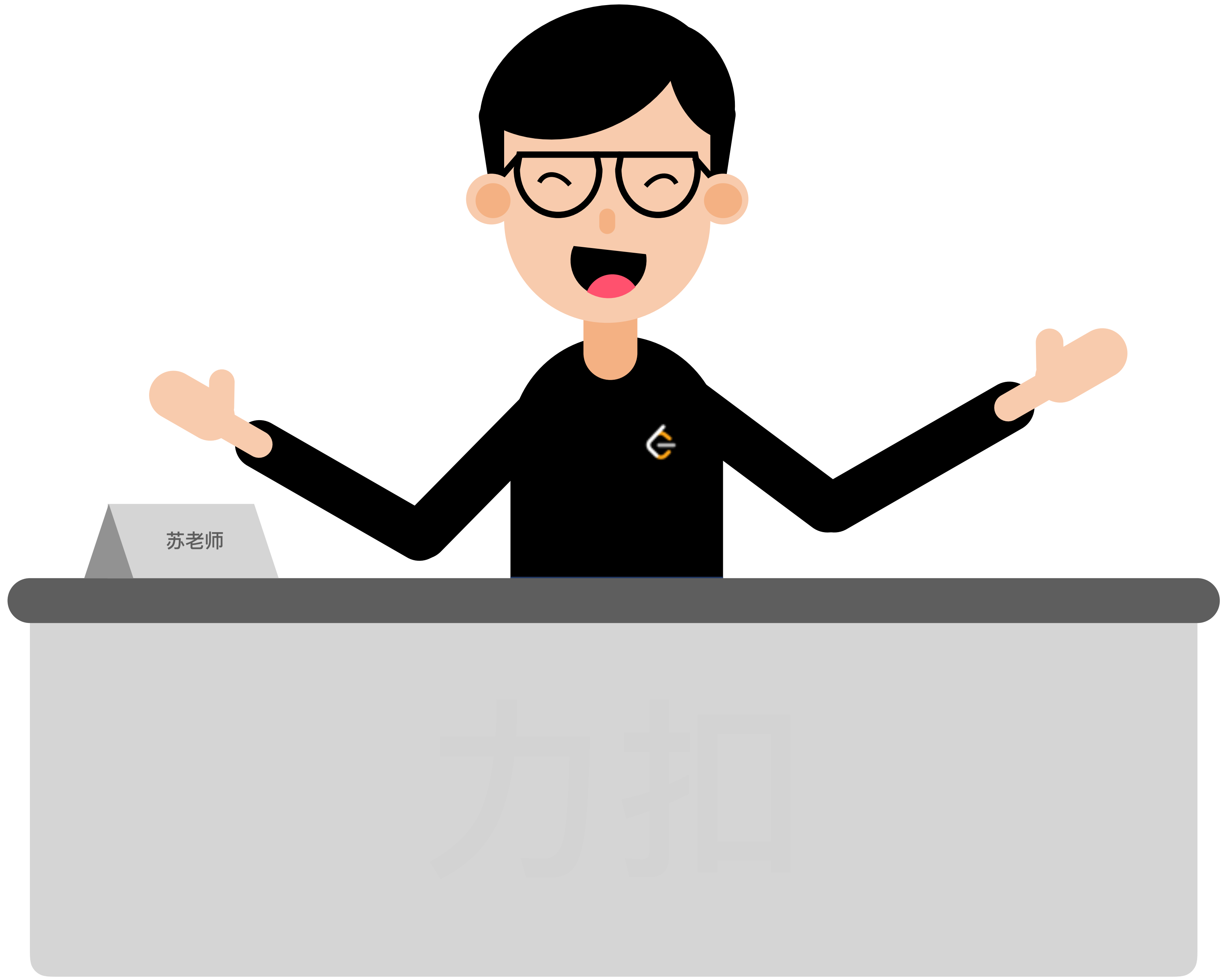
第六课

强化面试中常用的算法 - 动态规划

6.0 简介 / Introduction

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



6.1 动态规划 / Dynamic Programming

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

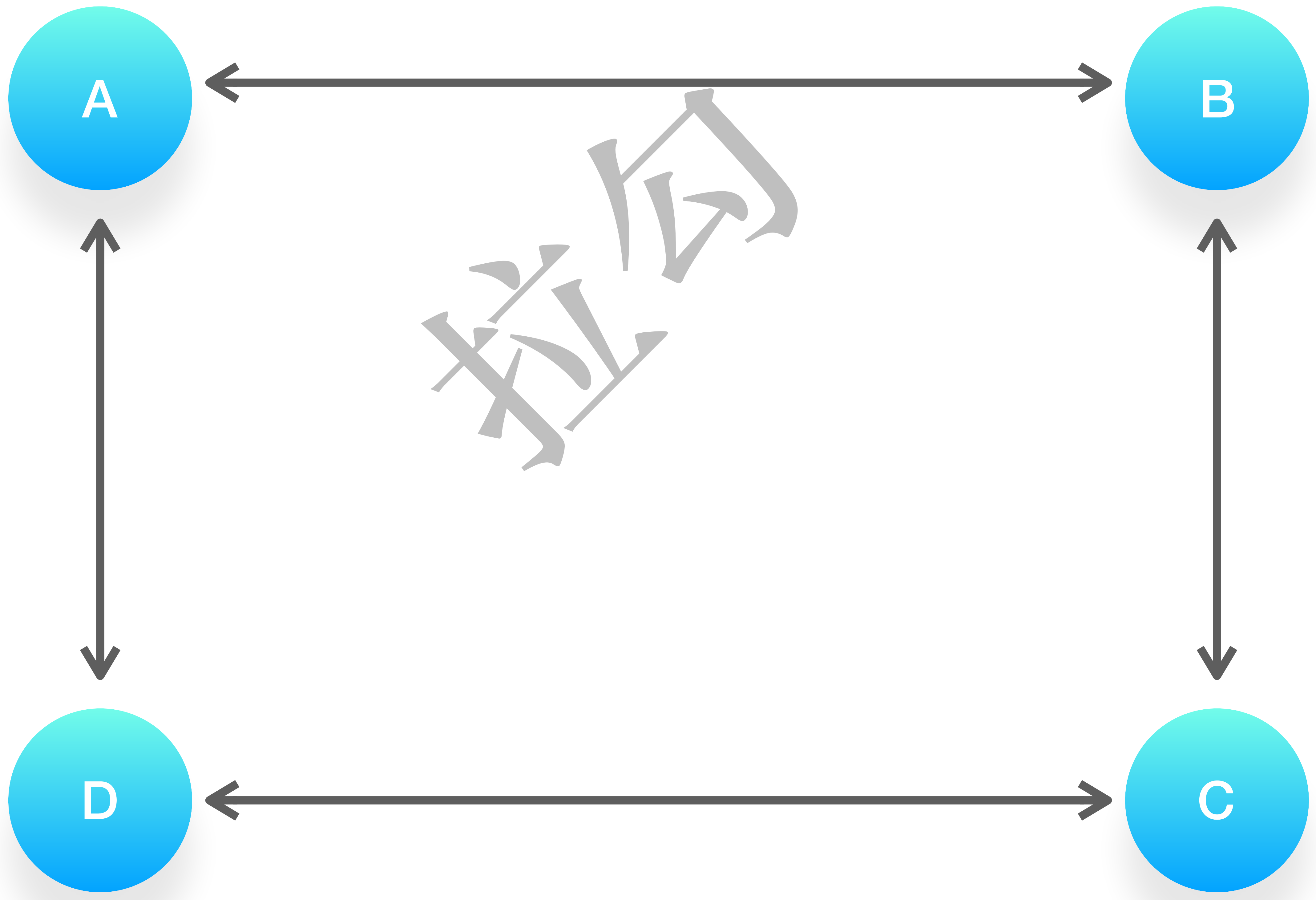
- 最优子结构 Optimal Substructure
  - 状态转移方程 $f(n)$

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。

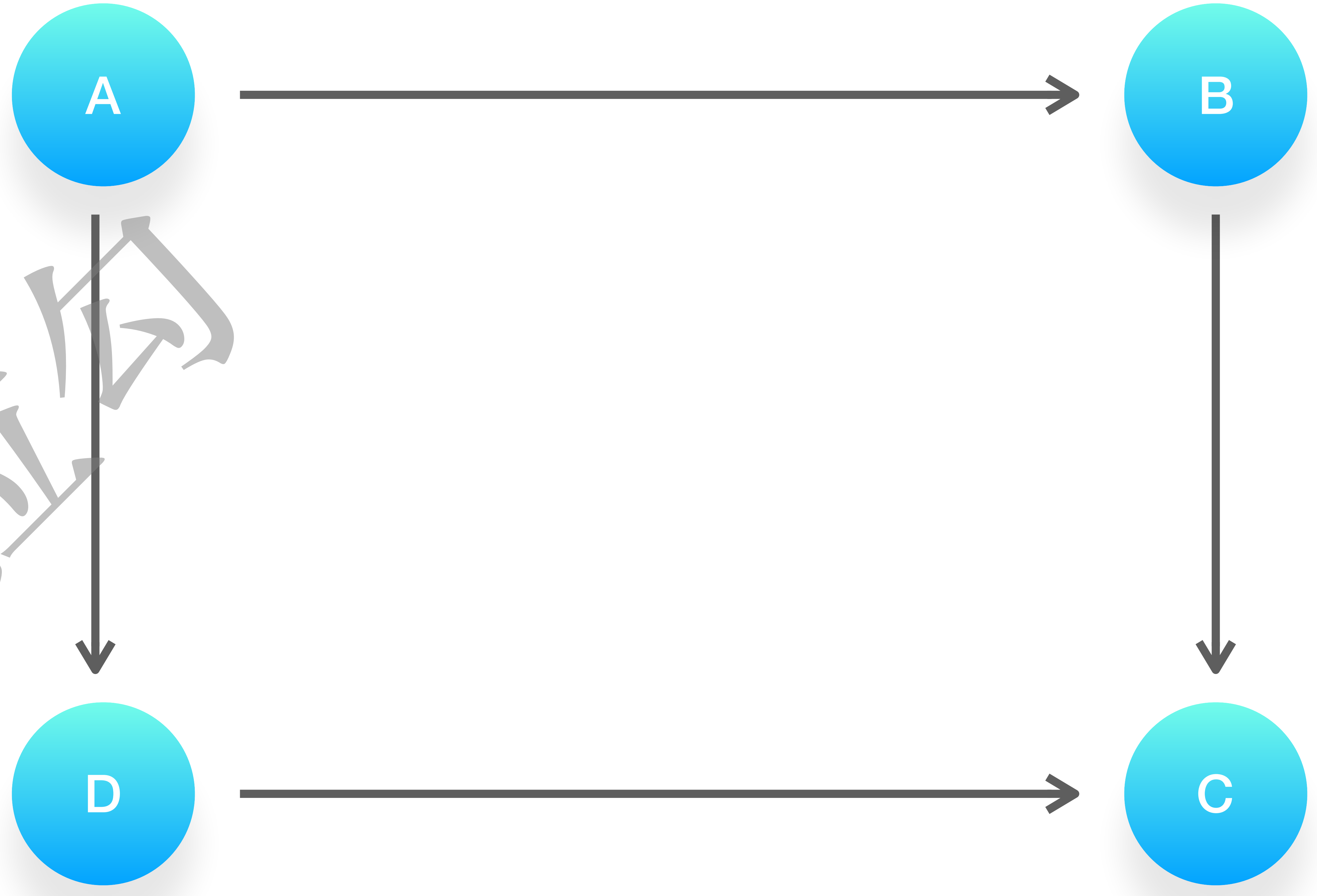
重要属性

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


A
B
C
D

A 到 C 的最长距离

- A -> B -> C
- A -> D -> C



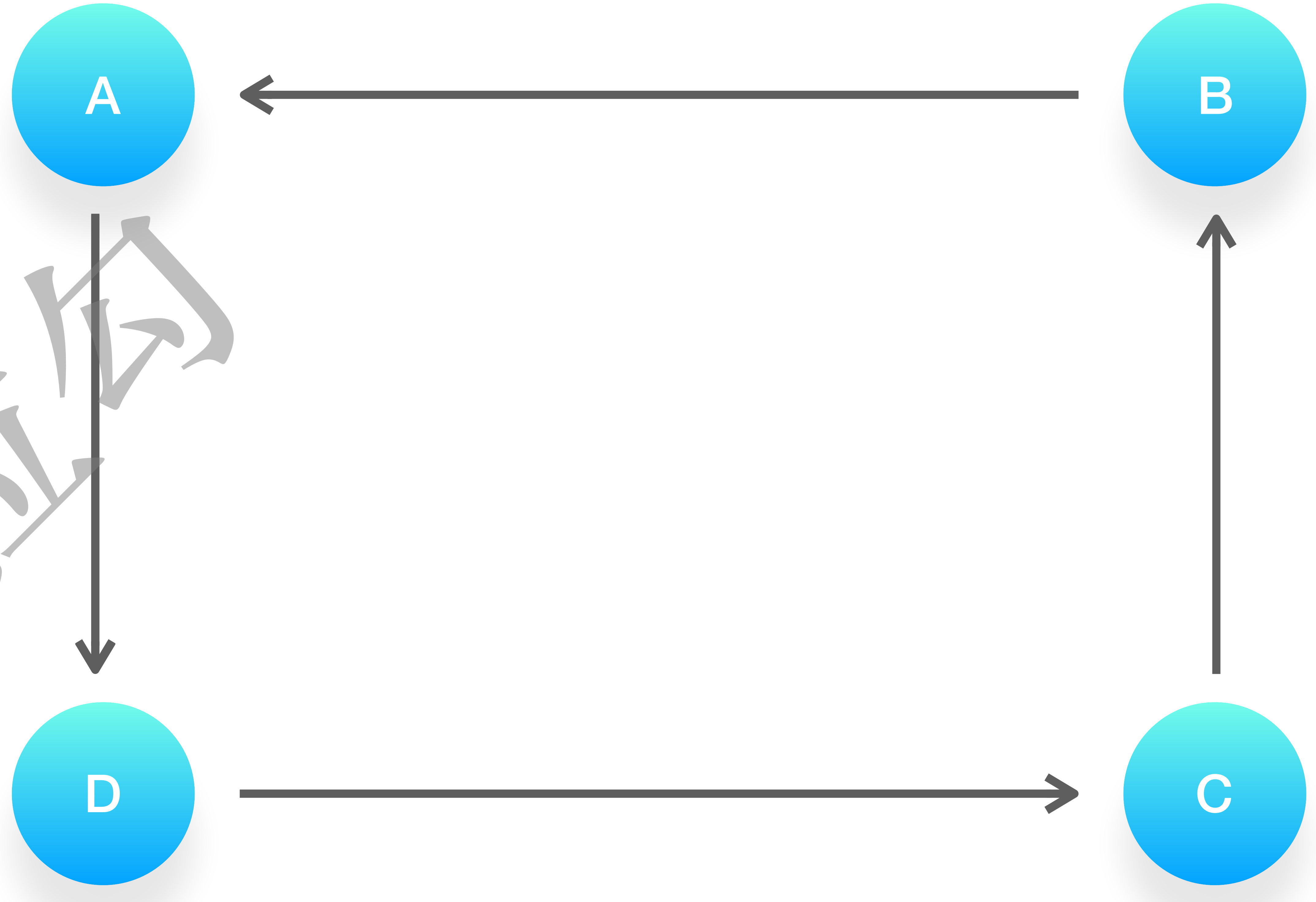
- A -> B -> C

A 到 B 的最长距离

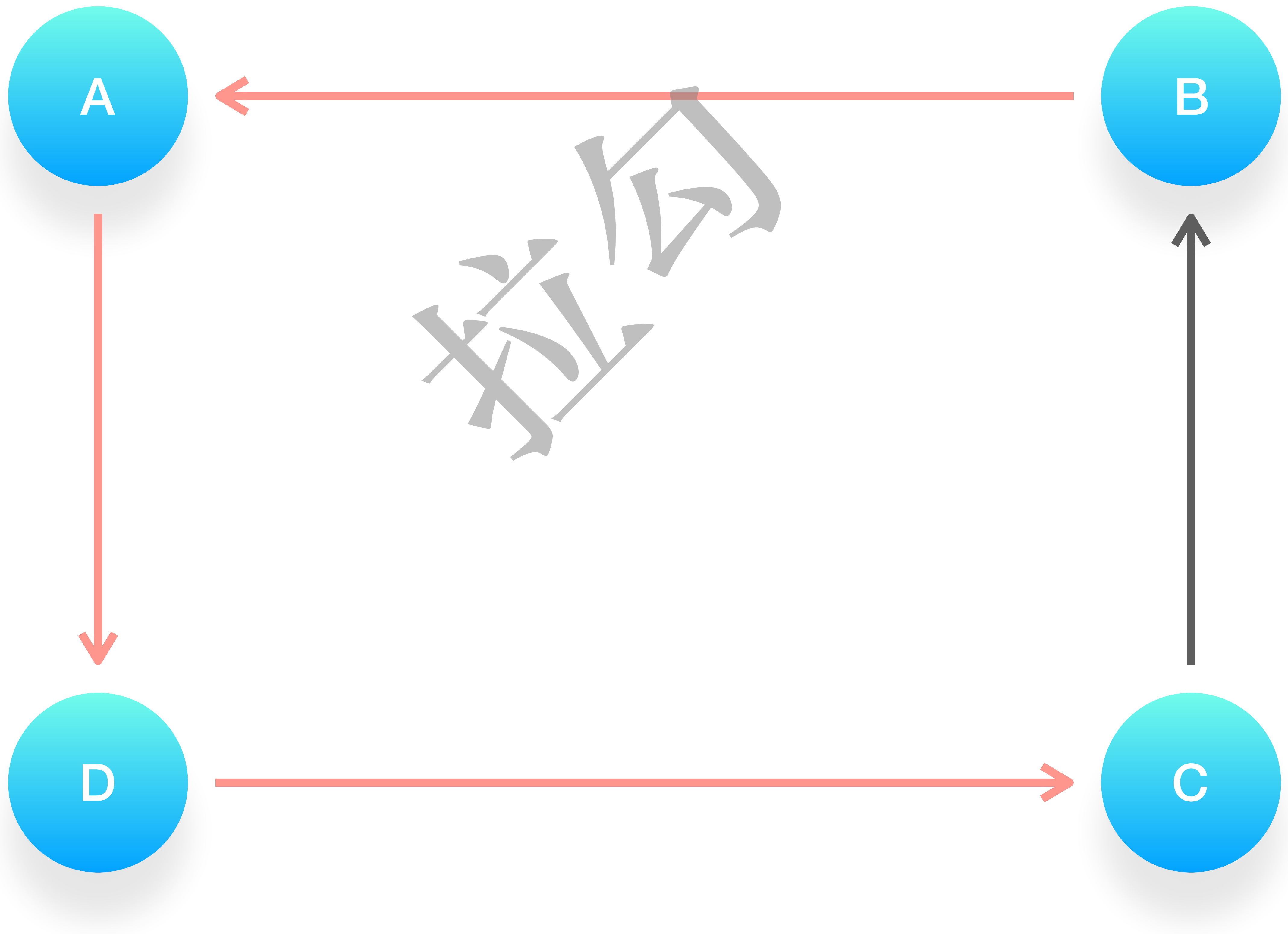
- A -> D -> C -> B

B 到 C 的最长距离

- B -> A -> D -> C



- A -> D -> C -> B -> A -> D -> C



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, 18]
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, 18]

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 $O(2^n)$

300. 最长子序列的长度

[10, 9, 2, 5]　　[2, 5]

[10, 9, 2, 5, 3, 7, 101, 18]

[3, 7, 101, 18]　　[3, 7, 101]

300. 最长子序列的长度

[10, 9, 2, 5]

[10, 9, 2, 5, 3, 7, 101, 18]

[2, 5] [3, 7, 101]

[3, 7, 101, 18]

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

$f(1)$ $f(2)$ **nums[n-1]** $f(n-1)$

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

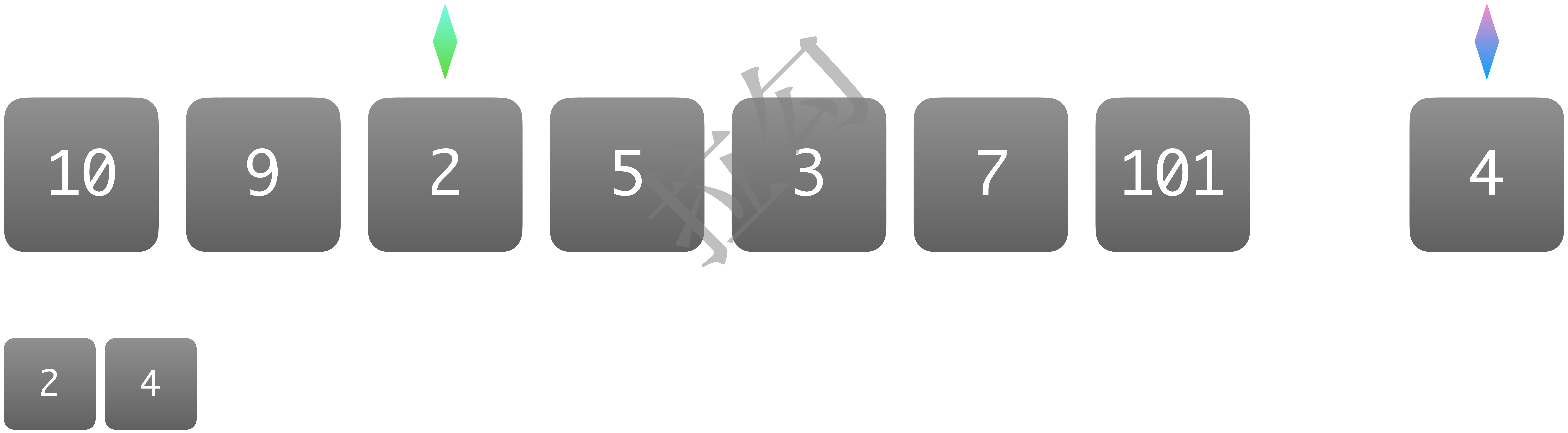
$f(n-1)$ 表示数组 nums[0, 1, 2, … n-2] 中最长的子序列

$f(1)$ 表示数组 nums[0] 的最长子序列

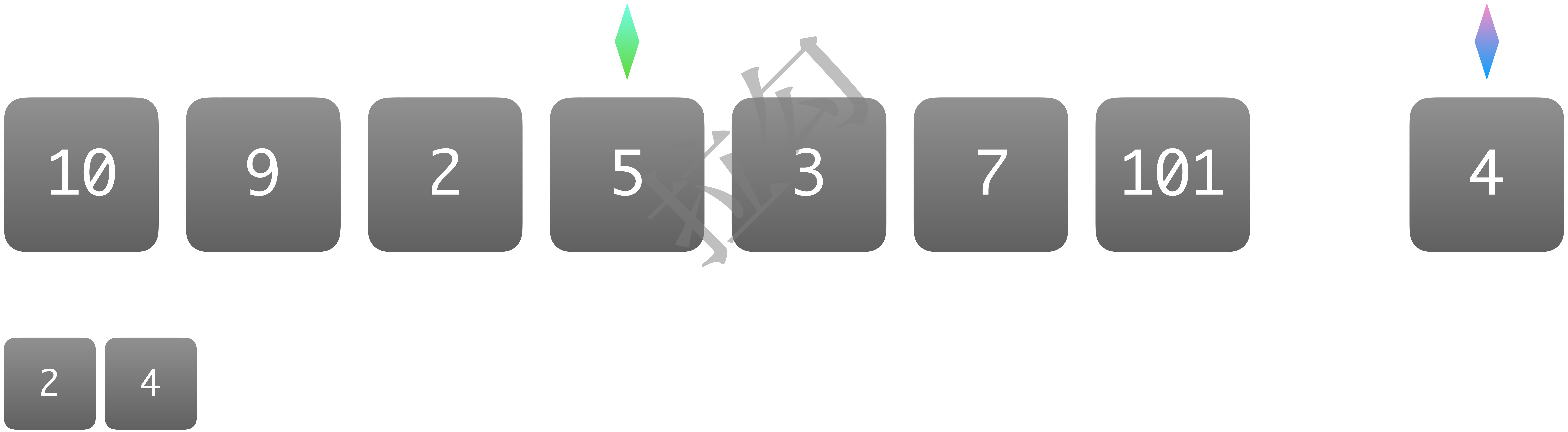
nums[n-2] **nums[n-1]**

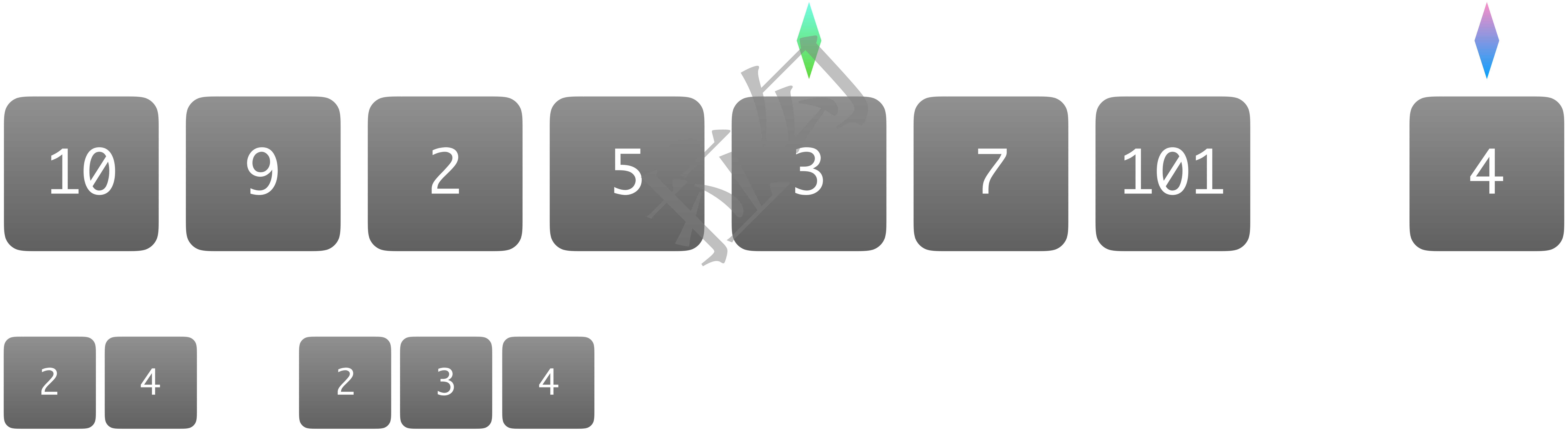

10
9
2
5
3
7
101
4

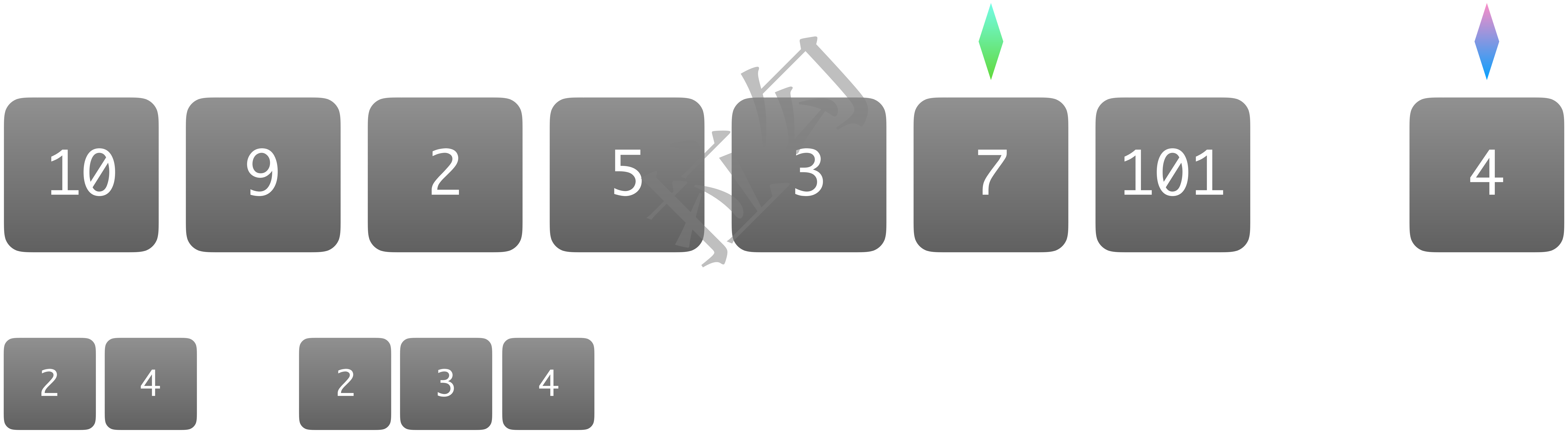

10
9
2
5
3
7
101
4


10
9
2
5
3
7
101
4
2
4

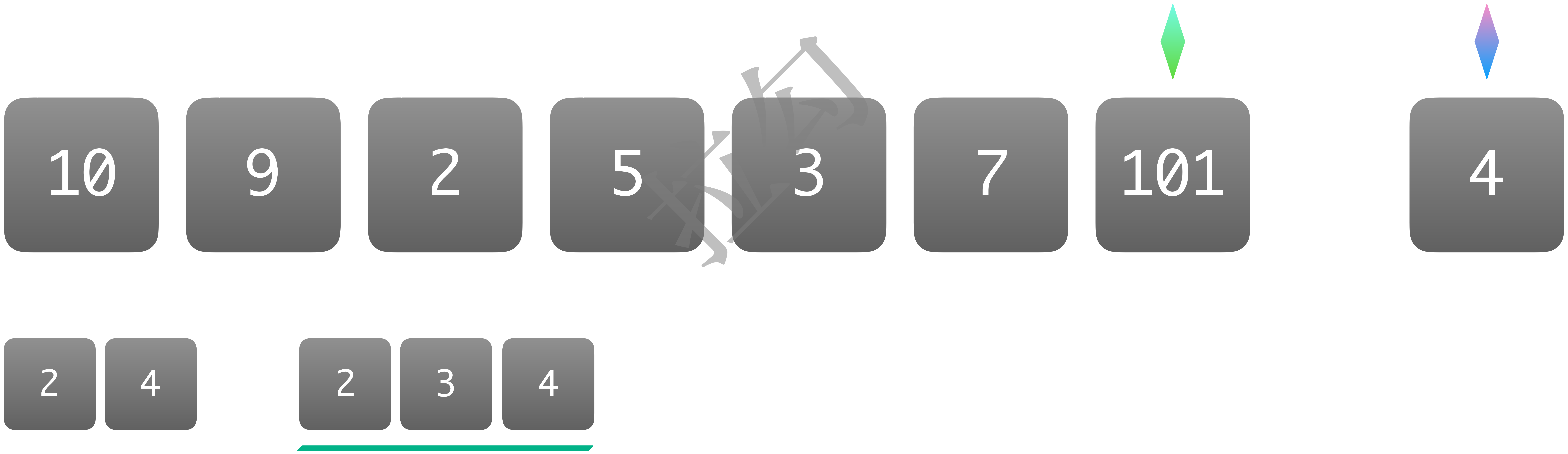
6.1 动态规划 / Dynamic Programming

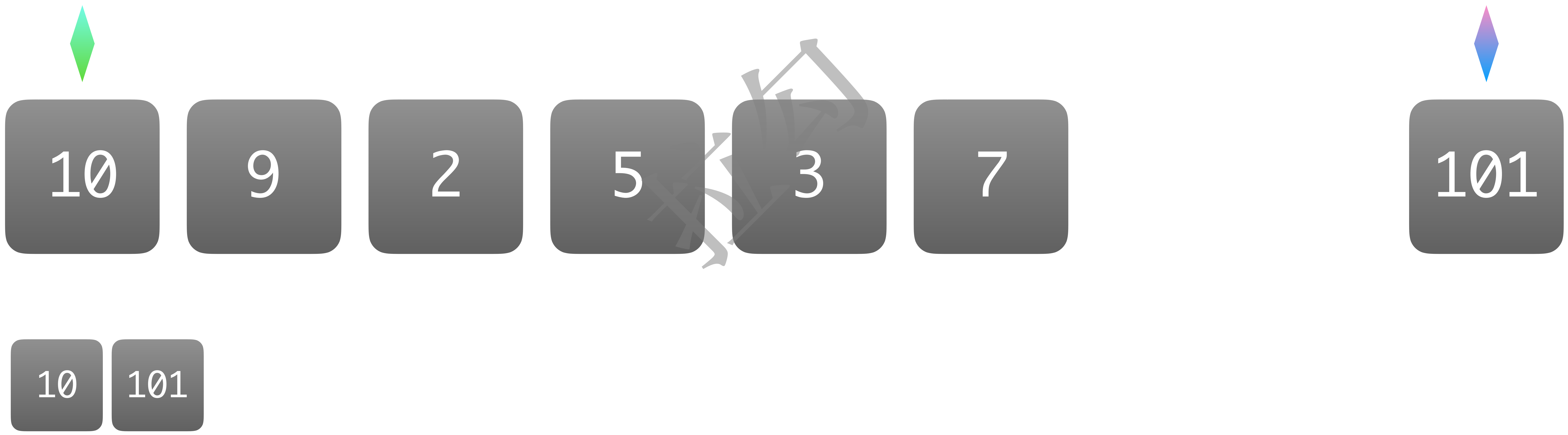


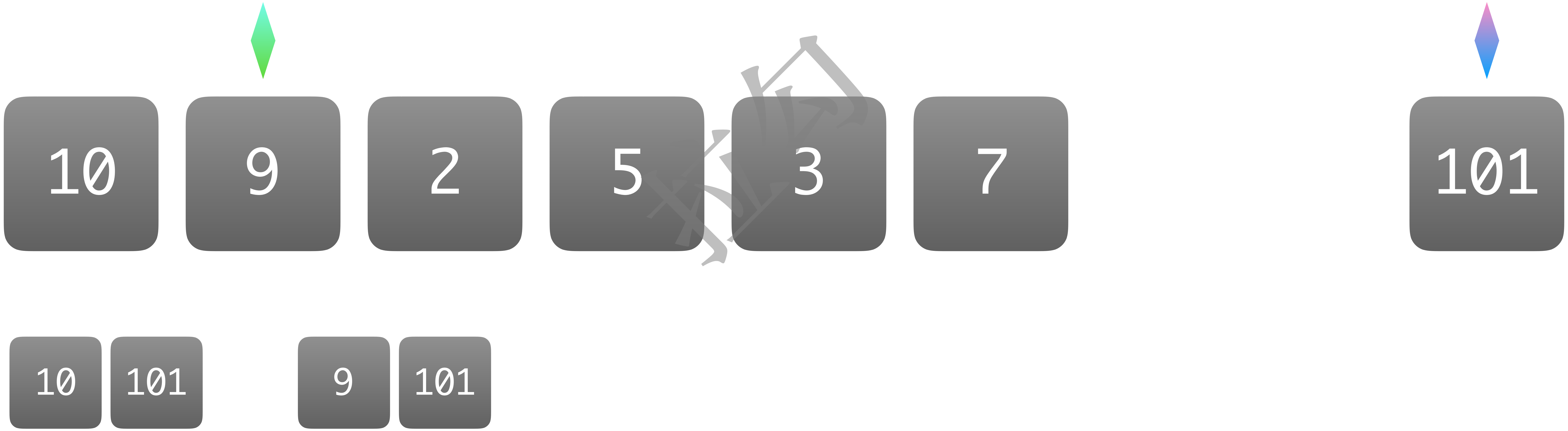

10
9
2
5
3
7
101
4
2
4
2
3
4

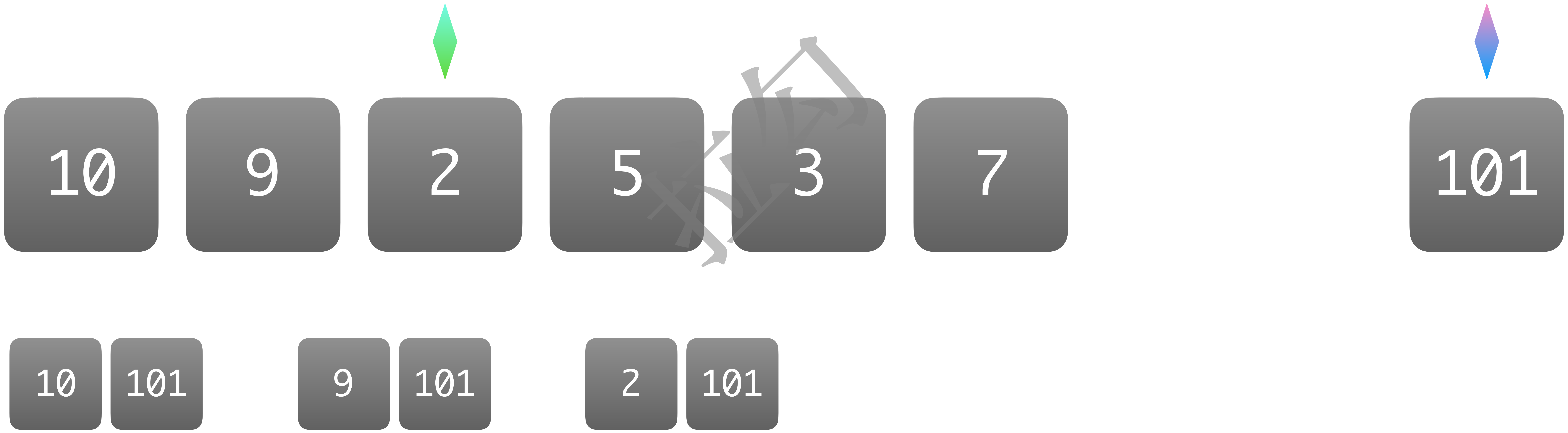


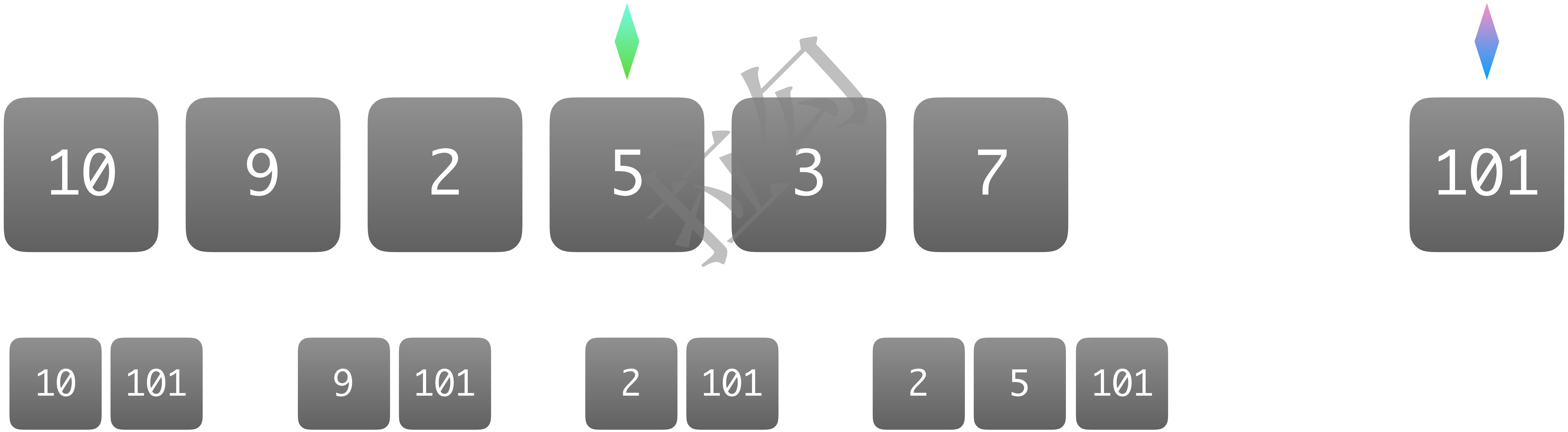
10
9
2
5
3
7
101
4
2
4
2
3
4

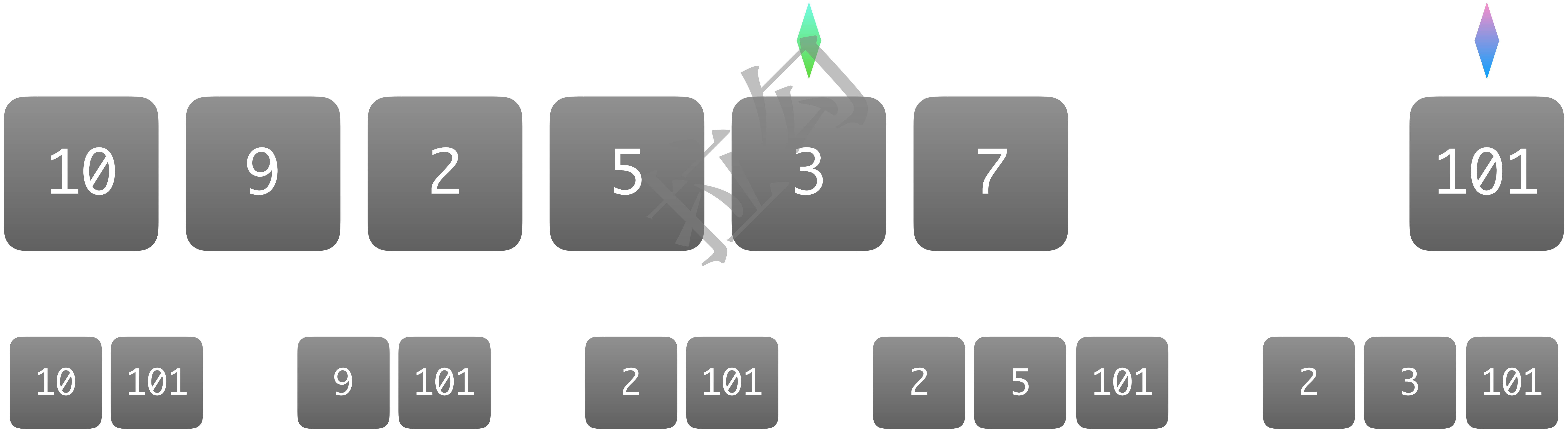

10
9
2
5
3
7
101
4
2
4
2
3
4

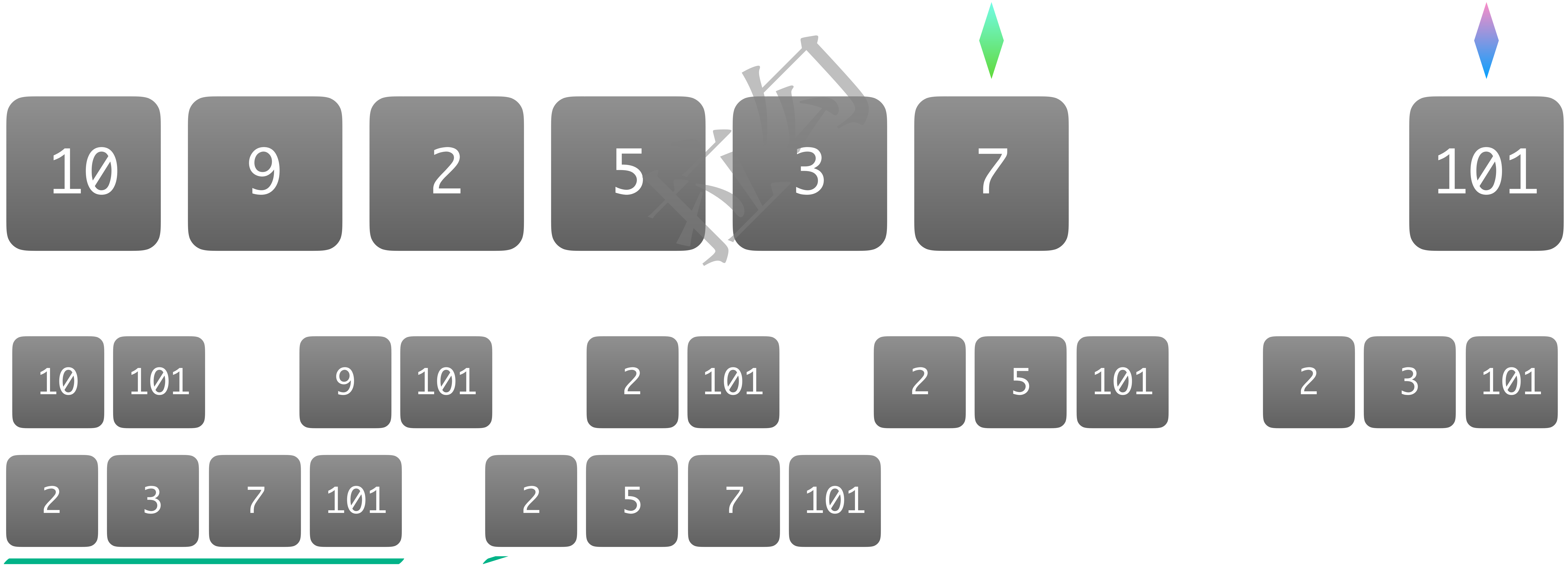

10
9
2
5
3
7
101
10
101


10
9
2
5
3
7
101
10
101
9
101


10
9
2
5
3
7
101
10
101
9
101
2
101


10
9
2
5
3
7
101
10
101
9
101
2
101
2
5
101


10 9 2 5 3 7 101
10 101
9 101
2 101
2 5 101
2 3 101


10 9 2 5 3 7 101
10 101
9 101
2 101
2 5 101
2 3 101
2 3 7 101
2 5 7 101

解决动态规划问题最难的两个地方：

- 如何定义 $f(n)$

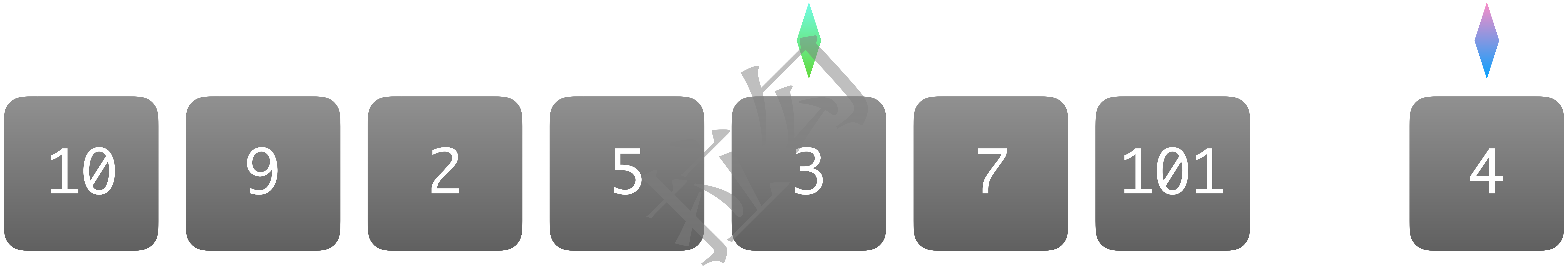
对于这道题而言，$f(n)$ 是以 nums[n-1] 结尾的最长的上升子序列的长度

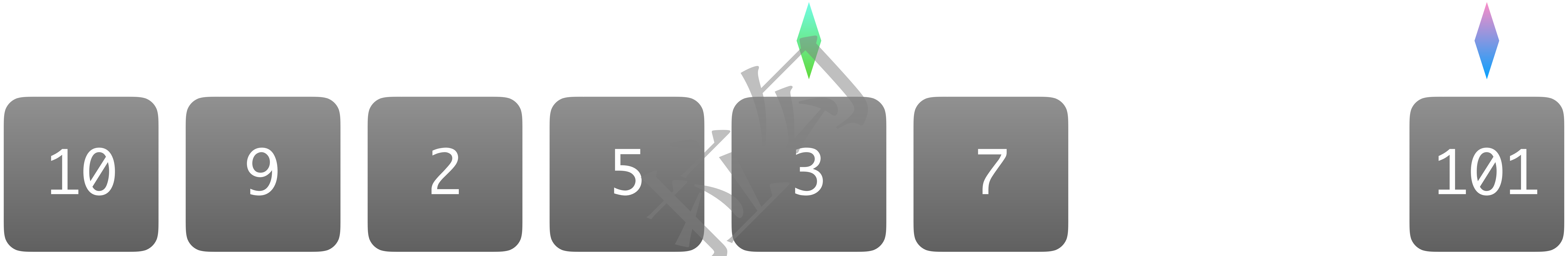
- 如何通过 $f(1), f(2), ..., f(n-1)$ 推导出 $f(n)$，即状态转移方程

- 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：

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

6.1 动态规划 / Dynamic Programming





$f(n) = max\{f(i)\} + 1$ （$1 \leq i < n - 1$，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;
  }
}
```

首先定义一个静态变量 max，

用来保存最终的最长的上升子序列的长度。

```
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;
  }
}
```

接下来看看如何实现状态转移方程，即f函数。

```
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 。

```
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。

6.2 递归 / Recursion

```
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。

```
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。

6.2 递归 / Recursion

```
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。

```
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

- 迭代法
- 公式法

```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

- 公式法

- 当 $n = 1$ 时，递归直接返回 1，执行时间为 $O(1)$

  $T(1) = O(1)$

- 当 $n = 2$ 时，内部调用了一次递归求解 $T(1)$

  $T(2) = T(1)$

- 当 $n = 3$ 时, $T(3) = T(1) + T(2)$

  …以此类推

- $T(n-1) = T(1) + T(2) + \ldots + T(n-2)$
- $T(n) = T(1) + T(2) + \ldots + T(n-1)$

```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;
  }
}
```

时间复杂度分析

- $T(n-1) = T(1) + T(2) + \ldots + T(n-2)$
- $T(n) = T(1) + T(2) + \ldots + T(n-1)$

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

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

```
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。

```
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。

记忆化 / Memoization

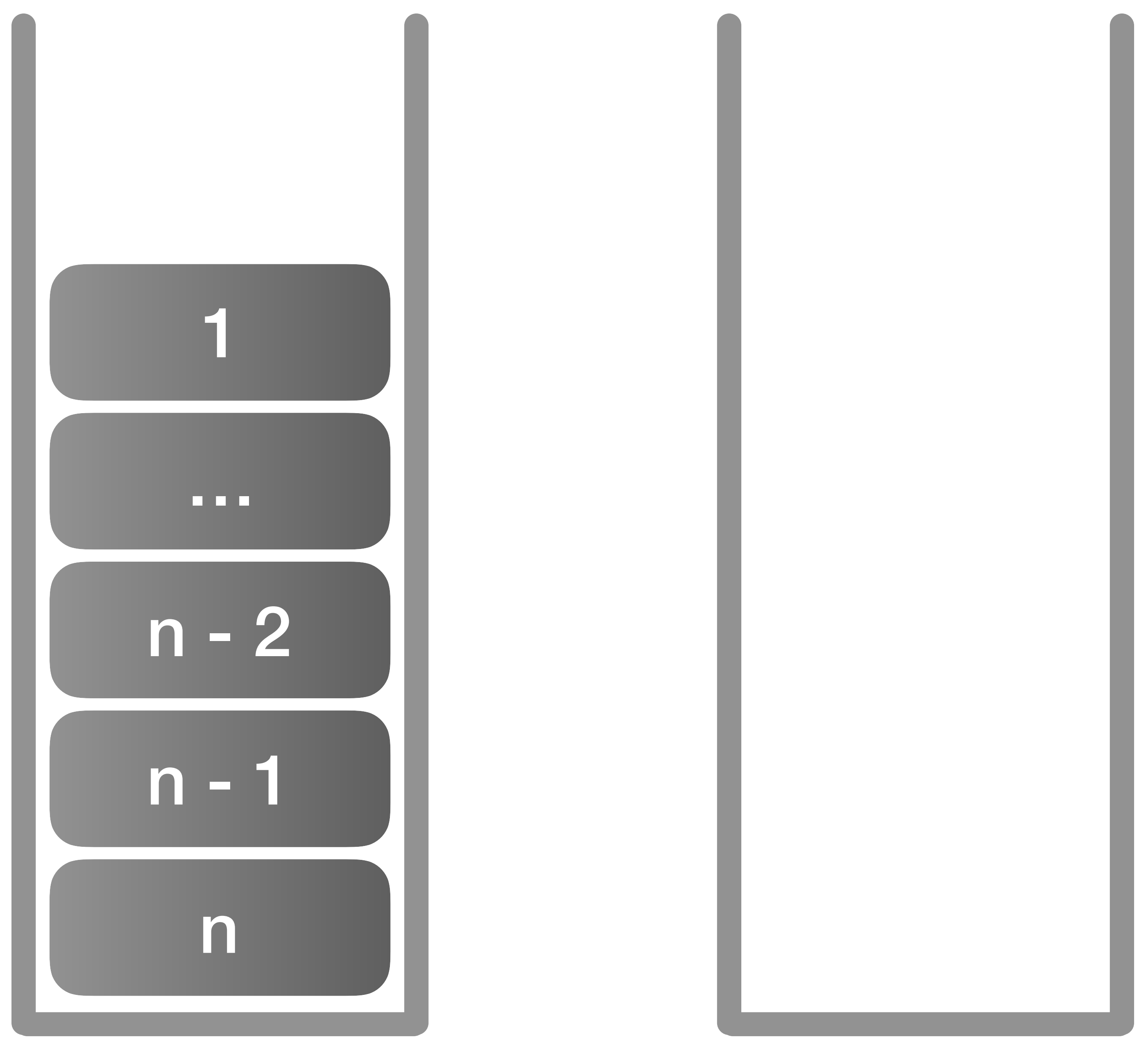

```
class LISMemoization {
  static int max;
  static HashMap<Integer, Integer> cache;

  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。

6.3 记忆化 / Memoization

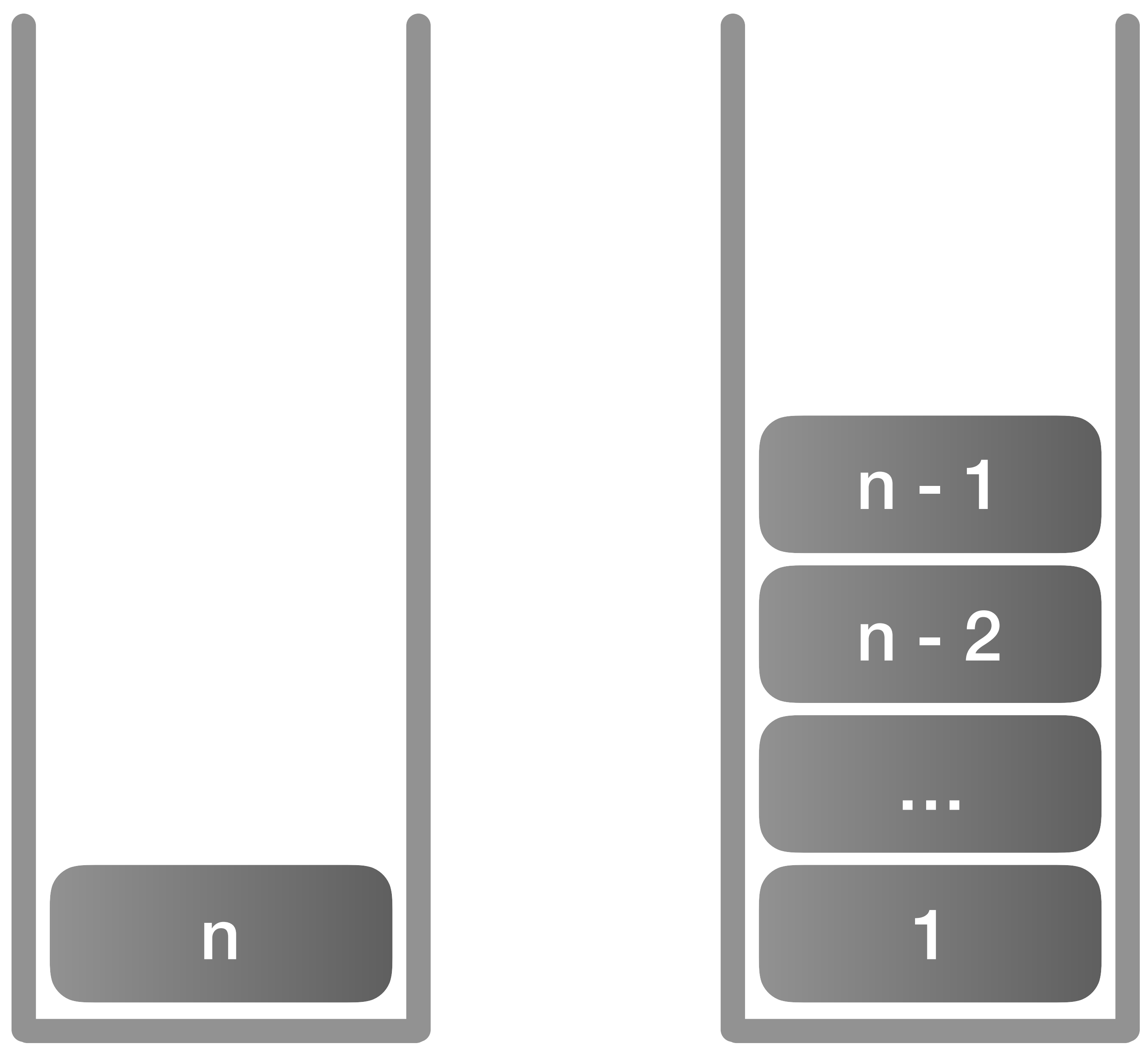

```java
class LISMemoization {
  static int max;
  static HashMap<Integer, Integer> cache;

  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;
  }
}
```


时间复杂度分析



```
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;
  }
}
```

时间复杂度分析



6.3 记忆化 / Memoization

```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;
  }
}
```


时间复杂度分析

$$O(T(n)) = O(T(\boxed{\;n\;} + \ldots + T(n-1))$$

$$= O(1 + 2 + \ldots + n - 1)$$

$$= O(\frac{n \cdot (n-1)}{2}) \quad = O(n^2)$$

$$O(f(n)) = O(n) + O(n^2) = O(n^2) \quad < \quad O(2^n)$$

6.3 记忆化 / Memoization

```
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;
  }
}
```

时间复杂度分析

对于这种将问题规模不断减少的做法，
我们把它称为自顶向下的方法。

由于递归的存在，
程序运行时对堆栈的消耗以及处理很慢，
在实际工作中并不推荐。

自底向上 / Bottom-Up

```
class LISDP {
  public int LIS(int[] nums, int n) {
    int[] cache = new int[n];
    int i, j, max = 0;

    for (i = 0; i < n; i++) cache[i] = 1;

    for (i = 0; i < n; i++) {
      for (j = 0; j < i; j++) {
        if (nums[j] < nums[i] && cache[i] < cache[j] + 1) {
          cache[i] = cache[j] + 1;
        }
      }

      max = Math.max(max, cache[i]);
    }

    return max;
  }
}
```

- 这次我们用一个一维数组 cache 来存储计算过的结果。

6.4 自底向上 / Bottom-Up

```
class LISDP {
  public int LIS(int[] nums, int n) {
    int[] cache = new int[n];
    int i, j, max = 0;

    for (i = 0; i < n; i++) cache[i] = 1;

    for (i = 0; i < n; i++) {
      for (j = 0; j < i; j++) {
        if (nums[j] < nums[i] && cache[i] < cache[j] + 1) {
          cache[i] = cache[j] + 1;
        }
      }

      max = Math.max(max, cache[i]);
    }

    return max;
  }
}
```

- 这次我们用一个一维数组 cache 来存储计算过的结果。
- 初始化 cache 里的每个元素的值为 1，
  表示以每个元素作为结尾的最长子序列的长度初始化为 1。

6.4 自底向上 / Bottom-Up

```
class LISDP {
  public int LIS(int[] nums, int n) {
    int[] cache = new int[n];
    int i, j, max = 0;

    for (i = 0; i < n; i++) cache[i] = 1;

    for (i = 0; i < n; i++) {
      for (j = 0; j < i; j++) {
        if (nums[j] < nums[i] && cache[i] < cache[j] + 1) {
          cache[i] = cache[j] + 1;
        }
      }

      max = Math.max(max, cache[i]);
    }

    return max;
  }
}
```

- 这次我们用一个一维数组 cache 来存储计算过的结果。
- 初始化 cache 里的每个元素的值为 1，
  表示以每个元素作为结尾的最长子序列的长度初始化为 1。
- 自底向上地求解每个子问题的最优解。

6.4 自底向上 / Bottom-Up

```
class LISDP {
  public int LIS(int[] nums, int n) {
    int[] cache = new int[n];
    int i, j, max = 0;

    for (i = 0; i < n; i++) cache[i] = 1;

    for (i = 0; i < n; i++) {
      for (j = 0; j < i; j++) {
        if (nums[j] < nums[i] && cache[i] < cache[j] + 1) {
          cache[i] = cache[j] + 1;
        }
      }

      max = Math.max(max, cache[i]);
    }

    return max;
  }
}
```

- 这次我们用一个一维数组 cache 来存储计算过的结果。
- 初始化 cache 里的每个元素的值为 1，
  表示以每个元素作为结尾的最长子序列的长度初始化为 1。
- 自底向上地求解每个子问题的最优解。
- 拿遍历中遇到的每个元素 nums[j] 与 nums[i] 比较，
  如果发现 nums[j] < nums[i]，
  说明 nums[i] 有机会构成上升序列，
  如果新的上升序列比之前计算过的还要长，
  更新一下，保存到 cache 数组里。

6.4 自底向上 / Bottom-Up

```
class LISDP {
  public int LIS(int[] nums, int n) {
    int[] cache = new int[n];
    int i, j, max = 0;

    for (i = 0; i < n; i++) cache[i] = 1;

    for (i = 0; i < n; i++) {
      for (j = 0; j < i; j++) {
        if (nums[j] < nums[i] && cache[i] < cache[j] + 1) {
          cache[i] = cache[j] + 1;
        }
      }

      max = Math.max(max, cache[i]);
    }

    return max;
  }
}
```

- 这次我们用一个一维数组 cache 来存储计算过的结果。
- 初始化 cache 里的每个元素的值为 1，
  表示以每个元素作为结尾的最长子序列的长度初始化为 1。
- 自底向上地求解每个子问题的最优解。
- 拿遍历中遇到的每个元素 nums[j] 与 nums[i] 比较，
  如果发现 nums[j] < nums[i]，
  说明 nums[i] 有机会构成上升序列，
  如果新的上升序列比之前计算过的还要长，
  更新一下，保存到 cache 数组里。
- 用当前计算好的长度与全局的最大值进行比较。

6.4 自底向上 / Bottom-Up

```
class LISDP {
  public int LIS(int[] nums, int n) {
    int[] cache = new int[n];
    int i, j, max = 0;

    for (i = 0; i < n; i++) cache[i] = 1;

    for (i = 0; i < n; i++) {
      for (j = 0; j < i; j++) {
        if (nums[j] < nums[i] && cache[i] < cache[j] + 1) {
          cache[i] = cache[j] + 1;
        }
      }

      max = Math.max(max, cache[i]);
    }

    return max;
  }
}
```

- 这次我们用一个一维数组 cache 来存储计算过的结果。
- 初始化 cache 里的每个元素的值为 1，
  表示以每个元素作为结尾的最长子序列的长度初始化为 1。
- 自底向上地求解每个子问题的最优解。
- 拿遍历中遇到的每个元素 nums[j] 与 nums[i] 比较，
  如果发现 nums[j] < nums[i]，
  说明 nums[i] 有机会构成上升序列，
  如果新的上升序列比之前计算过的还要长，
  更新一下，保存到 cache 数组里。
- 用当前计算好的长度与全局的最大值进行比较。
- 最后得出最长的上升序列的长度。

```java
class LISDP {
  public int LIS(int[] nums, int n) {
    int[] cache = new int[n];
    int i, j, max = 0;

    for (i = 0; i < n; i++) cache[i] = 1;

    for (i = 0; i < n; i++) {
      for (j = 0; j < i; j++) {
        if (nums[j] < nums[i] && cache[i] < cache[j] + 1) {
          cache[i] = cache[j] + 1;
        }
      }

      max = Math.max(max, cache[i]);
    }

    return max;
  }
}
```

时间复杂度分析

这是一个双重循环

i = 0 时，内循环执行 0 次

i = 1 时，内循环执行 1次

i = n - 1 时，内循环执行 n - 1 次

$$O(1+2+...+n-1) = O(\frac{n \cdot (n-1)}{2}) = O(n^2)$$

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

线性规划

- 各个子问题的规模以线性的方式分布
- 子问题的最佳状态或结果可以存储在一维线性的数据结构中，例如：一位数组，哈希表等
- 通常我们会用 dp[i] 表示第 i 个位置的结果，或者从 0 开始到第 i 个位置为止的最佳状态或结果

基本形式

- 当前所求的值仅仅依赖于有限个先前计算好的值，即 dp[i] 仅仅依赖于有限个 dp[i]，其中 j < i

例题1：求解斐波那契数列时，dp[i] = dp[i - 1] + dp[i - 2]。

当前值只依赖于前面两个计算好的值。

LeetCode 70 爬楼梯，就是一道求解斐波那契数列的题目

例题2：LeetCode 198 给定一个数组，不能选择相邻的数，求如何才能使总数最大。

198. 打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警**。

给定一个代表每个房屋存放金额的非负整数数组，计算你**在不触动警报装置的情况下**，能够偷窃到的最高金额。

示例：
输入：[1, 2, 3, 1]
输出：4

解释：
偷窃 1 号房屋（金额 = 1），
然后偷窃 3 号房屋（金额 = 3）。
偷窃到的最高金额 = 1 + 3 = 4

递归公式：dp[i] = max(nums[i] + dp[i - 2], dp[i - 1])

```java
public int rob(int[] nums) {
  int n = nums.length;

  if(n == 0) return 0;
  if(n == 1) return nums[0];

  int[] dp = new int[n];

  dp[0] = nums[0];
  dp[1] = Math.max(nums[0], nums[1]);

  for (int i = 2; i < n; i++) {
    dp[i] = Math.max(nums[i] + dp[i - 2], dp[i - 1]);
  }

  return dp[n - 1];
}
```

- 首先处理一些基本的情况，
  即当数组为空，或者只有一个元素的时候。

```
public int rob(int[] nums) {
  int n = nums.length;

  if(n == 0) return 0;
  if(n == 1) return nums[0];

  int[] dp = new int[n];

  dp[0] = nums[0];
  dp[1] = Math.max(nums[0], nums[1]);

  for (int i = 2; i < n; i++) {
    dp[i] = Math.max(nums[i] + dp[i - 2], dp[i - 1]);
  }

  return dp[n - 1];
}
```

- 首先处理一些基本的情况，即当数组为空，或者只有一个元素的时候。
- 定义一个 dp 数组，dp[i] 表示到第 i 个元素为止，我们所能收获到的最大总数。

```java
public int rob(int[] nums) {
  int n = nums.length;

  if(n == 0) return 0;
  if(n == 1) return nums[0];

  int[] dp = new int[n];

  dp[0] = nums[0];
  dp[1] = Math.max(nums[0], nums[1]);

  for (int i = 2; i < n; i++) {
    dp[i] = Math.max(nums[i] + dp[i - 2], dp[i - 1]);
  }

  return dp[n - 1];
}
```

- 首先处理一些基本的情况，
  即当数组为空，或者只有一个元素的时候。
- 定义一个 dp 数组，dp[i] 表示到第 i 个元素为止，
  我们所能收获到的最大总数。
- 初始化 dp[0]，dp[1]。

```java
public int rob(int[] nums) {
  int n = nums.length;

  if(n == 0) return 0;
  if(n == 1) return nums[0];

  int[] dp = new int[n];

  dp[0] = nums[0];
  dp[1] = Math.max(nums[0], nums[1]);

  for (int i = 2; i < n; i++) {
    dp[i] = Math.max(nums[i] + dp[i - 2], dp[i - 1]);
  }

  return dp[n - 1];
}
```

- 首先处理一些基本的情况，
  即当数组为空，或者只有一个元素的时候。
- 定义一个 dp 数组，dp[i] 表示到第 i 个元素为止，
  我们所能收获到的最大总数。
- 初始化 dp[0]，dp[1]。
- 对于每个 nums[i]，考虑两种情况：选还是不选。
  然后取最大值。

```java
public int rob(int[] nums) {
  int n = nums.length;

  if(n == 0) return 0;
  if(n == 1) return nums[0];

  int[] dp = new int[n];

  dp[0] = nums[0];
  dp[1] = Math.max(nums[0], nums[1]);

  for (int i = 2; i < n; i++) {
    dp[i] = Math.max(nums[i] + dp[i - 2], dp[i - 1]);
  }

  return dp[n - 1];
}
```


- 首先处理一些基本的情况，
  即当数组为空，或者只有一个元素的时候。
- 定义一个 dp 数组，dp[i] 表示到第 i 个元素为止，
  我们所能收获到的最大总数。
- 初始化 dp[0]，dp[1]。
- 对于每个 nums[i]，考虑两种情况：选还是不选。
  然后取最大值。
- 最后返回 dp[n-1]，即最终的结果。

62. 不同路径

一个机器人位于一个$m \times n$网格的左上角（起始点在下图中标记为“Start”）。

机器人每次只能向下或向右移动一步。机器人试图到达网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径?

说明： m和n的值均不超过100。

62. 不同路径

一个机器人位于一个$m \times n$网格的左上角（起始点在下图中标记为“Start”）。

机器人每次只能向下或向右移动一步。机器人试图到达网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径?

说明： m和n的值均不超过100。

示例：
输入： m = 3, n = 2
输出： 3

解释：
从左上角开始，总共有 3 条路径可以到达右下角。
1. 向右 -> 向右 -> 向下
2. 向右 -> 向下 -> 向右
3. 向下 -> 向右 -> 向右

递归公式：dp[i][j] = dp[i - 1][j] + dp[i][j - 1]

基本形式

- 当前所求的值仅仅依赖于有限个先前计算和的值，即 dp[i] 仅仅依赖于有限个 dp[i]，其中 $j < i$
- 当前所求的值仅仅依赖于所有先前计算和的值，即 dp[i] 是各个 dp[j] 的某种组合，其中 j 由 0 遍历到 $i - 1$

例题：在求解最长上升子序列时，$dp[i] = \max(dp[j] + 1) + 1, 0 \le j < i$。

当前值依赖于前面所有计算好的值。

区间规划

- 各个子问题的规模由不同区间来定义。
- 子问题的最佳状态或结果可以存储在二维数组中。
- 这类问题的时间复杂度一般为多项式时间，即对于一个大小为 n 的问题，时间复杂度不会超过 n 的多项式倍数。

516. 最长回文子序列

给定一个字符串 s，找到其中最长的回文子序列。

可以假设 s 的最大长度为 1000。

示例1：
输入：”dccac”
输出：3
一个可能的最长回文子序列为”ccc”

示例2：
输入：”bbbab”
输出：4
一个可能的最长回文子序列为”bbbb”

递归公式：

当首尾的两个字符相等时：dp[0][n - 1] = dp[1][n - 2] + 2

否则：dp[0][n - 1] = max(dp[1][n - 1], dp[0][n - 2])

```java
public static int LPS(String s) {
  int n = s.length();
  int[][] dp = new int[n][n];

  for (int i = 0; i < n; i++) dp[i][i] = 1;

  for (int len = 2; len <= n; len++) {
    for (int i = 0; i < n - len + 1; i++) {
      int j = i + len - 1;

      if (s.charAt(i) == s.charAt(j)) {
        dp[i][j] = 2 + (len == 2 ? 0: dp[i + 1][j - 1]);
      } else {
        dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
      }
    }
  }

  return dp[0][n - 1];
}
```

- 定义 dp 矩阵，
  dp[i][j] 表示从字符串第 i 个字符
  到第 j 个字符之间的最长回文。

```
public static int LPS(String s) {
  int n = s.length();
  int[][] dp = new int[n][n];

  for (int i = 0; i < n; i++) dp[i][i] = 1;

  for (int len = 2; len <= n; len++) {
    for (int i = 0; i < n - len + 1; i++) {
      int j = i + len - 1;

      if (s.charAt(i) == s.charAt(j)) {
        dp[i][j] = 2 + (len == 2 ? 0: dp[i + 1][j - 1]);
      } else {
        dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
      }
    }
  }

  return dp[0][n - 1];
}
```

- 定义 dp 矩阵，
  dp[i][j] 表示从字符串第 i 个字符到第 j 个字符之间的最长回文。
- 初始化 dp 矩阵，
  将对角线元素设为 1，即单个字符的回文长度为 1。

```
public static int LPS(String s) {
  int n = s.length();
  int[][] dp = new int[n][n];

  for (int i = 0; i < n; i++) dp[i][i] = 1;

  for (int len = 2; len <= n; len++) {
    for (int i = 0; i < n - len + 1; i++) {
      int j = i + len - 1;

      if (s.charAt(i) == s.charAt(j)) {
        dp[i][j] = 2 + (len == 2 ? 0: dp[i + 1][j - 1]);
      } else {
        dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
      }
    }
  }

  return dp[0][n - 1];
}
```

- 定义 dp 矩阵，
  dp[i][j] 表示从字符串第 i 个字符
  到第 j 个字符之间的最长回文。
- 初始化 dp 矩阵，
  将对角线元素设为 1，即单个字符的回文长度为 1。
- 从长度为 2 开始，尝试将区间扩大，一直扩大到 n。

```java
public static int LPS(String s) {
  int n = s.length();
  int[][] dp = new int[n][n];

  for (int i = 0; i < n; i++) dp[i][i] = 1;

  for (int len = 2; len <= n; len++) {
    for (int i = 0; i < n - len + 1; i++) {
      int j = i + len - 1;

      if (s.charAt(i) == s.charAt(j)) {
        dp[i][j] = 2 + (len == 2 ? 0: dp[i + 1][j - 1]);
      } else {
        dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
      }
    }
  }

  return dp[0][n - 1];
}
```

- 定义 dp 矩阵，
  dp[i][j] 表示从字符串第 i 个字符
  到第 j 个字符之间的最长回文。
- 初始化 dp 矩阵，
  将对角线元素设为 1，即单个字符的回文长度为 1。
- 从长度为 2 开始，尝试将区间扩大，一直扩大到 n。
- 在扩大的过程中，
  每次都得出区间的起始位置 i 和结束位置 j。

```
public static int LPS(String s) {
  int n = s.length();
  int[][] dp = new int[n][n];

  for (int i = 0; i < n; i++) dp[i][i] = 1;

  for (int len = 2; len <= n; len++) {
    for (int i = 0; i < n - len + 1; i++) {
      int j = i + len - 1;

      if (s.charAt(i) == s.charAt(j)) {
        dp[i][j] = 2 + (len == 2 ? 0: dp[i + 1][j - 1]);
      } else {
        dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
      }
    }
  }

  return dp[0][n - 1];
}
```

- 定义 dp 矩阵，
  dp[i][j] 表示从字符串第 i 个字符
  到第 j 个字符之间的最长回文。
- 初始化 dp 矩阵，
  将对角线元素设为 1，即单个字符的回文长度为 1。
- 从长度为 2 开始，尝试将区间扩大，一直扩大到 n。
- 在扩大的过程中，
  每次都得出区间的起始位置 i 和结束位置 j。
- 比较一下区间首尾的字符是否相等。

```
public static int LPS(String s) {
  int n = s.length();
  int[][] dp = new int[n][n];

  for (int i = 0; i < n; i++) dp[i][i] = 1;

  for (int len = 2; len <= n; len++) {
    for (int i = 0; i < n - len + 1; i++) {
      int j = i + len - 1;

      if (s.charAt(i) == s.charAt(j)) {
        dp[i][j] = 2 + (len == 2 ? 0: dp[i + 1][j - 1]);
      } else {
        dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
      }
    }
  }

  return dp[0][n - 1];
}
```

- 定义 dp 矩阵，
  dp[i][j] 表示从字符串第 i 个字符
  到第 j 个字符之间的最长回文。
- 初始化 dp 矩阵，
  将对角线元素设为 1，即单个字符的回文长度为 1。
- 从长度为 2 开始，尝试将区间扩大，一直扩大到 n。
- 在扩大的过程中，
  每次都得出区间的起始位置 i 和结束位置 j。
- 比较一下区间首尾的字符是否相等。
- 如果相等，就加2。

```java
public static int LPS(String s) {
  int n = s.length();
  int[][] dp = new int[n][n];

  for (int i = 0; i < n; i++) dp[i][i] = 1;

  for (int len = 2; len <= n; len++) {
    for (int i = 0; i < n - len + 1; i++) {
      int j = i + len - 1;

      if (s.charAt(i) == s.charAt(j)) {
        dp[i][j] = 2 + (len == 2 ? 0: dp[i + 1][j - 1]);
      } else {
        dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
      }
    }
  }

  return dp[0][n - 1];
}
```

- 定义 dp 矩阵，
  dp[i][j] 表示从字符串第 i 个字符
  到第 j 个字符之间的最长回文。
- 初始化 dp 矩阵，
  将对角线元素设为 1，即单个字符的回文长度为 1。
- 从长度为 2 开始，尝试将区间扩大，一直扩大到 n。
- 在扩大的过程中，
  每次都得出区间的起始位置 i 和结束位置 j。
- 比较一下区间首尾的字符是否相等。
- 如果相等，就加2。
- 如果不等，从规模更小的字符串中得出最长的回文长度。

在普通的线性规划和区间规划里，一般题目有两种需求：

- 统计
- 最优解

例题：0-1背包问题

给定 n 个物品，每个物品有各自的价值 v_i 和重量 w_i，在限定的最大重量 W 内，我们如何选择，才能使被带走的物品的价值总和最大?

非决定性多项式 / Non-deterministic Polynomial

- 时间复杂度

程序运行的时间随着问题规模扩大的增长得有多快。

- 非多项式级时间复杂度

指数级复杂度，如 $O(2^n)$，$O(3^n)$

全排列算法，复杂度为 $O(n!)$。

- 多项式级时间复杂度

$O(1)$，$O(n)$，$O(n \times log(n))$，$O(n^2)$，$O(n^3)$等。

6.5 约束规划 / Constraint Programming

例题：0-1背包问题

给定 n 个物品，每个物品有各自的价值 v_i 和重量 w_i，在限定的最大重量 W 内，我们如何选择，才能使被带走的物品的价值总和最大?

时间复杂度：$O(n \times W)$

0.00000　　0.00001　　0.00002　　…　　21.17008

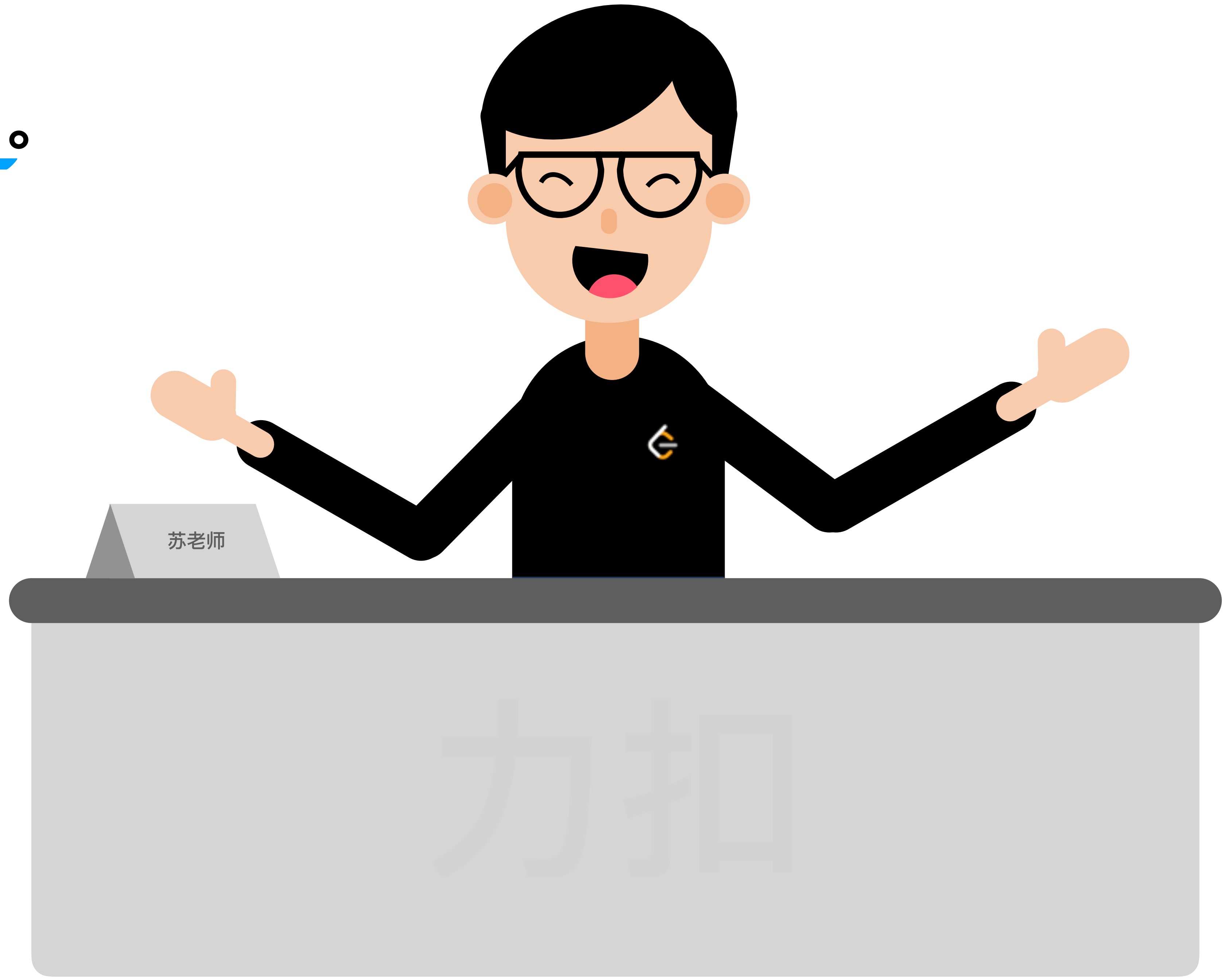
210多万个单元

假如要能覆盖从 0 到 W 的区间需要 m 位二进制数，则 W 可写作 2^m。

时间复杂度：$O(n \times 2^m)$　　-> 多项式级时间复杂度

6.6 结束 / Ending

学习动态规划，除了掌握好我们所讲的知识点，更重要的是多练。



6.6 结束 / Ending

Next：课时 7《15分钟搞懂二分搜索与贪婪》

多加练习，才能更好地巩固知识点。

关注 “拉勾教育”
学习技术干货

关注 “LeetCode力扣”
获得算法技术干货