

# 从记忆化搜索入门动态规划

## Memoization Search & Dynamic Programming

课程版本 v6.1

主讲 令狐冲



扫描二维码关注微信/微博  
获取最新面试题及权威解答

微信: [ninechapter](#)

微博: <http://www.weibo.com/ninechapter>

知乎: <http://zhuoanlan.zhihu.com/jiuzhang>

官网: <http://www.jiuzhang.com>

- <http://www.lintcode.com/problem/triangle/>
- <http://www.jiuzhang.com/solutions/triangle/>
- 解决方法:
- DFS: Traverse
- DFS: Divide Conquer
- Divide Conquer + Memoization
- Traditional Dynamic Programming

- 时间复杂度?
- A -  $O(n^2)$
- B -  $O(2^n)$
- C -  $O(n!)$
- D - I don't know

```
def minimumTotal(self, triangle):  
    self.minimum = sys.maxsize  
    self.traverse(triangle, 0, 0, 0)  
    return self.minimum  
  
def traverse(self, triangle, x, y, path_sum):  
    if x == len(triangle):  
        self.minimum = min(path_sum, self.minimum)  
        return  
  
    self.traverse(triangle, x + 1, y, path_sum + triangle[x][y])  
    self.traverse(triangle, x + 1, y + 1, path_sum + triangle[x][y])
```

- 时间复杂度?
- A -  $O(n^2)$
- B -  $O(2^n)$
- C -  $O(n!)$
- D - I don't know

```
def minimumTotal(self, triangle):  
    return self.divide_conquer(triangle, 0, 0)  
  
def divide_conquer(self, triangle, x, y):  
    if x == len(triangle):  
        return 0  
  
    left = self.divide_conquer(triangle, x + 1, y)  
    right = self.divide_conquer(triangle, x + 1, y + 1)  
    return min(left, right) + triangle[x][y]
```

# 记忆化搜索 Memoization Search

注意不是 Memorization

将函数的计算结果保存下来，下次通过同样的参数访问时，直接返回保存下来的结果

问：

1. 对这个函数有什么限制条件没有？
2. 和系统设计中的什么很像？

记忆化搜索通常能够将指数级别的时间复杂度降低到多项式级别。

# 独孤九剑 —— 破箭式

三种适用动态规划的场景

三种不适用动态规划的场景

- 时间复杂度?
- A -  $O(n^2)$
- B -  $O(2^n)$
- C -  $O(n!)$
- D - I don't know

```
def minimumTotal(self, triangle):  
    return self.divide_conquer(triangle, 0, 0, {})  
  
# 函数返回从 x, y 出发, 走到最底层的最短路径值  
# memo 中 key 为二元组 (x, y)  
# memo 中 value 为从 x, y 走到最底层的最短路径值  
def divide_conquer(self, triangle, x, y, memo):  
    if x == len(triangle):  
        return 0  
  
    # 如果找过了, 就不要再找了, 直接把之前找到的值返回  
    if (x, y) in memo:  
        return memo[(x, y)]  
  
    left = self.divide_conquer(triangle, x + 1, y, memo)  
    right = self.divide_conquer(triangle, x + 1, y + 1, memo)  
  
    # 在 return 之前先把这次找到的最短路径值记录下来  
    # 避免之后重复搜索  
    memo[(x, y)] = min(left, right) + triangle[x][y]  
    return memo[(x, y)]
```



# 记忆化搜索的本质：动态规划

动态规划为什么会快？

动态规划与分治的区别？

重复计算！

# 记忆化搜索 = 动态规划(DP)

记忆化搜索是动态规划的一种实现方式

记忆化搜索是用搜索的方式实现了动态规划

因此记忆化搜索，就是动态规划

# 三种适用DP的场景

求最值（max/min）

求方案总数（sum）

求可行性（or）

# 三种不适用DP的场景

求所有的具体方案

输入数据是无序的

暴力算法时间复杂度已经是多项式级别

## 三种不适用 DP 的场景

- 求出所有的具体方案
  - <http://www.lintcode.com/problem/palindrome-partitioning/>
  - 只求出一个具体方案还是可以用 DP 来做的
- 输入数据是无序的
  - <http://www.lintcode.com/problem/longest-consecutive-sequence/>
  - 背包类动态规划不适用此判断条件
- 暴力算法的复杂度已经是多项式级别
  - <http://www.lintcode.com/problem/largest-rectangle-in-histogram/>
  - 动态规划擅长与优化指数级别复杂度( $2^n, n!$ )到多项式级别复杂度( $n^2, n^3$ )
  - 不擅长优化 $n^3$ 到 $n^2$
- 则 **极不可能** 使用动态规划求解

# Wildcard Matching

<http://www.lintcode.com/problem/wildcard-matching/>

<http://www.jiuzhang.com/solution/wildcard-matching/>

类别：双序列型动态规划

适用场景：求可行性

# Follow up: Regular Expression Matching

<http://www.lintcode.com/problem/regular-expression-matching/>

<http://www.jiuzhang.com/solution/regular-expression-matching/>

面试是一定不会让你做完整版的 Regular Expression 的  
所以一定是阉割版的

**Strong Hire:** 两个都答出来，且写出来，Bug Free or Bug 很少

**Hire / Weak Hire:** 两个都答出来，写完第一个，第二个能基本在第一个的基础上改完，允许有一些提示和少量 Bug

**No Hire:** 没写完，或者需要很多提示

**Strong No:** 第一个都没写完



# 休息 5 分钟

Take a break

# Word Pattern II

<http://www.lintcode.com/problem/word-pattern-ii/>

<http://www.jiuzhang.com/solutions/word-pattern-ii/>

这个题是否可以记忆化？

# Word Break

<http://www.lintcode.com/problem/word-break/>

<http://www.jiuzhang.com/solution/word-break/>

类别：序列性动态规划

适用场景：求可行性

右边的代码正确性没有问题

但是存在一个问题导致其无法通过测试

这个问题是什么？

```
11 def is_possible(self, s, index, max_length, dict, memo):
12     if index in memo:
13         return memo[index]
14
15     if index == len(s):
16         return True
17
18     memo[index] = False
19     for i in range(index, len(s)):
20         if i + 1 - index > max_length:
21             break
22         word = s[index: i + 1]
23         if word not in dict:
24             continue
25         if self.is_possible(s, i + 1, max_length, dict, memo):
26             memo[index] = True
27             break
28
29     return memo[index]
30
31 def get_max_length(self, dict):
32     max_length = 0
33     for word in dict:
34         max_length = max(max_length, len(word))
35     return max_length
```

# 记忆化搜索的缺陷

递归深度太深，导致 StackOverflow

# Word Break II

<http://www.lintcode.com/problem/word-break-ii/>

<http://www.jiuzhang.com/solution/word-break-ii/>

不适用场景：求出所有具体方案而非方案总数  
但是可以使用动态规划进行优化

# 优化方案1

用 Word Break 这个题的思路

使用 `is_possible[i]` 代表从 `i` 开始的后缀是否能够被 break

在 DFS 找所有方案的时候，通过 `is_possible` 可以进行可行性剪枝

## 优化方案 2

直接使用 `memo[i]` 记录从位置 `i` 开始的后缀  
能够被 `break` 出来的所有方案



# 极端情况

以上两种方法在极端情况下是否能有优化效果呢？

$s = \text{"aaaaaaaaaaaa..."}$

$dict = \{\text{"a"}, \text{"aa"}, \text{"aaa"}, \dots\}$

## Word Break II 的面试评分标准

---

**Strong Hire:** DFS+DP优化

**Hire / Weak Hire:** DFS 能写完，且 Bug free or Bug 不多，不需要提示 or 需要少量提示

**No Hire:** DFS 写不完，或者需要很多提示

**Strong No:** 啥都想不出

# \* Palindrome Partitioning

<http://www.lintcode.com/problem/palindrome-partitioning/>

<http://www.jiuzhang.com/solutions/palindrome-partitioning/>

一个类似 Word Break II 的题  
但是使用记忆化搜索优化效果甚微

# Word Break III

<https://www.lintcode.com/problem/word-break-iii>

<https://www.jiuzhang.com/solution/word-break-iii>

类别：序列性动态规划

适用场景：求方案总数