动态规划

Dynamic Programming

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

动态规划的应用

- 生物信息学
- 控制科学
- 信息论
- 运筹学
- 计算机科学：图、AI(强化学习)、机器人等

如 edit distance(编辑距离)可以用于 DNA比较、防止剽窃

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

动态规划的应用例子：

- 如 edit distance(编辑距离)
  可以用于 DNA比较、防止剽窃

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

动态规划的应用例子：

- 隐马尔科夫模型HMM的Viterbi算法

 Viterbi算法是一种动态编程算法，用于获得最可能的隐藏状态序列的最大后验概率估计，即所谓的Viterbi路径。

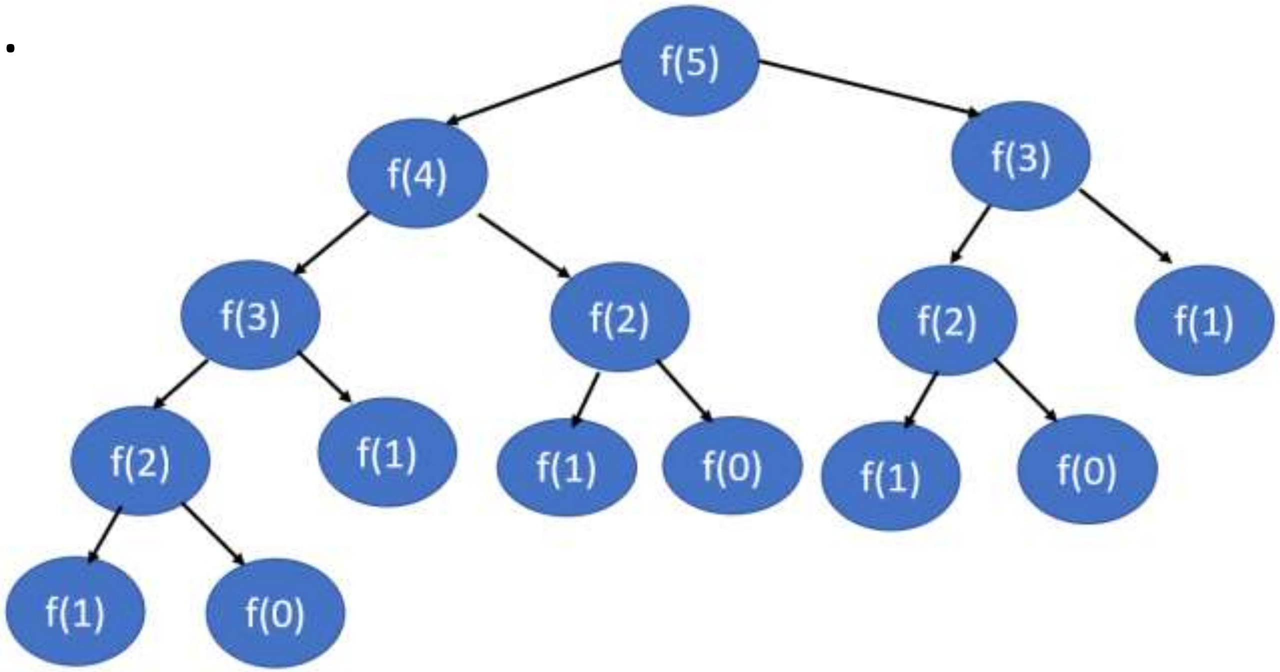
动态规划 =分治递归+记忆存储

- 是对分治递归的优化，通过存储子问题的解，避免重叠子问题的重复计算。
- 如斐波那契数的计算问题。

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

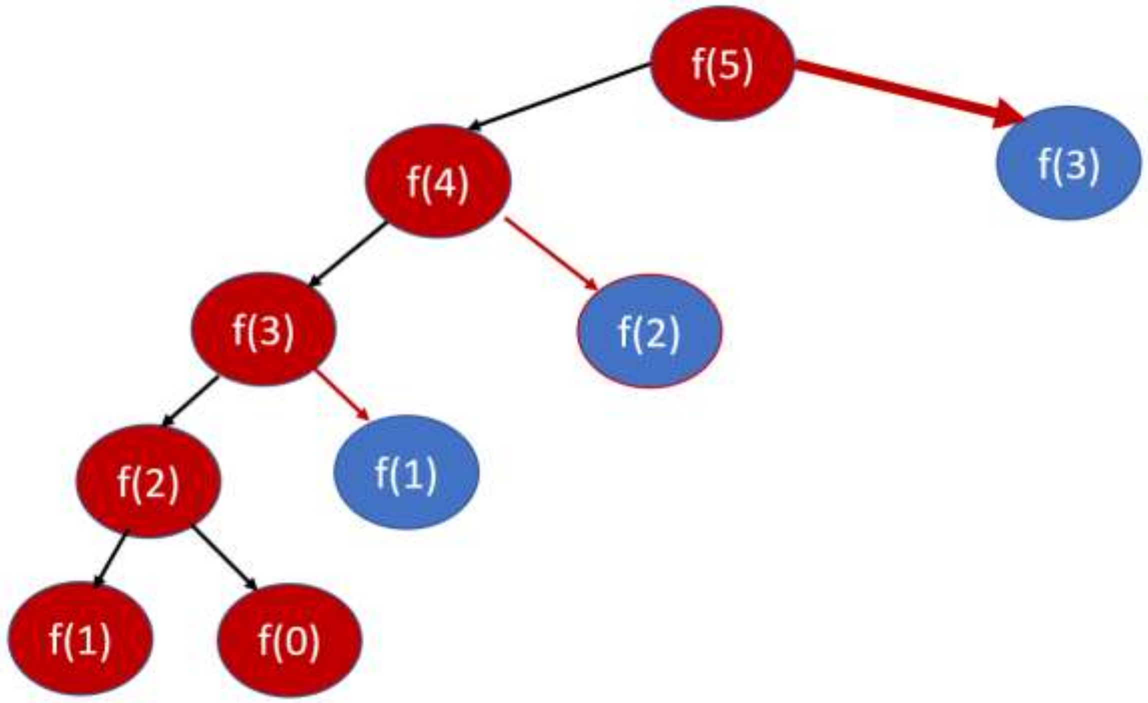
$F_0=0$, $F_1=1$

$F_n = F_{n-1} + F_{n-2}$，当n>=2



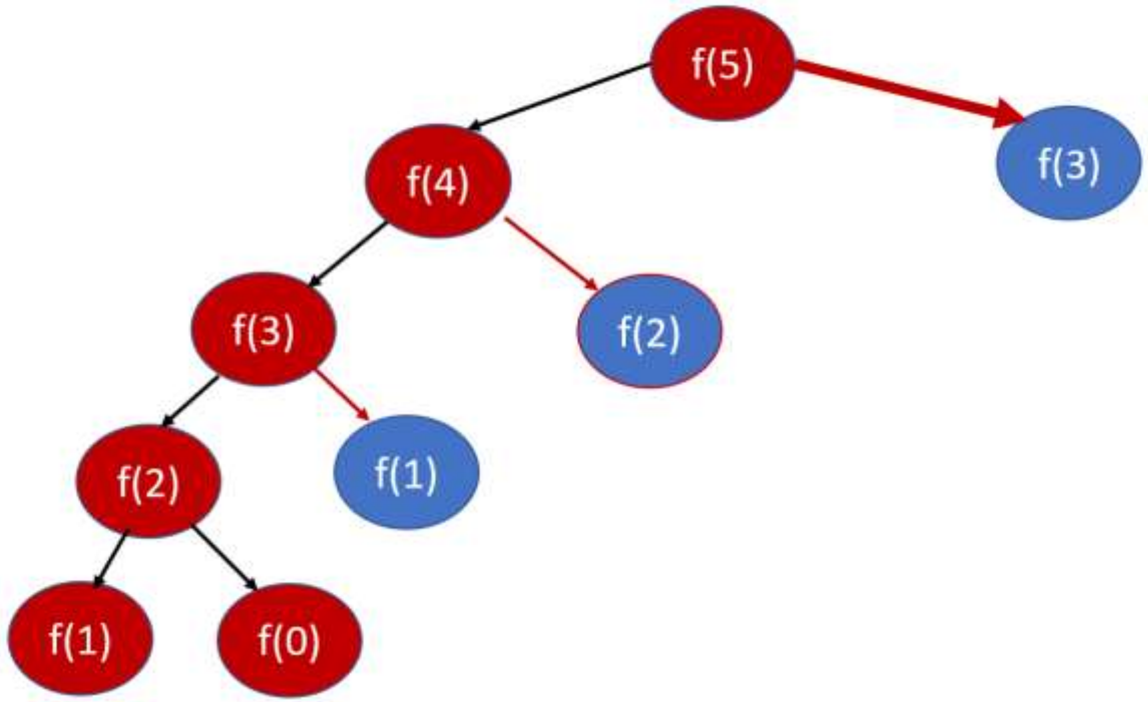
动态规划 =分治递归+记忆存储

- 是对分治递归的优化，通过存储子问题的解，避免重叠子问题的重复计算。
- 如斐波那契数的计算问题。



动态规划 =分治递归+记忆存储

- 是对分治递归的优化，通过存储子问题的解，避免重叠子问题的重复计算。
- 这种简单的优化可以将时间复杂度从指数型降低到多项式型。



```
int fib(int n):
    if n<=1 :  return n
    return fib(n-1)+fib(n-2)
```

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

$T(1) = 0$

$\Downarrow$

$T(n) < 2T(n-1)+1 = O(2^n)$

$T(n) > 2T(n-2)+1 = \Omega(2^{n/2})$

```
int fib(int n):
    if n<=1 :  return n
    return fib(n-1)+fib(n-2)
```

$T(n) = T(n-1)+T(n-2)+1$
$T(1) = 0$

$\Downarrow$

$T(n) < 2T(n-1)+1 = O(2^n)$
$T(n) > 2T(n-2)+1 = \Omega(2^{n/2})$

0	1	-1	-1	-1	-1
f(0)	f(2)	f(2)	f(3)	f(4)	f(5)

```
int fib(int n):
    if n<=1 :  return n
    return fib(n-1)+fib(n-2)
```

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

$T(1) = 0$

$T(n) < 2T(n-1)+1 = O(2^n)$

$T(n) > 2T(n-2)+1 = \Omega(2^{n/2})$

```
int fib(int n,f)
    if n<=1 return n;
    if f[n]<0:
        f[n] = fib(n-1)+fib(n-2)
    return f[n]
```

0	1	-1	-1	-1	-1
f(0)	f(2)	f(2)	f(3)	f(4)	f(5)

```
int fib(int n):
    if n<=1 :  return n
    return fib(n-1)+fib(n-2)
```

```
int fib(int n,f)
    if n<=1 return n;
    if f[n]<0:
        f[n] = fib(n-1)+fib(n-2)
    return f[n]
```

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

$T(1) = 0$

$\Downarrow$

$T(n) < 2T(n-1)+1 = O(2^n)$

$T(n) > 2T(n-2)+1 = \Omega(2^{n/2})$

0	1	-1	-1	-1	-1
f(0)	f(2)	f(2)	f(3)	f(4)	f(5)

```
int fib(int n):
    if n<=1 :  return n
    return fib(n-1)+fib(n-2)
```

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

$T(1) = 0$

$\Downarrow$

$T(n) < 2T(n-1)+1 = O(2^n)$

$T(n) > 2T(n-2)+1 = \Omega(2^{n/2})$

```
int fib(int n,f)
    if n<=1 return n;
    if f[n]<0:
        f[n] = fib(n-1)+fib(n-2)
    return f[n]
```

0	1	-1	-1	-1	-1
f(0)	f(2)	f(2)	f(3)	f(4)	f(5)

```
int fib(int n):
    if n<=1 :  return n
    return fib(n-1)+fib(n-2)
```

$$T(n) = T(n-1)+T(n-2)+1$$
$$T(1) = 0$$
$$\Downarrow$$
$$T(n) < 2T(n-1)+1=O(2^n)$$
$$T(n) > 2T(n-2)+1 = \Omega(2^{n/2})$$

```
int fib(int n,f)
    if n<=1 return n;
    if f[n]<0:
        f[n] = fib(n-1)+fib(n-2)
    return f[n]
```

0	1	-1	-1	-1	-1
f(0)	f(2)	f(2)	f(3)	f(4)	f(5)

```
int fib(int n):
    if n<=1 :  return n
    return fib(n-1)+fib(n-2)
```

$T(n) = T(n-1)+T(n-2)+1$
$T(1) = 0$

$\Downarrow$

$T(n) < 2T(n-1)+1 = O(2^n)$
$T(n) > 2T(n-2)+1 = \Omega(2^{n/2})$

```
int fib(int n,f)
    if n<=1 return n;
    if f[n]<0:
        f[n] = fib(n-1)+fib(n-2)
    return f[n]
```

0	1	1	-1	-1	-1
f(0)	f(2)	f(2)	f(3)	f(4)	f(5)

```
int fib(int n):
    if n<=1 :  return n
    return fib(n-1)+fib(n-2)
```

$$T(n) = T(n-1)+T(n-2)+1$$
$$T(1) = 0$$

$$\Downarrow$$

$$T(n) < 2T(n-1)+1 = O(2^n)$$
$$T(n) > 2T(n-2)+1 = \Omega(2^{n/2})$$

```
int fib(int n,f)
    if n<=1 return n;
    if f[n]<0:
        f[n] = fib(n-1)+fib(n-2)
    return f[n]
```

0	1	1	2	-1	-1
f(0)	f(2)	f(2)	f(3)	f(4)	f(5)

```
int fib(int n):
    if n<=1 :  return n
    return fib(n-1)+fib(n-2)
```

```
int fib(int n,f)
    if n<=1 return n;
    if f[n]<0:
        f[n] = fib(n-1)+fib(n-2)
    return f[n]
```

$T(n) = T(n\text{-}1)+T(n\text{-}2)+1$
$T(1) = 0$

$T(n) < 2T(n\text{-}1)+1 = O(2^n)$
$T(n) > 2T(n\text{-}2)+1 = \Omega(2^{n/2})$

0	1	1	2	3	-1
f(0)	f(2)	f(2)	f(3)	f(4)	f(5)

```
int fib(int n):
    if n<=1 :  return n
    return fib(n-1)+fib(n-2)
```

```
int fib(int n,f)
    if n<=1 return n;
    if f[n]<0:
        f[n] = fib(n-1)+fib(n-2)
    return f[n]
```

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

$T(1) = 0$

$\Downarrow$

$T(n) < 2T(n-1)+1=O(2^n)$

$T(n) > 2T(n-2)+1 = \Omega(2^{n/2})$

0	1	1	2	3	5
f(0)	f(2)	f(2)	f(3)	f(4)	f(5)

```
int fib(int n):
    if n<=1 :  return n
    return fib(n-1)+fib(n-2)
```

$$T(n) = T(n-1)+T(n-2)+1$$
$$T(1) = 0$$

$$T(n) < 2T(n-1)+1=O(2^n)$$
$$T(n) > 2T(n-2)+1 = \Omega(2^{n/2})$$

```
int fib(int n,f)
    if n<=1 return n;
    if f[n]<0:
        f[n] = fib(n-1)+fib(n-2)
    return f[n]
```

0	1	1	2	3	5
f(0)	f(2)	f(2)	f(3)	f(4)	f(5)

递归：自顶向下

递推：自底向上

```
int fib(int n){
    if (n<=1) return n;
    return fib(n-1)+fib(n-2)
}
```

```
int fib(int n){
    int *f = new int[n+1];
    f[0]=0;
    f[1] = 1;
    for (int i=2;i<=n; i++)
        f[i] = f[i-1]+f[i-2]
    return f[n]
}
```

0	1	-1	-1	-1	-1

递推：自底向上

```
int fib(int n){
    if (n<=1) return n;
    return fib(n-1)+fib(n-2)
}
```

```
int fib(int n){
    int *f = new int[n+1];
    f[0]=0;
    f[1] = 1;
    for (int i=2;i<=n; i++)
        f[i] =  f[i-1]+f[i-2]
    return f[n]
}
```

0	1	1	-1	-1	-1

递推：自底向上

```
int fib(int n){
    if (n<=1) return n;
    return fib(n-1)+fib(n-2)
}
```

```
int fib(int n){
    int *f = new int[n+1];
    f[0]=0;
    f[1] = 1;
    for (int i=2;i<=n; i++)
        f[i] =  f[i-1]+f[i-2]
    return f[n]
}
```

0	1	1	-1	-1	-1

递推：自底向上

```
int fib(int n){
    if (n<=1) return n;
    return fib(n-1)+fib(n-2)
}
```

```
int fib(int n){
    int *f = new int[n+1];
    f[0]=0;
    f[1] = 1;
    for (int i=2;i<=n; i++)
        f[i] =  f[i-1]+f[i-2]
    return f[n]
}
```

0	1	1	2	-1	-1

递推：自底向上

```
int fib(int n){
    if (n<=1) return n;
    return fib(n-1)+fib(n-2)
}
```

```
int fib(int n){
    int *f = new int[n+1];
    f[0]=0;
    f[1] = 1;
    for (int i=2;i<=n; i++)
        f[i] =  f[i-1]+f[i-2]
    return f[n]
}
```

0	1	1	2	3	-1

递推：自底向上

```
int fib(int n){
    if (n<=1) return n;
    return fib(n-1)+fib(n-2)
}
```

```
int fib(int n){
    int *f = new int[n+1];
    f[0]=0;
    f[1] = 1;
    for (int i=2;i<=n; i++)
        f[i] = f[i-1]+f[i-2]
    return f[n]
}
```

0	1	1	2	3	5

改进的递推：节省存储

```
int fib(int n){
    if (n<=1) return n;
    return fib(n-1)+fib(n-2)
}
```

```
int fib(int n){
    int *f = new int[n+1];
    f[0]=0;
    f[1] = 1;
    for (int i=2;i<=n; i++)
        f[i] =  f[i-1]+f[i-2]
    return f[n]
}
```

改进的递推：节省存储

```
int fib(int n){
    if (n<=1) return n;
    return fib(n-1)+fib(n-2)
}
```

```
int fib(int n){
    f1, f2 = 0,1
    for i=2 to n:
        f1,f2 = f2, f1+f2
    return f2
}
```

爬台阶

- 假设你正在爬台阶。需要 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？
- 到达第n阶台阶的总数=到达第n-1阶台阶的总数+到达第n-2阶台阶的总数. 即：

$$F(n) = F(n-1)+ F(n-2)$$

$F(1) = 1$， $F(2) = 2$，

找零钱问题

- 有不同种类的硬币，如何用最少数目硬币组合成指定金额?

1	5	10

15

贪婪法： **10，5**

也是最优解

找零钱问题

- 有不同种类的硬币，如何用最少数目硬币组合成指定金额？

1　　5　　11　　15

11，1，1，1，1

贪婪法失效！

5，5，5

最优解

分治递归

- 多步决策：先选择一枚硬币，再考虑子问题

1　　5　　11

15

15 = 11 + 4

15 = 5 + 10

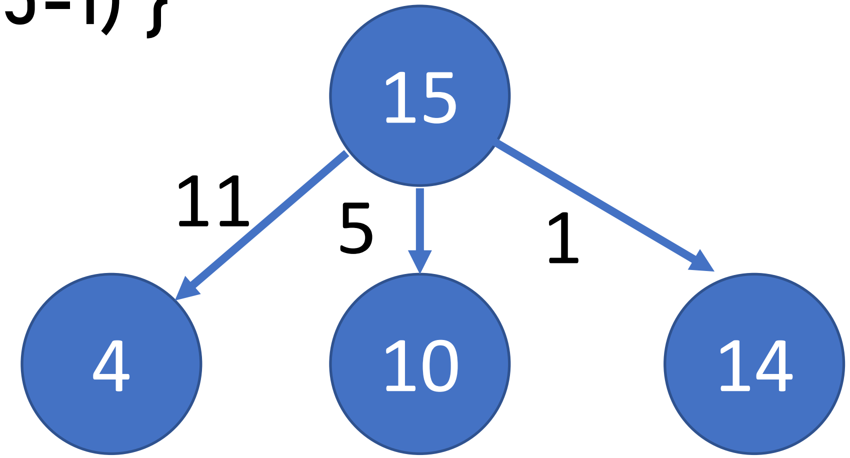
15 = 1 + 14

分治递归

- 设f(e)表示拼出e的最小硬币数目，则

  f(15) = min{ 1+f(15-11), 1+f(15-5), 1+f(15-1) }

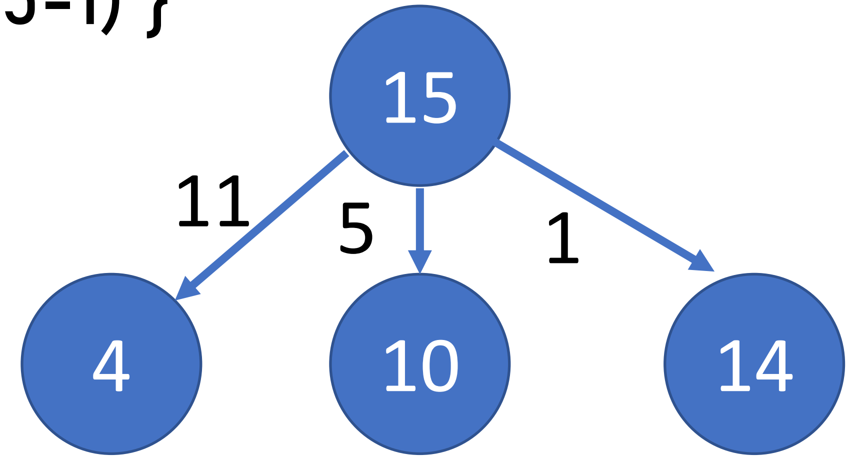
  = min{1+f(4), 1+f(10), 1+f(14)}



分治递归

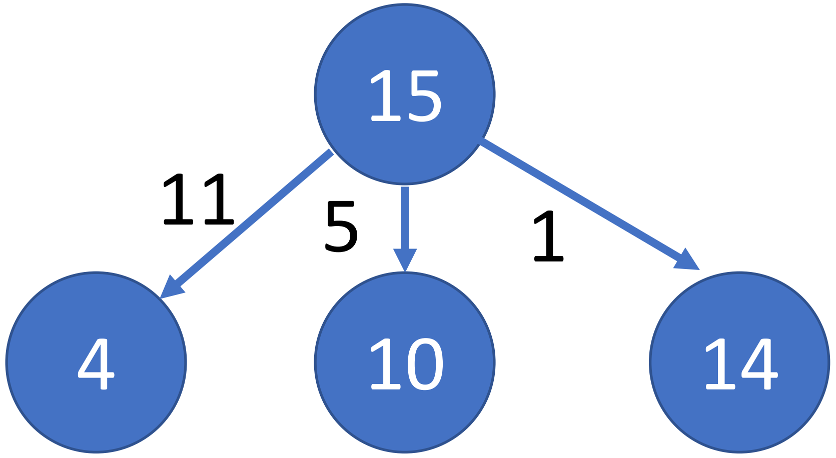
- 设f(e)表示拼出e的最小硬币数目，则

$$f(15) = \min\{ 1+f(15-11), 1+f(15-5), 1+f(15-1) \}$$
$$= \min\{1+f(4), 1+f(10), 1+f(14)\}$$



$$f[e] = \min\{ f[e-a_j]+1 \}$$

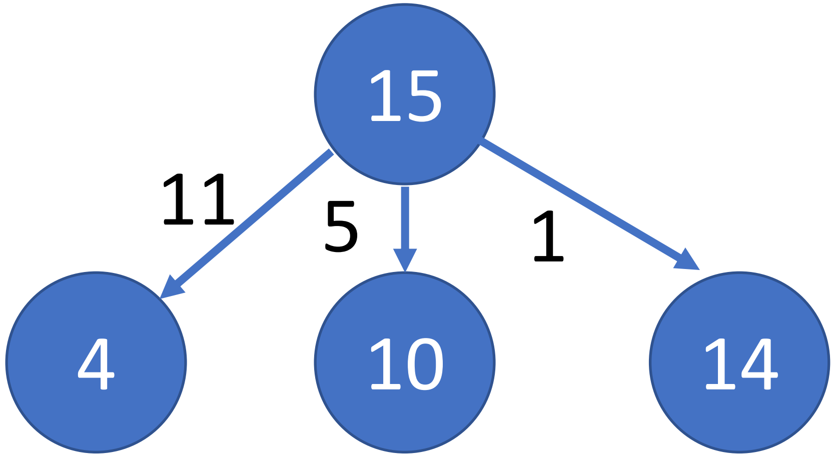
分治递归:递归实现




```
int  coinChange(E, a[],f)
    if E==0: return 0
    if f[E]<0:
        f[E]=infinity
        for j=1 to k:
            if E> a[j]:
                f[E] = min(f[E],coinChange(E-a[j], a, f)+1)
    return f[E]
```

0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

分治递归:递推实现



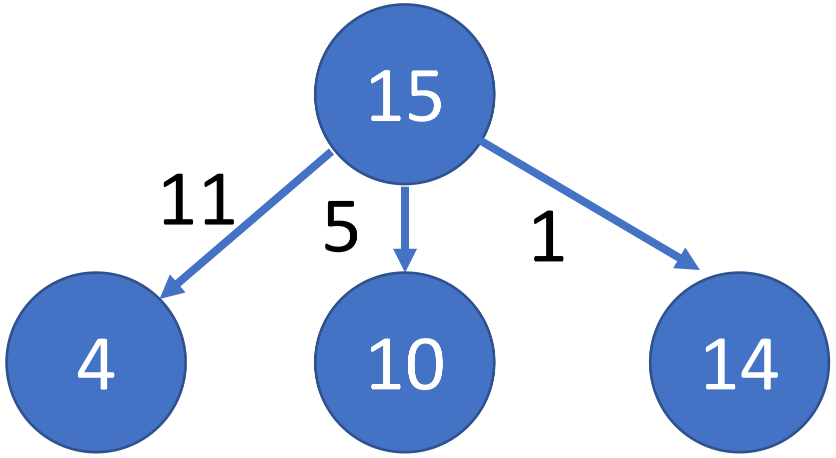

```
int  coinChange(E, a[])
        int  f = new int[E+1]
        f[0] = 0

        for e=1 to E:
                f(e) = infinity
                for j=1 to k:
                    if e>= a_j :
                        f[e] = min(f[e],f[e-a_j]+1)
```

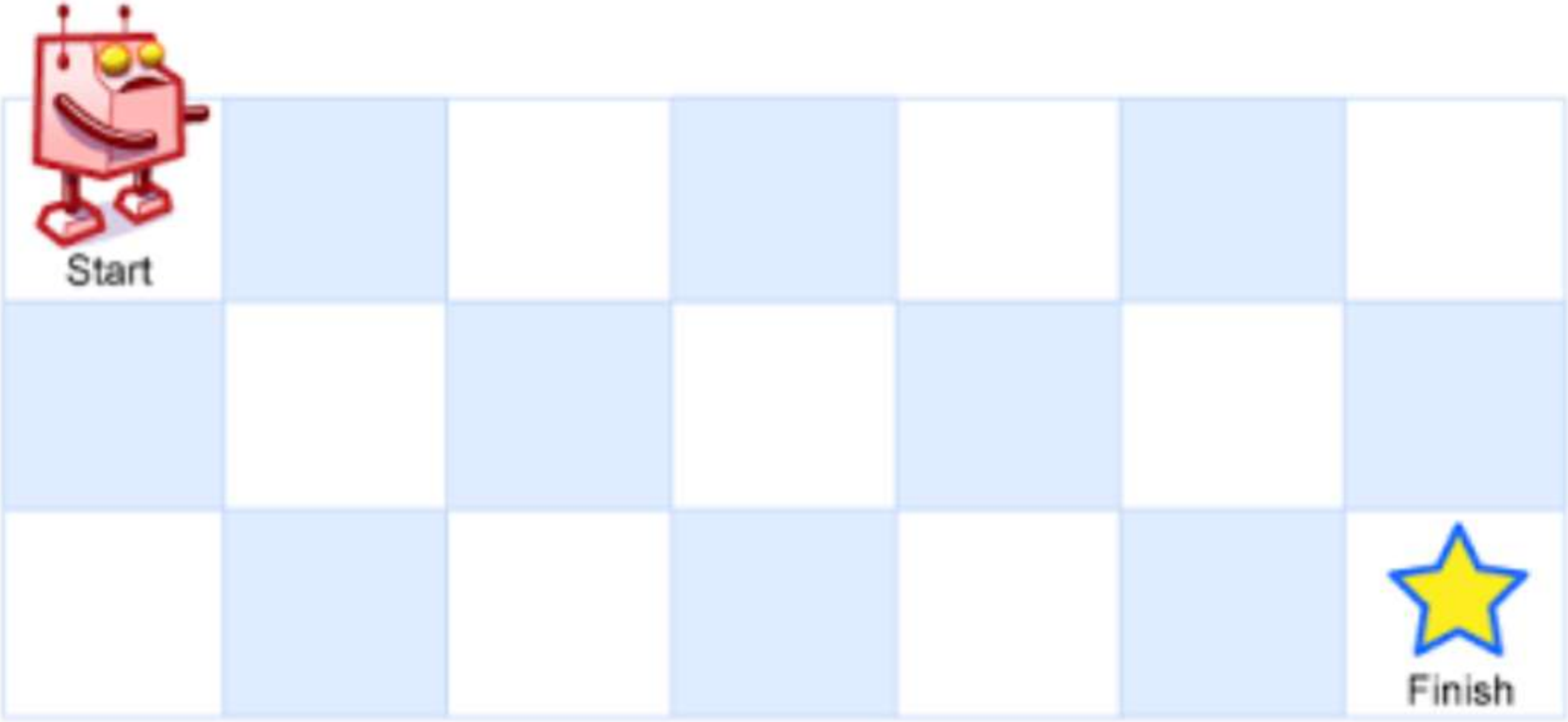
输出解决方案

```
int  coinChange(E, a[])
        int  f = new int[E+1]
        int p = new int[E+1]
        f[0] = 0

        for e=1 to E:
                f(e) = infinity
                for j=1 to k:
                        if e>= a_j :
                                if(f[e-a_j]+1< f[e]):
                                        f[e] = f[e-a_j]+1
                                        p[e] = j
```



机器人多少种不同的走法



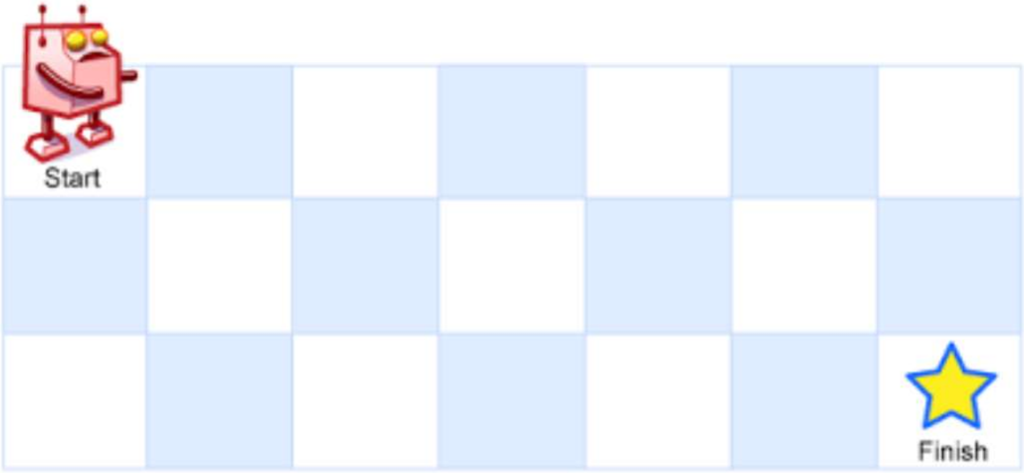

```
Input: m = 3, n = 7
Output: 28
```

62. Unique Paths

A robot is located at the top-left corner of a `m x n` grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?




```
Input: m = 3, n = 7
Output: 28
```

机器人多少种不同的走法 [LeetCode] 62

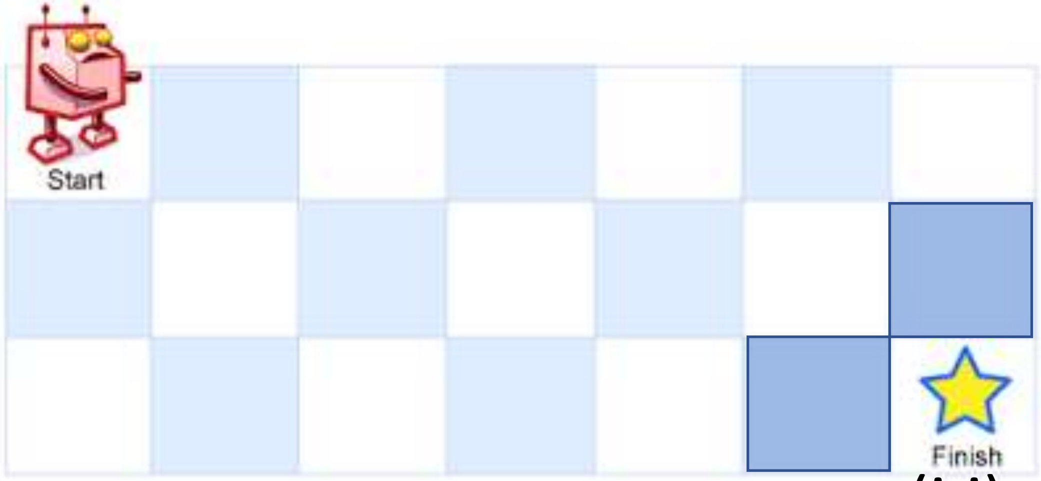
- 数学建模： f(i,j)表示到达(i,j)的不同路径数

f(i,j) = f(i-1,j)+f(i,j-1)

f(1,j)=1 , f(i,1)=1

初始化

```
f(i,j):
    if i==1 or j==1: return 1;
    return f(i-1,j)+f(i,j-1)
```



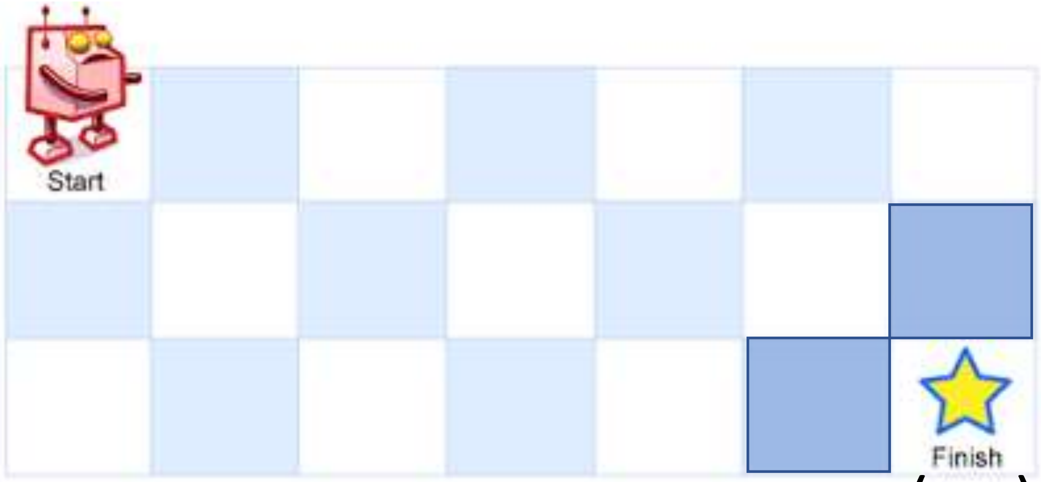
(i,j)

$T(n) = O(2^{m+n})$

递归的动态规划

```
dp(i, j, f):
    if i==1 or j==1: return 1;
    if f[i,j] <0:
        f[i,j] = dp(i-1,j)+dp(i,j-1)

    return f[i,j]
```

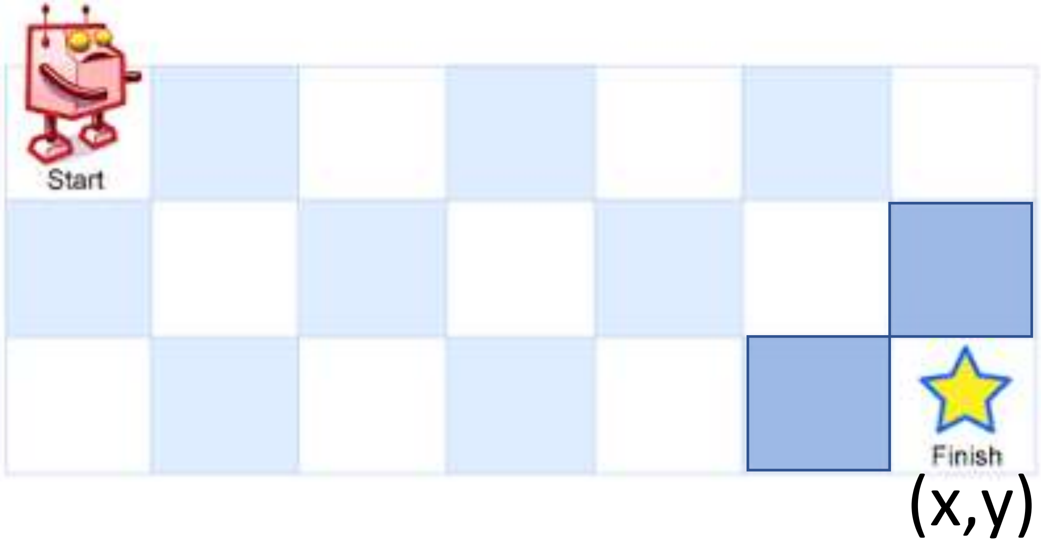


$T(n) = O(mn)$

(x,y)

递推的动态规划

```
dp(m,n):
   分配一个二维数组f[][]
   for j= 1 to n:   f[1,j] = 1
   for i= 1 to m:  f[i,1] = 1
   for i=2 to m
       for j= 2 to n
          f[i,j] = f[i-1,j]+f[i,j-1]
   return f[m,n]
```



$T(n) = O(mn)$

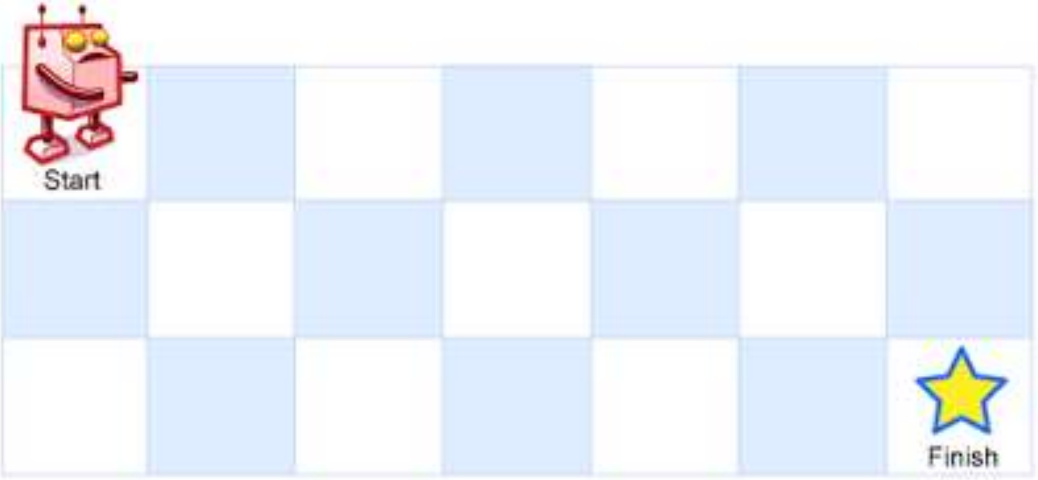
```cpp
int dp(int m, int n) {
    //分配一个二维数组f[][]
    vector<vector<int>> f(m + 1, vector<int>(n + 1, -1));
    for (int j = 1; j <= n; j++) f[1][j] = 1;
    for (int i = 1; i <= m; i++) f[i][1] = 1;

    for (int i = 2; i <= m; i++)
        for (int j = 2; j <= n; j++)
            f[i][j] = f[i - 1][j] + f[i][j - 1];
    return f[m][n];
}
```

(m,n)

```
int uniquePaths(int m, int n) {
    //分配一个二维数组f[][]
    vector<vector<int>> f(m , vector<int>(n , -1));
    for (int j = 0; j < n; j++) f[0][j] = 1;
    for (int i = 0; i < m; i++) f[i][0] = 1;

    for (int i = 1; i < m; i++)
        for (int j = 1; j < n; j++)
            f[i][j] = f[i - 1][j] + f[i][j - 1];
    return f[m-1][n-1];
}
```



$T(n) = O(mn)$

(m-1,n-1)

优化内存

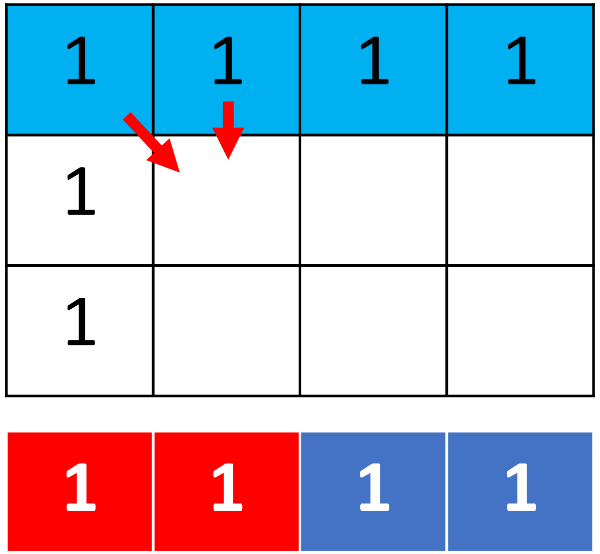
```cpp
int uniquePaths(int m, int n) {
    vector<int> f(n, 1);
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            f[j] += f[j - 1];
        }
    }
    return f[n - 1];
}
```

1	1	1	1
1			
1			

1	1	1	1

优化内存

```cpp
int uniquePaths(int m, int n) {
    vector<int> f(n, 1);
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            f[j] += f[j - 1];
        }
    }
    return f[n - 1];
}
```



优化内存

```
int uniquePaths(int m, int n) {
    vector<int> f(n, 1);
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            f[j] += f[j - 1];
        }
    }
    return f[n - 1];
}
```

1	1	1	1
1	2		
1			

1	2	1	1

优化内存

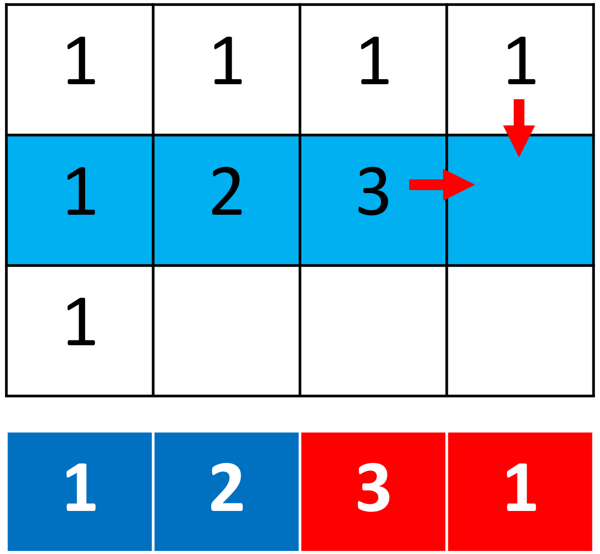
```
int uniquePaths(int m, int n) {
    vector<int> f(n, 1);
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            f[j] += f[j - 1];
        }
    }
    return f[n - 1];
}
```

1	1	1	1
1	2		4
1			

1	2	3	1

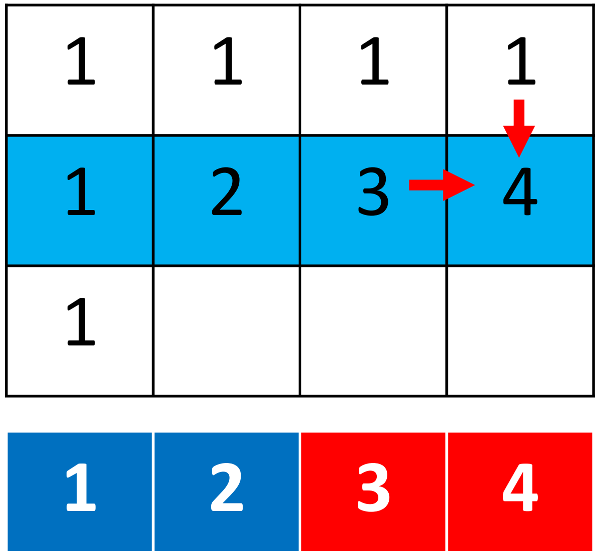
优化内存

```cpp
int uniquePaths(int m, int n) {
    vector<int> f(n, 1);
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            f[j] += f[j - 1];
        }
    }
    return f[n - 1];
}
```



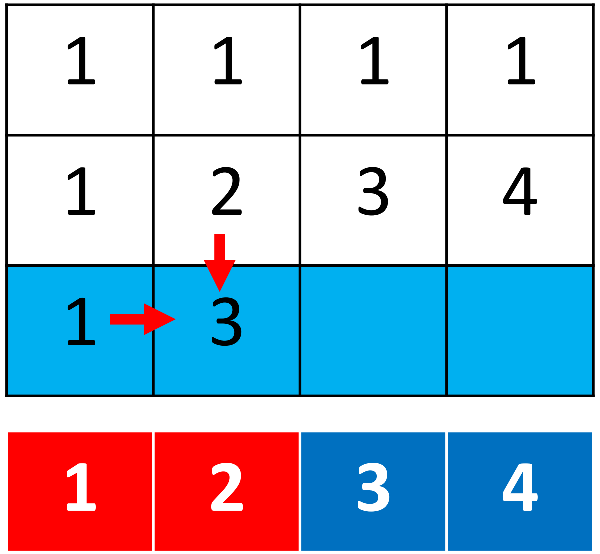
优化内存

```cpp
int uniquePaths(int m, int n) {
    vector<int> f(n, 1);
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            f[j] += f[j - 1];
        }
    }
    return f[n - 1];
}
```



优化内存

```cpp
int uniquePaths(int m, int n) {
    vector<int> f(n, 1);
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            f[j] += f[j - 1];
        }
    }
    return f[n - 1];
}
```



优化内存

```cpp
int uniquePaths(int m, int n) {
    vector<int> f(n, 1);
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            f[j] += f[j - 1];
        }
    }
    return f[n - 1];
}
```

1	1	1	1
1	2	3	4
1	3		

1	3	3	4

优化内存

```
int uniquePaths(int m, int n) {
    vector<int> f(n, 1);
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            f[j] += f[j - 1];
        }
    }
    return f[n - 1];
}
```

1	1	1	1
1	2	3	4
1	3	6	

1	3	3	4

优化内存

```cpp
int uniquePaths(int m, int n) {
    vector<int> f(n, 1);
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            f[j] += f[j - 1];
        }
    }
    return f[n - 1];
}
```

1	1	1	1
1	2	3	4
1	3	6	

1	3	6	4

优化内存

```cpp
int uniquePaths(int m, int n) {
    vector<int> f(n, 1);
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            f[j] += f[j - 1];
        }
    }
    return f[n - 1];
}
```

1	1	1	1
1	2	3	4
1	3	6	10

1	3	6	4

优化内存

```cpp
int uniquePaths(int m, int n) {
    vector<int> f(n, 1);
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            f[j] += f[j - 1];
        }
    }
    return f[n - 1];
}
```

1	1	1	1
1	2	3	4
1	3	6	10

滚动数组

1	3	6	10

动态规划 = 分治递归+记忆

- 可以求解问题规模是整数的**数值计算**问题。

➢计数问题：有多少

➢优化问题：最大小值

➢存在问题：值是1或0

动态规划 = 分治递归+记忆

状态：一定规模的问题

- 分治递归：

➢定义问题的数学模型，如f(n)

➢大问题分解成小问题： $f(n) = f(n-1)+f(n-2)$

$f(n) = \min\{ f(n-aj)+c, j=1,2,\ldots k\}$

➢递归实现（自顶向下）、递推实现（自底向下）

➢初始化、基情形

状态转移方程：递归方程

50年代初美国数学家R.E.Bellman

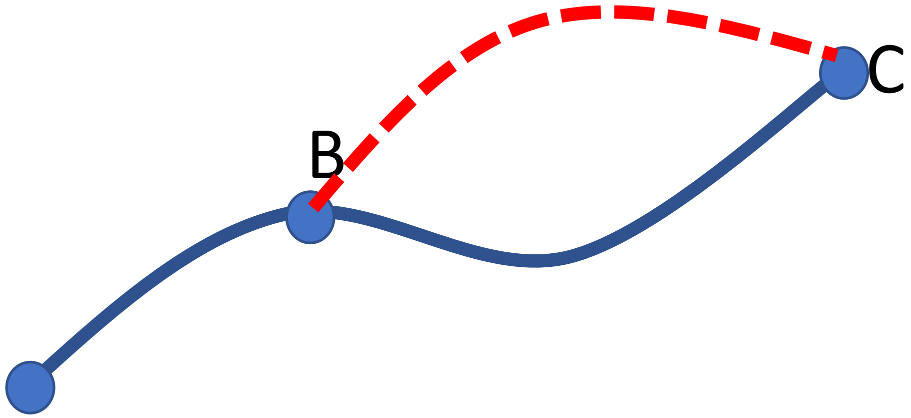
可以用动态规划求解的问题的2个特征

- **重叠子问题**：问题可以分解成更小的子问题，这些子问题的解在更大问题的解中多次重复使用。

- **最优子结构**：通过组合子问题的最优解可以得到问题的最优解。

最优子结构

- 如果问题的最优解包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理



A到C的最优价种包含了B到C的最优解

背包问题

Youtube频道： hwdong

博客： hwdong-net.github.io

背包问题

- **n个物品的重量w_1、 w_2 、…、 w_n，价值为v_1、 v_2 、…、v_n，背包限重为W。**问：如何装使价值最大？

item	weight	value
1	1	1
2	2	5
3	5	18
4	6	22
5	7	28

W = 11

按性价比w_2 / v_2排序

背包问题

- **n**个物品的重量w_1、 w_2 、…、 w_n， 价值为v_1、 v_2 、…、v_n， 背包限重为**W**。问：如何装使价值最大？

item	weight	value
1	1	1
2	2	5
3	5	18
4	6	22
5	7	28

按性价比w_2 / v_2排序

W = 11

贪婪法

(7,28), (2,5), (1,1)

最优解

(5， 18), (6,22)

分治递归

- 定义目标函数: **F(i, w)表示i个物品背包限重为w。**
- 分治递归，考虑第i个物品:

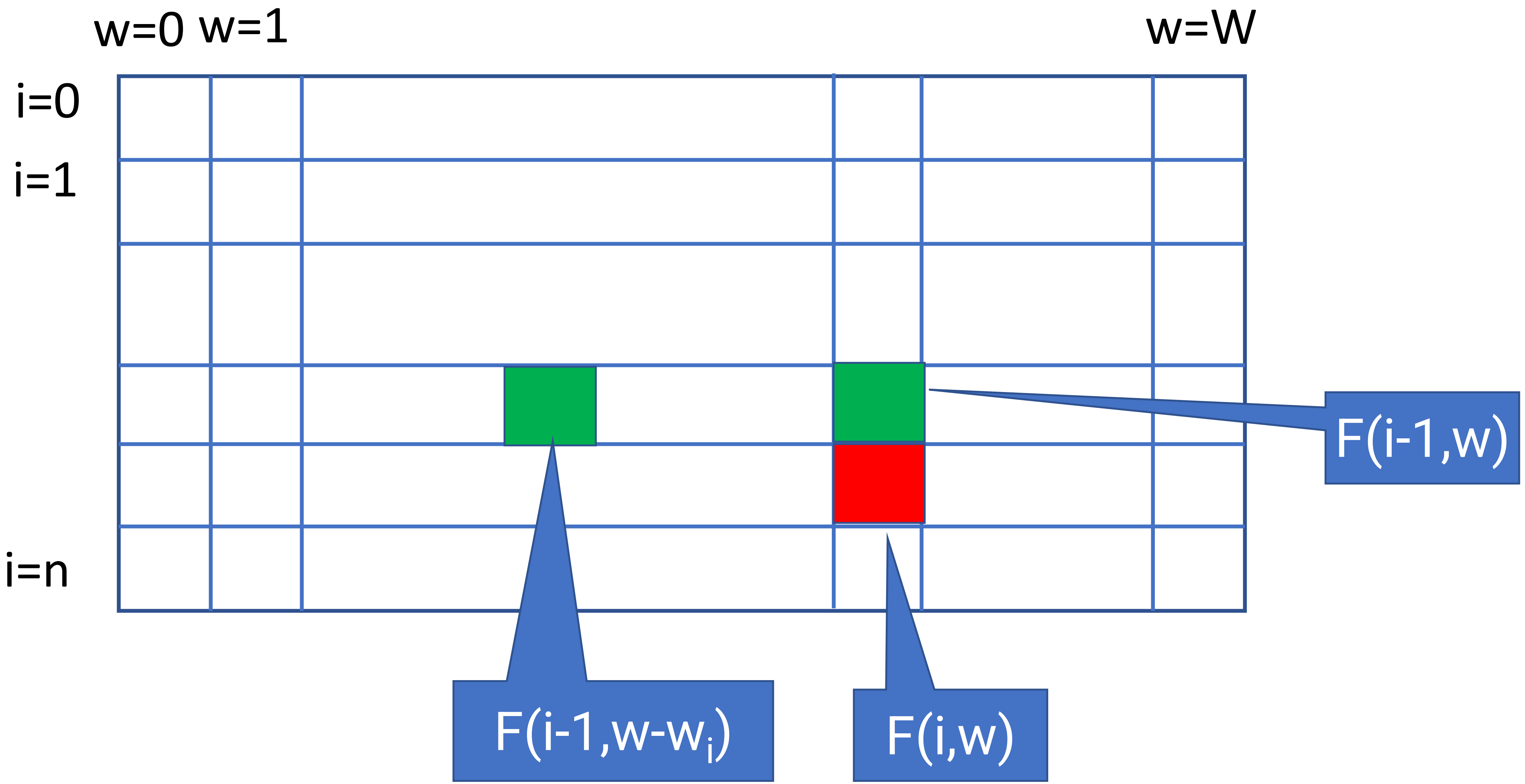
$$F(i,w) = \begin{cases} F(i-1,w) & \text{当 } w< w_i \\ \max\{F(i-1,w)\ ,\ v_i + F(i-1,w-w_i)\} & \text{当 } w>= w_i \end{cases}$$

基情形:

$F(0,w) = 0$，$F(i,0) = 0$

- F(i,w)可以用一个二维数组存储

- F(i,w)可以用一个二维数组存储



递归的动态规划

```
Recursive_DP(i, W,w[], v[], f):
    if i==0 or W==0: return 0;
    if F[i,W]<0:
        if W<w[i]
            return Recursive_DP(i, W,w,v,f)
        else
            return max(Recursive_DP(i-1, W,w,v,f),
                       v[i]+Recursive_DP(i-1, W-w[i],w,v,f))
```

递推的动态规划

```
iterative_DP(W,w[], v[],n ):
    F = new int[n][W]
    for w =0 to W:   F[0,w] = 0
    for i =0 to n:   F[i,0] = 0
    for i=1 to n:
        for WW = 1 to W:
            if w_i>WW:
                F[i,WW] = F[i-1,WW]
            else:
                F[i,WW] = max(F[i-1,WW], v_i +F[i-1,WW- w_i])
    return F[n,W]
```

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

i	w	v
1	2	12
2	1	10
3	3	20
4	2	15

$W = 5$

```
for i=1 to n:
    for w = 1 to W:
        if w_i>W:
            F[i,w] = F[i-1,w]
        else:
            F[i,w] = max(F[i-1,w], v_i +F[i-1,w- w_i])
```

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

i	w	v
1	2	12
2	1	10
3	3	20
4	2	15

W = 5

```
for i=1 to n:
    for w = 1 to W:
        if w_i>W:
            F[i,w] = F[i-1,w]
        else:
            F[i,w] = max(F[i-1,w], v_i +F[i-1,w- w_i])
```

F[1,1] = F[0,1]

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12			
2	0					
3	0					
4	0					

i	w	v
1	2	12
2	1	10
3	3	20
4	2	15

W = 5

```
for i=1 to n:
    for w = 1 to W:
        if wi>W:
            F[i,w] = F[i-1,w]
        else:
            F[i,w] = max(F[i-1,w], vi +F[i-1,w- wi])
```

$F[1,2] = \max(F[0,2], 12+F[0,2-2])$
$= \max(0,12)=12$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12		
2	0					
3	0					
4	0					

i	w	v
1	2	12
2	1	10
3	3	20
4	2	15

W = 5

```
for i=1 to n:
    for w = 1 to W:
        if w_i>W:
            F[i,w] = F[i-1,w]
        else:
            F[i,w] = max(F[i-1,w], v_i +F[i-1,w- w_i])
```

$F[1,3] = \max(F[0,3], 12+F[0,2-2])$
$= \max(0,12)=12$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0					
3	0					
4	0					

i	w	v
1	2	12
2	1	10
3	3	20
4	2	15

$W = 5$

```
for i=1 to n:
    for w = 1 to W:
        if wi>W:
            F[i,w] = F[i-1,w]
        else:
            F[i,w] = max(F[i-1,w], vi +F[i-1,w- wi])
```

$F[1,5] = \max(F[0,5], 12+F[0,2-2])$
$= \max(0,12)=12$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10				
3	0					
4	0					

i	w	v
1	2	12
2	1	10
3	3	20
4	2	15

W = 5

```
for i=1 to n:
    for w = 1 to W:
        if w_i>W:
            F[i,w] = F[i-1,w]
        else:
            F[i,w] = max(F[i-1,w], v_i +F[i-1,w- w_i])
```

$F[2,1] = \max(F[1,1], 10+F[0,1-1])$
$= \max(0,10)=10$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12			
3	0					
4	0					

i	w	v
1	2	12
2	1	10
3	3	20
4	2	15

W = 5

```
for i=1 to n:
    for w = 1 to W:
        if w_i>W:
            F[i,w] = F[i-1,w]
        else:
            F[i,w] = max(F[i-1,w], v_i +F[i-1,w- w_i])
```

$$F[2,2] = \max(F[1,2],10+F[1,2-1]) = \max(12,10+0)=12$$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22		
3	0					
4	0					

i	w	v
1	2	12
2	1	10
3	3	20
4	2	15

W = 5

for i=1 to n:

    for w = 1 to W:

        if w_i>W:

            F[i,w] = F[i-1,w]

        else:

            F[i,w] = max(F[i-1,w], v_i +F[i-1,w- w_i])

F[2,3] = max(F[1,3],10+F[1,3-1])

= max(12,10+12)=22

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

i	w	v
1	2	12
2	1	10
3	3	20
4	2	15

W = 5

```
for i=1 to n:
    for w = 1 to W:
        if w_i>W:
            F[i,w] = F[i-1,w]
        else:
            F[i,w] = max(F[i-1,w], v_i +F[i-1,w- w_i])
```

$F[4,5] = \max(F[3,5],15+F[3,5-2])$

$= \max(32,15+22)=37$

时间复杂度: Θ(nW)

```
iterative_DP(W,w[], v[] ):
    F = new int[n][W]
    for w =0 to W:   F[0,w] = 0
    for i =0 to n:   F[i,0] = 0
    for i=1 to n:
        for w = 1 to W:
            if w_i>W:
                F[i,w] = F[i-1,w]
            else:
                F[i,w] = max(F[i-1,w], v_i +F[i-1,w- w_i])
    return F[n,W]
```

输出解决方案

- 1）修改代码，增加一个path二维数组
- 2）从最后的F[n,W]逆向回退

```
i = n; Wt =  W
while i>1:
  if F[i,Wt]== F[i-1,Wt]:
    path[i] = false;
  else
    path[i] = true;
    i--;  Wt-=w[i]
```

最大公共子序列

Longest Common Subsequence (LCS)

Youtube频道：hwdong
博客：hwdong-net.github.io

子序列

- **子序列**： 原序列一些元素按照在原序列中的先后次序排列构成的序列。

  或者：原序列删除一些元素后剩余元素组成的序列。

- 如原始序列： 5 2 8 6 3 6 9 7
- 子序列如：

  2 3 7

  5 6 3 9

  ~~3 2 7~~

- 原始序列为$(a_1,a_2,\ldots,a_n)$，子序列是按序号依次抽取的元素构成的序列$(a_{i1},a_{i2},\ldots,a_{ik})$，其中 $1\le i1 < i2<\ldots<ik \le n$

最大公共子序列

- 是一个在一个序列集合中（通常为两个序列）用来查找所有序列中最长相同子序列的问题。这与查找最长公共子串的问题不同的地方是：子序列不需要在原序列中占用连续的位置 。
- 最长公共子序列问题是一个经典的计算机科学问题，也是数据比较程序，比如Diff工具，和生物信息学应用的基础。它也被广泛地应用在版本控制，比如Git用来调和文件之间的改变。

最大公共子序列

- 问题：给定2个序列，求出它们的最大公共子序列的长度。

  输入：text1 = "abcde", text2 = "ace"
  输出：3
  解释：最长公共子序列是 "ace"，它的长度为 3 。

动态规划

- 目标函数：**d(i,j)**表示序列$x_1x_2 \ldots x_i$和序列$y_1y_2, \ldots y_j$的最大公共子序列的长度。
- 分治递归：

$$d(i, j) = \begin{cases} d(i-1, j-1) + 1 & \text{if } x_i = y_j \\ \max(d(i, j-1),\ \ d(i-1, j)) & \text{if } x_i \,!= y_j \end{cases}$$

- 递推计算:

```
for i = 1 to m:
    for j = 1 to n:
        if xi==yj:  d(i, j) = d(i-1, j-1)+1
        else: d(i, j) = max(d(i, j-1), d(i-1, j))
```

- 初始化：**d(i,0) = 0　d(0,j) = 0**

序列比对

Sequence alignment

Youtube频道： hwdong

博客： hwdong-net.github.io

序列比对

- 序列比对是一种用于鉴定两个或多个生物序列（如DNA、RNA或蛋白质序列）之间相似程度的方法。
- 序列比对的目标是确定序列之间的相关程度，这可以提供有关其进化历史、功能和结构的有价值信息。
- 序列比对在生物信息学的许多领域都很重要，例如在进化研究中，它可以帮助识别不同物种之间的异同。它还用于分子生物学领域，它可以帮助识别基因和蛋白质的功能区域，以及药物设计，它可以帮助识别新药的潜在靶点。

序列比对

- DNA序列是由氨基酸残基A,T,C,G构成。
- 两个 DNA 序列：

  ATGCTAGTGGT

  ATGCTAGTGGG
- 两个蛋白质序列：

  RAVKQVPTY

  KAVKQVPTY。

多序列比对

- 三个 DNA 序列：

  ATGCTAGTGGT

  ATGCTAGTGGG

  ATGCTAGTGGC。

序列比对

- 序列比对中，错配与突变相应，而空位与插入或缺失对应。

A	C	T	A	C	C
A		T	A	C	G

空位　　错配

序列比对

- 序列比对中，错配与突变相应，而空位与插入或缺失对应。
  - Align two sequences of nucleotides

    AGGCTATCACCTGACCTCCAGGCCGATGCCC
    TAGCTATCACGACCGCGGTCGATTTGCCCGAC

  - Resulting alignment:

    -AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
    TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC

序列比对

- 序列排列也用于非生物序列，如2个字符串的相似性。
- 例如计算自然语言或金融数据中字符串之间的距离成本。Unix的diff命令、论文相似性等。

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6个mismatch, 1个gap

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1个mismatch, 1个gap

哪种更好?

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0个mismatch, 3个gap

Edit Distance

- Levenshtein 1996, Needleman & Wunsch 1997
- 两个字符串间的最小**编辑距离**（Edit distance）定义将一个字符串转化为另一个字符串所需的编辑操作的最小数目（代价)。

  insert

  delete

  substitution

Edit Distance

- 如果每个操作代价一样，则ED值为5
- 如果substitution(替换)代价为2，则ED值为8

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

序列比对

- 给定字符串$x_1x_2 \ldots x_m$和$y_1y_2 \ldots y_n$， 问题规模为(m, n).
- 参数化问题规模，定义目标函数：d(i,j) 表示长度为i和j的2个字符串的编辑距离。 d(i,j) : $x_1x_2 \ldots x_i$和$y_1y_2 \ldots y_j$的编辑距离
- 分治递归：

  当$x_i = y_j$ 时, d(i,j) = d(i-1,j-2)

  当x_i != y_j 时，有插入、删除、替换3个操作使得最后一个匹配

序列比对

- 分治递归：

  当$x_i = y_j$时, d(i,j) = d(i-1,j-2)

  当x_i != y_j时，有插入、删除、替换3个操作使得最后一个匹配

  x最后插入y_j：

$$\begin{array}{lllll} x_1 & x_2 & \dots & x_i & \_ \\ y_1 & y_2 & \dots & & y_j \end{array}$$

$$d(i, j) = d(i, j-1) + 1$$

序列比对

- 分治递归：

  当x_i = y_j 时, d(i,j) = d(i-1,j-2)

  当x_i != y_j 时，有插入、删除、替换3个操作使得最后一个匹配

  x最后x_i 删除：

$$
\begin{array}{llll}
x_1 & x_2 & \ldots & x_i \\
y_1 & y_2 & \ldots\ y_j & \_
\end{array}
$$

$$d(i, j) = d(i\text{-}1, j) + 1$$

序列比对

- 分治递归：

  当$x_i = y_j$时, $d(i,j) = d(i-1,j-2)$

  当$x_i != y_j$时，有插入、删除、替换3个操作使得最后一个匹配

  x_i替换y_j：

  $x_1 \ x_2 \ \ldots \quad x_i$

  $y_1 \ y_2 \ \ldots \ y_j \ \_$

  $$d(i, j) = d(i-1, j-1) + 2$$

序列比对

- 给定字符串$x_1x_2…x_m$和$x_1x_2…x_n$， 问题规模为(m,n).
- 参数化问题规模，定义目标函数：d(i,j) 表示长度为i和j的2个字符串的编辑距离。
- 分治递归：

$$
d[i,j] = \begin{cases} d[i-1,j-1] \quad if\ x_i = y_i & \text{Match} \\ min\begin{cases} d[i-1,j]+1 \quad if\ x_i \neq y_i & \text{Deletion from X} \\ d[i,j-1]+1 \quad if\ x_i \neq y_i & \text{Insertion to X} \\ d[i-1,j-1]+2 \quad if\ x_i \neq y_i & \text{Substitution} \end{cases} \end{cases}
$$

序列比对算法：递推的动态规划

- 分配二维数组 int d[][] = new int[m+1][n+1]
- 初始化： d[i,0] = i*delete_cost   d[0,j] = j*insert_cost
- 递推计算：

```
for i=1 to m:
    for j= 1 to n:
        if x_i= y_j:  d[i,j] = d[i-1,j-1]
        else:
            d[i,j] = min{d[i-1,j]+1, d[i,j-1]+1, d[i-1,j-1]+2}
```

- 返回d[m,n]

时间复杂度：$\Theta(mn)$

例子

X = ATCGTT
Y = AGTTAC

$$d[i,j] = \begin{cases} d[i-1,j-1] & if\ x_i = y_i \\ min\begin{cases} d[i-1,j]+1 & if\ x_i \neq y_i \\ d[i,j-1]+1 & if\ x_i \neq y_i \\ d[i-1,j-1]+2 & if\ x_i \neq y_i \end{cases} \end{cases}$$

		A	G	T	T	A	C
	0	1	2	3	4	5	6
A	1						
T	2						
C	3						
G	4						
T	5						
T	6						

例子

X = ATCGTT

Y = AGTTAC

$$d[i,j] = \begin{cases} d[i-1,j-1] & if\ x_i = y_i \\ \min \begin{cases} d[i-1,j]+1 & if\ x_i \neq y_i \\ d[i,j-1]+1 & if\ x_i \neq y_i \\ d[i-1,j-1]+2 & if\ x_i \neq y_i \end{cases} \end{cases}$$

		A	G	T	T	A	C
	0	1	2	3	4	5	6
A	1	0					
T	2						
C	3						
G	4						
T	5						
T	6						

d[1,1] = d[0,0]

例子

X = ATCGTT
Y = AGTTAC

$$d[i,j] = \begin{cases} d[i-1,j-1] & if\ x_i = y_i \\ min\begin{cases} d[i-1,j]+1 & if\ x_i \neq y_i \\ d[i,j-1]+1 & if\ x_i \neq y_i \\ d[i-1,j-1]+2 & if\ x_i \neq y_i \end{cases} \end{cases}$$

		A	G	T	T	A	C
	0	1	2	3	4	5	6
A	1	0	1				
T	2						
C	3						
G	4						
T	5						
T	6						

d[1,2] =min(d[0,2]+1,
d[1,1]+1,
d[0,1]+2)
= min(3,1,3) = 1

例子

X = ATCGTT
Y = AGTTAC

$$d[i,j] = \begin{cases} d[i-1,j-1] & if\ x_i = y_i \\ min \begin{cases} d[i-1,j]+1 & if\ x_i \neq y_i \\ d[i,j-1]+1 & if\ x_i \neq y_i \\ d[i-1,j-1]+2 & if\ x_i \neq y_i \end{cases} \end{cases}$$

		A	G	T	T	A	C
	0	1	2	3	4	5	6
A	1	0	1	2			
T	2						
C	3						
G	4						
T	5						
T	6						

d[1,3] =min(d[0,3]+1,
d[1,2]+1,
d[0,2]+2)
= min(4,2,4) = 2

最大子段和

Maximum Subarray

Youtube频道：hwdong

博客：hwdong-net.github.io

最大和数组(最大子段和)

- 最大和子数组(最大子段和)问题是寻找一个具有最大和的连续子数组。
- 如一维子数组A[1..n]，寻找索引i和j (1≤i≤j≤n)，使得：
- 最大 $$\sum_{x=i}^{j} A[x]$$

最大子段和

- 给定n个数(可以是负数)的序列，求该序列连续的子段和的最大值。

输入：nums = [-2,1,-3,4,-1,2,1,-5,4]
输出：6
解释：连续子数组 [4,-1,2,1] 的和最大，为 6 。

回顾：蛮力法

- 穷举所有可能的(i,j)对应的子数组之和$a_i + a_{i+1} + \ldots + a_j$，看看哪个最大？

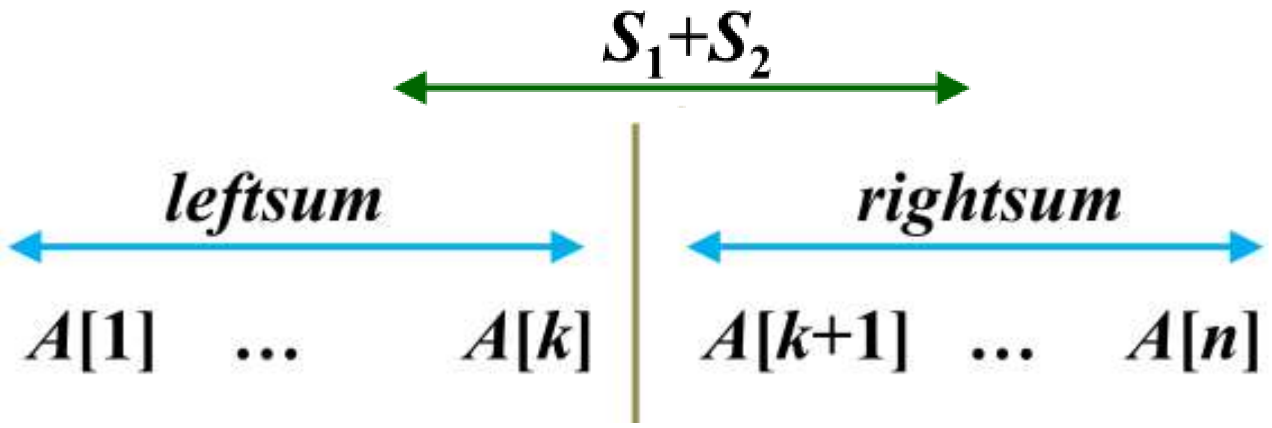
```
sum = -infinity
for i=1 to n:
    sum_ij=0
    for j=1 to n:
        sum_ij = sum_ij+ a_j
        if sum_ij>sum:
            sum = sum_ij
```

$T(n) = \Theta(n^2)$

回顾：分治法

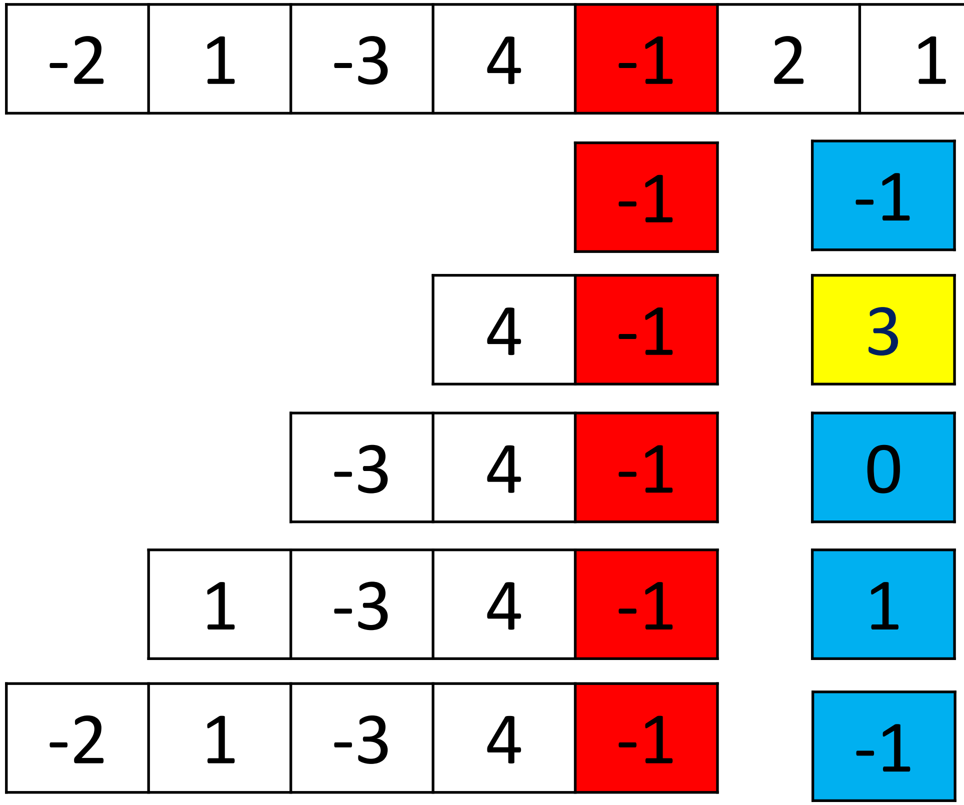
- 将序列分为一分为二
- 递归计算左、右子序列的最大子段和
- 跨越2子序列的最大子段和=S_1+S_2.
- max_sum = max(leftsum, rightsum, S_1+S_2)

$T(n) = \Theta(n\log n)$



动态规划:Kadane's algorithm

- 目标函数：S(i)表示数组A[1…i]中以A[i]结尾的最大子数组和。



动态规划:Kadane's algorithm

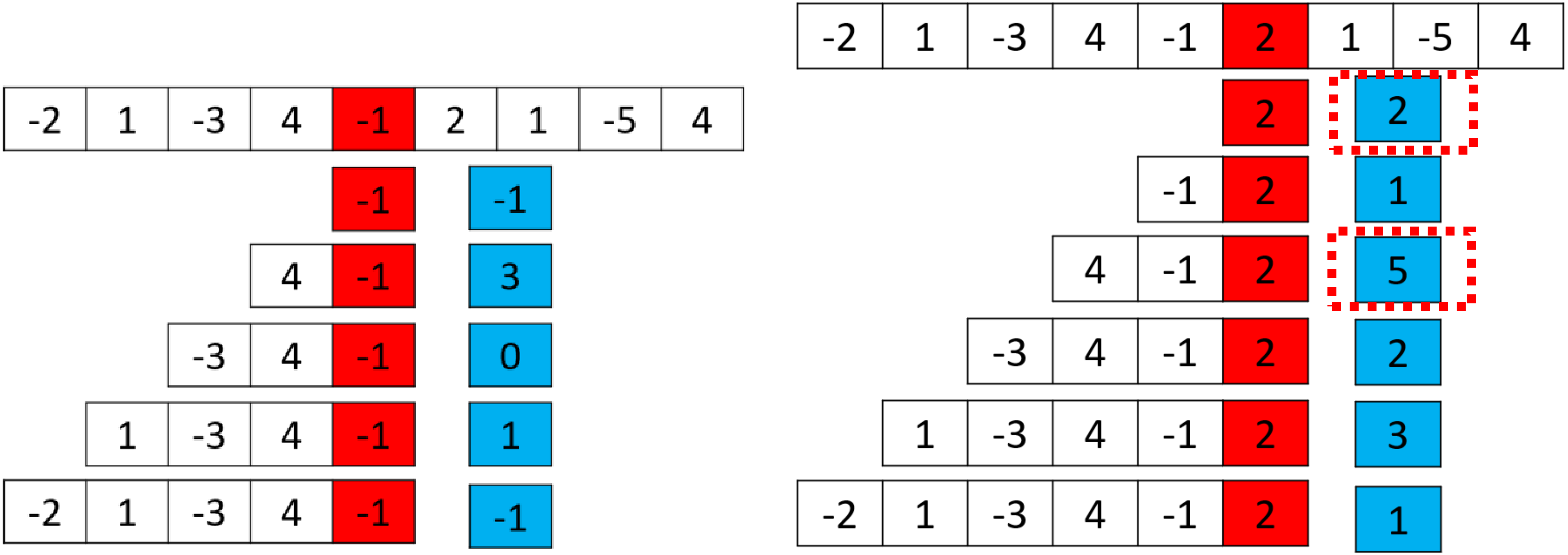
- 目标函数： S(i)表示数组A[1…i]中以A[i]结尾的最大子数组和。
- 分治递归： S[i] = max(S[i-1]+A[i], A[i])

-2	1	-3	4	-1	2	1	-5	4

-2	1	-3	4	-1	2	1	-5	4

动态规划:Kadane's algorithm

- 目标函数： S(i)表示数组A[1…i]中以A[i]结尾的最大子数组和。
- 分治递归： S[i] = max(S[i-1]+A[i], A[i])



动态规划:Kadane's algorithm

- 目标函数：S(i)表示数组A[1…i]中以A[i]结尾的最大子数组和。
- 分治递归：S[i] = max(S[i-1]+A[i], A[i])
- 递推计算S[i]:

```
for i=2 to n:
    S[i] = max(S[i-1]+Ai, Ai)
```

动态规划:Kadane's algorithm

- 目标函数：S(i)表示数组A[1…i]中以A[i]结尾的最大子数组和。
- 分治递归：S[i] = max(S[i-1]+A[i], A[i])
- 递推计算S[i]:

```
for i=2 to n:
    S[i] = max(S[i-1]+Ai, Ai)
```

- 初始条件：S[1] = A_1

动态规划:Kadane's algorithm

- 目标函数：S(i)表示数组A[1...i]中以A[i]结尾的最大子数组和。
- 分治递归：S[i] = max(S[i-1]+A[i], A[i])
- 递推计算S[i]:

  for i=2 to n:

  S[i] = max(S[i-1]+Ai, Ai)

- 初始条件：S[1] = A_1
- A[1...n]的最大子段和是所有S[1],S[2],...S[n]的最大值。

最长递增子序列

longest increasing subsequence

Youtube频道： hwdong

博客： hwdong-net.github.io

子序列

- **子序列**：　原序列一些元素按照在原序列中的先后次序排列构成的序列。

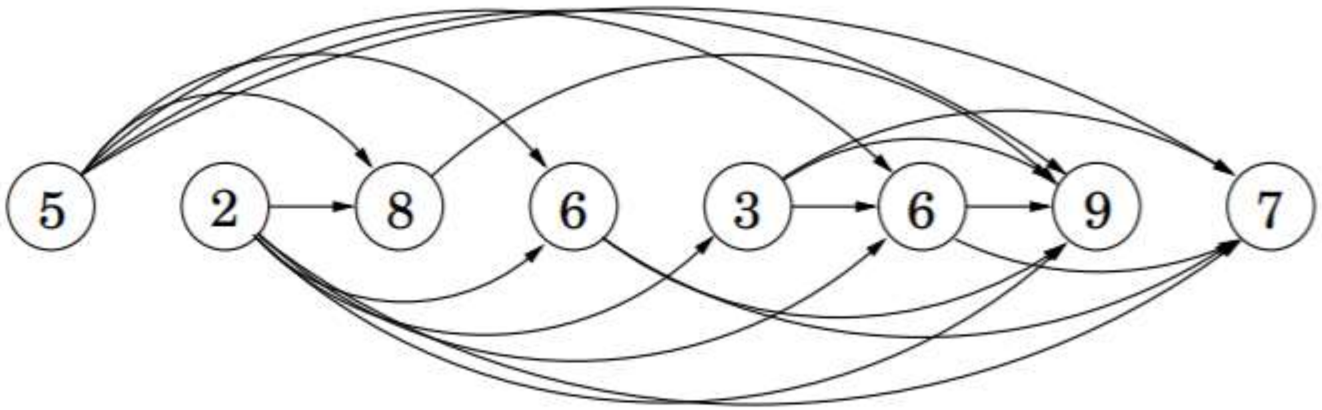
　　或者：原序列删除一些元素后剩余元素组成的序列。
- 如原始序列：　5 2 8 6 3 6 9 7
- 子序列如：

　2 3 7

　5 6 3 9
- 原始序列为$(a_1,a_2,\ldots,a_n)$，子序列是按序号依次抽取的元素构成的序列$(a_{i1},a_{i2},\ldots,a_{ik})$，其中 $1\le i1 < i2<\ldots<ik \le n$

最长递增子序列

- 在一个给定的数值**序列**中，找到一个**子序列**，使得这个**子序列**元素的数值依次递增，并且这个**子序列**的长度尽可能地大。
- 如原始序列：　5 2 8 6 3 6 9 7
- 最长递增子序列为：**2 3 6 9**



最长递增子序列

- 问题：给你一个序列，找到其中最长严格递增子序列的长度。

  输入：[5,2, 8,6,3, 6,9,7]，输出：4

  输入：[3,3,3,3,3]，输出：1

动态规划

- 目标函数： **d[i]** 表示以A_i 结尾的最长严格递增子序列的长度。
- 分治递归： $A_1, A_2, \dots, A_j, \dots, A_i$

$$d[i] = \max\{\, d[j]+1, \quad \text{if } A_j < A_i \}$$

- 递推计算： for i =2 to n:

  for j=1 to i-1:

  if $A_j < A_i$:

  d[i] = max(d[i], d[j]+1)

- 初始化：

  d[1]=1

- 所有d[i]的最大值就是最大递增子序列的长度

矩阵链乘法

chain matrix multiplication

Youtube频道： hwdong

博客： hwdong-net.github.io

矩阵链乘法

- 设有四个矩阵A, B, C, D，它们的维数分别是: A=50×20, B=20×1, C=1×10, D=10×100
- 四个矩阵相乘A×B×C×D，可以迭代的两两相乘：
  (((A×B)×C)×D)
- 矩阵乘法满足结合律：(A×B)×C) = (A×(B×C))
- 不同的组合，计算的代价是不同的：

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	$7,000$

矩阵链乘法

- 如果要计算矩阵乘积$A_1 \times A_2 \times \ldots \times A_n$， 如何确定最优的计算次序（加括号方式）？

穷举法

- 列举出所有可能的计算次序，并计算出每一种计算次序相应需要的数乘次数，从中找出一种Flops乘法次数最少的计算次序。
- 对于n个矩阵的连乘积，设其不同的计算次序为P(n)。
- 由于每种加括号方式都可以分解为两个子矩阵的加括号问题：$(A_1 \dots A_k)(A_{k+1} \dots A_n)$，由此可以得到关于P(n)的递推式如下：

$$P(n)=\begin{cases}1 & \text{if } n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & \text{if } n>1\end{cases}$$

- P(n)=C(n-1)

$$C(n)=\underbrace{\frac{1}{n+1}\binom{2n}{n}}_{\text{Catalan数}}=\underbrace{\Omega\left(\frac{4^n}{n^{3/2}}\right)}_{\text{Stirling公式}}$$

catalan-numbers

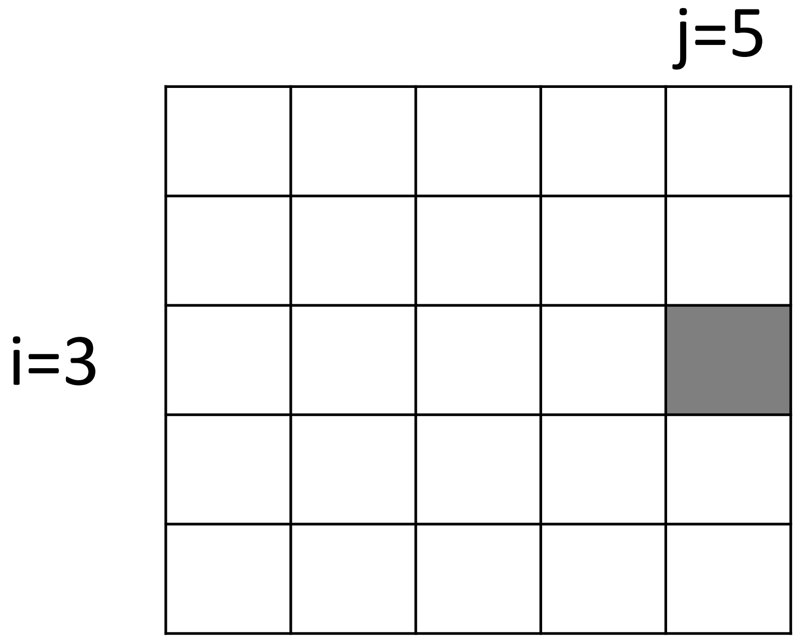
- n个右括号和n个左括号的合法的括号表达式的数目称为catalan-numbers，记为C_n。
- 如C_n=5表示3对括号有5种合法的表达式
    - ()()()
    - (())()
    - ()(())
    - ((()))
    - (()())

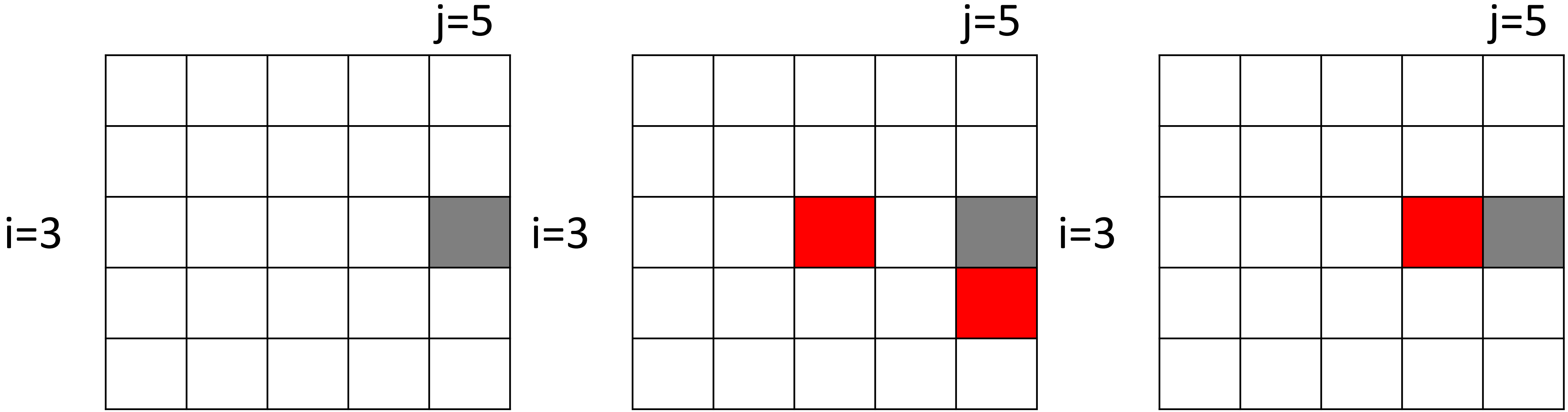
动态规划

- 目标函数：设C(i,j)表示$A_i \times A_{i+1} \times \ldots \times A_j$的最小代价.
- 分治递归： $A_i \times A_{i+1} \times \ldots \times A_j$可能分解为$A_i \times A_{i+1} \times \ldots \times A_k$和$A_{k+1} \times A_{k+2} \times \ldots \times A_j$，其中，$i \le k \le j$

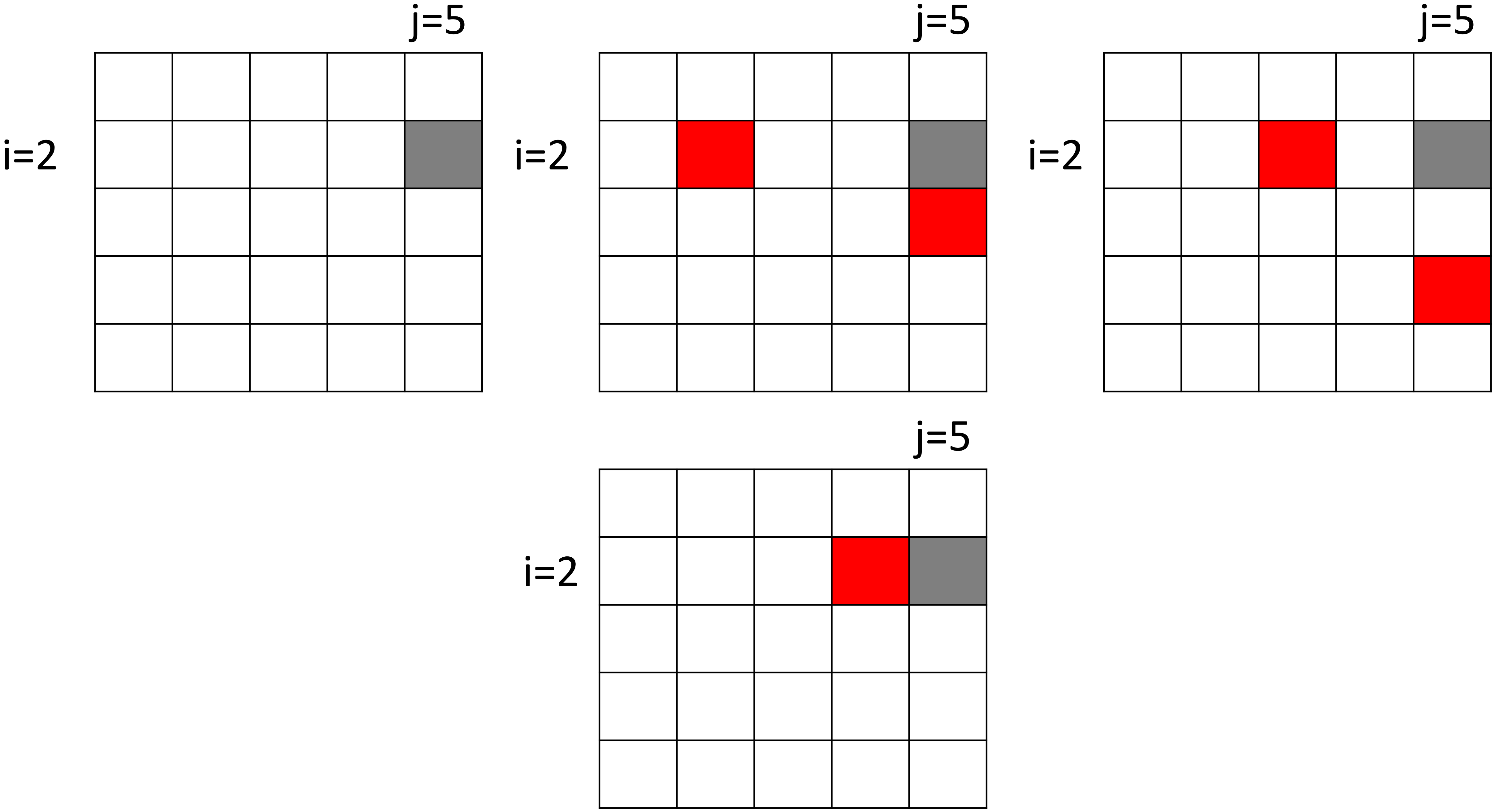
$$C(i,j) = \min_{i \le k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

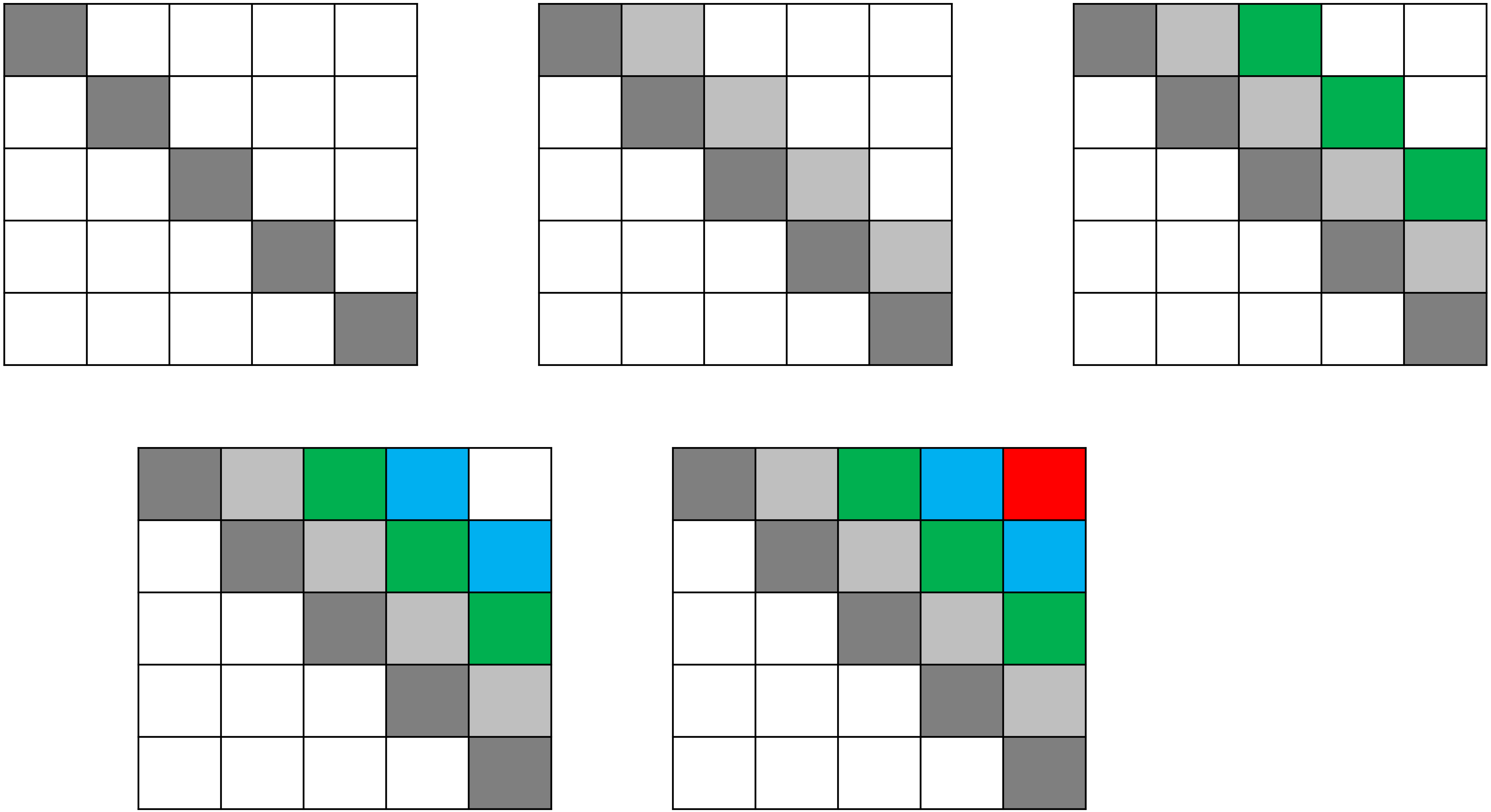
- 初始化：for i=1 to n:C(i,i) = 0



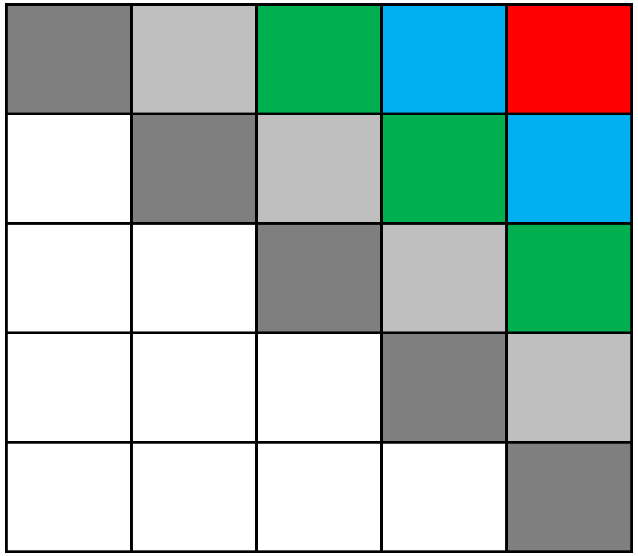


$$C(i,j) = \min_{i \le k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$


j=5
i=2
j=5
i=2
j=5
i=2
j=5
i=2



令 $s = |j-i|$



for $i=1$ to n: $C(i,i) = 0$

for $s=1$ to $n-1$:

  for $i=1$ to $n-s$:

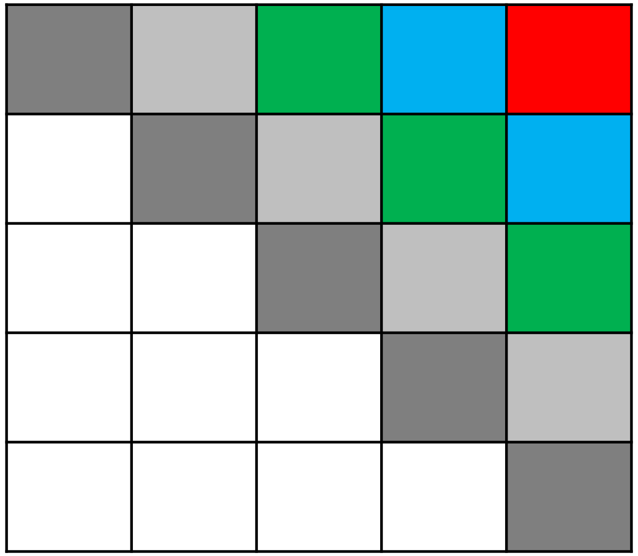
    $j = i+s$

    $C(i,j) = \min\{C(i,k)+C(k+1,j)+m_{i-1}\, m_k\, m_j : i \le k < j\}$

return $C(1,n)$

$T(n) = O(n^3)$

输出解决方案



for i=1 to n: C(i,i) = 0

for s=1 to n-1:

    for i=1 to n-s:

        j = i+s

        for k=l to j-1:

            if $C(i,k)+C(k+1,j)+m_{i-1}\, m_k\, m_j < C(i,j)$

              $C(i,j) = C(i,k)+C(k+1,j)+m_{i-1}\, m_k\, m_j$

              P(i,j) =k

return C(1,n)

```
Print_optimal_parentenses(P , i, j ):
    if i==j: print(Ai); return
    print("(")
    Print_optimal_parentenses(P,i,P[i][j])   //(i…k)
    Print_optimal_parentenses(P,P[i][j]+1,j)
    print(")")
```