暴力递归动态规划

步骤：暴力尝试 ——> 记忆化搜索 ——> 严格表 DP ——> 精致的表结构 DP ——>（枚举行为无必要）斜率优化：看能否用邻近替代枚举

暴力尝试的方法：

1. 从左至右的尝试方法

评价尝试的好坏：

1. 单可变参数维度
2. 可变参数个数越少越好

严格表结构推导：

1. 分析可变参数的变化范围
2. 标出要计算的终止位置
3. 表出不用计算直接出答案的位置 base case
4. 推出普遍位置如何依赖其他位置（看依赖）
5. 确定依次计算的顺序，定出严格表是从那些格子到哪些格子

机器人到达指定位置方法数

【题目】
假设有排成一行的 N 个位置，记为 1~N，N 一定大于或等于 2。开始时机器人在其中的 M 位置上(M 一定是 1~N 中的一个)，机器人可以往左走或者往右走，如果机器人来到 1 位置， 那么下一步只能往右来到 2 位置;如果机器人来到 N 位置，那么下一步只能往左来到 N-1 位置。规定机器人必须走 K 步，最终能来到 P 位置(P 也一定是 1~N 中的一个)的方法有多少种。给定四个参数 N、M、K、P，返回方法数。

【举例】
N=5,M=2,K=3,P=3
上面的参数代表所有位置为 1 2 3 4 5。机器人最开始在 2 位置上，必须经过 3 步，最后到达 3 位置。走的方法只有如下 3 种:

1. 从 2 到 1，从 1 到 2，从 2 到 3
2. 从 2 到 3，从 3 到 2，从 2 到 3
3. 从 2 到 3，从 3 到 4，从 4 到 3

所以返回方法数 3。 N=3,M=1,K=3,P=3

上面的参数代表所有位置为 1 2 3。机器人最开始在 1 位置上，必须经过 3 步，最后到达 3 位置。怎么走也不可能，所以返回方法数 0。

暴力递归

```
// 暴力尝试
function RobotWalk(N, M, K, P) {
  if (N < 2 || K < 1 || M < 1 || M > N || P < 1 || P > N) {
    return 0;
  }
```

```
  function process(cur, P, rest) {
    if (rest === 0) {
      return cur === P ? 1 : 0;
    }

    if (cur === 1) {
      return process(cur + 1, P, rest - 1);
    }

    if (cur === N) {
      return process(cur - 1, P, rest - 1);
    }

    return process(cur + 1, P, rest - 1) + process(cur - 1, P, rest - 1);
  }

  return process(M, P, K);
}
```

记忆搜索优化

N P 是固定参数，rest 和 cur 是可变参数

可变参数固定，返回值一样，无后效性

```
// 记忆缓存
function RobotWalk2(N, M, K, P) {
  if (N < 2 || K < 1 || M < 1 || M > N || P < 1 || P > N) {
    return 0;
  }
  const dp = new Array(K + 1);
  for (let i = 0; i < K + 1; i++) {
    dp[i] = new Array(N + 1).fill(-1);
  }
  function process(N, P, rest, cur) {
    if (dp[rest][cur] != -1) {
      return dp[rest][cur];
    }
    //缓存没命中
    if (rest == 0) {
      dp[rest][cur] = cur == P ? 1 : 0;
    } else {
      //rest > 0 还有路可以走
      if (cur == 1) {
        // 1 - > 2
        dp[rest][cur] = process(N, P, rest - 1, 2);
      } else if (cur == N) {
        // N -> N -1
        dp[rest][cur] = process(N, P, rest - 1, N - 1);
      } else {
        //中间位置
        dp[rest][cur] =
          process(N, P, rest - 1, cur - 1) + process(N, P, rest - 1, cur + 1);
      }
    }

    return dp[rest][cur];
  }

  return process(N, P, K, M);
}
```

严格表结构

```
// 严格表结构
function RobotWalk3(N, M, K, P) {
  if (N < 2 || K < 1 || M < 1 || M > N || P < 1 || P > N) {
    return 0;
  }
  const dp = new Array(K + 1);
  for (let i = 0; i < K + 1; i++) {
    dp[i] = new Array(N + 1).fill(0);
  }
  dp[0][P] = 1;
  for (let i = 1; i <= K; i++) {
    for (let j = 1; j <= N; j++) {
      if (j === 1) {
        dp[i][j] = dp[i - 1][j + 1];
      } else if (j === N) {
        dp[i][j] = dp[i - 1][j - 1];
      } else {
        dp[i][j] = dp[i - 1][j + 1] + dp[i - 1][j - 1];
      }
    }
  }
  console.log(dp);
  return dp[K][M];
}
```

换钱的最少货币数

【题目】

给定数组 arr，arr 中所有的值都为正数且不重复。每个值代表一种面值的货币，每种面值的货币可以使用任意张，再给定一个整数 aim，代表要找的钱数，求组成 aim 的最少货币数。

【举例】

arr=[5,2,3]，aim=20。

4 张 5 元可以组成 20 元，其他的找钱方案都要使用更多张的货币，所以返回 4。

arr=[5,2,3]，aim=0。

不用任何货币就可以组成 0 元，返回 0。

arr=[3,5]，aim=2。

根本无法组成 2 元，钱不能找开的情况下默认返回-1。

暴力递归

```
function CoinsMin(arr, aim) {
  function process(index, rest) {
    if (index == arr.length) {
      return rest === 0 ? 0 : -1;
    }
    let res = -1;
```

```
    for (let i = 0; i * arr[index] <= rest; i++) {
      let next = process(index + 1, rest - i * arr[index]);
      if (next !== -1) {
        res = res === -1 ? next + i : Math.min(res, next + i);
      }
    }

    return res;
  }
  return process(0, aim);
}
```

记忆搜索优化

```
function CoinsMin2(arr, aim) {
  const dp = new Array(arr.length + 1);
  for (let i = 0; i < dp.length; i++) {
    dp[i] = new Array(aim + 1).fill(-2);
  }
  function process(index, rest) {
    if (index === arr.length) {
      return rest === 0 ? 0 : -1;
    }
    if (dp[index][rest] !== -2) {
      return dp[index][rest];
    }
    let res = -1;
    for (let i = 0; i * arr[index] <= rest; i++) {
      let next = process(index + 1, rest - i * arr[index]);
      if (next !== -1) {
        res = res === -1 ? next + i : Math.min(res, next + i);
      }
    }
    dp[index][rest] = res;
    return res;
  }
  return process(0, aim);
}
```

严格表结构

```
function CoinsMin3(arr, aim) {
  const dp = new Array(arr.length + 1);
  for (let i = 0; i < dp.length; i++) {
    dp[i] = new Array(aim + 1).fill(-1);
  }
  dp[arr.length][0] = 0;

  // 从下往上计算
  for (let i = arr.length - 1; i >= 0; i--) {
    // 从左往右
    for (let rest = 0; rest <= aim; rest++) {
      if (dp[i + 1][rest] !== -1) {
        dp[i][rest] = dp[i + 1][rest];
      }
      if (rest - arr[i] >= 0 && dp[i][rest - arr[i]] !== -1) {
        if (dp[i][rest] === -1) {
          dp[i][rest] = dp[i][rest - arr[i]] + 1;
        } else {
```

```
          dp[i][rest] = Math.min(dp[i][rest - arr[i]] + 1, dp[i][rest]);
        }
      }
    }
  }

  return dp[0][aim];
}
```

排成一条线的纸牌博弈问题

【题目】

给定一个整型数组 arr，代表数值不同的纸牌排成一条线。玩家 A 和玩家 B 依次拿走每张纸 牌，规定玩家 A 先拿，玩家 B 后拿，但是每个玩家每次只能拿走最左或最右的纸牌，玩家 A 和 玩家 B 都绝顶聪明。请返回最后获胜者的分数。

【举例】

arr=[1,2,100,4]。

开始时，玩家 A 只能拿走 1 或 4。如果玩家 A 拿走 1，则排列变为[2,100,4]，接下来玩家 B 可以拿走 2 或 4，然后继续轮到玩家 A。如果开始时玩家 A 拿走 4，则排列变为[1,2,100]，接下来玩家 B 可以拿走 1 或 100，然后继续轮到玩家 A。玩家 A 作为绝顶聪明的人不会先拿 4，因为 拿 4 之后，玩家 B 将拿走 100。所以玩家 A 会先拿 1，让排列变为[2,100,4]，接下来玩家 B 不管 怎么选，100 都会被玩家 A 拿走。玩家 A 会获胜，分数为 101。所以返回 101。

arr=[1,100,2]。

开始时，玩家 A 不管拿 1 还是 2，玩家 B 作为绝顶聪明的人，都会把 100 拿走。玩家 B 会获胜，分数为 100。所以返回 100

暴力递归

```
function CardsInLine(arr) {
  // 先手
  function f(i, j) {
    if (i === j) {
      return arr[i];
    }
    return Math.max(arr[i] + s(i + 1, j), arr[j] + s(i, j - 1));
  }

  // 后手
  function s(i, j) {
    if (i === j) {
      return 0;
    }

    return Math.min(f(i + 1, j), f(i, j - 1));
  }

  return Math.max(f(0, arr.length - 1), s(0, arr.length - 1));
}
```

记忆搜索优化

严格表结构

```
// 动态规划
function CardsInLine2(arr) {
  const dpf = new Array(arr.length);
  for (let i = 0; i < arr.length; i++) {
    dpf[i] = new Array(arr.length);
    dpf[i][i] = arr[i];
  }
  const dps = new Array(arr.length);
  for (let i = 0; i < arr.length; i++) {
    dps[i] = new Array(arr.length);
    dps[i][i] = 0;
  }
  let col = 1,
    row = 0;
  while (col < arr.length) {
    let i = row,
      j = col;
    while (i < arr.length && j < arr.length) {
      dpf[i][j] = Math.max(arr[i] + dps[i + 1][j], arr[j] + dps[i][j - 1]);
      dps[i][j] = Math.min(dpf[i + 1][j], dpf[i][j - 1]);
      i++;
      j++;
    }
    col++;
  }
  return Math.max(dpf[0][arr.length - 1], dps[0][arr.length - 1]);
}
```

象棋中马的跳法

【题目】

请同学们自行搜索或者想象一个象棋的棋盘，然后把整个棋盘放入第一象限，棋盘的最左下角是(0,0)位置。那么整个棋盘就是横坐标上 9 条线、纵坐标上 10 条线的一个区域。给你三个参数，x，y，k，返回如果“马”从(0,0)位置出发，必须走 k 步，最后落在(x,y)上的方法数有多少种?

暴力递归

```
// 暴力递归
function HorseJump(x, y, k) {
  function process(x, y, k) {
    if (x < 0 || x > 8 || y < 0 || y > 9) {
      return 0;
    }
    if (k === 0) {
      return x === 0 && y === 0 ? 1 : 0;
    }

    let ways =
      process(x - 1, y + 2, k - 1) +
      process(x - 2, y + 1, k - 1) +
```

```
      process(x - 2, y - 1, k - 1) +
      process(x - 1, y - 2, k - 1) +
      process(x + 1, y - 2, k - 1) +
      process(x + 2, y - 1, k - 1) +
      process(x + 2, y + 1, k - 1) +
      process(x + 1, y + 2, k - 1);

    return ways;
  }

  return process(x, y, k);
}
```

记忆搜索优化

```
// 记忆搜索优化
function HorseJump2(x, y, k) {
  const dp = new Array(9);
  for (let i = 0; i < 9; i++) {
    dp[i] = new Array(10);
    for (let j = 0; j < 10; j++) {
      dp[i][j] = new Array(k + 1);
      for (let m = 0; m < k + 1; m++) {
        dp[i][j][m] = -1;
      }
    }
  }

  function process(x, y, k) {
    if (x < 0 || x > 8 || y < 0 || y > 9) {
      return 0;
    }
    if (k === 0) {
      dp[x][y][k] = x === 0 && y === 0 ? 1 : 0;
      return x === 0 && y === 0 ? 1 : 0;
    }
    if (dp[x][y][k] !== -1) {
      return dp[x][y][k];
    }

    dp[x][y][k] =
      process(x - 1, y + 2, k - 1) +
      process(x - 2, y + 1, k - 1) +
      process(x - 2, y - 1, k - 1) +
      process(x - 1, y - 2, k - 1) +
      process(x + 1, y - 2, k - 1) +
      process(x + 2, y - 1, k - 1) +
      process(x + 2, y + 1, k - 1) +
      process(x + 1, y + 2, k - 1);
    return dp[x][y][k];
  }

  return process(x, y, k);
}
```

严格表结构

```
// 动态规划
function HorseJump3(x, y, k) {
```

```
  const dp = new Array(9);
  for (let i = 0; i < 9; i++) {
    dp[i] = new Array(10);
    for (let j = 0; j < 10; j++) {
      dp[i][j] = new Array(k + 1);
      for (let m = 0; m < k + 1; m++) {
        dp[i][j][m] = 0;
      }
    }
  }
  dp[0][0][0] = 1;

  // m从1开始
  for (let m = 1; m < k + 1; m++) {
    for (let i = 0; i < 9; i++) {
      for (let j = 0; j < 10; j++) {
        dp[i][j][m] =
          getValue(i - 1, j + 2, m - 1) +
          getValue(i - 2, j + 1, m - 1) +
          getValue(i - 2, j - 1, m - 1) +
          getValue(i - 1, j - 2, m - 1) +
          getValue(i + 1, j - 2, m - 1) +
          getValue(i + 2, j - 1, m - 1) +
          getValue(i + 2, j + 1, m - 1) +
          getValue(i + 1, j + 2, m - 1);
      }
    }
  }

  function getValue(x, y, k) {
    if (x < 0 || x > 8 || y < 0 || y > 9) {
      return 0;
    }
    return dp[x][y][k];
  }

  return dp[x][y][k];
}
```

Bob 的生存概率

【题目】

给定五个参数 n,m,i,j,k。表示在一个 N*M 的区域，Bob 处在(i,j)点，每次 Bob 等概率的向上、下、左、右四个方向移动一步，Bob 必须走 K 步。如果走完之后，Bob 还停留在这个区域上，就算 Bob 存活，否则就算 Bob 死亡。请求解 Bob 的生存概率，返回字符串表示分数的方式。

暴力递归

```
function BobDie(N, M, i, j, K) {
  // 从(i,j)剩余K步，Bob的存活的可能路径数
  function process(i, j, rest) {
    if (i < 0 || i >= N || j < 0 || j >= M) {
      return 0;
    }
    if (rest === 0) {
      return 1;
```

```
    }
    return (
      process(i - 1, j, rest - 1) +
      process(i + 1, j, rest - 1) +
      process(i, j - 1, rest - 1) +
      process(i, j + 1, rest - 1)
    );
  }

  let live = process(i, j, K);
  let all = Math.pow(4, K);
  return `${live}/${all}`;
}
```

记忆搜索优化

严格表结构

动态规划技巧

空间压缩

矩阵的最小路径和

【题目】

给定一个矩阵 m，从左上角开始每次只能向右或者向下走，最后到达右下角的位置，路径上所有的数字累加起来就是路径和，返回所有的路径中最小的路径和。

【举例】

如果给定的 m 如下: 1359 8134 5061 8840
路径 1，3，1，0，6，1，0 是所有路径中路径和最小的，所以返回 12

四边形不等式

枚举过程的状态化简

复杂状态用位信息代替

用业务反推动态规划表的初始状态