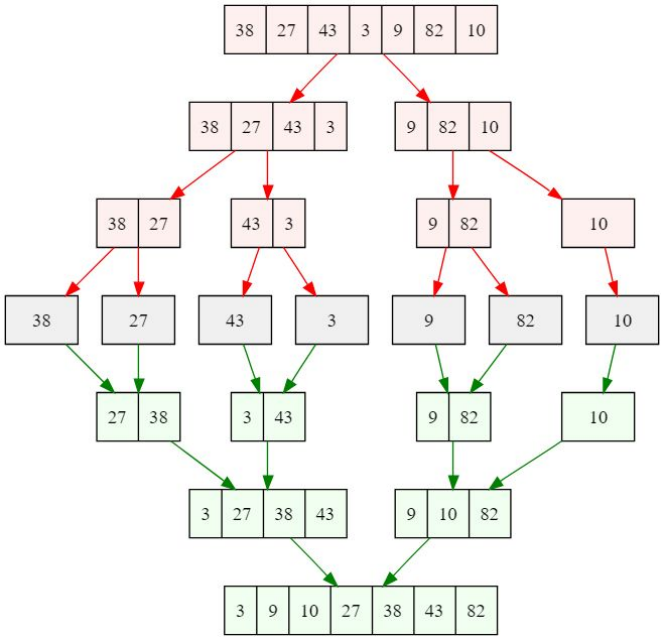
上一章介绍了算法分析，这一章继续回到图论专题。既然有了算法分析的基础，之后讲算法的时候就可以更多地从分析角度看待算法，作者也会在算法章节中穿插算法分析章节，包括复杂度计算和证明等等。根据民主投票，这一章将介绍众望所归的动态规划（dynamic programming）和记忆化搜索（search with memoization，真没拼错），其它建议在后续章节中同样会讲。

动态规划既是数学优化方法也是编程方法。它于二十世纪五十年代由Richard Bellman提出，在许多领域都有应用。与其说动态规划是一种数学或计算机方法，不如说它是一种思维模式，其本质是一种递归（recursion）：把庞大的问题拆分成子问题解决。说到递归，一个典型的例子就是归并排序（merge sort）。



（from Wikipedia）

如上图所示，待排序数组为[38, 27, 43, 3, 9, 82, 10]，我们使用归并排序从小到大排序。首先如红色部分所示进行拆分（divide），把数组不断从中间分成两份，直到每份包含一到两个元素，即上图第三行。接下来把每份数字排序（每份至多一次比较）得到上图第五行，如绿色部分所示进行合并（merge），具体算法这里不再赘述。

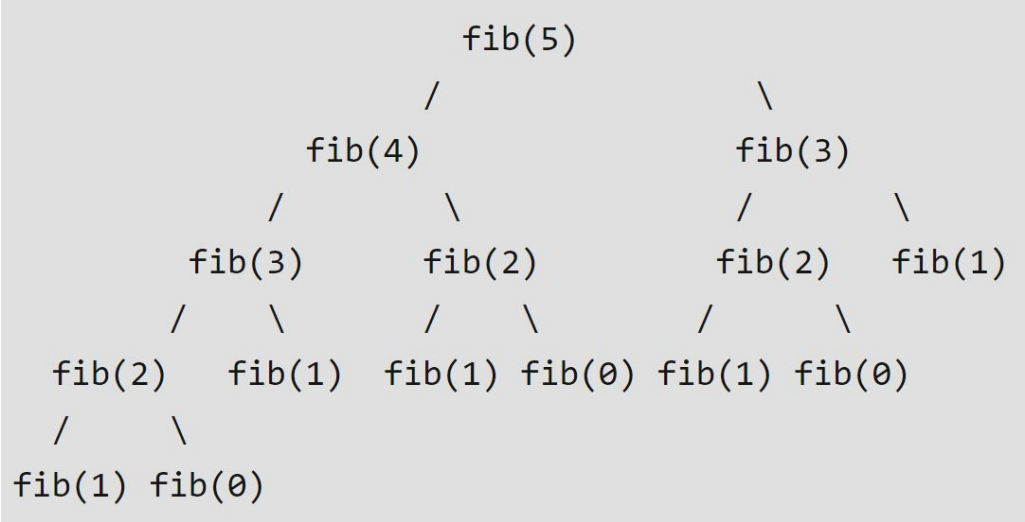
可以看到归并排序先使用递归将数组拆分，再使用递归进行排序合并，类似的方法被称之为分治法（divide-and-conquer），即把一个问题分解成一系列相似的子问题，分别递归解决这些子问题，再把子问题的解合并得到原问题的解。动态规划和分治法相似，区别在于分治法的子问题是独立、互不相干的，而动态规划的子问题是相互关联的。动态规划适用的问题一个典型的特征就是，子问题依赖子问题的子问题（或子问题的前置问题）的解来解决，一些子问题还会依赖相同的前置问题。

求斐波那契数列（Fibonacci zequence）第n个数是适用动态规划的问题之一。虽然它可以用数学公式直接得出，但如果我们只有递推公式f(n) = f(n - 1) + f(n - 2)呢？下图是一个朴素的计算机解法（C++）。

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

（from GeeksForGeeks）

不难看出这个解法的时间复杂度是指数级的，递归第一层需要调用fib()一次，第二层需要调用fib()两次，第三层需要调用fib()四次，第四层需要调用fib()八次，以此类推。如何优化这个算法呢？通过观察我们可以发现其中存在大量重复的步骤。以fib(5)为例，fib()的调用流程如下图所示，可以发现fib(3)、fib(2)、fib(1)被调用了很多次，fib(i)中i越小，被重复调用的次数就越多。



（from GeeksForGeeks）

如果把这些结果存起来的话，每个fib(i)只要计算一次就够了。前两个数已经给出了，我们可以依次计算出从第三个数到第n个数的值，计算第i个数的时候只需要调取前面两个已经

计算好的数字即可，这一步骤时间复杂度为常数，计算fib(n)的总体时间复杂度为O(n)。C++代码实现如下图所示。

```
int fib(int n)
{
  /* Declare an array to store Fibonacci numbers. */
  int f[n+2];   // 1 extra to handle case, n = 0
  int i;

  /* 0th and 1st number of the series are 0 and 1*/
  f[0] = 0;
  f[1] = 1;

  for (i = 2; i <= n; i++)
  {
      /* Add the previous 2 numbers in the series
         and store it */
      f[i] = f[i-1] + f[i-2];
  }

  return f[n];
}
```

（from GeeksForGeeks）

这个方法就是动态规划。我们可以把上图for循环中每个f[i]的计算（即斐波那契数列第i个数字的计算）都看作一个子问题，第i个数字之后每个数字的计算都依赖第i个数字的值，也就是之前提到的“一些子问题还会依赖相同的前置问题”。由此可见，动态规划也可以被理解为对暴力搜索的优化，而暴力搜索的时间复杂度通常是指数级的。动态规划只删除了暴力搜索中大量重复的步骤，因此也不会改变最优解。

那么大家最关心的问题来了：遇到一个问题，我们如何知道它可以用动态规划解决，可以的话又如何找到子问题呢？首先，动态规划分为三个主要步骤：

1. 拆分成相似的子问题，找到原问题的最优子结构（optimal substructure，即子问题的结构，该结构使得子问题的最优解都包含在原问题最优解中）
2. 不断递归依次得到子问题的最优解
3. 根据已经得出的子问题的最优解计算出原问题的最优解

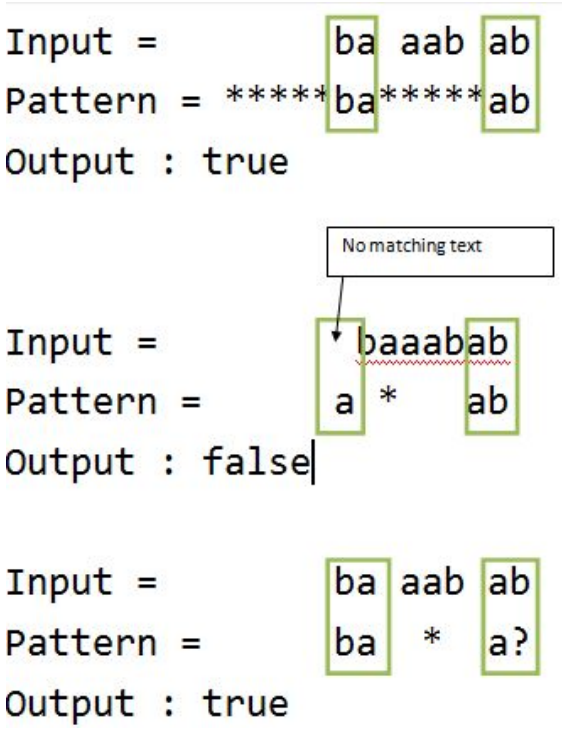
如果从图论的角度看待这些步骤，子问题的结构可以用一个有向无环图来表示，其中每个子问题为一个顶点，每个子问题间的依赖关系为一条有向边，原问题为终点。我们不断计算入度为0的顶点（即不依赖其它未计算的子问题）所对应的子问题的值，并去掉从该顶点出发的所有边，这样一来会产生新的入度为0的顶点（即所有前置问题有解，该子问题

可计算），最终得出原问题的最优解。但实际上并不一定这么操作，因为子问题的计算顺序通常本身就符合拓扑次序，比如之前斐波那契数列的例子。

概括来说，动态规划就是把一个问题的解拆分为一系列通向最优解的选择。一次只做一最优选择，即遍历由当前状态S可抵达的所有状态，选出其中的最佳状态记录下来，此后每当依赖S做选择的时候只需参考这一最佳状态即可。这一点很像贪心算法，但由于动态规划的最优解不一定局部最优，因此区别在于贪心算法做选择时基于前置的一种状态（即局部最优状态），而动态规划做选择时基于前置的一种或多种状态（即前置且相邻的所有状态，不一定局部最优 ）。

考虑到动态规划本身可能较难理解，在这里举几个例子：

1. Leetcode 44. Wildcard Matching（仅包含’?’和’*’两个特殊字符的正则表达式匹配）
   ’?’没啥好说的，特殊处理一下就好，重点在’*’。在这里’*’可以匹配任意长度的任意字符，因此带来的变数是每个’*’分别匹配了多少字符。根据之前提到的“把一个问题的解拆分为一系列通向最优解的选择”，我们可以依次选择从左到右每个’*’匹配到的字符数目。例如“a*b*a”匹配“aaabbbaaa”，我们首先对第一个’*’做出选择，遍历它匹配到一个字符的状态，两个字符的状态，三个字符的状态，以此类推，直到它匹配成功或到达字符串末尾。一旦成功，我们就记录它匹配成功的状态（即正则表达式当前匹配到的位置，文本当前匹配到的位置，和是否匹配成功，可以用二维矩阵表示），并继续对第二个’*’用类似方法做出选择。对于这个例子结果是第一个’*’匹配到了2个字符，第二个’*’匹配到了4个字符。

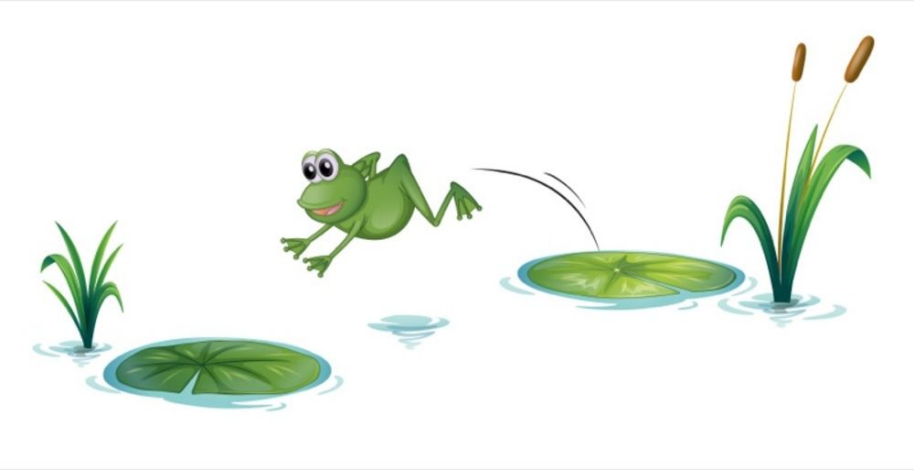


（from GeeksForGeeks）

2. Leetcode 188. Best Time to Buy and Sell Stock IV（给出n天的股票价格，进行不超过k笔交易，且同时只能进行一笔交易，最多能赚多少钱）
   根据之前提到的“把一个问题的解拆分为一系列通向最优解的选择”，我们能做什么选择呢？可以想到在某一天，如果我们手上有股票，我们可以选择把它卖掉或不作为；如果我们手上没股票，我们可以选择买股票或不作为。此时我们需要记录天数，是否持股，当前交易次数，以及当前最大盈利（衡量最优解的标准），可以用三维矩阵表示。对动态规划熟练的话不难发现每次交易可以用两个日期组成的区间表示，由此可以把题目转换为在股票价格数组中放不超过k个不重叠的区间，此时即可把三维矩阵转成二维矩阵，所需记录的分别是天数，当前交易次数，和当前最大盈利。

（from Leetcode）

3. Leetcode 403. Frog Jump（一条河从左到右被分成x个单位，每份上可能有石头，所有石头的位置由数组给出，每个数代表石头在第几个单位，青蛙从左出发，第一次只能跳一个单位，此后如果上一次跳了k个单位，则下一次只能跳k-1或k+1个单位，问能不能到达最后一块石头）
   根据之前提到的“把一个问题的解拆分为一系列通向最优解的选择”，不难想到我们可以选择每次青蛙跳的距离。此时我们需要记录青蛙的当前位置，步长，和青蛙在当前位置是否能跳该步长，可以用二维矩阵表示。值得一提的是，青蛙跳的次数也是变量，为什么不像之前记录交易次数一样记录青蛙跳的次数呢？因为我们只需要知道青蛙能不能跳到最后的石头，所以最终结果不依赖跳的次数，中间经历的任何选择也不依赖跳的次数。类似的迷惑变量有很多，需要在不同问题中甄别出真正有用的变量。



通过这些例子也许会发现，动态规划本身都是一个套路。构思动态规划的一种方式是上面提到的思考能做哪些选择，另一种方式是先想递归再想去重，还有一种更简单粗暴的方式就是思考如果列矩阵可以列哪些维度（即定义域有限的离散变量）。动态规划确实在算法题中算比较难的类型，因为有些问题不是很直白，可能难以联想到动态规划，希望这一章的内容可以帮到大家。

由于字（zuo）数（zhe）限（tai）制（lan），动态规划中最核心的转移方程和记忆化搜索将放到下一章讲解，敬请期待 :)

相关材料：
https://leetcode.com/problems/wildcard-matching/
https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/
https://leetcode.com/problems/frog-jump/