数据结构与算法（五）

1：今日课程内容介绍(了解)

今天的课程中我们主要来讲解四大算法思想：贪心，分治，回溯，动态规划；因为这四种并不是具体的算法而是一种算法思想，更多了偏向理论，需要我们去理解。除此之外我们再讲解一个数据结构 B+树。

2：今日课程目标(了解)

1：理解贪心算法思想

2：理解分治算法思想

3：理解回溯算法思想

4：理解动态规划算法思想

5：理解 B+树的构建过程及特点

3：算法思想(理解并掌握)

在本章节中我们会讲解几种基本的算法，分别是：贪心算法，分治算法，回溯算法，动态规划。如果更加确切的说他们应该算是一种算法思想而并非一种具体的算法，这些算法思想经常用以指导我们设计具体的算法和应用程序编码。

既然这几种都是属于算法思想，所以原理性的东西有很多，但是要想真正的掌握并且灵活应用并非易事，在接下来的讲解过程中每一个都会配备具体的案例去讲解，这样能让我们更加的认识到这些算法实现在实际软件编程过程中的应用。

3.1.贪心

贪心算法（英语：greedy algorithm），又称**贪婪算法**，是一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是最好或最优的算法。 比如在旅行推销员问题中，如果旅行员每次都选择最近的城市，那这就是一种**贪心算法**。 贪心算法在有最优子结构的问题中尤为有效。

看这个算法的名字：贪心，贪婪，两个字的含义最关键，好像一个贪婪的人所有事情都只想到眼前，看不到长远，也不为最终的结果和将来着想，贪图眼前局部的利益最大化。接下来我们看几个关于贪心算法的案例情形。

3.1.1、背包问题

有一个可以容纳 100kg 物品的背包，可以装各种物品。有以下 5 种水果，每种水果的重量和总价值都各不相同。为了让背包中所装物品的总价值最大，但不能超过背包所能容纳的总重量，我们如何选择在背包中装哪些水果？每种水果又该装多少呢？

水果	质量(kg)	价值(元)
苹果	100	100
梨	30	90
香蕉	60	120
菠萝	20	80
圣女果	50	75

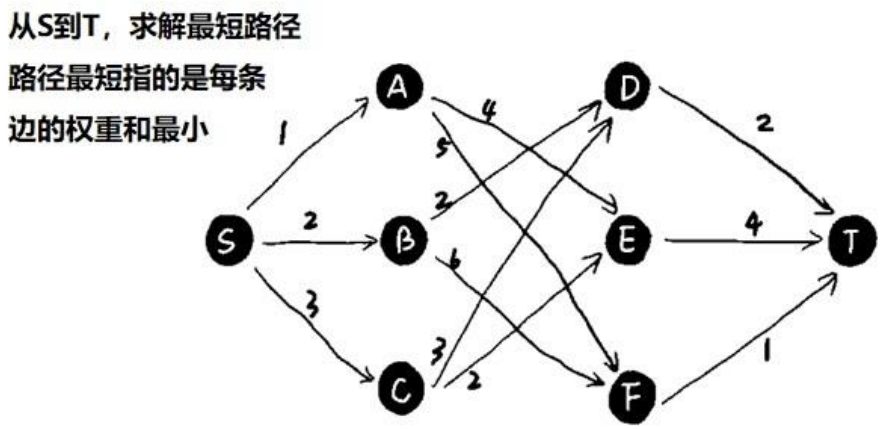
对于这个问题我们只要先算一算每个物品的单价，按照单价由高到低依次来装就好了。单价从高到低排列，依次是：菠萝、梨、香蕉、圣女果、苹果，所以，我们可以往背包里装 20kg 菠萝、30kg 梨、50kg 圣女果。这个问题的解决思路显而易见，它本质上借助的就是贪心算法。

从这里我们能简单总结出贪心算法的基本思路：

1. 建立数学模型来描述问题。
2. 把求解的问题分成若干个子问题。
3. 对每一子问题求解，得到子问题的局部最优解。
4. 把子问题的解局部最优解合成原来解问题的一个解。

当然了贪心算法给出的问题的解并不一定是最优的解，比如下面这个例子，在一个有权图中，我们从顶点 S 开始，找一条到顶点 T 的最短路径（路径中边的权值和最小）。

如果我们用贪心算法来解决这个问题那贪心算法的解决思路是，每次都选择一条跟当前顶点相连的权最小的边，直到找到顶点 T。按照这种思路，我们求出的最短路径是 S->A->E->T，路径长度是 1+4+4=9



但是实际上，用贪心算法选择的路径并不是最短的，因为 S->B->D->T 才是最短路径，因为这条路径的长度是 2+2+2=6。那为什么此时贪心算法无法给出最优解呢？

在这个问题上，前面的选择会影响后面的选择，如果我们第一步从顶点 S 走到顶点 A，那接下来面对的顶点和边，跟第一步从顶点 S 走到顶点 B，是完全不同的。所以，即便我们第一步选择最优的走法（边最短），但有可能因为这一步选择，导致后面每一步的选择都很糟糕，最终也就无缘全局最优解了。

因此适合用贪心算法解决问题的前提是：**局部最优策略能导致产生全局最优解。**

实际上，贪心算法适用的情况很少。一般，对一个问题分析是否适用于贪心算法，可以先选择该问题下的几个实际数据进行分析，就可做出判断。

因为用贪心算法只能通过解局部最优解的策略来达到全局最优解，因此，一定要注意判断问题是否适合采用贪心算法策略，找到的解是否一定是问题的最优解。

3.1.2、钱币找零问题

假设我们有 1 元、2 元、5 元、10 元、20 元、50 元、100 元这些面额的纸币，它们的张数分别是 c1、c2、c5、c10、c20、c50、c100。我们现在要用这些钱来支付 K 元，最少要用多少张纸币呢？

如果我们用贪心算法的思想来解决这个问题，为了使用的纸币的张数最少每一步尽可能用面值大的纸币即可。在日常生活中我们自然而然也是这么做的

3.1.3、区间问题

有 n 个需要在同一天使用同一个活动场地的活动[a1,a2,...,an]，活动场地同一时刻只能由一个活动使用。每个活动 ai 都有一个开始时间 si 和结束时间 fi 。一旦被选择后，活动 ai 就占据半开时间区间[si,fi)。如果[si,fi]和[sj,fj]互不重叠，ai 和 aj 两个活动就可以被安排在这一天。该问题就是如何安排这些活动使得尽量多的活动能不冲突的举行？

这个问题的处理思路稍微复杂一点，不过弄懂这个问题的思路有助于我们后续处理其他同类型的问题，因为这个处理思想在很多贪心算法问题中都有用到，比如任务调度、教师排课等等问题。

这个问题的解决思路是这样的：因为每个活动有一个活动时间区间，我们假设这 n 个区间中最左端点 lmin，最右端点是 rmax。这个问题就相当于，我们选择几个不相交的区间，从左到右将 [lmin, rmax] 覆盖上。我们按照起始端点从小到大的顺序对这 n 个区间排序。我们每次选择的时候，左端点跟前面的已经覆盖的区间不重合的，右端点又尽量小的，这样可以让剩下的未覆盖区间尽可能的大，就可以放置更多的区间。这实际上就是一种贪心的选择方法

3.2.分治

分治算法在维基百科上的定义为：在计算机科学中，**分治法**是建基于多项分支递归的一种很重要的算法范式。字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

通过维基百科的定义我们可以发现分治算法的核心就是分而治之，当然这个定义和递归有点类似，这里我们要说一下分治和递归的区别：**分治算法是一种处理问题的思想，递归是一种编程技巧**，当然了在实际情况中，分治算法大都采用递归来实现，并且用递归实现的分治算法的基本步骤为：

1：分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

2：解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题

3：合并：将各个子问题的解合并为原问题的解。

什么样的问题适合用分治算法去解决呢？我总结出如下几条

- 原问题与分解成的小问题具有相同的模式；
- 原问题分解成的子问题可以独立求解，子问题之间没有相关性，这一点是分治算法跟动态规划的

  明显区别，至于动态规划下一小节会详细讲解并对比这两种算法；
- 具有分解终止条件，也就是说，当问题足够小时，可以直接求解；
- 可以将子问题合并成原问题，并且合并操作的复杂度不能太高，否则就起不到减小算法总体复

  杂度的效果了，这也是能否使用分治法的关键特征

分治算法应用场景举例：

3.2.1、排序

我们前面讲解的排序算法中就有很多使用了分治的思想，比如归并排序，快速排序，比如我们可以来回顾一下归并排序的代码实现

```
public int[] mergeSort(int[] arr){
        if(arr.length < 2){
            return arr;
        }
        //将数组从中间拆分成左右两部分
        int mid = arr.length/2;
        int[] left = Arrays.copyOfRange(arr,0,mid);
        int[] right = Arrays.copyOfRange(arr,mid,arr.length);
        return merge(mergeSort(left),mergeSort(right));
    }
    /**
     * 合并两个有序数组并返回新的数组
     * @param left
     * @param right
     */
    public int[] merge(int[] left,int[] right){
        //创建一个新数组,长度为两个有序数组的长度之和
        int[] newArray = new int[left.length+right.length];

        //定义两个指针,分别代表两个数组的下标
        int lindex=0;
        int rindex=0;
        for(int i=0;i<newArray.length;i++){
```

```
            if(lindex >= left.length){
                newArray[i] = right[rindex++];
            }else if(rindex >= right.length){
                newArray[i] = left[lindex++];
            }else if(left[lindex] < right[rindex] ){
                newArray[i] = left[lindex++];
            }else{
                newArray[i] = right[rindex++];
            }
        }
        return newArray;
    }
```

归并排序的核心思就是如果要排序一个数组，我们先把数组从中间分成前后两部分，然后对前后两部分分别排序，再将排好序的两部分合并在一起，这样整个数组就都有序了。

3.2.2、海量数据处理

分治算法思想还经常用在海量数据处理的场景中。比如我们在前面的课程讲解中给出过的例子：给 10GB 的订单文件按照金额排序这样一个需求，看似是一个简单的排序问题，但是因为数据量大，有 10GB，而我们的机器的内存可能只有 2、3GB ，总之就是小于订单文件的大小因而无法一次性加载到内存，所以基础的排序算法在这样的场景下无法使用。

要解决这种数据量大到内存装不下的问题，我们就可以利用分治的思想。我们可以将海量的数据集合根据某种方法，划分为几个小的数据集合，每个小的数据集合单独加载到内存来解决，然后再将小数据集合合并成大数据集合。实际上，利用这种分治的处理思路，不仅仅能克服内存的限制，还能利用多线程或者多机处理，加快处理的速度。

假设现在要给 10GB 的订单排序，我们就可以先扫描一遍订单，根据订单的金额，将 10GB 的文件划分为几个金额区间。比如订单金额为 1 到 100 元的放到一个小文件，101 到 200 之间的放到另一个文件，以此类推。这样每个小文件都可以单独加载到内存排序，最后将这些有序的小文件合并，就是最终有序的 10GB 订单数据了。

如果订单数据存储在类似 GFS 这样的分布式系统上，当 10GB 的订单被划分成多个小文件的时候，每个文件可以并行加载到多台机器上处理，最后再将结果合并在一起，这样并行处理的速度也加快了很多。

3.3.回溯

讲回溯算法之前我们先提一部电影，04 年上映的《蝴蝶效应》，影片中讲述的是主人公为了实现自己的目标一直通过回退的方法回到童年，在一些重要的人生岔路口重新做出选择，最终实现整个美好人生的故事，当然了这只是电影，现实中人生是无法倒退的，但是这其中蕴含的就是思想就是我们要讲的回溯思想。

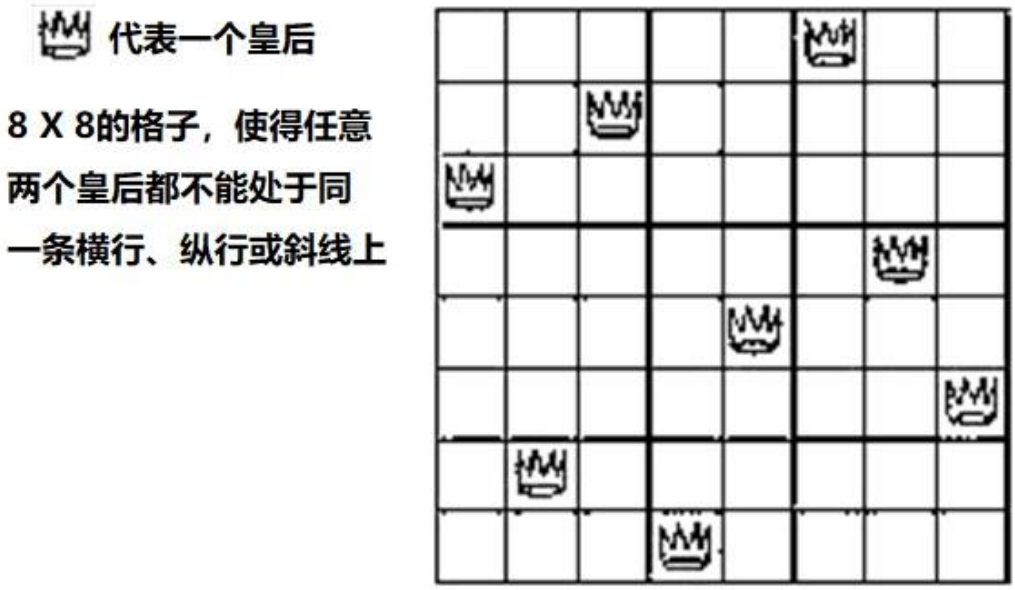
回溯算法实际上是一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。

回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

我们之前讲解的图的广度优先搜索 DFS 的实现就是借助了回溯算法的思想，除此之外很多经典的数学问题都可以运用回溯算法来解决，比如数独，八皇后，0-1 背包等等，下面我们以其中的八皇后问题来讲解回溯算法的应用。

3.3.1、八皇后问题

问题描述：八皇后问题是一个以国际象棋为背景的问题：如何能够在 8×8 的国际象棋棋盘上放置八个皇后，使得任意两个皇后都不能处于同一条横行、纵行或斜线上，如下图所示，请问有多少种摆法？

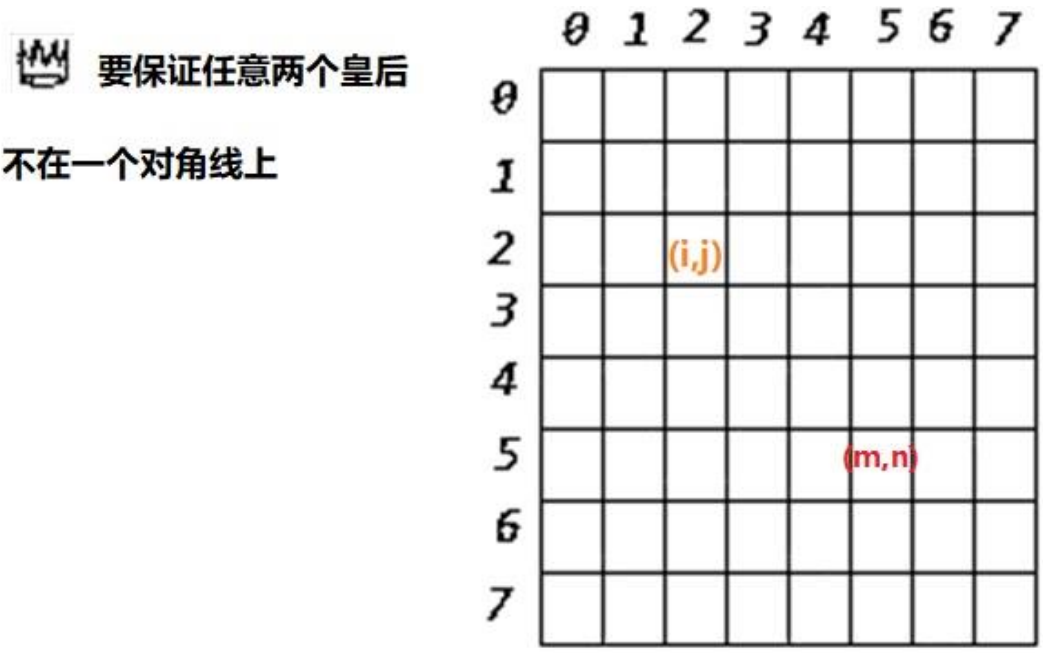


其实八皇后的问题可以更加的延申至 n 皇后的问题，比如一个 n x n 的棋盘上摆放 n 个皇后，要求任意两个皇后都不能处于同一条横行、纵行或斜线上。当然了这个问题当 n=1 或者 n>=4 时才有解。

求解思路：我们可以定义一个一维数组 a[n]，用以保存所有的解，其中 a[i]就表示了把第 i 个皇后放在第 i 行的列数(当然了 i 的值是从 0 开始计算的)，我们知道八皇后有两个约束条件

1：所有任意两个皇后不能在同一列，因此任意两个 a[i]和 a[j]的值不能相同，当然了 i != j，

2：所有任意两个皇后不能在对角线上，那如何判断或者检测任意两个皇后是否在同一个对角线上呢？我们将棋盘的方格变成一个二维数组，如下图所示：



假设有两个皇后被放置在(i,j)和(m,n)的位置上，我们通过分析发现当且仅当 |m-i| = |n-j|时，两个皇后是在同一个对角线上。

实现过程：下面我们对八皇后的问题进行一下实现

```
/**
 * 算法思想：回溯
 *  8 皇后问题
 */
public class Queens8 {

    /** 皇后数组，数组的下标表示第几个皇后，也表示皇后在哪一行，元素的值表示皇后对应的列位置 */
    private int[] queen;

    /**
     * 计算并输出 n 皇后问题的位置
     * @param n 有几个皇后
     */
    public void backtrackMethod(int n) {
```

```
        // 初始化数组
        queen = new int[n];
        for (int i = 0; i < queen.length; i++) {
            queen[i] = -1;// 初始化皇后的起点
        }
        // 从第1个皇后开始
        int k = 0;
        while (true) {
            queen[k] += 1;// 第k个皇后移动一个
                  /* 判断是否应该回溯到上一行开始搜索 */
            if (queen[k] >= n) {// 第k个皇后移动后溢出，即跑出边界，则确定这一行的皇后没有任何位置可选
                if (k > 0) {// 如果不是第一个皇后，则目标皇后更新为地k-1个皇后
                    queen[k] = -1;// 该皇后的起点回归到原始
                    k--;// 目标更新为第k-1个皇后
                    continue;// 跳过下面的判断，从方法体开始处开始
                } else {// 如果该皇后是第一个，则已经遍历所有位置，跳出循环
                    break;
                }
            }
            /**
             * 判断皇后在该位置是否不冲突，更改目标为下一行进行搜索
             */
            if (!willBeEaten(k)) {// 如果不冲突
                k++;// 目标还后更新为下一个
                if (k >= n) {// 如果下标k溢出，即超过了皇后数量则已经确定一个组合，输出
                    for (int i = 0; i < n; i++) {
                        System.out.print(queen[i] + " ");
                    }
                    System.out.println();
                    k--;// k溢出，将k回溯到最后一行继续搜索其他可能性
                }
            }
        }
    }

    /**
     * 判断放置第k个皇后之后是否与之前的皇后冲突，判定冲突的条件是，第k个皇后的于前面的第i个皇后在横轴坐标相等，
     * 或者横坐标和纵坐标之差相等（两者连线于横轴夹角为45度）如果冲突返回true，否则返回false
     * @param k 第k个皇后
     * @return 放置地k个皇后之后是否与之前的皇后冲突
     */
    public boolean willBeEaten(int k) {
```

```
        for (int i = k - 1; i > -1; i--) {// i 的起点为k-1，即k 个皇后的
上一行
            if (queen[k] == queen[i]
                    || Math.abs(queen[k] - queen[i]) == Math.abs(k - i))
 {
                return true;
            }
        }
        return false;
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        new Queens8().backtrackMethod(8);
    }
}
```

3.4.动态规划

动态规划（英语：Dynamic programming，简称 DP）是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。

那动态规划算法要表达的核心思想到底是什么？我们来看一个例子

```
A : "2+2+2+2+2=? 请问这个等式的值是多少？ "
B : "计算 ing 。。。。。。结果为 10 "
A : "那如果在等式左边写上 1+ ，此时等式的值为多少？ "
B : "quickly 结果为 11 "
A : "你怎么这么快就知道答案了"
A : "只要在 10 的基础上加 1 就行了 "
A : "所以你不用重新计算因为你记住了第一个等式的值为 10 ,动态规划算法也可以说是
'记住求过的解来节省时间'
```

由上可知：动态规划算法的核心就是记住已经解决过的子问题的解；而记住求解的方式有两种：**①自顶向下的备忘录法 ②自底向上。**

我们先来看一个最简单的例子，我们曾经求解过的斐波拉契数列 **Fibonacci**。

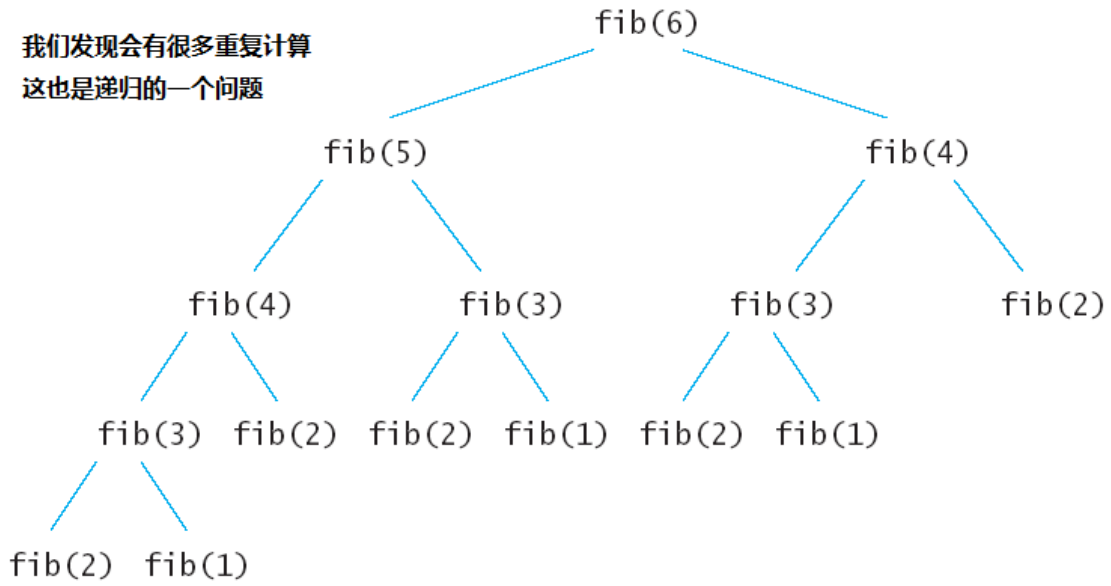
```
Fibonacci (1) = 1;
Fibonacci (2) = 1;
Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2);其中 n >=3
```

对于斐波拉契数列的实现我们一般用递归方式进行实现

```
public int fib(int n){
    if(n <= 1 ){
```

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

实现起来很简单，我们来分析一下这个递归算法的执行过程，假设 n=6，则计算机递归执行如下：



通过图发现有很多节点被重复执行，那如何来解决这个问题，如果在执行的时候把执行过的子节点保存起来，后面要用到的时候直接查表调用的话可以节约大量的时间。下面就看看动态规划的两种方法怎样来解决斐波拉契数列 **Fibonacci** 数列问题。

3.4.1、自顶向下备忘录法

```
public static int fibonacci(int n){
        if(n<=0) {
            return -1;
        }
        //创建备忘录
        int [] memo=new int[n+1];
        for(int i=0;i<=n;i++){
            memo[i]=-1;
        }
        return fib(n, memo);
    }
    public static int fib(int n,int [] memo){
        //如果已经求出了fib（n）的值直接返回
```

```
        if(memo[n]!=-1) {
            return memo[n];
        }
        //否则将求出的值保存在 memo 备忘录中。
        if(n<= 2){
            memo[n]=1;
        }else {
            memo[n]=fib( n-1,memo)+fib(n-2,memo);
        }
        return memo[n];
    }

    @Test
    public void test1(){
        System.out.println(fibonacci(6));
    }
```

3.4.2、自底向上的动态规划

备忘录法是利用了递归，上面算法不管怎样，计算 fib（6）的时候最后还是要计算出 fib（1），fib（2），fib（3）……,那么何不先计算出 fib（1），fib（2），fib（3）……,呢？这也就是**动态规划的核心，先计算子问题，再由子问题计算父问题。**

```
public static int fib(int n){
        if(n<=0){
            return -1;
        }
        //创建备忘录
        int [] memo=new int[n+1];
        memo[0]=0;
        memo[1]=1;
        //自底向上 先求解子问题 由子问题求解父问题
        for(int i=2;i<=n;i++){
            memo[i]=memo[i-1]+memo[i-2];
        }
        return memo[n];
    }
```

自底向上方法也是利用数组保存了先计算的值，方便后续的调用。如果我们仔细观察，参与循环的只有 i，i-1 ,i-2 三项，因此该方法还可以继续优化如下，你可能意向不到

```
 public static int fib(int n)
    {
        if(n<=1) {
            return -1;
        }

        int memo_i_2=0;
```

```
        int memo_i_1=1;
        int memo_i=1;
        for(int i=2;i<=n;i++)
        {
            memo_i   = memo_i_2 + memo_i_1;
            memo_i_2 = memo_i_1;
            memo_i_1 = memo_i;
        }
        return memo_i;
    }
```

是不是很神奇呢？

但是如果你认为动态规划就仅仅如此吗？那你就想的太简单了，下面我们从算法导论一书中摘抄出一个例子如下：

假定我们知道 Serling 公司出售一段长度为 i 英寸的钢条的价格为 p_i（$i=1, 2, \cdots$，单位为美元）。钢条的长度均为整英寸。图 15-1 给出了一个价格表的样例。

长度 i	1	2	3	4	5	6	7	8	9	10
价格 p_i	1	5	8	9	10	17	17	20	24	30

图 15-1　钢条价格表样例。每段长度为 i 英寸的钢条为公司带来 p_i 美元的收益

钢条切割问题是这样的：给定一段长度为 n 英寸的钢条和一个价格表 p_i（$i=1, 2, \cdots, n$），求切割钢条方案，使得销售收益 r_n 最大。注意，如果长度为 n 英寸的钢条的价格 p_n 足够大，最优解可能就是完全不需要切割。

我们发现，将一段长度为 4 英寸的钢条切割为两段各长 2 英寸的钢条，将产生 $p_2+p_2=5+5=10$ 的收益，为最优解。

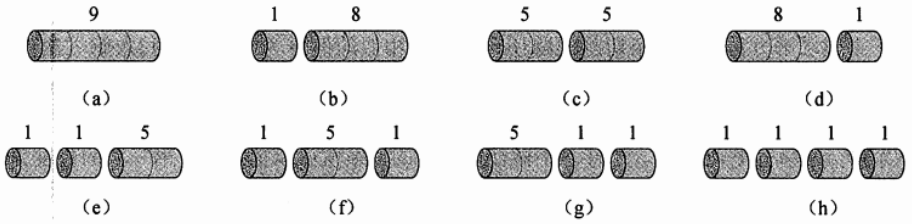


图 15-2　4 英寸钢条的 8 种切割方案。根据图 15-1 中的价格表，在每段钢条之上标记了它的价格。最优策略为方案(c)——将钢条切割为两段长度均为 2 英寸的钢条——总价值为 10

更一般地，对于 $r_n(n\geqslant 1)$，我们可以用更短的钢条的最优切割收益来描述它：

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1) \tag{15.1}$$

第一个参数 p_n 对应不切割，直接出售长度为 n 英寸的钢条的方案。其他 $n-1$ 个参数对应另外 $n-1$ 种方案：对每个 $i=1, 2, \cdots, n-1$，首先将钢条切割为长度为 i 和 $n-i$ 的两段，接着求解这两段的最优切割收益 r_i 和 r_{n-i}（每种方案的最优收益为两段的最优收益之和）。由于无法预知哪种方案会获得最优收益，我们必须考察所有可能的 i，选取其中收益最大者。如果直接出售原钢条会获得最大收益，我们当然可以选择不做任何切割。

关子问题的最优解，并在所有可能的两段切割方案中选取组合收益最大者，构成原问题的最优解。我们称钢条切割问题满足**最优子结构**(optimal substructure)性质：问题的最优解由相关子问题的最优解组合而成，而这些子问题可以独立求解。

除了上述求解方法外，钢条切割问题还存在一种相似的但更为简单的递归求解方法：我们将钢条从左边切割下长度为 i 的一段，只对右边剩下的长度为 $n-i$ 的一段继续进行切割(递归求解)，对左边的一段则不再进行切割。即问题分解的方式为：将长度为 n 的钢条分解为左边开始一段，以及剩余部分继续分解的结果。这样，不做任何切割的方案就可以描述为：第一段的长度为 n，收益为 p_n，剩余部分长度为 0，对应的收益为 $r_0=0$。于是我们可以得到公式(15.1)的简化版本：

$$r_n = \max_{1\leqslant i\leqslant n}(p_i + r_{n-i}) \tag{15.2}$$

前面我们也讲了动态规划的几种做法：我们依次来实现一下

递归实现：

```
/**
     * 钢条切割问题
     * @param p 钢条长度和价格对应表
     * @param n 钢条长度
     * @return
     */
    public static int cutSteelBar(int [] p,int n)
    {
        if(n==0){
            return 0;
        }
        int q=Integer.MIN_VALUE;
        for(int i=1;i<=n;i++) {
            q=Math.max(q, p[i-1]+cutSteelBar(p, n-i));
        }
        return q;
    }
```

备忘录实现：

```
/**
     * 钢条切割:备忘录实现
     * @param p
     * @return
     */
    public static int cutBydemo(int []p)
    {
        int []r=new int[p.length+1];
        for(int i=0;i<=p.length;i++)
            r[i]=-1;
        return cut(p, p.length, r);
    }
    public static int cut(int []p,int n,int []r)
    {
        int q=-1;
        if(r[n]>=0)
            return r[n];
        if(n==0)
            q=0;
        else {
            for(int i=1;i<=n;i++)
                q=Math.max(q, cut(p, n-i,r)+p[i-1]);
        }
        r[n]=q;

        return q;
    }
```

备忘录实现不难理解，但是这道钢条切割问题的经典之处在于自底向上的动态规划问题的处理，理解了这个也就理解了动态规划的精髓。

自底向上实现：

```
/**
     * 钢条切割:自底向上实现
     * @param p
     * @return
     */
    public static int buttom2up_cut(int []p)
    {
        int [] r=new int[p.length+1];
        for(int i=1;i<=p.length;i++)
        {
            int q=-1;
            // 动态规划的核心
            for(int j=1;j<=i;j++)
                q=Math.max(q, p[j-1]+r[i-j]);
            r[i]=q;
        }
```

```
        return r[p.length];
    }
```

自底向上的动态规划问题中最重要的是理解代码中的第二层循环，这里外面的循环是求 r[1],r[2]……，里面的循环是求出 r[1],r[2]……的最优解，也就是说 r[i]中保存的是钢条长度为 i 时划分的最优解，这里面涉及到了最优子结构问题，也就是一个问题取最优解的时候，它的子问题也一定要取得最优解。

3.4.3、动态规划的适用场景

1：具有最优子结构性质的问题：指的是问题的最优解包含子问题的最优解。反过来说就是，我们可以通过子问题的最优解，推导出问题的最优解。如果我们把最优子结构，对应到我们前面定义的动态规划问题模型
上，那我们也可以理解为，后面阶段的状态可以通过前面阶段的状态推导出来

2：无后效性：无后效性有两层含义，第一层含义是，在推导后面阶段的状态的时候，我们只关心前面阶段的状态值，不关心这个状态是怎么一步一步推导出来的。第二层含义是，某阶段状态一旦确定，就不受之后阶段的决策影响，也就是说某状态以后的过程不会影响以前的状态，只与当前状态有关。

2：具有重叠子问题的问题：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）；通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，从而减少计算量：一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。这种做法在重复子问题的数目关于输入的规模呈指数增长时特别有用。

动态规划比较适合用来求解最优问题，比如求最大值、最小值等等。它可以非常显著地降低时间复杂度，提高代码的执行效率。不过，它也是出了名的难学。它的主要学习难点跟递归类似，那就是，求解问题的过程不太符合人类常规的思维方式。

3.4.4、动态规划的经典案例模型

这节我们来说明一下动态规划常见的案例模型，并抛出几个问题，大家可以利用我们学习过的知识来分析一下。

（1）、线性模型

线性模型的是动态规划中最常用的模型，上面的钢条切割问题就是经典的线性模型，这里的线性指的是状态的排布是呈线性的。我们以一个面试题为例进行说明

面试题：在一个夜黑风高的晚上，有 n（n <= 50）个小朋友在桥的这边，现在他们需要过桥，但是由于桥很窄，每次只允许不大于两人通过，他们只有一个手电筒，所以每次过桥的两个人需要把手电筒带回来，i 号小朋友过桥的时间为 T[i]，两个人过桥的总时间为二者中时间长者。问所有小朋友过桥的总时间最短是多少？

（2）、区间模型：

区间模型的状态表示一般为 di，表示区间[i, j]上的最优解，然后通过状态转移计算出[i+1, j]或者[i, j+1]上的最优解，逐步扩大区间的范围，最终求得[1, len]的最优解。

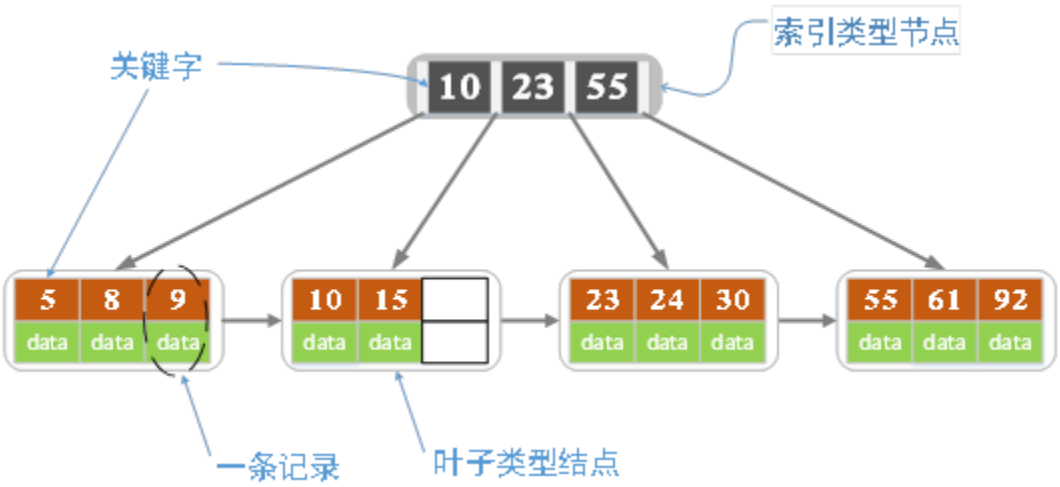
面试题 给定一个长度为 n（n <= 1000）的字符串 A，求插入最少多少个字符使得它变成一个回文串。

（3）、背包模型

背包问题是动态规划中一个最典型的问题之一，例如：对于一组不同重量、不可分割的物品，我们需要选择一些装入背包，在满足背包最大重量限制的前提下，背包中物品总重量的最大值是多少呢？

4：B+树

4.1、B+树的定义



各种资料上 B+树的定义各有不同，一种定义方式是关键字个数和孩子结点个数相同。这里我们采取维基百科上所定义的方式，即关键字个数比孩子结点个数小 1，这种方式是和 B 树基本等价的。上图就是一颗阶数为 4 的 B+树。

除此之外 B+树还有以下的要求。

1）B+树包含 2 种类型的结点：内部结点（也称索引结点）和叶子结点。根结点本身即可以是内部结点，也可以是叶子结点。根结点的关键字个数最少可以只有 1 个。

2）B+树与 B 树最大的不同是内部结点不保存数据，只用于索引，所有数据（或者说记录）都保存在叶子结点中。

3） m 阶 B+树表示了内部结点最多有 m-1 个关键字（或者说内部结点最多有 m 个子树），阶数 m 同时限制了叶子结点最多存储 m-1 个记录。

4）内部结点中的 key 都按照从小到大的顺序排列，对于内部结点中的一个 key，左树中的所有 key 都小于它，右子树中的 key 都大于等于它。叶子结点中的记录也按照 key 的大小排列。

5）每个叶子结点都存有相邻叶子结点的指针，叶子结点本身依关键字的大小自小而大顺序链接。

B+树优点：

1、单次请求涉及的磁盘 IO 次数少（出度 d 大，且非叶子节点不包含表数据，树的高度小）；

2、查询效率稳定（任何关键字的查询必须走从根结点到叶子结点，查询路径长度相同）；

3、遍历效率高（从符合条件的某个叶子节点开始遍历即可）

B+树缺点：

B+树最大的性能问题在于会产生大量的随机 IO，主要存在以下两种情况：

1、主键不是有序递增的，导致每次插入数据产生大量的数据迁移和空间碎片；

2、即使主键是有序递增的，大量写请求的分布仍是随机的；

5：今日课程总结与作业安排

5.1.今日总结：

今天的课程第一部分主要是算法思想，主要有

1：贪心

2：分治

3：回溯

4：动态规划

第二部分主要讲解了一个 B+树的数据结构，分析了数据库索引底层的一个实现原理。

5.2.作业安排：

1：理解四大算法思想

2：完成动态规划部分三种模型中分别留下的题目

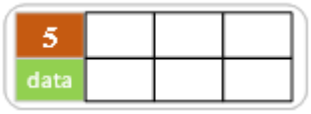
【附录】B+树

B+树的插入操作

1）若为空树，创建一个叶子结点，然后将记录插入其中，此时这个叶子结点也是根结点，插入操作结束。
2）针对叶子类型结点：根据 key 值找到叶子结点，向这个叶子结点插入记录。插入后，若当前结点 key 的个数小于等于 m-1，则插入结束。否则将这个叶子结点分裂成左右两个叶子结点，左叶子结点包含前 m/2 个记录，右结点包含剩下的记录，将第 m/2+1 个记录的 key 进位到父结点中（父结点一定是索引类型结点），进位到父结点的 key 左孩子指针向左结点,右孩子指针向右结点。将当前结点的指针指向父结点，然后执行第 3 步。
3）针对索引类型结点：若当前结点 key 的个数小于等于 m-1，则插入结束。否则，将这个索引类型结点分裂成两个索引结点，左索引结点包含前(m-1)/2 个 key，右结点包含 m-(m-1)/2 个 key，将第 m/2 个 key 进位到父结点中，进位到父结点的 key 左孩子指向左结点, 进位到父结点的 key 右孩子指向右结点。将当前结点的指针指向父结点，然后重复第 3 步。

下面是一颗 5 阶 B 树的插入过程，5 阶 B 数的结点最少 2 个 key，最多 4 个 key。

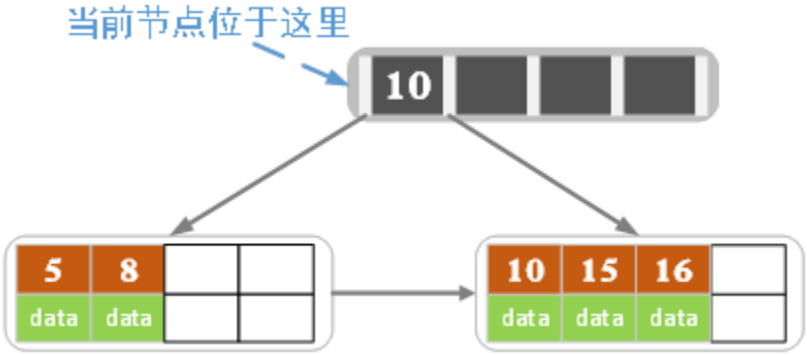
a）空树中插入 5



b）依次插入 8，10，15

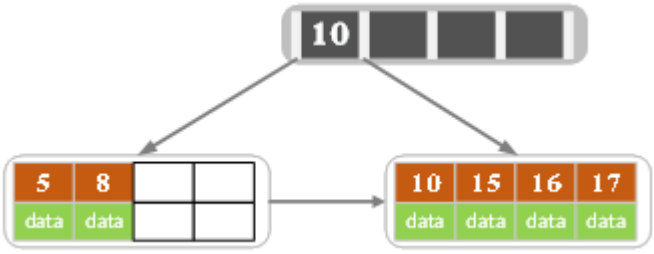
c）插入 16

插入 16 后超过了关键字的个数限制，所以要进行分裂。在叶子结点分裂时，分裂出来的左结点 2 个记录，右边 3 个记录，中间 key 成为索引结点中的 key，分裂后当前结点指向了父结点（根结点）。结果如下图所示。

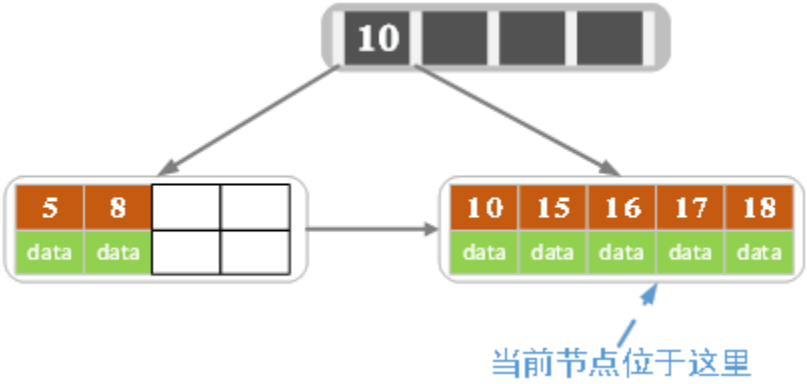


当然我们还有另一种分裂方式，给左结点 3 个记录，右结点 2 个记录，此时索引结点中的 key 就变为 15。

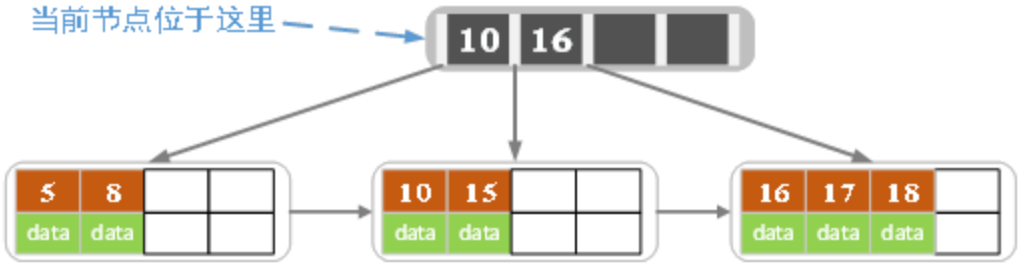
d）插入 17



e）插入 18，插入后如下图所示

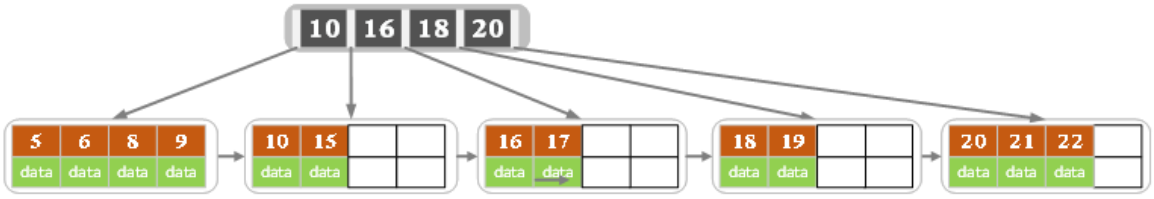


当前结点的关键字个数大于 5，进行分裂。分裂成两个结点，左结点 2 个记录，右结点 3 个记录，关键字 16 进位到父结点（索引类型）中，将当前结点的指针指向父结点。

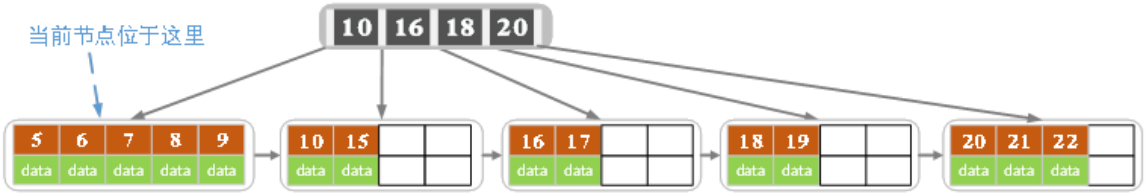


当前结点的关键字个数满足条件，插入结束。

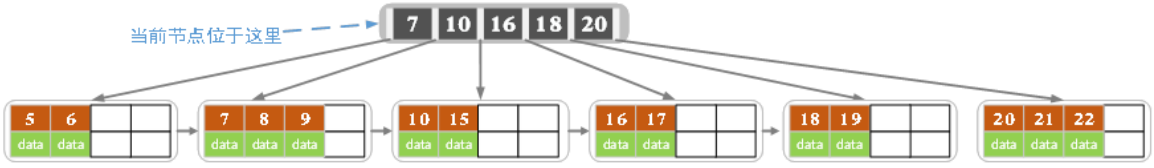
f）插入若干数据后



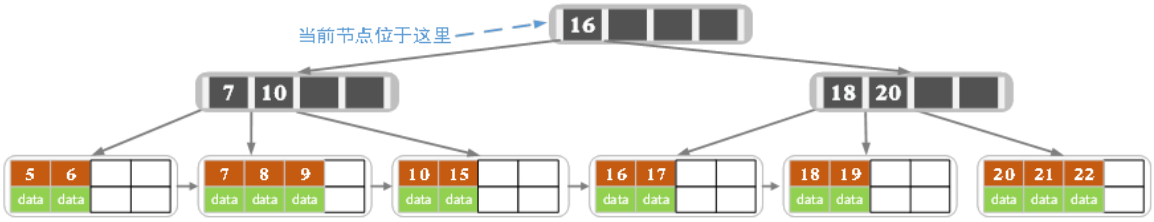
g）在上图中插入 7，结果如下图所示



当前结点的关键字个数超过 4，需要分裂。左结点 2 个记录，右结点 3 个记录。分裂后关键字 7 进入到父结点中，将当前结点的指针指向父结点，结果如下图所示。



当前结点的关键字个数超过 4，需要继续分裂。左结点 2 个关键字，右结点 2 个关键字，关键字 16 进入到父结点中，将当前结点指向父结点，结果如下图所示。



当前结点的关键字个数满足条件，插入结束。