此章节回答了

1、如何定义动态规划数组的含义?

2、如何设计动态规划转移方程?

动态规划:

寻找状态和选择->定义dp数组和函数定义->寻找状态之间的关系

进行动态规划设计就是通过数学归纳法来解决:

1、因此我们可以问自己先假设dp[0.....n-1]已经计算出来那么dp[n]如何得出呢?

2、

例题1:

最长递增子序列:

> 注意:子序列不一定是连续的但是子串必须是连续的
> step1:先定义dp数组的含义
> 此题目dp[i]表示以nums[i]为结尾的最长递增子序列长度
> **step2:假设当我们已经知道了dp[0....n-1]那么dp[n]的值会是多少呢?**
> nums[n]=m,那么只要找到前面结尾小于等于m的子序列，然后把m接到其后面就会形成新的子序列，然后从新的子序列中寻找长度最长的子序列，**dp=|最长新子序列的长度|**
> step3:最终结果是dp数组经过何种变法得出结果呢?
> 最终结果就是**dp数组中的最大值即为最后的结果**

```
int lengthOfLIS(int[] nums){
 int dp[nums.length];//且初始值都设为1
 int res=1;
 for(int i=0;i<nums.length;i++){
 for(int j=0;j<i;j++){
 if(nums[i]>=nums[j]){
 dp[i]=max(dp[i],dp[j]+1);
 }
 }
  res=max(res,dp[i]);
  }
  return res;
}
```

优化解法--二分搜索的解法

与一种patience game有关有一种**patient sort(耐心排序)**

> **耐心纸牌游戏**:如果有一堆纸牌，则遍历该堆纸牌， I 抽出的纸牌必须要存放在点数大的纸牌上方，如果抽出的纸牌点数较大则创建新堆，如果当前牌有多个堆可以选择时，则选择最左边的牌堆在这种情况下，牌的堆数就是**最长递增子序列的长度**
> **可以利用左侧搜索找到抽出的牌找出合适的堆位置**
> **为什么采用左侧搜索呢?** **因为左侧搜索的结果为堆顶点数小于抽出牌的牌数则正好满足条件 I**

```java
int lengthOfLIS(int[] nums){
 int[] top=new int[nums.length];
 int piles=0;//表示堆数
 for(int i=0;i<nums.length;i++){
 int poker=nums[i];

 /******左侧边界搜索即向左减小搜索区间算法，采用左闭右开的方式*******/
 int left=0;
 int right=piles;
  while(left<right){
 int mid=left+(right-left)/2;
 if(nums[mid]==poker){
 right=mid;
 }else if(nums[mid]<poker){
 left=mid+1;
 }else if(nums[mid]>poker){
 right=mid;
 }
 }
 //此时如果现有的堆顶的点数都小于poker那么新建一个堆
 if(left==piles) top[++piles]=poker;
 }
 return piles;
}
```

例题2二维递增子序列:信封嵌套问题

> 问题:给出一些信封，每个信封用宽度和高度的整数对形式(w,h)表示,当一个信封A的宽度和高度都比另一个信封B大的时候,则B就可以放进A里.请计算最多有多少个信封能组成一组"俄罗斯套娃"信封?
> 思想:题目中是有两个维度的比较，但是我们可以先解决一个维度然后另一个维度采用最长递增子序列来解决即可
> 主要核心思路:先对宽度w进行升序排序，如果遇到相同的宽度w则按照高度h进行降序排序。之后把所有的h作为一个数组，在这个数组上计算出LIS
> **为什么要对h进行逆序排序?**
> **因为两个宽度w相同的信封不能互相包含，w相同时将h逆序排序，则保证w相同的信封至多只有一个会被加入递增序列。**

例题3最大子数组之和

> 给出一个数组nums,里面的元素可能为正数也可能为负数,给出最大的子数组之和
> 思路：最大的子数组之和即子数组为连续

```
step1：定义dp数组
dp[i]表示以nums[i]结尾的最大子数组之和
step2：dp[i]=max(nums[i],dp[i-1]+nums[i]);
```

```
4
5  res=dp[0]=nums[0];
6  for(int i=1;i<n;i++){
7   dp[i]=max(dp[i-1]+nums[i],nums[i]);
8   res=max(dp[i],res);
9  }
10
11  又因为状态i只与状态i-1有关因此可以用状态压缩
12  int res=i=i_1=nums[0];
13  for(int j=1;j<n;j++){
14   i=max(nums[j],i_1+nums[j]);
15   i_1=i;
16   res=max(i,res);
17  }
```

例题4最长公共子序列(二维dp数组)

> step1:dp[i][j]:表示对于s1[0....i-1]和s2[0....j-1]，他们的最长公共子序列为dp[i][j]
> step2:
> 状态转移方程为
> if(str1[i]==str2[j])dp[i][j]=dp[i-1][j-1]+1;
> else dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
> 对于s1[i]!=s2[j]疑问?

```
1  code:
2  dp[0][0];
3  for(int i=0;i<str1.length;i++) dp[i][0]=0;
4  for(int j=0;j<str2.length;j++) dp[0][j]=0;
5
6  for(int i=1;i<str1.length;i++){
7   for(int j=1;j<st2.length;j++){
8   if(str1[i]==str2[j]) dp[i][j]=dp[i-1][j-1]+1;
9   else dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
10   }
11  }
```

例题5：编辑距离算法****（ispeak开发就是这个算法）

> 对于字符串可以进行三种操作分别为:**插入一个字符，删除一个字符，替换一个字符**
> **现在给你两个字符串s1和s2，请计算将s1转换成s2最少需要多少次操作?**

```
对于S1变为S2过程中可能会有4种操作
即S1什么都不做，S1中删除字符，S1中替换字符，S1中插入字符
int minDistance(int i,int j){
 if(i==-1) return j+1;
 if(j==-1) return i+1;
 if(s1[i]==s2[j]){
 return dp(i-1,j-1);
 }else{
 return min(
  dp(i-1,j)+1,//删除字符
  dp(i-1,j-1)+1,//替换字符
  dp(i,j-1)+1//插入字符
  )
  }
 }
dp(i-1,j)+1
 解释：把s1[i]字符删除，然后i指针往前移动一位与s2[j]进行匹配,由于此时发生了删除操作所以距离加1
dp(i-1,j-1)+1
 解释:把s1[i]替换成s2[j],此时s1[i]与s2[j]已发生匹配成功，所以i,j都均向前移动一位
 此时发生替换操作所以+1
dp(i,j-1)+1
 解释:在s1[i]字符后面加入一个s2[j]与s2[j]发生匹配所以j指针向后移动一位
 此时发生了插入操作所以+1
```

此时把递归解法改写为迭代写法，区别在于迭代写法是自底向上解法，递归在于自顶向下解法

```
所有的思想都从s1转换为s2考虑
int minDistance(string s1,string s2){
 int m=s1.length,n=s2.length;
 int[][] dp=new int[m+1][n+1];
 dp[0][0]=0;
 for(int i=1,i<=m;i++)dp[i][0]=i;//s1字符串删除i个字符
 for(int j=1;j<=n;j++)dp[0][j]=j;//s1字符串插入j个字符
 for(int i=1;i<=m;i++){
 for(int j=1;j<=n;j++){
  if(s1[i-1]==s2[j-1])
  dp[i][j]=dp[i-1][j-1];
  else{
  dp[i][j]=min(
  dp[i-1][j]+1,//删除操作
  dp[i][j-1]+1,//插入操作
```

```
  dp[i-1][j-1]+1,//替换操作
  )
  }
  }
  }
  return dp[m][n];
}
```

如何记录编辑的具体操作呢?

可以增加一个数据结构

```
class Node{
   int val;
   int choice;
/**
0:代表啥都不做
1:代表插入
2:代表删除
3:代表替换
**/
}
```

```
在原有的代码部分增加记录具体操作choice的代码即可
Node minNode(Node a,Node b,Node c){
 Node res=new Node(a.val,1);
 if(res.val>b.val){
 res.val=b.val;
 res.choice=b.choice;
 }
 if(res.val>c.val){
 res.val=c.val;
  res.choice=c.choice;
  }
 }
 接下来是打印具体操作的部分
 void printResult(Node dp[][],string s1,strin s2){
  int i=s1.length,j=s2.length;
  Stack<int> stack;
  while(i!=0||j!=0){
  stack.push_back(dp[i][j].choice);
  switch(dp[i][j].choice){
  case 0:i--;j--;break;
  case 1:j--;break;
  case 2:i--;break;
  case 3:i--;j--;
```

```
  }
  }
  while(!stack.isEmpty()){
  cout<<stack.top()<<endl;
  stack.pop();
  }
}
```

总结：有些算法问题需要进行对原来数据进行排序然后才能用现有算法解决

动态规划答疑：最优子结构及dp遍历方向

概念：

1、最优子结构:可以从子问题的最优结果里面推出最优结果

条件:**子问题**之间要**互相独立**

思想：**从简单的basecase往后推导，以小博大**

2、dp数组的遍历方向

原则：

1、遍历过程中，**所需的状态**必须是已经计算出来的

2、**遍历的终点**必须是**存储结构的位置**

遍历的方式

```
正向遍历：
for(int i=0;i<m;i++){
 for(int j=0;j<n;j++){

 }
}

反向遍历：
for(int i=m-1;i>=0;i--){
  for(int j=n-1;j>=0;j--){

  }
 }

 斜向遍历：
 从对角线开始遍历
 for(int l=1;l<=n;l++){
```

```
 for(int i=0;i<=n-l;i++){
 int j=l+i-1;
 dp[i][j]=.....
 }
}

```