

电子科技大学计算机科学与工程学院

# 标准实验报告

(实验) 课程名称 数据结构与算法

电子科技大学教务处制表

电子科技大学

# 实验报告 1

学生姓名：唐以恒      学 号：20171108101017      指导教师：董强

一、实验时间和地点：2019 年 10 月 20 日 09:00-12:00，主楼 A2-412

二、实验名称：

十进制数到 N 进制数的转换

三、实验目的：

掌握顺序栈的入栈出栈等基本操作的编程实现

四、实验内容：

利用顺序栈实现将从键盘输入的十进制数转换为 N（如二进制、八进制、十六进制）进制数据。

五、实验原理：

转换方法利用除留余数法。所转换的 N 进制数按低位到高位顺序产生，而通常的输出是从高位到低位的，恰好与计算过程相反，因此转换过程中每得到一位 N 进制数则进栈保存，转换完毕后依次出栈则正好是转换结果。

六、实验器材（设备、元器件）：PC

七、实验步骤：

先设计一个栈结构的类 Stack.java，用于实现 LIFO 存取。

```
1.import java.util.ArrayList;
2.import java.util.List;
3.
4.public class Stack<E> {
5.    List<E> list;
6.    //initial
7.    protected Stack () {
8.        list=new ArrayList<E>();
9.    }
10.    //push
11.    protected void push(E e) {
12.        list.add(e);
13.    }
14.    //peek
```

```

15.  protected E peek() {
16.      return list.get(list.size()-1);
17.  }
18.  //pop
19.  protected E pop() throws Exception{
20.      if(empty()) throw new Exception("stack is empty!");
21.      E ret=list.get(list.size()-1);
22.      list.remove(list.size()-1);
23.      return ret;
24.  }
25.  //isEmpty
26.  protected boolean empty() {
27.      return list.size()==0;
28.  }
29. }

```

再在设计一个将 10 进制数转换为 2~32 进制数的工具类 DecimalToNDigit.java  
 对于十进制数的整数部分，采用保留余数法，放入栈中，逆序输出。  
 对于十进制数的小数部分，采用乘法保留整数。

```

1. public class DecimalToNDigit {
2.     private double num; //要转换的十进制数
3.     private int radix; //指定进制
4.     public DecimalToNDigit(double num, int radix) {
5.         this.num=num;
6.         this.radix=radix;
7.     }
8.     public String convert() throws Exception{
9.         int num_integer = (int)num; //整数部分
10.        double num_decimal = num % 1; //小数部分
11.        StringBuilder ans=new StringBuilder(); //目标字符串
12.        Stack<Character> helper=new Stack<>();
13.        //转换整数部分
14.        while(num_integer / radix != 0) {
15.            helper.push(
16.                Character.forDigit(
17.                    num_integer % radix, radix));
18.            num_integer /= radix;
19.        }
20.        helper.push(Character.forDigit(num_integer, radix));
21.        while(!helper.empty()) {
22.            ans.append(helper.pop());
23.        }

```

```

24. //转换小数部分
25. if(num_integer!=0) {
26.     ans.append('.');
27.     while(num_decimal!=0) {
28.         num_decimal *= radix;
29.         ans.append(Character.forDigit((int)num_decimal, radix));
30.         num_decimal %= 1;
31.     }
32. }
33. return ans.toString();
34. }
35. }

```

最后写一个测试代码 Test.java, 对输入的十进制数进行转化。

```

1.import java.util.Scanner;
2.public class Test {
3.    public static void main(String[] args) throws Exception{
4.        Scanner scanner=null;
5.        try {
6.            scanner=new Scanner(System.in);
7.            System.out.println("请输入你想转换的十进制数: ");
8.            double num=scanner.nextDouble();
9.            System.out.println("请输入你想转换的进制: ");
10.           int radix=scanner.nextInt();
11.           System.out.println(" 转 换 结 果
是:"+new DecimalToNDigit(num, radix).convert());
12.        }catch (Exception e) {
13.            e.printStackTrace();
14.        }finally {
15.            scanner.close();
16.            scanner=null;
17.        }
18.    }
19. }

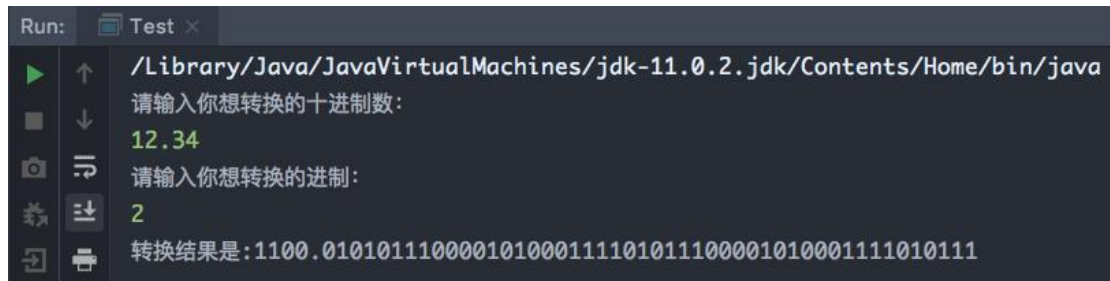
```

## 八、实验数据及结果分析:

测试数据: 12.34

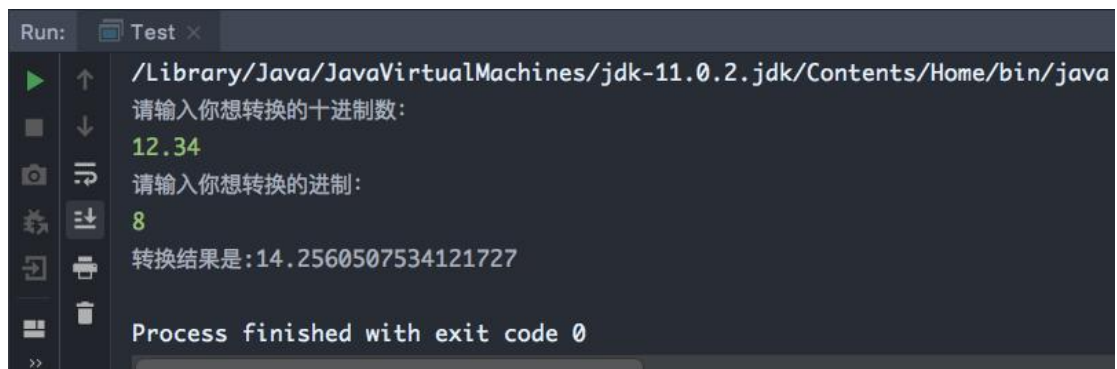
测试结果:

2 进制:



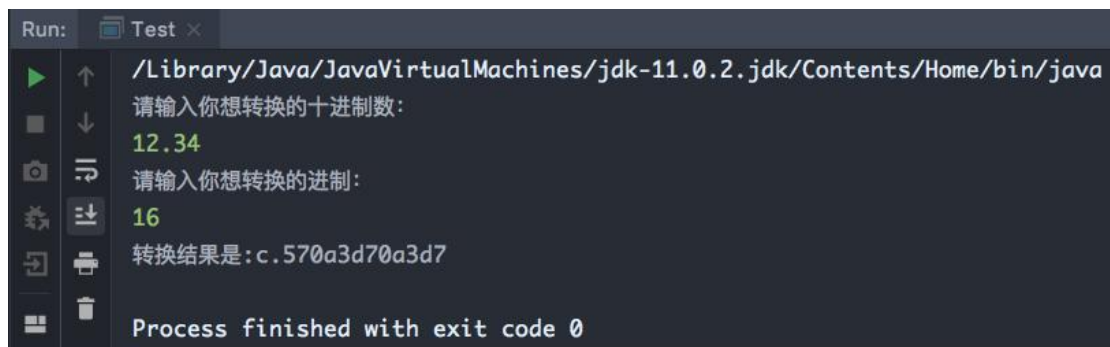
```
Run: Test x
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bin/java
请输入你想转换的十进制数:
12.34
请输入你想转换的进制:
2
转换结果是:1100.0101011110000101000111101011100001010001111010111
```

8 进制:



```
Run: Test x
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bin/java
请输入你想转换的十进制数:
12.34
请输入你想转换的进制:
8
转换结果是:14.2560507534121727
Process finished with exit code 0
>>
```

16 进制:



```
Run: Test x
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bin/java
请输入你想转换的十进制数:
12.34
请输入你想转换的进制:
16
转换结果是:c.570a3d70a3d7
Process finished with exit code 0
```

## 九、实验结论:

基本完成了 10 进制数(含小数)对 2~32 进制数的转化。

## 十、总结及心得体会:

用线性表自定义栈结构,并进行进制数的转换,主要是利用了栈的后进先出的特性,实现逆序输出。

## 十一、对本实验过程及方法、手段的改进建议:

报告评分:

指导教师签字:

# 实验报告 2

学生姓名：唐以恒      学 号：2017110801017      指导教师：董强

一、实验时间和地点：2019 年 10 月 20 日 09:00-12:00，主楼 A2-412

二、实验名称：

用户偏好相似度计算

三、实验目的：

掌握分治算法的原理以及合并排序算法的编程实现

四、实验内容：

给定某音乐网站给出的大众用户对于  $n$  首歌曲的喜好程度排序，以及某个特定用户  $X$  对于这  $n$  首歌曲的喜好程度排序，计算用户  $X$  与大众用户的偏好相似度。

五、实验原理：

以大众用户的排序为标准，利用分治算法来计算用户  $X$  排序序列中的逆序数量，数值越低则用户  $X$  的偏好月接近于大众偏好。具体来说，将用户  $X$  的排序序列等分为左右两个子序列，递归地计算这两个子序列中的逆序数量，再加上一个元素在左子序列、一个元素在右子序列的逆序数量，即得原始序列中的逆序数量。

六、实验器材（设备、元器件）：PC

七、实验步骤：

设计一个工具类 ComputeUserSimilarity.java，用于计算个人用户给出的歌曲喜好程度序列与大众的相似度，采用分治递归的思想，对序列进行归并排序，同时计算逆序对个数。

```
1. public class ComputeUserSimilarity {
2.     private int[] privateLike; //个人对歌曲喜好程度排序
3.     private double similarity; //相似度: 1 - (逆序对数/总对数)
4.     public ComputeUserSimilarity(int[] privateLike) {
5.         this.privateLike=privateLike;
6.     }
7.     public double getSimilarity() {
8.         int n=privateLike.length;
9.         similarity=1 - compute() / (n*(n-1)/2.0);
```

```

10.     return similarity;
11. }
12. public int[] getPrivateLike() {return privateLike;}
13. //计算逆序对数
14. public int compute() {
15.     return helper(0, privateLike.length-1);
16. }
17. public int helper(int start, int end) {
18.     if(start == end) return 0;
19.     //分解, 递归
20.     int middle=(start+end) / 2;
21.     int left_similar=helper(start, middle);
22.     int right_similar=helper(middle+1, end);
23.     //合并
24.     int[] left_tmp=new int[middle-start+1];
25.     int[] right_tmp=new int[end-middle];
26.     for(int i=start; i<=end; i++) {
27.         if(i<=middle) left_tmp[i-start]=privateLike[i];
28.         else right_tmp[i-(middle+1)]=privateLike[i];
29.     }
30.     int lr_similar=0, m=0, n=0;
31.     for(int i=start; i<=end; i++) {
32.         if(m<left_tmp.length && n<right_tmp.length) {
33.             if(left_tmp[m] <= right_tmp[n]) { privateLike[i]=left_tmp[m]; m++; }
34.             else { privateLike[i]=right_tmp[n]; lr_similar+=left_tmp.length-m; n
                ++; }
35.         }
36.         else if(m==left_tmp.length) { privateLike[i]=right_tmp[n]; n++; }
37.         else if(n==right_tmp.length) { privateLike[i]=left_tmp[m]; m++; }
38.     }
39.     return left_similar+right_similar+lr_similar;
40. }
41. }

```

然后，设计一个测试类，用于对封装好的工具类进行测试。

```

1.import java.util.Scanner;
2.public class Test {
3.    public static void main(String[] args) {
4.        Scanner scanner=null;
5.        try {
6.            scanner=new Scanner(System.in);
7.            String[] nums=null;

```



```

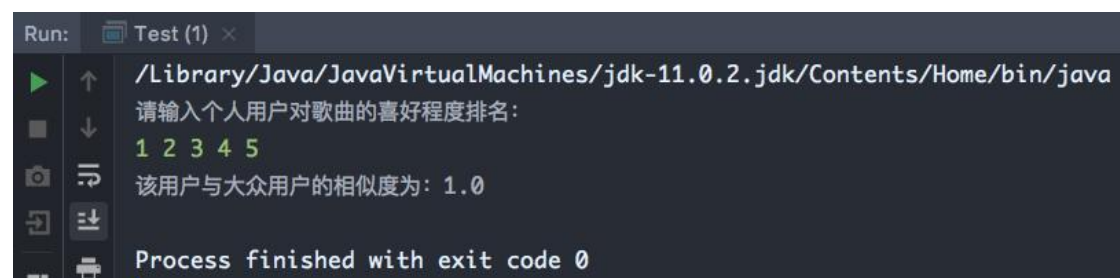
8.      System.out.println("请输入个人用户对歌曲的喜好程度排名: ");
9.      nums=scanner.nextLine().split(" ");
10.     int[] array=new int[nums.length];
11.     for(int i=0; i<nums.length; i++) {
12.         array[i]=Integer.valueOf(nums[i]);
13.     }
14.     System.out.println("该用户与大众用户的相似度为: "+
15.         new ComputeUserSimilarity(array).getSimilarity());
16. }catch (Exception e) {
17.     e.printStackTrace();
18. }finally {
19.     scanner.close();
20.     scanner=null;
21. }
22. }
23. }

```

## 八、实验数据及结果分析: g

测试数据: [1,2,3,4,5], [2,1,3,5,4], [5,4,3,2,1]

测试结果:



```

Run: Test (1) x
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bin/java
请输入个人用户对歌曲的喜好程度排名:
1 2 3 4 5
该用户与大众用户的相似度为: 1.0
Process finished with exit code 0

```



```

Run: Test (1) x
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bin/java
请输入个人用户对歌曲的喜好程度排名:
2 1 3 5 4
该用户与大众用户的相似度为: 0.8
Process finished with exit code 0

```



```

Run: Test (1) x
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bin/java
请输入个人用户对歌曲的喜好程度排名:
5 4 3 2 1
该用户与大众用户的相似度为: 0.0
Process finished with exit code 0

```

## 九、实验结论:

基本完成了对个人用户歌曲喜好程度序列与大众的相似度计算,

## 十、总结及心得体会:

通过对分治算法的理解,在数组上进行递归的排序,降低排序的时间复杂度到  $O(\log n)$ 。了解了排序的本质其实就是将逆序对调整为正序对的过程。

## 十一、对本实验过程及方法、手段的改进建议:

报告评分:

指导教师签字:

# 实验报告 3

学生姓名：唐以恒    学 号：2017110801017    指导教师：董强

一、实验时间和地点：2019 年 11 月 17 日 09:00-12:00，主楼 A2-412

二、实验名称：

最优前缀编码设计

三、实验目的：

利用贪心算法设计最优前缀编码算法的编程实现

四、实验内容：

用户输入  $n$  个正整数作为文件中  $n$  个字符出现的次数，请实现霍夫曼编码算法对该文件  $n$  个字符进行编码，并输出编码表

五、实验原理：

根据给定的  $n$  个权值  $\{w_1, w_2, \dots, w_n\}$ ，构造  $n$  棵只有根结点的二叉树。在森林中选取两棵根结点权值最小的树作左右子树，构造一棵新的二叉树，置新二叉树根结点权值为其左右子树根结点权值之和。在森林中删除这两棵树，同时将新得到的二叉树加入森林中。重复上述两步，直到只含一棵树为止，这棵树即霍夫曼树。按左 0 右 1 的规定，从根结点走到一个叶结点，完成一个字符的译码。

六、实验器材（设备、元器件）：PC

七、实验步骤：

自定义数据结构 HuffmanTree.java 用于定义霍夫曼树，构造方法为空构造和两个霍夫曼树合并构造。

```
1. public class HuffmanTree {  
2.     TreeNode root;    //根结点  
3.
```

```

4.  public HuffmanTree(char element, int weight) { //创建只有一个结点的
    huffman 树
5.      root = new TreeNode(element, weight);
6.  }
7.  public HuffmanTree(HuffmanTree t1, HuffmanTree t2) { //合并两个
    huffman 树
8.      root = new TreeNode();
9.      root.left = t1.root;
10.     root.right = t2.root;
11.     root.weight = t1.root.weight + t2.root.weight; //更新权重
12. }
13.
14. public class TreeNode {
15.     char element; //字符
16.     int weight; //权重
17.     TreeNode left;
18.     TreeNode right;
19.     public TreeNode() {}
20.     public TreeNode(char e, int w) {
21.         element = e;
22.         weight = w;
23.     }
24. }
25. }

```

创建工具类 HuffmanEncode.java 用于对输入的字符-权重映射表进行前缀编码，利用优先队列，每次弹出权重最小的两个霍夫曼树，进行合并，然后再放回优先队列中。迭代上述操作直到优先队列中只有一个霍夫曼树，这就是我们最后要得到的最优前缀编码树，然后通过 dfs 深度优先搜索将每个字符的编码构造出来。

```

1. import java.util.*;
2. /*
3. 生成霍夫曼树
4. 得到各个字符的编码
5. */
6. public class HuffmanEncode {
7.     int size; //字符的种数
8.     int cost;
9.     Map<Character, Integer> elementWeight; //字符-权重表
10.    Map<Character, String> elementEncode; //字符-前缀码表
11.    Comparator<HuffmanTree> comparator; //自定义比较器
12.    Queue<HuffmanTree> pq; //优先队列(最小堆实现)

```

```

13. HuffmanTree huffmanTree; //huffman 树
14.
15. public HuffmanEncode(char[] elements, int[] weights) {
16.     initial(elements, weights); //初始化
17.     createHuffmanTree(); //利用优先队列构造 huffman 树
18.     generateEncode(); //生成编码表
19. }
20.
21. private void initial(char[] elements, int[] weights) {
22.     elementWeight = new HashMap<>();
23.     elementEncode = new HashMap<>();
24.     comparator = new Comparator<HuffmanTree>() {
25.         @Override
26.         public int compare(HuffmanTree t1, HuffmanTree t2) {
27.             if(t1.root.weight < t2.root.weight) {
28.                 return -1;
29.             }else if(t1.root.weight > t2.root.weight) {
30.                 return 1;
31.             }else {
32.                 return 0;
33.             }
34.         }
35.     };
36.     pq = new PriorityQueue<>(comparator);
37.     size = elements.length;
38.     for(int i=0; i<size; i++) {
39.         elementWeight.put(elements[i], weights[i]);
40.     }
41. }
42.
43. private void createHuffmanTree() {
44.     for(Character element: elementWeight.keySet()) { //每个字符结点作为
        一棵 huffman 树
45.         pq.add(new HuffmanTree(element, elementWeight.get(element)));
46.     }
47.     while(pq.size() > 1) { //直到队列只有一个 huffman 树
48.         HuffmanTree t1 = pq.poll(); //每次取出来两个权重最小的两个
        huffman 树
49.         HuffmanTree t2 = pq.poll();
50.         HuffmanTree ret = new HuffmanTree(t1, t2); //合并为一个 huffman
        树
51.         pq.add(ret); //放回队列
52.     }
53.     if(pq.size() == 1) {

```

```

54.     huffmanTree = pq.poll();
55. }
56. }
57.
58. private void generateEncode() {
59.     solve(huffmanTree.root, "");
60. }
61. private void solve(HuffmanTree.TreeNode root, String tmp) { //递归搜索
    huffman 树的叶子结点
62.     if(root.left==null && root.right==null) { //叶子结点 表示字符
63.         elementEncode.put(root.element, tmp);
64.     }else {
65.         solve(root.left, tmp+"0");
66.         solve(root.right, tmp+"1");
67.     }
68. }
69. public void getCost() {
70.     cost = 0;
71.     for(Character element : elementWeight.keySet()) {
72.         cost += elementWeight.get(element) * elementEncode.get(element).len
            gth();
73.     }
74. }
75. }

```

最后，设计测试类 Test.java，对优先队列实现的霍夫曼编码树进行测试。

```

1.import java.util.Map;
2.import java.util.Scanner;
3.
4.public class Test {
5.     public static void main(String[] args) {
6.         Scanner scanner=null;
7.         try {
8.             scanner=new Scanner(System.in);
9.             System.out.println("请输入需要编码的字符序列: ");
10.            String[] chars=scanner.nextLine().split(" ");
11.            char[] elements=new char[chars.length];
12.            for(int i=0; i<elements.length; i++) {
13.                elements[i]=chars[i].charAt(0);
14.            }

```

```

15.      System.out.println("请输入字符序列的权重: ");
16.      chars=scanner.nextLine().split(" ");
17.      int[] weights=new int[chars.length];
18.      for(int i=0; i<weights.length; i++) {
19.          weights[i]=Integer.parseInt(chars[i]);
20.      }
21.      System.out.println("最优前缀编码如下: ");
22.      HuffmanEncode huffmanEncode=new HuffmanEncode(elements, weights);
23.      Map<Character, String> encodeMap=huffmanEncode.elementEncode;
24.      for(Character key : encodeMap.keySet()) {
25.          System.out.println(key+":"+encodeMap.get(key));
26.      }
27.  } catch (Exception e) {
28.      e.printStackTrace();
29.  } finally {
30.      scanner.close();
31.      scanner=null;
32.  }
33.  }
34. }

```

## 八、实验数据及结果分析:

测试数据: [1,2,3,4,5,6]

测试结果:

```

Run: Test (2) ×
/opt/idea/idea-IC-192.7142.36/jbr/bin/java
请输入需要编码的字符序列:
a b c d e f
请输入字符序列的权重:
1 2 3 4 5 6
最优前缀编码如下:
a:1010
b:1011
c:100
d:00
e:01
f:11

Process finished with exit code 0

```

## 九、实验结论:

基本完成使用贪心算法构造最优前缀编码的霍夫曼树。

## 十、总结及心得体会:

优先队列内部是通过最小堆实现的，每次从优先队列中取出权重最小的霍夫曼树，体现了贪心算法的思想。

通过最后生成的霍夫曼编码树还原各字符的编码过程中，需要从根节点遍历每个叶子结点上的路径，使用到了深度优先搜索的思想。

## 十一、对本实验过程及方法、手段的改进建议:

报告评分:

指导教师签字:



# 实验报告 4

学生姓名：唐以恒      学 号：2017110801017      指导教师：董强

一、实验时间和地点：2019 年 11 月 17 日 09:00-12:00，主楼 A2-412

二、实验名称：

矩阵连乘最优计算次序问题

三、实验目的：

利用动态规划算法设计矩阵连乘积最优加括号算法的编程实现

四、实验内容：

输入  $n + 1$  个自然数  $p_0, p_1, p_2, p_3, \dots, p_n$  作为  $n$  个矩阵  $A_i$  的维数， $A_i = (a_{mn})_{p_{i-1} \times p_i}$ ，用动态规划算法求解这  $n$  个矩阵连乘积  $A = A_1 \times A_2 \times A_3 \times \dots \times A_n$  的最优加括号方式及最少乘法次数。

五、实验原理：

设计算  $A[i:j]$  所需要的最少乘法次数  $m[i, j]$ ， $1 \leq i \leq j \leq n$ ，则原问题的最优值为  $m[1, n]$ 。当  $i = j$  时， $A[i:j] = A_i$ ，因此， $m[i, i] = 0$ ， $i = 1, 2, \dots, n$ ；当  $i < j$  时，可以递归地定义  $m[i, j]$  为： $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ ，这里  $A_i$  的维数为  $p_{i-1} \times p_i$ 。可以递归地定义  $m[i, j]$  为：

$$m[i, j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & i < j \end{cases}$$

$k$  的位置只有  $j - i$  种可能。

六、实验器材（设备、元器件）：PC

七、实验步骤：

设计工具类 MatrixChainMultiply.java 用于解决矩阵链乘法问题，找到状态转移方程，构造 dp 数组， $dp[i][j]$  表示从  $i \sim j$  的矩阵子链所需要的最小乘法次数，从长度入手，构造 dp 状态数组，最后返回  $dp[0][n-1]$

```

1. public class MatrixChainMultiply {
2.     int[] p; //规模数组
3.     int n; //矩阵链中个数
4.     int[][] dp; //保存矩阵子链的乘次数
5.     int[][] loc; //记录划分位置的数组
6.
7.     public MatrixChainMultiply(int[] p) {
8.         this.p = p;
9.         n = p.length-1;
10.        dp = new int[n+1][n+1];
11.        loc = new int[n+1][n+1];
12.        generateDP_Loc(); //构造 dp 和 loc
13.    }
14.    //构造状态数组
15.    private void generateDP_Loc() {
16.        for(int i=1; i<dp.length; i++) { //初始化
17.            for(int j=i; j<dp[i].length; j++) {
18.                if(i == j) {
19.                    dp[i][j] = 0; //矩阵链长度为 1
20.                } else {
21.                    dp[i][j] = Integer.MAX_VALUE;
22.                }
23.            }
24.        }
25.        for(int l=2; l<=n; l++) { //l 表示矩阵链的长度 矩阵链长度大于 1
26.            for(int i=1; i<=n-l+1; i++) {
27.                int j = i+l-1;
28.                //状态转移 得到使得父问题最优的对于两个子问题的划分
29.                for(int k=i; k<j; k++) {
30.                    int tmp = dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j];
31.                    if(tmp < dp[i][j]) {
32.                        dp[i][j] = tmp;
33.                        loc[i][j] = k;
34.                    }
35.                }
36.            }
37.        }
38.    }
39.    //打印状态数组
40.    void printDP() {
41.        for(int i=1; i<dp.length; i++) {
42.            for(int j=1; j<dp[i].length; j++) {
43.                System.out.printf("%-8d", dp[i][j]);
44.            }

```

```

45.         System.out.println();
46.     }
47. }
48. //打印分割数组
49. void printLOC() {
50.     for(int i=1; i<loc.length; i++) {
51.         for(int j=1; j<loc[i].length; j++) {
52.             System.out.printf("%-8d", loc[i][j]);
53.         }
54.         System.out.println();
55.     }
56. }
57. //返回最优解所需要的代价
58. int getCost() {
59.     return dp[1][n];
60. }
61. //回溯构造最优解
62. void printParens() {
63.     solve(1, n);
64. }
65. private void solve(int i, int j) {
66.     if(i == j) {
67.         System.out.print("A"+i);
68.     }else {
69.         System.out.print("(");
70.         solve(i, loc[i][j]);
71.         solve(loc[i][j]+1, j);
72.         System.out.print(")");
73.     }
74. }
75. }

```

然后定义测试类 Test.java，对工具类进行测试。

```

1. import java.util.Scanner;
2.
3. public class Test {
4.     public static void main(String[] args) {
5.         Scanner scanner=null;
6.         try {
7.             scanner=new Scanner(System.in);
8.             System.out.println("请输入矩阵链的规模: ");
9.             String[] nums=scanner.nextLine().split(" ");
10.            int[] p=new int[nums.length];

```

```

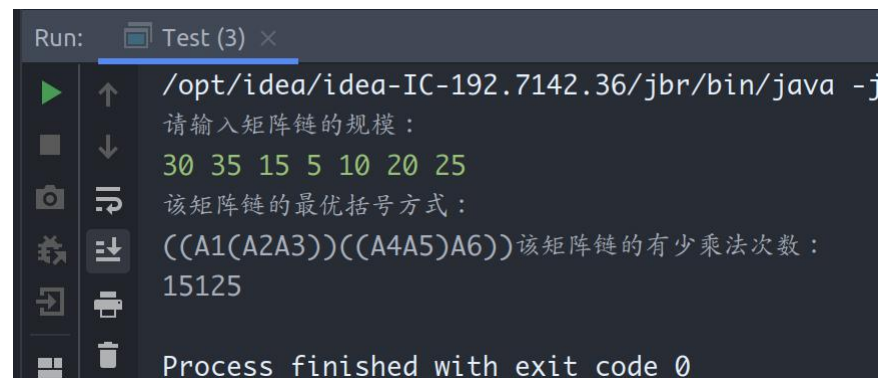
11.     for(int i=0; i<p.length; i++) {
12.         p[i]=Integer.parseInt(nums[i]);
13.     }
14.     MatrixChainMultiply matrixChain=new MatrixChainMultiply(p);
15.     System.out.println("该矩阵链的最优括号方式: ");
16.     matrixChain.printParens();
17.     System.out.println("该矩阵链的有少乘法次数: ");
18.     System.out.println(matrixChain.getCost());
19. } catch (Exception e) {
20.     e.printStackTrace();
21. } finally {
22.     scanner.close();
23.     scanner=null;
24. }
25. }
26. }

```

## 八、实验数据及结果分析:

测试数据: [30, 35, 15, 5, 10, 20, 25]

测试结果:



```

Run: Test (3) ×
/opt/idea/idea-IC-192.7142.36/jbr/bin/java -j
请输入矩阵链的规模 :
30 35 15 5 10 20 25
该矩阵链的最优括号方式 :
((A1(A2A3))((A4A5)A6))该矩阵链的有少乘法次数 :
15125
Process finished with exit code 0

```

## 九、实验结论:

用动态规划算法解决矩阵链乘法问题。

## 十、总结及心得体会:

动态规划算法的套路:

- 1.刻画一个最优解的结构特征 (最优子结构)
- 2.递归定义最优解 (递归式)
- 3.计算最优解的值 (自底向上, 表格法)
- 4.利用计算出来的信息构造一个最优解 (回溯)

通过递归式来构造 dp 状态数组。

## 十一、对本实验过程及方法、手段的改进建议:

报告评分:

指导教师签字: