# 快速排序算法

## 算法

假设对以下10个数进行快速排序：

6 1 2 7 9 3 4 5 10 8

我们先模拟快速排序的过程：首先，在这个序列中随便找一个数作为基准数，通常为了方便，以第一个数作为基准数。

**6** 1 2 7 9 3 4 5 10 8

在初始状态下，数字6在序列的第1位。我们的目标是将6挪到序列中间的某个位置，假设这个位置是k。现在就需要寻找这个k，并且以第k位为分界点，左边的数都≤ 6，右边的数都≥ 6。那么如何找到这个位置k呢？

我们要知道，快速排序其实是冒泡排序的一种改进，冒泡排序每次对相邻的两个数进行比较，这显然是一种比较浪费时间的。

而快速排序是分别从两端开始“探测”的，先从右往左找一个小于6的数，再从左往右找一个大于6的数，然后交换他们。这里可以用两个变量i和j，分别指向序列最左边和最右边。我们为这两个变量起个好听的名字“哨兵i”和“哨兵j”。刚开始的时候让哨兵i指向序列的最左边，指向数字6。让哨兵j指向序列的最右边，指向数字8。

**6** 1 2 7 9 3 4 5 10 8

i j

首先哨兵j开始出动。因为此处设置的基准数是最左边的数，所以需要让哨兵j先出动，这一点非常重要。哨兵j一步一步地向左挪动（即j = j − 1），直到找到一个小于6的数停下来。接下来哨兵i再一步一步向右挪动（即i = i + 1），直到找到一个数大于6的数停下来。最后哨兵j停在了数字5面前，哨兵i停在了数字7面前。

**6** 1 2 7 9 3 4 5 10 8

i j

现在交换哨兵i和哨兵j所指向的元素的值。交换之后的序列如下。

**6** 1 2 5 9 3 4 7 10 8

i j

到此，第一次交换结束。接下来开始哨兵j继续向左挪动（再友情提醒，每次必须是哨兵j先出发）。他发现了4 < 6，停下来。哨兵i也继续向右挪动的，他发现了9 > 6，停下来。此时再次进行交换，交换之后的序列如下。

**6** 1 2 5 4 3 9 7 10 8

i j

第二次交换结束。哨兵j继续向左挪动，他发现了3 < 6，又停下来。哨兵i继续向右移动，此时哨兵i和哨兵j相遇了，哨兵i和哨兵j都走到3面前。说明此时“探测”结束。我们将基准数6和3进行交换。交换之后的序列如下。

3 1 2 5 4 **6** 9 7 10 8

i,j

到此第一轮“探测”真正结束。现在基准数6已经归位，此时以基准数6为分界点，6左边的数都小于等于6，6右边的数都大于等于6。回顾一下刚才的过程，其实哨兵j的使命就是要找小于基准数的数，而哨兵i的使命就是要找大于基准数的数，直到i和j碰头为止。

现在我们将第一轮“探测”结束后的序列，以6为分界点拆分成两个序列，左边的序列是“3 1 2 5 4”，右边的序列是“9 7 10 8”。接下来还需要分别处理这两个序列。因为6左边和右边的序列目前都还是很混乱的。不过不要紧，我们已经掌握了方法，接下来只要模拟刚才的方法分别处理6左边和右边的序列即可。现在先来处理6左边的序列现吧。

**3** 1 2 5 4 6

重复第一轮的过程，应该得到如下序列：

2 1 **3** 5 4 6

OK，现在3已经归位。接下来需要处理3左边的序列：

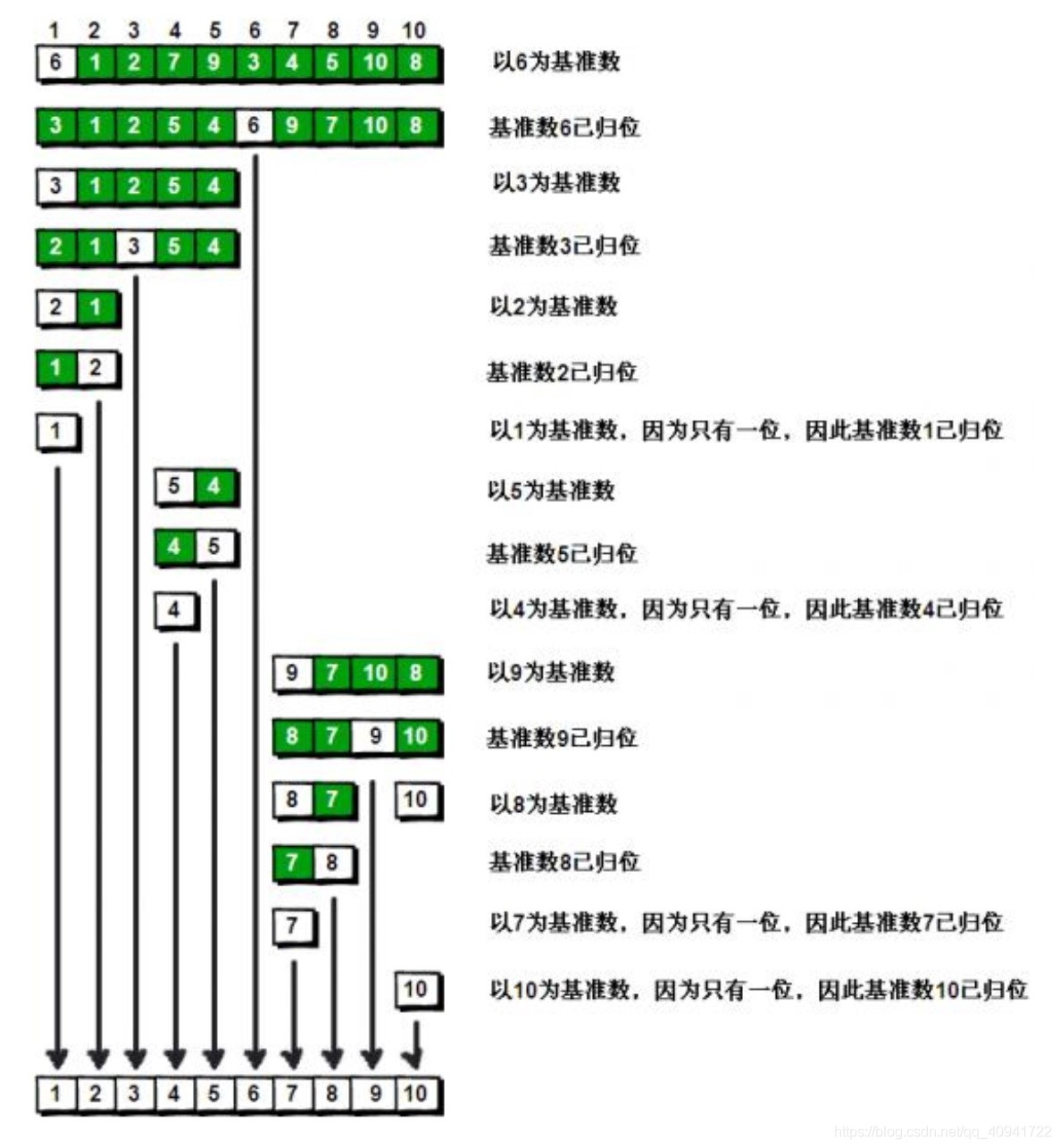
**2** 1 3 6

处理之后，2已经归位，序列“1”只有一个数，也不需要进行任何处理，因此“1”也归位。

1 2 3 6

对于基数右边的序列，采用和左边相同的过程；最终将会得到这样的序列，如下。

1 2 3 4 5 6 7 8 9 10

****细心的同学可能已经发现，快速排序的每一轮处理其实就是将这一轮的基准数归位，直到所有的数都归位为止，排序就结束了。接下来用图示的方法来展示完整的过程：

快速排序之所以比较快，是因为与冒泡排序相比，每次的交换时跳跃式的，每次排序的时候设置一个基准点，将小于等于基准点的数全部放到基准点的左边，将大于等于基准点的数全部放到基准点的右边。这样在每次交换的时候就不会像冒泡排序一样每次只能在相邻的数之间进行交换，交换的距离就大的多了。因此总的比较和交换次数就少了，速度自然就提高了。当然在最坏的情况下，仍可能是相邻的两个数进行了交换。因此快速排序的最差时间复杂度和冒泡排序是一样的都是O(n^2)，它的平均时间复杂度为O ( n log2 n )。

## C语言设计

|  |
| --- |
| #include"stdio.h"  int arr[8] = {3,8,1,5,2,4,6,7};  void Quick\_Sort(int \*arr, int begin, int end)  {  // 参数校验 begin<=end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 重点 ：也是递归调用的结束条件  if(begin > end)  return;    // 初始化 选定基准数，初始化哨兵  int tmp = arr[begin];  int i = begin;  int j = end;    // 开始排序  while(i != j) // 一轮遍历结束条件：两哨兵相遇  {  while(arr[j] >= tmp && j > i) // 哨兵j先移动，找到小于基准数的数值  j--;  while(arr[i] <= tmp && j > i) // 哨兵i再移动，找到大于基准数的数值  i++;    if(j > i) // 如果两哨兵没相遇，交换两个哨兵找到的值  {  int t = arr[i];  arr[i] = arr[j];  arr[j] = t;  }  }    // 一轮循环结束，交换基准数与哨兵相遇时的值  arr[begin] = arr[i];  arr[i] = tmp;    Quick\_Sort(arr, begin, i-1); // 前段 ——递归调用  Quick\_Sort(arr, i+1, end); // 后段 ——递归调用  }  int main()  {  int i;  // 显示排序前的序列  for(i=0;i<=7;i++)  printf("%d ",arr[i]);  printf("\n");  // 排序  Quick\_Sort(arr,0,7);  // 显示排序后的序列  for(i=0;i<=7;i++)  printf("%d ",arr[i]);  printf("\n");    } |

## 汇编语言设计

用LoongIDE新建一个工程。

移除core文件夹下的bsp\_start.c源文件。

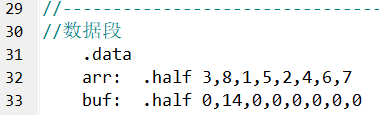
移除main.c源文件。

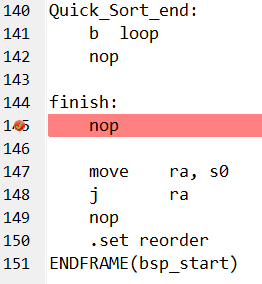
添加bsp\_start.S源文件。

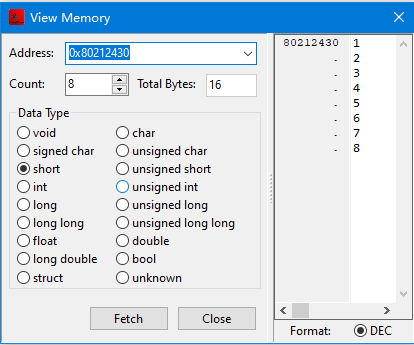
在bsp\_start.S下添加如下代码：

|  |
| --- |
| /\*  \* bsp\_start.S  \*  \* created: 2022/1/26  \* author: Li TianLing  \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*  \* v0 存储器寻址辅助寄存器  \* s0 待排序数据arr基地址  \* s1 参数传递-->begin  \* s2 参数传递-->end  \* s3 跳转辅助寄存器  \* s4 递归缓存区buf基地址  \* s5 递归缓存区buf偏移地址（指针）  \* t0 基准数  \* t1 哨兵i  \* t2 哨兵j  \* t3 哨兵i指向的值，即arr[i]  \* t4 哨兵j指向的值，即arr[j]  \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include "regdef.h"  #include "cpu.h"  #include "asm.h"  //-----------------------------------------------------------------------------  //数据段  .data  arr: .half 3,8,1,5,2,4,6,7 // 待排序数据（数组）  buf: .half 0,14,0,0,0,0,0,0 // 递归缓存区（栈）,其中第二个数字 = （待排序数字个数-1）\*2  //-----------------------------------------------------------------------------  //代码段  .text  FRAME(bsp\_start,sp,0,ra)  .set noreorder  move s6, ra /\* 返回地址 \*/  la s4,buf // 加载递调用缓存区基地址（常量）  la s5,4 // 初始化递归调用偏移地址（变量），指向buf中的第三个数  la s0,arr // 加载待排序数据基地址（常量）    loop: // 递归调用排序  blez s5,finish // 递归调用结束条件，如果s5<=0结束递归  nop  sub s5,2 // end 出栈  add v0,s4,s5  lh s2,(v0)  sub s5,2 // begin 出栈  add v0,s4,s5  lh s1,(v0) // 执行完此行，开始递归  Quick\_Sort\_start:  sub s3,s2,s1  blez s3,Quick\_Sort\_end // 参数校验：如果s2<=s1则结束此函数（不入栈）  nop  add v0,s0,s1 // 选定基准数  lh t0,(v0)  la t1,0 // 将传入参数begin加载到哨兵i  add t1,s1  la t2,0 // 将传入参数end加载到哨兵j  add t2,s2  while\_i\_j\_begin:  beq t1,t2,while\_i\_j\_end // 一轮遍历结束条件：两哨兵相遇  nop  while\_arr\_j\_begin: // 哨兵j先移动，找到小于基准数的数值  add v0,s0,t2 // 将哨兵j指向的值arr[j]加载到t4寄存器  lh t4,(v0)  sub s3,t4,t0 // 找到了小于基准数(t0)的值(t4)，即t4<t0结束寻找  bltz s3,while\_arr\_j\_end  nop  sub s3,t2,t1 // 参数校验：如果哨兵j与哨兵i相遇或者跑到哨兵i之前就结束寻找，即t2<=t1跳转  blez s3,while\_arr\_j\_end  nop  sub t2,2 // 哨兵j向前移动  b while\_arr\_j\_begin // 继续寻找  nop  while\_arr\_j\_end:  while\_arr\_i\_begin: // 哨兵i后移动，找到大于基准数的数值  add v0,s0,t1 // 将哨兵i指向的值arr[i]加载到t3寄存器  lh t3,(v0)  sub s3,t3,t0 // 找到了大于基准数(t0)的值(t3)，即t3>t0结束寻找  bgtz s3,while\_arr\_i\_end  nop  sub s3,t2,t1 // 参数校验：如果哨兵i与哨兵j相遇或者跑到哨兵j之后就结束寻找，即t1>=t2跳转  blez s3,while\_arr\_i\_end  nop  add t1,2 // 哨兵i向后移动  b while\_arr\_i\_begin // 继续寻找  nop  while\_arr\_i\_end:  if\_i\_j\_begin: // 如果两个哨兵没有相遇，就交换两个哨兵所指向的值  sub s3,t2,t1 // 如果两哨兵相遇或错位，则不执行这部分，即t2<=t1跳转  blez s3,if\_i\_j\_end  nop  add v0,s0,t1 // 将哨兵j指向的值(t4)存储到arr[i]  sh t4,(v0)  add v0,s0,t2 // 将哨兵i指向的值(t3)存储到arr[j]  sh t3,(v0)  if\_i\_j\_end:  b while\_i\_j\_begin // 开始下一轮遍历  nop  while\_i\_j\_end:  add v0,s0,s1 // 将哨兵i指向的值(t3)存到arr[begin]  sh t3,(v0)  add v0,s0,t1 // 将基准数(t0)存储到arr[i]  sh t0,(v0)  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 入栈  add v0,s4,s5 // 入栈begin  sh s1,(v0)  add s5,2  sub t1,2 // 入栈i-2  add v0,s4,s5  sh t1,(v0)  add s5,2  add t2,2 // 入栈j+2  add v0,s4,s5  sh t2,(v0)  add s5,2  add v0,s4,s5 // 入栈end  sh s2,(v0)  add s5,2  Quick\_Sort\_end:  b loop  nop  finish:  nop  move ra, s6  j ra  nop  .set reorder  ENDFRAME(bsp\_start) |

## 调试运行

初始数据如下所示：

如下图所示，在finish后打上断点：

联合调试后，双击s0寄存器，查看对应存储器数据：

如上图所示，排序成功。