## 实验7 排序算法实现与比较

1. 实验目的

（1）掌握几种低效的排序方法；

（2）掌握几种高效的排序方法。快排必须有

2. 实验内容

第一题：educoder中完成插入排序和快速排序。

第二题：线下实验中进行排序算法的比较

1. 通过实验记录同一类排序算法的比较次数和移动次数，从而比较算法的时间效率；
2. 产生特殊的数据初始状态，如正序、逆序、基本有序、数据规模等，从而对算法进行综合比较。

重点：

排序算法的比较

难点：

1、随机生成要排序的数据，使用rand函数的之前记得调用函数srand

2、至少掌握一种排序算法并实现升序排序和降序排序。  
3、有能力希望能掌握多种排序算法，并进行比较。

##### 具体讲解

1、编写排序算法时，经常会出现程序能运行，但是运行结果有误的情况，这就需要掌握程序的单步调试方法。单步调试的思维C语言复习文件夹中对于每种程序结构都有介绍。而单步调试的快捷键对于不同的编译环境是不同的，比如VC6与codeblock就有所区别。

另外，编写排序算法后，还需要多用不同的数据运行测试几次。

2、介绍随机数生成的方法和注意事项，使用rand函数

3、介绍如何获得程序片段的运行时间

4、对请学生自行学习阅读快速排序算法的优化方法

5、由学生自行选择排序算法编写程序

### 附录1（帮助文档中的随机数生成）

// crt\_rand.c

// This program seeds the random-number generator

// with the time, then exercises the rand function.

//

#include <stdlib.h>

#include <stdio.h>

#include <time.h>

void SimpleRandDemo( int n )

{

// Print n random numbers.

int i;

for( i = 0; i < n; i++ )

printf( " %6d\n", rand() );

}

void RangedRandDemo( int range\_min, int range\_max, int n )

{

// Generate random numbers in the half-closed interval

// [range\_min, range\_max). In other words,

// range\_min <= random number < range\_max

int i;

for ( i = 0; i < n; i++ )

{

int u = (double)rand() / (RAND\_MAX + 1) \* (range\_max - range\_min)

+ range\_min;

printf( " %6d\n", u);

}

}

int main( void )

{

// Seed the random-number generator with the current time so that

// the numbers will be different every time we run.

srand( (unsigned)time( NULL ) );

SimpleRandDemo( 10 );

printf("\n");

RangedRandDemo( -100, 100, 10 );

}

结果：

22036

18330

11651

27464

18093

3284

11785

14686

11447

11285

74

48

27

65

96

64

-5

-42

-55

66

### 附录2 获得程序片段的运行时间

**1、clock()函数**

C语言中求程序执行的时间可以使用clock()函数，该函数原型为clock\_t clock(void);

这个函数返回从“开启这个程序进程”到“程序中调用clock()函数”时之间的CPU时钟计时单元（clock tick）数，其中clock\_t是用来保存时间的数据类型，在time.h文件中，我们可以找到对它的定义：

#ifndef \_CLOCK\_T\_DEFINED

typedef long clock\_t;

#define \_CLOCK\_T\_DEFINED

#endif

很明显，clock\_t是一个长整形数

**2、常量CLOCKS\_PER\_SEC**

在time.h文件中，还定义了一个常量CLOCKS\_PER\_SEC，它用来表示一秒钟会有多少个时钟计时单元，其定义如下：

#define CLOCKS\_PER\_SEC ((clock\_t)1000)

可以看到每过千分之一秒（1毫秒），调用clock（）函数返回的值就加1。

**3、示例代码**

#include <stdio.h>

#include <time.h>

int main()

{

clock\_t start\_time = clock();

//程序代码片段

clock\_t end\_time = clock();

printf("Runing time is %lf ms\n",(double)(end\_time-start\_time)\*1000/CLOCKS\_PER\_SEC);

return 0;

}

### 附录3 根据课本补充代码

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

typedef int KeyType;

typedef int DataType;

typedef struct{

KeyType key;

DataType info;

}RecordNode;

typedef struct{

int n;

RecordNode \*record;

}SortObject;

SortObject\* creat(int num)

{

int i;

SortObject\* sortObject = (SortObject\*)malloc(sizeof(SortObject));

sortObject->n = num;

sortObject->record = (RecordNode\*)malloc(sizeof(RecordNode)\*num);

srand( (unsigned)time( NULL ) );

for(i=0;i<num;i++)

{

sortObject->record[i].key = rand();

}

return sortObject;

}

int main()

{

int i;

SortObject \*mysort = creat(20000);

//测试而已

for(i=0;i<10;i++)

{

printf("%d\n",mysort->record[i].key);

}

return 0;

}

### 附录4 快速排序的优化算法

**快速排序的4种优化**

2018-09-15 22:14:28 [Tyler\_Z](https://me.csdn.net/qq_38289815" \t "_blank) 阅读数 5846更多

分类专栏： [数据结构](https://blog.csdn.net/qq_38289815/article/category/8060683)

版权声明：本文为博主原创文章，遵循[CC 4.0 BY-SA](http://creativecommons.org/licenses/by-sa/4.0/)版权协议，转载请附上原文出处链接和本声明。

本文链接：<https://blog.csdn.net/qq_38289815/article/details/82718428>

[快排思想](https://blog.csdn.net/qq_38289815/article/details/82718428#%E5%BF%AB%E6%8E%92%E6%80%9D%E6%83%B3)

[快排基准的选择](https://blog.csdn.net/qq_38289815/article/details/82718428#%E5%BF%AB%E6%8E%92%E5%9F%BA%E5%87%86%E7%9A%84%E9%80%89%E6%8B%A9)

[固定基准](https://blog.csdn.net/qq_38289815/article/details/82718428#(1)%E5%9B%BA%E5%AE%9A%E5%9F%BA%E5%87%86)

[随机基准](https://blog.csdn.net/qq_38289815/article/details/82718428#(2)%E9%9A%8F%E6%9C%BA%E5%9F%BA%E5%87%86)

[三数取中](https://blog.csdn.net/qq_38289815/article/details/82718428#(3)%E4%B8%89%E6%95%B0%E5%8F%96%E4%B8%AD)

[快速排序的优化](https://blog.csdn.net/qq_38289815/article/details/82718428#%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F%E7%9A%84%E4%BC%98%E5%8C%96)

[优化1：序列长度达到一定大小时，使用插入排序](https://blog.csdn.net/qq_38289815/article/details/82718428#%E4%BC%98%E5%8C%961%EF%BC%9A%E5%BA%8F%E5%88%97%E9%95%BF%E5%BA%A6%E8%BE%BE%E5%88%B0%E4%B8%80%E5%AE%9A%E5%A4%A7%E5%B0%8F%E6%97%B6%EF%BC%8C%E4%BD%BF%E7%94%A8%E6%8F%92%E5%85%A5%E6%8E%92%E5%BA%8F)

[优化2：尾递归优化](https://blog.csdn.net/qq_38289815/article/details/82718428#%E4%BC%98%E5%8C%962%EF%BC%9A%E5%B0%BE%E9%80%92%E5%BD%92%E4%BC%98%E5%8C%96)

[优化3：聚集元素](https://blog.csdn.net/qq_38289815/article/details/82718428#%E4%BC%98%E5%8C%963%EF%BC%9A%E8%81%9A%E9%9B%86%E5%85%83%E7%B4%A0)

[优化4：多线程处理快排](https://blog.csdn.net/qq_38289815/article/details/82718428#%E4%BC%98%E5%8C%964%EF%BC%9A%E5%A4%9A%E7%BA%BF%E7%A8%8B%E5%A4%84%E7%90%86%E5%BF%AB%E6%8E%92)

**快排思想**

快排算法是基于分治策略的排序算法，其基本思想是，对于输入的数组a[low, high]，按以下三个步骤进行排序。

(1)分解：以a[p]为基准将a[low: high]划分为三段a[low：p-1]，a[p]和a[p+1：high]，使得a[low：p-1]中任何一个元素小于等于a[p]， 而a[p+1: high]中任何一个元素大于等于a[p]。

(2)递归求解：通过递归调用快速排序算法分别对a[low：p-1]和a[p+1：high]进行排序。

(3)合并：由于对a[low：p-1]和a[p+1：high]的排序是就地进行的，所以在a[low：p-1]和a[p+1：high]都已排好序后，不需要执行任何计算，a[low：high]就已经排好序了。

**快排基准的选择**

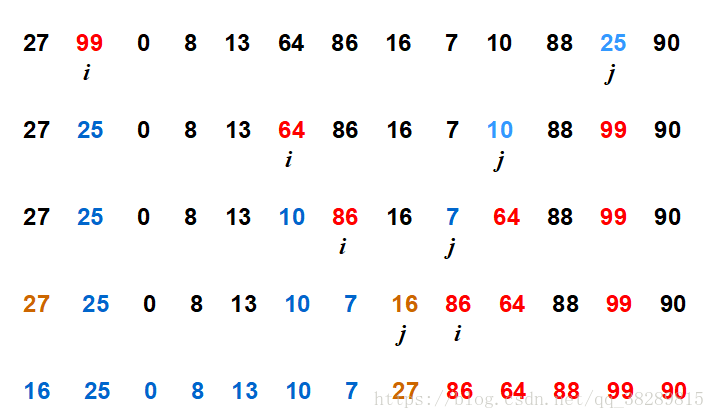
快速排序的运行时间与划分是否对称有关。最坏情况下，每次划分过程产生两个区域分别包含n-1个元素和1个元素，其时间复杂度会达到O(n^2)。在最好的情况下，每次划分所取的基准都恰好是中值，即每次划分都产生两个大小为n/2的区域。此时，快排的时间复杂度为O(nlogn)。所以基准的选择对快排而言至关重要。快排中基准的选择方式有以下三种。

**(1)固定基准**

以下Partition()函数对数组进行划分时，以元素x = a[low]作为划分的基准。

1. template <class T>
2. int Partition(T a[], int low, int high)
3. {
4. int i = low, j = high+1;
5. T x = a[low];
6. while(true)
7. {
8. while(a[++i] < x && i < high);
9. while(a[--j] > x);
10. if(i >= j)break;
11. Swap(a[i], a[j]);
12. }
13. a[low] = a[j];
14. a[j] = x;
15. return j;
16. }

快排过程一趟：



如果数组元素是随机的，划分过程不产生极端情况，那么程序的运行时间不会有太大的波动。如果数组元素已经基本有序时，此时的划分就容易产生最坏的情况，即快速排序变成冒泡排序，时间复杂度为O(n^2)。

例如：序列[1][2][3][5][4][6]以固定基准进行快排时。

第一趟：[1][2][3][5][4][6]

第二趟：[1][2][3][5][4][6]

第三趟：[1][2][3][5][4][6]

第四趟：[1][2][3][4][5][6]

程序中要用的函数：

(1)C++可以使用以下方法产生随机数，而单纯的使用rand()%M产生的是伪随机数。

1. *#define M 100*
2. srand(time(0));
3. for(int i = 0; i < M; i++)
4. a[i] = rand()%(M);

(2)方法一：获得程序片段运行的时间：

1. clock\_t start\_time = clock();
2. *//程序代码片段*
3. clock\_t end\_time = clock();
4. cout<<"Runing time is:"<<static\_cast<double>(end\_time-start\_time)/CLOCKS\_PER\_SEC\*1000<<"ms"<<endl;

方法二：获得程序片段运行的时间：

1. #include<sys/time.h>
2. int gettimeofday(struct timeval\*tv,struct timezone \*tz )
3. struct timeval
4. {
5. long tv\_sec;*/\*秒\*/*
6. long tv\_usec;*/\*微妙\*/*
7. };
8. struct timezone
9. {
10. int tz\_minuteswest;
11. */\*和greenwich时间差\*/*
12. int tz\_dsttime;
13. };

例如：

1. int main()
2. {
3. float Time=0;
4. struct timeval start;
5. struct timeval end;
7. gettimeofday(&start,NULL); *//gettimeofday(&start,&tz);结果一样*
8. *//程序片段*
9. gettimeofday(&end,NULL);
11. Time=(end.tv\_sec-start.tv\_sec)\*1000000+(end.tv\_usec-start.tv\_usec);*//微秒*
12. cout << Time <<endl;
13. return 0;
14. }

完整代码如下：

1. #include <iostream>
2. #include <time.h>
3. #include <stdlib.h>
4. #include <cstdlib>
5. #define M 1000000
6. using namespace std;
8. template <class T>
9. void Print(T a[], int n, int m)
10. {
11. for(int i = n; i < m; i++)
12. {
13. cout << "[" << a[i] << "]";
14. }
15. cout <<endl;
16. }
18. template <class T>
19. void Swap(T &a, T &b)
20. {
21. T asd;
22. asd = a;
23. a = b;
24. b = asd;
25. }
27. template <class T>
28. int Partition(T a[], int p, int r)
29. {
30. int i = p, j = r+1;
31. T x = a[p];
32. while(true)
33. {
34. while(a[++i] < x && i < r);
35. while(a[--j] > x);
36. if(i >= j)break;
37. Swap(a[i], a[j]);
38. }
39. a[p] = a[j];
40. a[j] = x;
41. return j;
42. }
44. template <class T>
45. void QuickSort(T a[], int p, int r)
46. {
47. if(p < r)
48. {
49. int q = Partition(a, p, r);
50. QuickSort(a, p, q-1);
51. QuickSort(a, q+1, r);
52. }
53. }
55. int main()
56. {
57. srand(time(0));
58. int a[M] = {0};
59. for(int i = 0; i < M; i++)
60. *//a[i] = i+1; //设置升序数组*
61. a[i] = rand()%(M); *//设置随机数组*
62. *//if(i < M/2-1) //设置重复数组*
63. *// a[i] = 5;*
64. *//else*
65. *// a[i] = 10;*
66. *//在检验数据的时候可以用Print()函数将其打印出来*
67. *//Print(a, 0, M);*
68. clock\_t start\_time = clock();
69. QuickSort(a, 0, M-1);
70. clock\_t end\_time = clock();
71. cout<<"Runing time is:"<<static\_cast<double>(end\_time - start\_time)/CLOCKS\_PER\_SEC\*1000<<"ms"<<endl;
72. *//Print(a, 0, M);*
73. return 0;
74. }

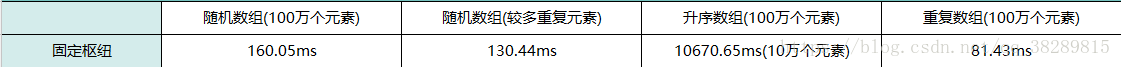
p.s：(1)在Codeblocks下处理升序数组时，元素最好设置少一点。设置的太大可能会出现下图提示：



(2)重复数组中的元素值只有两个。

(3)随机数组(较多重复元素)的设置是：a[i] = rand()%(M/100);。

数据如下：



固定基准对升序数组的分割及其糟糕，排序时间特别长，所以只设置了10万个元素(从上面代码可以看出，设置的是栈数组，如果设置的是堆数组，处理元素的数量可以很大，程序段处理时间也会相应减少)。

**(2)随机基准**

在待排数组有序或基本有序的情况下，选择使用固定基准影响快排的效率。为了解决数组基本有序的问题，可以采用随机基准的方式来化解这一问题。算法如下：

1. int Random(int a[], int low, int high)*//在low和high间随机选择一元素作为划分的基准*
2. {
3. srand(time(0));
4. int pivot = rand()%(high - low) + low;
5. Swap(a[pivot],a[low]); *//把随机基准位置的元素和low位置元素互换*
6. return a[low];
7. }

此时，原来Partition()函数里的T x = a[low];相应的改为T x = Random(a, low, high);

虽然使用随机基准能解决待排数组基本有序的情况，但是由于这种随机性的存在，对其他情况的数组也会有影响(若数组元素是随机的，使用固定基准常常优于随机基准)。随机数算法(Sherwood算法)能有效的减少升序数组排序所用的时间，数组元素越多，随机数算法的效果越好。可以试想，上述升序数组中有10万个元素而且各不相同，那么在第一次划分时，基准选的最差的概率就是十万分之一。当然，选择最优基准的概率也是十万分之一，随机数算法随机选择一个元素作为划分基准，算法的平均性能较好，从而避免了最坏情况的多次发生。许多算法书中都有介绍随机数算法，因为算法对程序的优化程度和下面所讲的三数取中方法很接近，所以我只记录了一种方法的运行时间。

**(3)三数取中**

由于随机基准选取的随机性，使得它并不能很好的适用于所有情况(即使是同一个数组，多次运行的时间也大有不同)。目前，比较好的方法是使用三数取中选取基准。它的思想是：选取数组开头，中间和结尾的元素，通过比较，选择中间的值作为快排的基准。其实可以将这个数字扩展到更大(例如5数取中，7数取中等)。这种方式能很好的解决待排数组基本有序的情况，而且选取的基准没有随机性。

例如：序列[1][1][6][5][4][7][7]，三个元素分别是[1]、[5]、[7]，此时选择[5]作为基准。

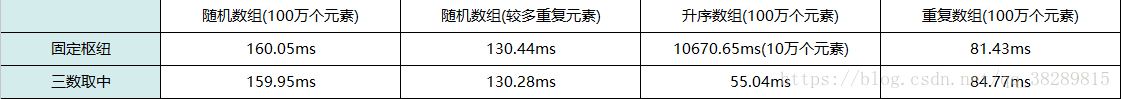
第一趟：[1][1][4][5][6][7][7]

三数取中算法如下：

1. int NumberOfThree(int arr[],int low,int high)
2. {
3. int mid = low + ((high - low) >> 1);*//右移相当于除以2*
5. if (arr[mid] > arr[high])
6. {
7. Swap(arr[mid],arr[high]);
8. }
9. if (arr[low] > arr[high])
10. {
11. Swap(arr[low],arr[high]);
12. }
13. if (arr[mid] > arr[low])
14. {
15. Swap(arr[mid],arr[low]);
16. }
17. *//此时，arr[mid] <= arr[low] <= arr[high]*
18. return arr[low];
19. }

同理，Partition()函数里的T x = a[low];相应的改为T x = NumberOfThree(a, low, high);

数据如下：



三数取中(随机数算法效果相同)在处理升序数组时有质的飞越，而且处理的还是100万个元素。

**快速排序的优化**

**优化1：序列长度达到一定大小时，使用插入排序**

当快排达到一定深度后，划分的区间很小时，再使用快排的效率不高。当待排序列的长度达到一定数值后，可以使用插入排序。由《数据结构与算法分析》(Mark Allen Weiness所著)可知，当待排序列长度为5~20之间，此时使用插入排序能避免一些有害的退化情形。

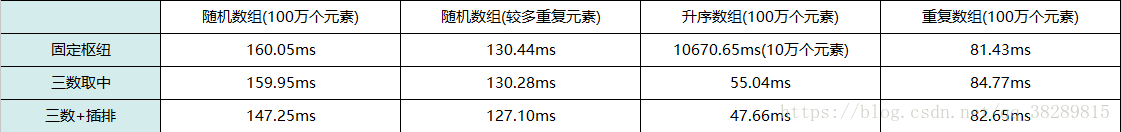
1. template <class T>
2. void QSort(T arr[],int low,int high)
3. {
4. int pivotPos;
5. if (high - low + 1 < 10)
6. {
7. InsertSort(arr,low,high);
8. return;
9. }
10. if(low < high)
11. {
12. pivotPos = Partition(arr,low,high);
13. QSort(arr,low,pivotPos-1);
14. QSort(arr,pivotPos+1,high);
15. }
16. }

完整代码如下：

1. */\**
2. *次快排代码采用了 三数取中&插入排序*
3. *\*/*
4. #include <iostream>
5. #include <time.h>
6. #include <stdlib.h>
7. using namespace std;
9. #define M 1000000
10. int NumberOfThree(int arr[],int low,int high);
12. template <class T>
13. void Print(T a[], int n)
14. {
15. for(int i = 0; i < n; i++)
16. {
17. cout << "[" << a[i] << "]";
18. }
19. cout <<endl;
20. }
22. template <class T>
23. void Swap(T &a, T &b)
24. {
25. T asd;
26. asd = a;
27. a = b;
28. b = asd;
29. }
31. template <class T>
32. int Partition(T a[], int p, int r)
33. {
34. int i = p, j = r+1;
35. T x = NumberOfThree(a, p, r);
36. while(true)
37. {
38. while(a[++i] < x && i < r);
39. while(a[--j] > x);
40. if(i >= j)break;
41. Swap(a[i], a[j]);
42. }
43. a[p] = a[j];
44. a[j] = x;
45. return j;
46. }

49. void InsertSort(int arr[], int m, int n)
50. {
51. int i, j;
52. int temp; *// 用来存放临时的变量*
53. for(i = m+1; i <= n; i++)
54. {
55. temp = arr[i];
56. for(j = i-1; (j >= m)&&(arr[j] > temp); j--)
57. {
58. arr[j + 1] = arr[j];
59. }
60. arr[j + 1] = temp;
61. }
62. }
64. int NumberOfThree(int arr[],int low,int high)
65. {
66. int mid = low + ((high - low) >> 1);
68. if (arr[mid] > arr[high])
69. {
70. Swap(arr[mid],arr[high]);
71. }
72. if (arr[low] > arr[high])
73. {
74. Swap(arr[low],arr[high]);
75. }
76. if (arr[mid] > arr[low])
77. {
78. Swap(arr[mid],arr[low]);
79. }
80. *//此时，arr[mid] <= arr[low] <= arr[high]*
81. return arr[low];
82. }
84. template <class T>
85. void QSort(T arr[],int low,int high)
86. {
87. int pivotPos;
88. if (high - low + 1 < 10)
89. {
90. InsertSort(arr,low,high);
91. return;
92. }
93. if(low < high)
94. {
95. pivotPos = Partition(arr,low,high);
96. QSort(arr,low,pivotPos-1);
97. QSort(arr,pivotPos+1,high);
98. }
99. }
101. int main()
102. {
103. srand(time(0));
104. int a[M] = {0};
105. for(int i=0;i<M;i++)
106. *//a[i] = i+1; //设置升序数组*
107. a[i] = rand()%(M); *//设置随机数组*
108. *//if(i < M/2-1) //设置重复数组*
109. *// a[i] = 1;*
110. *//else*
111. *// a[i] = 10;*
112. *//Print(a, M);*
113. clock\_t start\_time = clock();
114. QSort(a, 0, M-1);
115. clock\_t end\_time = clock();
116. cout<<"Runing time is:"<<static\_cast<double>(end\_time - start\_time)/CLOCKS\_PER\_SEC\*1000<<"ms"<<endl;
117. *//Print(a, M);*
118. return 0;
119. }

数据如下：



如上所述，在划分到很小的区间时，里面的元素已经基本有序了，再使用快排，效率就不高了。所以，在结合插入排序后，程序的执行效率有所提高。

**优化2：尾递归优化**

快排算法和大多数分治排序算法一样，都有两次递归调用。但是快排与归并排序不同，归并的递归则在函数一开始， 快排的递归在函数尾部，这就使得快排代码可以实施尾递归优化。使用尾递归优化后，可以缩减堆栈的深度，由原来的O(n)缩减为O(logn)。

尾递归概念：

如果一个函数中所有递归形式的调用都出现在函数的末尾，当递归调用是整个函数体中最后执行的语句且它的返回值不属于表达式的一部分时，这个递归调用就是尾递归。尾递归函数的特点是在回归过程中不用做任何操作，这个特性很重要，因为大多数现代的编译器会利用这种特点自动生成优化的代码。

尾递归原理：

当[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8)检测到一个函数调用是尾递归的时候，它就覆盖当前的活动记录而不是在栈中去创建一个新的。编译器可以做到这点，因为递归调用是当前活跃期内最后一条待执行的语句，于是当这个调用返回时栈帧中并没有其他事情可做，因此也就没有保存栈帧的必要了。通过覆盖当前的栈帧而不是在其之上重新添加一个，这样所使用的栈空间就大大缩减了，这使得实际的运行效率会变得更高。

代码如下：

1. #include <bits/stdc++.h>
2. using namespace std;
4. int fact(int n) *//线性递归*
5. {
6. if (n < 0)
7. return 0;
8. else if(n == 0 || n == 1)
9. return 1;
10. else
11. return n \* fact(n - 1);
12. }
14. int facttail(int n, int a) *//尾递归*
15. {
16. if (n < 0)
17. return 0;
18. else if (n == 0)
19. return 1;
20. else if (n == 1)
21. return a;
22. else
23. return facttail(n - 1, n \* a);
24. }
26. int main()
27. {
28. int a = fact(5);
29. int b = facttail(5, 1);
30. cout << "A:" << a <<endl;
31. cout << "B:" << b <<endl;
32. }

示例中的函数是尾递归的，因为对facttail的单次递归调用是函数返回前最后执行的一条语句。在facttail中碰巧最后一条语句也是对facttail的调用，但这并不是必需的。换句话说，在递归调用之后还可以有其他的语句执行，只是它们只能在递归调用没有执行时才可以执行。尾递归是极其重要的，不用尾递归，函数的堆栈耗用难以估量，需要保存很多中间函数的堆栈。比如f(n, sum) = f(n-1) + value(n) + sum; 会保存n个函数调用堆栈，而使用尾递归f(n, sum) = f(n-1, sum+value(n)); 这样则只保留后一个函数堆栈即可，之前的可优化删去。

代码当n=5时，线性递归的递归过程如下:

1. fact(5)
2. {5\*fact(4)}
3. {5\*{4\*fact(3)}}
4. {5\*{4\*{3\*fact(2)}}}
5. {5\*{4\*{3\*{2\*fact(1)}}}}
6. {5\*{4\*{3\*{2\*1}}}}
7. {5\*{4\*{3\*2}}}
8. {5\*{4\*6}}
9. {5\*24}
10. 120

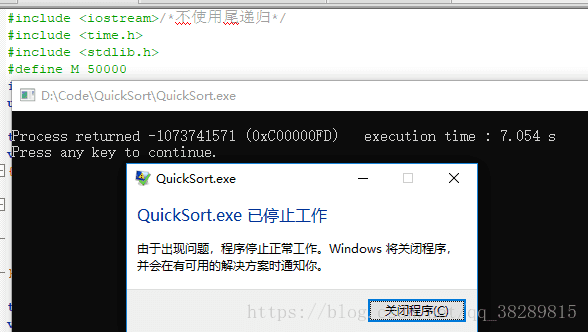
而尾递归的递归过程如下:

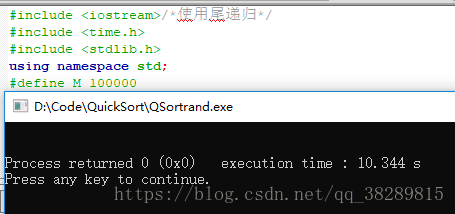
1. facttail(5,1)
2. facttail(4,5)
3. facttail(3,20)
4. facttail(2,60)
5. facttail(1,120)
6. 120

从上面的例子可以看出，普通的线性递归比尾递归更加消耗资源，因为每次重复的过程调用都使得调用的深度不断加深。 系统不得不使用栈进行数据保存和恢复。而尾递归就不存在这样的问题， 它的状态完全由n和a保存。所以，尾递归可以缩减栈的深度，由原来的O(n)缩减为O(logn)。

在Codeblocks里运行快排代码处理升序数组，一个进行尾递归优化，而另一个不变。没有使用尾递归的代码处理4万个数组元素时，由于超过了栈的深度，程序会异常结束。而使用了尾递归的代码，就算处理10万个数组元素，也不会出现异常(结合三数取中，可以处理100万个数组元素)。

2018年10月2日补充：结合我的另一篇博文[《内存四区》](https://blog.csdn.net/qq_38289815/article/details/82919323)，对上述问题有更全面的认识。

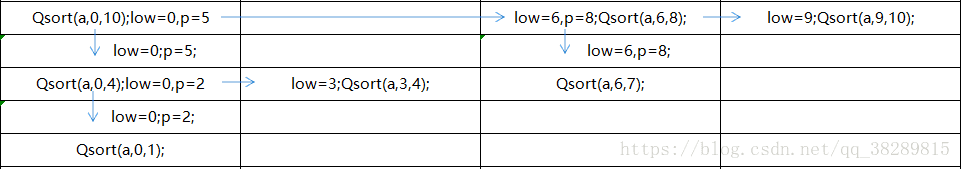




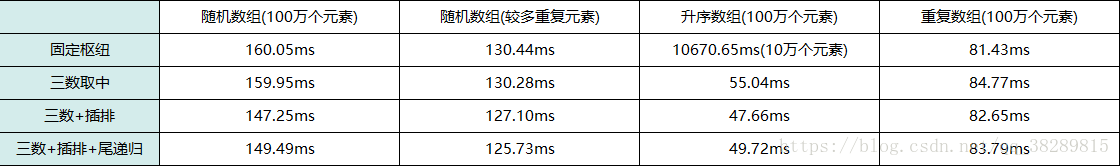
快排尾递归代码如下：

1. template <class T>
2. void QSort(T arr[],int low,int high)
3. {
4. int pivotPos;
5. if (high - low + 1 < 10)
6. {
7. InsertSort(arr,low,high);
8. return;
9. }
10. while(low < high)
11. {
12. pivotPos = Partition(arr,low,high);
13. QSort(arr,low,pivotPos-1);
14. low = pivotPos + 1;
15. }
16. }

第一次递归以后，变量low就没有用处了， 也就是说第二次递归可以用迭代控制结构代替。快排尾递归过程如下，纵向是递归，横向是迭代。



数据如下：



对递归的优化，主要是为了减少栈深度。在处理随机数组时，(三数取中+插排+尾递归)的组合并不一定比(三数取中+插排)的效率高。

**优化3：聚集元素**

聚集元素的思想：在一次分割结束后，将与本次基准相等的元素聚集在一起，再分割时，不再对聚集过的元素进行分割。具体过程有两步，①在划分过程中将与基准值相等的元素放入数组两端，②划分结束后，再将两端的元素移到基准值周围。

普通过程例如：[7][2][7][1][7][4][7][6][3][8] 由三数取中可得基准为[7]

第一趟：[7] [2] [3] [1] [6] [4] [7] [7] [7] [8]

第二趟：[1] [2] [3] [4] [6] [7] [7] [7] [7] [8]

第三趟：[1] [2] [3] [4] [6] [7] [7] [7] [7] [8]

第四趟：[1] [2] [3] [4] [6] [7] [7] [7] [7] [8]

聚集相同元素：

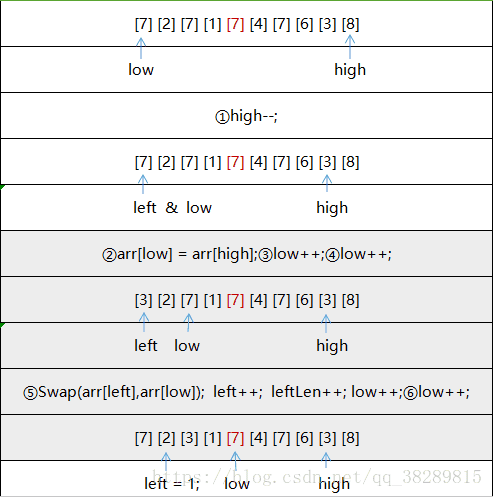
第一步：[7] [7] [7] [1] [2] [4] [3] [6] [7] [8]

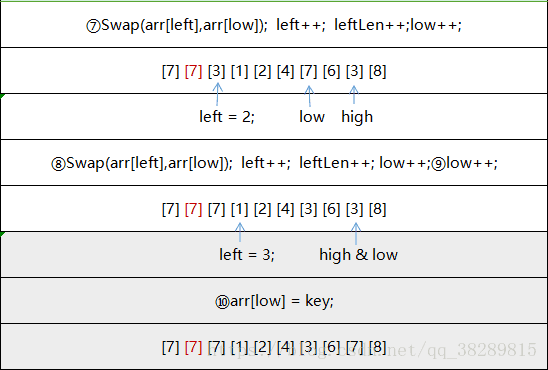
第二步：[6] [3] [4] [1] [2] [7] [7] [7] [7] [8]

接下来是对[6] [3] [4] [1] [2] 和 [8]进行快排。(具体过程可以按照以下代码走一遍)

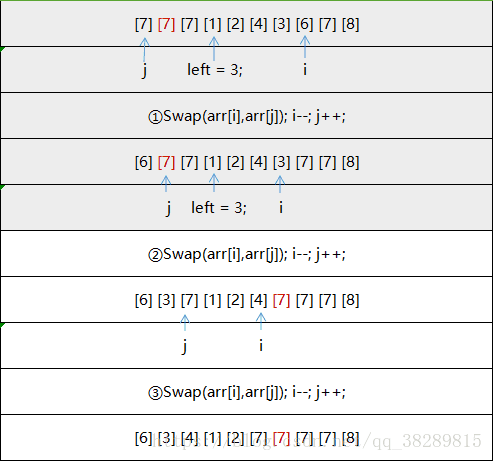
1. template <class T>
2. void QSort(T arr[],int low,int high)
3. {
4. int first = low;
5. int last = high;
7. int left = low;
8. int right = high;
10. int leftLen = 0;
11. int rightLen = 0;
13. if (high - low + 1 < 10)
14. {
15. InsertSort(arr,low,high);
16. return;
17. }
19. *//一次分割*
20. int key = NumberOfThree(arr,low,high);*//使用三数取中选择枢轴*
22. while(low < high)
23. {
24. while(high > low && arr[high] >= key)
25. {
26. if (arr[high] == key)*//处理相等元素*
27. {
28. Swap(arr[right],arr[high]);
29. right--;
30. rightLen++;
31. }
32. high--;
33. }
34. arr[low] = arr[high];
35. while(high > low && arr[low] <= key)
36. {
37. if (arr[low] == key)
38. {
39. Swap(arr[left],arr[low]);
40. left++;
41. leftLen++;
42. }
43. low++;
44. }
45. arr[high] = arr[low];
46. }
47. arr[low] = key;
49. *//一次快排结束，把与基准相等的元素移到基准周围*
50. int i = low - 1;
51. int j = first;
52. while(j < left && arr[i] != key)
53. {
54. Swap(arr[i],arr[j]);
55. i--;
56. j++;
57. }
58. i = low + 1;
59. j = last;
60. while(j > right && arr[i] != key)
61. {
62. Swap(arr[i],arr[j]);
63. i++;
64. j--;
65. }
66. QSort(arr,first,low - 1 - leftLen);
67. QSort(arr,low + 1 + rightLen,last);
68. }

聚集元素第一步：



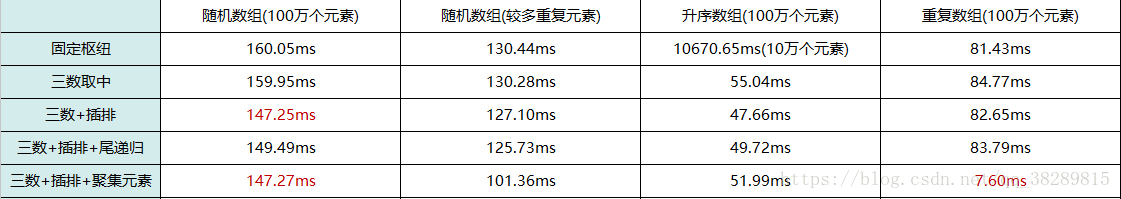


聚集元素第二步：



下一次就是对[6] [3] [4] [1] [2] 进行快排。当划分区间达到插入排序的要求时，就使用插入排序完成后续工作，所以进入插入排序那一段代码是停止继续递归的标志。

数据如下：



从上表中可以看到，通过对快排聚集元素的优化，在处理数组中的重复元素时有很大的提升。而对于升序数组而言，因为其本身就是有序的，而且没有重复元素，所以结果没有(三数取中+插排)效率高。

**优化4：多线程处理快排**

分治法的基本思想是将一个规模为n的问题分解为k个规模较小的子问题，这些子问题互相独立且与原问题相同。求解这些子问题，然后将各子问题的解合并，从而得到的原问题的解。由此，在处理快排的时候，可以使用多线程提高排序的效率。

要使用的函数：  
(1)pthread\_create  
创建一个线程的函数是pthread\_create。其定义如下：

1. #include <pthread.h>*//(Linux下编译需要加 -lpthread)*
2. int pthread\_create(pthread\_t\* thread, const pthread\_attr\_t\* attr, void\* *(\*start\_routine)(void\*)*, void \*arg);

第一个参数是一个整数类型，它表示的是资源描述符，实际上，Linux上几乎所有的资源标识符都是一个整型数。第二个attr参数用于设置新线程的属性。给它传递NULL表示使用默认线程属性。start\_routine和arg参数分别指定新线程将运行的函数及其参数。  
pthread\_create()成功时返回0，失败时返回错误码。

(2)pthread\_barrier\_init  
多线程编程时，可以使用这个函数来等待其它线程结束，例如：主线程创建一些线程，这些线程去完成一些工作，而主线程需要去等待这些线程结束(pthread\_join也能实现一种屏障)。可以把屏障理解为：为了协同线程之间的工作而使得某一具体线程进入等待状态的一种机制。其原型：

int pthread\_barrier\_init(pthread\_barrier\_t \*restrict barrier, const pthread\_barrierattr\_t \*restrict attr, unsigned int count);

函数执行成功返回 0，执行失败则返回一个错误号，我们可以通过该错误号获取相关的错误信息。  
第一个参数：一个指向pthread\_barrier\_t 类型的指针，我们必须要指出的是pthread\_barrier\_init函数不会给指针分配相关内存空间，因此我们传入的指针必须为一个pthread\_barrier\_t 变量。  
第二个参数：用于指定屏障的细节参数，我们这里可以暂且不去管它，如果我们传入NULL，那么系统将按照默认情况处理。  
第三个参数：设计屏障等待的最大线程数目。

(3)pthread\_barrier\_wait  
当一个线程需要等待其它多个线程结束时，调用该函数。  
原型：

int pthread\_barrier\_wait(pthread\_barrier\_t \*barrier);

函数执行成功返回 0，执行失败则返回一个错误码，我们可以通过该错误码获取相关的错误信息。  
函数参数：指向pthread\_barrier\_t 变量的指针。  
注意：使用barrier这个屏障，无法获取线程的结束状态。若想要获取相关线程结束状态，则需要调用pthread\_join函数。

代码如下：

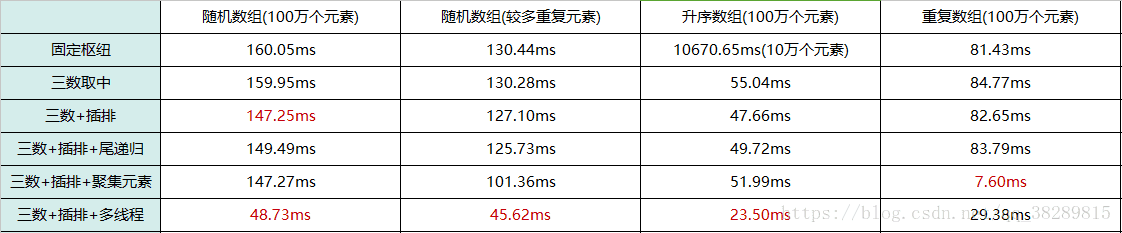
1. #include <cstdio> /\*三数取中+插入+聚集元素+多线程组合 && 三数取中+插入+尾递归+多线程组合\*/
2. #include <iostream>
3. #include <stdlib.h>
4. #include <sys/time.h>
5. #include <pthread.h>
6. using namespace std;
8. const long MAX = 1000000L; *//数组中最大数*
9. const long long MaxNumber = 1000000L; *//排序数*
10. const int thread = 4; *//线程数*
11. const long NumberOfSort = MaxNumber / thread; *//每个线程排序的个数*
13. int array\_a[MaxNumber];
14. int array\_b[MaxNumber]; *//合并后，由b数组记录最终序列*
16. pthread\_barrier\_t barrier;
18. void initial() *//数组初始化函数*
19. {
20. srand(time(0));
21. for(int i = 0; i < MaxNumber; ++i)
22. array\_a[i] = rand()%(MAX);
23. *//if(i < MaxNumber/2)*
24. *// array\_a[i] = 5;*
25. *//else*
26. *// array\_a[i] = 10;*
27. *//array\_a[i] = i+1;*
28. }

31. template <class T>
32. void Print(T a[], int n)
33. {
34. for(int i = 0; i < n; i++)
35. {
36. cout << "[" << a[i] << "]";
37. }
38. cout <<endl;
39. }
41. template <class T>
42. void Swap(T &a, T &b)
43. {
44. T asd;
45. asd = a;
46. a = b;
47. b = asd;
48. }
50. void InsertSort(int arr[],int start,int end)
51. {
52. int low,high,median,tmp;
53. for(int i = start+1;i<= end;i++)
54. {
55. low = start;
56. high = i-1;
58. while(low <= high)
59. {
60. median = (low + high) /2;
61. if(arr[i] < arr[median])
62. {
63. high = median -1;
64. }else
65. {
66. low = median + 1;
67. }
68. }
70. tmp = arr[i];
72. for(int j = i-1;j>high;j--)
73. {
74. arr[j+1] = arr[j];
75. }
76. arr[high+1] = tmp;
77. }
78. }
80. int NumberOfThree(int arr[],int low,int high)
81. {
82. int mid = low + ((high - low) >> 1);
83. if (arr[mid] > arr[high])
84. {
85. Swap(arr[mid],arr[high]);
86. }
87. if (arr[low] > arr[high])
88. {
89. Swap(arr[low],arr[high]);
90. }
91. if (arr[mid] > arr[low])
92. {
93. Swap(arr[mid],arr[low]);
94. }
95. return arr[low];
96. }
98. template <class T>
99. int Partition(T a[], int p, int r)
100. {
101. int i = p, j = r+1;
102. T x = NumberOfThree(a, p, r);
103. while(true)
104. {
105. while(a[++i] < x && i < r);
106. while(a[--j] > x);
107. if(i >= j)break;
108. Swap(a[i], a[j]);
109. }
110. a[p] = a[j];
111. a[j] = x;
112. return j;
113. }
115. #if 1 *//有聚集元素的快排*
116. template <class T>
117. void QSort(T arr[],int low,int high)
118. {
119. int first = low;
120. int last = high;
121. int left = low;
122. int right = high;
123. int leftLen = 0;
124. int rightLen = 0;
126. if (high - low + 1 < 6)
127. {
128. InsertSort(arr,low,high);
129. return;
130. }
132. *//一次分割*
133. int key = NumberOfThree(arr,low,high);*//使用三数取中法选择枢轴*
135. while(low < high)
136. {
137. while(high > low && arr[high] >= key)
138. {
139. if (arr[high] == key)*//处理相等元素*
140. {
141. Swap(arr[right],arr[high]);
142. right--;
143. rightLen++;
144. }
145. high--;
146. }
147. arr[low] = arr[high];
148. while(high > low && arr[low] <= key)
149. {
150. if (arr[low] == key)
151. {
152. Swap(arr[left],arr[low]);
153. left++;
154. leftLen++;
155. }
156. low++;
157. }
158. arr[high] = arr[low];
159. }
160. arr[low] = key;
162. *//一次快排结束*
163. *//把与枢轴key相同的元素移到枢轴最终位置周围*
164. int i = low - 1;
165. int j = first;
166. while(j < left && arr[i] != key)
167. {
168. Swap(arr[i],arr[j]);
169. i--;
170. j++;
171. }
172. i = low + 1;
173. j = last;
174. while(j > right && arr[i] != key)
175. {
176. Swap(arr[i],arr[j]);
177. i++;
178. j--;
179. }
180. QSort(arr,first,low - 1 - leftLen);
181. QSort(arr,low + 1 + rightLen,last);
182. }
183. #endif
185. #if 0 *//没有聚集元素的快排*
186. template <class T>
187. void QSort(T arr[],int low,int high)
188. {
189. int pivotPos;
190. if (high - low + 1 < 10)
191. {
192. InsertSort(arr,low,high);
193. return;
194. }
196. while(low < high)
197. {
198. pivotPos = Partition(arr,low,high);
199. QSort(arr,low,pivotPos-1);
200. low = pivotPos + 1;
201. }
202. }
203. #endif *// 1*
205. void\* work(void \*arg) *//线程排序函数*
206. {
207. long length = (long)arg;
208. QSort(array\_a, length, length + NumberOfSort - 1);
209. pthread\_barrier\_wait(&barrier);
210. pthread\_exit(NULL);
211. }
213. void meger() *//最终合并函数*
214. {
215. long index[thread];
216. for (int i = 0; i < thread; ++i)
217. {
218. index[i] = i \* NumberOfSort;
219. }
221. for(long i = 0; i < MaxNumber; ++i)
222. {
223. long min\_index;
224. long min\_num = MAX;
225. for(int j = 0; j < thread; ++j)
226. {
227. if((index[j] < (j + 1) \* NumberOfSort)&& (array\_a[index[j]] < min\_num))
228. {
229. min\_index = j;
230. min\_num = array\_a[index[j]];
231. }
232. }
233. array\_b[i] = array\_a[index[min\_index]];
234. index[min\_index]++;
235. }
236. }

239. int main(int argc, char const \*argv[])
240. {
241. initial();
242. *//Print(array\_a,MaxNumber);*
243. struct timeval start, end;
244. pthread\_t ptid;
245. gettimeofday(&start, NULL);
247. pthread\_barrier\_init(&barrier, NULL, thread + 1);
248. for(int i = 0; i < thread; ++i)
249. pthread\_create(&ptid, NULL, work, (void \*)(i \* NumberOfSort));
251. pthread\_barrier\_wait(&barrier);
252. meger();
254. gettimeofday(&end, NULL);
255. long long s\_usec = start.tv\_sec \* 1000000 + start.tv\_usec;
256. long long e\_usec = end.tv\_sec \* 1000000 + end.tv\_usec;
258. double Time = (double)(e\_usec - s\_usec) / 1000.0;
259. printf("sort use %.4f ms\n", Time);
260. *//Print(array\_b, MaxNumber);*
261. return 0;
262. }

上传完这段代码，同学告诉我说，这段代码在Linux和Codeblocks里运行的时间不一样(本篇博文的数据都是在Codeblocks上测得的)。然后我立马就测试了一下，发现这之间存在误差，初步猜测是由于编译器引起的。由于我不是双系统，是在虚拟机上运行的Linux系统，这可能是造成误差原因之一(个人认为可以忽略误差，虽然每组数据在不同环境下平均运行时间有差距，但其整体优化的方向是不变的)。

数据如下：



从上表可以看出，结合了多线程的快排(三数+插排+多线程)在处理前三种数组时都有明显的提升。重复数组处理时间增加的原因是：聚集元素在处理重复数组时的表现已经很好了，因为在多线程的组合中，各个线程排完序后要合并，所以增加了(三数+插排+多线程)这一组合的排序时间。因为时间原因，以上的数据，是运行相应代码10次所取得平均值。如果想要得到更精确的数据，需要大量的运行上述代码(即使存在一些不稳定的数据，也不会影响到代码优化的方向)。

参考：

<http://blog.csdn.net/zuiaituantuan/article/details/5978009>

<https://blog.csdn.net/hacker00011000/article/details/52176100>

<https://baike.baidu.com/item/%E5%B0%BE%E9%80%92%E5%BD%92/554682>

<https://blog.csdn.net/qq_25425023/article/details/72705285>