

项目十 8种排序算法的比较案例

一.设计思路

8种排序算法比较要求随机产生若干个随机数，并对他们进行排序。
项目的核心是各种排序算法的编写。因次可以将程序分为三部分：界面交互模块，随机数模块以及排序模块。
模块间关系可表示如下：



二.项目主体结构

1.主界面模块 （menu）

定义了函数 `void menu()` 作为菜单函数，进行主界面初始化。
其具体实现如下：

```
/*
 * 打印主菜单
 * */
void menu(){

    cout<<"**          排序算法比较          **\n";
    cout<<"===== \n";
    cout<<"**          1 --- 冒泡排序          **\n";
    cout<<"**          2 --- 选择排序          **\n";
    cout<<"**          3 --- 直接插入排序        **\n";
    cout<<"**          4 --- 希尔排序          **\n";
    cout<<"**          5 --- 快速排序          **\n";
    cout<<"**          6 --- 堆排序            **\n";
    cout<<"**          7 --- 归并排序          **\n";
    cout<<"**          8 --- 基数排序          **\n";
    cout<<"**          9 --- 退出程序          **\n";
    cout<<"===== \n\n";

}
```

2.随机数生成 （rand）

调用了 C 语言标准库的 `rand ()` 函数进行随机数的生成。
为了避免生成“伪随机数”，利用 `srand()` 函数置随机数种子，将 C 语言标准库的 `time()` 函数值作为随机数种子，从而达到生成真正的随机数的效果。

相关代码如下：

```

36 srand((unsigned)time(NULL)); // 置随机数种子
37 int n;
38 cout<<"请输入要产生的随机数的个数: ";
39 cin>>n; // 输入随机数个数
40
41 int* array = new int[n]; // 待排数据的数组
42 for(int i = 0; i < n; ++i)
43     array[i] = rand()%MAXNUM; // 生成随机数据
44
45 int choice;

```

3. 算法计时 (clock)

利用 C++ 标准库 `<ctime>` 中的 `clock()` 函数进行计时。

定义了 `clock_t` 类型的变量 `start` 和 `stop` 分别记录排序开始和结束的时间。

定义了 `double` 类型的变量 `time_consuming` 来记录排序算法运行时间。

相关代码如下：

```

15 using namespace std;
16
17 clock_t start, stop; // 记录排序起始和终止时间
18 double time_consuming; // 排序时间
19 int swap_num = 0; // 交换次数
20
21 // 冒泡排序

```

```

switch (choice){
case 1:
    start = clock();
    s = bubble_sort(array, n);
    stop = clock();
    time_consuming = (double)(stop - start)/CLOCKS_PER_SEC; // 格式化输出排序算法的性能
    cout<<"冒泡排序所用时间: ";
    cout<<time_consuming<<endl;
    cout<<"冒泡排序交换次数: "<<s<<endl<<endl;
    break;
case 2:
    start = clock();
    s = select_sort(array, n);

```

4. 排序模块

定义了8种排序算法的函数。为了风格统一，定义它们具有统一的接口，以及相同意义的返回值。

所有排序算法传入的参数均为 `(int array[], int n)`

其中 `int array[]` 为待排数据集，`int n` 为数据规模。
所有函数返回值均为 `int` 类型，表示该排序算法的交换 / 比较次数。
定义了全局变量 `int swap_num` 来记录排序过程中的交换次数。八个排序算法定义如下：

```
int bubble_sort(int array[],int n);    // 冒泡排序
```

```
int select_sort(int array[],int n);    // 选择排序
```

```
int insert_sort(int array[],int n);    // 插入排序
```

```
int shell_sort(int array[],int n);    // 希尔排序
```

```
int quick_sort(int array[],int n);    // 快速排序
```

```
int heap_sort(int array[],int n);    // 堆排序
```

```
int merge_sort(int array[],int n);    // 归并排序
```

```
int radix_sort(int array[],int n);    // 基数排序
```

三.排序算法实现

1.冒泡排序

1.1算法策略

- 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
- 每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
- 针对所有的元素重复以上的步骤，除了最后一个。
- 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

1.2 示意图

初始元素序列：	<u>8</u>	<u>3</u>	<u>2</u>	<u>5</u>	<u>9</u>	<u>3*</u>	<u>6</u>
第一趟排序：	3	2	5	8	3*	6	【 9 】
第二趟排序：	2	3	5	3*	6	【 8	9 】
第三趟排序：	2	3	3*	5	【 6	8	9 】
第四趟排序：	2	3	3*	【 5	6	8	9 】
第五趟排序：	2	3	【 3*	5	6	8	9 】
第六趟排序：	2	【 3	3*	5	6	8	9 】

1.3代码实现

```

154
155  /*
156  * 冒泡排序算法
157  * 输入待排数据数组和数据规模
158  * 返回排序交换次数
159  * 引入了交换标记 flag
160  * 及时判断算法的结束
161  */
162  int bubble_sort(int array[],int n){
163
164      int swap_count = 0;    // 初始化交换次数
165      int flag = 0;          // 交换标记
166      for(int i = 0; i < n-1; ++i){    // 遍历 n - 1 次
167          for(int j = 0; j < n - i - 1; ++j){    // 每次规模减小
168              if(array[j] > array[j + 1]){
169                  int t = array[j];
170                  array[j] = array[j + 1];    // 交换不满足顺序的元素
171                  array[j + 1] = t;
172                  ++swap_count;    // 记录交换次数
173                  flag = 1;
174              }
175          }
176          if(flag == 0)    // 未发生交换则提前终止算法
177              break;
178      }
179      return swap_count;
180  }
181

```

定义了变量 `int swap_count` 用于记录冒泡排序过程中的交换次数。

定义了变量 `int flag` 用于判断排序的提前终止。

从第 0 号位置元素开始循环遍历 $n - 1$ 次，每次都将在不符合此序的元素交换位置，下次遍历时遍历规模减小 1。

当某次遍历过程中，没有发生元素交换，即：`flag = 0` 说明所有元素都已经有序，排序算法可以提前终止。

1.4 算法分析

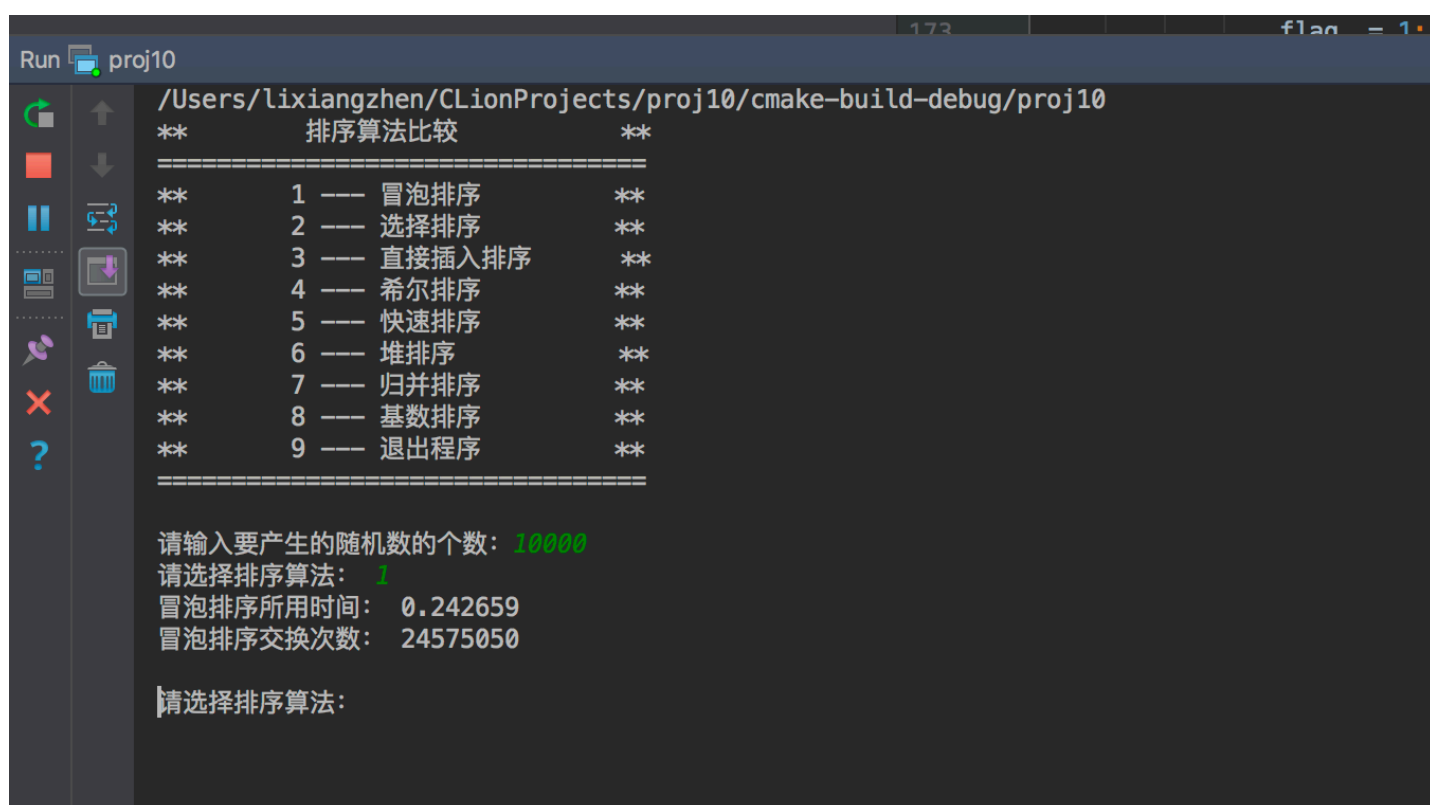
时间复杂度

- 若文件的初始状态是正序的，一趟扫描即可完成排序。所需的关键字比较次数和记录移动次数均达到最小值。所以，冒泡排序最好的时间复杂度为 $O(n)$ 。
- 若初始文件是反序的，需要进行 $n - 1$ 趟排序。每趟排序要进行 $n - i$ 次关键字的比较 ($1 \leq i \leq n - 1$)，且每次比较都必须移动记录三次来达到交换记录位置。在这种情况下，比较和移动次数均达到最大值。冒泡排序的最坏时间复杂度为 $O(n^2)$ 。
- 综上，因此冒泡排序总的平均时间复杂度为 $O(n^2)$ 。

算法稳定性

冒泡排序就是把小的元素往前调或者把大的元素往后调。比较是相邻的两个元素比较，交换也发生在这两个元素之间。所以，如果两个元素相等，是不会再把他们交换的；如果两个相等的元素没有相邻，那么即使通过前面的两两交换把两个相邻起来，这时候也不会交换，所以相同元素的前后顺序并没有改变，所以冒泡排序是一种稳定排序算法。

1.5运行情况



```
Run proj10
/Users/lixiangzhen/CLionProjects/proj10/cmake-build-debug/proj10
**      排序算法比较      **
=====
**      1 --- 冒泡排序      **
**      2 --- 选择排序      **
**      3 --- 直接插入排序  **
**      4 --- 希尔排序      **
**      5 --- 快速排序      **
**      6 --- 堆排序        **
**      7 --- 归并排序      **
**      8 --- 基数排序      **
**      9 --- 退出程序      **
=====

请输入要产生的随机数的个数: 10000
请选择排序算法: 1
冒泡排序所用时间: 0.242659
冒泡排序交换次数: 24575050

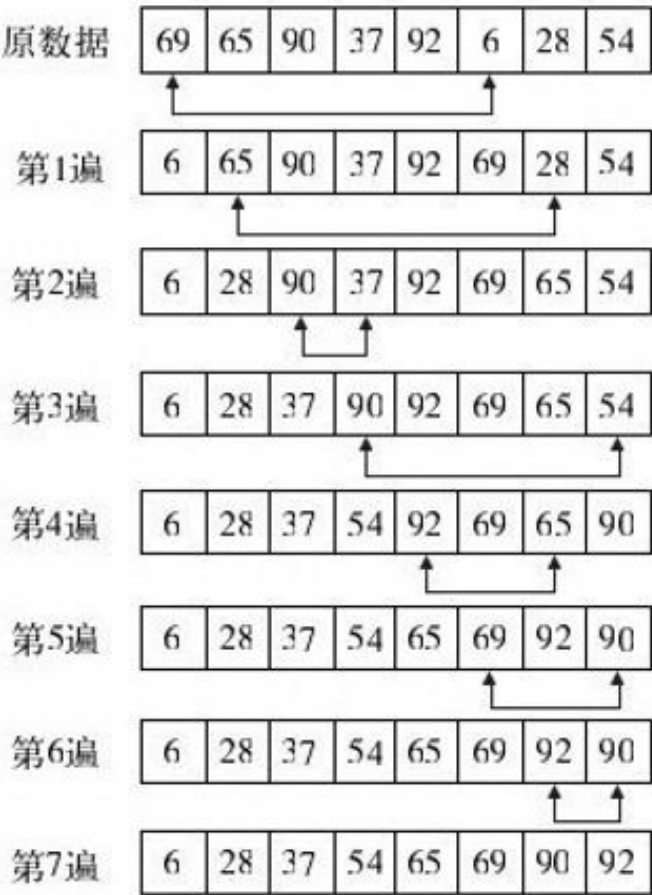
请选择排序算法:
```

2.选择排序

1.1算法策略

比如在一个长度为N的无序数组中，在第一趟遍历N个数据，找出其中最小的数值与第一个元素交换，第二趟遍历剩下的N-1个数据，找出其中最小的数值与第二个元素交换.....第N-1趟遍历剩下的2个数据，找出其中最小的数值与第N-1个元素交换，至此选择排序完成。

1.2 示意图



1.3代码实现

```

182
183  /*
184   * 选择排序算法
185   * 输入待排数据数组和数据规模
186   * 返回排序交换次数
187   * */
188  int select_sort(int array[],int n){
189
190      int swap_count = 0;    // 初始化交换次数
191      for(int i = 0;i < n-1;++i){
192          int mintag = i;    // 最小元素下标
193          int min = 100001;  // 初始化最小值
194          for(int j = i;j < n;++j){    // 遍历找出最小元素
195              if(array[j] < min){
196                  min = array[j];
197                  mintag = j;
198              }
199          }
200          int t = array[i];    // 最小元素交换到正确位置
201          array[i] = array[mintag];
202          array[mintag] = t;
203          ++swap_count;    // 记录交换次数
204      }
205
206      return swap_count;
207  }
208

```

定义了变量 `int swap_count` 用于记录选择排序过程中的交换次数。

定义了变量 `int mintag` 用于记录每趟遍历中找到的最小元素下标。

从第 0 号位置元素开始循环遍历 $n - 1$ 次，每次都把最小的元素交换到未排序序列队首位置，下次遍历时遍历规模减小 1。

当便利 $n - 1$ 次后，所有元素均有序。

1.4 算法分析

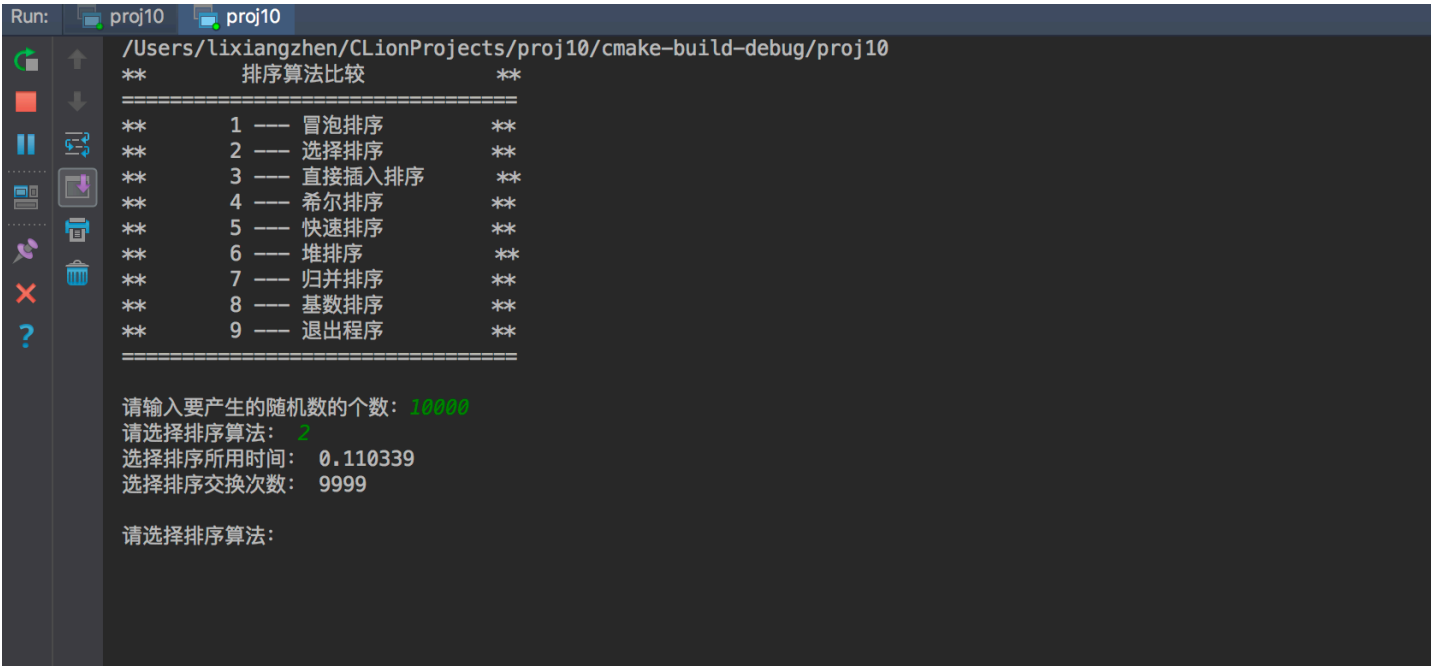
时间复杂度

- 选择排序的交换操作介于 0 和 $(n - 1)$ 次之间。
- 选择排序的比较操作为 $n(n - 1) / 2$ 次之间。
- 选择排序的赋值操作介于 0 和 $3(n - 1)$ 次之间。
- 比较次数 $O(n^2)$ ，比较次数与关键字的初始状态无关，总的比较次数 $N = (n - 1) + (n - 2) + \dots + 1 = n(n - 1) / 2$ 。
- 交换次数 $O(n)$ ，最好情况是，已经有序，交换 0 次；最坏情况交换 $n - 1$ 次，逆序交换 $n / 2$ 次。

算法稳定性

选择排序是给每个位置选择当前元素最小的，比如给第一个位置选择最小的，在剩余元素里面给第二个元素选择第二小的，依次类推，直到第n-1个元素，第n个元素不用选择了，因为只剩下它一个最大的元素了。那么，在一趟选择，如果一个元素比当前元素小，而该小的元素又出现在一个和当前元素相等的元素后面，那么交换后稳定性就被破坏了。

1.5运行情况



3.直接插入排序

1.1算法策略

- 每次从无序表中取出第一个元素，把它插入到有序表的合适位置，使有序表仍然有序。
- 第一趟比较前两个数，然后把第二个数按大小插入到有序表中
- 第二趟把第三个数据与前两个数从后向前扫描，把第三个数按大小插入到有序表中
- 依次进行下去，进行了(n-1)趟扫描以后就完成了整个排序过程。

直接插入排序是由两层嵌套循环组成的。外层循环标识并决定待比较的数值。内层循环为待比较数值确定其最终位置。直接插入排序是将待比较的数值与它的前一个数值进行比较，所以外层循环是从第二个数值开始的。当前一数值比待比较数值大的情况

下继续循环比较，直到找到比待比较数值小的并将待比较数值置入其后一位置，结束该次循环。

1.2 示意图

初始元素序列：	【8】	3	2	5	9	3*	6
第一趟排序：	【3	8】	2	5	9	3*	6
第二趟排序：	【2	3	8】	5	9	3*	6
第三趟排序：	【2	3	5	8】	9	3*	6
第四趟排序：	【2	3	5	8	9】	3*	6
第五趟排序：	【2	3	3*	5	8	9】	6
第六趟排序：	【2	3	3*	5	6	8	9】

1.3 代码实现

```
209  /*
210   * 插入排序算法
211   * 输入待排数据数组和数据规模
212   * 返回排序交换次数
213   */
214  int insert_sort(int array[],int n){
215      |
216      int swap_count = 0;
217      for(int i = 1;i < n;++i){
218          int current = array[i];
219          int j;
220          for(j = i;j > 0&&current < array[j - 1];--j){
221              array[j] = array[j - 1];
222              ++swap_count;
223          }
224          array[j] = current;
225      }
226
227      return swap_count;
228  }
```

定义了变量 `int swap_count` 用于记录冒泡排序过程中的交换次数。

定义了变量 `int current` 用于记录当前待插入元素。

每次将新插入的元素不断向前比较，直到找到合适的插入位置。

1.4 算法分析

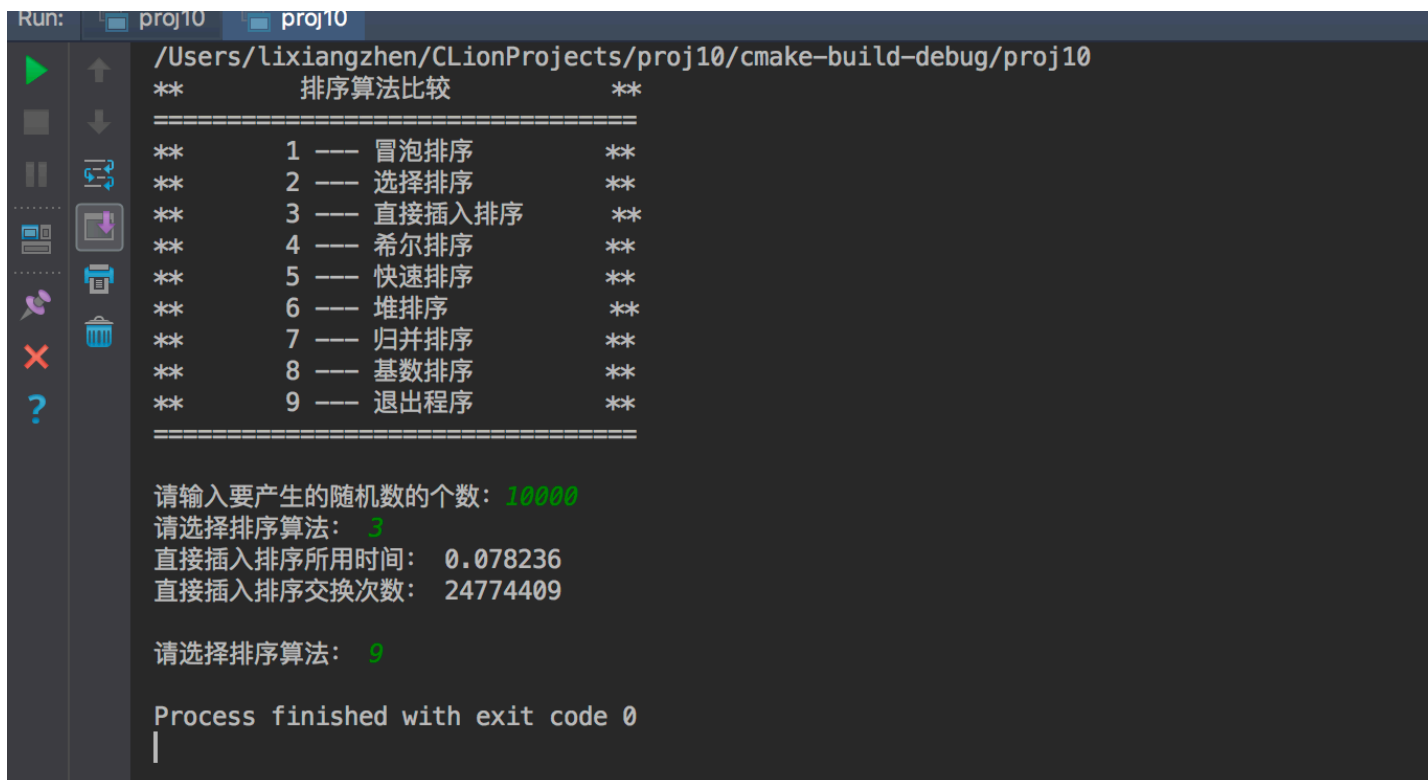
时间复杂度

- 当数据正序时，执行效率最好，每次插入都不用移动前面的元素，时间复杂度为 $O(N)$ 。
- 当数据反序时，执行效率最差，每次插入都要前面的元素后移，时间复杂度为 $O(N^2)$ 。
- 所以，数据越接近正序，直接插入排序的算法性能越好。

算法稳定性

直接插入排序的过程中，不需要改变相等数值元素的位置，所以它是稳定的算法。

1.5 运行情况



```
Run: proj10 proj10
/Users/lixiangzhen/CLionProjects/proj10/cmake-build-debug/proj10
**      排序算法比较      **
=====
**      1 --- 冒泡排序      **
**      2 --- 选择排序      **
**      3 --- 直接插入排序  **
**      4 --- 希尔排序      **
**      5 --- 快速排序      **
**      6 --- 堆排序        **
**      7 --- 归并排序      **
**      8 --- 基数排序      **
**      9 --- 退出程序      **
=====

请输入要产生的随机数的个数: 10000
请选择排序算法: 3
直接插入排序所用时间: 0.078236
直接插入排序交换次数: 24774409

请选择排序算法: 9

Process finished with exit code 0
|
```

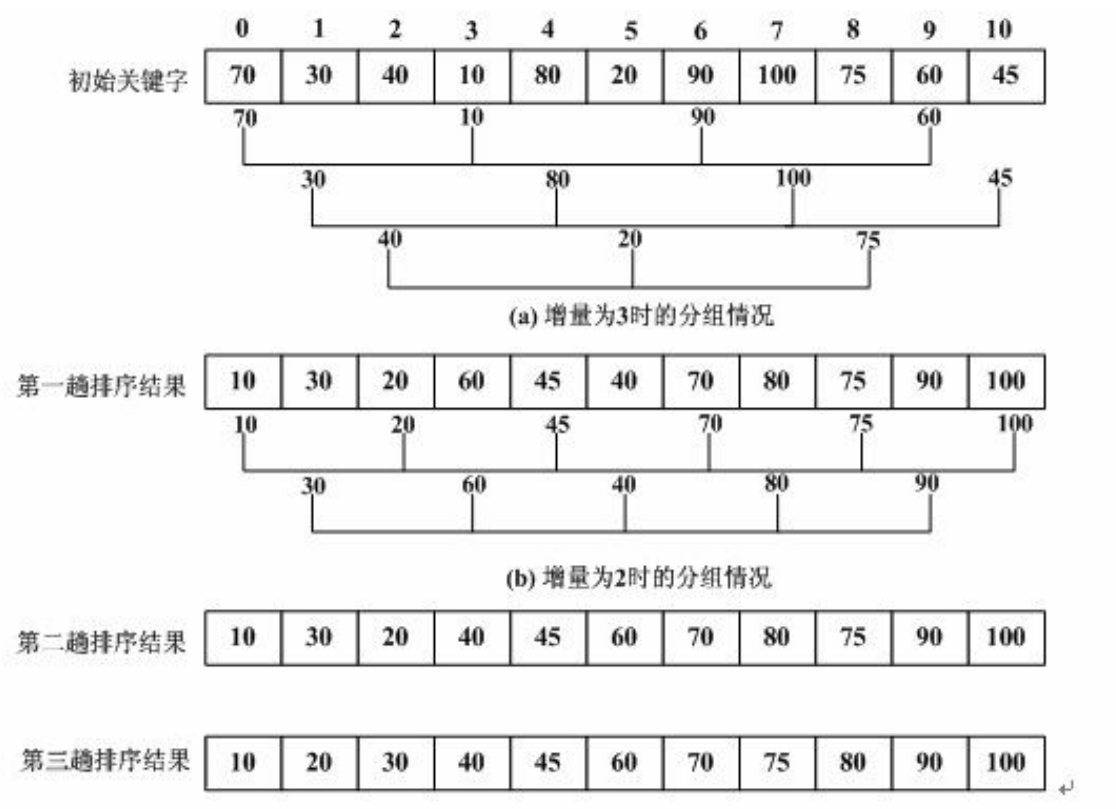
4. 希尔排序

1.1 算法策略

- 先取一个小于 n 的整数 d_1 作为第一个增量，把文件的全部记录分组。

- 所有距离为 d_1 的倍数的记录放在同一个组中。先在各组内进行直接插入排序
- 然后，取第二个增量 $d_2 < d_1$ 重复上述的分组和排序
- 直至所取的增量 $= 1$ ($< \dots < d_2 < d_1$)，即所有记录放在同一组中进行直接插入排序为止。

1.2 示意图



1.3 代码实现

```

229
230
231  /*
232  * 希尔排序算法
233  * 输入待排数据数组和数据规模
234  * 返回排序交换次数
235  * 自定义了 0, 1, 5, 19 为希尔排序序列数
236  */
237  int shell_sort(int array[],int n){
238
239      int swap_count = 0;          // 初始化交换次数
240      int shell[4] = {19,5,1,0};  // 自定义希尔数
241      int beg = 0;
242      while(beg < 4&&shell[beg] > n)    // 确定初始希尔数
243          ++beg;
244      for(;beg < 4; ++beg)    // 循环进行希尔排序
245          for(int i = 1; i < n; ++i){
246              int current = array[i];
247              int j;
248              for(j = i; j > 0&&current < array[j - shell[beg]]; j -= shell[beg]){    // 记录交换次数
249                  array[j] = array[j - shell[beg]];
250                  ++swap_count;
251              }
252              array[j] = current;
253          }
254
255      return swap_count;
256  }
257

```

定义了变量 `int swap_count` 用于记录冒泡排序过程中的交换次数。

定义了变量 `int shell [4]` 用于记录所选的希尔序列。

现根据数据规模，选择一个希尔距离，进行跳跃式插入排序。随着希尔距离减小，数据集的有序性也不断加强。直到距离为1，直接插入排序。

1.4算法分析

时间复杂度

Shell排序的执行时间依赖于增量序列。好的增量序列的共同特征：① 最后一个增量必须为1；

② 应该尽量避免序列中的值(尤其是相邻的值)互为倍数的情况。

有人通过大量的实验，给出了较好的结果：当n较大时，比较和移动的次数约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 之间。

算法稳定性

希尔排序过程中，对元素进行了分组插入排序，具有相同值的元素可能被排到不同组中，无法保证其相对位置不变。因此希尔排序不是稳定的。

1.5运行情况

```
Run: proj10 proj10
/Users/lixiangzhen/CLionProjects/proj10/cmake-build-debug/proj10
**      排序算法比较      **
=====
**      1  ---  冒泡排序      **
**      2  ---  选择排序      **
**      3  ---  直接插入排序  **
**      4  ---  希尔排序      **
**      5  ---  快速排序      **
**      6  ---  堆排序        **
**      7  ---  归并排序      **
**      8  ---  基数排序      **
**      9  ---  退出程序      **
=====

请输入要产生的随机数的个数: 10000
请选择排序算法: 4
希尔排序所用时间: 0.010466
希尔排序交换次数: 1487977

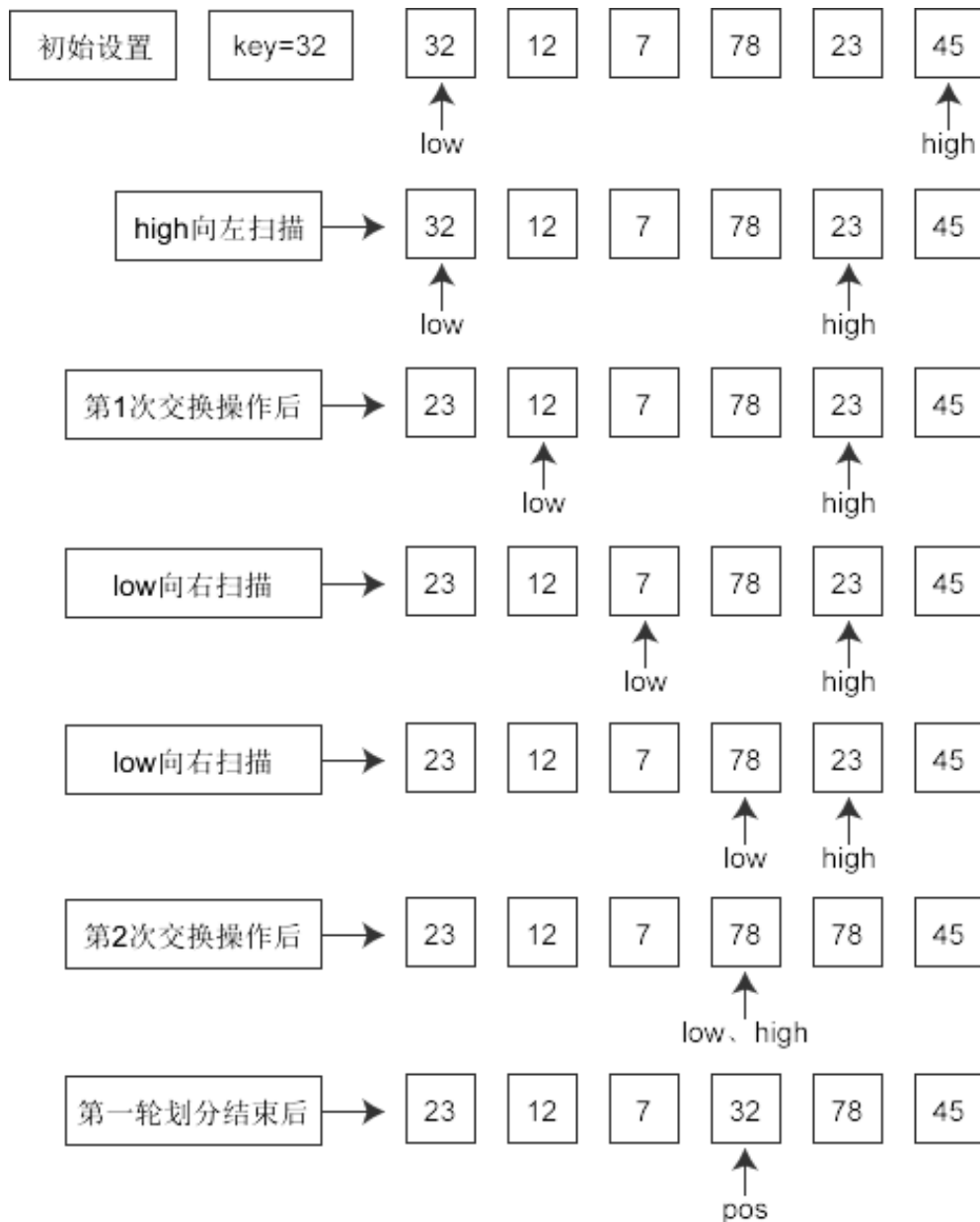
请选择排序算法:
```

5.快速排序

1.1 算法策略

- 在待排序的元素任取一个元素作为基准(通常选第一个元素，但最的选择方法是从待排序元素中随机选取一个作为基准)，称为基准元素
- 将待排序的元素进行分区，比基准元素大的元素放在它的右边，比其小的放在它的左边；
- 对左右两个分区重复以上步骤直到所有元素都是有序的。

1.2 示意图



1.3代码实现

```

312  /*
313   * 快速排序算法接口
314   * 适配其他算法接口
315   * 输入待排数据数组和数据规模
316   * 返回排序交换次数
317   */
318  int quick_sort( int array[], int n ) {
319      Qsort( array, 0, n-1 );    // 调用快排递归算法
320      return swap_num;
321  }
322
323

```

定义了快速排序的接口函数 `int quick_sort(int array[], int n)` 提供统一的

排序接口。

内部调用 `void Qsort(int A[], int Left, int Right)` 以递归的策略对数组进行排序。

该函数实现如下：

```
284  /*
285  * 快速排序算法核心函数
286  * 在待排元素少于100时直接执行插入排序
287  * 大于100 时，将元素按中值划分，进行快排
288  */
289  void Qsort( int A[], int Left, int Right ) {
290      int tag, Low, High;
291
292      if ( 100 <= Right-Left ) { // 根据数据规模判断是否需要快排
293          tag = Median( A, Left, Right ); // 找出数据中值
294          Low = Left; High = Right-1;
295          while ( 1 ) { // 按数据段中值划分元素
296              while ( A[++Low] < tag );
297              while ( A[--High] > tag );
298              if ( Low < High ){
299                  ++swap_num;
300                  swap( A[Low], A[High] );
301              }
302              else break;
303          }
304          swap( A[Low], A[Right-1] );
305          ++swap_num;
306          Qsort( A, Left, Low-1 ); // 对中值左右数据段递归执行快排
307          Qsort( A, Low+1, Right );
308      }
309      else insert_sort( A+Left, Right-Left+1 ); // 数据量较小直接执行快排
310  }
```

三平均分区法，选用待排数组最左边、最右边和最中间的三个元素的中间值作为中轴。

定义了变量 `int tag, high, low` 作为数据分段的标记。

调用 `int Median(int A[], int Left, int Right)` 函数调整当前序列的中间值，以其为轴划分元素。

将左右元素划分好后，递归的调用快速排序函数，对左右区间进行划分。

`int Median()` 函数实现如下：

```

257
258  /*
259  * 中值函数
260  * 快速排序算法的辅助函数
261  * 将指定序列首, 尾, 中间三个数排序, 并返回中间值
262  * 中间值别交换到最右侧元素的左侧
263  * 便于后续序列按中值大小划分的执行
264  */
265  int Median( int A[], int Left, int Right ) {
266      int Center = (Left+Right) / 2; // 序列中值
267      if ( A[Left] > A[Center] ){ // 多次比较, 交换, 使三个数有序
268          ++swap_num;
269          swap( A[Left], A[Center] );
270      }
271      if ( A[Left] > A[Right] ){
272          ++swap_num;
273          swap( A[Left], A[Right] );
274      }
275      if ( A[Center] > A[Right] ){
276          ++swap_num;
277          swap( A[Center], A[Right] );
278      }
279      swap( A[Center], A[Right-1] ); // 中值藏到最右值的左侧
280      ++swap_num;
281      return A[Right-1];
282  }
283

```

定义了变量 `int center` 用于标记中间变量。进过三次比较交换挑战元素位置。

1.4 算法分析

时间复杂度

- 当分区选取的基准元素为待排序元素中的最大或最小值时, 为最坏的情况, 时间复杂度和直接插入排序的一样, 移动次数达到最大值

$C_{max} = 1+2+\dots+(n-1) = n*(n-1)/2 = O(n^2)$ 此时最好时间复杂为 $O(n^2)$

- 当分区选取的基准元素为待排序元素中的"中值", 为最好的情况, 时间复杂度为 $O(n\log_2 n)$ 。
- 快速排序的空间复杂度为 $O(\log_2 n)$ 。

算法稳定性

当待排序元素类似[6,1,3,7,3]且基准元素为6时, 经过分区, 形成[1,3,3,6,7],两个3的相对位置发生了改变, 所以快速排序是一种不稳定排序。

1.5 运行情况

```
Run: proj10 proj10
/Users/lixiangzhen/CLionProjects/proj10/cmake-build-debug/proj10
**      排序算法比较      **
=====
**      1 --- 冒泡排序      **
**      2 --- 选择排序      **
**      3 --- 直接插入排序  **
**      4 --- 希尔排序      **
**      5 --- 快速排序      **
**      6 --- 堆排序        **
**      7 --- 归并排序      **
**      8 --- 基数排序      **
**      9 --- 退出程序      **
=====

请输入要产生的随机数的个数: 10000
请选择排序算法: 5
快速排序所用时间: 0.001537
快速排序交换次数: 18128

请选择排序算法: 9

Process finished with exit code 0
|
```

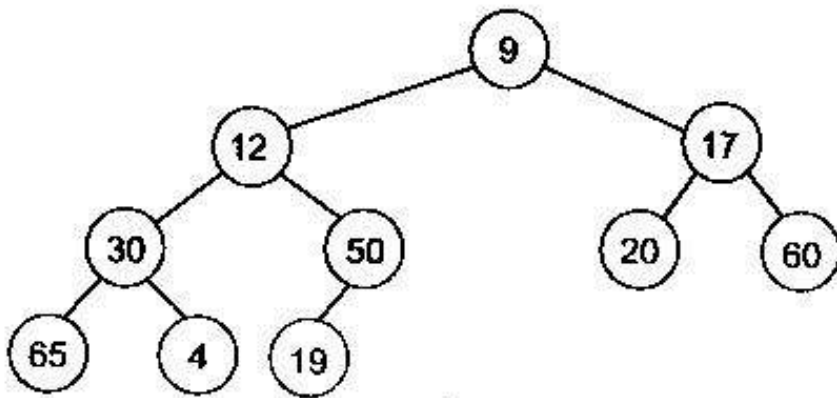
6.堆排序

1.1 算法策略

- 先将初始文件 $R[1..n]$ 建成一个大根堆，此堆为初始的无序区
- 再将关键字最大的记录 $R[1]$ （即堆顶）和无序区的最后一个记录 $R[n]$ 交换，由此得到新的无序区 $R[1..n-1]$ 和有序区 $R[n]$ ，且满足 $R[1..n-1].keys \leq R[n].key$
- 由于交换后新的根 $R[1]$ 可能违反堆性质，故应将当前无序区 $R[1..n-1]$ 调整为堆。然后再次将 $R[1..n-1]$ 中关键字最大的记录 $R[1]$ 和该区间的最后一个记录 $R[n-1]$ 交换，由此得到新的无序区 $R[1..n-2]$ 和有序区 $R[n-1..n]$ ，且仍满足关系 $R[1..n-2].keys \leq R[n-1..n].keys$ ，同样要将 $R[1..n-2]$ 调整为堆。……直到无序区只有一个元素为止。

1.2 示意图

```
int A[0] = {9,12,17,30,50,20,60,65,4,49};
```



初始表

1.3代码实现

```
349  /*
350  * 堆排序算法
351  * 输入待排序数据数组和数据规模
352  * 返回排序交换次数
353  * 内部调用向下过滤算法实现建堆以及删除操作
354  * */
355  int heap_sort(int array[], int n){
356
357      swap_num = 0;          // 初始化交换次数
358      for(int i = n/2-1; i >= 0; --i){
359          percdwn(array,i,n);    // 将数据调整为最大堆
360      }
361      for(int i = n-1; i > 0; --i){    // 每次删除堆顶元素，将其调到数组尾部
362          swap(array[0],array[i]);
363          ++swap_num;
364          percdwn(array,0,i);    // 向下过滤保持堆序
365      }
366      return swap_num;
367  }
```

将数据集 `array[]` 调整为最大堆。

调整过程调用 `void percdwn(int array[], int position, int n)` 函数。

随后依次删除堆顶元素，相当于每次将最大元素调至尾部。

`void percdwn()` 函数实现如下

```

325 * 向下过滤算法
326 * 堆排序辅助算法
327 * 在左右子树均已经保持的堆序的情况下
328 * 调节根节点位置使整棵树保持堆序
329 */
330 void percdown(int array[], int position, int n){
331     int parent, child;
332
333     int current = array[position]; // 记录根元素
334     for(parent = position; parent*2 + 1 < n; parent = child){ // 向下过滤寻找插入位置
335         child = parent*2 + 1;
336         if(child < n-1 && array[child] < array[child + 1]) // 找到左右子女中的较大者
337             ++child;
338         if(array[child] > current){ // 向下过滤
339             array[parent] = array[child];
340             ++swap_num;
341         }
342
343         else
344             break;
345     }
346     array[parent] = current; // 插到合适位置
347 }
348

```

定义了变量 `int parent, child` 作为正在比较的一对父子。
每次将该元素向下过滤，使整个堆回复为堆序。

1.4 算法分析

时间复杂度

堆的存储表示是顺序的。因为堆所对应的二叉树为完全二叉树，而完全二叉树通常采用顺序存储方式。当想得到一个序列中第 k 个最小的元素之前的部分排序序列，最好采用堆排序。因为堆排序的时间复杂度是 $O(n+k\log_2 n)$ ，若 $k \leq n/\log_2 n$ ，则可得到的时间复杂度为 $O(n)$ 。

算法稳定性

堆排序是一种不稳定的排序方法。因为在堆的调整过程中，关键字进行比较和交换所走的是该结点到叶子结点的一条路径，因此对于相同的关键字就可能出现排在后面的关键字被交换到前面来的情况。

1.5 运行情况

```
Run proj10
/Users/lixiangzhen/CLionProjects/proj10/cmake-build-debug/proj10
**      排序算法比较      **
=====
**      1 --- 冒泡排序      **
**      2 --- 选择排序      **
**      3 --- 直接插入排序  **
**      4 --- 希尔排序      **
**      5 --- 快速排序      **
**      6 --- 堆排序        **
**      7 --- 归并排序      **
**      8 --- 基数排序      **
**      9 --- 退出程序      **
=====

请输入要产生的随机数的个数: 10000
请选择排序算法: 6
堆排序所用时间: 0.001899
堆排序交换次数: 124280

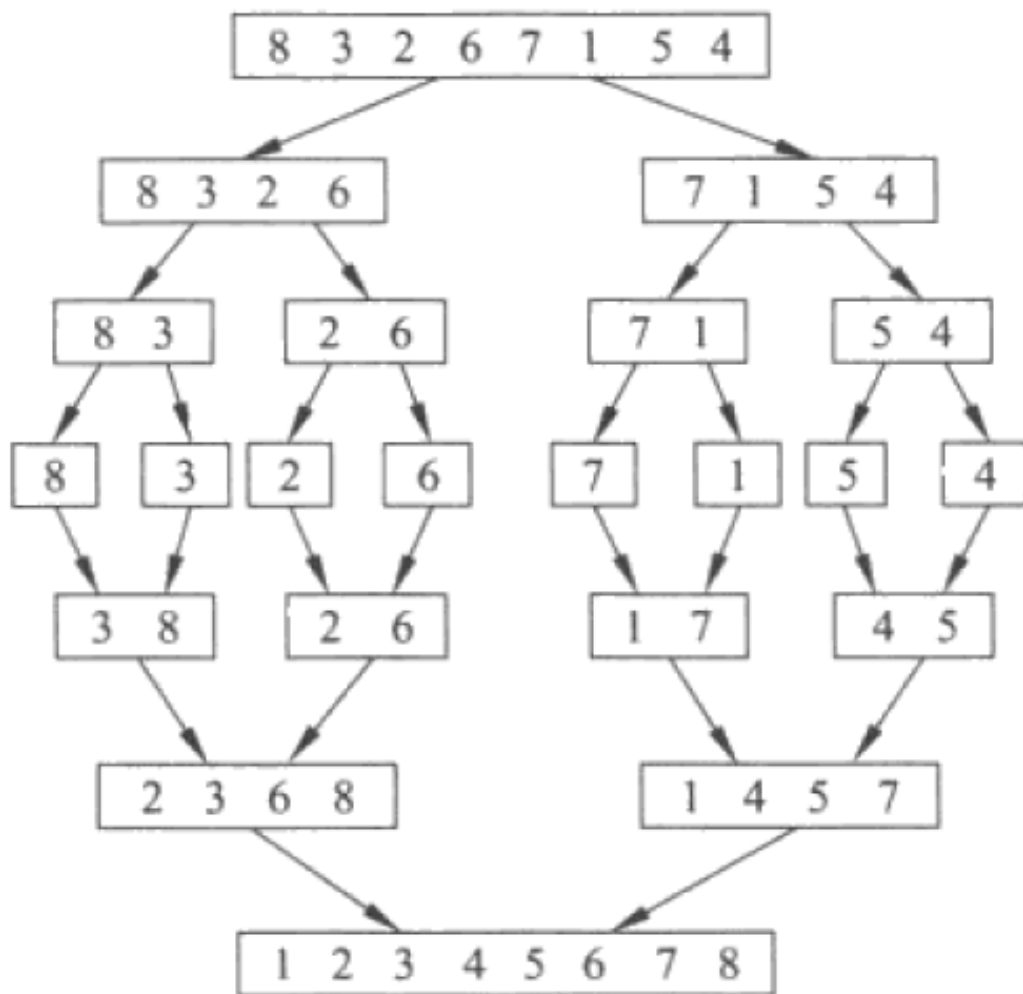
请选择排序算法:
```

7.归并排序

1.1 算法策略

- 比较 $a[i]$ 和 $b[j]$ 的大小，若 $a[i] \leq b[j]$ ，则将第一个有序表中的元素 $a[i]$ 复制到 $r[k]$ 中，并令 i 和 k 分别加上1
- 否则将第二个有序表中的元素 $b[j]$ 复制到 $r[k]$ 中，并令 j 和 k 分别加上1
- 如此循环下去，直到其中一个有序表取完
- 然后再将另一个有序表中剩余的元素复制到 r 中从下标 k 到下标 t 的单元
- 归并排序的算法我们通常用递归实现，先把待排序区间 $[s, t]$ 以中点二分，接着把左边子区间排序，再把右边子区间排序，最后把左区间和右区间用一次归并操作合并成有序的区间 $[s, t]$ 。

1.2 示意图



1.3代码实现

```

414
415  /*
416   * 归并排序算法接口
417   * 适配其他排序算法接口
418   * 输入待排序数据数组和数据规模
419   * 返回排序交换次数
420   */
421  int merge_sort(int array[],int n){
422      swap_num = 0; // 初始化交换次数
423      int* temp_array = new int[n + 1]; // 开一个辅助数组，用于合并
424      msort(array,temp_array,0,n-1); // 调用归并算法
425      delete[] temp_array;
426      return swap_num;
427  }
428

```

定义了归并排序的接口函数 `int merge_sort(int array[], int n)` 提供统一的排序接口。

内部调用

`void msort(int array[],int temp_array[],int left, int right_end)` 以递

归的策略对数组进行排序。

该函数实现如下：

```
98
99  /*
100  * 归并排序算法
101  * 递归将左右序列排序
102  * 然后将其合并
103  * 调用 merge 函数
104  * */
105 void msort(int array[],int temp_array[],int left, int right_end){
106
107     if(left < right_end){ // 判断序列合法行
108         int center = (left + right_end)/2;
109         msort(array,temp_array,left,center); // 递归处理左右子列
110         msort(array,temp_array,center + 1,right_end);
111         merge(array,temp_array,left,center + 1,right_end); // 合并左右子列
112     }
113 }
114
```

将区间等分为左右两段，对左右两部分分别递归进行归并排序。

传入参数 `int left, int right_end` 作为划分数据区间的标记。

传入参数 `int temp_array[]` 作为辅助数组，存储归并过程中的中间结果。

调用

```
void merge(int array[],int temp_array[],int left,int right,int right_end)
```

函数对归并划分好的左右元素集进行合并。

`void merge()` 函数实现如下：

```

370  /*
371  * 归并函数
372  * 归并排序算法辅助函数
373  * 将两个有序序列归并为一个
374  * */
375  void merge(int array[],int temp_array[],int left,int right,int right_end){
376      int left_end = right - 1;    // 记录左侧数组终止位置
377      int current = left;          // 记录辅助数组中元素位置
378      int size = right_end - left + 1;    // 数据规模
379      while(left <= left_end&&right <= right_end){        // 从左右两序列头开始比较遍历
380          ++swap_num;
381          if(array[left] <= array[right])
382              temp_array[current++] = array[left++];
383          else
384              temp_array[current++] = array[right++];
385      }
386  }
387
388      while (left <= left_end){        // 处理左右数组长度不同的情况
389          temp_array[current++] = array[left++];
390      }
391      while (right <= right_end){
392          temp_array[current++] = array[right++];
393      }
394
395      for(int i = 0;i < size;++i,--right_end){        // 将排好的序列写入原数组
396          array[right_end] = temp_array[right_end];
397      }
398  }

```

变量 `int left, int right` 作为指向两端数据的标记，从首至尾顺序比较遍历，按大小归并元素。

1.4算法分析

时间复杂度

- 比较操作的次数介于 $(n \log n) / 2$ 和 $n \log n - n + 1$ 。
- 赋值操作的次数是 $(2n \log n)$ 。
- 综合来看，时间复杂度为 $O(n \log_2 n)$ 这是该算法中最好、最坏和平均的时间性能。

算法稳定性

归并排序比较占用内存，但却是一种效率高且稳定的算法。

1.5运行情况

```
Run proj10
/Users/lixiangzhen/CLionProjects/proj10/cmake-build-debug/proj10
**      排序算法比较      **
=====
**      1  ——  冒泡排序      **
**      2  ——  选择排序      **
**      3  ——  直接插入排序   **
**      4  ——  希尔排序      **
**      5  ——  快速排序      **
**      6  ——  堆排序        **
**      7  ——  归并排序      **
**      8  ——  基数排序      **
**      9  ——  退出程序      **
=====

请输入要产生的随机数的个数: 10000
请选择排序算法: 7
归并排序所用时间: 0.002015
归并排序比较次数: 120454

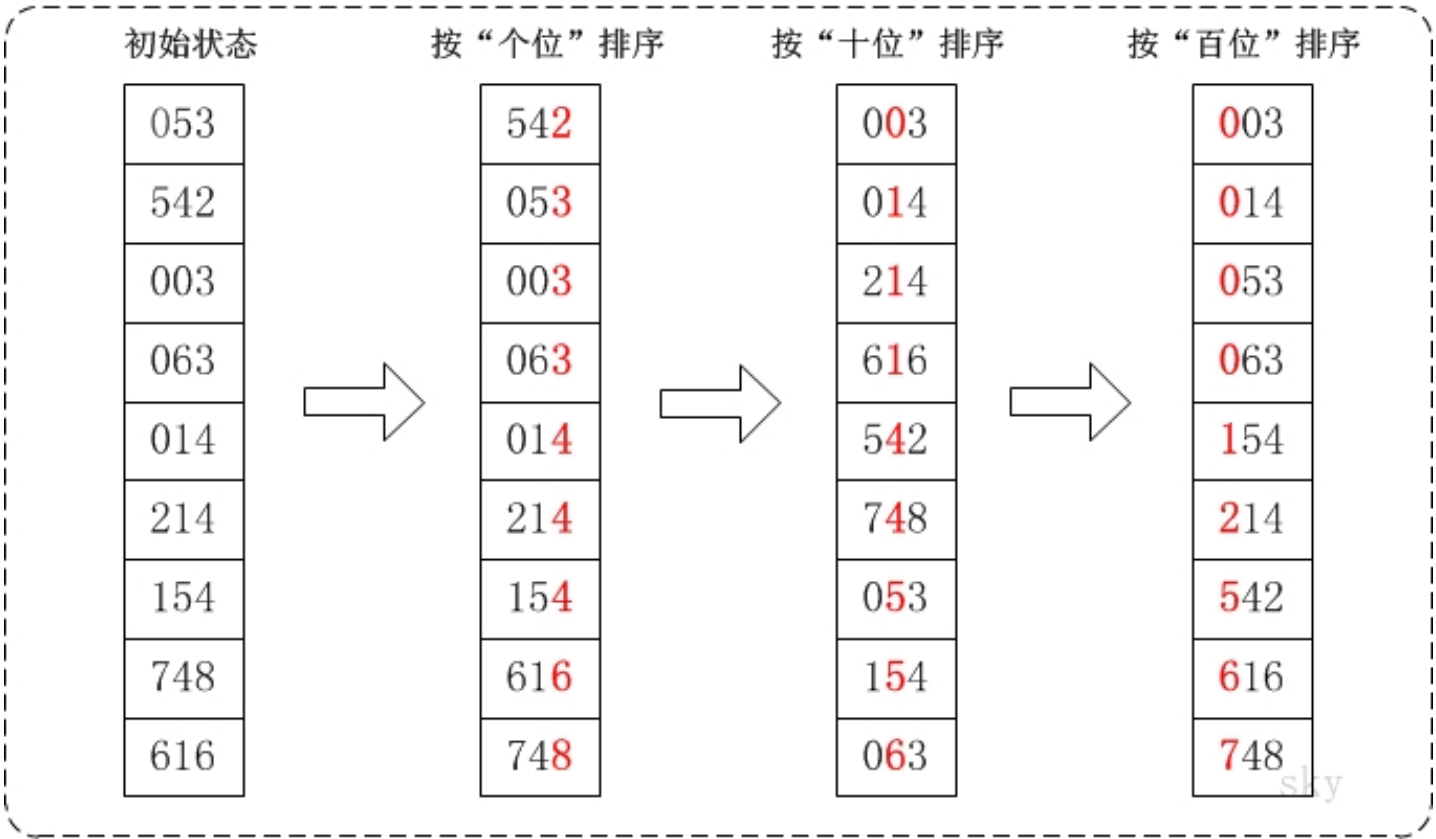
请选择排序算法:
```

8.基数排序

1.1 算法策略

- 将整形10进制按每位拆分，然后从低位到高位依次比较各个位。主要分为两个过程：
- (1)分配，先从个位开始，根据位值(0-9)分别放到0~9号桶中
- (2)收集，再将放置在0~9号桶中的数据按顺序放到数组中
- 重复(1)(2)过程，从个位到最高位

1.2 示意图



1.3 Code Implementation

```

9  /*
10  * 基数排序算法
11  * 输入待排数据数组和数据规模
12  * 返回排序交换次数
13  * 按低位到高位顺序排
14  */
15  int radix_sort(int array[], int n)
16  {
17      int *temp = new int[n];    // 辅助数组，存储过程中的数据集
18      int bucket[10];           // 0-9 十个桶
19      int i, j, k;
20      int radix = 1;            // 排序位次
21      for(i = 1; i <= 5; i++)
22      {
23          for(j = 0; j < 10; j++)    // 初始化桶
24              bucket[j] = 0;
25          for(j = 0; j < n; j++)    // 桶排序
26          {
27              k = (array[j] / radix) % 10;
28              bucket[k]++;
29          }
30          for(j = 1; j < 10; j++)    // 算出个桶元素在辅助数组中的位次
31              bucket[j] = bucket[j - 1] + bucket[j];
32          for(j = n - 1; j >= 0; j--)    // 桶中数据倒入辅助数组
33          {
34              k = (array[j] / radix) % 10;
35              temp[bucket[k] - 1] = array[j];
36              bucket[k]--;
37          }
38          for(j = 0; j < n; j++)    // 辅助数组数据写回原数据集
39              array[j] = temp[j];
40          radix = radix * 10;        // 增加排序位次
41      }
42      delete[] temp;
43      return 0;
44  }

```

定义了变量 `int *temp` 作为辅助数组，用于记录排序过程中中间结果。

定义了变量 `int bucket[10]` 用于记录每次排序的结果。

按照低位优先的顺序，从个位开始排序，每次排好后，将中间结果写回原数据集，每次排序保留前一次排序结果的相对顺序。

1.4 算法分析

时间复杂度

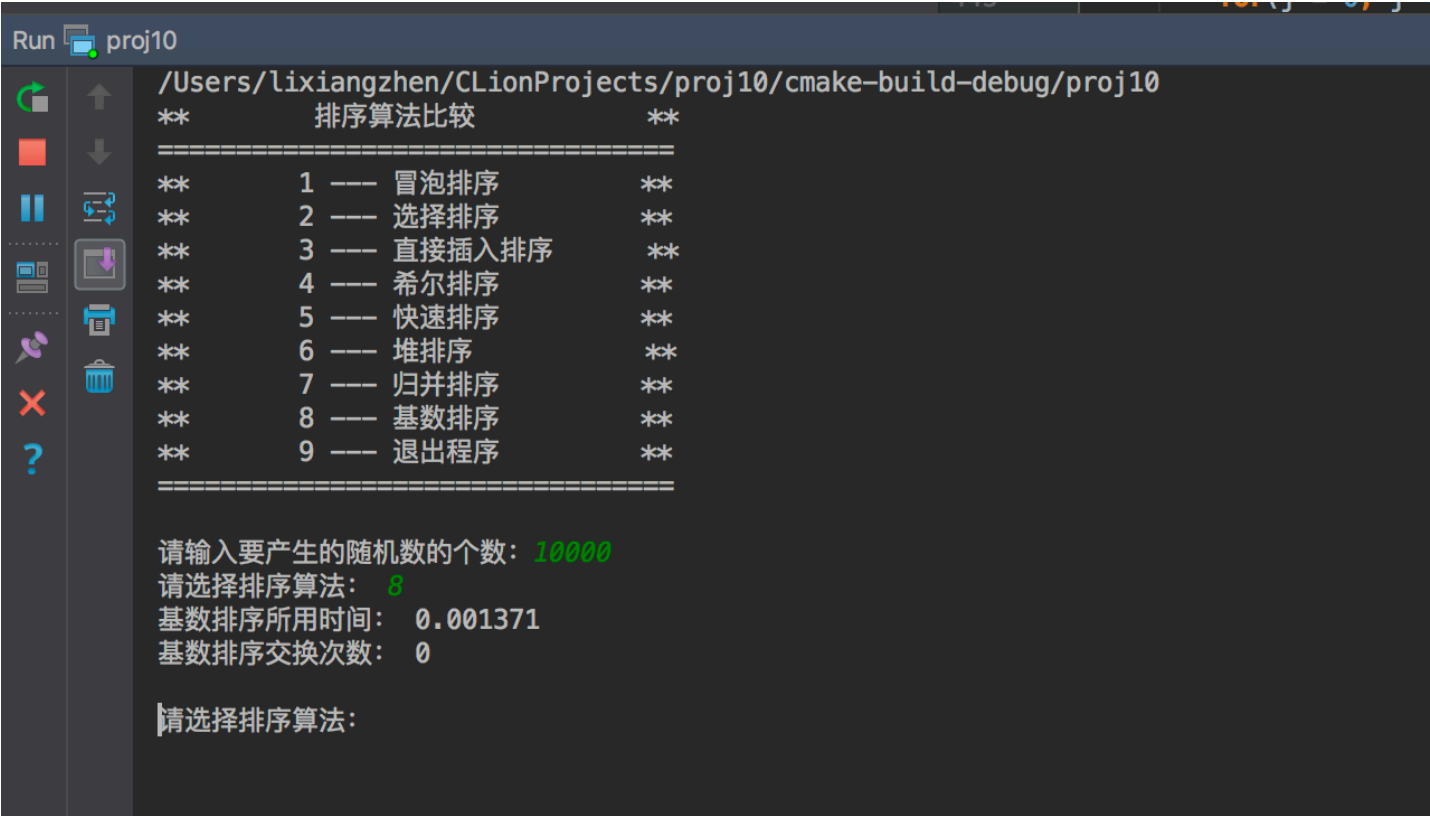
设待排序列为 n 个记录， d 个关键码，关键码的取值范围为 $radix$ ，则进行链式基数排序的时间复杂度为 $O(d(n+radix))$ ，其中，一趟分配时间复杂度为 $O(n)$ ，一趟收集时间复

杂度为 $O(radix)$ ，共进行 d 趟分配和收集。

算法稳定性

低位优先的基数排序因为在是把低位按顺序映射到一个临时序列中去,是依次序映射,没有涉及到数据位置的变动.然后再按高位顺序映射.所以相同元素也是按次序映射过去.所以是稳定的。

1.5运行情况



四.测试

1.随机数生成测试

测试用例

生成随机数量: 50000

程序执行情况如下：



2.排序测试

程序执行情况如下：

Run proj10

/Users/lixiangzhen/CLionProjects/proj10/cmake-build-debug/proj10

```

**          排序算法比较          **
=====
**      1  ---  冒泡排序          **
**      2  ---  选择排序          **
**      3  ---  直接插入排序      **
**      4  ---  希尔排序          **
**      5  ---  快速排序          **
**      6  ---  堆排序            **
**      7  ---  归并排序          **
**      8  ---  基数排序          **
**      9  ---  退出程序          **
=====

```

请输入要产生的随机数的个数: 50000

请选择排序算法: 1

冒泡排序所用时间: 6.22632

冒泡排序交换次数: 623955191

请选择排序算法: 2

选择排序所用时间: 2.43809

选择排序交换次数: 49999

请选择排序算法: 3

直接插入排序所用时间: 0.00026

直接插入排序交换次数: 0

请选择排序算法: 4

希尔排序所用时间: 0.001135

希尔排序交换次数: 0

请选择排序算法: 5

快速排序所用时间: 0.001828

快速排序交换次数: 3214

请选择排序算法: 6

堆排序所用时间: 0.008654

堆排序交换次数: 773851

请选择排序算法: 7

归并排序所用时间: 0.00656

归并排序比较次数: 401952

请选择排序算法: 8

基数排序所用时间: 0.005255

基数排序交换次数: 0