项目说明文档

数据结构课程设计

——8种排序算法的比较案例

作 者 姓 名： 曹峰源

学 号： 1951328

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目录

[1. 冒泡排序 3](#_Toc60248995)

[1.1 算法分析 3](#_Toc60248996)

[1.2 算法功能实现 3](#_Toc60248997)

[2. 选择排序 5](#_Toc60248998)

[2.1 算法分析 5](#_Toc60248999)

[2.2 算法功能实现 5](#_Toc60249000)

[3. 直接插入排序 7](#_Toc60249001)

[3.1 算法分析 7](#_Toc60249002)

[3.2 算法功能实现 8](#_Toc60249003)

[4. 希尔排序 9](#_Toc60249004)

[4.1 算法分析 9](#_Toc60249005)

[4.2 算法功能实现 10](#_Toc60249006)

[5. 快速排序 12](#_Toc60249007)

[5.1 算法分析 12](#_Toc60249008)

[5.2 算法功能实现 12](#_Toc60249009)

[6. 堆排序 14](#_Toc60249010)

[6.1 算法分析 14](#_Toc60249011)

[6.2 算法功能分析 14](#_Toc60249012)

[6.2.1 堆调整 14](#_Toc60249013)

[6.2.2 堆排序 15](#_Toc60249014)

[7. 归并排序 17](#_Toc60249015)

[7.1 算法分析 17](#_Toc60249016)

[7.2 算法功能实现 17](#_Toc60249017)

[8. 基数排序 19](#_Toc60249018)

[8.1 算法分析 19](#_Toc60249019)

[8.2 算法功能实现 20](#_Toc60249020)

[9. 主函数 22](#_Toc60249021)

1. 冒泡排序
   1. 算法分析

冒泡排序是算法性能较差的一种排序方式，对于一个数组，一共要进行N-1次循环，每一次循环中又需要每个相邻的元素进行一次比较，较大的交换到下标大的数组位置中，这样能够使得每一次循环中都将剩下元素较大者交换到数组剩下的最后位置。N-1次循环后便能达到排序大的目的。

冒泡排序总的平均时间复杂度为O(n^2)。

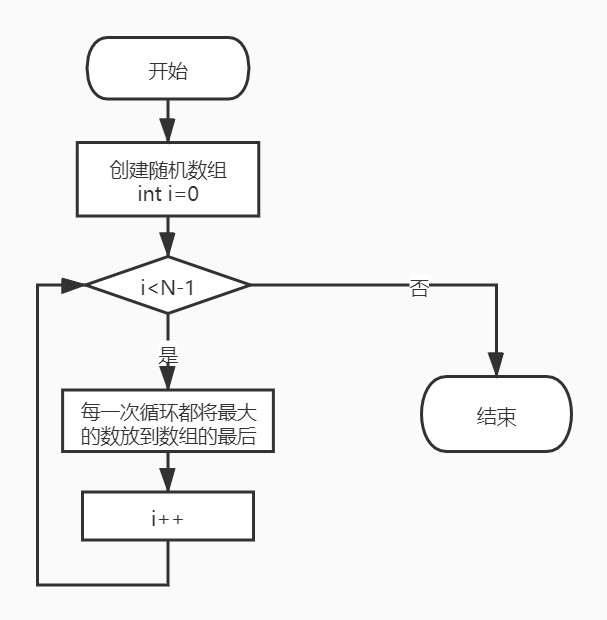
对于相同元素，排序算法只是比较相邻两个元素，所以相同元素的前后位置并不会改变，所以是一种稳定的排序。

优点:比较简单,空间复杂度较低,是稳定的

缺点:时间复杂度太高,效率不好

* 1. 算法功能实现

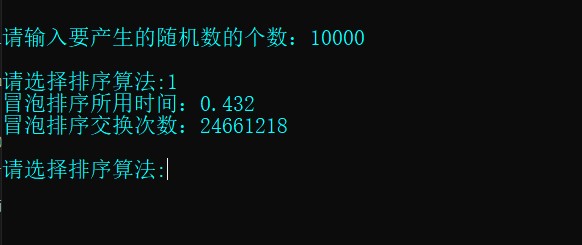
流程图：



代码实现：

1. //冒泡排序
2. **void** bubble\_sort(**int** N)
3. {
4. **clock\_t** start, end;//定义时间量用于计算排序程序前后所用时间
5. **int**\* p = **new** **int**[N];//开辟动态数组
6. srand(time(NULL));//利用时间设置产生随机数防止每一次程序运行产生的随机数相同
7. **if** (p==NULL)//如果内存不够开辟失败的情况
8. {
9. cout << "内存不足";
10. exit(0);
11. }
12. **for** (**int** i=0;i<N;i++)//产生随机数
13. {
14. \*(p + i) = rand();
15. }
16. start = clock();
17. **int** temp=0;
18. **long long** count=0;
19. **for** (**int** i=0;i<N-1;i++)//冒泡排序循环，一共进行N-1次，每一次都将为排序序列中最大的数通过交换放到数组的最后，这样经过N-1次后排序成功
20. {
21. **for** (**int** j=0;j<N-1-i;j++)
22. {
23. **if** (\*(p+j)>\*(p+j+1))
24. {
25. temp = \*(p + j);
26. \*(p + j) = \*(p + j + 1);
27. \*(p + j + 1) = temp;
28. count++;//每次交换计数器增加一次
29. }
30. }
31. }
32. end = clock();
33. **delete**[]p;//释放数组空间
34. cout << "冒泡排序所用时间：";
35. cout << (**double**)(end - start)/1000<<endl;
36. cout << "冒泡排序交换次数：";
37. cout << count<<endl<<endl;
38. }

结果：



1. 选择排序
   1. 算法分析

选择排序与冒泡排序没有太大的区别，冒泡排序是通过不断的交换来将每一次未排序序列中的最大数交换到未排序数组的最后位置，而选择排序是每一次循环找出未排序序列的最大数的下标直接与未排序下标最大的位置进行交换。

选择排序总的平均时间复杂度为O(n^2)。

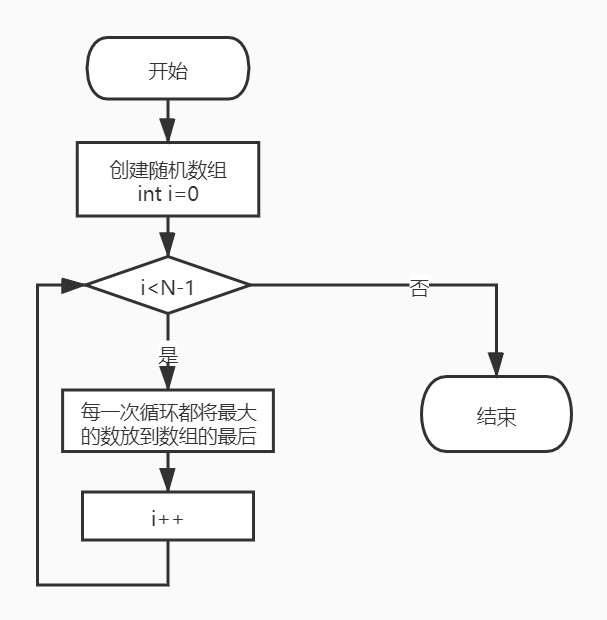
在一趟选择，如果一个元素比当前元素小，而该小的元素又出现在一个和当前元素相等的元素后面，那么交换后稳定性就被破坏了，所以说选择排序是一种不稳定的排序，会打乱相同元素的前后顺序。

优点：移动数据的次数已知（n-1次）

缺点：比较次数多且性能较差

* 1. 算法功能实现

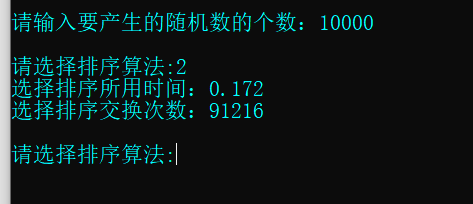
流程图：



代码实现：

1. //选择排序
2. **void** selection\_sort(**int** N) {
3. **clock\_t** start, end;
4. **int**\* p = **new** **int**[N];//建立动态数组
5. srand(time(NULL));
6. **if** (p == NULL)
7. {
8. cout << "内存不足";
9. exit(0);
10. }
11. **for** (**int** i=0;i<N;i++)
12. {
13. \*(p + i) = rand();
14. }
15. start = clock();
16. **int** temp;
17. **long long** count=0;
18. **for** (**int** i=0;i<N-1;i++)//一共经过N-1次循环，与冒泡排序相似，不同的是冒泡是通过交换来实现最大的数放到数组最后，选择排序是每一次找出最大的数
19. {
20. temp = 0;
21. **for** (**int** j=0;j<N-i-1;j++)//通过二级循环找出未排序序列中最大的数
22. {
23. **if** (\*(p+temp)<\*(p+j+1))
24. {
25. temp = j + 1;
26. count++;
27. }
28. }
29. **int** wrap = 0;//将其交换到数组的最后位置
30. wrap = \*(p+N-1-i);
31. \*(p + N-1 - i) = \*(p+temp);
32. \*(p + temp) = wrap;
33. count++;//每一次交换计数器增加一次
34. }
35. end = clock();
36. **delete**[]p;
37. cout << "选择排序所用时间：";
38. cout << (**double**)(end - start)/1000<<endl;
39. cout << "选择排序交换次数：";
40. cout << count<<endl<<endl;
41. }

结果：



1. 直接插入排序
   1. 算法分析

直接插入排序是一种较为简单的排序方式，对应一个随机数组，从第二个位置开始，如果前一个数据比它大，则前一个元素后移，要插入的元素再向前比较。

N-1次循环后实现排序功能。

直接插入排序的平均时间复杂度为0(n^2)，空间复杂度为0(1)。

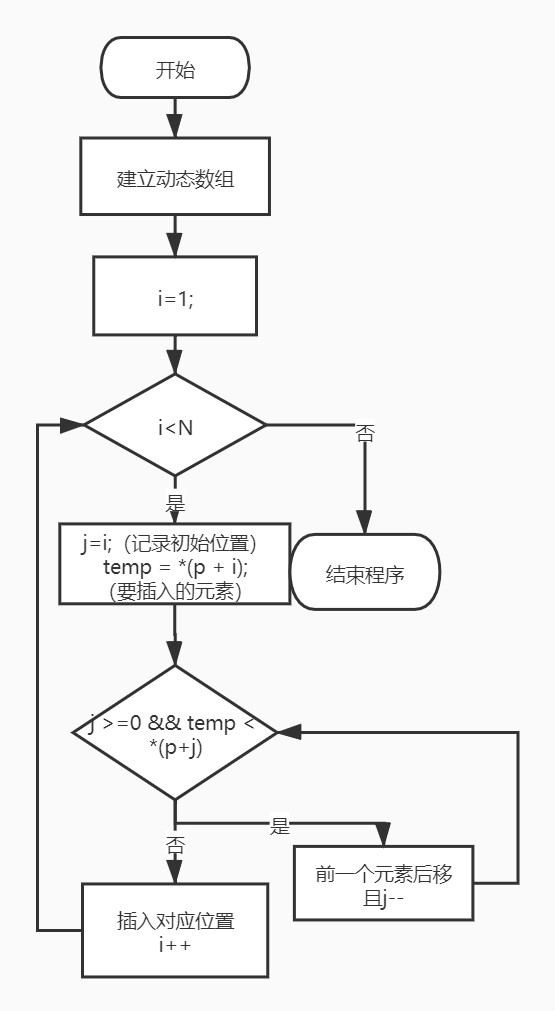
直接插入排序在向前比较的过程中不会越过与其相等的元素，所以说直接插入排序是一种稳定排序，不会改变相同元素的前后位置。

优点：稳定，快

缺点：比较次数不一定，比较次数越少，插入点后的数据移动越多，特别是当数据总量庞大的时候

* 1. 算法功能实现

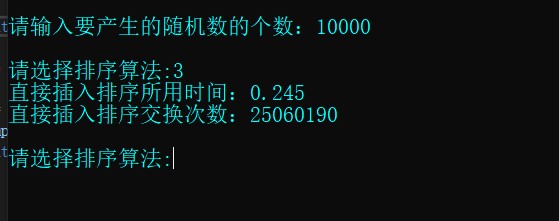
流程图：



代码实现：

1. //直接插入排序
2. **void** stright\_insertion(**int** N){
3. **clock\_t** start, end;
4. **int**\* p = **new** **int**[N];
5. srand(time(NULL));
6. **if** (p == NULL)
7. {
8. cout << "内存不足";
9. exit(0);
10. }
11. **for** (**int** i = 0; i < N; i++)
12. {
13. \*(p + i) = rand();
14. }
15. start = clock();
16. **int** temp;
17. **long long** count = 0;
18. **for** (**int** i=1;i<N;i++)//一共经过N-1次循环，从数组的第二个位置开始循环，将其插入到前面的符合升序要求的位置
19. {
20. **int** temp = \*(p+i);
21. **int** j = i;
22. **while** (temp<\*(p+j-1)&&j>0)//通过循环与前面已经排好的位置比较，直到插入到对应的位置
23. {
24. \*(p +j) = \*(p + j - 1);
25. count++;//每次交换计数器加1
26. j--;
27. }
28. \*(p + j) = temp;
29. count++;
30. }
31. end = clock();
32. **delete**[]p;
33. cout << "直接插入排序所用时间：";
34. cout << (**double**)(end-start)/1000<<endl;
35. cout << "直接插入排序交换次数：";
36. cout << count<<endl<<endl;
37. }

结果：



1. 希尔排序
   1. 算法分析

希尔排序是在直接插入排序基础上进行改进的一种算法，又称缩小增量排序。最开始按下标固定增量分组，由每一组内两个元素进行比较，之后每一次减少组数，增加每一组的关键词个数，在每一个组里进行直接插入排序，当组数等于1的时候最后实现排序功能。

希尔排序的时间复杂度为O(n^1.3-2)，空间复杂度为O(1)。

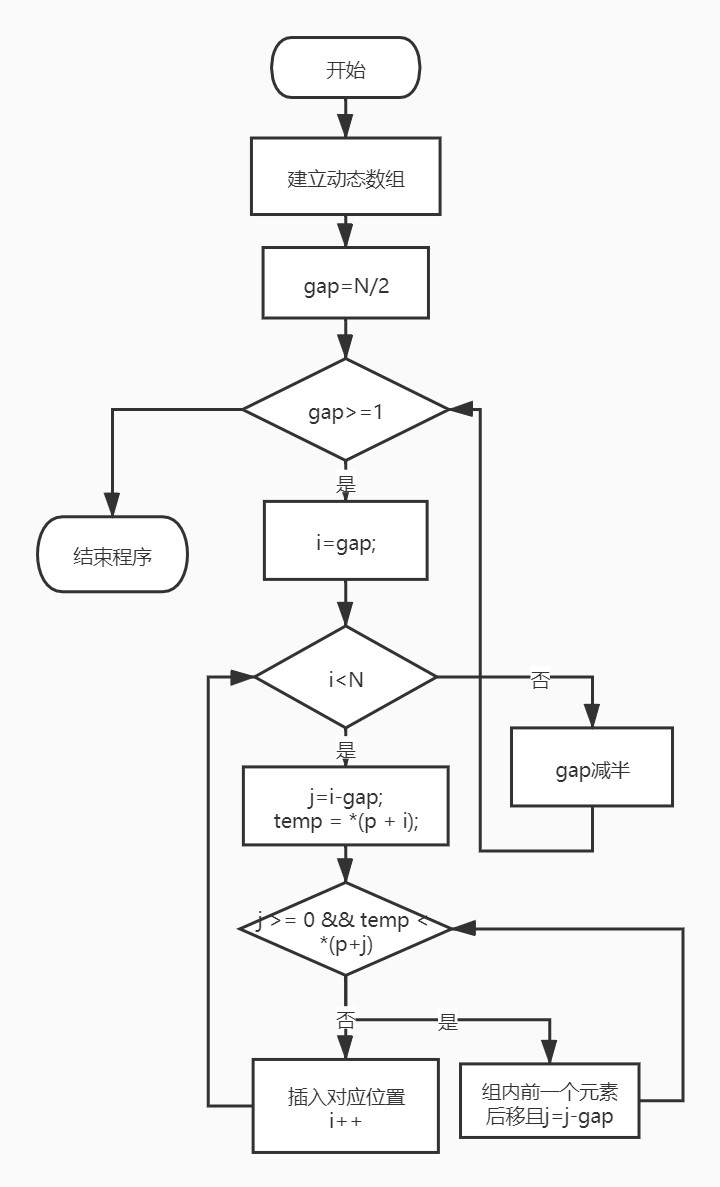
希尔排序的过程中会涉及到分组，所以相同元素的前后顺序会被打乱，是一种不稳定的排序算法。

优点：快，数据移动少

缺点：不稳定，gap的取值是多少，应取多少个不同的值，都无法确切知道，只能凭经验来取

* 1. 算法功能实现

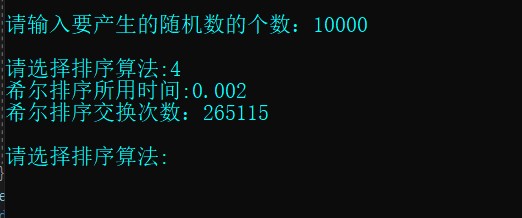
流程图：



代码实现：

1. //希尔排序
2. **void** shell\_sort(**int** N)
3. {
4. **clock\_t** start, end;
5. **int**\* p = **new** **int**[N];
6. srand(time(NULL));
7. **if** (p == NULL)
8. {
9. cout << "内存不足";
10. exit(0);
11. }
12. **for** (**int** i = 0; i < N; i++)
13. {
14. \*(p + i) = rand();
15. }
16. start = clock();
17. **int** gap=N/2,temp=0 ;
18. **long long** count = 0;//初始化定义跨度为N/2
19. **while** (gap>=1)//跳出循环的条件为gap变为1，即组数变为1
20. {
21. **for** (**int** i=gap;i<N;i++)//i=gap，从第一组的第二个元素开始
22. {
23. **int** j = i-gap;
24. temp = \*(p + i);
25. **for** (j = i - gap; j >= 0 && temp < \*(p+j); j = j - gap)//每一个元素都与之前组内的元素比较，思想与直接插入排序相同
26. {
27. \*(p+j + gap) =\*(p+j);
28. count++;
29. }
30. \*(p+j+gap) = temp;
31. count++;
32. }
33. gap = gap / 2;//每一次循环过后gap减半
34. }
35. end = clock();
36. **delete**[]p;
37. cout << "希尔排序所用时间:";
38. cout << (**double**)(end-start) / 1000<<endl;
39. cout << "希尔排序交换次数：";
40. cout << count<<endl<<endl;
41. }

结果：



1. 快速排序
   1. 算法分析

快速排序是对冒泡排序的一种优化算法，让最左边的值成为一个分界值，如果比这个值大就放到右边，比这个值小就放到左边。再通过递归实现排序。

快速排序的时间复杂度为O(nlogn)，空间复杂度最好为O(logn)，最差退化为冒泡为O(n)。

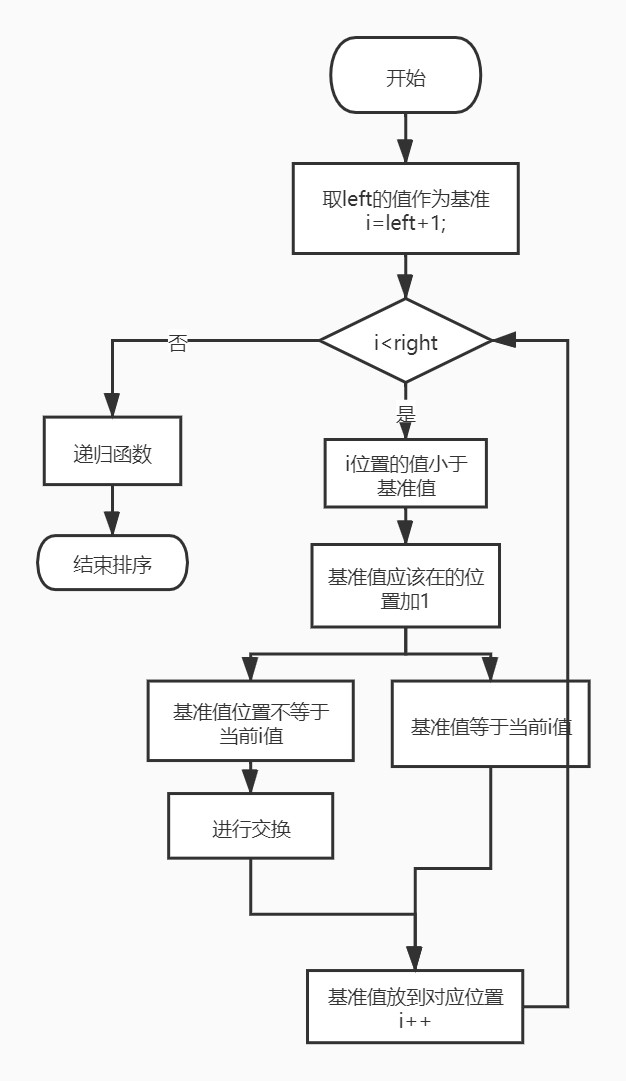
快速排序中在实现细节中，如果记录基准值位置的数pivotpos与循环的i不同时，说明需要将i位置上较小的数与基准值位置上的数交换，这时有可能会导致相同元素的前后位置被打乱。所以说快速排序是一种不稳定的排序算法。

优点：极快，数据移动少

缺点：不稳定

* 1. 算法功能实现

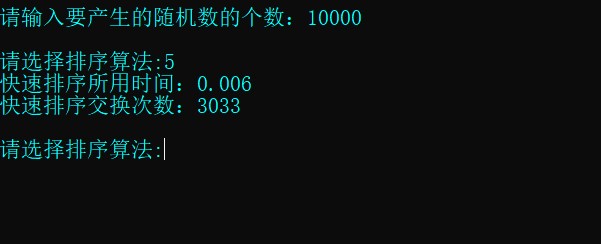
流程图：



代码实现：

1. **//快速排序**
2. **long long** quick\_sort(**int** left,**int** right,**int**\*p)
3. {
4. **int** count = 0;
5. **if** (left<right)
6. {
7. **int** pivotpos = left;
8. **int** pivot = \*(p+left);//以下标最小的位置值作为基准
9. **for** (**int** i=left+1;i<=right;i++)
10. {
11. **if** (\*(p+i)<pivot)//如果出现右侧值比基准小的情况
12. {
13. pivotpos++;//基准值应放到的位置向右移动一位
14. **if** (pivotpos!=i)//如果这个值不等于当前i，说明前面有比基准值大的情况
15. {
16. **int** swap;//交换i位置与pivotpos位置的值，让比基准值大的值移到右边
17. swap = \*(p+pivotpos);
18. \*(p + pivotpos) = \*(p + i);
19. \*(p + i) = swap;
20. count++;
21. }
22. }
23. }
24. \*(p + left) = \*(p + pivotpos);//将基准值放到对应位置上
25. \*(p + pivotpos) = pivot;
26. quick\_sort(left,pivotpos,p);//两次递归函数实现快速排序
27. quick\_sort(pivotpos+1,right,p);
28. }
29. **return** count;//返回计数器的值
30. }

结果：



1. 堆排序
   1. 算法分析

堆排序是通过堆这样的数据结构来实现排序的问题，堆是一种完全二叉树，在每一次对叶结点堆调整过后，总可以将最大值调整到堆顶，此时将最后一个叶节点值与其交换，再将这个最大值放到新的数组中，删掉这个叶节点，再进行堆调整，这样进行N-1后就可以实现排序的目的。

堆排序的时间复杂度为O(nlogn)，空间复杂度为O(1)。

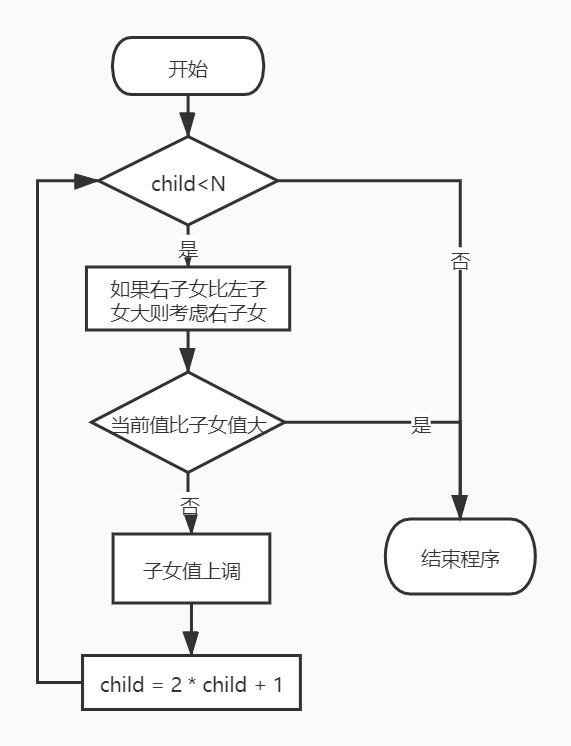
堆排序在调整堆的过程中并不能保证满足相同元素的前后顺序不变，所以堆排序是一种不稳定的算法。

优点：对数字序列的有序性不敏感，性能较好

缺点：实现过程要实现建堆及堆调整两个过程，不适合较小的序列，适合较大序列的排序性能提升

* 1. 算法功能分析
     1. 堆调整

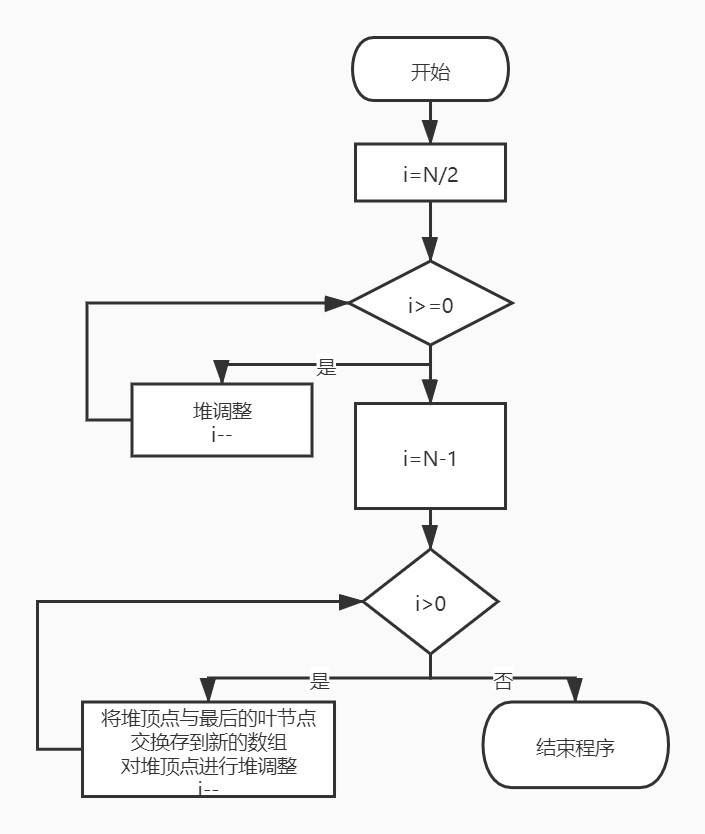
流程图：



代码实现：

1. //堆的调整
2. **long long** heap\_adjust(**int** \*p,**int** parent,**int** N)
3. {
4. //parent位置值的向下调整
5. **int** temp = \*(p+parent);
6. **int** child = parent \* 2 + 1;
7. **int** count = 0;
8. **while** (child<N)
9. {
10. **if** (child + 1 < N && \*(p+child) < \*(p+child + 1)) //如果右子女存在且比左子女大
11. {
12. child++;
13. }
14. **if** (temp >= \*(p+child)) //如果当前父母的值比子女的值大跳出循环
15. **break**;
16. \*(p + parent) = \*(p + child);
17. count++;
18. parent = child; //目前初始parent值应该存放的位置
19. child = 2 \* child + 1;
20. }
21. \*(p + parent) = temp; //存放需要调整位置的值
22. **return** count;
23. }
    * 1. 堆排序

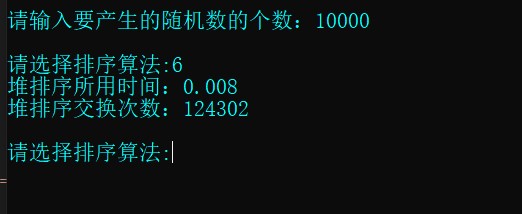
流程图：



代码实现：

1. //堆排序
2. **long long** heap\_sort(**int**\* p,**int** N,**int**\* q)
3. {
4. **int** count = 0;
5. **for** (**int** i = N / 2; i >= 0; i--)//对每一个非叶节点进行堆调整
6. {
7. count=count+heap\_adjust(p, i, N);
8. }
9. **for** (**int** i = N - 1; i > 0; i--) //进行N-1次循环，每一次结束时对堆顶点进行堆调整，将每一次的最大值存入新开辟的q数组中
10. {
11. **int** temp = \*(p+i);
12. \*(q + i) = \*p;
13. \*(p+i) = \*p;
14. \*p = temp;
15. count++;
16. count=count+heap\_adjust(p, 0, i);
17. }
18. \*q = \*p;//最小值存进q数组的第一个位置
19. **return** count;
20. }

结果：



1. 归并排序
   1. 算法分析

归并排序是建立在归并操作上的一种有效，稳定的排序算法，该算法采用分治法，首先通过不断递归进行不断分组归并，之后再将两组的数据按照大小依次进入暂存的数组，然后暂存数组的值返回给原数组，这样进行下去就可以实现排序的功能。

归并排序的时间复杂度为O(nlogn)，空间复杂度为O(n)。

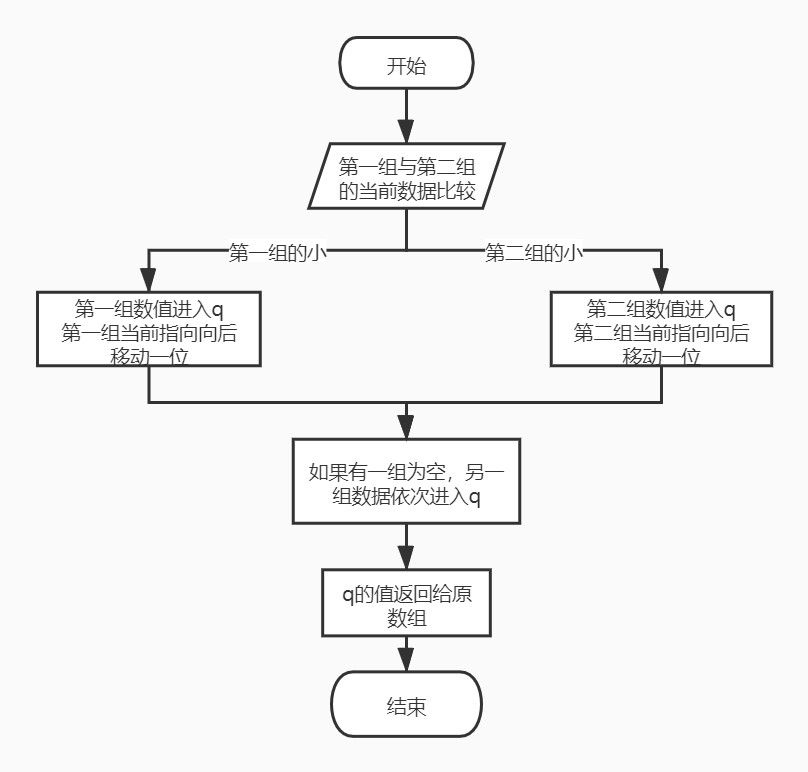
在归并排序的过程中并不会使得相同的元素前后位置被打乱，所以归并排序是一种稳定的排序算法。

优点：速度快

缺点：需要的辅助存储空间很大

* 1. 算法功能实现

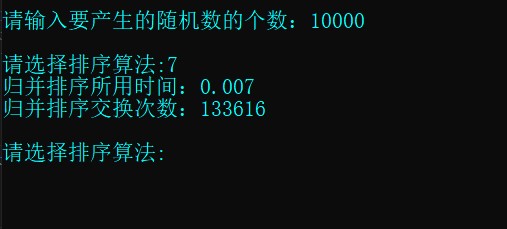
流程图：



代码实现：

1. //归并排序
2. **void** merge\_sort(**int**\* p,**int**\* q,**int** start,**int** end,**long long** &count)
3. {
4. **if** (start >= end)
5. **return** ;
6. **int** len = end - start, mid = len /2 + start;
7. **int** start1 = start, end1 = mid;
8. **int** start2 = mid + 1, end2 = end;
9. merge\_sort(p, q, start1, end1,count);//不断递归到达分组只剩下一个值
10. merge\_sort(p, q, start2, end2,count);
11. **int** k = start;
12. **while** (start1 <= end1 && start2 <= end2)//对于每个组，内部排序是已经完成的，需要依次比较两个组的前置元素放到新的数组q中
13. {
14. **if** (\*(p + start1) < \*(p + start2))
15. {
16. \*(q + k) = \*(p + start1);
17. start1++;
18. }
19. **else** {
20. \*(q + k) = \*(p + start2);
21. start2++;
22. }
23. count++;
24. k++;
25. }
26. **while** (start1 <= end1)
27. {
28. \*(q + k) = \*(p + start1);
29. k++;
30. start1++;
31. count++;
32. }
33. **while** (start2 <= end2)
34. {
35. \*(q + k) = \*(p + start2);
36. k++;
37. start2++;
38. count++;
39. }
40. **for** (k = start; k <= end; k++)//把暂存的q里的值返回给p
41. \*(p+k) = \*(q+k);
42. }

结果：



1. 基数排序
   1. 算法分析

基数排序是一种比较型整数排序算法，将整数按位数切割成不同的数字，然后按每个位数进行排序，称每一次依据的位数称为判断位数，每一次排序都分别记录判断位数相同的数据的个数，然后每一组数据一次加上前面的数据个数，得到的是每一种情况最后一个数的下标位置，此时将每一种情况依次放到原数组中，通过每一位都循环过后即能实现排序。该程序使用的是LSD排序算法。

基数排序的时间复杂度为O(nlog(r)m)，r为基数，m为维数。空间复杂度为O(n)。

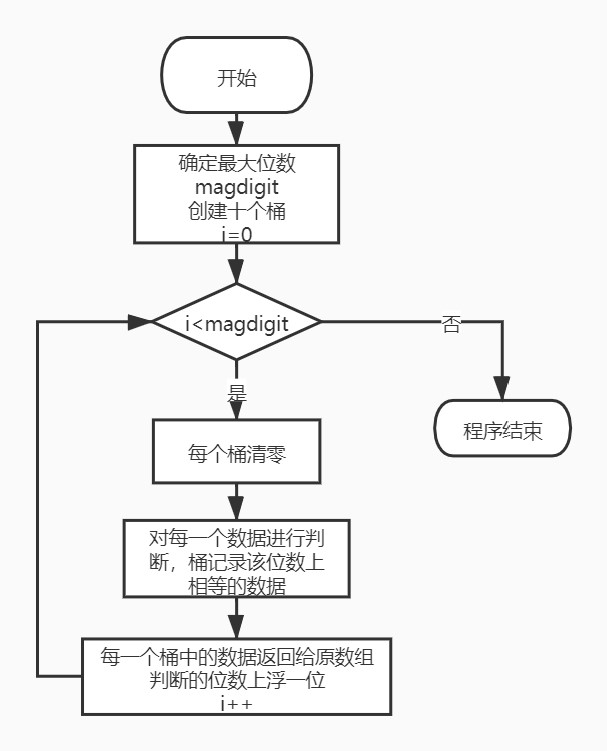
基数排序的过程中若出现相同的元素，它们在排序过程中也是按序操作的，所以基数排序不会打乱原有的顺序，是一种稳定的排序算法。

优点：快

缺点：数据范围必须为正整数并且比较小

* 1. 算法功能实现

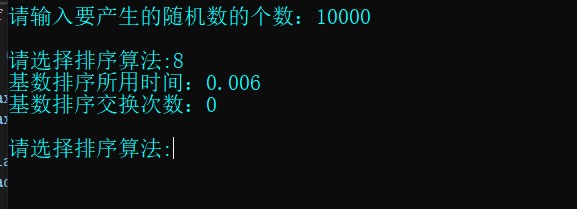
流程图：



代码实现：

1. //基数排序
2. **void** radix\_sort(**int**\* p,**int** N)
3. {
4. **int** maxdigit=0,maxnum=\*p;
5. **int**\* q = **new** **int**[N];
6. **for** (**int** i=0;i<N;i++)
7. {
8. **if** (\*(p + i) > maxnum) maxnum = \*(p + i);
9. }
10. **while** (maxnum%10!=0)//确定最大值的位数，来确定需要进行几次循环
11. {
12. maxnum = maxnum/10;
13. maxdigit++;
14. }
15. **int** place[10];//创建0到9十个桶
16. **int** radix = 1;//通过除以radix余数确定按第几位分类
17. **int** i, j, k;
18. **for** (i=0;i<maxdigit;i++)
19. {
20. **for** (j=0;j<10;j++)//首先先把每个桶的值清零
21. {
22. place[j] = 0;
23. }
24. **for** (j=0;j<N;j++)//对每一个值分类，分到该位数相同的桶中
25. {
26. **int** m=0;
27. k=(p[j] / radix) % 10;
28. place[k]++;
29. }
30. **for** (j=1;j<10;j++)//通过循环，让每一个place的值记录该桶中最后的元素对应的下标位置
31. {
32. place[j] = place[j - 1] + place[j];
33. }
34. **for** (j = N - 1; j >= 0; j--)//将每一个桶中的值返回给暂存数组
35. {
36. k = (p[j] / radix) % 10;
37. q[place[k] - 1] = p[j];
38. place[k]--;
39. }
40. **for** (j=0;j<N;j++)//暂存数组的值返回给原数组
41. {
42. p[j] = q[j];
43. }
44. radix = radix \* 10;//每次循环乘以10
45. }
46. **delete**[]q;
47. }

结果：



1. 主函数

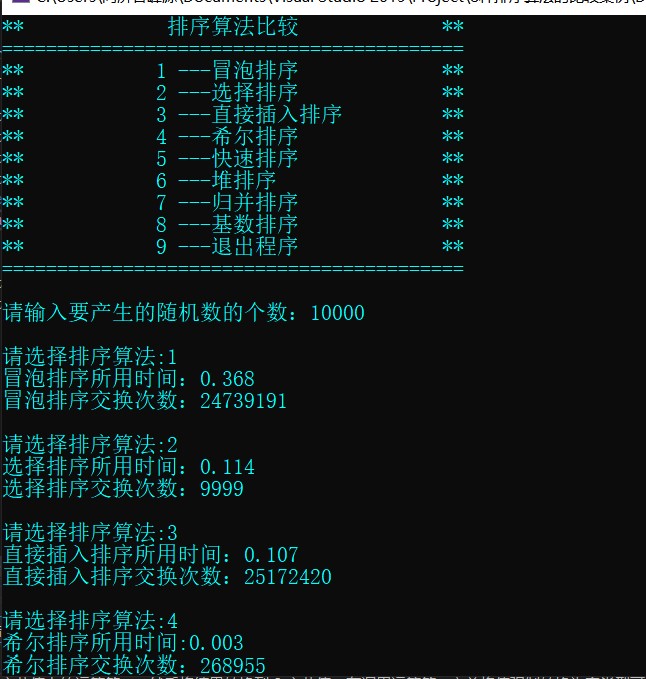
说明：主函数中首先按题目要求打印相应的文档，然后运用switch语句进行操作的选择，1代表选用冒泡排序，2代表选用选择排序，3代表选用直接插入排序，4代表选用希尔排序，5代表选用快速排序，6代表选用堆排序，7代表选用归并排序，8代表选用基数排序，9代表退出程序。在排序过程中，我们希望知道该排序所使用的的时间和比较次数，这就需要在程序运行前后加入时间节点，以及在程序运行中加入计数器记录交换次数。

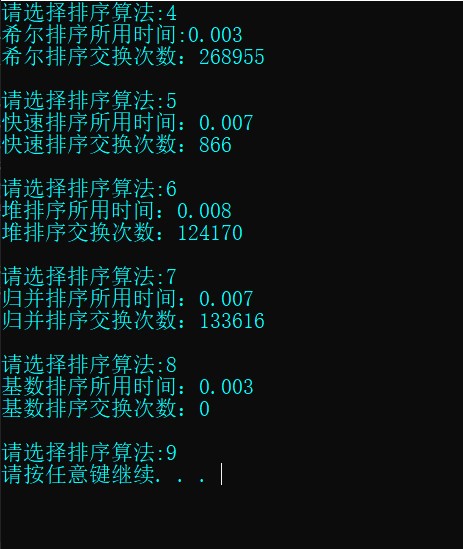
下面是代码以及截图显示的结果，结果是以10000个随机数为例进行排序的结果。

代码实现：

1. **int** main()
2. {
3. cout << "\*\*             排序算法比较             \*\*"<<endl;
4. cout << "=========================================="<<endl;
5. cout << "\*\*            1 ---冒泡排序             \*\*"<<endl;
6. cout << "\*\*            2 ---选择排序             \*\*"<<endl;
7. cout << "\*\*            3 ---直接插入排序         \*\*"<<endl;
8. cout << "\*\*            4 ---希尔排序             \*\*"<<endl;
9. cout << "\*\*            5 ---快速排序             \*\*"<<endl;
10. cout << "\*\*            6 ---堆排序               \*\*"<<endl;
11. cout << "\*\*            7 ---归并排序             \*\*"<<endl;
12. cout << "\*\*            8 ---基数排序             \*\*"<<endl;
13. cout << "\*\*            9 ---退出程序             \*\*"<<endl;
14. cout << "=========================================="<<endl<<endl;
15. cout << "请输入要产生的随机数的个数：";
16. **int** N=0;
17. cin >> N;
18. cout << endl;
19. **int** num;
20. **while** (1)
21. {
22. cout << "请选择排序算法:";
23. cin >> num;
24. **switch** (num) {
25. **case** 1:
26. bubble\_sort(N);
27. **break**;
28. **case** 2:
29. selection\_sort(N);
30. **break**;
31. **case** 3:
32. stright\_insertion(N);
33. **break**;
34. **case** 4:
35. shell\_sort(N);
36. **break**;
37. **case** 5:
38. {
39. **int**\* p = **new** **int**[N];
40. srand(time(NULL));
41. **clock\_t** start, end;
42. **if** (p == NULL)
43. {
44. cout << "内存不足";
45. exit(0);
46. }
47. **for** (**int** i = 0; i < N; i++)
48. {
49. \*(p + i) = rand();
50. }
51. start = clock();
52. **int** left = 0, right = N - 1;
53. **long long** count = quick\_sort(left, right, p);
54. end = clock();
55. cout << "快速排序所用时间：";
56. cout << (**double**)(end - start) / 1000 << endl;
57. cout << "快速排序交换次数：";
58. cout << count << endl << endl;
59. **break**;
60. }
61. **case** 6:
62. {
63. **int**\* p = **new** **int**[N];
64. srand(time(NULL));
65. **clock\_t** start, end;
66. **if** (p == NULL)
67. {
68. cout << "内存不足";
69. exit(0);
70. }
71. **for** (**int** i = 0; i < N; i++)
72. {
73. \*(p + i) = rand();
74. }
75. start = clock();
76. **int**\* q = **new** **int**[N];
77. **long long** count = heap\_sort(p, N, q);
78. end = clock();
79. **delete**[]p;
80. **delete**[]q;
81. cout << "堆排序所用时间：";
82. cout << (**double**)(end - start) / 1000 << endl;
83. cout << "堆排序交换次数：";
84. cout << count << endl << endl;
85. **break**;
86. }
87. **case** 7:
88. {
89. **int**\* p = **new** **int**[N];
90. srand(time(NULL));
91. **clock\_t** start, end;
92. **if** (p == NULL)
93. {
94. cout << "内存不足";
95. exit(0);
96. }
97. **for** (**int** i = 0; i < N; i++)
98. {
99. \*(p + i) = rand();
100. }
101. start = clock();
102. **int**\* q = **new** **int**[N];
103. **long long** count = 0;
104. merge\_sort(p,q,0,N-1,count);
105. end = clock();
106. **delete**[]p;
107. **delete**[]q;
108. cout << "归并排序所用时间：";
109. cout << (**double**)(end - start) / 1000 << endl;
110. cout << "归并排序交换次数：";
111. cout << count << endl << endl;
112. **break**;
113. }
114. **case** 8:
115. {
116. **int**\* p = **new** **int**[N];
117. srand(time(NULL));
118. **clock\_t** start, end;
119. **long long** count = 0;
120. **if** (p == NULL)
121. {
122. cout << "内存不足";
123. exit(0);
124. }
125. **for** (**int** i = 0; i < N; i++)
126. {
127. \*(p + i) = rand();
128. }
129. start = clock();
130. radix\_sort(p,N);
131. end = clock();
132. cout << "基数排序所用时间：";
133. cout << (**double**)(end - start) / 1000 << endl;
134. cout << "基数排序交换次数：";
135. cout << count << endl << endl;
136. **delete**[]p;
137. **break**;
138. }
139. **case** 9:
140. system("pause");
141. exit(0);
142. }
143. }
144. **return** 0;
145. }

结果：





每一种排序算法的时间复杂度、空间复杂度都已在各自目录下给出，优缺点也都进行了一定程度上的分析。