[常见的排序算法总结 2](#_Toc456646480)

[1. 插入排序 3](#_Toc456646481)

[2. 希尔排序 shell排序 5](#_Toc456646482)

[3. 归并排序 7](#_Toc456646483)

[4. 简单选择排序 9](#_Toc456646484)

[5. 冒泡排序 bubbleSort 10](#_Toc456646485)

[6. 堆排序 12](#_Toc456646486)

[7. 快速排序 15](#_Toc456646487)

[8. 计数排序 16](#_Toc456646488)

[9. 基数排序 18](#_Toc456646489)

[10. 桶排序 19](#_Toc456646490)

[11. 两种二分查找 19](#_Toc456646491)

常见的排序算法总结

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 编号 | 算法 | 时间复杂度 | | | 空间复杂度 | 稳定性 | 就地 |
| 平均情况 | 最好情况 | 最差情况 | 辅助内存 |
| 1 | 插入排序 | O(n^2) | O(n) | O(n^2) | O(1) | 稳定 | 是 |
| 2 | 希尔排序 | O(n^1.5) | O(n^1.5) | O(n^1.5) | O(1) | 不稳定 | 是 |
| 3 | 冒泡排序 | O(n^2) | O(n) | O(n^2) | O(1) | 稳定 | 是 |
| 4 | 简单选择 | O(n^2) | O(n^2) | O(n^2) | O(1) | 稳定 | 是 |
| 5 | 归并排序 | O(nlgn) | O(nlgn) | O(nlgn) | O(n+lgn) | 稳定 | 否 |
| 6 | 堆排序 | O(nlgn) | O(nlgn) | O(nlgn) | O(1) | 不稳定 | 是 |
| 7 | 快速排序 | O(nlgn) | O(nlgn) | O(n^2) | O(nlgn)~O(n) | 不稳定 | 是 |
| 8 | 计数排序 | O(n+k) | O(n+k) | O(n+k) | O(n+k) | 稳定 | 否 |
| 9 | 基数排序 | O(d(n+k)) | O(d(n+k)) | O(d(n+k)) | O(d(n+k)) | 稳定 | 否 |
| 10 | 桶排序 | O(n) | O(n) | O(n^2) |  | 稳定 | 否 |

1. 插入排序

定义：插入排序是将一个记录插入到已排好序的有序表中,从而得到一个新的，记录数增1的有序表。

插入排序的时间复杂度

最好情况：要排序的数组与要求结果顺序一致O(n)，每次不需要移动元素。

最坏情况: 要排序的数组与要求结果顺序相反O(n^2),已排好序的元素都需要向后移动。

平均情况：O(n^2).

空间复杂度：就地排序。

是否稳定：稳定。

算法简单，当n比较小时，效率较高。

比如对一组数据{49,38,65,97,76,13,27}进行排序，则是首先初始化第0个49数字有序，然后从第1个开始，依次把后面的数据插入到前面有序的结果中。

针对第i个数字找其相应的插入位置，则是在前0…i-1个有序的数字中找第i个数字的位置。那么就从第i-1个开始向前比较，如果当前元素比要插入的元素大，则向后移动当前元素，一直进行比较直到找到不比要插入的元素大的元素，把插入的元素放到该元素的后面，即找到了要插入元素的正确位置。

排序过程如下:

|  |  |
| --- | --- |
| i = 0 | 49 38 65 97 76 13 27 |
| i = 1 | 排序前：49 38 65 97 76 13 27 排序后： 38 49 65 97 76 13 27 |
| i = 2 | 排序前：38 49 65 97 76 13 27 排序后： 38 49 65 97 76 13 27 |
| i = 3 | 排序前：38 49 65 97 76 13 27 排序后： 38 49 65 97 76 13 27 |
| i = 4 | 排序前：38 49 65 97 76 13 27 排序后： 38 49 65 76 97 13 27 |
| i = 5 | 排序前：38 49 65 76 97 13 27 排序后： 13 38 49 65 76 97 27 |
| i = 6 | 排序前：13 38 49 65 76 97 27 排序后： 13 27 38 49 65 76 97 |

Java实现代码：

|  |
| --- |
| /\*\***@author**  \* 插入排序 运行时间是 n的平方 = n\*n  \* 稳定的算法  \* 就地排序  \* 排序最快的时候是原来数组已经有序 运行时间就是n  \* 排序最差的时候就是原来的数组是倒序 运行时间就是n\*n  \* \* **@param** A 待排序的数组  \*/  **public** **void** INSERT\_SORT(**int** A[]){  **long** a=System.*currentTimeMillis*();  **int** n=A.length;  **int** i;  **int** key;  **for**(**int** j=1;j<n;j++){  i=j-1;  key = A[j];//在此之前A[0],A[1]...A[j-1]已经有序  **while**(i>=0 && A[i]>key){ //从A[j-1]开始找，找到 所有的比key大的元素依次向后移动  A[i+1]=A[i];//如果A[i]比关键字大则向后移动一个  i=i-1; //继续判断前一个元素与key的大小关系  }  A[i+1]=key;//找到了关键字的插入位置  }  System.*out*.println("执行耗时 : "+(System.*currentTimeMillis*()-a)/1000f+" 秒 ");  } |

1. 希尔排序 shell排序

时间复杂度 O(n^1.5)

空间复杂度：就地排序。

是否稳定：不稳定。

将整个待排序的记录序列分割成若干子序列分别进行直接插入排序，待整个序列中的记录基本有序时，再对整体记录进行一次直接插入排序。

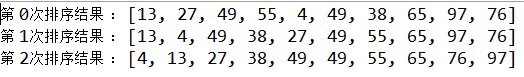
注意的是子序列的构成不是简单地“逐段分割”，而是将相隔某个“增量”的记录组成一个子序列。增量序列可以有各种取法，但是需要注意的是应使递增序列中的值没有除1之外的公因子，并且最后一个增量值必须等于1.

Step1:依次遍历递增序列中的每个值dk；

Step2:根据选到的dk，把数组分成dk个子序列，每个子序列中的相邻元素的下标相差dk；

Step3:针对第二步中的每个子序列进行直接插入排序。

|  |
| --- |
| /\*\*  \* 希尔排序。  \* 时间复杂度 O(n^1.5).  \* 不是稳定的算法。  \* 将整个待排序的记录序列分割成若干子序列分别进行直接插入排序，待整个序列中的记录基本有序时，再对整体记录进行一次直接插入排序.  \* Step1:依次遍历递增序列中的每个值dk；  \* Step2:根据选到的dk，把数组分成dk个子序列，每个子序列中的相邻元素的下标相差dk；  \* Step3:针对第二步中的每个子序列进行直接插入排序。  \* **@param** A 待排序的数组。  \* **@param** dlta 排序时依据的递增序列  \*/  **public** **void** shellSort(**int**[] A,**int**[] dlta){  **int** len = dlta.length;  /\*Step1:依次遍历递增序列中的每个值dk；\*/  **for**(**int** i=0;i<len;i++){  shellInsert(A,dlta[i]);  }  }  /\*\*  \* Step2:根据选到的dk，把数组分成dk个子序列，每个子序列中的相邻元素的下标相差dk；  \* Step3:针对第二步中的每个子序列进行直接插入排序。  \* **@param** A  \* **@param** dk  \*/  **public** **void** shellInsert(**int**[] A,**int** dk){  **int** len = A.length;  **for**(**int** i=0;i<dk;i++){/\*根据选到的dk，把数组分成dk个子序列\*/  /\*对每个子序列进行直接插入排序\*/  **for**(**int** k=i+dk;k<len;k=k+dk){  **int** j = k-dk;  **int** key = A[k];  **while**(j>=0 && A[j]>key){  A[j+dk] = A[j];  j = j-dk;  }  A[j+dk] = key;  }  }  System.*out*.println(Arrays.*toString*(A));  }    **public** **static** **void** main(String[] args){  **int**[] A ={49,38,65,97,76,13,27,49,55,4};  **int**[] dlta = {5,3,1};  ShellSort test = **new** ShellSort();  test.shellSort(A, dlta);  } |



1. 归并排序

归并排序是假设初始序列是有n个记录，则可以看成是n个长度为1有序的子序列，然后两两合并，得到n/2个长度为2或者1的子序列；再两两合并，…，如此重复，直到得到一个长度为n的序列。

核心是将一组数组中前后相邻的两个有序序列合并为一个有序序列。

时间复杂度O(nlgn)，空间复杂度O(n)，是稳定的排序算法。

|  |
| --- |
| /\*\*  \* 把两个有序的数组进行合并，数组A[p...q]和A[q+1...r]分别有序，现在将其合并成数组A[p...r]有序  \* 运行时间是nlgn  \* 稳定的排序  \* 不是就地排序 空间复杂度O(n)  \* **@param** A  \* **@param** p  \* **@param** q  \* **@param** r  \*/  **public** **void** MERGE(**int** A[],**int** p,**int** q,**int** r){    **int** m=q-p+1;  **int** n=r-q;  **int** L[]=**new** **int**[m+1];  **int** R[]=**new** **int**[n+1];    **for**(**int** i=0;i<m;i++){  L[i]=A[p+i];  }  L[m]=999999;//设置最大的结尾元素。    **for**(**int** j=0;j<n;j++){  R[j]=A[q+j+1];  }  R[n]=999999;    **int** i=0;  **int** j=0;  **for**(**int** k=p;k<=r;k++){  **if**(L[i]<=R[j]){//保证了是稳定的排序  A[k]=L[i];  i++;  }**else**{  A[k]=R[j];  j++;  }  }  }  /\*\*  \* 递归的方式对数组进行划分。  \*/  **public** **void** Merge\_sort(**int** A[],**int** p,**int** r){    **if**(p<r){  **int** q = (p+r)/2;  Merge\_sort(A,p,q);  Merge\_sort(A,q+1,r);  MERGE(A,p,q,r);  }**else**{  **return**;  }  } |

1. 简单选择排序

简单选择排序是每次循环找出剩下的元素中最小的那个。

时间复杂度为O(n^2)：相对于冒泡排序算法来说可能交换的次数少了，但是比较的次数没有少。

就地排序、稳定的算法。

简单选择排序的基本原理如下：

对于给定的一组记录，经过第一轮比较后得到最小的记录，然后将该记录与第一个记录的位置进行交换；接着对不包含第一个记录之外的其他记录进行第二轮交换，得到最小的记录并与第二个记录进行位置交换，重复该过程，直到进行比较的记录只有一个时为止。

每次只需要交换那次循环找到的最小元素即可。

特点是：交换移动数据次数少。

|  |
| --- |
| /\*\*  \* 简单选择排序算法.  \* 时间复杂度都是O(n^2).  \* 稳定。  \* 就地。  \* 每次都是选择出当前剩下的元素中最小的。 \*  \* **@param** A 待排序的数组  \*/  **public** **void** selectSort(**int**[] A){  **int** len = A.length;  **int** min = 0;  **int** temp;  **for**(**int** i=0;i<len;i++){// i<len-1也可以，最后只剩下一个元素时则没有必要比较  min = i;//下一次选中的最小元素的下标  **for**(**int** j=i+1;j<len;j++){  **if**(A[min]>A[j]){  min = j;  }  }  **if**(i!=min){  temp = A[i];  A[i] = A[min];  A[min] = temp;  }  }  System.*out*.println("排序后的结果 ： "+Arrays.*toString*(A));  } |

1. 冒泡排序 bubbleSort

冒泡排序是经过比较数组中相邻元素，如果反序则交换，直到没有反序的记录为止。

每次冒泡是把当前剩下的元素经过两两比较选出最小的。

在每次循环开始前，0...i-1的数组元素已经有序，在剩下的i...len-1经过两两交换选出最小的元素。设置标志位flag，表示某次冒泡排序时是否有经过相邻元素的交换，若没有则表示数组已经有序，不需要再进行循环。

时间复杂度：最好情况:给定数组与目标数组一样，复杂度为O(n)；平均情况:O(n)

就地排序，稳定的排序算法。

|  |
| --- |
| /\*\*  \* 最好情况:给定数组与目标数组一样。O(n)  \* 平均情况:O(n)  \* 就地排序.  \* 稳定。  \* 在每次循环开始前，0...i-1的数组元素已经有序，在剩下的i...len-1经过两两交换选出最小的元素。  \* 设置标志位flag，表示某次冒泡排序时是否有经过相邻元素的交换，若没有则表示数组已经有序，不需要再进行循环。  \*  \* **@param** A 待排序的数组。  \*/  **public** **void** bubbleSort(**int**[] A){  **boolean** flag = **true**;  **int** len = A.length;  **int** temp = 0;  **for**(**int** i=0;i<len && flag == **true**;i++){  flag = **false**;  **for**(**int** j = len-1;j>i;j--){  flag = **true**;  **if**(A[j]<A[j-1]){//如果相邻两个元素逆序，则交换  temp = A[j-1];  A[j-1] = A[j];  A[j] = temp;  }  }  }  System.*out*.println("排序后的结果 ： "+Arrays.*toString*(A));  } |

1. 堆排序

堆排序是将待排序的序列构成一个大顶堆，数组中的最大元素总是在根结点中A[1]中，通过它与A[n]进行互换，则让该元素放到正确的位置，然后对堆进行调整（此时有效元素个数为n-1），找出剩下的最大的元素放到A[1]中，重复该过程，直到堆中的元素个数变为1.此时就完成了对所有元素的排序。

建堆的时间复杂度是O(n)，调整堆的时间复杂度是O(lgn)，堆排序的时间复杂度是O(nlgn).

就地排序，非稳定的排序算法。

|  |
| --- |
| /\*\*  \* 20150701 这个类主要介绍对排序的相关知识  \* 堆排序的运行时间是nlgn  \* 就地排序  \* 但不是稳定的排序算法  \*/  **public** **class** Heap {  **int** heap\_size = 0;// 表示堆中有多少个元素  /\*\*  \* 维护最大堆的性质 时间复杂度是lgn  \* **@param** A  \* **@param** i  \*/  **public** **void** max\_Heapify(**int** A[], **int** i) {  **int** l = i \* 2 + 1;  **int** r = (i + 1) \* 2;  **int** largest = i;  **if** (l < heap\_size && A[l] > A[i]) {  largest = l;  }  **if** (r < heap\_size && A[r] > A[largest]) {  largest = r;  }  **if** (i != largest) {  **int** temp;  temp = A[largest];  A[largest] = A[i];  A[i] = temp;  max\_Heapify(A, largest);  }  }  /\*\*  \* 给一个数组把它建成大顶堆 运行时间复杂度是 O(n)  \* **@param** A  \*/  **public** **void** build\_Max\_Heap(**int** A[]) {  heap\_size = A.length;  **for** (**int** i = (heap\_size - 1) / 2; i >= 0; i--) {  max\_Heapify(A, i);  }  }  /\*\*  \* 堆排序的运行时间是nlgn  \* 就地排序  \* 但不是稳定的排序算法  \* **@param** A  \*/  **public** **void** heap\_Sort(**int** A[]) {  build\_Max\_Heap(A);// 首先建立大顶堆  **int** temp;  **for** (**int** i = heap\_size - 1; i >= 1; i--) {  temp = A[0];// A[0]和堆中最后的那个元素交换，这样最大的元素就放在了数组的最后  A[0] = A[i];  A[i] = temp;  heap\_size--;  max\_Heapify(A, 0);  }  }  **public** **static** **void** main(String[] args) {  **int** A[] = { 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 };  **for** (**int** i = 0; i < A.length; i++) {  System.*out*.print(A[i] + " ");  }  System.*out*.println();  Heap h = **new** Heap();  h.heap\_Sort(A);  **for** (**int** i = 0; i < A.length; i++) {  System.*out*.print(A[i] + " ");  }  }  } |

1. 快速排序

快速排序是通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字比另一部分的关键字都小，然后可以分别对着两部分的记录继续进行排序，以达到整个序列有序的目的。

时间复杂度最坏情况是：当输入的数组完全有序时（不管正序还是反序），O(n^2);

最好情况：O(nlgn).划分均衡

是就地排序，但是由于递归进行划分，所以会使用O(lgn)~O(n)的辅助空间。

|  |
| --- |
| /\*\*  \* 快排算法的运行时间是nlgn，  \* 就地排序  \* 不是稳定的算法。  \* 由递归造成的空间复杂度O(lgn)~O(n)  \*/  **public** **int** partition(**int** A[],**int** low,**int** high){    **int** pivot = A[high];  **int** i=low-1;    **for**(**int** j=low;j<high;j++){ //保证所有比pivot小的元素都放在A[low]...A[i]中  **if**(A[j]<=pivot){  i++;  **int** temp = A[j];  A[j] = A[i];  A[i] = temp;  }  }    **int** temp2 = A[high];  A[high] = A[i+1];  A[i+1] = temp2;    **return** i+1;  }    **public** **void** quick\_sort(**int** A[],**int** low,**int** high){  **if**(low < high){  **int** mid = partition(A,low,high);  quick\_sort(A,low,mid-1);  quick\_sort(A,mid+1,high);  }  } |

1. 计数排序

假设n个输入元素中的每一个都是在0到k之间的整数，其中k为某个整数。计数排序的时间复杂度是O(n+k)，空间复杂度是O(n+k)。当k=O(n)时，排序的时间复杂度和空间复杂度为O(n).

非就地排序。稳定的排序算法。

|  |
| --- |
| /\*\*  \* 非就地排序  \* 稳定的排序算法  \* 时间复杂度O(n+k)  \* 空间复杂度O(n+k)  \*  \* 数组A中的元素都在0到k之间。  \* **@param** A 待排序的数组  \* **@param** B 排序之后存放结果的数组  \* **@param** k  \*/  **public** **void** countSort(**int**[] A,**int**[] B,**int** k){  **int** len = A.length;  /\*C[i]中存的是整数i在数组A中小于等于i的个数\*/  **int**[] C = **new** **int**[k+1];  **for**(**int** i=0;i<len;i++){  C[A[i]]++;  }  **for**(**int** i=1;i<=k;i++){  C[i]=C[i-1]+C[i];  }  System.*out*.println("排序后 ： "+Arrays.*toString*(C));  /\*从后向前遍历数组A的元素，保证其稳定的插入到正确的位置\*/  **for**(**int** i=len-1;i>=0;i--){  B[C[A[i]]-1] = A[i];  C[A[i]]--;  }  System.*out*.println("排序后 ： "+Arrays.*toString*(B));  }    **public** **static** **void** main(String[] a){  CountSort test = **new** CountSort();  **int**[] A ={4,1,3,4,3};  **int**[] B = **new** **int**[5];  test.countSort(A, B, 4);  } |

1. 基数排序

从最低有效位开始依次往高有效位对输入的n个元素进行排序，或者从高有效位开始依次往低有效位对输入的n个元素进行排序。

给定的n个的d位数，其中每一个数位有k个可能的取值。如果对每个有效位排序采用的是稳定的计数排序算法，则时间复杂度是O(d(n+k)),空间复杂度也是O(d(n+k))。

|  |
| --- |
| /\*\*  \* 基数排序。  \* 给定的n个的d位数，其中每一个数位有k个可能的取值。  \* 如果对每个有效位排序采用的是稳定的计数排序算法，则时间复杂度是O(d(n+k)),空间复杂度也是O(d(n+k))  \* **@param** A  \*/  **public** **void** RadixSort(**int**[] A,**int** d,**int** k){  **for**(**int** i = 0;i<d;i++){  countSort(A,i,10);  }  }  /\*\*  \* 对数组A中所有元素按照第i个有效位进行排序。  \* **@param** A  \* **@param** d  \* **@param** k  \*/  **public** **void** countSort(**int**[] A,**int** i,**int** k){  } |

1. 桶排序

本排序算法的总结摘自：http://www.cnblogs.com/openeim/p/3921645.html

基本思想：是将阵列分到有限数量的桶子里。每个桶子再个别排序（有可能再使用别的排序算法或是以递回方式继续使用桶排序进行排序）。桶排序是鸽巢排序的一种归纳结果。当要被排序的阵列内的数值是均匀分配的时候，桶排序使用线性时间（Θ（n））。但桶排序并不是比较排序，他不受到 O(n log n) 下限的影响。

简单来说，就是把数据分组，放在一个个的桶中，然后对每个桶里面的在进行排序。例如要对大小为[1..1000]范围内的n个整数A[1..n]排序首先，可以把桶设为大小为10的范围，具体而言，设集合B[1]存储[1..10]的整数，集合B[2]存储(10..20]的整数，……集合B[i]存储( (i-1)\*10, i\*10]的整数，i = 1,2,..100。总共有 100个桶。

然后，对A[1..n]从头到尾扫描一遍，把每个A[i]放入对应的桶B[j]中。 再对这100个桶中每个桶里的数字排序，这时可用冒泡，选择，乃至快排，一般来说任 何排序法都可以。最后，依次输出每个桶里面的数字，且每个桶中的数字从小到大输出，这样就得到所有数字排好序的一个序列了.

1. 两种二分查找

返回第一个次在数组中出现的下标：

|  |
| --- |
| /\*如果关键字在数组中存在则返回第一次出现的下标，不存在的时候则返回-1\*/  public int binarySearch11(int nums[],int key){//返回的是与关键字相等的元素下界,即如果有多个数组元素与关键字相等的时候，返回的是第一个元素的下标。  int high = nums.length-1;  int low = 0;  int mid = 0;  while(low<high){ //当数组中不存在的时候且比数组元素都大的时候返回的下标是len  mid=low + (high-low)/2;  System.out.println("mid = "+mid);  if(nums[mid]<key){  low = mid+1;  }else {  high = mid;  }  }  return nums[low]==key?low:-1;  } |

返回最后一次在下标中出现的下标：

|  |
| --- |
| int upperbound(int[] nums, int target) { // 找关键字在数组中出现的上界  int high = nums.length - 1;  int low = 0;  int mid = low + (high - low + 1) / 2;  while (low <high) {  mid = low + (high - low + 1) / 2;  if (nums[mid] <= target) {  low = mid ;  } else {  high = mid - 1;  }  }    return nums[low] == target ? low : -1; //这样就避免了溢出  } |