1. 算法工程师定义
2. 算法（Algorithm）是一系列解决问题的清晰指令，也就是说，能够对一定规范的输入，在有限时间内获得所要求的输出。如果一个算法有缺陷，或不适合于某个问题，执行这个算法将不会解决这个问题。不同的算法可能用不同的时间、空间或效率来完成同样的任务。一个算法的优劣可以用空间复杂度与时间复杂度来衡量。  
   　　算法可以理解为有基本运算及规定的运算顺序所构成的完整的解题步骤。或者看成按照要求设计好的有限的确切的计算序列，并且这样的步骤和序列可以解决一类问题。  
   　　一个算法应该具有以下五个重要的特征：  
   　　1、有穷性： 一个算法必须保证执行有限步之后结束；  
   　　2、确切性： 算法的每一步骤必须有确切的定义；  
   　　3、输入：一个算法有0个或多个输入，以刻画运算对象的初始情况，所谓0个输入是指算法本身定除了初始条件；  
   　　4、输出：一个算法有一个或多个输出，以反映对输入数据加工后的结果。没有输出的算法是毫无意义的；  
   　　5、可行性： 算法原则上能够精确地运行，而且人们用笔和纸做有限次运算后即可完成。  
   　　计算机科学家尼克劳斯-沃思曾著过一本著名的书《数据结构十算法= 程序》，可见算法在计算机科学界与计算机应用界的地位。  
   [编辑本段]  
   算法的复杂度  
   　　同一问题可用不同算法解决，而一个算法的质量优劣将影响到算法乃至程序的效率。算法分析的目的在于选择合适算法和改进算法。一个算法的评价主要从时间复杂度和空间复杂度来考虑。  
   　　时间复杂度  
   　　算法的时间复杂度是指算法需要消耗的时间资源。一般来说，计算机算法是问题规模n 的函数f(n)，算法的时间复杂度也因此记做  
   　　T(n)=Ο(f(n))  
   　　因此，问题的规模n 越大，算法执行的时间的增长率与f(n) 的增长率正相关，称作渐进时间复杂度（Asymptotic Time Complexity）。  
   　　空间复杂度  
   　　算法的空间复杂度是指算法需要消耗的空间资源。其计算和表示方法与时间复杂度类似，一般都用复杂度的渐近性来表示。同时间复杂度相比，空间复杂度的分析要简单得多。  
   　　详见百度百科词条"算法复杂度"  
   [编辑本段]  
   算法设计与分析的基本方法  
   　　1.递推法  
   　　递推法是利用问题本身所具有的一种递推关系求问题解的一种方法。它把问题分成若干步，找出相邻几步的关系，从而达到目的，此方法称为递推法。  
   　　2.递归  
   　　递归指的是一个过程：函数不断引用自身，直到引用的对象已知  
   　　3.穷举搜索法  
   　　穷举搜索法是对可能是解的众多候选解按某种顺序进行逐一枚举和检验，并从众找出那些符合要求的候选解作为问题的解。  
   　　4.贪婪法  
   　　贪婪法是一种不追求最优解，只希望得到较为满意解的方法。贪婪法一般可以快速得到满意的解，因为它省去了为找最优解要穷尽所有可能而必须耗费的大量时间。贪婪法常以当前情况为基础作最优选择，而不考虑各种可能的整体情况，所以贪婪法不要回溯。  
   　　5.分治法  
   　　把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。  
   　　6.动态规划法  
   　　动态规划是一种在数学和计算机科学中使用的，用于求解包含重叠子问题的最优化问题的方法。其基本思想是，将原问题分解为相似的子问题，在求解的过程中通过子问题的解求出原问题的解。动态规划的思想是多种算法的基础，被广泛应用于计算机科学和工程领域。  
   　　7.迭代法  
   　　迭代是数值分析中通过从一个初始估计出发寻找一系列近似解来解决问题（一般是解方程或者方程组）的过程，为实现这一过程所使用的方法统称为迭代法。  
   [编辑本段]  
   算法分类  
   　　算法可大致分为基本算法、数据结构的算法、数论与代数算法、计算几何的算法、图论的算法、动态规划以及数值分析、加密算法、排序算法、检索算法、随机化算法、并行算法。  
   　　算法可以宏泛的分为三类：  
   　　有限的，确定性算法 这类算法在有限的一段时间内终止。他们可能要花很长时间来执行指定的任务，但仍将在一定的时间内终止。这类算法得出的结果常取决于输入值。  
   　　有限的，非确定算法 这类算法在有限的时间内终止。然而，对于一个（或一些）给定的数值，算法的结果并不是唯一的或确定的。  
   　　无限的算法 是那些由于没有定义终止定义条件，或定义的条件无法由输入的数据满足而不终止运行的算法。通常，无限算法的产生是由于未能确定的定义终止条件。  
   [编辑本段]  
   举例  
   　　经典的算法有很多，如："欧几里德算法，割圆术，秦九韶算法"。  
   [编辑本段]  
   算法经典专著  
   　　目前市面上有许多论述算法的书籍，其中最著名的便是《计算机程序设计艺术》（The Art Of Computer Programming) 以及《算法导论》（Introduction To Algorithms）。  
   [编辑本段]  
   算法的历史  
   　　“算法”即演算法的大陆中文名称出自《周髀算经》；而英文名称Algorithm 来自于9世纪波斯数学家al-Khwarizmi，因为al-Khwarizmi在数学上提出了算法这个概念。“算法”原为"algorism"，意思是阿拉伯数字的运算法则，在18世纪演变为"algorithm"。欧几里得算法被人们认为是史上第一个算法。 第一次编写程序是Ada Byron于1842年为巴贝奇分析机编写求解解伯努利方程的程序，因此Ada Byron被大多数人认为是世界上第一位程序员。因为查尔斯·巴贝奇(Charles Babbage)未能完成他的巴贝奇分析机，这个算法未能在巴贝奇分析机上执行。 因为"well-defined procedure"缺少数学上精确的定义，19世纪和20世纪早期的数学家、逻辑学家在定义算法上出现了困难。20世纪的英国数学家图灵提出了著名的图灵论题，并提出一种假想的计算机的抽象模型，这个模型被称为图灵机。图灵机的出现解决了算法定义的难题，图灵的思想对算法的发展起到了重要作用的。

5. ===========================================================================  
   相关知识介绍（所有定义只为帮助读者理解相关概念，并非严格定义）：  
   1、稳定排序和非稳定排序  
      
    简单地说就是所有相等的数经过某种排序方法后，仍能保持它们在排序之前的相对次序，我们就  
   说这种排序方法是稳定的。反之，就是非稳定的。  
    比如：一组数排序前是a1,a2,a3,a4,a5，其中a2=a4，经过某种排序后为a1,a2,a4,a3,a5，  
   则我们说这种排序是稳定的，因为a2排序前在a4的前面，排序后它还是在a4的前面。假如变成a1,a4,  
   a2,a3,a5就不是稳定的了。
6. 2、内排序和外排序
7. 在排序过程中，所有需要排序的数都在内存，并在内存中调整它们的存储顺序，称为内排序；  
    在排序过程中，只有部分数被调入内存，并借助内存调整数在外存中的存放顺序排序方法称为外排序。
8. 3、算法的时间复杂度和空间复杂度
9. 所谓算法的时间复杂度，是指执行算法所需要的计算工作量。  
    一个算法的空间复杂度，一般是指执行这个算法所需要的内存空间。  
   =====================================================  
   ================================================  
    功能：**选择排序**  
    输入：数组名称（也就是数组首地址）、数组中元素个数  
   ====================================================  
   算法思想简单描述：
10. 在要排序的一组数中，选出最小的一个数与第一个位置的数交换；  
     然后在剩下的数当中再找最小的与第二个位置的数交换，如此循环  
     到倒数第二个数和最后一个数比较为止。
11. 选择排序是不稳定的。算法复杂度O(n2)--[n的平方]  
    =====================================================  
    \*/  
    void select\_sort(int \*x, int n)  
    {  
     int i, j, min, t;
12. for (i=0; i<n-1; i++) /\*要选择的次数：0~n-2共n-1次\*/  
     {  
      min = i; /\*假设当前下标为i的数最小，比较后再调整\*/  
      for (j=i+1; j<n; j++)/\*循环找出最小的数的下标是哪个\*/  
      {  
       if (\*(x+j) < \*(x+min))  
       {     
        min = j; /\*如果后面的数比前面的小，则记下它的下标\*/  
       }  
      }    
        
      if (min != i) /\*如果min在循环中改变了，就需要交换数据\*/  
      {  
       t = \*(x+i);  
       \*(x+i) = \*(x+min);  
       \*(x+min) = t;  
      }  
     }  
    }  
    ================================================  
     功能：**直接插入排序**  
     输入：数组名称（也就是数组首地址）、数组中元素个数  
    ====================================================  
    算法思想简单描述：
13. 在要排序的一组数中，假设前面(n-1) [n>=2] 个数已经是排  
     好顺序的，现在要把第n个数插到前面的有序数中，使得这n个数  
     也是排好顺序的。如此反复循环，直到全部排好顺序。  
     直接插入排序是稳定的。算法时间复杂度O(n2)--[n的平方]  
    =====================================================  
    \*/  
    void insert\_sort(int \*x, int n)  
    {  
     int i, j, t;
14. for (i=1; i<n; i++) /\*要选择的次数：1~n-1共n-1次\*/  
     {  
      /\*  
       暂存下标为i的数。注意：下标从1开始，原因就是开始时  
       第一个数即下标为0的数，前面没有任何数，单单一个，认为  
       它是排好顺序的。  
      \*/  
      t=\*(x+i);  
      for (j=i-1; j>=0 && t<\*(x+j); j--) /\*注意：j=i-1，j--，这里就是下标为i的数，在它前面有序列中找插入位置。\*/  
      {  
       \*(x+j+1) = \*(x+j); /\*如果满足条件就往后挪。最坏的情况就是t比下标为0的数都小，它要放在最前面，j==-1，退出循环\*/  
      }
15. \*(x+j+1) = t; /\*找到下标为i的数的放置位置\*/  
     }  
    }  
    ================================================  
     功能：**冒泡排序**  
     输入：数组名称（也就是数组首地址）、数组中元素个数  
    ====================================================  
    算法思想简单描述：
16. 在要排序的一组数中，对当前还未排好序的范围内的全部数，自上  
     而下对相邻的两个数依次进行比较和调整，让较大的数往下沉，较  
     小的往上冒。即：每当两相邻的数比较后发现它们的排序与排序要  
     求相反时，就将它们互换。  
     下面是一种改进的冒泡算法，它记录了每一遍扫描后最后下沉数的  
     位置k，这样可以减少外层循环扫描的次数。
17. 冒泡排序是稳定的。算法时间复杂度O(n2)--[n的平方]  
    =====================================================
18. void bubble\_sort(int \*x, int n)  
    {  
     int j, k, h, t;  
        
     for (h=n-1; h>0; h=k) /\*循环到没有比较范围\*/  
     {  
      for (j=0, k=0; j<h; j++) /\*每次预置k=0，循环扫描后更新k\*/  
      {  
       if (\*(x+j) > \*(x+j+1)) /\*大的放在后面，小的放到前面\*/  
       {  
        t = \*(x+j);  
        \*(x+j) = \*(x+j+1);  
        \*(x+j+1) = t; /\*完成交换\*/  
        k = j; /\*保存最后下沉的位置。这样k后面的都是排序排好了的。\*/  
       }  
      }  
     }  
    }
19. ================================================  
     功能：**希尔排序**  
     输入：数组名称（也就是数组首地址）、数组中元素个数  
    ====================================================  
    算法思想简单描述：  
     在直接插入排序算法中，每次插入一个数，使有序序列只增加1个节点，  
     并且对插入下一个数没有提供任何帮助。如果比较相隔较远距离（称为  
     增量）的数，使得数移动时能跨过多个元素，则进行一次比较就可能消除  
     多个元素交换。D.L.shell于1959年在以他名字命名的排序算法中实现  
     了这一思想。算法先将要排序的一组数按某个增量d分成若干组，每组中  
     记录的下标相差d.对每组中全部元素进行排序，然后再用一个较小的增量  
     对它进行，在每组中再进行排序。当增量减到1时，整个要排序的数被分成  
     一组，排序完成。  
     下面的函数是一个希尔排序算法的一个实现，初次取序列的一半为增量，  
     以后每次减半，直到增量为1。
20. 希尔排序是不稳定的。  
    void shell\_sort(int \*x, int n)  
    {  
     int h, j, k, t;
21. for (h=n/2; h>0; h=h/2) /\*控制增量\*/  
     {  
      for (j=h; j<n; j++) /\*这个实际上就是上面的直接插入排序\*/  
      {  
       t = \*(x+j);  
       for (k=j-h; (k>=0 && t<\*(x+k)); k-=h)  
       {  
        \*(x+k+h) = \*(x+k);  
       }  
       \*(x+k+h) = t;  
      }  
     }  
    }
22. ====================================================
23. 功能：**快速排序**  
     输入：数组名称（也就是数组首地址）、数组中起止元素的下标  
    ====================================================  
    算法思想简单描述：
24. 快速排序是对冒泡排序的一种本质改进。它的基本思想是通过一趟  
     扫描后，使得排序序列的长度能大幅度地减少。在冒泡排序中，一次  
     扫描只能确保最大数值的数移到正确位置，而待排序序列的长度可能只  
     减少1。快速排序通过一趟扫描，就能确保某个数（以它为基准点吧）  
     的左边各数都比它小，右边各数都比它大。然后又用同样的方法处理  
     它左右两边的数，直到基准点的左右只有一个元素为止。它是由  
     C.A.R.Hoare于1962年提出的。  
     显然快速排序可以用递归实现，当然也可以用栈化解递归实现。下面的  
     函数是用递归实现的，有兴趣的朋友可以改成非递归的。
25. 快速排序是不稳定的。最理想情况算法时间复杂度O(nlog2n)，最坏O(n2)  
       
    =====================================================  
    \*/  
    void quick\_sort(int \*x, int low, int high)  
    {  
     int i, j, t;
26. if (low < high) /\*要排序的元素起止下标，保证小的放在左边，大的放在右边。这里以下标为low的元素为基准点\*/  
     {  
      i = low;  
      j = high;  
      t = \*(x+low); /\*暂存基准点的数\*/
27. while (i<j) /\*循环扫描\*/  
      {  
       while (i<j && \*(x+j)>t) /\*在右边的只要比基准点大仍放在右边\*/  
       {  
        j--; /\*前移一个位置\*/  
       }
28. if (i<j)   
       {  
        \*(x+i) = \*(x+j); /\*上面的循环退出：即出现比基准点小的数，替换基准点的数\*/  
        i++; /\*后移一个位置，并以此为基准点\*/  
       }
29. while (i<j && \*(x+i)<=t) /\*在左边的只要小于等于基准点仍放在左边\*/  
       {  
        i++; /\*后移一个位置\*/  
       }
30. if (i<j)  
       {  
        \*(x+j) = \*(x+i); /\*上面的循环退出：即出现比基准点大的数，放到右边\*/  
        j--; /\*前移一个位置\*/  
       }  
      }
31. \*(x+i) = t; /\*一遍扫描完后，放到适当位置\*/  
      quick\_sort(x,low,i-1);  /\*对基准点左边的数再执行快速排序\*/  
      quick\_sort(x,i+1,high);  /\*对基准点右边的数再执行快速排序\*/  
     }  
    }
32. ================================================  
     功能：**堆排序**  
     输入：数组名称（也就是数组首地址）、数组中元素个数  
    ====================================================  
    算法思想简单描述：
33. 堆排序是一种树形选择排序，是对直接选择排序的有效改进。  
     堆的定义如下：具有n个元素的序列（h1,h2,...,hn),当且仅当  
     满足（hi>=h2i,hi>=2i+1）或（hi<=h2i,hi<=2i+1）(i=1,2,...,n/2)  
     时称之为堆。在这里只讨论满足前者条件的堆。
34. 由堆的定义可以看出，堆顶元素（即第一个元素）必为最大项。完全二叉树可以  
     很直观地表示堆的结构。堆顶为根，其它为左子树、右子树。  
     初始时把要排序的数的序列看作是一棵顺序存储的二叉树，调整它们的存储顺序，  
     使之成为一个堆，这时堆的根节点的数最大。然后将根节点与堆的最后一个节点  
     交换。然后对前面(n-1)个数重新调整使之成为堆。依此类推，直到只有两个节点  
     的堆，并对它们作交换，最后得到有n个节点的有序序列。
35. 从算法描述来看，堆排序需要两个过程，一是建立堆，二是堆顶与堆的最后一个元素  
     交换位置。所以堆排序有两个函数组成。一是建堆的渗透函数，二是反复调用渗透函数  
     实现排序的函数。
36. 堆排序是不稳定的。算法时间复杂度O(nlog2n)。
37. ====================================================  
     功能：**渗透建堆**  
     输入：数组名称（也就是数组首地址）、参与建堆元素的个数、从第几个元素开始  
    \*/  
    void sift(int \*x, int n, int s)  
    {  
     int t, k, j;
38. t = \*(x+s); /\*暂存开始元素\*/  
     k = s;  /\*开始元素下标\*/  
     j = 2\*k + 1; /\*右子树元素下标\*/
39. while (j<n)  
     {  
      if (j<n-1 && \*(x+j) < \*(x+j+1))/\*判断是否满足堆的条件：满足就继续下一轮比较，否则调整。\*/  
      {  
       j++;  
      }
40. if (t<\*(x+j)) /\*调整\*/  
      {  
       \*(x+k) = \*(x+j);  
       k = j; /\*调整后，开始元素也随之调整\*/  
       j = 2\*k + 1;  
      }  
      else /\*没有需要调整了，已经是个堆了，退出循环。\*/  
      {  
       break;  
      }  
     }  
       
     \*(x+k) = t; /\*开始元素放到它正确位置\*/  
    }  
    ====================================================  
     功能：**堆排序**  
     输入：数组名称（也就是数组首地址）、数组中元素个数  
    \*/  
    void heap\_sort(int \*x, int n)  
    {  
     int i, k, t;  
     int \*p;
41. for (i=n/2-1; i>=0; i--)  
     {  
      sift(x,n,i); /\*初始建堆\*/  
     }   
       
     for (k=n-1; k>=1; k--)  
     {  
      t = \*(x+0); /\*堆顶放到最后\*/  
      \*(x+0) = \*(x+k);  
      \*(x+k) = t;  
      sift(x,k,0); /\*剩下的数再建堆\*/   
     }  
    }
42. void main()  
    {   
     #define MAX 4  
     int \*p, i, a[MAX];  
       
     /\*录入测试数据\*/  
     p = a;  
     printf("Input %d number for sorting :\n",MAX);  
     for (i=0; i<MAX; i++)  
     {  
      scanf("%d",p++);  
     }  
     printf("\n");
43. /\*测试选择排序\*/
44. p = a;  
     select\_sort(p,MAX);  
     /\*\*/
45. /\*测试直接插入排序\*/
46. /\*  
     p = a;  
     insert\_sort(p,MAX);  
     \*/  
     /\*测试冒泡排序\*/
47. /\*  
     p = a;  
     insert\_sort(p,MAX);  
     \*/
48. /\*测试快速排序\*/
49. /\*  
     p = a;  
     quick\_sort(p,0,MAX-1);  
     \*/
50. /\*测试堆排序\*/
51. /\*  
     p = a;  
     heap\_sort(p,MAX);  
     \*/
52. for (p=a, i=0; i<MAX; i++)  
     {  
      printf("%d ",\*p++);  
     }  
       
     printf("\n");  
     system("pause");  
    }