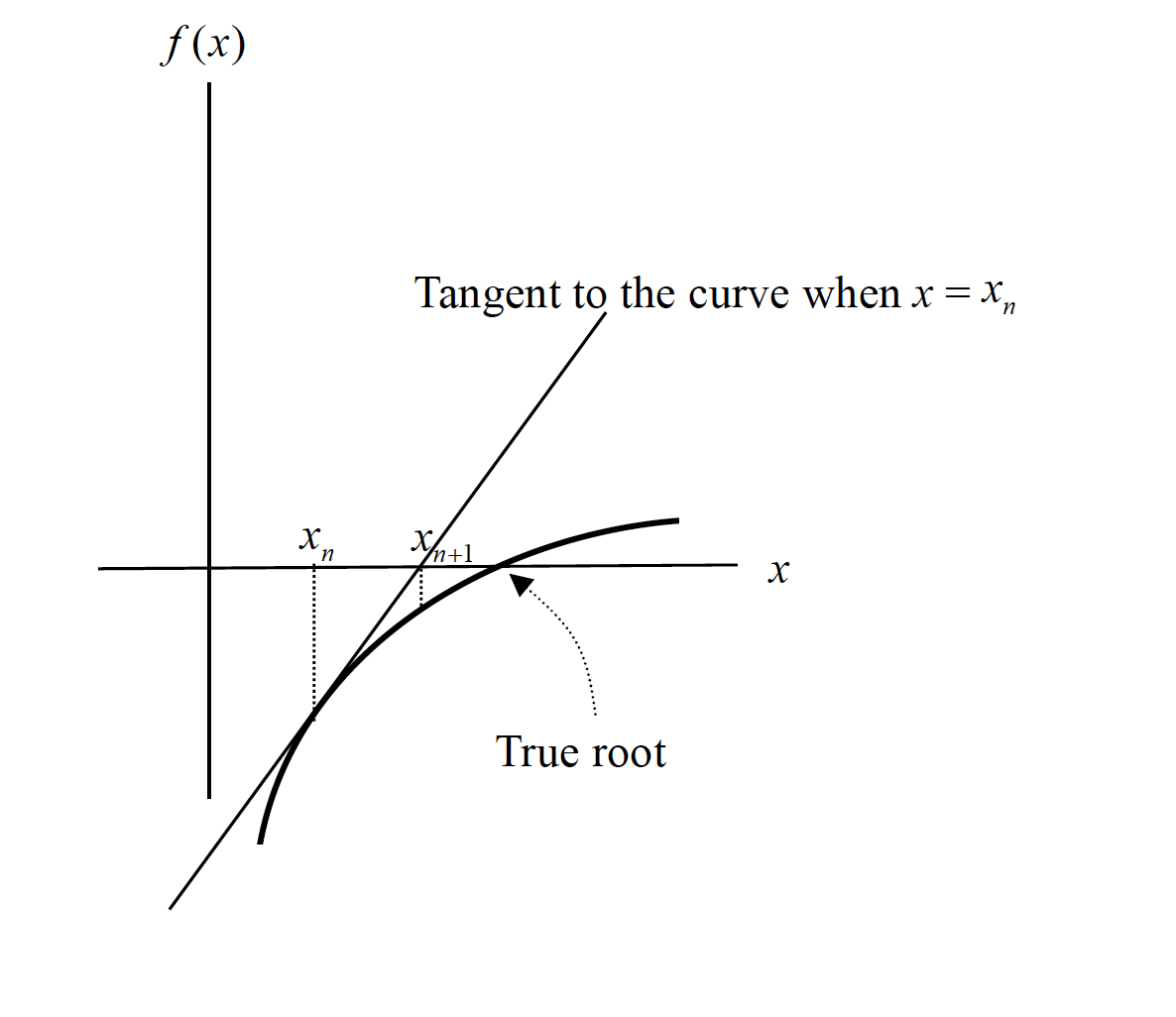
Algorithms in Basic C Programming

Phase 1

误人子弟者：SM2 21st 最菜的人

1. Newton-Raphson’s Method



算法描述：

在函数图像上任取一点(x0,f(x0))作为起始点，求该点的切线，将切线与x轴的交点横坐标x1作为新的起始点横坐标，以(x1,f(x1))为起始点重复此操作，直到f(xn)足够接近0为止。

注意事项：

某些情况下这种方法会死循环（如果(x1,f(x1))的切线与x轴的交点是(x0,0)就尴尬了）；并且对于有多个解的函数，该方法的起始点可能影响最终得到哪个解

实现要点：

xn+1 = xn-[f(xn)/f’(xn)]

#include **<stdio.h>**#define **f**(**x**) ((**x**)\*(**x**)+2\*(**x**)-3)  
#define **fpr**(**x**) (2\*(**x**)+2)  
#define **torlerance** 0.00001  
  
**double** New(**double** init){  
 **double** now = init;  
 **while**(fabs(**f**(now))>**torlerance**){  
 now = now - **f**(now)/**fpr**(now);  
 }  
 **return** now;  
}

**double** Ral(**double** now){  
 **if**(fabs(**f**(now))<=**torlerance**) **return** now;  
 **else return** Ral(now - **f**(now)/**fpr**(now));  
}

1. Selection Sort

算法描述：

将最小的放在第一个，然后只考虑从第二个开始的数列，将新数列中最小的放在新数列的第一个，以此类推。

时间复杂度分析：

根本就没有什么可以break的地方，所以选择排序是稳定排序，对于n个数字的数组：

比较次数为：(n-1)+(n-2)+…+1 = n\*(n-1)/2

交换次数为：n

时间复杂度为：O(n^2)

（写错函数名了不要在意）

**void** Bub(**int** \*arr, **int** length){  
 **int** i,j,index,now,temp;  
 **for**(i=0;i<length;i++){  
 now = arr[i];  
 index = i;  
 **for**(j=i+1;j<length;j++){  
 **if**(arr[j]<now) {  
 index = j;  
 now = arr[j];  
 }  
 }  
 temp = arr[i];  
 arr[i] = arr[index];  
 arr[index] = temp;  
 }  
 **return**;  
}  
  
**void** Ble(**int** \*arr, **int** length){  
 **int** j,index,now,temp;  
 **if**(length<=0) **return**;  
 **else**{  
 now = arr[0];  
 index = 0;  
 **for**(j=1;j<length;j++){  
 **if**(arr[j]<now) {  
 index = j;  
 now = arr[j];  
 }  
 }  
 temp = arr[0];  
 arr[0] = arr[index];  
 arr[index] = temp;  
 }  
 Ble(arr+1,length-1);  
 **return**;  
}

1. Bubble Sort

算法描述：

就是冒泡泡嘛！

实现要点：

在某一次冒泡的时候如果没有发生交换，就可以不用再继续比较了（否则冒泡排序就是稳定n^2排序了🤦‍）

时间复杂度分析：（对于大小为n的数组）

最差情况——倒序数组，比较次数和交换次数相等，为n\*(n-1)/2

最好情况——正序数组，在第一次比较n-1次以后，就没有继续比较

故最好情况时间复杂度为O(n)；最坏情况为O(n^2)

**void** Bubble1(**int** \*arr, **int** length){  
 **int** i,j,temp,exchanged = 0;  
 **for**(i=length-1;i>=1;i--) {  
 **for** (j = 0; j < i; j++) {  
 **if** (arr[j] > arr[j + 1]) {  
 temp = arr[j];  
 arr[j] = arr[j + 1];  
 arr[j + 1] = temp;  
 exchanged = 1;  
 }  
 }  
 **if**(!exchanged) **break**;  
 **else** exchanged = 0;  
 }  
 **return**;  
}  
  
**void** Bubble2(**int** \*arr, **int** length){  
 **static int** exchanged = 0;  
 **int** i,temp;  
 **if**(length<=1) **return**;  
 **for**(i=0;i<length-1;i++){  
 **if**(arr[i]>arr[i+1]){  
 temp = arr[i];  
 arr[i] = arr[i+1];  
 arr[i+1] = temp;  
 exchanged = 1;  
 }  
 }  
 **if**(exchanged) {  
 exchanged = 0;  
 Bubble2(arr,length-1);  
 }  
 **else return**;  
 **return**;  
}

1. Heap Sort

算法描述：

堆性质：父节点大于等于任意一个子节点，对于任意节点均符合这一条件就说这一颗二叉树具有堆性质。

建立堆：每次将新节点加在一棵树的最后一个节点，利用sifting-up上浮操作将该新节点放在正确的位置。

Sifting-up：和自己的父节点比较，如果自己比父节点大，就交换自己和父节点，直到父节点大于等于自己或到达根节点（root node）为止。~~因为原二叉树是具有堆性质的，所以父节点大于自己的所有直系后代，将父节点下沉仍然保证了父节点比下面的节点大；因为自己比父节点大，上浮以后不会影响自己之下的树的堆性质，~~故可以证明该方法的正确性。

堆排序：利用堆顶元素是最大元素的性质，将最后一个元素与堆顶元素交换，从此无视最后一个元素，然后恢复堆性质，再将堆顶元素与新堆的最后一个元素互换（注意此时的“最后一个”是无视之前放下来的元素后的最后一个，即“新堆”会越来越小，每次少1个），直到新堆只剩下一个元素为止。

恢复堆性质：基于堆排序的方法，除了堆顶元素（被交换上去的那个）以外，树的其他部分仍具有堆性质，所以将堆顶元素依照sifting-up的类似方法下浮至正确位置即可；

时间复杂度分析：

建立堆：假设每次元素都会上浮到堆顶，上浮次数为；大概就当做吧！（）

恢复堆：假设每次元素都会上浮到堆顶，上浮次数为；大概就当做吧！（）

加起来：假设每次元素都会上浮到堆顶，上浮次数为；大概就当做吧！（）

时间复杂度：O()

实现注意事项：

用线性数组表示二叉树时，a[i]的左儿子是a[2\*i]，右儿子是a[2\*i+1]；当j是偶数时，a[j]的父节点是a[j/2]；当j是奇数时，a[j]的父节点是a[(j-1)/2]

**void** hpst(**int** \*arr, **int** length){  
 **int** heap[200];  
 **int** i,j,temp,now,k,ex1,ex2;  
 *//建立堆* **for**(i=0;i<length;i++){  
 ex1 = 1; now = i;  
 *//将元素放置在当前堆的末尾* heap[i] = arr[i];  
 **while**(ex1){  
 ex1 = 0;  
 **if**(now>0 && now%2){  
 *//如果当前元素的编号是一个奇数的话  
 //父节点编号就应该是(now-1)/2* **if**(heap[(now-1)/2]<heap[now]){  
 ex1 = 1;  
 temp = heap[now];  
 heap[now] = heap[(now-1)/2];  
 heap[(now-1)/2] = temp;  
 now = (now-1)/2;  
 }  
 } **else** {  
 *//如果当前元素的编号是一个偶数的话  
 //父节点编号就应该是now/2-1* **if**(heap[now/2-1]<heap[now]){  
 ex1 = 1;  
 temp = heap[now];  
 heap[now] = heap[now/2-1];  
 heap[now/2-1] = temp;  
 now = now/2-1;  
 }  
 }  
 }  
 }  
 *//删除堆顶元素并恢复堆* **while**(length>1){  
 *//交换堆顶元素与末尾元素，并将新堆长度-1* temp = heap[0];  
 heap[0] = heap[length-1];  
 heap[length-1] = temp;  
 length--;  
 *//下浮堆顶元素恢复堆性质* ex2 = 1; now = 0;  
 **while**(ex2){  
 ex2 = 0; k = heap[now];  
 **if**(2\*now+1<length && k < heap[2\*now+1]){  
 k = heap[2\*now+1];  
 ex2 = 1;  
 j = 2\*now+1;  
 }  
 **if**(2\*now+2<length && k < heap[2\*now+2]){  
 k = heap[2\*now+2];  
 ex2 = 1;  
 j = 2\*now+2;  
 }  
 **if**(ex2){  
 temp = heap[j];  
 heap[j] = heap[now];  
 heap[now] = temp;  
 now = j;  
 }  
 }  
 }  
  
 **for**(**int** i=0;i<9;i++)  
 printf(**"%d "**,heap[i]);  
 printf(**"\n"**);  
  
 **return**;  
}

1. Bucket Sort

太难了不会，tedious.

1. Insertion Sort

算法描述：

对于已经有序的一个数列，将一个新数字插入这个数列有两种办法：

1. 从左往右比较，找到第一个比新数字大的数，插在这个数的前面
2. 从右向左比较，找到第一个比新数字小的数，插在这个数的后面

因此，把单个数字看做有序数列，可以以此类推排序一个数组

时间复杂度分析：

对于从左向右比较，对于长度为n的原数列，比较次数与赋值次数总和始终为n+1，故该排序的总操作次数为2+3+…+n≈O(n^2)

对于从右向左比较，最好情况下原数列已经有序，每次只比较新数字的前一个便停止，故最好情况下总操作次数为n-1次比较=O(n)；最差情况下原数列倒序，对于长度为n的原数列，比较次数与赋值次数之和为2n+1，总操作次数仍在O(n^2)级别

实现注意事项：

从左向右比较，如果没有数字比该数字大，该数字就留在原地

※从右向左比较，如果没有数字比该数字小，该数字就插到最前面

从左向右比较：

**for**(i=0;i<MAXN;i++){  
 **for**(j=0;j<=i;j++){  
 **if**(arr[j]>arr[i]){  
 **for**(k=j;k<i;k++) swap(&arr[i],&arr[k]);  
 **break**;  
 }  
 }  
}

从右向左比较：（我这里从右开始考察了，避免了一些实现上细节的问题…当然你也可以从左开始考察，即最外层i从0开始）

**for**(i=MAXN-2;i>=0;i--){  
 **for**(j=MAXN-1;j>i;j--){  
 **if**(arr[j]<arr[i]){  
 **for**(k=j;k>i;k--) swap(&arr[i],&arr[k]);  
 **break**;  
 }  
 }  
}

1. Grid Sort

算法描述：

对于排在k\*k网格中的数据，行排序、列排序以后的数据具有了“堆性质”——右边的数与左边的数一定不小于自己；因此**对于一个在(x,y)处的数据，它的最终停留位置一定在[(1,y+1);(x-1,k)]的范围内**。（如图示，\*表示数据，红色部分表示该数据最终的停留位置）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  | \* |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Figure 随便编号多少了

所以对于每一个数据，把它和它的下一行的第一个与它的右边比较，如果它比较大，就和“邻居”中比较小的那个交换，交换后以新起点继续这样的操作，直到没有交换发生为止。

对每一个元素做这样的操作。

时间复杂度分析：

行排序、列排序假设使用冒泡排序，时间复杂度为O(k^2)

恢复过程中，最差情况如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | 0 |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 | 5 | 6 | 7 | 8 |

Figure 不知道多少

所以恢复的时间复杂度为O(k^3)

总的而言时间复杂度为O(k^3) = O(n^1.5)

*// pre-processing row sort to be programmed by you***for**(i=0;i<n;i++){  
 **for**(j=n-1;j>=0;j--) {  
 **for**(k=0;k<j;k++){  
 **if**(G[k][i]>G[k+1][i]) swap(&G[k][i],&G[k+1][i]);  
 }  
 }  
}  
*// pre-processing column sort to be programmed by you***for**(i=0;i<n;i++){  
 **for**(j=n-1;j>=0;j--) {  
 **for**(k=0;k<j;k++){  
 **if**(G[i][k]>G[i][k+1]) swap(&G[i][k],&G[i][k+1]);  
 }  
 }  
}  
*// exchange and heap restoration to be programmed by you***for**(i=0;i<n;i++){  
 **for**(j=0;j<n;j++){  
 **if**(i+1 < n && G[i][j]>G[i+1][0]){  
 swap(&G[i][j],&G[i+1][0]);  
 }  
 row = i+1; col = 0;  
 **do**{  
 **if**(row+1 < n && G[row][col]>G[row+1][0]){  
 interchange = 1;  
 **if**(col+1 < n && G[row][col+1]<G[row+1][col]){  
 swap(&G[row][col],&G[row][col+1]);  
 col += 1;  
 } **else** {  
 swap(&G[row][col],&G[row+1][0]);  
 row += 1; col = 0;  
 }  
 } **else if**(col+1 < n && G[row][col]>G[row][col+1]){  
 swap(&G[row][col],&G[row][col+1]);  
 col += 1; interchange = 1;  
 } **else** interchange = 0;  
 }**while**(interchange);  
 }  
}

1. Babylonian Algorithm（计算平方根）

**double** baby(**double** guess, **double** p){  
 **double** r = p/guess;  
 **double** next = (guess + r)/2;  
 **if**(fabs(guess-next)<=0.01\*guess){  
 **return** guess;  
 } **else** {  
 **return** baby(next, p);  
 }  
}  
  
**double** baby2(**double** p){  
 **double** guess = p/2;  
 **double** r = p/guess;  
 **double** next = (guess + r)/2;  
 **while**(fabs(guess-next)>0.01\*guess){  
 guess = next;  
 r = p/guess;  
 next = (guess + r)/2;  
 }  
 **return** guess;  
}

1. Euclidean Algorithm（辗转相除法求最大公约数）

算法证明：

假定我们要求k=gcd(a,b)，假设a>b；那么a=mk,b=nk且(m,n)=1；因为m与n互质，所以在m中减去若干个n，得数依然与n互质；所以gcd(a,b) = gcd(a%b,b)

因为a%b<b，为了保证a>b，gcd(a,b) = gcd(b,a%b)

最后会得到a是b的正整数倍即a%b=0，此时返回b即可

如果最开始给的两个数字中a<b，根据规则会自动被转换成gcd(b,a)

**int** gcd(**int** a,**int** b){  
 **if**(a%b==0) **return** b;  
 **else return** gcd(b,a%b);  
}

1. 走到头会回到起点那种（即一个环）——虽然不是算法但好像常见

e.g. 对于数列1,2,3,4：1->2->3->4->1->2…

对于长度为k的数列，给定起始编号为m，向前走n步，最后会停在(m+n)%k处。对于21st Pre-mock test中的Q3，在每一个位置上之后怎么走都是一定的——即不管是第一次到这个位置还是第n次到这个位置，我下一步走到哪里都是一定的；所以能第二次到达的位置有且仅有第0个位置，否则一定是死循环。对于21st Mock test中的Q3，检查循环节时也可以应用这个思想，如果原字符串以abcd为循环节，那么第n个字符应该与该循环节中第n%4+1个相等（C语言以0为起点，就没有+1了）。

1. 子串(sub sequence)匹配（天真版本）

sub-sequence与sub-string不同，并不需要匹配时所有字符都连在一起。

**int** match(**char** tar[], **char** des[]){  
 **char** \*ptr1 = tar, \*ptr2 = des;  
 **while**(\*ptr1 != **'\0'** && \*ptr2 != **'\0'**){  
 **if**(\*ptr1 == \*ptr2)  
 ptr1++;  
 ptr2++;  
 }  
 **if**(\*ptr1 == **'\0'**) **return** 1;  
 **else return** 0;  
}

1. 字符重映射

即给你一个对应关系，按照对应关系替换原字符串中的一些字符。

**void** remap(**char** map[], **char** tar[]){  
 *//接收一个10字符长的map字符作为[0-9]的参考* **char** \*ptr = tar;  
 **while**(\*ptr != **'\0'**){  
 \*ptr = map[\*ptr-**'0'**];  
 ptr++;  
 }  
 **return**;  
}

**int** main() {  
 **char** a[]=**"13348818391"**;  
 remap(**")!@#$%^&\*("**,a);  
 printf(**"%s\n"**,a);  
 **return** 0;  
}

可能你会问，如果题目条件中只替换了一部分字符怎么办？

**void** remap2(**char** numap[], **char** lomap[], **char** upmap[], **char** tar[]){  
 **char** \*ptr = tar;  
 **while**(\*ptr != **'\0'**){  
 **if**(\*ptr>=**'0'** && \*ptr<=**'9'**){  
 \*ptr = numap[\*ptr-**'0'**];  
 } **else if**(\*ptr>=**'a'** && \*ptr<=**'z'**){  
 \*ptr = lomap[\*ptr-**'a'**];  
 } **else**{  
 \*ptr = upmap[\*ptr-**'A'**];  
 }  
 ptr++;  
 }  
 **return**;  
}

**int** main() {  
 **char** a[]=**"13348818391"**;  
 **char** b[]=**"af01t9q8u4hbkfasgdjASLAHJDGF"**;  
 **char** numap[] = **"012#$%^&\*("**;  
 **char** lomap[] = **"ABCDEFGHIJKLMNOPQRSTUVWXYZ"**;  
 **char** upmap[] = **"abcdefghijklmnopqrstuvwxyz"**;  
 remap(**")!@#$%^&\*("**,a);  
 remap2(numap,lomap,upmap,b);  
 printf(**"%s\n"**,b);  
 **return** 0;  
)

1. 回文串

**int** pali1(**char** a[], **int** e){  
 **char** \*p1 = a, \*p2 = &a[e];  
 **while**(p2>p1){  
 **if**(\*p1!= \*p2) **return** 0;  
 **else**{  
 p1++; p2--;  
 }  
 }  
 **return** 1;  
}  
  
**int** pali2(**char** a[], **int** s, **int** e){  
 **if**(s>=e) **return** 1;  
 **else** {  
 **if**(a[s]==a[e]){  
 **return** pali2(a,s+1,e-1);  
 } **else return** 0;  
 }  
}