PRACTICE-11_FUN-PTR

Circular Queue (p1v1d1)

Queues are very widely used in Computer Science (see week 10 lab problems Tuesday problem 1 and Thursday problem 1 - please understand those questions before solving this question). However, when implementing queues using arrays, managing the limited memory available becomes an issue. Suppose we have an array of size 5 and wish to implement a queue using this array.

Suppose the operations given to us are (E = enqueue, D = dequeue, X = terminate)
E 1
E 2
E 3
E 4
E 5
D
D
Initially the array is empty (* denotes an empty location)
[* * * * *]
Then after the five enqueue operations, the array looks like
[1 2 3 4 5]
whereas after the next two dequeue operations, the array looks like
[* * 3 4 5]
the first two positions are no longer a part of the queue. Now if there are two more enqueue operations
E 6
E 7
then we have a problem in our hands. There is no more space to the right of the array (new elements in the queue always go to the back of the queue) so we can do either of the following three things

1. Refuse to enqueue the two new elements: although simple, this is a waste of space since there are two empty locations in the array
2. Shift the array to fill up the empty spaces: shift all entries two locations left so that the array looks like [3 4 5 * *]
   Now there is space to enqueue 6 and 7. However, this shifting business can take up a lot of time (imagine a queue with 3 million entries)
3. Reuse the space to the left using a circular queue: this is what we will implement in this question.

A circular queue is very efficient in reusing space in that it cycles around if the array has no more space to the right. Thus, in a circular queue, E 6 would result in
[6 * 3 4 5]
and then E 7 would result in
[6 7 3 4 5]
If we now see a D instruction, we would pop 3 and the array pop an element from the front of the queue which is the element 3, and the array would look like
[6 7 * 4 5]
If we see 3 more D instructions, elements would keep getting popped from the front of the queue and the array would look like
[6 7 * * 5]
[6 7 * * *]
[* 7 * * *]
If we now see the instruction E 8, the array will get added to the back of the queue
[* 7 8 * *]

If we pop two elements now, the array would become all empty again
[* * * * *]
If we add a new element now E 9, it would get added to the back of the queue which gets reset to the first element of the array
[9 * * * *]
Remember, dequeuing from an empty queue results in an "UNDERFLOW" error and enqueuing to an already full queue results in an "OVERFLOW" error.

You have to implement enqueue and dequeue operations in a circular queue as described above and print the state of the array after every operation (or else print the error message). The first line of the input will tell you the size of the array you should use.

All Test Cases (Visible + Hidden)

Input	Output
5 E 1 E 2 E 3 E 4 E 5 E 6 D D E 7 E 8 E 9 D D D E 0 D D D D X	 [1 * * * *] [1 2 * * *] [1 2 3 * *] [1 2 3 4 *] [1 2 3 4 5] OVERFLOW [* 2 3 4 5] [* * 3 4 5] [7 * 3 4 5] [7 8 3 4 5] OVERFLOW [7 8 * 4 5] [7 8 * * 5] [7 8 * * *] [7 8 0 * *] [* 8 0 * *] [* * 0 * *] [* * * * *] UNDERFLOW
5 E 1 E 2 E 3 D E 4 D E 5 D E 6 D E 7 D E 8 D E 9 D E 0 D X	 [1 * * * *] [1 2 * * *] [1 2 3 * *] [* 2 3 * *] [* 2 3 4 *] [* * 3 4 *] [* * 3 4 5] [* * * 4 5] [6 * * 4 5] [6 * * * 5] [6 7 * * 5] [6 7 * * *] [6 7 8 * *] [* 7 8 * *] [* 7 8 9 *] [* * 8 9 *] [* * 8 9 0] [* * * 9 0]

4	[0 * * *]
E 0	[0 0 * *]
E 0	[0 0 0 *]
E 0	[0 0 0 0]
E 0	[* 0 0 0]
D	[* * 0 0]
D	[* * * 0]
D	[* * * *]
D	UNDERFLOW
D	[1 * * *]
E 1	[1 1 * *]
E 1	[1 1 1 *]
E 1	[1 1 1 1]
E 1	OVERFLOW
E 1	[* 1 1 1]
D	[2 1 1 1]
E 2	OVERFLOW
E 2	OVERFLOW
E 2	[2 * 1 1]
D	[2 * * 1]
D	[2 * * *]
D	[* * * *]
D	
X	

Primes are here again (p1v2d1)

WARNING: This problem is otherwise very simple. The point of this question is to practice writing functions to perform simple operations to write nice code. Use the template provided and practice writing functions.

You will be given a positive number n and then n positive integers. Store these numbers in an array arr. As your output, you have to print n numbers where ith number is arr[i-1]*(sum of all primes strictly less than arr[i-1]) where i runs from 1 to n. For this problem, 0 and 1 are not considered primes.

WARNING: use long variables to perform computations even though the input will only be integer variables. The template is given below as well in case you erase it.

Input Format:
n (integer denoting the size of the array)
n space separated positive integer

Output Format:
n space separated integers

Example:
Input:
1
2

Output:
0

#include <stdio.h>

```
int check_prime(int n){
if(n < 2) return 0;
for(int i = 2; i < n; i++)
if(n % i == 0)
return 0;
return 1;
}

long prime_sum(int n){
long sum = 0;
for(int i = 2; i < n; i++)
if(check_prime(i))
sum += i;
return sum;
}

int main(){
int n, i;
scanf("%d", &n);
int arr[n];

for(i = 0; i < n; i++){
scanf("%d", arr + i);
printf("%ld", arr[i]*prime_sum(arr[i]));
if(i < n-1) printf(" "); // No trailing spaces
}

return 0;
}
```

All Test Cases (Visible + Hidden)

Input	Output
1 2	0
2 2 3	0 6
20 41 59 66 2 80 12 52 30 33 16 89 61 10 31 78 67 93 88 40 74	8077 22479 33066 0 63280 336 17056 3870 5280 656 77786 26840 170 3999 55536 33567 89559 76912 7880 52688
50 5 22 12 59 31 62 68 11 30 63 39 30 53 6 71 30 97 12 99 61 39 80 81 18 71 45 54 61 94 52 46 10 49 82 51 42 22 33 78 6 55 72 22 60 57 66 89 80 6 73	25 1694 336 22479 3999 31062 38624 187 3870 31563 7683 3870 17384 60 40328 3870 93411 336 104940 26840 7683 63280 64071 1044 40328 12645 20574 26840 90522 17056 12926 170 16072 64862 16728 9996 1694 5280 55536 60 20955 46008 1694 26400 21717 33066 77786 63280 60 46647
50 5944 5281 4060 8065 8822 9209 7644 4020 5630 1983 9183 9051 5233 5560 6362 6535 3615 3490 2872 7461 6553 7760 1690 7418 5687 9924 4945 8842 6761 1022 7153 5558 4361 4748 5262 733 3465 1552 6549 1634 844	12820096472 9016193209 4278245300 30540171010 39817955848 45052472398 26367465852 4154871000 10808620380 533573742 44671924839 42955901184 8796704398 10487972560 15647784340 16704682045 3046252050 2717914280 1587506616 24327366444 16836524605 27664881120 343570240 23966104072 11207513075 56335630344 7458741300 40142600422 18404577848 81953158 21756264374 10484199908 5238267482 6655005712 8928230094 31129044 2674411740 268025744 16783344966 318634902 47126428

3126 6939 2491 2051 2142 2880 3238 6985 4038	1999802232 19885571091 1048242692 593099976 682359804 1600220160 2195383428 20406558755 4205972724
5 2 10 1 1 5	0 170 0 0 25

The Clones of the Clones (p1v3d1)

We have seen that C functions can call each other. However, C functions can call themselves as well. The process of a function calling itself is called recursion and it can be used to write very elegant solutions to complex-looking problems.

The template code given with this problem gives you the recursive code to print the locations of all occurrences of a given character in a string. Modify this code to print all occurrences of a given substring in a string. Both the string, as well as the substring will be at most 99 characters long and will be given in a single line. We will first give the substring then the string in two separate lines.

WARNING: programs that use recursion excessively can be slow due to the cloning process being a bit expensive. All C programs that are written using recursion, can also be written in a way that does not use any recursion. This result is a landmark and fundamental result in the theory of computation (CSE students will study this topic in the course CS340), guaranteed by the Church-Turing thesis. However, the non-recursive code can sometimes look very messy to humans whereas the recursive code looked very elegant, even though the messy code is faster than the elegant code.

The template code is reproduced below in case you erase it while coding.

```
---------- TEMPLATE CODE ----------
#include <stdio.h>
#include <string.h>

// whereIsChar recursively calls itself to find out
// 1. the locations of all occurrences of c
// 2. total number of occurrences of c
// offset tells us how many chars of the string have we already searched
int whereIsChar(char* string, char c, int offset){
char* ptr = strchr(string, c); // Find next occurrence of c
if(ptr != NULL){ // Did we find c ?
int foundIndex = offset + (int)(ptr - string);
printf("Found at index %d\n", foundIndex);
// Continue searching for c in the remaining string
return 1 + whereIsChar(ptr + 1, c, foundIndex + 1);
}else
return 0; // Nothing more to do - no more occurences of c
}

int main(){
char c, str[100];
scanf("%c\n", &c);
gets(str);

// Offset is 0 since none of the string has been searched so far
printf("Found %d occurrence(s) of char %c", whereIsChar(str,c,0), c);

return 0;
}
```

All Test Cases (Visible + Hidden)

Input	Output
abc abc	Found at index 0 Found 1 occurrence(s) of substring "abc"
abc defghi	Found 0 occurrence(s) of substring "abc"
abc abcdefabcdefabcdef	Found at index 0 Found at index 6 Found at index 12 Found 3 occurrence(s) of substring "abc"
aba abacbabcbabababa	Found at index 0 Found at index 9 Found at index 11 Found at index 13 Found 4 occurrence(s) of substring "aba"
aaa aacaaaabaaaaaba	Found at index 3 Found at index 4 Found at index 8 Found at index 9 Found at index 10 Found 5 occurrence(s) of substring "aaa"