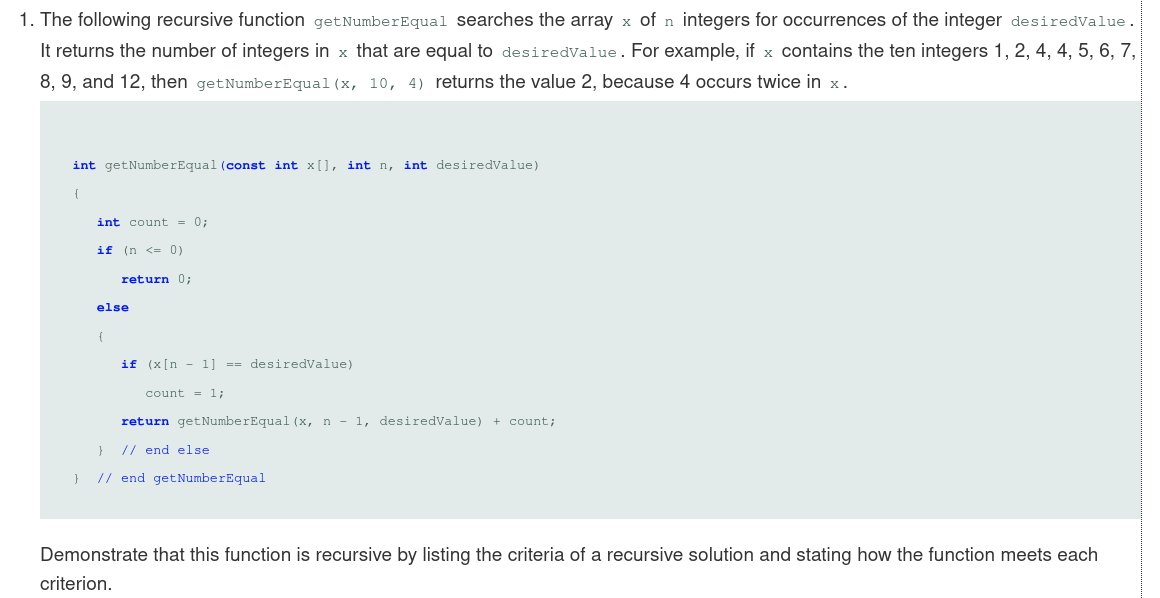
CPSC 250

Homework #1

Chapter 2 Recursion & Chapter 10 Algorithm EfficiencyDue Date: Sunday, Sep. 26, 2021(No Late Submissions will be Accepted.) Grades depend on neatness and clarity. Write your answers with enough detail about your approach and concepts used, so that the grader will be able to understand it easily. Submissions should be one file containing all answers.

Chapter 2 Recursion1.(10 points each for each exercise question) Textbook p 89 -92, Exercises #1, 2, 6, 10, 11, 12, 13, 14, 15, and 16.



Criteria for a recursive function:

**Is the problem defined in terms of a smaller problem of the same type?**

**Does each call diminish the size of the problem?**

**Is there an instance that serves as a base case?**

**When the problem size diminishes is the base case reached?**

Answers:

**Is the problem defined in terms of a smaller problem of the same type?**

You solve the getNumberEqual problem by calling the same problem with a smaller getNumberEqual problems, or at least with a relatively smaller n value.

The array’s last value is checked, then the function passes the parts left in the array, are passed to the function, until no parts are left.

**Does each call diminish the size of the problem?**

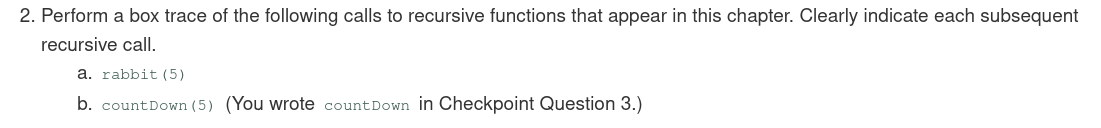
The getNumberEqual function search's for a value and counts the total number of index’s holding a particular value, by searching the last element of increasingly smaller arrays. Every time the function is called n decreases by one with each recursive call. This reduces the size of the unsearched array by 1.

**Is there an instance that serves as a base case?**

When n is less than or equal to zero the base case executes, so there is a base case that solves the solution directly. This terminates the recursion.

**When the problem size diminishes is the base case reached?**

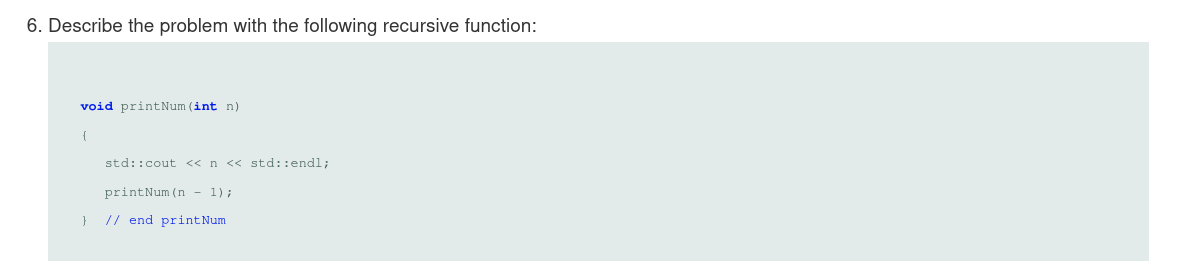
Because the base case is less than or equal to zero, it ensures n will eventually get to a small enough size to exit, and it will eventually reach the base case. After the completion if “n” recursive calls, “n” parameter in the nth call contains the “0” value and the base case is always to be reached



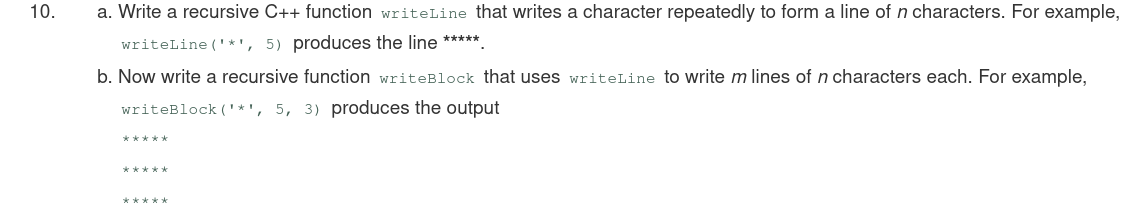
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| n = 5  rabbit(4) = ?  rabbit(3) = ?  return =? | call rabbit 4 |  |  |  |  |
| n = 5  rabbit(4) = ?  rabbit(3) = ?  return =? | n = 4  rabbit(3) = ?  rabbit(2) = ?  return =? | follow call rabbit(3) |  |  |  |
| n = 5  rabbit(4) = ?  rabbit(3) = ?  return =? | n = 4  rabbit(3) = ?  rabbit(2) = ?  return =? | n = 3  rabbit(2) = ?  rabbit(1) = ?  return =? | rabbit(2)  base case n=2 |  |  |
| n = 5  rabbit(4) = ?  rabbit(3) = ?  return =? | n = 4  rabbit(3) = ?  rabbit(2) = ?  return =? | n = 3  rabbit(2) = ?  rabbit(1) = ?  return =? | n=2  return 1 | end rabbit(2) |  |
| n = 5  rabbit(4) = ?  rabbit(3) = ?  return =? | n = 4  rabbit(3) = ?  rabbit(2) = ?  return =? | n = 3  rabbit(2) = ?  rabbit(1) = ?  return =? | n=2  return 1 |  |  |
| n = 5  rabbit(4) = ?  rabbit(3) = ?  return =? | n = 4  rabbit(3) = ?  rabbit(2) = ?  return =? | n = 3  rabbit(2) = 1  rabbit(1) = ?  return =? | call rabbit(1)  base case n=1 |  |  |
| n = 5  rabbit(4) = ?rabbit(3) = ?  return =? | n = 4  rabbit(3) = ?  rabbit(2) = ?  return =? | n = 3  rabbit(2) = ?  rabbit(1) = ?  return =? | n=1  return 1 | end rabbit(1) |  |
| n = 5  rabbit(4) = ?  rabbit(3) = ?  return =? | n = 4  rabbit(3) = ?  rabbit(2) = ?  return =? | n = 3  rabbit(2) = ?  rabbit(1) = ?  return =? | n=1  return 1 |  |  |
| n = 5  rabbit(4) = ?  rabbit(3) = ?  return =? | n = 4  rabbit(3) = ?  rabbit(2) = ?  return =? | n = 3  rabbit(2) = 1  rabbit(1) = 1  return = 2 | n=1  return 1 |  |  |
| n = 5  rabbit(4) = ?  rabbit(3) = ?  return =? | n = 4  rabbit(3) = 2  rabbit(2) = ?  return =? | rabbit(2)  base case n=2 |  |  |  |
| n = 5  rabbit(4) = ?  rabbit(3) = ?  return =? | n = 4  rabbit(3) = 2  rabbit(2) = ?  return =? | n = 2  return 1 | end rabbit(2) |  |  |
| n = 5  rabbit(4) = ?  rabbit(3) = ?  return =? | n = 4  rabbit(3) = 2  rabbit(2) = 1  return =3 | n = 2  return 1 | end rabbit(4) |  |  |
| n = 5  rabbit(4) = ?  rabbit(3) = ?  return =? | n = 4  rabbit(3) = 2  rabbit(2) = 1  return =3 | n = 2  return 1 |  |  |  |
| n = 5  rabbit(4) = 3  rabbit(3) = ?  return =? | rabbit(3) |  |  |  |  |
| n = 5  rabbit(4) = 3  rabbit(3) = ?  return =? | n = 3  rabbit(2) = ?  rabbit(1) = ?  return = ? | rabbit(2) |  |  |  |
| n = 5  rabbit(4) = 3  rabbit(3) = ?  return =? | n = 3  rabbit(2) = ?  rabbit(1) = ?  return = ? | n = 2  return 1 | end rabbit(2) |  |  |
| n = 5  rabbit(4) = 3  rabbit(3) = ?  return =? | n = 3  rabbit(2) = ?  rabbit(1) = ?  return = ? | n =2  return 1 |  |  |  |
| n = 5  rabbit(4) = 3  rabbit(3) = ?  return =? | n = 3  rabbit(2) = 1  rabbit(1) = ?  return = ? | rabbit(1)  n = 1 |  |  |  |
| n = 5rabbit(4) = 3  rabbit(3) = ?  return =? | n = 3  rabbit(2) = 1  rabbit(1) = ?  return = ? | n = 1  return 1 | end rabbit(1) |  |  |
| n = 5  rabbit(4) = 3  rabbit(3) = ?  return =? | n = 3  rabbit(2) = 1  rabbit(1) = 1  return = 2 | n = 1  return 1 |  |  |  |
| n = 5  rabbit(4) = 3  rabbit(3) = 2  return =5 | n = 3  rabbit(2) = 1  rabbit(1) = 1  return = 2 | n = 1  return 1 | end rabbit(3) |  |  |
| n = 5  rabbit(4) = 3  rabbit(3) = 2  return =5 | n = 3  rabbit(2) = 1  rabbit(1) = 1  return = 2 | n = 1  return 1 | end rabbit(5) |  |  |

countDown(5)

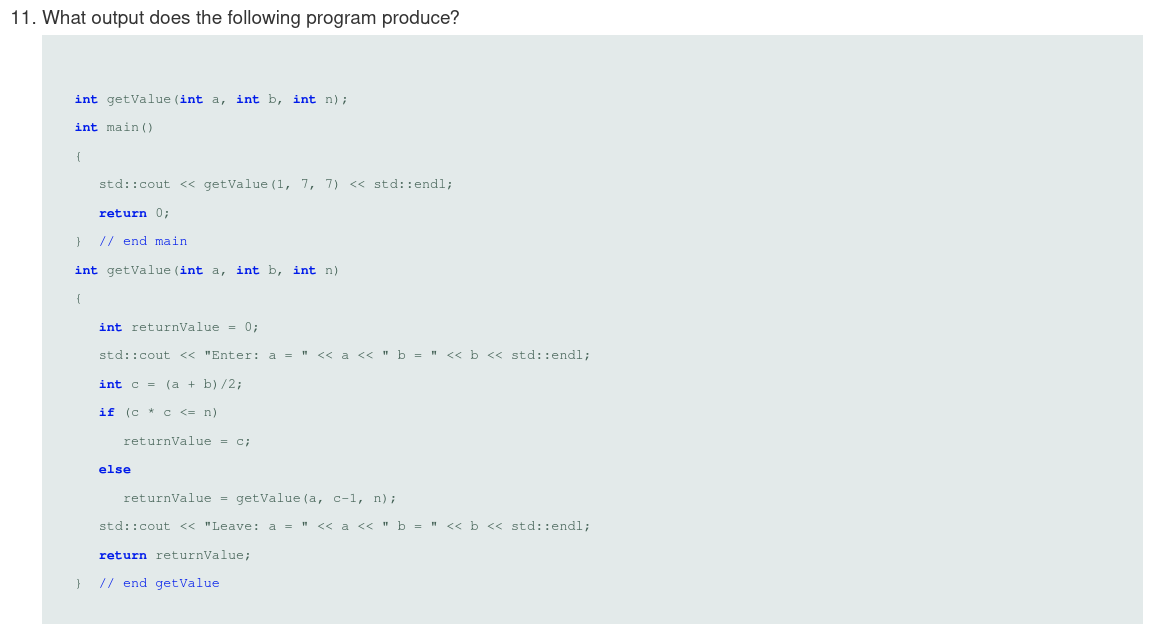
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| n = 5  cout << 5 | 5 printed |  |  |  |  |  |
| n = 5  cout << 5 | n = 4  cout << 4 | 4 printed |  |  |  |  |
| n = 5  cout << 5 | n = 4  cout << 4 | n = 3  cout << 3 | 3 printed |  |  |  |
| n = 5  cout << 5 | n = 4  cout << 4 | n = 3  cout << 3 | n = 2  cout << 2 | 2 printed |  |  |
| n = 5  cout << 5 | n = 4  cout << 4 | n = 3  cout << 3 | n = 2  cout << 2 | n = 1  cout << 1 | 1 printed |  |
| n = 5  cout << 5 | n = 4  cout << 4 | n = 3  cout << 3 | n = 2  cout << 2 | n = 1  cout << 1 | n = 0  cout << endl (possibly) | endl printed at  end countDown(0) |
| n = 5  cout << 5 | n = 4  cout << 4 | n = 3  cout << 3 | n = 2  cout << 2 | n = 1  cout << 1 | n = 0  cout << endl (possibly) | end countDown(1) |
| n = 5  cout << 5 | n = 4  cout << 4 | n = 3  cout << 3 | n = 2  cout << 2 | n = 1  cout << 1 | n = 0  cout << endl (possibly) | end  coutDown(2) |
| n = 5  cout << 5 | n = 4  cout << 4 | n = 3  cout << 3 | n = 2  cout << 2 | n = 1  cout << 1 | n = 0  cout << endl (possibly) | end  coutDown(3) |
| n = 5  cout << 5 | n = 4  cout << 4 | n = 3  cout << 3 | n = 2  cout << 2 | n = 1  cout << 1 | n = 0  cout << endl (possibly) | end  coutDown(4) |
| n = 5  cout << 5 | n = 4  cout << 4 | n = 3  cout << 3 | n = 2  cout << 2 | n = 1  cout << 1 | n = 0  cout << endl (possibly) | end  coutDown(5) |



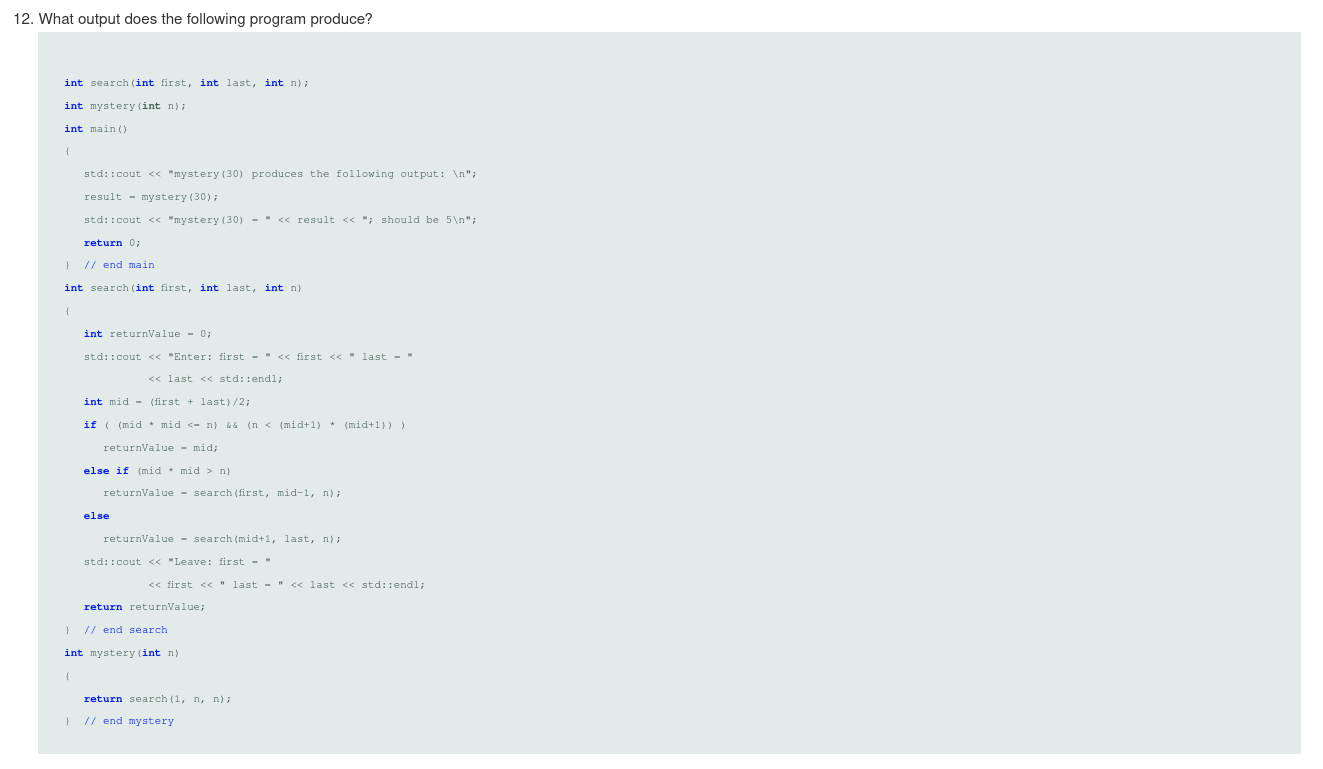
The problem with printNum is that there is no base case, there is no condition where the function exits other than the limit of the computer.



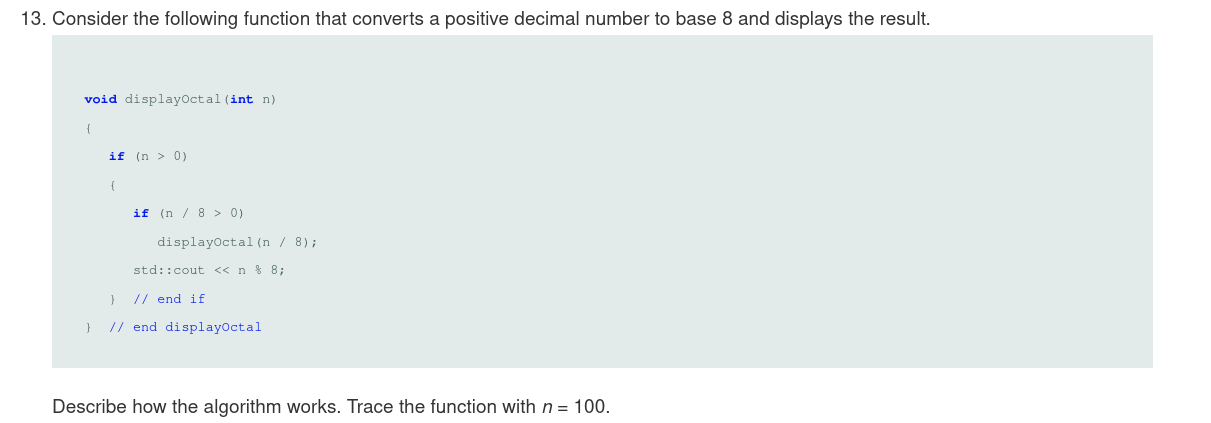
writeLine(char star, int n)



In the “getValue()” function, first parameters a and b are added together and dived by two which is then assigned to c, then c^2 is compared to the third and final parameter n. If n^2 is equal to n, base case is trigged, and the value of “c” is returned. Else, “getValue()” will print the values in “a” and “b”, until c^2 is less than equal to n.

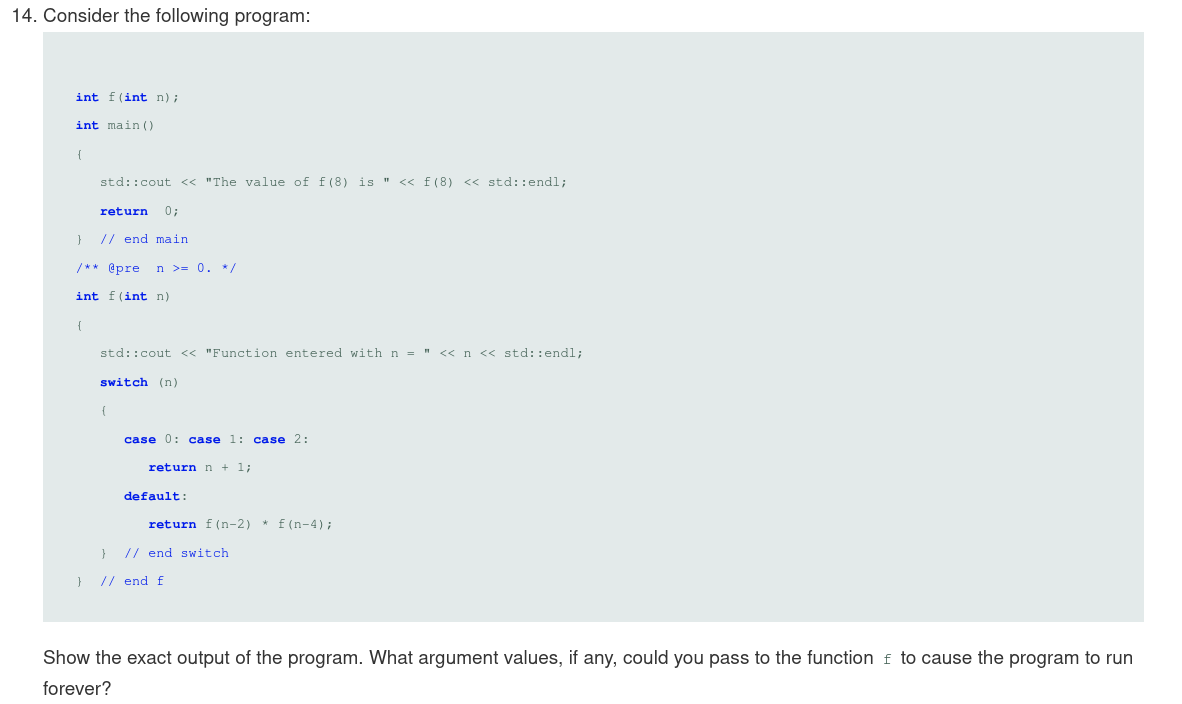


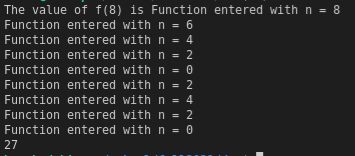
This is 2 functions, search() and mystery(), with 3 and 1 parameters, respectively. In search, mid, is calculated, by (first+last)/2, then mid^2 is compared to n, and (mid+1)^2 is compared to n. If the if is false, search will be recursively called through the else if, or the else until it is true, then the if returns mid. The mystery function is used to call the search() function and defines searches parameters. It is only involved to define the parameters for, and originally call the search function.

output:

144

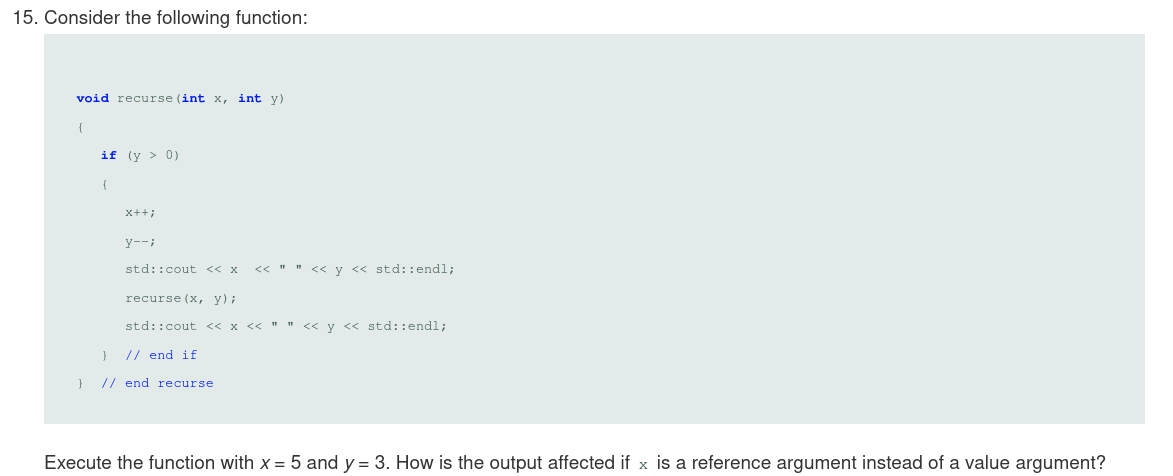
In the display octal function, the first case checks if n is greater than zero, if false the function immediately terminates. If n is greater then 0, the function then checks if n/8 is greater than zero, checking if n is greater than 8. If all returns true, displayOctal() is recursively called until n is less then 8, then it starts printing the numbers out in the reverse order that they where computed. 100/8, 12/8, 1, 1%8=1, 12%8=4, 100%8=4, 144.

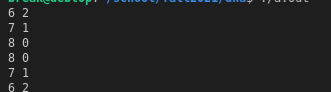




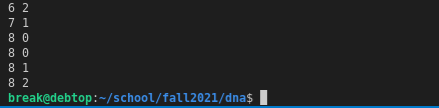
If the parameter n, is set to an odd integer then the default case of f(n-2) \* f(n-4) will result in a situation where the integers being passed to f(n-2) \* f(n-4) will just decrease forever. So all 1 < !(n%2=0), or the

If the parameter n, is set to an odd integer then the default case of f(n-2) \* f(n-4) will result in a situation where the integers being passed to f(n-2) \* f(n-4) will just decrease forever. So all 1 < (2n+1), or the function will continue forever under any odd number greater then 1, since the cases 0, 1, and 2, are handled.

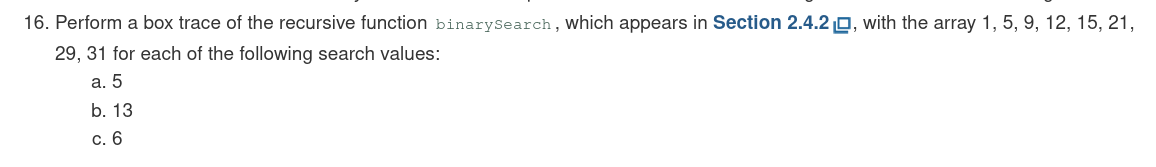
value:



reference:



The difference comes about because when you pass by reference you are using an x value at a single location in memory, and when you pass by value, each recursive call to the function creates its own version of x, allowing the changes in the return values between the two functions,



a.

binarySearch(5)

|  |  |
| --- | --- |
| value = 5  first = 1  last = 8  mid = 4  value < anArray [4] | value = 5  first = 1  last = 3  mid = 2  value = anArray [2]  return 2 |

b.

binarySearch(13)

|  |  |  |  |
| --- | --- | --- | --- |
| value = 13  first = 1  last = 8  mid = 4  value < anArray [4] | value = 13  first = 1  last = 8  mid = 6  value < anArray [6] | value = 13  first = 5  last =5  mid = 5  value < anArray [5] | value = 13  first = 5  last = 4  first > last  return 0 |

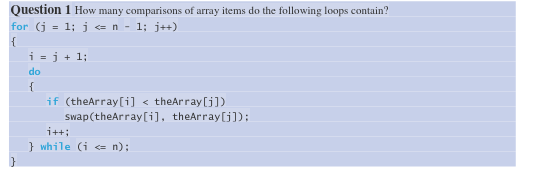
since 13 is not found

c.

binarySearch(6)

|  |  |  |  |
| --- | --- | --- | --- |
| value = 6  first = 1  last = 8  mid = 4  value < anArray [4] | value = 6  first = 1  last = 3  mid = 2  value < anArray [2] | value = 6  first = 3  last = 2  first > last  return 0 |  |

Chapter 10 Algorithm Efficiency



There are two loops, an outer, and an inner loop

for:

(n-1)

(n-1) = n+n

do..while:

(n-2)

(n-2)

total comparison is (n(n-1))/2

or O(n^2)





1. 8\*n^3-9\*n

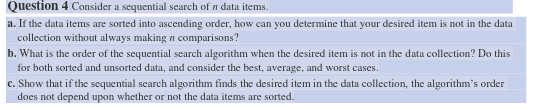
= O(n^3)

1. 7\*(log\_2)(n)+20

= O(log\_2(n))

7\*(log\_2)(n))+(n)

=O(n)



You often won’t make n comparisons with a sequential search. This is only when the ascending list is searched for the largest element, or an element that doesn’t exist. In all other cases sequential searches will end in less than n comparisons. If you are searching for an element, say ten, and the list is in ascending order, if you encounter an 11, you know 10 is not in the list if you haven’t found it yet. So, you can return without searching the whole list.

b.

When item is not present = O(n)

**sorted:**

best = O(1)

average = O(n)

worst = O(n)

**unsorted:**

best = O(1)

average = O(n)

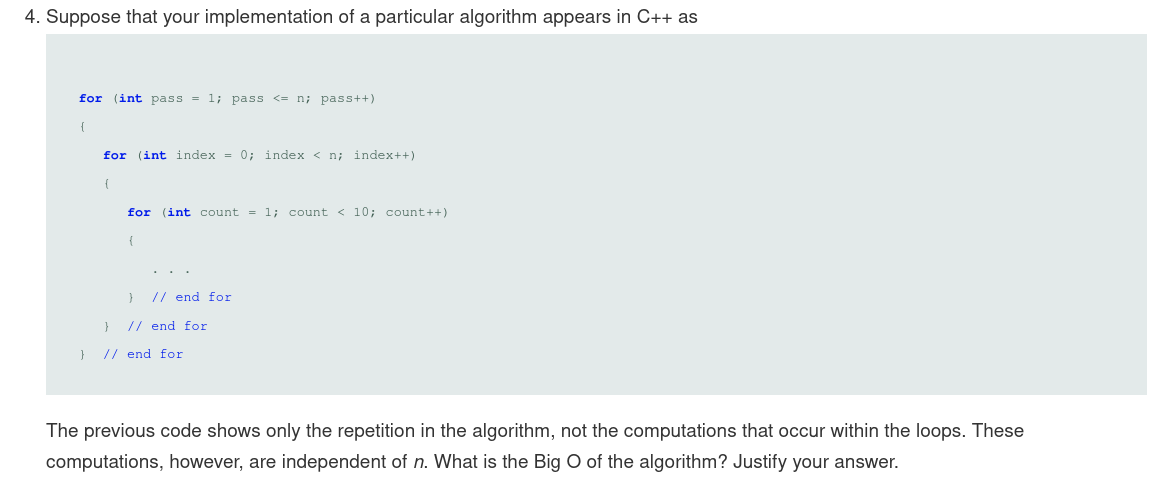
worst = O(n)

c.

There are several reasons why the order of the sequential search algorithm does not depend on being sorted or unsorted.

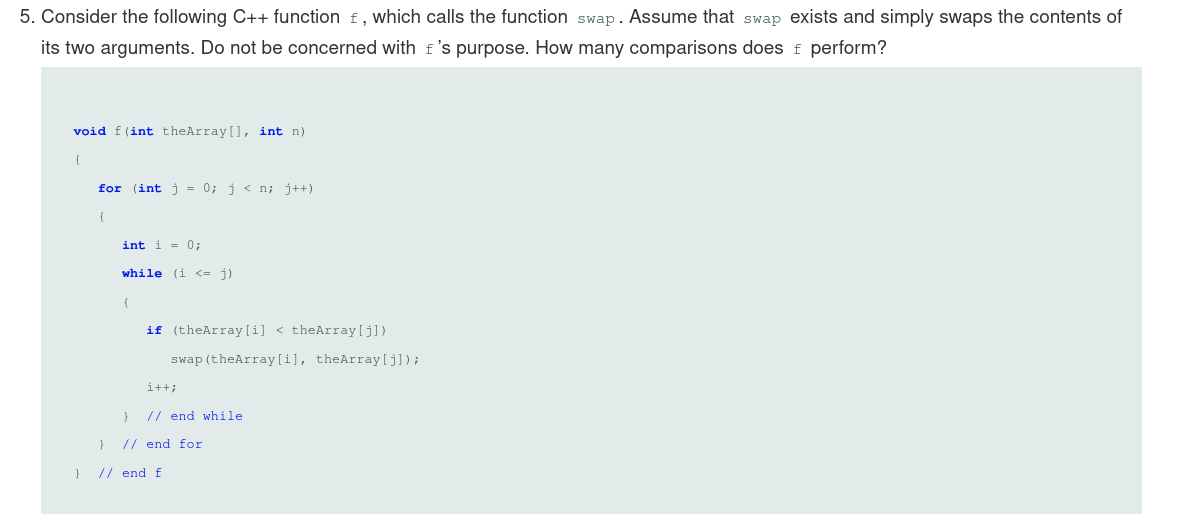
If the item is found the rest of the list or array does not need to be traversed. If the list is sorted or unsorted, if the element you are searching for is random, there is an equally likely chance, for unsorted that an element can be anywhere in the list/array. So if your choice is random it does not matter if the list/array is sorted or unsorted.

Textbook p 321 -322, Exercises #4, 5, and 6

The loop bigO’s from inner to outer loop are as follows:

O(10)\*O(n)\*O(n)

= O(n^2)

the outer loop = O(n)

the inner loop = O(n\*(n+1))/2) or O(n^2)

Therefore, f is = O(n^2)



Since a binary search is O (log\_2 n) in the worst case, and a sequential search is O(n) in the worst case, binary search will always work better in the case of large, sorted arrays. So, the answer is no sequential search is not faster than binary search in the worst case. The growth rate of binary search is much smaller, in terms of comparisons, than sequential search is, and this becomes only more evident as the size of the array increases.