Assignment 2

Asymptotic analysis and Divide and Conquer

NOHAN JOEMON CH19B072

==

Qn 1:

Everywhere I assume log is to the base 2

To compare asymptotic growth rate, we use big - O notation which is defined as follows (T(n) is the worst case runtime)
T(n) = O(f(n)) ⇔ ∃ c , n_0 > 0 s.t. ∀ n ≥n_0 , T(n) ≤ c · f(n)
ie,
T(n) scales like f(n)

Main features in asymptotic notation:

1: suppressing constant factors:
The constant factors are system dependent. But in asymptotic analysis, we want a form which depends only on the implementation of the algorithm and not on where the algorithm is implemented. Therefore, constant factors should not be considered in asymptotic analysis

2: suppress lower-order terms:
In asymptotic analysis, we consider high input sizes(which may even be greater than actual input sizes). Therefore, lower order terms can be neglected.

Using the above principles, we can represent each function in big O notation

1: $4nlogn + 2n$ **= O(nlogn)** [neglecting lower order term:2n and constants 2 and 4]

2: 2^{10} **= O(1)** [neglecting constant 2^{10}]

3: 2^{logn} **= n = O(n)** [no constants or lower order terms to neglect] - ASSUMPTION: log base=2

4: $3n + 100logn$ **= O(n)** [neglecting lower order term:100logn and constants 3 and 100]

5: $4n$ **= O(n)** [neglecting constant 4]

6: 2^n **= O(2^n)** [no constants or lower order terms to neglect]

7: $n^2 + 10n$ **= O(n^2)** [neglecting lower order term:10n]

8: n^3 **= O(n^3)** [no constants or lower order terms to neglect]

9: $nlog(n)$ **= O(n logn)** [no constants or lower order terms to neglect]

Comparing bigO representations, we have **O(1) < O(n) < O(nlog(n)) < O(n^2) < O(n^3) < O(2^n)**
Therefore, order of asymptotic growth rate is:
(2^{10}) < (2^{logn}) = ($3n + 100logn$) = ($4n$) < ($4nlogn + 2n$) = ($nlog(n)$) < ($n^2 + 10n$) < (n^3) < (2^n)

==

Qn 2:

Everywhere I assume log is to the base 2

a) f(n) = n−100 and g(n) = n−200 : f = Θ(g)

It is f = Θ(g): For that we need to prove f = O(g) and f = Ω(g)
To prove f = Ω(g):

We have to prove that f(n) > = cg(n) for all n>=n0 , where c and n0 are positive constants.

Assume c = 0.25 and n0 = 67.

Clearly, n-100 > = 0.25(n-200) for all n>=67

Thus, from the definition of Big- Ω , f=Ω(g) .
To prove f = O(g):

We have to prove that f(n) < = cg(n) for all n>=n0, where c and n0 are positive constants.

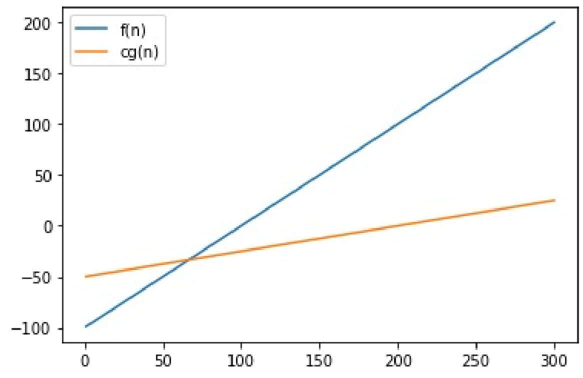
Assume c=2 and n0=300.

Clearly, n-100 < = 2(n-200) for all n>=300

Thus, from the definition of Big-O, f=O(g)

verification using graphs

In [47]:
```
# for proving f = Ω(g) ie f(n) >= c.g(n)
import numpy as np
import matplotlib.pyplot as plt

n = np.linspace(1, 300, 100)
plt.plot(n, n-100, label = "f(n)")
plt.plot(n, 0.25*(n-200), label = "cg(n)") # c=0.25
plt.legend()
plt.show
# Clearly, from the graph,  f(n) >= c.g(n)
```
Out[47]:
```
<function matplotlib.pyplot.show(*args, **kw)>
```


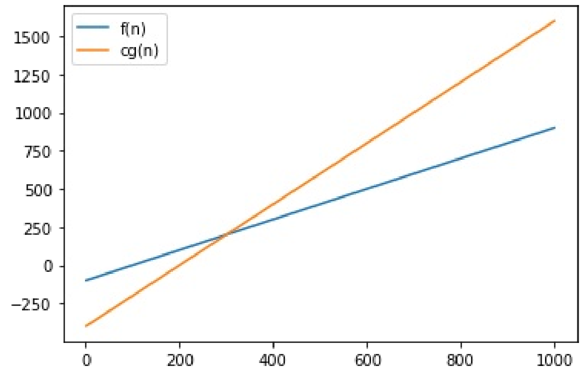
In [20]:

```
# for proving f = O(g) ie f(n) <= c.g(n)
import numpy as np
import matplotlib.pyplot as plt

n = np.linspace(1, 1000, 100)
plt.plot(n, n-100, label = "f(n)")
plt.plot(n, 2*(n-200), label = "cg(n)") #c=2
plt.legend()
plt.show
# Clearly, from the graph, f(n) <= c.g(n)
```

Out[20]:

```
<function matplotlib.pyplot.show(*args, **kw)>
```



b) f(n) = 100n + logn and g(n) = $n + (logn)^2$: f = Θ(g)

It is f = Θ(g): For that we need to prove f = O(g) and f = Ω(g)
To prove f=Ω(g):

We have to prove that f(n)>=cg(n) for all n>=n0 , where c and n0 are positive constants.

Assume c=1 and n0=1

Clearly, 100n + logn > = ($n + (logn)^2$) for all n>=1 (The reason is not very straightforward, therefore, it is better to use graph to check it)

Thus, from the definition of Big- Ω , f=Ω(g) .
To prove f = O(g):

We have to prove that f(n) < = cg(n) for all n>=n0, where c and n0 are positive constants.

Assume c=100 and n0=2.

Clearly, 100n + logn < = 100($n + (logn)^2$) for all n>=2 (Because 100$(logn)^2$ > logn for all logn>1 ie n>2)

Thus, from the definition of Big-O, f=O(g)

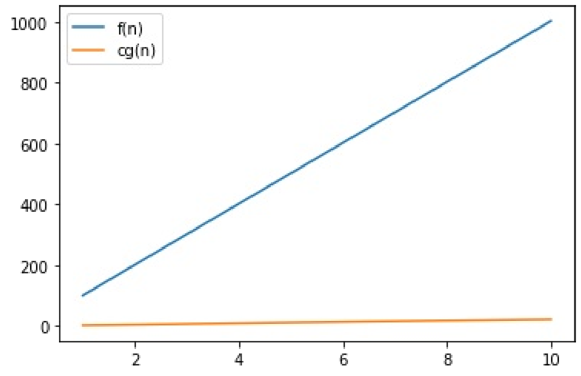
verification using graphs

In [49]:

```
# for proving f = Ω(g) ie f(n) >= c.g(n)
n = np.linspace(1, 10,10)
f = 100*n + np.log2(n)
g = (n + np.log2(n)*np.log2(n))
plt.plot(n, f, label = "f(n)")
plt.plot(n, g, label = "cg(n)") # c = 1
plt.legend()
plt.show
# Clearly, from the graph,  f(n) >= c.g(n)
```

Out[49]:

```
<function matplotlib.pyplot.show(*args, **kw)>
```

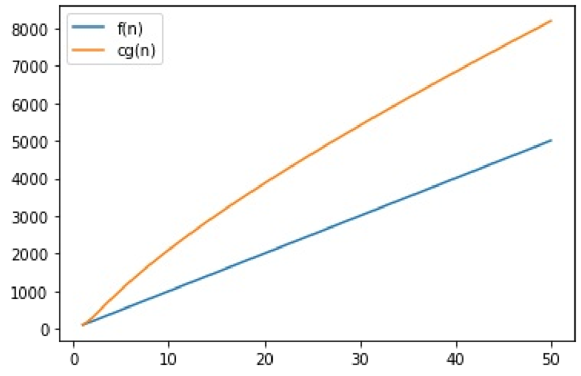


In [21]:

```
# for proving f = O(g) ie f(n) <= c.g(n)
n = np.linspace(1, 50, 100)
f = 100*n + np.log2(n)
g = 100*(n + np.log2(n)*np.log2(n)) # c = 100
plt.plot(n, f, label = "f(n)")
plt.plot(n, g, label = "cg(n)")
plt.legend()
plt.show
# Clearly, from the graph, f(n) <= c.g(n)
```

Out[21]:

```
<function matplotlib.pyplot.show(*args, **kw)>
```



c) f(n)=log(2n) and g(n)=log(3n) : f = Θ(g)

It is f = Θ(g): For that we need to prove f = O(g) and f = Ω(g)
To prove f=Ω(g):

We have to prove that f(n)>=cg(n) for all n>=n0 , where c and n0 are positive constants.

Assume c=0.5 and n0=1

Clearly, log(2n) > = 0.5log(3n) for all n>=1 (Because 2log(2n)=log($4n^2$)>log(3n) for all n>=1)

Thus, from the definition of Big- Ω , f=Ω(g) .
To prove f = O(g):

We have to prove that f(n) < = cg(n) for all n>=n0, where c and n0 are positive constants.

Assume c=1 and n0=1.

Clearly, log(2n) < = log(3n) for all n>=1

Thus, from the definition of Big-O, f = O(g)

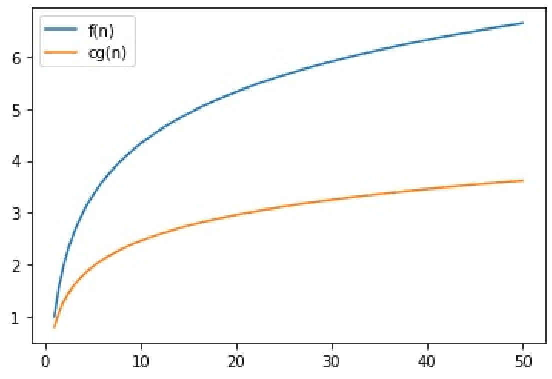
verification using graphs

In [51]:

```
# for proving f = Ω(g) ie f(n) >= c.g(n)
n = np.linspace(1, 50, 100)
f = np.log2(2*n)
g = 0.5*np.log2(3*n) # c = 0.5
plt.plot(n, f, label = "f(n)")
plt.plot(n, g, label = "cg(n)")
plt.legend()
plt.show
# Clearly, from the graph, f(n) >= c.g(n)
```

Out[51]:

```
<function matplotlib.pyplot.show(*args, **kw)>
```

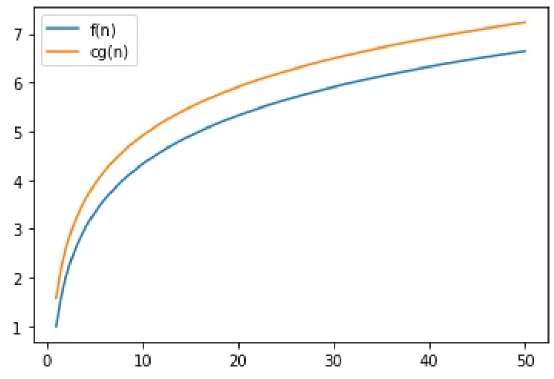


In [22]:

```
# for proving f = O(g) ie f(n) <= c.g(n)
n = np.linspace(1, 50, 100)
f = np.log2(2*n)
g = np.log2(3*n) # c = 1
plt.plot(n, f, label = "f(n)")
plt.plot(n, g, label = "cg(n)")
plt.legend()
plt.show
# Clearly, from the graph, f(n) <= c.g(n)
```

Out[22]:

```
<function matplotlib.pyplot.show(*args, **kw)>
```



d) f(n)=$n^{1.01}$) and g(n)=$n(log(n))^2$: f = Ω(g)

It is f = Ω(g). For that we need to prove f ≠ O(g) and f = Ω(g)
To prove f = Ω(g):

We have to prove that f(n)>=cg(n) for all n>=n0 , where c and n0 are positive constants.

Assume c=0.00005 and n0=1

Clearly, $n^{1.01}$ > = 0.00005$n(log(n))^2$ for all n>=1 (The reason is not very straightforward, therefore, it is better to use graph to check it)

Thus, from the definition of Big- Ω , f=Ω(g) .

To prove f ≠ O(g): We have to prove that there are no possible values of positive constants c and n0 such that f(n) < = cg(n) for all n>=n0

Let us assume that there exists positive constants c and n0 such that f(n) < = cg(n) for all n>=n0

Then,

$$cn(logn)^2 - n^{1.01} >= 0$$

$$c >= \frac{n^{0.01}}{(logn)^2}$$

That is, $k(n) = \frac{n^{0.01}}{(logn)^2}$ has c as its upper bound

However,$\lim_{n \to +\infty} k(n) = \infty$, and therefore, upper bound of k(n) is not c

Hence, our assumption is wrong

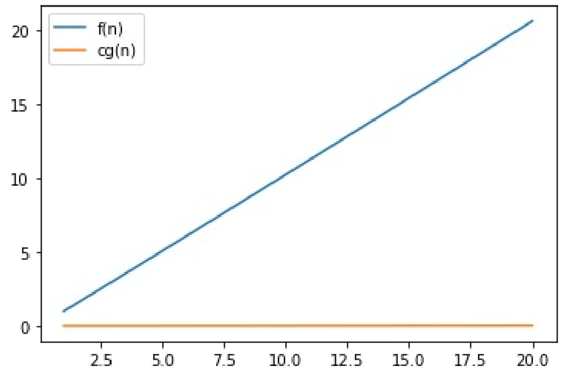
Hence, f ≠ O(g)

In [59]:

```
# for proving f = Ω(g) ie f(n) >= c.g(n)
n = np.linspace(1, 20, 100)
f = np.power(n, 1.01)
g = (0.00005)*(n*np.log2(n)*np.log2(n)) # c = 0.00005
plt.plot(n, f, label = "f(n)")
plt.plot(n, g, label = "cg(n)")
plt.legend()
plt.show
# Clearly, from the graph,f(n) >= c.g(n)
# There is no graphical way to verify that f ≠ O(g), but the mathematical proof itself is strong
```

Out[59]:

```
<function matplotlib.pyplot.show(*args, **kw)>
```



==

Qn 3:

To find 10 largest elements in a sequence of size n, we can sort it using mergesort and then select first 10 elements. But, this would have time complexity O(nlogn).

So, instead we can just do a linear search which is of time complexity O(n). Hence, I am using an algorithm that does a linear search 10 times.

PSEUDO CODE:

(We have an array "a" of n numbers. We need to return an array "maxi" of 10 largest numbers of "a").
1: Loop 10 times through the array "a" with each iteration identified by the value of i (ranging from 0 to 9)
2: For each iteration i,

```
[

1: Assume the first element of "a" as the largest element and store its value in maxi[i]

2: Loop through each item of the array "a"(each item identified by the variable "item"):

    [

    1: if item>maxi[i],

        then maxi[i]=item

    2: go to next item

    ]

3: Now, delete the element obtained as maxi[i] from the array "a"

]
```

3: After 10 iterations, we get maxi as the required list of 10 largest numbers

Note that in each iteration i, we delete one value from the array "a" (which is the largest element of "a" at that iteration), so that in the next iteration, we get the next largest value in the array. Therefore, "a" is getting modified

RUNTIME:

(n = number of elements in the array "a")

We traverse 10 times through the array.

Each time the number of elements of the array decreases by one.

Therefore, number of elements visited = n + (n-1) +(n-2) + ... + (n-9) = 10n-45.

The runtime is of the order **O(n)** as constants and lower order terms doesnt matter in asymptotic analysis

(*NOTE: To find the largest number from an unsorted sequence, we need to traverse through the entire list atleast one time. Hence, the runtime of the algorithm is atleast of the order O(n). Hence, the runtime for finding 10 largest numbers is also atleast O(n). In practice, there are algorithms better than linear search to find the 10 largest numbers. However, we are considering asymptotic analysis for comparison, and therefore every algorithm of the order O(n) is considered as equal.)

PYTHON CODE IMPLEMENTATION

In [7]:

```python
def ten_largest(a):
    maxi=[]     # maxi stores the resulting list of 10 largest elements
    for i in range(10):  # Iterating 10 times through the array
        maxi.append(a[0]) # We assume the first element as the largest element and store it in maxi[i]
        for item in a:   # iterating through each item of the array
            if(item>maxi[i]): # Updating the value of maxi[i] if any item is found larger than maxi[i]
                maxi[i]=item
        a.remove(maxi[i])   # Removing the element maxi[i] from a so that next iteration gives NEXT largest value
    return maxi

#========TESTING=============================================================================================
==========

# provided test array:
arr=[21.3,27,562.9,7,-4,-37,5,82.6,7,27,77,101,-71] # This is the test array
arr_org=arr[:] # We make a copy of the original contents of the array in case we need it later
print("The given array is",arr)
print("The ten largest elements in the array is",ten_largest(arr))


# user-input test array:
lst = []
n = int(input("\nEnter number of elements(should be greater than 10) : "))
for i in range(0, n):
    ele = float(input("Enter value:"))
    lst.append(ele) # adding the element
lst_org=lst[:] # We make a copy of the original contents of the array in case we need it later
print("The ten largest elements in the array is",ten_largest(lst))

#===========================================================================================================
==========
```

```
The given array is [21.3, 27, 562.9, 7, -4, -37, 5, 82.6, 7, 27, 77, 101, -71]
The ten largest elements in the array is [562.9, 101, 82.6, 77, 27, 27, 21.3, 7, 7, 5]

Enter number of elements(should be greater than 10) : 12
Enter value:22
Enter value:12
Enter value:34
Enter value:44
Enter value:-3
Enter value:0.3
Enter value:-32
Enter value:-32
Enter value:-43
Enter value:90
Enter value:23.4
Enter value:46
The ten largest elements in the array is [90.0, 46.0, 44.0, 34.0, 23.4, 22.0, 12.0, 0.3, -3.0, -32.0
]
```

==

Qn 4:

We can use divide and conquer strategy to multiply two numbers. In this case, we particularly use Karatsuba's algorithm for better efficiency

Explaining karatsuba algorithm

The numbers to be multiplied(x and y) are taken as input by the multiply_karat function.

If the numbers to be multiplied are single digit numbers, then the multiplication is trivial and the result is returned.

If not, then each number is split as follows.

x = a.$2^{n/2}$ + b

y = c.$2^{n/2}$ + d

Next, we find ac, bd and (a + b)(c + d) by recursive calls of the same function

Finally, we find the result as (ac).2^{n} + ((a + b)(c + d) - ac - bd).$2^{n/2}$ + bd (Karatsuba 's method)

NOTE: The output of the multiply_karat function will be in the decimal format (as, in Python, this is the default output type even for binary operations). So, this is converted into binary format using bin() while printing the answer out.

Karatsuba implementation manually

Given below is a manual explanation of what happens inside the algorithm:

We need to multiply 10011011 and 10111010 (n=8)

10011011 = 1001.2^{4} + 1011 [Here, a = 1001, b = 1011 (a + b = 10100)]

10111010 = 1011.2^{4} + 1010 [Here, c = 1011, d = 1010 (c + d = 10101)]

ac = 1001 x 1011

bd = 1011 x 1010

ad + bc = (a + b) x (c + d) - ac - bd = (10100 x 10101) - (1001 x 1011) - (1011 x 1010)

We have, 10011011 x 10111010 = (ac).2^{8} + (ad + bc).2^{4} + bd

= (1001 x 1011).2^{8} + ((10100 x 10101) - (1001 x 1011) - (1011 x 1010)).2^{4} + (1011 x 1010)

Clearly, we have three multiplication steps in the above equation: which are: 1001 x 1011, 10100 x 10101 and 1011 x 1010

Therefore, we have three multiplications to perform recursively in the next step. This recursive multiplication goes on until it becomes one-digit multiplication and then they are combined to form the final answer

Karatsuba implementation in Python

In [17]:

```
def multiply_karat(x,y):
    n = max(len(str(x)),len(str(y))) # finds the maximum no of digits of the two input numbers

# If the numbers to be multiplied are single digit numbers, then the multiplication is trivial and the result is
returned.
    if n<=1:
        return x*y

# If not single digit numbers, then each number is split into two
# and then we find ac, bd and(a + b)(c + d) using recursive calls of multiply_karat
    a = x//(2**(n//2))
    b = x%(2**(n//2))
    c = y//(2**(n//2))
    d = y%(2**(n//2))
    ac = multiply_karat(a,c)
    bd = multiply_karat(b,d)
    aplusb_into_cplusd = multiply_karat(a+b,c+d)

# Finally, we find the result as (ac). 2^  + ((a + b)(c + d) - ac - bd). 2^(/2)  + bd (Karatsuba 's method)
# Note: we use 2**(2*(n//2)) instead of doing 2**n directly because 2**n leads to unexpected results when n is od
d

    return ((ac*(2**(2*(n//2))) + ((aplusb_into_cplusd-ac-bd)*(2**(n//2))) + bd))
print("Product in binary:", bin(multiply_karat(0b10011011, 0b10111010)))
```

```
Product in binary: 0b111000010011110
```

===

Qn 5:

Aim is to make an algorithm which finds the largest element of a unimodal array in O(logn) time. It is a unimodal array(An array is called unimodal iff it can be split into an increasing sequence followed by a decreasing sequence).

Main property: If we are able to find an element which is larger than the element before it and after it, then that element is the largest element. Since the elements are distinct, there will be only one largest value

Description of the algorithm

The algorithm has been implemented using divide-and-conquer.

In each iteration of the algorithm, the algorithm compares the value at the middle of the array(arr[mid]) with the value before it(arr[mid-1]) and the value after it(arr[mid+1].

1: If arr[mid-1] is smaller and arr[mid+1] is larger, then clearly this is the increasing part of the array and the largest element is to the right of arr[mid].

2: If arr[mid-1] is larger and arr[mid+1] is smaller, then clearly this is the descending part of the array and the largest element is to the left of arr[mid].

3: If both arr[mid-1] and arr[mid+1] are smaller than arr[mid], then arr[mid] is the largest element

In either of the cases 1 and 2, the part of the array containing the largest element is passed on for the next iteration of the function. This is repeated till the largest element is found.

Runtime

In each iteration, we find the middle value and compare the value with value before it and the value after it and return only one half of the original array(in the worst case).

Since we use recursion and return half of the array each time, it is of O(logn) [log is of base 2]

Implementation in python

In [24]:

```
# L represents the starting index of the array(or the subarray) to be scanned and R represents the last index of
the array
# arr is the array which is passed on to the largest_search function in each iteration.
# In each iteration, a subarray of the original unimodal array is passed on to largest_search

def largest_search(L,R, arr):

# L==R means length of the array becomes 1 i.e, clearly there is only one element remaining
# and this is guaranteed to be the largest element
    if L == R:
        return arr[L]

# if the length of the array is 2, then either the first or second element has to be the largest
# the following two if  statements check for the larger element and returns it
    elif R == L + 1:
        if arr[L] < arr[R]:
            return arr[R]
        if arr[L] > arr[R]:
            return arr[L]

# for all other cases, the following sequence gets executed.
# mid is the index of the middle value
    else:
        mid = (L + R)//2

    # If both arr[mid-1] and arr[mid+1] are smaller than arr[mid], then it is the largest element
        if arr[mid] > arr[mid - 1] and arr[mid] > arr[mid + 1]:
            return arr[mid]

    #  If arr[mid-1] is smaller and arr[mid+1] is larger,
    # then clearly this is the increasing part of the array and the largest element is to the right of arr[mid].
        elif arr[mid] > arr[mid - 1] and arr[mid] < arr[mid + 1]:
            return largest_search(mid + 1, R, arr)

    # if the above two statements are false, it means arr[mid-1] is larger and arr[mid+1] is smaller,
    # then clearly this is the decreasing part of the array and the largest element is to the left of arr[mid].
        else:
            return largest_search(L, mid - 1, arr)

# ===============  TESTING ======================================================================================
=========
# provided test array
arr=[1,2,3,4,5,6,8,10,45,43,21,20,5,-2] # This is the test unimodal array
print("The given array is",arr)
print("Thelargest element in the unimodal array is",largest_search(0,len(arr)-1,arr))

# user-input test array
lst=[]
n = int(input("\nEnter number of elements of the unimodal array : "))
for i in range(0, n):
    ele = float(input("Enter value: "))
    lst.append(ele) # adding the element
print("The largest element in the unimodal array is",largest_search(0,len(lst)-1,lst))
```

```
The given array is [1, 2, 3, 4, 5, 6, 8, 10, 45, 43, 21, 20, 5, -2]
Thelargest element in the unimodal array is 45

Enter number of elements of the unimodal array : 7
Enter value: 0
Enter value: 0.3
Enter value: 4
Enter value: 5
Enter value: -4
Enter value: -5.4
Enter value: -10
The largest element in the unimodal array is 5.0
```

==

Qn 6:

In the question it is not mentioned whether we should preserve the order of the array, hence I am assuming that order preservation is not required and therefore the algorithm I am providing sorts the array along with removing duplicates.

I am using an algorithm which uses the divide and conquer principle. Actually, the provided algorithm is just a simple modification of MergeSort algorithm. We can modify the mergesort algorithm by adding an extra condition to remove duplicates.

Explanation of the algorithm:

The given array is passed to the merge_sort function.

mergesort function: The array is split into two halves - (these halves are unequal if the number of elements of the array is odd) - and each half is passed through the mergesort function recursively.Finally, after many recursions, we end up with arrays of length1. Then we pass those values to the merge function.

merge function: The merge function combines the arrays of length 1 to form arrays of length 2 and so on until we get back a single array. During this process, the merge function traverses through both the subarrays being merged, similar to the MergeSort algorithm. However, whenever the merge function finds elements that are common to both the subarrays, the common element is added only once to the merged array.

Here, the subarrays are themselves already sorted and already the duplicates are removed as well, so there is no possibility that an element is repeated in the same subarray
Finally, after merging all elements, we obtain a new array which is sorted without duplicates.

Explanation of the running time:

It is known that mergesort algorithm has time complexity O(n logn) - (It can be proved just like any divide-and-conquer algorithm). In this algorithm, we just add a single step modification which is of time complexity O(1) to the merge function in mergesort algorithm. Therefore, this algorithm also has same complexity as mergesort, which is **O(n logn)**

Python implementation

In [26]:

```
# The merge function combines the arrays of length 1 to form arrays of length 2 and so on until we get back a sin
gle array.
# During this process, the merge function traverses through both the subarrays being merged, similar to the Merge
Sort algorithm.
# However, whenever the merge function finds elements that are common to both the subarrays,
# the common element is added only once to the merged array.
def merge(L, R):
    arr = []
    n = len(L) + len(R)
    i = 0
    j = 0
    for k in range(n):
        while i < len(L) and j < len(R):

            if L[i] < R[j]:
                arr.append(L[i])
                i += 1
            #========This is the extra step========
# Whenever there is a common element in L and R, that element is appended once, and we increase BOTH i and j so t
hat
# the index of  L moves to next element in L and the index of  R moves to next element in R.
# So, the common value is eliminated from the list
# [Note that in the other expressions in the conditional statement(ie, less than case and more than case),
# we increment only one index, acc to which element is smaller.
# Hence the other element can still be there in unsorted subarray]
            elif L[i] == R[j]:
                arr.append(L[i])
                i += 1
                j += 1

            #======================================
            else:
                arr.append(R[j])
                j += 1

        while i < len(L):
            arr.append(L[i])
            i += 1

        while j < len(R):
            arr.append(R[j])
            j += 1

    return arr

# The array is split into two halves - (these halves are unequal if the number of elements of the array is odd) -
# and each half is passed through the mergesort function recursively.
# Finally, after many recursions, we end up with arrays of length1. Then we pass those values to the merge functi
on.
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    n=len(arr)
    m = n//2
    l = merge_sort(arr[0:m])
    r = merge_sort(arr[m:n])
    return merge(l, r)

# ================================= TESTING =====================================================================
=======

# provided test array
arr = [12, 11, 13, 11, 5, 6, 7, 7]
print("array without duplicates=",merge_sort(arr))

# user-input test array
lst=[]
n = int(input("\nEnter number of elements of the array : "))
for i in range(0, n):
    ele = float(input("Enter value: "))
    lst.append(ele) # adding the element
print("array without duplicates=",merge_sort(lst))

# ==============================================================================================================
========
array without duplicates= [5, 6, 7, 11, 12, 13]

Enter number of elements of the array : 7
Enter value: 2
Enter value: 2
Enter value: 2
Enter value: 1
Enter value: 4
Enter value: 5
Enter value: 5
array without duplicates= [1.0, 2.0, 4.0, 5.0]
```

== END ==