Home Work 10

The problems in this problem set cover lectures C11 and C12

1.

a. Define a recursive binary search algorithm.

```
If lb > ub
        Return -1
else
        Mid := (lb+ub)/2
        If Array(Mid) = element
                Return Mid
        Elsif Array(Mid) < Element
                Return Binary_Search(Array, mid+1, ub, Element)
        Else
                Return Binary_Search(Array, lb, mid-1, Element)
        End if
End if
```

b. Implement your algorithm as an Ada95 program.

```
46. function Binary_Search (My_Search_Array : My_Array; Lb : Integer; Ub: Integer; Element : Integer)
return Integer is
   47.    mid : integer;
   48. begin
   49.    if (Lb> Ub) then
   50.       return -1;
   51.    else
   52.       Mid := (Ub+Lb)/2;
   53.    if My_Search_Array(Mid) = Element then
   54.       return(Mid);
   55.    elsif My_Search_Array(Mid) < Element then
   56.       return (Binary_Search(My_Search_Array, Mid+1, Ub, Element));
   57.    else
   58.       return (Binary_Search(My_Search_Array, Lb, Mid-1, Element));
   59.       end if;
   60.    end if;
   61.
   62. end Binary_Search;
   63. end Recursive_Binary_Search;
```

c. What is the recurrence equation that represents the computation time of your algorithm?

Recursive Binary Search		**Cost**
if (Lb> Ub) then		c1
return -1;		c2
else		c3
Mid := (Ub+Lb)/2;		c4
if My_Search_Array(Mid) = Element then		c5
return(Mid);		c6
elsif My_Search_Array(Mid) < Element then		c7
return (Binary_Search(My_Search_Array, Mid+1, Ub, Element));		T(n/2)
else	c8	
return (Binary_Search(My_Search_Array, Lb, Mid-1, Element));		T(n/2)
end if;		c9
end if;		c10

In this case, only one of the recursive calls is made, hence only one of the T(n/2) terms is included in the final cost computation.

Therefore T(n) = (c1+c2+c3+c4+c5+c6+c7+c8+c9+c10) + T(n/2)
= T(n/2) + C

d. What is the Big-O complexity of your algorithm? Show all the steps in the computation based on your algorithm.

T(n) = T(n/2) + C

¥ $T(n) = aT(n/b) + cn^k$,
where a,c > 0 and b > 1

T(n) =

$$\begin{cases} O(n^{\log_b a}) & a > b^k \\ O(n^k \log_b n) & a = b^k \\ \text{?} & a < b^k \end{cases}$$

$1 = 2^0$, hence the second term is used,

T(n) =

2. What is the Big-O complexity of :

a. Heapify function

A heap is an array that satisfies the heap properties i.e., A(i) ≤ A(2i) and A(i) ≤ A(2i+1).

The heapify function at 'i' makes A(i .. n) satisfy the heap property, under the assumption that the subtrees at A(2i) and A(2i+1) already satisfy the heap property.

Heapify function	**Cost**
Lchild := Left(I);	c1
Rchild := Right(I);	c2
if (Lchild <= Heap_Size and Heap_Array(Lchild) > Heap_Array(I))	c3
Largest:= Lchild;	c4
else	c5
Largest := I;	c6
if (Rchild <= Heap_Size)	c7
if Heap_Array(Rchild) > Heap_Array(Largest)	c8
Largest := Rchild;	c9
if (Largest /= I) then	c10
Swap(Heap_Array, I, Largest);	c11
Heapify(Heap_Array, Largest);	T(2n/3)

T(n) = T(2n/3) + C'
= T(2n/3) + O(1)

a = 1, b = 3/2, f(n) = 1, therefore by master theorem,

T(n) = $O\left(n^{\log_b a} \log n\right)$
= $O\left(n^{\log_{3/2} 1} \log n\right)$
= O(1 * log n)
= O(log n)

The important point to note here is the T(2n/3) term, which arises in the worst case, when the heap is asymmetric, i.e., the right subtree has one level less than the left subtree (or vice-versa).

b. Build_Heap function

Code	**Cost t(n)**
Heap_Size := Size;	c1
for I in reverse 1 .. (Size/2) loop	n/2+1
Heapify(Heap_Array, I);	(n/2) log n
end loop;	n/2

Therefore T(n) = c1+ n/2+1 + (n/2)log n + n/2
= (nlog(n))/2 + n + (c1+1)

Simplifying

=> T(n) = O(n log(n))

c. Heap_Sort

Heap Sort	**Cost t(n)**
Build_Heap(Heap_Array, Size);	O(nlogn))
for I in reverse 2.. size loop	n
Swap(Heap_Array, 1, I);	c1(n-1)
Heap_Size:= Heap_Size -1;	c2(n-1)
Heapify(Heap_Array, 1);	O(log n)(n-1)

T(n) = 2 O(nlogn) + (c1+c2+1)n - O(log n) +

= 2 O(nlog n) - O(log n) + c'n

Simplifying,

=> T(n) = O(nlogn)