Tanveer Bariana

Csc 140

Hmwrk 1

(c ) = constant

2.1-3

int search(number a[], number v){

int n= a.length-1;

int nil= 0;

while( n => 0 && a[n]!= v ){ // n \* (c )= n

n--; // (c )

}

return nil + n; //(c )

}

Init: n = a.length, a[] = a[]orig, v = v orig, nil= 0; it starts at end of array so the element either exists to the left of it, is at it, or doesn’t exist in the array.

LI: v == a[n], v== a[n-1], or v !ϵ a, the element either exists to the left of it, is at it, or doesn’t exist in the array.

Maintenance: n decrements each time until it hits -1, each time it checks if the element is there and the element either exists to the left of it, is at it, or doesn’t exist in the array.

Termination: v == a[n], v== a[n-1], or v !ϵ a, at the end we know that the element either exists to the left of it, is at it, or doesn’t exist in the array.

2.2-1

Ω( n3)

2.2-2

sort (Array A){

int i, j; //(c)

int n = A.length; //(c)

for(j to n-1){ //(n-1) // end result is (n-1)\*(n-j+1)\*c= (n2- nj + j -1 )\* c

int min = j; //(c)

for(i= j+1; i to n){ //(n- j+1)\*(c)

if(A[i] < A[min]){ //(c)

min = i; //(c)

}

}

if(min !=j){ //(c)

swap(A[j], A[min]); //(c)

}

}

}

LI: A is unsorted for indexes < j

It only needs to run the first n-1 elements because the nth element will always be the highest value after this sort ends. Doing the nth element would be an extra step of the nth element comparing itself to the nth element. O(n2), Ω(n2)

2.2-3

Worst case: O(n), it can only do as many operations as there are inputs plus a constant.

Average case: O(n), the average amount of operations it would do is n/2 so the growth rate that matters is n.

2.2-4

We could cut down the number of operations the algorythm requires, it might not matter in larger inputs but in the best case we may not need to account for that rate of growth as much.