DAA Assignment

Riya Negi
28
CE

TCS-505

Answers

Q1. Asymptotic notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as input size for the algorithm increases.

Types of Asymptotic notations.

- Big O Notation - It defines an upper bound of an algorithm. It bounds a function only from above. Eg. In insertion sort, it takes linear time in best case & quadratic time in worst case. So time complexity is $O(n^2)$

- Omega-Ω Notation - It gives the tighter lower bound. Eg. Time complexity of Insertion sort is $\Omega(n)$.

- Theta - Θ Notation - It decides whether the upper & lower bounds of a given function are the same. The average running time of an algorithm is always between the lower bound & the upper bound. If the upper & lower bound give the same result, then Θ will have same rate of growth.

Eg. $f(n) = 10n + n$ is the expression. Then, its upper bound $g(n)$ is $O(n)$. The rate of growth in the best case is $g(n) = O(n)$.

Q2. Logarithmic complexity. $(O(\log n))$

$$\log 2^k = \log n$$
$$k \log 2 = \log n$$
$$k = \log n$$

Q3.
$$T(n) = 3T(n-1)$$
$$T(n) = 3(3T(n-2)) = 3^2 T(n-2)$$
$$T(n) = 3^2 (3T(n-3))$$

$T(n) = 3^n T(n-n) = 3^n T(0) = 3^n$

Q4. $T(n) = 2T(n-1) - 1$

$T(n) = 2(2T(n-2)-1) - 1 = 2^2 T(n-2) - 2 - 1$

$= 2^2(2T(n-3) - 2 - 1) - 1 = 2^3 T(n-4) - 2^2 - 2^1 - 2$

$= 2^n T(n-n) - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots 2^2 - 2^1 - 2^0$

$= 2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots - 2^0$

$= 2^n - (2^n - 1) = 1$

$\therefore$ Time complexity is $O(1)$

Q5: $O(\sqrt{n})$

Q6. $O(\sqrt{n})$

Q7. $O(n(\log^2 n))$

Q8: $O(n)$

Q9. $O(n \log n)$

Q10: For functions n^k & a^n, the asymptotic relation is n^k is $O(a^n)$ i.e. Time complexity of n^k is of order $O(n^k)$

Q11.
```
void fun (int n)
{ int j = 1, i = 0;
  while (i < n)
  { i += j;
    j++; }
}
```

Thus, $i_j = i_{j-1} + j$, i.e. a recurrence relation which gives time complexity as $O(\sqrt{n})$

Q12. The fibonacci series is 0, 1, 1, 2, 3, 5, 8 . . . . . so on.

i.e. $f(0) = 0$

$f(1) = 1$

$f(n) = f(n-1) + f(n-2)$

The recursive eq^n for TC -

$T(n) = T(n-1) + T(n-2) + O(1)$

This converges to a non-tight upper bound of

$T(n) = O(2^n)$

The space complexity can be imagined by.

Space complexity

$= O(N + N/2)$

$= O(3N/2)$

$= O(N)$

N stacks frames

$f(i-1)$ 000

$N/2$ 00

$f(i-2)$

T.C. $= O(2^n)$

S.C. $= O(N)$

Q13. (i) T.C. $= O(n \log n)$

```
for (i=0; i<=n; i++)
  { for (j=0; j<=n; j*=2)
      { // O(1); }
  }
```

(ii) TC $= O(n^3)$

```
for (i=0; i<=n; i++)
  { for (j=0; j<=n; j++)
      { for (k=0; k<=n; k++)
          { // O(1) }
      }
  }
```

(iii) T.C. = $O(\log(\log n))$

for (i = n; i > 0; i = $\sqrt{i}$)

{ // O(1) }

Q14. $T(n) = T(n/4) + T(n/2) + cn^2$

We know that $T(n/2) > T(n/4)$

$\rightarrow$ $T(n) = 2T(n/2) + cn^2$

This is of the form $aT\left(\frac{n}{b}\right) + f(n)$ so we can apply master's theorem

$T(n) < \log_b^a = \log_2^2 = 1$

$= \Theta(n^2)$ [case 3]

$T(n) = \Theta(n^2)$

Q15. for (i=1; i<=n; i++)

{ for (j=1; j<n; j+=i)

{ O(1) }

}

$i = 1, 2, 3, 4 \ldots n \rightarrow O(n)$

$j = 1, 2, 3, 4 \ldots n \rightarrow i = 1$

$j = 1, 3, 5, 7 \ldots n \rightarrow i = 2$.

$j = 1, 4, 7, 10 \ldots n \rightarrow i = 3$.

$\rightarrow$ each run of 'i' loop forms an A.P. for 'j' terms

$\rightarrow$ $O(n-i) \Rightarrow O(n)$

so total $T(n) = O(n \times n)$

$= O(n^2)$

Q16. for $(i = 2;\ i <= n;\ i = pow(i, k))\ \{ \ldots \}$

$i = 2,\ 2^k,\ 2^{k^2} \ldots\ 2^{k(\log_k(\log(n)))}$

$\rightarrow 2^{k(\log_k(\log(n)))}$

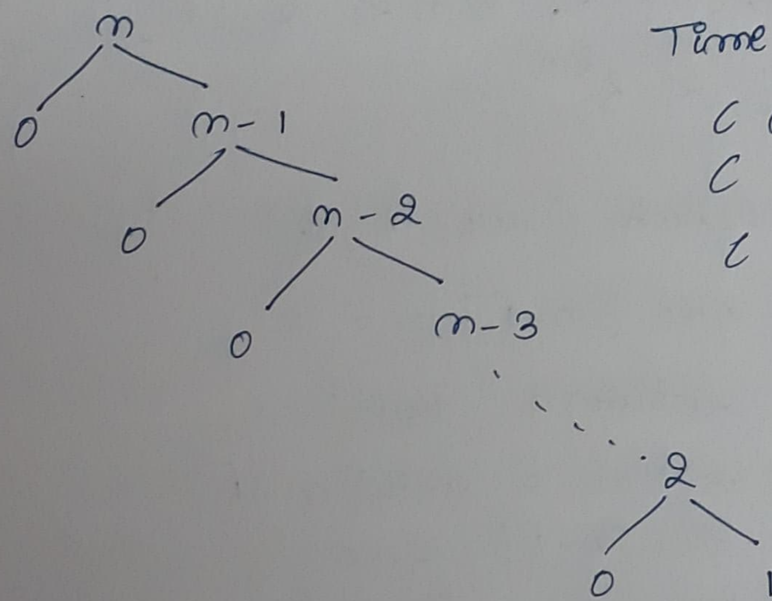
$= 2^{\log n} = n$

$\rightarrow$ Total No. of iterations $= \log_k(\log(n))$

$\therefore T(n) = O(\log(\log(n)))$

Q17. The partitioning scheme of 99% & 1% is one of the most unbalanced partition possible.

The tree is :



Time complexity

Time cn.

$c(n-1)$

$c(n-2)$

$c(n-3)$

$\vdots$

$2c$

0

Total time :

$\rightarrow cn + c(n-1) + c(n-2) + \ldots\ 2c$

$= c((n+1)(n/2) - 1)$

Using the big theta $\rightarrow \Theta(\ldots)$ notation, we can ignore trivial terms, $\Theta(n^2)$

i.e. worst case $TC = O(n^2)$

Assumptions - The original call takes (n) time, where c is some constant

Difference between ~~int~~ extremes = n (Input size)

Q18.

(a) $100 < root(n) < \log\log(n) < \log(n) < n\log(n) <$
$n < n^2 < \log(n!) < 2^n < 2^{2n} < 4^n$

(b) $1 < \log(\log(n)) < \sqrt{\log(n)} < \log(n) <$
$\log(2n) < n\log(n) < 2\log(n) < n < \log n! <$
$2n < 4n < n^2 < n! < 2(2^n)$

(c) $96 < \log 8(n) < \log_2(n) < n\log_6(n) <$
$n\log_2(n) < 5n < 8n^2 < \log(n!) < 7n^3$
$< n! < 8^{2n}$

Q19. linearSearch (arr, n, x)

```
{ if arr [n-1] == x
      return "true"
  lastVal = arr [n-1]
  arr [n-1] = x
  for i = 0, i += 1
      if arr [i] == x
          arr [n-1] = last Val
          return (i < n-1)
}
```

arr = Sorted array
n = No. of elements in array
x = key.

Q20. Iterative

```
insertion Sort (arr, n)
    for i=1 to n, +1
        key = arr [i]
        j = i-1
        while (j >= 0 && arr [j] > key)
        { arr [j+1] = arr [j]
          }
          j = j-1
        arr[ j+1] = key;
        }
    }
```

Recursive

```
insertion Sort (arr, n)
{ if (n <= 1)
      return;
  insertion Sort (arr, n-1)
    last = arr [n-1]
    j = n-2.
  while ( j >= 0 && arr [j] > last
      { arr [j+1] = arr [j]
           j--;
        }
      arr [j+1] = last
}
```

Insertion Sort is called online sort because it doesn't need to know anything about what values it will sort while algorithm in running.
other sorting methods: Insertion sort, Reservoir sampling etc.

Q21.

Algorithm	Time complexity Best	Avg.	Worst	Space Complexity Worst.
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(N)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(\log n)$

Q22

Inplace	Stable	Online
Bubble Sort	Merge Sort	Insertion Sort
Selection Sort	Insertion Sort	
Heap Sort	Bubble Sort	
Quick Sort		
Insertion Sort		

```
binarySearch (arr, l, r, key)
{ if (l >= r) return -1
     x = key
   mid = l+(r-l)/2
   if (arr [mid] == x return mid;
   if (arr [mid] > x)
   return binarySearch (arr, l, mid-1, key)
   return binarySearch (arr, mid+1, r, key)

}
```

TC = O(log n)
SC = O(log n)

Iterative

```
binarySearch (arr, l, r, x)
{ while (l<=r)
   { m = l+(r-l)/2
      if (arr[m] == x)
         return m
      if (arr[m] < m)
         l = m+1
      else
         r = m-1
   }
   return -1
}
```

TC = O(log n)
SC = O(1)

Recurrence relation for recursive binary Search.

$$T(n) = 1 + T[n/2] + 1$$
$$= T(n/2) + c$$

So, $T(n) = T(n/2) + c$

This can be solved using Master's method

Case 2 TC = $O(\log n)$