24/3/22

PIYUSH CHAUHAN
CSE SPL-1
34

DAA Tutorial 2

Ans1.

```
void fun (int n)
{ int j=2, i=0;
  while (i<n)
  { i = i+j;
    j++;
  }
}
```

values after execution.

1st time → i = 1

2nd time → i = 1+2

3rd time → i = 1+2+3

4th time → $i = (1+2+3+\dots i) < n$

$= \frac{i(i+1)}{2} < n$

$= i^2 < n$

$= i = \sqrt{n}$

Time complexity = $O\sqrt{n}$ Ans

Ans2. RECURENCE RELATION.

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

Let $T(n)$ denote the time complexity of $f(n)$

For $f(n-1)$ and $f(n-2)$ time will be $T(n-1)$ and $T(n-2)$. We have one more addition to sum on results,

for $n > 1$

$$T(n) = T(n-1) + T(n-2) + 1 \quad \text{—— (1)}$$

for $n=0$ and $n=1$, no addition occurs

$\therefore T(0) = T(1) = 0$

Let $T(n-1) \approx T(n-2)$ — (2)

Adding (2) in (1)

$$T(n) = T(n-1) + T(n-1) + 1$$
$$= 2 \times T(n-1) + 1$$

using Backward substitution.

$\therefore T(n-1) = 2 \times T(n-2) + 1$

$$T(n) = 2 \times [2 \times T(n-2) + 1] + 1$$
$$= 4 \times T(n-2) + 3$$

we can substitute

$$T(n-2) = 2 \times T(n-3) + 1$$

$$\Rightarrow T(n) = 8 \times T(n-3) + 7$$

General eqn

$$T(n) = 2^k * T(n-k) + (2^k - 1) \quad \text{—— (3)}$$

for $T(0)$

$n-k=0 \Rightarrow k=n$

substituting value in (3)

$$T(n) = 2^n \times T(0) + 2^n - 1$$
$$= 2^n + 2^n - 1$$

$$\boxed{T(n) = O(2^n)}$$ Ans

space complexity = O(N)

Reason:
The function calls are executed sequentially, sequential execution guarantee that the stack size will not exceed the depth of calls, for first f(n-1) it will create N stack frames, the other f(n-2) will create N/2. So the largest is N.

Ans3) O(nlogn) :

```
#include <iostream>
using namespace std;
int partition (int arr[], int start, int end)
{ int pivot = arr[start];
  int count = 0;
  for (int i = start+1; i <= end; i++)
  { if (arr[i] <= pivot)
      count++;
  }
  int pivot_ind = start + count;
  swap (arr[pivot_ind], arr[start]);
  int i = start, j = end;
  while (i < pivot_ind && j > pivot_ind)
  { while (arr[i] <= pivot)
    { i++;
    }
```

```
while ( arr[j] > pivot )
    { j-- ; }
if ( i < pivot_ind && j > pivot_ind )
    { swap ( arr[i++], arr[j--] );
    }
}
return pivot_ind;
}
void quick ( int arr[], int start, int end)
{ if ( start >= end ) return;
  int p = partition (ar, start, end);
  quicksort (ar, start, p-1);
  quicksort (ar, p+1, end);
}
int main()
{ int arr[] = {6, 8, 5, 2, 1}
  int n = 5;
  quicksort ( ar, 0, n-1);
  return 0;
}
```

ii) $O(n^3)$:-

```
int main()
{ int n=10
  for(int i=0; i<n; i++)
   { for(int j=0; j<n; j++)
     { for(int k=0; k<n; k++)
       { printf("*");
       }
     }
   }
  return 0;
}
```

iii) $O(\log(\log n))$

```
int countPrimes(int n)
{ if(n<2) return 0;
  boolean[] nonprime = new
                       boolean[n];

  nonprime[1] = true;
  int numNonPrimes = 1;
  for(int i=2; i<n; i++)
   { if(nonPrime[i])
     { continue;
     }
    int j=i*2;
```

```
while (j < n)
  { if (! nonPrime [j])
      { nonPrime [j] = true;
        numNonPrime ++ ;
      }
    j += i;
  }
}
return (n-1) - numNonPrime;
}
```

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

using Master's Theorem.

we can assume $T(n/2) >= T(n/4)$

eq^n can be rewritten as

$T(n) <= 2T(n/2) + cn^2$

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

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

Also $T(n) >= cn^2$

$\Rightarrow T(n) >= O(n^2)$

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

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

and $T(n) = \Omega(n^2)$

$$\boxed{T(n) = O(n^2)}$$

Ans.

Ans 5)

for i=1, inner loop is executed n times
for i=2; inner loop is executed n/2 times
for i=3; inner loop is executed n/3 times

It is forming a series :-

$\Rightarrow n + n/2 + n/3 + \dots + n/n$

$\Rightarrow n(1 + \frac{1}{2} + \frac{1}{3} \dots + \frac{1}{n})$

$\Rightarrow$ ~~$n + n \times \sum_{k=2}^{n} \frac{1}{k}$~~

$\Rightarrow n \times \log n$

Time complexity = $O(n \log n)$ Ans

Ans 6 for(int i=2; i<=n ; i=pow(i,k))

{ // O(1)

}

with iterations

i take values
for 1st iteration → 2
for 2nd iteration → 2^k
for 3rd " $= (2^k)^k$

:

for n iterations → $2^{k \log_k(\log(n))}$

∴ Therefore last term must be less than or equal to A

$$2^{k \log_n(\log(n))} = 2^{\log n} = n$$

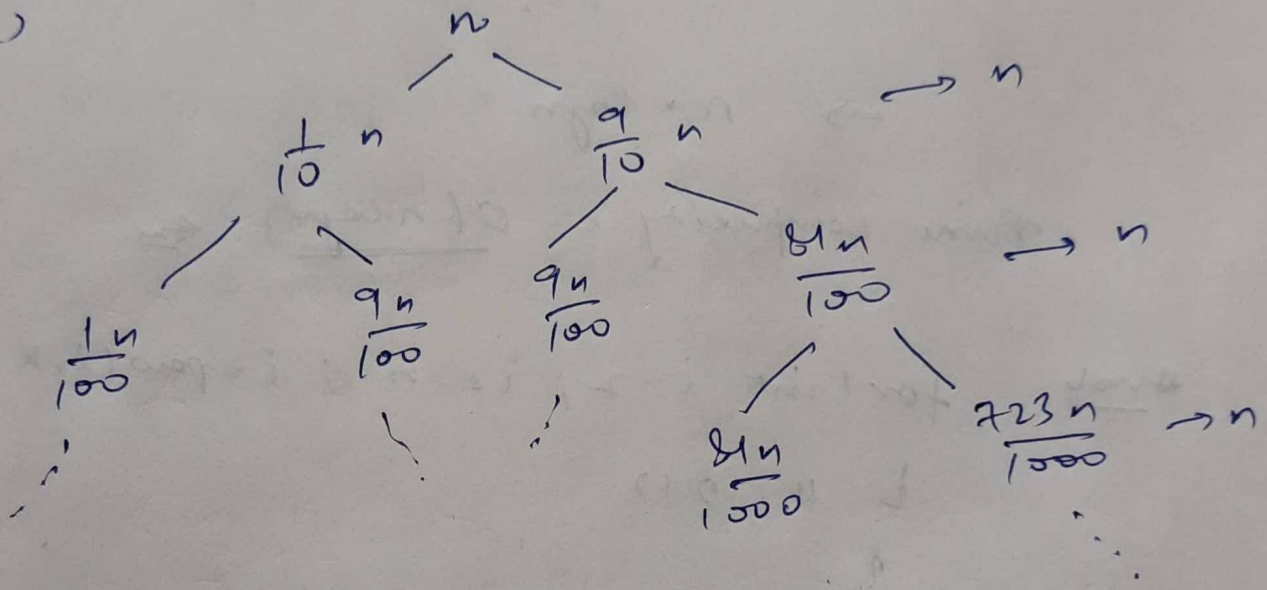
Each iteration takes constant time.

∴ Total iteration $= \log_k(\log(n))$

Time complexity $= O(\log(\log(n)))$

Ans

Ans 7)



If we split in this manner

Recursive Relation :-

$$T(n) = T(9n/10) + T(n/10) + O(n)$$

where first branch is of size $\frac{9n}{10}$ and 2nd one is $\frac{n}{10}$

solving the above using recursion tree approach calculating values.

At 1st level, value = n

At 2nd level, value = $\frac{9n}{10} + \frac{n}{10} = n$

value remains same at all levels i.e n

Time complexity = summation of values.

$= O(n * \log_{10/9} n)$ (upper bound)

$= \Omega(n \log_{10} n)$ (lower bound)

$\Rightarrow \underline{O(n \log n)}$ Ans.

Q4) a) $100 < \log(\log n) < \log n < (\log n)^2 < \sqrt{n}$
$< n < n(\log n) < \log(n!) < n^2 < 2^n$
$< 4^n < 2^{2^n}$.

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

c) $96 < \log_e n < \log 2n < 5n < n(\log_5 n) <$
$n(\log_2 n) < \log(n!) < 8n^2 < 7n^3$
$< n! < 8^{2n}$