DAA

Assignment - 1.

PAGE NO.:
DATE: / /20

Q.1).

Ans. The main idea of asympotic analysis is to have a measure of the efficiency of algorithms that don't depend on machine specific constant and doesn't require algorithms to be implemented and time taken by programs to be compared. Asympotic notations are mathematical tools to represent the time complexity of algorithms for asympotic analysis. The following 3 asympotic notations are mostly used to represent the time complexity of algo.

→ notation → The theta notation bounds a function from above and below, so it defines exact asympotic behaviour.

A simple way to get theta notations of an expression is to drop low order terms and ignore leading constant.

for example.

$$3x^3 + 6x^2 + 6000 = \Theta(n^3)$$

→ Big o notation → The Big o notation defines an upper bound of an algo, it bounds a function only from above.

for eg.; consider the case of insertion sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of insertion sort is $O(n^2)$, it covers linear time.

If we use O notations to represent time complexity of insertion sort, we have to use 3 statements for best and worst cases.

- The worst case time complexity of insertion sort is $O(n^2)$
- The best case time complexity of insertion sort is $O(n)$

→ Ω notation → Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω notation can be useful when we have lower bound on time complexity of an algo. The best case preferences of an algo. is generally not useful, the omega notation is the least used notations among all three.

Q2)

Ans. for(i=1 to n) (i = i*2)

$i = 1, 2, 4, 8 \; - - - \; n$

$i = 2^0, 2^1, 2^2, 2^3, - - - \; 2^n$

This is an G.P.

So,

$a = 1; \quad r = \frac{t_2}{t_1} = \frac{2}{1} = 2.$

$t_k = ar^{k-1}$

$n = 1 \cdot 2^{k-1}$

$2^k = \log_2(2n)$

$= \log_2(n) + \log_2(2).$

$k = \log n + 1$

Time complexity = $o(\log n)$

Q3.)

Ans. $T(n) = \{3T(n-1)$ if $n>0$, otherwise $1\}$

$T(n) = 3T(n-1)$ — ①

$T(1) = 1$

Put $n = n-1$ in eq. ①

$T(n-1) = 3T((n-1)-1)$

$T(n-1) = 3T(n-2)$ — ②.

Put eq. ② in eq. ①.

- $T(n) = 9T(n-2)$ — ③.

Put $n = n-2$ in eq. ①

$T(n-2) = 3T(n-3)$ — ④.

Put ④ in eq. ③

$T(n) = 3[3T(n-3)]$

$= 27T(n-3)$ — (5)

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

Put $n-k=1$

$n = k+1 \Rightarrow k = n-1$

$T(n) = 3^{n-1} T[n-(n-1)]$

$T(n) = 3^{n-1} T(1)$

$T(n) = 3^{n-1} \cdot 1$

$T(n) = \frac{3^n}{3}$

$T(n) = O(3^n)$

Q4)

Ans. $T(n) = \{ 2T(n-1)-1$ if $n>0$, otherwise $1\}$

$T(1) = 1$

$T(n) = 2T(n-1)-1$ — (1)

Put $n = n-1$ in eq. (1).

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

$= 2T(n-2)-1$ — (2)

Put (2) in (1)

$T(n) = 2[2T(n-2)-1]-1$

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

$= 4T(n-2)-3$ — (3)

Put $n-2$ in eq. (1)

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

$T(n-2) = 2T(n-3) - 1$ — (4).

Put eq. (4) in eq. (3)

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

$= 8T(n-3) - 4 - 2 + 1$

$= 8T(n-3) - 7$ — (5)

$T(n) = 2^k T(n-k) - (2^k - 1)$ — (6)

So, Put $n-k=1$

$n = k+1$

$k = n-1$

$T(n) = 2^{n-1} T(n-n+1) - (2^{n-1} - 1)$

$= \frac{2^n}{2} T(1) - \left(\frac{2^n}{2} - 1\right)$

$= \frac{2^n}{2} - \left(\frac{2^n}{2} - 1\right)$

$= \frac{2^n}{2}(1-1) - 1$

$T(n) = O(1^n) = O(1)$

Q5)

Ans. $S(k) = 1 + 2 + 3 - - - + k$

k steps when $S(k) > n$

$\therefore S(k) \Rightarrow (k + (k+1)/2) \leq n$

$O(x^2) \leq n$

$x = O(\text{root } n)$

Time complexity $= O(\sqrt{n})$.

PAGE NO.:
DATE: / /20

Q6)

Ans. for(i=1; i*i<n; i++)

$T(n) = 1+1+(n+1)+n+n$
$= (3n+3)$
$= O(n)$

Q7.)

Ans. for i : Executes $O(n)$ times
for j : Executes $O(\log n)$ times
for k : Executes $O(\log n)$ times.

So, time complexity
$T(n) = O(n * \log n * \log n)$
$= O(n \log^2 n)$

Q8.

Ans. Inner loop executes only one time due to break statement
$T(n) = O(n*1)$
$= O(n)$

Q9.

Ans. for outer loop time
Complexity = $O(n)$

for inner loop time
complexity = $O(n)$

So, time complexity
$T(n) = O(n*n)$
$= O(n^2)$

PAGE NO.:
DATE: / /20

Q10.

Ans: To answer this we need to think about the function, how it grows and what function binds it together.

n^k is a polynomial function and a^n is a exponential function we know that polynomial always grow more slowly then exponential.

If we were to say that n^k is $o(a^n)$, then we would be saying that n^k has an asympotic upper bound of (a^n). As polynomial grow more slowly than exponential.

If we were to say that n^k is $\Omega(a^n)$, then we would be saying that n^k has an asympotic lower bound of $\Omega(a^n)$ - that for a large enough n, n^k always grows faster than a^n. Is that true? No, because, polynomial always grow slower than exponential.

If we were to say that n^k is $\Theta(a^n)$, then we would be saying that n^k is "tightly bound" by $\Theta(a^n)$ that for large enough n, n^k is always sandwiched btw $k+a^n$ and $k+a^2$. Is that true? No because polynomial always grow slower than exponentials.

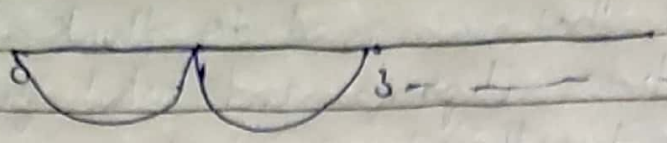
In order for n^k to $\Theta(c^n)$ it would need to be both $o(a^n)$ and $\Omega(a^n)$ which is not possible.

In conclusion, the only true statement here is that n^k is $O(a^n)$.

PAGE NO.:
DATE: / /20

Q11.

Ans.



$= 1 + n + n$

$= 1 + 2n$

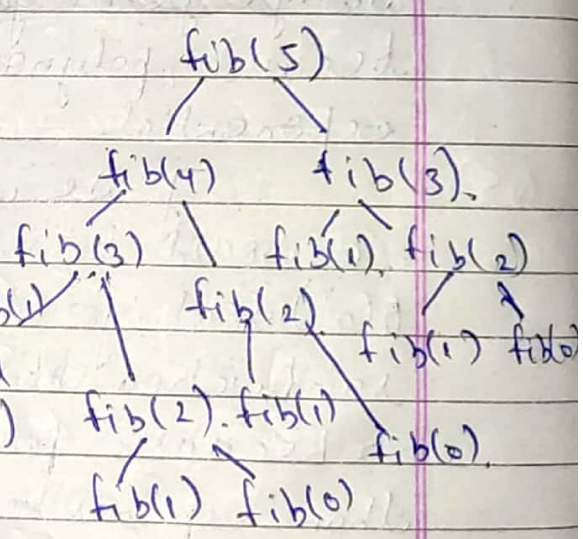
$T(n) = O(n)$.

Q12.

Ans.

```
int fib (int n)
 {
  if (n<=1)
    return n;
    return fib(n-1) * fib(n-2);
  }
int main()
 {
  int n=9;
 printf("%d", fib(n));
  getchar();
  return 0;
  }
```

Extra Space: O(n) if we consider function call stack size otherwise O(1)



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

which is exponential

We can observe that implementation does a lot of repeated work. So this is a bad implementation for n^{th} fibonacci number.

PAGE NO.:
DATE: / /20

Q13.

Ans: $T(n) = O(n \log n)$

```
int i, j, k = 0;
for (i = n/2; i <= n; i++)
  {
    for (j = 2; j <= n; j * 2)
      {
        k = k + n/2;
      }
  }
```

→ $T(n) = O(n^3)$

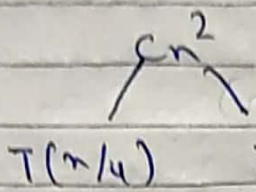
```
Sum = 0;
for (int i = 1; i <= n; i++)
for (int j = 1; j < n; j = j + 2)
for (int k = 1; k <= n; k = k + 2)
  Sum += k;
```

→ $T(n) = O(\log(\log n))$

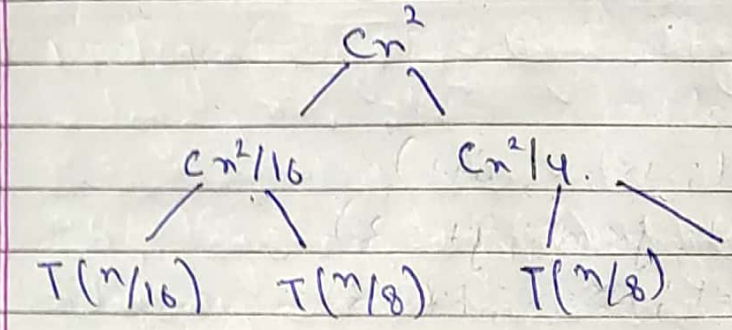
```
// Here c is a constant greater than 1
for (int i = 2; i <= n; i = pow(i, c))
  {
    // Same O(1) expression
  }
for (int i = n; i > 1; i = fun(i))
  {
    // Same O(1) expression
  }
```

Q14.

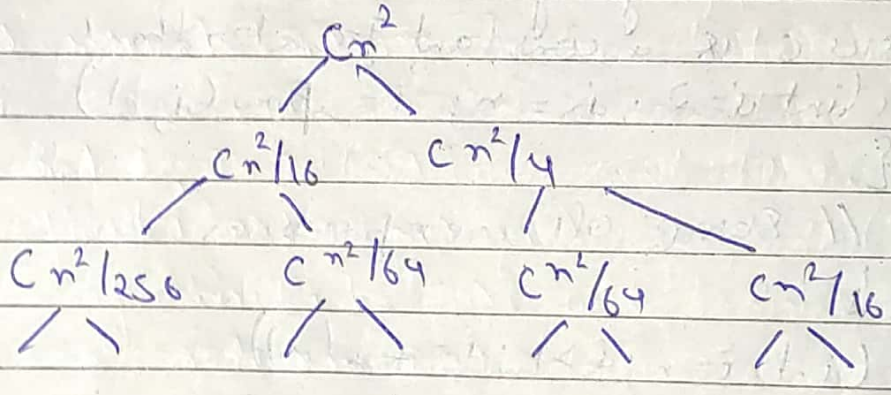
Ans. Function is initial recurrence tree for following recurrence relation.



if we break it again, we get function, recurrence tree.



Breaking down further gives us



$$T(n) = C(n^2 + 5(n^2/16) + 25(n^2/256) + \cdots$$

$$T(n) = \frac{n^2}{(1-5/16)}$$

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

Q15.

Ans. for outer loop.

$= 1 + (n+1) + n$

$= 2n + 2$

$T(n) = O(n)$

for inner loop

$T(n) = O(n)$.

So, time complexity of fun()

$T(n) = O(n+n)$

$= O(n^2)$

Q16.

Ans. Time complexity of loop is considered as $O(\log\log n)$ if the loop variable is increased / decreased exponentially by a constant amount.

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

Q17.

Ans. Quick sort's worst case is when the choosen pivot is either the largest (99%) or smallest element in the list. When this happens, one of the two sublists will be empty. So Quick sort is only called on one list during sort step.

$T(n) = T(n-1) + n + 1$ (Recurrence relation)

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

$T(n) = T(n-3) + 3n - 1 - 2 - 3.$

$T(n) = T(1) + \sum_{i=0}^{n-1} (n-i)$

$$T(n) = \frac{n(n-1)}{2}$$

Now, time complexity $T(n) = O(n^2)$.

Q18.

Ans.

a) 100, $\log n$, $\log \log n$, $\log(n!)$, n, $n!$, root(n), $n \log n$, n^2, 3^n, 2^{2n}, 4^n.

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

c) 96, $\log 8(n)$, $\log_2(n)$, $\log(n!)$, $5n$, $n!$, $n \log_2 n$, $n \log_8 n$, $8n^2$, $7n^3$, 8^{2n}.

Q19.

Ans.

```
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
        if (arr[i] == x)
            return i;
            return -1;
}

int main(void)
{
    int arr[] = {2, 3, 4, 10, 40};
    int n = 10;
```

```
int n = sizeof(arr) / sizeof(arr[0]);

int result = search(arr, n, x);
(result == -1)
    printf("Element is not present in array");
    printf("Element is present at index %d",
                                    result);

return 0;
}
```

The time complexity of above is O(n).

Q20.

Ans. Recursive insertion sort alg.

```
// sort an arr[] of size n
insertion sort(arr, n)
Loop from i = 1 to n-1
```

a) Pick element arr[i] & insert it into sorted sequence arr[0 ... i-1]

Iterative insertion sort alg.

To sort an array of size n in ascending order.

1. Iterate from arr[1] to arr[n] over the array.
2. Compare the current element (key) to its predecessor.
3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

An online algo. is one that can process its input peice-by-piece in a serial fashion i.e. in the order that the input is fed to the algo. without having entire input available from the beginning.

Insertion sort considers one input element per iteration and produces a partial solution without considering future elements. Thus insertion sort is an online algo.

Q21.

Ans:-

→ Selection Sort → It is sound and easy to understand. It's also very slow and has a time complexity of $o(n^2)$ for both its worst and best case inputs.

→ Insertion Sort → Insertion sort has $T(n) = o(n)$ when the input is a sorted list. for an arbitary sorted list $T(n) = o(n^2)$.

→ Merge Sort → worst case complexity $T(n) = o(n \log n)$

→ Quick Sort → worst case complexity $T(n) = o(n^2)$

Best case complexity $T(n) = o(n)$.

Q22.

Ans. Inplace/outplace technique → A sorting technique is inplace if it does not use any extra array to merge the sorted sub array.

Online/offline technique → Only insertion sort is online technique because of the underlying algo. it uses.

Stable/unstable technique → A sorting technique is stable if it does not change the order of elements with the same value.

Bubble sort, insertion sort and merge sort are stable techniques while selection sort is unstable as it may change the order of elements with the same value.

Q23.

Ans. 1. Compose x with the middle element.
2. If x matches with the middle element, we return the mid index.
3. Else if x is greater than the middle element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
4. Else (x is smaller) recur for left half.

Linear Search → Time complexity $T(n) = O(n)$

Space complexity = $O(1)$.

We don't need any extra space to store anything.

Binary Search → Time complexity : $T(n) = O(\log n)$
Space complexity = $O(1)$
implementation and in case of recursive implementation, $O(\log n)$ recursion call stack space.

Q24.

Ans.

```
bool binarysearch(int *arr, int l, int r, int key)
  {
   if (l > r) return false;
    int mid = (l+r/2);
   if (arr[mid] == key) return true;
```

$T(n/2)$ → `else if (arr[mid] < key) return binary search (arr, mid+1, r, key);`

$T(n/2)$ → `else return binarysearch (arr, l, mid-1, key);`

```
  }
```

So, recursive relation

$$T(n) = T(n/2) + 1$$

$T(1) = 1$ // Base case.
