Design and Analysis of Algorithms

• Name: Sapna Bhatt
• BTech CSE (5th Sem)
Univer. Rno 1961149
Sec B
Roll No: 29.

Assignment-01

Ques ① What do you understand by Asymptotic Notation. Define different Asymptotic notation with Examples?

Ans ① The main idea of Asymptotic analysis is to measure of the efficiency of the algorithms that don't depend on machine specific constant and doesn't require algorithm to be implemented and time taken by program to be compared Asymptotic notations are mathematical tools to represent the time complexity of algorithms. For Asymptotic analysis following 3 asymptotic notation are mostly used to represent the time complexity of algo.

① Big O notation: The big O notation defines the upper bound of an algorithm 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 converts linear time.

If we use, O notation to represent time complexity of insertion sort, we have to use 2 statements for best and worst cases:

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

$O(g(n)) = \{ f(n) :$ there exist +ve constant c and no such that

$0 <= f(n) <= c * g(n)$ for all $n > no. \}$

(11) Θ notation: The Θ notation bounds a function from above and below, so it defines exact asymptotic behaviours

A simple way to get theta notation of an expressions is to drop low order terms and ignore leading constant. for example

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

$\Theta g(n) = \{ f(n) :$ there exists +ve constants no, c_1, c_2 that

$0 <= c_1 * g(n) < f(n) <= c_2 * g(n)$ for all $n >= no \}$

The above defination means if $f(n)$ is theta $g(n)$, then the value $f(n)$ is always b/w $c * g(n)$ & $c_2 * g(n)$ for large values of n ($n >= no$). The defination of theta also requires that $f(n)$ must be non negative for values of n greater than no.

(12) Ω notation: Just as Big O notation provides an asymptotic upper bound on a function Ω notation provides lower bound

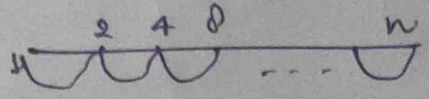
Ω notation can be useful when we have lower bound on time complexity of an algo is generally not useful, the omega notation is the least used notation among all three.

$\Omega g(n) = \{ f(n) :$ there exists +ve constants c & no such that

$0 <= c * g(n) <= f(n)$ for all $n >= no \}$

(2) what should be time complexity of -

for (i=1 to n) { i = i * 2 }



i = 1, 2, 4, 8 ... n

→ $2^0, 2^1, 2^2, 2^3 \ldots 2^k$

This is a GP so

$q = 1 \quad r = \frac{t_2}{t_1} = \frac{2}{1} = 2$

kth term $< t_k = ar^{k-1}$

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

$2^k = 2n$

$k = \log_2(2n)$

$\rightarrow \log_2(n) + \log_2(2)$

$k = \log n + 1$

time complexity $= O(\log(n+1))$

$= O(\log n)$

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

$T(n) = 3T(n-1)$ — ①

$T(1) = 1$

But $n = n-1$ in eqn ①

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

Put eq 2 in ①

$T(n) = 9T(n-2)$ — ②

Put $n = n-2$ in eqn ①

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

Put 1 in eqn ③

$T(n) = 9[3T(n-3)]$

$= 27T(n-3)$

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

Put $n-k = 1$

$n = k+1$, $k = n-1$

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

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

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

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

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

This is the time complexity

④ $T(n) = [2T(n-1)-1$ if $n>0$ otherwise ①

$T(1) = 1$

$T(n) = 2T(n-1) - 1$ — ①

put $n = n-1$ in eqn ①

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

$\rightarrow 2T(n-2) - 1$ — ②

put ② in ①

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

$\rightarrow 4T(n-2) - 2 - 1$

$\rightarrow 4T(n-2) - 3$ — ③

put $n-2$ in eqn ①

$T(n-2) = 2T(n-3) - 1$ — ④

put ④ in ③

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

$\rightarrow 8T(n-3) - 4 - 2 - 1$

$\rightarrow 8T(n-3) - 7$ — ⑤

$T(n) = 2^k T(n-k) - 2^{k-1} - 0$

so put $n-k = 1$

$n = k+1$

$k = n-1$

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

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

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

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

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

⑤ what should be time complexity of

```
int i=1, s=1
while (s<=n) {
    i++; s=s+i;
    printf ("#");
}
```

$s(k) = 1+2+3+\ldots+k$

& stops when $s(k) > n$

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

$O(k^2) \leq n$

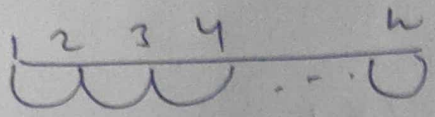
$k = O(\text{root } n)$

time complexity $= O(\sqrt{n})$

⑥ Time complexity

```
void function(int n) {
    int i, count=0;
    for(i=1; i*i<=n; i++)
```

```
    count++;
}
```



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

$\Rightarrow 3n+3$

$\Rightarrow O(n)$

(3) Time complexity of

```
void function(int n)
{
    int i,j,k,count=0;
for(i=n/2; i<=n; i++)
    for(j=1; j<=n; j=j*2)
      for(k=1; k<=n; k=k*2)
       count++;
}
```

for i : executes $O(n)$ times

for j : executes $O(\log n)$ times

for k : executes $O(\log n)$ times

so time complexity

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

$\Rightarrow O(n \log^2 n)$

(4) Time complexity of

```
Function(int n){
if(n==1) return; //.
```

```
for (i = 1 to n) {
    for (j = 1 to n) {
        printf ("*");
    }
    function (n-3);
}
```

Inner loop execute only one time due to break statement

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

$\rightarrow O(n)$

(9) Time complexity of -

```
void function(int n)
{
    for (i = 1 to n) {    // O(n)
        for (j = 1; j <= n; j = j+1)
            printf ("*");
    }
}
```

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

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

so time complexity

$T(n) = O(n * n)$

$\rightarrow O(n^2)$

(10) For the func^n, n^k, & a^n what is asymptotic relationship between functions?

Assume that $k >= 1$ & $a > 1$ are constants find out the value of c and n_0 for which relation holds.

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

n^k is a polynomial func^n and c^n is a exponential function we know that polynomials always grow more slowly than exponential.

If we were to say that n^k is $O(c^n)$ then we would be saying that n^k has an asymptotic upper bound of c^n. As polynomial grow more slowly then exponential.

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

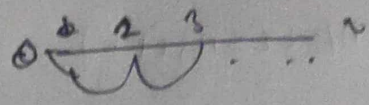
If we were to say that n^k is Θc^n then we would be saying that n^k is "tightly bound" by $O(c^n)$ that for large enough n, n^k is always sandwiched b/w $1/c c^n$ & $k^2 \times c^2$. Is that true? No because polynomial always grows slower than exponential

In order for n^k to $O(c^n)$ it would need to be both $O(c^n)$ & Ωc^n which is not possible.

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

Ques 11. What is time complexity of below code and why?

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



$\Rightarrow 1+n+n$

$\Rightarrow 1+2n$

$T(n) = O(n)$

(12) Write recurrence relation for the recursive func that prints fibonacci series. Solve the recurrence to get time complexity of program and space complexity?

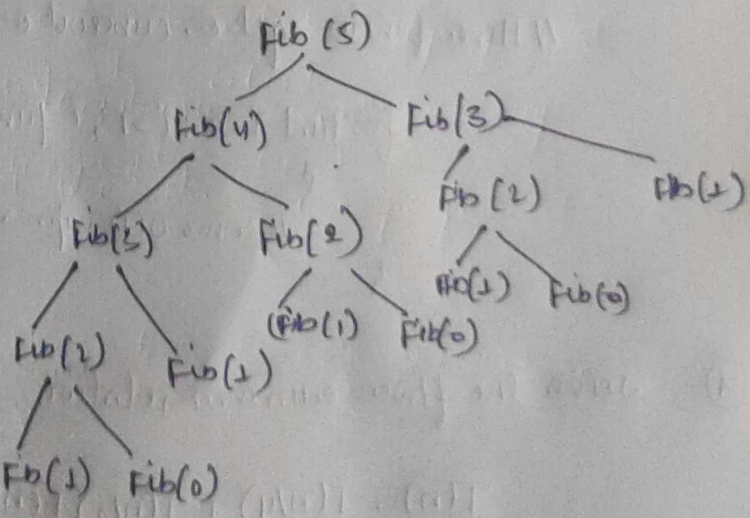
Ans 12.

```
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;
}
```

Time complexity $\Rightarrow T(n)$

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

which is exponential



Extra space: $O(n)$ if we consider function call stack size, otherwise $O(1)$.

We can observe that implementation does a lot of repeated work. So this is bad implementation for nth fibonacci number

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

```
int i, j, k=0
for(i= n/2; i<=n; i++)
  {
    for(j= 2; j<=n; j=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+=2)
for (int k=1; k<=n; k+=2)
  sum += k;
```

→ $T(n) = O(\log(\log n))$

```
// Here c is constant greater than 1
for (int i=2; i<=n; i = pow(i,c))
  {
    //some O(1) expressions
  }
```

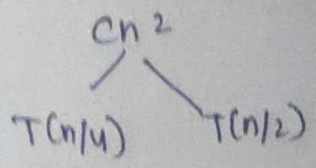
// Here fun is sqrt or cuberoot or any other constant root.

```
for (int i=n; i>1; i=fun(i))
    {
      // some O(1) expressions
    }
```

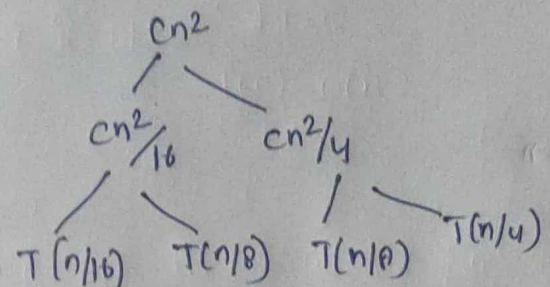
(14) solve the flwn recurrence relation:-

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

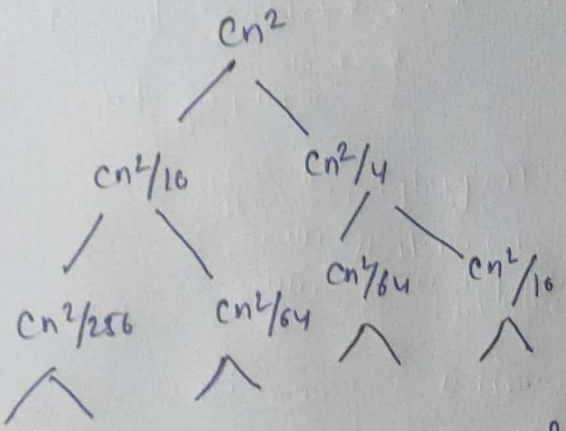
flwn is initial recurrence theree for following recurrence relation



if we break it again, we get flow recurrence tree



Breaking down further gives us:



To know T(n) we need to calculate sum of tree nodes level by level. If we sum the above three level by level we get flow series

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

Above series is G.P with Ratio 5/16 we can sum above tree for infinite sum

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

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

Ques 15. what is time complexity of following function fun(n) ?

```
int fun(int n)
{
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j < n; j += i)
        {
            // some O(1) task
        }
    }
}
```

For outer loop

$\Rightarrow 1 + 1 + (n+1) + n$

$\Rightarrow 2n + 3$

$T(n) = O(n)$

for inner loop

$T(n) = O(n)$

so time complexity of fun()

$T(n) = O(n * n)$

$\Rightarrow O(n^2)$

Ques 16. what should be time complexity of :

```
for (int i = 2; i <= n; i = pow(i, k))
{
    // some O(1) expression or statements.
}
```

where k is constant

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

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

Ques 17. Write a recurrence relation when quicksort repeatedly divides the array into 2 parts. Derive time complexity.

Quick sort worst case is when the choosen pivot is either the largest (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 the sort step.

$$T(n) = T(n-1) + n+1 \quad \text{(Recurrence relation)}$$
$$T(n) = T(n-2) + n-1 + n-2$$
$$T(n) = T(n-3) + 3n-1-2-3$$
$$T(n) = T(1) + \sum_{i=0}^{n-1} (n-i)$$
$$T(n) = \frac{n(n-1)}{2}$$

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

(18) Arrange the f/wn in increasing order of rate of growth

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

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

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

→ $\log n, \log 2n, \sqrt{\log} n, 2\log n, \log n!), 2n, 4n, n, n!, n\log n, n^2, 2(2^n)$.

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

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

(19) Write linear search code to search an element in a sorted array with minimum comparisons

```
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 x=10;
     int n= size of (arr) / size of arr[0];

     // function call

  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 algo is O(n).

(20) Write pseudocode for iterative and recursive insertionsort, Insertion sort is called online sorting. why?

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

To sort an array of n in ascending order.

Iterate from arr[1] to arr[n] over the array.
compare the current element (key) to its predecessors
If the key element is smaller than its predecessors.
compare it to the element 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 piece 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 begining.

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

(21) complexity of all the sorting algo that has been discussed.

→ 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 & best case inputs

→ Insertion sort : Insertion sort has $T(n) = O(n)$ when the input is a sorted list. For an arbitary sorted list $T(n) = O(n^2)$.

... relation

Merge sort: worst case complexity $T(n) = O(n \log n)$

Quick sort: worst case complexity $T(n) = O(n^2)$
best case complexity $T(n) = O(n)$

(20) Divide all sorting algo into a place / stable / online sorting.

In place / outplace technique: A sorting technique is inplace if its does not use any extra memory to sort array. Among all techniques merge sort is outplaced technique. as it require an 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.

(21) Write pseudocode for binary search. what is time and space complexity of linear & binary search?

Ans: 1. compare x with the middle element
2. If x matches with the middle element, then x can only lie in the right half subarray after the mid element. so we can recur for left half.

(3) Else if x is greater than the middle element then x can easily lie in the right half subarray after the mid element. so we recur for the right halfway.

(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 : $O(1)$ in case of recursive implementation,

$O(\log n)$ recursion call stack space.

(24) Write recurrence relation for binary recursive search.

T(n) → size of array

```
bool binary search (int *arr , int l, int r, int key)
{
  if(l>r) return false;
    int mid = (l+r)/2;
  if (arr[mid] == key ) return true; // 1
T(n/2) → else if (arr[mid] < key ) return binary search
                    (arr, mid+1, r, key);
T(n/2) → else return binary search (arr, l, mid-1, key);
}
```

so, recurrance relation

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

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