Name - Dhruv Saini
Section - ML

Roll No - 24, 2015241
Subject - Design and Analysis of Algorithm (TCS 505)

Assignment - 1

① Asymtotic Notation

Asymtotic Notations are used to tell the complexity of an input algorithm when the input is very large.

The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that does not depend on the machine.

Following asymtotic notations are used -

① Big O Notation

- Big O Notation defines the upper bound of an algorithm, it bound a function only from above

For ex - Insertion Sort $O(n^2)$ (worst case)

② Θ Notation

- The Θ notation bounds a function from above and below, so it defines exect asymtotic behaviour

For ex - Expression : $2n^2 + n + 1$

Time complexity = $\Theta(n^2)$

③ Ω Notation

- The Ω notation provides the lower bound of an algorithm

For ex - Selection sort $\Omega(n^2)$ (Best case)

④ little-o Notation

- The little-o notation provides the upper bound of an algorithm, but it is not a tight bound

For ex Expression : $n + 2$

Time complexity = $o(n^2)$

⑤ little-ω Notation

- The little-ω notation provides the lower bound of an algorithm.

For ex: Expression - $n^2 + 3n + 1$

Time complexity = $\omega(n)$

(2)

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

Time complexity : $O(\log n)$

(3)

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

$T(n) = 3T(n-1)$ — (1)

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

Subs $T(n-1)$ in (1)

$T(n) = 3^2(T(n-2))$ — (3)

Subs $T(n-2)$ in (3)

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

$\therefore$ Time complexity $= O(3^n)$

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

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

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

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

$= 2^3(T(n-3)) - 2^2 - 2^1 - 2^0$

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

$T(n) = 1$

$= O(1)$

(5)

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

$s \Rightarrow 1, 3, 6, 10$

In general

$\frac{k(k+1)}{2}$

$\Rightarrow \frac{k^2+k}{2} > n$

$\therefore TC = O(\sqrt{n})$

⑥ void function(int n){
```
    int i, count=0;
    for(i=0; i*i<=n; ++i)      - O(√n)
        count++;               — O(1)
}
```
TC = $O(\sqrt{n})$

⑦ void function(int n){
```
    int i, j, k, count=0;           — O(n)
    for(i=n/2; i<=n; i++)
        for(j=1; j<=n; j=j*2)       - O(log n)
            for(k=1; k<=n; k=k*2)   - O(log n)
                ++count;
```
TC = $O(n (\log n)^2)$

⑧ function(int n){
```
    if(n==1) return;           — O(n)
    for(i=1 to n){
        for(j=1 to n){         - O(n)
            printf("*");
        }
    }
    function(n-3);             - O(n)
}
```
TC = $O(n^3)$

⑨ void function(int n){   — O(n)
```
    for(i=1 to n){
        for(j=1; j<=n; j=j+1)
            printf("*");
    }
}
```
TC = $\Theta(n \log n)$

(10) we have, n^k a^n

$k >= 1$ $a > 1$

let $k = a = 2$

$\Rightarrow n^2$ 2^n

So we can say $n^2 = O(2^k)$

$n^k = O(a^n)$

(11)

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

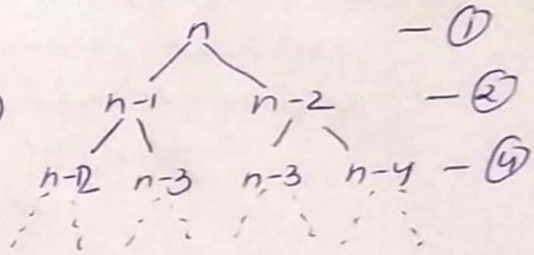
T.C. = $O(\sqrt{n})$

(Reference Question-05)

(12) Recurrence Relation for recursive fibonacci series

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

$\hookrightarrow 1, 2, 4, \_\_\_\_ 2^n$ (It is GP)



Here $a = 1$ $r = 2$

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

$= 2^{n+1} - 1$

$TC = O(2^n . 2) = O(2^n)$

Space complexity = $O(n)$

- The space complexity is $O(n)$ because recursive calls to the function take up space in stack.

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

$F(n-2)$ is called once $f(n-1)$ is finised executing which take up $O(n)$ space

$\therefore SC = O(n)$

(13)

(a) n(log n)

```
for (int i=0; i<n; ++i){
    for(int j=1; j<=n; j+=i){
        count +=i;
    }
}
```

(b) n^3

```
for (int i=0; i<n; ++i)
    for(int j=0; j<n; ++j)
        for(k=0; k<n; ++k)
            count +=i;
```

(c) log log n

```
int function(int n){
    if (n<=2)
        return n;
    else
        return (function(floor(sqrt(n)))+n);
}
```

(14)

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

Here, we can assume

$T(n/2) >= T(n/4)$

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

Now, using Master's theorem

$a = 2 \quad b = 2$

$\therefore$ $k = \log_2 2 = 1$

Here

$n^k = n^1 = n$

But $f(n) = n^2$

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

(16)

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

$TC = O(n \log n)$

(18)

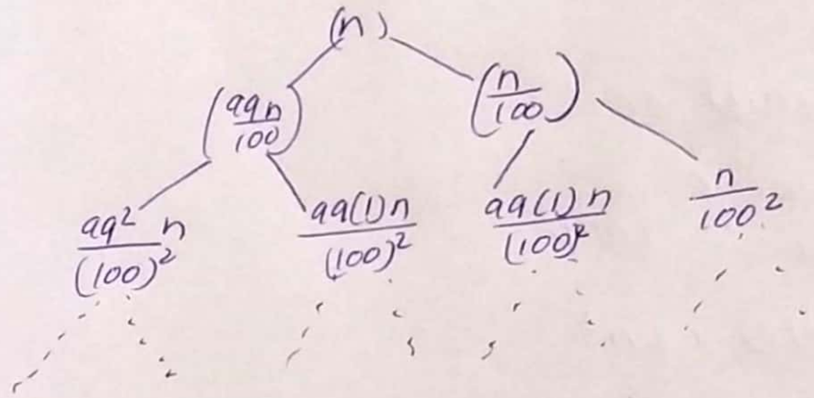
```
for(int i=2; i<=n; i=pow(i,k)){
    //some O(1) expression
}
```

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

↳ For any k greater than 1

(17) Recurrence Relation

$$T(n) = T\left(\frac{99n}{100}\right) + T\left(\frac{n}{100}\right)$$



Here, Taking $\left(\frac{99n}{100}\right)$

$\therefore TC = \log_{\frac{100}{99}} n \doteq \log_b n$

Since, it is a log complexity
here we can see our base is constant
therefore it does not matter as compared to n.

(18)

Inrcasing order

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

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

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

(19) Linear Search pseudo code

```
int linear_search ( array, size, key ){
        for ( i <- 0 to size ) {
            if ( array[i] = key )
                return i ;
        {
}
```

(20) Iterative Insertion sort

```
void insertion_sort (int arr[], int n ){
    int i, temp, j;
    for( i <- 1 to n ){
        temp <- arr[i];
        j <- i-1;
        while (j >= 0 && arr[j] > temp ){
            arr[j +1] <- arr[j];
            j = j -1 ;
        }
        arr[j +1] <- temp;
    }
}
```

Recursive Insertion Sort

```
void  insertion_sort(int arr[], int n){
        if (n <= 1)
            return;
        insertion_sort( arr, n-1);

        int last <= arr[n-1];
        int j <= n - 2;

        while( j >= 0 AND arr[j] > last){
            arr[j+1] <- arr[j];
            j--;
        }
        arr[j+1] = last;
}
```

- It considers one input element per iteration and produces a partial solution without considering future element
  ⇒ It is called online sorting.

<u>21 & 22</u>

Sorting Algorithm

Algorithm	Best case	Avg. case	worst case	space complexity	stable	Inplace	Online
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	yes	yes	NO
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$(O(1))$	NO	yes	NO
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	yes	yes	yes

(23) Pseudo Binary Search

```
int binary_search( int arr[], int l, int r,  int key){
        if (r >= l){
            int mid <- (l+r)/2;
            if( arr[mid] > x){
                return binary_search( arr, l, mid-1, x);
            }
            else{
                return binary_search( arr, mid+1, r, x);
            }
        }
        return -1;
}
```

	linear Search	(Recursive) Binary Search	(Iterative) Binary Search
Time complexity	$O(n)$	$O(\log n)$	$O(\log n)$
Space complexity	$O(1)$	$O(\log n)$	$O(1)$

(24) Recursive Binary search

Recurrence Relation

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