Design and Analysis of Algorithms

Assignment-1

Mamta Mehta
B.tech (C.S.E) Vth Sem
Sec-B
Roll No-17 (1961086)

Ques1 What do you understand by Asymptotic notations. Define Different asymptotic notations with example.

Ans. The main idea of asymptotic 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. Asymptotic notation are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. the following 3 asymptotic notation are mostly used to represent the time complexity of algo.

① Notation: The theta notation bounds a function from above & below so it define exact asymptotic behaviour A simple way to get theta notation of an expression is to drop low order terms & ignore leading constant, for example. $3n^3+6n^2+6000 = \Theta(n^3)$

② $\Theta(g(n)) = \{f(n)$: there exist +ve constants n_0, c_1, c_2 that $0 <= c_1 * g(n) <= f(n) <= c_2 * g(n)$ for all $n >= n_0\}$

The above definition means if $f(n)$ is theta $g(n)$, then the value $f(n)$ is always b/w $c_1 * g(n)$ & $c_2 * g(n)$ for large

value of n (n >= no). the definition of theta also requires that f(n) must be non-negative for values of n greater than no.

(ii) **Big O notation:** The Big O notation define an upper boun of an algo. it bounds a function only from above for eg. consider the case of Insertain Sort. it takes linear time in best case & quadratic time in worst case. we can apply say that the time complexity of Insertion sort is $O(n^2)$, it covers linear time.

If we use O notation to represent time complexity of insertionsort, we have to use 2 statement for best & 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 & no such that $0 <= f(n) <= c*g(n)$ for all $n >= no\}$

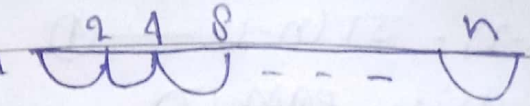
(iii) **Ω notation:** Just as Big O notation provides an asymptotic upper bound an Ω function notation provides an asymptotic lower bound.

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

$\Omega(g(n)) = \{f(n)$: there exist +ve constants c & no such that $0 <= c*g(n) <= f(n)$ for all $n >= no\}$

Ques 2: What should be time complexity of.

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



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

$= 2^0, 2^1, 2^2, 2^3 \ldots 2^n$

This is an G.P {0,

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

term^{K+n} $t_k = ar^{K-1}$

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

$2^K = 2^n$

$K = \log_2(2n)$

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

$K = \log n + 1$

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

Ques 3.

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 eqn (1)

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

$T(n-1) = 3T(n-2)$ —— (2)

Put eqn (2) in eqn (1)

$T(n) = 9T(n-1)$ ——— (3)

Put $n = n-2$ in eqn (1)

$T(n-2) = 3T(n-1)$ ——— (4)

Put (4) in eqn (3)

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

$= 27T(n-3)$ ——— (5)

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

Put $n-k = 1$

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

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

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

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

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

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

This is 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)

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

$= 2T(n-1) - 1$ ——— (2)

Put (2) in (1)

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

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

$= 4T(n-2)-3$ ——— (3)

Put n-2 in eqn (1)

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

$T(n-2) = 2T(n-3) - 1$ ——— (4)

Put eqn (4) in eqn (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 $n-k = 1$

$n = k+1$

$k = n-1$

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

$= 2^n/2 \, T(1) - (2^n/2 - 1)$

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

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

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

(5) What should be ~~time~~ complexity of

```
int i=1; S<1;
while (S<=n) {
i++ ; S=S+i;
Printf("#");
}
```

$S(K) = 1+2+3 - - - - +K$

Stops when $S(K) > n$

$\therefore S(K) = [K + (K+1)]/2 \leq n$

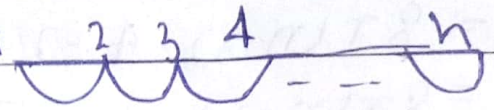
$O(x^2) \leq n$

$x = O(\text{root of } n)$ time complexity $= O(\sqrt{n})$.

(6) Time complexity of -

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

n, n+1, n



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

$= (3n+3)$

$= O(n)$

(7) 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(logn) times

for k: Executes O(logn) times

So Time complexity,

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

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

(8) Time complexity of :-

```
function(int n){
 if (n==1) return; //O(n)
 for(i=1 to n){
 for(j=1 to n){ ; //O(1)
  printf("*");
 }
 }
  function(n-3);
 }
```

inner loop execute only one time due to break statement

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

$= O(n)$

(9) Time complexity of -

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

for outer loop time complexity = O(n)

for inner loop time complexity = O(n)

So time complexity

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

$= O(n^2)$.

(10) for the func^n, n^k & a^n what is asymptatic relationship between these functions.

Assume that $k >= 1$ & $a > 1$ are constant find out the value of c & no for which relation holds.

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

n^k is a polynomial function & c^n is a Exponential function. We know that polynomial always grows 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 grows more slowly than exponential.

If we were to say that n^k is $\Omega(c^n)$, then we would be saying that n^k has an asymptatic 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 than exponential

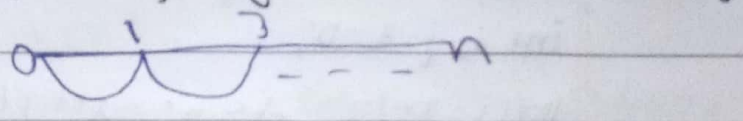
If we were to say that n^k is $\Theta(c^n)$, then we would be saying that n^k is tightly bound by $\Theta(c^n)$ that for large enough n, n^k is always sandwiched b/w $k1*c^n$ & $k2*c^2$ is that true? No because polynomial always grow slower than exponentials.

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

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

(11) What is the time complexity of below code & why?

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



$= 1+n+n$

$= 1+2n$

$T(n) = O(n)$

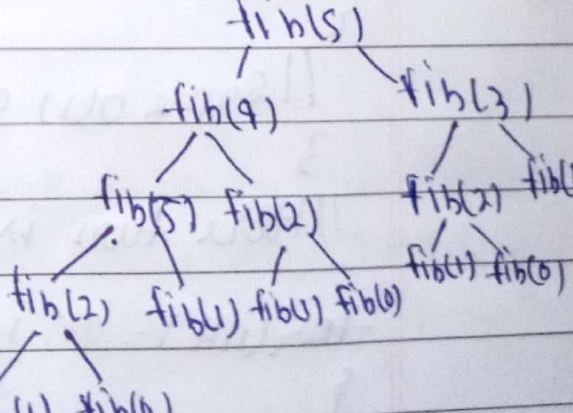
(12) Write recurrence relation for the recursive func^n that prints Fibonacci series solve the recurrence relation to get time complexity of the program & space complexity.

```
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 a bad implementation for nth fibonacci number.

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

```
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++)
for(int k = 1; k <= n; k++)
    Sum + = k;
```

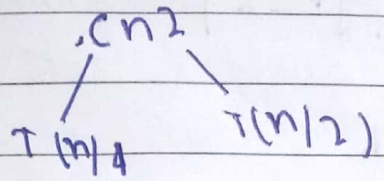
→ $T(n) = O(\log(\log n))$

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

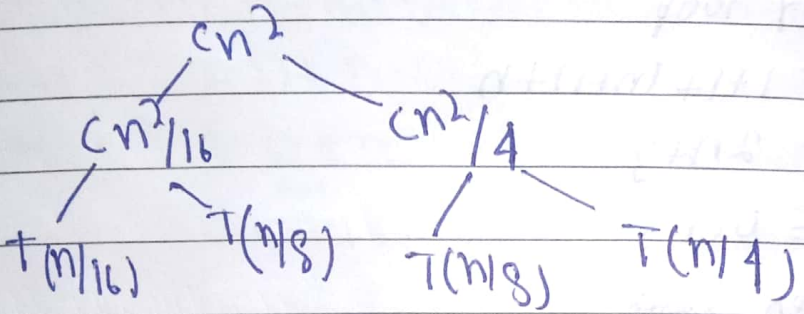
(14) Solve the function recurrence relation:

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

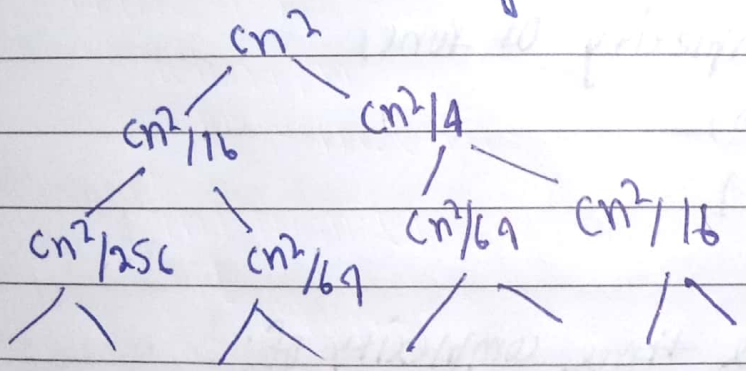
following is initial recurrence tree for following recurrence relation.



if we break it again, we get 2nd 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 tree level by level we get following 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.

$T(n) = n^2/(1 - 5/16)$

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

(15) What is time complexity of following function fun()?

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

for outer loop

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

$= 2n+3$

$T(n) = O(n)$

for inner loop

$T(n) = O(n)$

So, time complexity of fun()

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

$= O(n^2)$

(16) What should be the time complexity of:

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

where k is a constant.

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

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

(17) Write a recurrence relation when quick sort repeatedly divides array into 2 parts. Derive time complexity.

Quick sorts worst case is when the chosen 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 the 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 $T(n) = O(n^2)$

(18) Arrange the following in increasing order of rate of growth

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

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

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

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

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

→ $96, \log 8(n), \log 2(n), \log(n!), nn, n!, n\log_2 n, n\log_6 n, 8n^2, 7n^3, 8^{2n}$

(19) Write linear search code to reach 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= sizeof(arr)
//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 & recursive 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 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 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 & 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 & 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 $T(n) = O(n^2)$

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

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

best case complexity $T(n) = O(n)$

(21) Divide all the sorting algo into in place/stable/online sorting.

In place/outplace technique: A sorting technique is inplace if it does not use any extra memory to sort the array. Among all techniques merge sort is outplaced technique as it requires 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 & merge sort are stable techniques. While selection sort is unstable as it many change the order of elements with the same value.

(23) Write pseudocode for binary search. What is time & space complexity of linear & binary search.

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: $O(1)$ in case of iteration implementation & in case of recursive implementation $O(\log n)$ recursion call stack space.

(24) Write recurrence relation for binary recursive search.

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

So recursive relation $T(n) = T(n/2) + 1$

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