Assignement - 1

(1) Asymptotic notations are used to tell the complexity of an algorithm when the input is very large

i) Bigonotation (O) - It represents the upper bound of a function. for eg if a function $f(n)$ is $O(g(n))$, it means that $f(n)$ grows no faster than $g(n)$

ii) Omega Notation (Ω) - Lower bound of a functions growth. Ex - $f(n) = n^2 + 3n$ is $\Omega(n^2)$

iii) Theta notation (Θ) - Both upper & lower bounds indicating tight asymptotic behaviour.
Eg - $f(n) = 2n^2 + 3n$ is $\Theta(n^2)$.

(2) ~~O(logn)~~ ~~for(int i=0; i<n; i=i*2)~~

for(int i=0; i<n)

(3) ~~I~~

$i = i*2$

{
}

$t_k = ar^{k-1}$

$n = 2^{k-1}$

Taking log both sides

$\log_2^n = (k-1)\log_2^2 \Rightarrow \log_2^n = k-1$

$k = \log_2^n + 1$

Time complexity = $O(\log_2^n)$

Deepak

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

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

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

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

& so on

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

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

$T(n) = 3^n T(0)$

$T(n) = 3^n$

(5)

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

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

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

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

& so on

if we expand this —

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

we continue this until $n-k=0$, which means $k=n$.

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

since $T(0)$ is given as 1:

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

So the solution to the recurrence relation $T(n) = 2T(n-1) - 1$ is $T(n) = 2^n - n$.

(5) int i=1, s=1; —— (1)

```
while (s<=n)
{
    i++;
    s=s+i;
    print("#");
}
```

~~$i^2 <= n$~~

~~$i <= \sqrt{n}$~~

$i = 1, 2, 3, 4, 5$ ——

$s = 1, 3, 6, 10, 15$ ——

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

~~$i \times i <= n$~~

$i <= \sqrt{n}$

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

$i(i+1) < 2n$

$i^2 + i - 2n \le 0$

$i < \frac{-1 + \sqrt{1+8n}}{2}$

complexity = $O\sqrt{n}$

(6) void function (int n)

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

$i = 1, 2, 3, 4$ ———— $\sqrt{n}$

$i^2 = 1, 4, 9, 16$ ———— n^2

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

(7) void function(int n)
{
int i, j, k, count=0

$n/2$ —— for(i = n/2 ; i <= n; i++)
$\log_2 n$ —— for (j=1 ; j <= n; j = j*2)
$\log_2 n$ —— for (k=1; k <= n; k = k*2)
count ++;

$\frac{n}{2} \times \log_2 n \times \log_2 n$

complexity = $O(n \log_2^2 n)$

(8) function (int n)
{
if(n==1) return;
for(i = 1 to n) { —— n
for(j = 1 to n) { —— n^2
print ("*"); —— n^2
}
}
function(n-3); —— $T(n-3)$

function (n-3); —— $T(n-3)$
}

$T(n) = O(n^2) + T(n-3)$

$TC = O(n^3)$

(9) void

(10)

(9) void function (int n)
{
  for (i=1 to n){
    for (j=1; j<=n; j=j+i)
      print("*");
  }
}

$i = 1, 2, 3, 4$ ——

$j = 1, 3, 6, 10$ ——

$i = 1$ to, n times

$i = 2$ ~~$j=2$~~ to, $\frac{n}{2} + 1$ times

$i = 3$, $\frac{n}{3}$ times

$1 + \frac{n}{2} + \frac{n}{3} + \text{——} \text{——} 1$

<u>complexity $O(n \log n)$</u>

(10) n^k & c^n

Assume $k >= 1$ & $c > 1$ $c = ?$

for n^k growth rate $\propto k$

for c^n growth rate $\propto c$

set $n = 2$

$n = 2$

$n^k = 2^k$

$c^n = 2^2$

$c^2 > 2^k$

$c^2 > 2^k$

6

set $n=3$

$k=1$

$3^2=9$

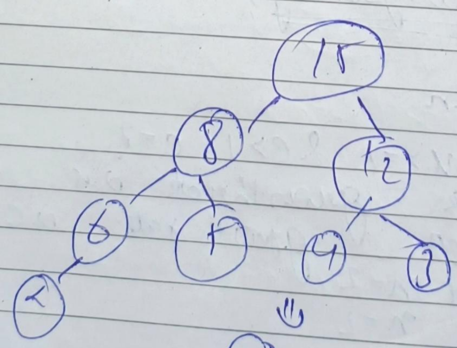
$2^1=2$

$9>2$ $\quad c^n$ grows faster than n^k

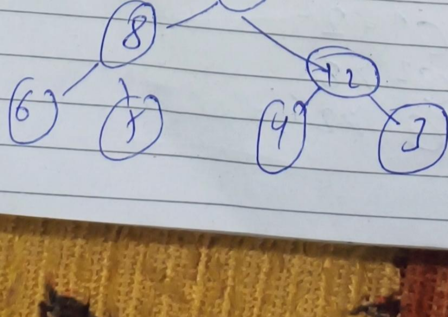
relationship $c^n > n^k$ holds for large values of n. $\quad n=2$

(11) The time complexity for extracting the min. element using 'extractMin()' depend on the underlying data structure. In the case of a binary heap, which is commonly used for priority queues, the complexity of is $O(\log n)$.

(12)



⇓



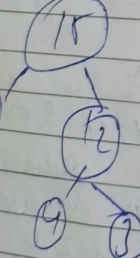
$n=3$
$k=1$

$=9$
$=2$

c^n grows faster than n^k
$n=2$

$c^n > n^k$ holds for large
of n.

...plexity for extracting the
... using 'extractMin()' depends
...erlying data structure. In the
...y heap, which is commonly
...iority queue, the complexity
...g n).



⇓