Date

Page

DAA Assignment-1

Ques1:- What do you understand by Asymptotic notations. Define different Asymptotic notation with example.

ans1: Asymptotic Notation is used to describe the running time of an algorithm - how much time an algorithm take with a given input, n.

Different types of notations:-

① Big O Notation

$$f(n) <= O(g(n))$$

Big O Notation refers an upper bounds on the growth rate of a function.

Example:- If an algorithm has time complexity of O(n), it means that the running time of an algorithm grows linearly with the size of input.

Date

Page

(2) Omega Notation (Ω) :-

$$f(n) >= g(n)$$

Omega notation represents a lower bound on growth rate of a function.

Example:- If an algorithm have time complexity of ~~$\Theta(n^2)$~~ $\Omega(n^2)$. It means its running time grows at least quadratically with size of input.

(3) Theta Notation (Θ):-

$$f(n) = \Theta(g(n))$$

Theta Notation represents a lower bound and upper bound, both on the growth rate of function.

Example:- If an algorithm have time complexity of $\Theta(n)$, it means running time grows linearly with size of input.

$f(n) < g(g(n))$

small o notation represents an upper bound that is not tight. It indicates that a function grows strictly slower than another function.

Ex:- If an Algorithm has time complexity of $o(n^2)$ it means that running time grows strictly slower than n^2.

(5) Small omega (ω)

$$f(n) > \omega(g(n))$$

Small omega notation represents a lower bound that is not tight. It indicates that function grows, strictly faster than another function

Example:- if an algorithm has a time complexity of $\omega(n)$, it means that the running time grows strictly faster than linear.

Date

Page

Q2: What should be time complexity of -

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

the loop continues until 'i' exceeds or equal to 'n'. Therefore the no. of iterations can be expressed as :-

$2^k \geq n$

log both sides

$k \geq \log_2(n)$

So, loop runs $O(\log(n))$.

Q3: $T(n) = \{ 3T(n-1)$ if $n > 0$, otherwise $1 \}$

1) Base case → $T(0) = 1$.

2) Recursive case → for $n > 0$, $T(n) = 3T(n-1)$

Expanding $T(n)$:-

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

$\rightarrow 3^2 T(n-2) = 3^3 T(n-3) = \ldots = 3^n T(0)$

substitute → $T(0) = 1$.

Date

Page

$T(n) = 3^n$

So, sol^n to recurrence relation is $T(n) = 3^n$.

So, sol^n to recurrence relation is $T(n) = 3^n$.

Time complexity is in recurrence relation is exponential in term of n.

Q4: $T(n) = \{2T(n-1) - 1$ if $n > 0$, otherwise $1\}$

1) Base case: $T(0) = 1$.

2) Recursive case:

for $n > 0$, the recurrence relation $T(n) = 2T(n-1)-1$

Now, let's expand $T(n)$

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

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

$2^3T(n-3) - 3 = \ldots .2^nT(0) -$

$(2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 1)$

The sum $2^{n-1} + 2^{n-2} + \ldots 1$ is a geometric series and its sum is $2^n - 1$.

Date

Page

Substitute the base case :-

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

$T(n) = 1$

$O(1)$

Q5: What should be time complexity of :-

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

After 1st iteration = $s = 1+2 = 3$

After 2nd iteration = $s = 3+3 = 6$

After 3rd iteration = $s = 6+4 = 10$

After Kth iteration = $1+2+3+4+5+\dots K$

$= \frac{K(K+1)}{2}$

So, finding value of K

$\frac{K(K+1)}{2} > n$

$K^2 + K > n$

$K^2 + K > 2n$

$K > \sqrt{n}$

Date

Page

Time complexity is $O(K)$

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

Q6: Time complexity of :-

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

Loop have condition $i*i \rightarrow i*i \leq n$ so, loop will continue until it become greater than root n.

count → take constant time.

$\boxed{O(\sqrt{n})}$

Q7: 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)
```

Date
Page

```
for (k=1; k<=n; k=k*2)
  counter;
}
```

Outer loop runs → $n/2$ times.
middle j loop runs → $O(\log n)$ times because j doubles in iteration.
inner loop also runs → $O(\log_2 n)$ times because K doubles in each iteration.

total no. of iterations = $O\left(\frac{n}{2} \cdot \log n \cdot \log_2 n\right)$

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

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

Qn: time complexity of :-

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

Date

Page

→ base case is when n=1, then function returns without further recursion.

→ Output outer loop runs from i=1 to n.
→ inner loop runs from j=1 to n

$O(n^2)$ since, loop inside loop.

the recursive call is made with the parameter 'n-3'. Assuming the initial value of 'n' is divisible by 3, the recursive calls approximately n/3 times before reaching the base case where 'n' becomes 1.

$T(n) = T(n-3) + O(n^2)$ → over all time complexity.

Q9:- time complexity of :-

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

Date

Page

In inner loop, variable 'j' is incremented by i in each iteration. The no. of iterations of the inner loop depends on value of 'i'. Let's denote the no. of iterations of inner loop for specific 'i' as $\frac{n}{i}$

Calculate total no. of iterations :-

$$\frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \ldots\ldots + \frac{n}{n}$$

Sum of h.p $\quad$ O(log n).

time complexity $\Rightarrow$ O(n log n)

Q10 for the function n^k and c^n what is the asymptotic relationship b/w these functions? Assume that $k >= 1$ and $c > 1$ are constant. find out the value of c & n_0 for which relation holds.

The asymptotic relationship b/w n^k and c^n depends on values of k and c.

1) If $k > 1$, then n^k grows faster than c^n. In this case, n^k is said to have a polynomial growth rate, c^n has an exponential growth rate. Asymptotic relation :- n^k is $O(c^n)$

Date

Page

2) If $K = 1$ then both n^K and c^n have same growth rate and they are considered to be in the same complexity class. Asymptotic relation $\boxed{n^K \text{ is } O(c^n)}$

3) If $0 < K < 1$ then c^n grows faster than n^K. In this case, n^K has a slower growth rate than c^n. Asymptotic relation $\boxed{n^K \text{ is } O(c^n)}$

To find the value of c and no where the relationship holds, you need to consider the limits:

1) for n^K growing faster ($K > 1$):

- $\lim_{n \to \infty} \frac{n^K}{c^n} = 0$
- In this case any $c > 1$ will work and no can be any positive constant.

2) for c^n growing faster ($0 < K < 1$):

- $\lim_{n \to \infty} \frac{n^K}{c^n} = \infty$
- In this case, c must be greater than 1 and no can be any positive constant.

Date _____

Page _____

Q11: What will be the time complexity for extraction()? Given min heap of n nodes extractMin(): return the min element for the heap.

```
void heapifydown (int heap[], int n, int i) {
 int smallest = i;
 int left = 2*i+1;
 int right = 2*i+2;
 if (left < n && heap [left] < heap [smallest])
  smallest = left;
 if (right < n && heap[right] < heap [smallest])
  smallest = right;
 if (smallest != i) {
 swap (&heap [i], &heap [smallest]);
  heapifydown (heap, n, smallest);
  }
}
int extractMin (int heap [], int *n) {
   if (*n <= 0)
    return -1;
  swap (&heap[0], &heap[*n-1]);
  int minElement = heap[*n-1];
   (*n)--;
  heapifydown (heap, *n, 0);
   return minElement;
}
```

Date

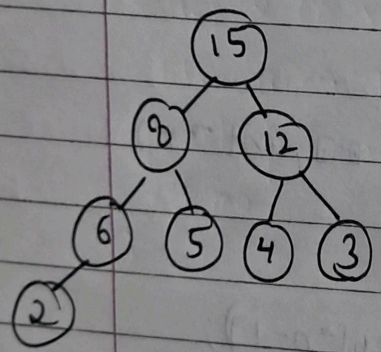
Page

```
int main() {
int heap[] = {4, 8, 15, 2, 10, 20, 6, 12};
 int n = sizeof(heap) / sizeof(heap[0]);

printf("Min heap before extraction");
for (int i = 0; i < n; i++) {
  printf("%d", heap[i]);
}
 return 0;
}
```

Time complexity - O(log n)

Q12:- Given the following max heap, delete & show the heap-




```
#include <iostream>
#include <vector>

void swap (int &a, int &b) {
  int temp = a;
   a = b;
```

Date _____

Page _____

```
    b = temp;
}
void heapifyDown (std::vector <int> &heap,
                  int index) {
  int leftchild = 2 * index + 1;
  int rightchild = 2 * index + 2;
  int largest = index;

if (leftChild < heap.size() && heap [leftChild] >
        heap [largest])
  largest = leftChild;

if (rightchild < heap.size() && heap [rightchild] >
    heap [largest])
  largest = rightchild;

 if (largest != index) {
   swap (heap[index], heap [largest]);
   heapifyDown (heap, largest);
   }
 }

void deleteMax (std::vector <int> &heap) {
    if (heap.empty()) {
  std::cout << "heap is empty" << std::endl;
   return;
}
```

Date

Page

```
    heap[0] = heap.back();
    heap.pop_back();
    heapifyDown(heap, 0);
}

void printHeap(const std::vector<int>& heap) {
    for (int i : heap) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> maxHeap = {15, 8, 12, 6, 5, 4, 3, 2};
    std::cout << "Max heap before deletion: ";
    printHeap(maxHeap);
    deleteMax(maxHeap);
    std::cout << "Max heap after deletion: ";
    printHeap(maxHeap);
    return 0;
}
```