1. Which of the following is the time complexity of Binary search in a sorted array of size(n) ?

→

$$T(n) = T\left(\frac{n}{2}\right) + c$$

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + c$$

$$T(n) = \left[T\left(\frac{n}{4}\right) + c\right] + c$$

$$T(n) = T\left(\frac{n}{4}\right) + 2c.$$

$$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + c$$

$$T(n) = T\left(\frac{n}{8}\right) + 3c.$$

$$\therefore T(n) = T\left(\frac{n}{2^k}\right) + kc.$$

Let, consider, $n = 2^k$

$$\log n = k.$$

$$\therefore T(n) = T(1) + \log n \cdot c$$

$$\therefore \boxed{O(\log n)}.$$

2. What is the space complexity of the iterative version of Binary search.

→ ~~space complexity of~~ $O(1)$ because no extra space is Required.

3. For $n = 10 \rightarrow 10^2 = 100$

For $n = 50 \rightarrow 50^2 = 400$ ~~2500~~

for $n = 100 \rightarrow 100^2 = 10000$.

for $n = 500 \rightarrow 500^2 = 250000$.

∵ ~~50~~ for $n = 500$, the largest value of n^2 exists.

So, the answer is $\boxed{500}$.

4. function solve (array)

for i from 1 to length (array)

loop runs 'n' time as the length of 'n'

for j from 1 to length (array)

loop runs 'n' times as length is 'n'

print (array[i] + array[j]).

Takes constant time O(1).

∵ Nested loop ∴ Time Complexity $O(n^2)$.

for input array space Required O(n).

for loop variable space Required O(1).

∴ Space Complexity O(n).

5. function Solve (n)

Sum = 0

for i from 1 to n → 'i' takes constant space O(1)

Loop runs 1 to n. ∴ O(n).

Sum = Sum + i

Take constant time O(1)

return sum. → 'sum' takes constant space O(1)

∴ space complexity O(1)

∴ Time complexity = O(n)

6.

```
function solve (array)
    for i from 1 to length (array)
        if array[i] % 2 == 0
            print (arr[i])
        else
            print (arr[i] * 2)
```

Loop Runs 'n' time. ∴ O(n). as length is n.

Take Constant time ∴ O(1)

Take Sp... O(n). → i, else Constant Spa... O(

∴ Space complexity O(n).

∴ Time complexity O(n).

7.

```
function solve (base, exponent)
    if exponent == 0
        return 1
    half = power (base, floor (exponent/2))
    return half * half
    else
        return half * half * base
```

O(1) O(1) Recursive call $T(n) = T(\frac{n}{2}) + c$ O(log n).

∴ Time complexity = O(log n).

Space complexity = O(log (exponent)).

8.

```
function sumCalculator (n)
    sum = 0
    i = 1
    while i < n
        sum = sum + i
        i = i * 2
    return sum.
```

O(1) ← for each iteration the loop doubles the value. i = 1, 2, 4, 8 ($2^0, 2^1, 2^2, 2^3$)

$2K = n$

$K = \log n$

∴ Time complexity = O(log n).

NO extra space Required.

∴ space complexity = O(1)

9.

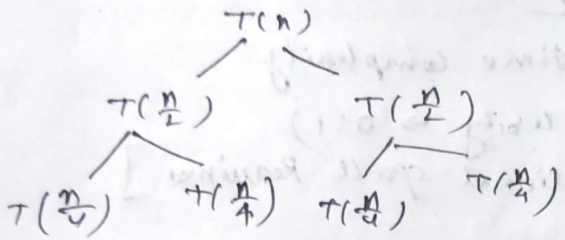
```
function solve(n)
O(1) ← [if n == 0
           return 1 ] → O(1) space
       result = 1
For loop runs 1 to n ∴ O(n). ← [for i from 1 to n
                                   result = result * i ] - O(1) space.
       return result
```

∴ Time Complexity = O(n).

No extra space.

∴ Space complexity = O(1).

10.



∴ Time complexity = O(log n)

∴ Space complexity = O(1)

11.

```
function solve(n)
O(1) ← [if n <= 1
           return 1
       sum = 0
loop runs 'n' times ∴ O(n) ← [for i from 1 to n
                                 sum = sum + solve(i-1)
       return sum.
```

∴ Time complexity = O(n)

∴ Space complexity = O(n).

12.

```
function solve(array, k)
O(K) ← for i from 1 to k
O(n) ← for j from 1 to length(array)
           print(array[j]).
```

∴ Nested loop ∴ Time complexity = O(n * k)

Space complexity = O(1)

13. i < array(length)

j = i+1, j < array(length)

i = 0, loop runs n-1 :

i = 2, " " n-2

⋮

i = n-2 loop runs 1

i = n-1 loop " 0.

$\therefore T(n) = (n-1) + (n-2) + \cdots + 1$

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

$O(n^2)$ → time complexity

space complexity → O(1)

[∵ NO additional space Required]

14.

15. Time complexity : $O(n^2)$
space complexity : $O(1)$.

16. function solve (array)
result = 0
$O(n)$ ← [for i from 1 to length(array)
result = result + array[i]
$O(n)$ ← [for i from 1 to length(array)
result = result − array[i]
return result.

∴ Time complexity = $O(n)$.
∴ space complexity = $O(1)$
[∵ No additional space required].

17. $T(n) = 2T(n-1) + O(1)/c$

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

$T(n) = 2[2T(n-2) + c] + c$

$T(n) = 4T(n-2) + 3c = 2^2T(n-2) + (2^2-1)c$

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

$T(n) = 4[2T(n-3) + c] + 3c.$

$= 8T(n-3) + 7c.$

$= 2^3T(n-3) + (2^3-1)c$

$= 2^kT(n-k) + (2^k-1)c$

$n - k = 0$

$n = k.$

$T(n) = 2^nT(1) + 2^n$

∴ Time Complexity = $O(2^n)$.

∴ Space complexity = $O(n)$

18. Time Complexity = $O(m1 * m2 * m1)$
Space complexity = $O(m1 * m2)$.

19. Time complexity = $O(n^3$ [illegible] $)$
space Complexity = $O(1)$.

20. Time Complexity = $O(2^n * n)$.
space complexity = $O(n)$.

21. Time complexity = $O(n)$
Space complexity = $O(n)$.

22. Time complexity = $O(n^2)$
space complexity = $O(1)$

23. Time Complexity = $O(n^2)$
space complexity = $O(1)$.

24. Time complexity = $O(n)$
space complexity = $O(1)$

25. Time complexity = $O(n \times m)$
space complexity = $O(n \times m)$.

26. Time complexity = $O(n)$
space complexity = $O(1)$

[illegible].

27.

d.

26: function solve (array)
count = 0
O(n) ← for i from 1 to length (array)
if array[i] > 0
count = count + 1
return count.

∴ TC = O(n) ∴ SC = O(1).

27. function solve (array)
result []
Take O(n). ← [for i from 1 to length (array)
if not result.contains (array[i])
O(1) ← result.append (array[i])
return result.

∴ TC = O(n)
∴ SC = O(n) → as result can stores n unique element.

28. TC = O(log n)
SC = O(1)

29. $1 \longrightarrow n$
$i \longrightarrow 1$

$\frac{n(n+1)}{2} = O(n^2)$.

TC = $O(n^2)$ SC = O(1)