**Time complexity and space complexity :**

**[answers with Explanation]**

1 ) **What is the time, and space complexity of the following code:**

int a = 0, b = 0;

for (i = 0; i < N; i++) {

a = a + rand();

}

for (j = 0; j < M; j++) {

b = b + rand();

}

**Options:**

O(N \* M) time, O(1) space

O(N + M) time, O(N + M) space

O(N + M) time, O(1) space

O(N \* M) time, O(N + M) space

**Ans:**

3. O(N + M) time, O(1) space

**Explanation**: The first loop is O(N) and the second loop is O(M). Since N and M are independent variables, so we can’t say which one is the leading term. Therefore Time complexity of the given problem will be O(N+M).

Since variables size does not depend on the size of the input, therefore Space Complexity will be constant or O(1)

2 ) 2. What is the time complexity of the following code:

int a = 0;

for (i = 0; i < N; i++) {

for (j = N; j > i; j--) {

a = a + i + j;

}

}

**Options:**

O(N)

O(N\*log(N))

O(N \* Sqrt(N))

O(N\*N)

**Ans:**

4. O(N\*N)

**Explanation**:

The above code runs total no of times

= N + (N – 1) + (N – 2) + … 1 + 0

= N \* (N + 1) / 2

= 1/2 \* N^2 + 1/2 \* N

O(N^2) times.

3. What is the time complexity of the following code:

int i, j, k = 0;

for (i = n / 2; i <= n; i++) {

for (j = 2; j <= n; j = j \* 2) {

k = k + n / 2;

}

}

**Options:**

O(n)

O(N log N)

O(n^2)

O(n^2Logn)

**Ans:**

2. O(nLogn)

**Explanation**: If you notice, j keeps doubling till it is less than or equal to n. Several times, we can double a number till it is less than n would be log(n).

Let’s take the examples here.

for n = 16, j = 2, 4, 8, 16

for n = 32, j = 2, 4, 8, 16, 32

So, j would run for O(log n) steps.

i runs for n/2 steps.

So, total steps = O(n/ 2 \* log (n)) = O(n\*logn)

4. What does it mean when we say that an algorithm X is asymptotically more efficient than Y?

**Options:**

X will always be a better choice for small inputs

X will always be a better choice for large inputs

Y will always be a better choice for small inputs

X will always be a better choice for all inputs

**Ans:**

2. X will always be a better choice for large inputs

**Explanation**: In asymptotic analysis, we consider the growth of the algorithm in terms of input size. An algorithm X is said to be asymptotically better than Y if X takes smaller time than y for all input sizes n larger than a value n0 where n0 > 0.

5. What is the time complexity of the following code:

int a = 0, i = N;

while (i > 0) {

a += i;

i /= 2;

}

**Options:**

O(N)

O(Sqrt(N))

O(N / 2)

O(log N)

**Ans:**

4. O(log N)

**Explanation**: We have to find the smallest x such that ‘(N / 2^x )< 1 OR 2^x > N’

x = log(N)

6. Which of the following best describes the useful criterion for comparing the efficiency of algorithms?

**Options:**

Time

Memory

Both of the above

None of the above

**Ans:**

3. Both of the above

**Explanation**: Comparing the efficiency of an algorithm depends on the time and memory taken by an algorithm. The algorithm which runs in lesser time and takes less memory even for a large input size is considered a more efficient algorithm.

7. How is time complexity measured?

**Options:**

By counting the number of algorithms in an algorithm.

By counting the number of primitive operations performed by the algorithm on a given input size.

By counting the size of data input to the algorithm.

None of the above

**Ans:**

2. By counting the number of primitive operations performed by the algorithm on a given input size.

8. What will be the time complexity of the following code?

for(var i=0;i<n;i++)

i\*=k

**Options:**

O(n)

O(k)

O(logkn)

O(lognk)

**Ans:**

**Ans:**

3. O(logkn)

**Explanation**: Because the loop will run kc-1 times, where c is the number of times i can be multiplied by k before i reaches n. Hence, kc-1=n. Now to find the value of c we can apply log and it becomes logkn.

9. What will be the time complexity of the following code?

int value = 0;

for(int i=0;i<n;i++)

for(int j=0;j<i;j++)

value += 1;

**Options:**

n

(n+1)

n(n-1)

n(n+1)

**Ans:**

3. n(n-1)

**Explanation**: First for loop will run for (n) times and another for loop will be run for (n-1) times as the inner loop will only run till the range i which is 1 less than n , so overall time will be n(n-1).

10. Algorithm A and B have a worst-case running time of O(n) and O(logn), respectively. Therefore, algorithm B always runs faster than algorithm A.

**Options:**

True

False

**Ans:**

False

**Explanation**: The Big-O notation provides an asymptotic comparison in the running time of algorithms. For n < n0​​, algorithm A might run faster than algorithm B, for instance.

11) What is the time complexity of the following code?

def function(N,M):

counter=0

for i in range(N):

for j in range(M):

counter+=1

print(counter)

**Ans:** Time complexity: O(N x M).

Here the outer loop runs for N times. In the ith iteration, the inner loop runs for M times and hence the time complexity is O(M). As the loop runs for N times, the time complexity is N x M i.e. O(N x M)

12. What is the time complexity of the following code?

def fun(n):

for i in range(n):

print(pow(i,n))

**Ans:** time complexity : O(n x log(n))

Here the pow(i,n) takes log(n) time complexity as it uses binary exponentiation.

And the for loop runs for N times.

Thus time complexity is O(n x log(n)).

13) What is the time complexity of the following code?

def f(n):

if n == 0 or n == 1:

return 1

return f(n - 1) + f(n - 2)

**Ans:** Time complexity: O(2n)

This is because fib(i) will call fib(i-1) and fib(i-2). But the fib(i) isn’t stored. So when we calculate fib(i+1), it will call fib(i) and so it will be calculated repeatedly for a number of times.

Here, the recurrence relation is: T(n) = T(n-1) + T(n-2) <= 2 x T(n-1)

Thus T(n) <= 2 x T(n-1) <= 4 x T(n-2) <= … <= 2n-1 x T(n-(n-1))

Since T(1) is constant. T(n) has a upper bound of O(2n) i.e. exponential time complexity.

Similarly we can find the lower bound by using T(n)= T(n-1) + T(n-2) >= 2 x T(n-2)

T(n) >= 2 x T(n-2) >= 4 x T(n-4) … >= 2(n-1)/2 x T(n- (n-1).

Thus the lower bound of T(n) is also exponential.

Thus the time complexity of fib(n) is exponentially increasing i.e. T(n) = O(2n).

Must Read Lower Bound in C++

14) What are the time complexity and space complexity of the following code?

def fun(n,m):

arr=[[0]\*m for i in range(n)]

for i in range(n):

for j in range(m):

k=1

while k<n\*m:

k\*=2

**Ans:**

Time complexity : O(n\*m\* log (n\*m))

Space complexity: O(n\*m)

As we are making an array of size O(n\*m) and no other extra space, the space complexity is O(n\*m).

Here the first loop runs for n times.

The inner loop runs for m times.

The k gets multiplied by 2 every time in every iteration. Thus it will be greater than n\*m in log2(n\*m) iterations.

Thus time complexity is O(n\*m\* log (n\*m)).

15) What are the time complexity and space complexity of the following code?

# given N>0

def recursion(N):

if N==0:

return

print(N)

recursion(N-1)

**Ans:** Time complexity: O(N), Space complexity: O(N)

Here the recursion will run for N times as the base case is N=0. The recursion will call itself as many times until the value of N becomes 0.

Thus N times calling results in O(N) time complexity and O(N) space complexity

Alternative method: Here we can see that the print statement gets executed for N times. This indicates that the recursion is called for N times thus time complexity is O(N) and space complexity is O(N).

Must Read Recursion in Data Structure

16) What is the time complexity of the following code?

def fun(N):

counter=1

for i in range(1,N+1):

for j in range(1,i+1):

counter+=1

print(counter)

**Ans:** O(N2)

Here i runs from 1 to N.

So in the ith iteration, the inner loop will run from 1 to i.

So our time complexity will be 1+ 2+ 3 + 4 + … N = N x (N+1) /2

We can thus write N x (N+1) /2 as O(N2).

Alternative method: Here we can print counter for different n and we can easily find out that counter will be N x (N+1)/2

17) What is the time complexity for the following program?

def bisect\_left(arr,item):

# arr is sorted array

low=0

high=len(arr)

while low<high:

mid=low+(high-low)//2

if arr[mid]>=item:

high=mid

else:

low=mid+1

return low

**Ans:**

Time complexity : O(log(length of array))

Let N be the length of the array.

Here we are using binary search on the array “arr” to search the element item. Here in each search half of the array is eliminated. Thus the time complexity can also be written as

T(N) = T(N/2) + 1.

Now T(N/2) = T(N/4) + 1 substituting it in the above equation we get T(N) = T(N/4) + 2.

Thus T(N) = T(N/2k) + k.

Let us find the smallest integer k such that 2k >=N, thus k >= log2(N). As k is integer k= ceil(log2(N))

Thus T(N) = T(1) + ceil(log2(N)). Since T(1) = constant , T(N)= constant+ ceil(log2(N)).

Thus we get T(N) =O( log2(N))

18) What is the time complexity for the following program?

void fun(int n)

{

int k = 0;

for (int i = n; i > 0; i = i / 2)

{

for (int j = 0; j < i; ++j)

{

++k;

}

}

cout << k << endl;

}

**Ans: O(n)**

Here the values of i will be n, n/2, n/4, n/8 and so on until becomes 1.

After each iteration in the outer for loop, the values of i get divided by 2.

Thus the inner loop will run for n + n/2 + n/4 + n/8 + n/16 + and so on until it becomes 0.

From the above sequence, we can take out n common. So our sequence is n x (1+ ½ + ¼ + ⅛ + 1/16 … up to 0).

Now in 1+ ½ + ¼+ ⅛ + … forms Geometric progression with first term as a = 1 and common ratio “r” = ½.

Thus the 1+ ½ + ¼+ ⅛ + is upper bounded by a/(1-r) = 1/(1-½) = 2.

So, we get the upper bound of the sequence as 2 x n.

Thus time complexity will be O(N).

19) What is the time complexity for the following program?

def calc(arr,N):

j=0

for i in range(N):

while j<N and arr[i]<arr[j]:

j+=1

**Answer: O(N).**

Here the inner loop will be executed for a maximum of N times. The reason is that the j is incremented every time in the while loop. The condition in the while loop will be executed for 2 x N time because the outer for loop runs for N time and while loop runs for N times.

Thus the time complexity is O(N).

20) What are the time and space complexity for the following program?

maxa=50

dp=[-1 for i in range(maxa+1)]

def fib(n):

if dp[n]!=-1:

return dp[n]

if n<=1:

return n

dp[n]=fib(n-1)+fib(n-2)

return dp[n]

print(fib(maxa))

**Ans:**

Time complexity: O(n) and space complexity: O(n)

Here the dp array is used for memorization. So it stores the fib(n) after it was calculated. Thus in calculating fib(n), the dp[n-1] and dp[n-2] are already calculated, so fib(i) takes O(1) time after its last two numbers are calculated. So fib(i) = constant + fib(i-1). Thus time complexity is O(N).

We are making an array of size n. Similarly, the maximum depth of the recursion will be n.

Thus the time and space complexity both are O(N).