**Time Complexity**

Contents

[Guidelines for Asymptotic Notations 1](#_Toc118021661)

[1. General Statements 1](#_Toc118021662)

[2. If-then-else statements 1](#_Toc118021663)

[3. Loops 1](#_Toc118021664)

[4. Nested loops 2](#_Toc118021665)

[5. Consecutive Statements 2](#_Toc118021666)

[6. Logarithmic Complexity 3](#_Toc118021667)

[Practice Questions 3](#_Toc118021668)

# Guidelines for Asymptotic Notations

1. General Statements

* X = 0;
* X = X + 1;
* length () == 0;
* j==0? j++: j— etc.

Time Complexity – O (1)

## If-then-else statements

Worst case running time: if-then or else part whichever has the largest time complexity

// condition check: constant time

if (length () == 0)

return false;

else {

// else part: (constant + constant) \* n

for n = 0 to n< length {

// another if: constant + constant (no else part)

if (! list[n]. equals (otherList.list[n]) )

return false;

} }

Time Complexity – O(n)

## Loops

// executes n times

for i from 1 to n

m = m + 2; // constant time c

i++;

Time Complexity – O (n)

## Nested loops

// outer loop executed n times

for i from 1 to n {

i++;

// inner loop executed n time

for j from 1 to n {

k = k+1; // constant time

j++

} }

Time Complexity – O (n2)

## Consecutive Statements

Example 1. Total running time is the product of the sizes of all the loops.

x = x + 1; // constant time

// executed n times

for i from 1 to n {

m = m + 2; // constant time c

i++;

}

// outer loop executed n time

for i from 1 to n {

// inner loop executed m time

for j from 1 to m {

k = k + 1; // constant time

j++;

} }

Time Complexity – O(n\*m)

Example 2.

int a = 0, b = 0;

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

a = a + rand(); => O(1)

}

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

b = b + rand(); => O(1)

}

Time Complexity – O(N+M)

## Logarithmic Complexity

for i from 1 to n

i = i \* k

Time Complexity – O (logk(n))

for i from 1 to n

i = i / 2

Time Complexity – O (log₂(n))

# Practice Questions

1. if(i>j) {

j==0? j++: j--;

}

Time Complexity – O (1)

Explanation: A Conditional statement takes O (1) time to execute. there are only two conditional statements. So, it takes constant time to execute.

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

i\*=k

Time Complexity – O (logk(n))

Explanation: Because loops for the kn-1 times, so after taking log it becomes logk(n).

1. int a = 0, i = N;

while (i > 0) {

a += i;

i /= 2;

}

Time Complexity – O (log₂(n))

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

x = log(N)

1. int a = 0, b = 0;

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

a = a + rand ();

}

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

b = b + rand ();

}

Time Complexity – O (N+M)

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).

1. int i, j, k = 0;

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

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

k = k + n / 2;

}

}

Time Complexity – 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).

1. for (int i = 0; i < n; i++) => O (n)

for (int j = i; j > 0; j--) => O (n)

cout << i << j;

Time Complexity – O (n2)

Explanation: Outer loop will run (n) times, and for each value of i, the inner loop will run (i) times. So, the total number of operations is n\*i, and the maximum value of i will be n, so the overall time complexity of the problem will be n O(n\*n).

1. for (int i=n/2; i<=n; i++) { => O (n)

for (int j=1; j<=n; j=j\*2) { => O (log n)

cout<<i<<j<<endl;

}

}

Time Complexity – O (nlogn)

Explanation: Outer loop will run (n/2) times, and for each value of i, the inner loop will run (log n) times. So, the total number of operations is n/2\*log n. We can say the time complexity of the above code is O (n\*log n).

1. var value = 0;

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

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

value += 1;

Time Complexity – O (n2)

Explanation: First for loop will run for (n) times and another for loop will be run for (n-1) times so overall time will be O (n2).

1. for (i=1; i<n; i=i\*2) { => O (log n)

for (j=i; j<n; j++) => O (n)

print("x");

}

Time Complexity – O (nlogn)

Explanation: First for loop will run for (log n) times and another for loop will be run for (n) times so overall time will be O (nlogn).

1. int a = 0;

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

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

a = a + i + j;

}

}

Time Complexity – O (*N*2)

Explanation: First for loop will run for (N) times and another for loop will be run for (N) times so overall time will be O (*N*2).

1. int fun (int n) {

int count = 0;

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

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

count += 1;

return count;

}

Time Complexity – O (n)

Explanation: in general, it is not the worst of the two loops, but the total number of iterations of the innermost loop. Thus, as pointed out in this answer, we need to perform the respective sum which gives the total of O(n). Note however that the usual procedure is to take the product of the complexities of each loop (thus not the max), but this results here in a too loose bound, because the number of iterations of the innermost loop varies depending on the current value of i.

n =16

i = 16, j = 0 -16 => Highest **n** no. of time loop will run

i = 8, j = 0-8

i = 4, j = 0-4

i = 2, j = 0-2

i = 1, j = 0-1

i = 0, j = X