Time Complexity

Assume that someone in your class has stolen your favourite chocolate. You need to find that chocolate. Below are some ways to find it.

By asking each and every person in the class, which means if n number of students are there, then you need to ask n persons. Hence, the complexity is O(n).

By asking only one of the students in the class and if he/she doesn't have that chocolate then ask the same person about the remaining students. So, to each and every person you are asking about the same person and also about the remaining persons. Hence, the complexity is twice when compared to the previous case. i.e. O(n2). The above two ways seem to be difficult.

So, say if you are divide the entire class into two halves and ask whether the chocolate is on the right side or the left side. If it is on the right side, then divide the number of people on the right side again into two and repeat the same process. Here, the number of enquiries is decreasing exponentially. Hence, the complexity is O(log n).

1)

#include <stdio.h>

int main()

{

int a = 4;

int b = 6;

int c;

c = a + b;

printf(%d, c); }

Time Complexity Calculation: The time complexity of the above-given program is O(1), as this program consists of only assignment, arithmetic operations and all those will be executed only once.

2)

int count(int arr[], int n)

{

int sum = 0, i;

for(i = 0; i < n; i++) //Control statement

{

sum = sum + arr[i];

}

return sum;

}

Time Complexity Calculation: In the above-given snippet, we have a control statement which executes for n times. Along with that we also have operations like assignment, arithmetic and a return statement. Hence, the time complexity is O(n +

3).

For larger values of n, the constant values become negligible. So if a program consists of a control statement, then the complexities of assignment, arithmetic, logical and return statements can be ignored.

Hence, the final time complexity of the above-given snippet is O(n).

3)

int i,j, n = 8;

for (i = 1; i <= n; i++)

{

for (j = 1; j <= n; j++)

{

printf("Anand");

} } }

Time Complexity Calculation: In the above snippet, the first & the second for loops get executed n times individually. So the time complexity accounts to n\*n = O(n2)

4)

while(low<=high)

{

mid=(low+high)/2;

if(n<arr[mid]) high=mid-1;

elseif(n>arr[mid]) low=mid+1;

elsebreak;

}

Time Complexity Calculation: This is the algorithm of binary search. It breaks the given set of elements into two halves and then searches for a particular element. Further, it keeps dividing these two halves into further halves until each individual element is a set. All such algorithms which work on the principle of recursive division of elements into halves will have a O(Log n) complexity.

:Algorithm Best Case Complexity :Average Case Complexity :Worst Case Complexity

:Bubble Sort O(n)  O(n²) O(n²)

:Selection Sort O(n²) O(n²) O(n²)

SPACE COMPLEXITY

Example #1

#include<stdio.h>

int main()

{

int a = 5, b = 5, c;

c = a + b;

printf("%d", c); }

Explanation: Do not misunderstand space-complexity to be 1364 Kilobytes as shown in the output image. The method to calculate the actual space complexity is shown below.

In the above program, 3 integer variables are used. The size of the integer data type is 2 or 4 bytes which depends on the compiler. Now, lets assume the size as 4 bytes. So, the total space occupied by the above-given program is 4 \* 3 = 12 bytes. Since no additional variables are used, no extra space is required.

Hence, space complexity for the above-given program is O(1), or constant.

Example #2

#include <stdio.h>

int main()

{ int n, i, sum = 0;

scanf("%d", &n);

int arr[n];

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

scanf("%d", &arr[i]);

sum = sum + arr[i]; }

printf("%d", sum); }

Explanation:

In the above-given code, the array consists of n integer elements. So, the space occupied by the array is 4 \* n. Also we have integer variables such as n, i and sum. Assuming 4 bytes for each variable, the total space occupied by the program is 4n + 12 bytes. Since the highest order of n in the equation 4n + 12 is n, so the space complexity is O(n) or linear.

Big O Notation

Space Complexity details

O(1) - Constant Space Complexity occurs when the program doesn’t contain any loops, recursive functions or call to any other functions.

O(n) - Linear space complexity occurs when the program contains any loops.

Hash map/table:

Consists of key and data pairs.Usually key indicates the array index(address) where the data is stored.

How to calculate the hash key?

Let's take hash table size as 7.

size = 7

arr[size];

Formula to calculate key is,

key = element % size

In that key(array index) store the value.

After calculating key(of that array index) insert the element.

What if we insert an element say 15 to existing hash table?

Insert : 15

Key = element % key

Key = 15 % 7

Key = 1

But already arr[1] has element 8 !

Here, two or more different elements pointing to the same index under modulo size. This is called collision.

Collision Avoidance

Linear Probing

Separate Chaining

Linear Probing

Calculate the hash key. key = data % size;

If hashTable[key] is empty, store the value directly. hashTable[key] = data.

If the hash index already has some value, check for next index.

key = (key+1) % size;

If the next index is available hashTable[key], store the value. Otherwise try for next index.

Do the above process till we find the space.

Seperate Chaining

If the hash index already has some value, check for next index.

Create a linked list to that index(address).

Which would mostly represent Arrays of Arrays.