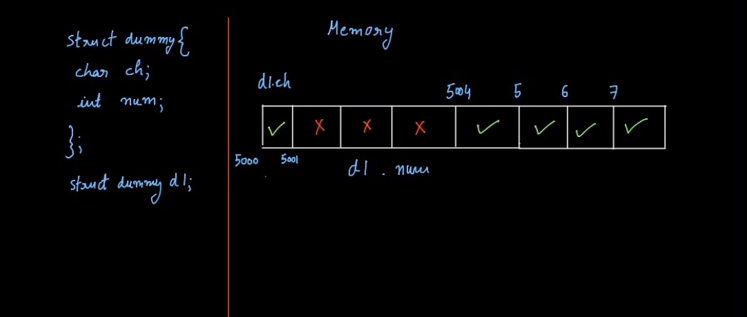
What is the size of a char struct in C?

Structure Padding in C:  
  
The structure padding is automatically done by the compiler to make sure all its members are byte aligned. Here 'char' is only 1 byte but after 3 byte padding, the number starts at 4 byte boundary. For 'int' and 'double', it takes up 4 and 8 bytes respectively.



Example1:

Struct{

Double..8bytes

Char…..1bytes

}

Total 16bytes

8+1+7

Example2:

Struct{

Int…4bytes

Double..8bytes

Char…..1bytes

}

Total 24bytes

4+4+8+1+7

TIME COMPLEXITY OF STATEMENTS

How many times statement is getting executed why? Efficiency of the program

TYPE 1:

For (i=0;i<n;i++) -------- n+1

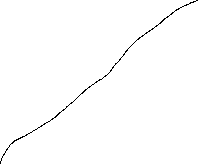
{

Statement -------n+1

}

Polynomial therm f(n)=n+1

Which is O(n) big o of n order of n



10 20 30



#include <stdio.h>

int main() {

int i, n = 5;

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

printf("%d\n", i);

printf("final i : %d", i);

return 0;

}

```

Now, let's analyze the time complexity of this corrected code:

1. Variable Initialization: `int i, n = 5;` - This step takes constant time and can be considered O(1).

2. The for loop: `for (i = 0; i < n; i++)` - This loop iterates from `i = 0` to `i = n-1`, incrementing `i` by 1 in each iteration. The loop runs exactly `n` times. Therefore, the time complexity of the loop is O(n).

3. Inside the loop, we have `printf("%d\n", i);`, which involves printing the value of `i`. The `printf` function typically has a time complexity of O(1) because it performs a constant number of operations regardless of the value being printed.

4. After the loop, we have `printf("final i : %d", i);`, which again involves printing the value of `i`. This `printf` function also has a time complexity of O(1).

So, the overall time complexity of the code is dominated by the loop, which is O(n). In this specific case, with `n` set to 5, the loop will run 5 times, so the code will have a time complexity of O(5), which simplifies to O(1) since it's a constant number of iterations.

```c

#include <stdio.h>

int main() {

int i, n = 5;

for (i = 1; i < n; i += 2) {

// Your statement here

}

return 0;

}

```

Now, let's analyze the time complexity of this corrected code:

1. Variable Initialization: `int i, n = 5;` - This step takes constant time and can be considered O(1).

2. The for loop: `for (i = 1; i < n; i += 2)` - This loop initializes `i` to 1 and increments it by 2 in each iteration until `i` is less than `n`. This means that the loop will run approximately `n / 2` times since it skips every other number. Therefore, the time complexity of the loop is O(n/2).

3. Inside the loop, you have the statement, which we assume takes constant time and can be considered O(1).

So, the overall time complexity of the code is O(n/2), which simplifies to O(n) in big O notation.

F(n)=n/2

Degree of polynomial is n .so n/ anything is n .so here also O(n).

So irrespective of iterations time is gonna be the same O(n).

NESTED LOOP:

For (i=0;i<n;i++)……n+1

{

For(j=0;j<n;j++)….n\*(n+1)

{

Statement;…n\*n------n square

}

}

Time complexity = O(n^2)

Explanation:

The time complexity of a nested loop like the one you've described, where you have a loop inside another loop, can be analyzed by multiplying the number of iterations each loop performs. In this case, you have two nested loops:

1. The outer loop runs from `i = 0` to `i < n` with `i` incrementing by 1 in each iteration. It performs `n` iterations.

2. The inner loop runs from `j = 0` to `j < n` with `j` incrementing by 1 in each iteration. It also performs `n` iterations for each iteration of the outer loop.

Now, to find the total number of iterations or the time complexity, you multiply the number of iterations of the outer loop by the number of iterations of the inner loop:

Total iterations = (Number of iterations of outer loop) \* (Number of iterations of inner loop)

Total iterations = n \* n

As you can see, the total number of iterations is `n \* n`, which is equivalent to `n^2`. Therefore, the time complexity of this nested loop is O(n^2).

In big O notation, we focus on the dominant term or the term with the highest growth rate, and in this case, it's `n^2`. So, we say that the time complexity is O(n^2), which means that the algorithm's runtime grows quadratically with the size of the input (`n`).

This type of time complexity is common in algorithms that involve nested loops, and it's important to be aware of it because it can help you understand how the algorithm's performance scales with larger input sizes. A quadratic time complexity like O(n^2) can become inefficient for large values of `n`.

Example2:

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

For(j=0;j<I;j++){

statement

}

Explanation:

The time complexity of the nested loops you've described, where you have an outer loop and an inner loop, can be analyzed similarly to the previous example. In this case, you have:

1. The outer loop runs from `i = 0` to `i < n` with `i` incrementing by 1 in each iteration. It performs `n` iterations.

2. The inner loop runs from `j = 0` to `j < i` with `j` incrementing by 1 in each iteration. The number of iterations for the inner loop depends on the value of `i` because it goes up to `i`.

To find the total number of iterations or the time complexity, you need to consider the cumulative effect of the inner loop for each value of `i`. Let's break it down:

- When `i = 0`, the inner loop runs 0 times.

- When `i = 1`, the inner loop runs 1 time.

- When `i = 2`, the inner loop runs 2 times.

- When `i = 3`, the inner loop runs 3 times.

- ...

- When `i = n-1`, the inner loop runs `n-1` times.

To calculate the total number of iterations, you sum up the number of iterations of the inner loop for each value of `i` from 0 to `n-1`:

Total iterations = 0 + 1 + 2 + 3 + ... + (n-1)

This is an arithmetic series, and you can calculate its sum using the formula for the sum of an arithmetic series:

Total iterations = (n-1) \* n / 2

Now, in big O notation, we focus on the dominant term or the term with the highest growth rate. In this case, the dominant term is `(n-1) \* n / 2`, which simplifies to `(n^2 - n) / 2`.

Therefore, the time complexity of this nested loop is O(n^2), as the `(n^2)` term dominates the time complexity, and we can ignore constant factors and lower-order terms when expressing it in big O notation.

Loops:

Example 3:

N=5

P=0

For(i=1;p<=n;i++){

P=p+i

}

1. 0+1 bcs p=p+1..when N is 0;
2. 1+2=3……N is 1;
3. 1+2+3=6
4. 1+2+….k

Not n times assuming

When will stop when p>n

P=k square >n

So n=sqrt of n

Time complex O(sqrt(n)).

O(Logn) time complex:

For(i=1;i<n;i\*2)

{

Statements;

}

ANALYSE:

i=1 1time

i=2 2times (1\*2)

i=3 4times (1\*2)\*2=2^2

i=4 8times (1\*2)\*2\*2=2^3

so when stopes i>=n

i=2 power k

2 power k>=n

K=log n base 2

So time complex = O(log n base 2).

Create an array it should contain no between 10 to 30.

Example 2:

For(i=n;i>=1;i=i/2)

{

Statements;

}

n/2

n/2 power 2

…….

n/2 power k

assume i<1 it stops right?

n/2 power k<1

n/2 power k=1;

n=2 power k

k= log n base 2 so (O(logn)).

Derived formulas:

1. For (i=0;i<n;i++) O(n)
2. For (i=0;i<n;i+2) O(n)
3. For (i=n;i>1;i--) O(n)
4. For (i=1;i<n;i=i\*2) O(log n base 2)
5. For (i=1;i<n;i=i\*3) O(log n base 2)
6. For (i=n;i>1;i/2) O(log n base 2)
7. For (i=0;i<n;i++){

For (i=0;i<n;i++) O(n^2)

}

* Constant time complex: O(1)..
* linear time complex: O(n)..
* logarithmic time complex: O(logn)..
* quadratic time complex: O(n^2)..
* exponential time complex: O(2^n)..

SPACE COMPLEXITY:

Parallel concept to time complexity.

Array of size n, require O(n) space.

Two dimensional array of size n\*n : O(n2) space.

Linear search O(1)

Mearge sort O(n)

Depth first search(DFS) O(n)

Breadth first search(BFS) O(n)

Dynamic programming O(n^2) or O(n\*m)

Constant space complexity: same amount of space irrespective of the input size n it is called complex complexity.

Example 1.sum of array elements .

def sumof(arr):

b=0

for i in range(len(arr)):

b=b+arr[i]

return b

a=[1,2,3,4,5,6,7,8,9,10 ]

print(sumof(a))

2.linear search.

def sumof(arr,b):

for i in range(len(arr)):

if(arr[i]==b):

print("element found at index of ",i)

break;

a=list(map(int,input().split(" ")))

b=int(input("enter the element to search"))

sumof(a,b)

Why? because space is not depending on values.