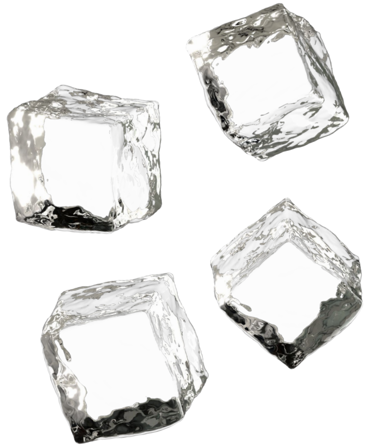
Python Track

Time and Space Complexity

Contents

Africa To Silicon Valley

What is time and space complexity?



Africa To Silicon Valley

Time complexity

It measures an algorithm’s execution time using input size.

Space complexity

- The space Complexity of an algorithm is the total space taken by the algorithm with respect to the input size.
- Expected memory is not considered when calculating space complexity. E.g If a question asks to build a list and return it, it doesn't count.
- Stack space in recursive calls counts too.

Cont...

- Input space: is the the expected space.
- Auxiliary space: The additional space used by the algorithm, e.g., to hold temporary variables or the space used by the call stack.

Asymptotic Analysis

- Asymptotic analysis evaluates the performance of algorithms as input size grows, crucial for predicting efficiency and resource usage.
- Focuses on limiting behavior of an algorithm, disregarding hardware and software factors.
- Enables comparison of worst, average, and best-case scenarios across different algorithms.

What are the different cases in time and space complexity analysis?

- Best Case
- Worst Case
- Average Case
- Amortized Case

A2SV
Africa To Silicon Valley

Best case - (Big Ω)

- In the best-case analysis, we calculate the lower bound on the running time of an algorithm.

- We must know the case that causes a minimum number of operations to be executed.

Average case

- In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs.

- Sum all the calculated values and divide the sum by the total number of inputs

A2SV
Africa To Silicon Valley

Worst case- (Big O)

- In the worst-case analysis, we calculate the upper bound on the running time of an algorithm.
- We must know the case that causes a maximum number of operations to be executed.
- As Competitive Programmers, we care about worst case complexities.

Amortized case

- Amortized time is the way to express the time complexity when an algorithm has the very bad time complexity only once in a while besides the time complexity that happens most of time.

Example 1 - Linear search

- Best case: the best case occurs when x is present at the first location.

- Worst case: the worst case happens when the element to be searched (x) is not present in the array.

Example 2 - Python List append

- Amortized case: for list append, the time complexity is amortized O(1) which means list append first reserve n space on creation then when it fills up, it will copy the values to another space with 2n space capacity and waits for future fill up to double and increase space again

How to calculate time complexity?

Africa To Silicon Valley

Basic Steps to calculate time complexity

Identify the basic operations

- Assignment
- Comparison
- Arithmetic operations
- Array element access
- Conditional Statements
- Loop operations
- Function calls
- Logical operations
- Bitwise operations

=> When analyzing an algorithm, go through the code and identify these basic operations

Count the operations

- Analyze loops and iterations
- Consider conditional statements
- Look for recursion
- Account for nested operations
- Count function calls

Analyze loops and iterations

```
for i in range(n):
    # code

for i in range(3 * n):
    # code

for i in range(3 + n):
    # code
```

```
mx = float('-inf')
mn = float('inf')

for x in array:
    mx = max(mx, x)

for x in array:
    mn = min(mx, x)
```

```
mx = float('-inf')
mn = float('inf')

for x in array:
    mx  = max(mx, x)
    mn  = min(mx, x)
```

A2SV
Africa To Silicon Valley

Consider conditional statements

Assuming `is_perfect_square(i)` is constant time operation

```
for i in range(n):
    if is_perfect_square(i):
        for j in range(n):
            # constant time code here
```

Look for recursion

```python
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)

# Example usage
result = factorial(5)
print(result)
```

Account for nested operations

```python
for i in range(n):
    for j in range(i, n):
        # code
```

```python
def example_code(arr):
    n = len(arr)
    result = 0
    for element in arr:
        i = 1
        while i * i <= n:
            result += element
            i += 1
    return result
```

Count function calls

```python
def example_function_1(x):
    return x*2

def example_function_2(x):
    total = 0
    for element in arr:
        total += example_function_1(element)
    return total

def main_algorithm(arr):
    result = 0
    for _ in range(3):
        result += example_function_2(arr)
    return result
```

Express the relationship

- Express the total number of operations as a mathematical expression of the input size ‘n’.

Simplify to Big O Notation

- Drop the non-dominant terms
- Drop the constants

Drop the non-dominant Terms

- We should drop the non-dominant terms
- Hence,
  - $O(N^2 + N) = O(N^2)$
  - $O(N + \log N) = O(N)$
- We might still have a sum in the runtime.
  - $O(B^2 + A)$ can not be reduced
  - without some special knowledge of A and B.

A2SV
Africa To Silicon Valley

Drop the Constants

- We drop the constants in run time
- An algorithm that one might have described as O(2N) is actually O(N).

```
mx = float('-inf')
mn = float('inf')

for x in array:
    mx = max(mx, x)

for x in array:
    mn = min(mx, x)
```

```
mx = float('-inf')
mn = float('inf')

for x in array:
    mx  = max(mx, x)
    mn  = min(mx, x)
```

A2SV
Africa To Silicon Valley

General Cases

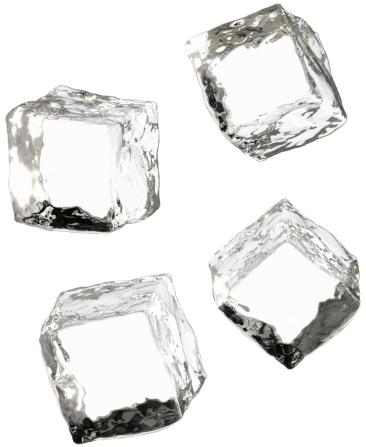
```
n, i = 100, 1
while i <= n:
    i *= 2

while n != 0:
    n //= 2
```

Anytime we are multiplying or dividing our loop variable by some constant other than 1, we are dealing with logarithmic time complexities.

Logarithmic time is very good; for a given dataset of maximum length 5 * 10 ^ 9, an nlogn solution is the same as 32 * n.

How to calculate space complexity?



Africa To Silicon Valley

O(1) - Constant Space

```
def getMin(arr):
   min_ = float('inf')
   N = len(arr)
   for i in range(N):
       min_ = min(min_,arr[i])
return min_
```

O(m) - Linear Space

```
for i in range(n):
    temp = []
    for j in range(m):
        temp.append(i*j)
    print(temp)
```

Why is it not O(n x m)?

O(c) -> O(1)

```
# s is string
freq = {}
max = 0
for i in range(len(s)):
    if s[i] in freq:
        freq[s[i]] += 1
    else:
        freq[s[i]] = 1
    max = max(max,freq[s[i]])
print(max)
```

Why is it O(1)?

O(n*m) - Quadratic space

```
grid = []
for i in range(n):
    temp = []
    for j in range(m):
        temp.append(0)
    grid.append(temp)
```

Time complexities of common data structures

Data Structure	Access	Search	Insertion	Deletion
List	O(1)	O(n)	O(1)*	O(n)
Tuple	O(1)	O(n)	N/A	N/A
Set	N/A	O(1)*	O(1)*	O(1)*
Dictionary	O(1)*	O(1)*	O(1)*	O(1)*
Stack (via list)	O(n)	O(n)	O(1)	O(1)
Queue (via deque)	O(n)	O(n)	O(1)	O(1)
Linked List	O(n)	O(n)	O(1)	O(1)
Binary Search Tree	O(log n)*	O(log n)*	O(log n)*	O(log n)*
Heap	O(n)	O(n)	O(log n)	O(log n)

A2SV
Africa To Silicon Valley

Note:

List: The amortized time for insertion is O(1) when adding to the end of the list. Insertions or deletions at arbitrary positions require shifting elements, making it O(n).

Set and Dictionary: Average-case complexities are O(1), but these can degrade to O(n) in the worst case due to hash collisions.

Binary Search Tree: For a balanced tree, like AVL or Red-Black Tree, operations are O(log n). In an unbalanced tree, they can degrade to O(n).

Tuples: Tuples are immutable, so they do not support insertion or deletion.

Tips and Pitfalls

Africa To Silicon Valley

Tips - Common time complexities

n: input size of the problem	Acceptable time complexity
n ≤ 10	O(n!)
n ≤ 20	O(2^n)
n ≤ 30	O(n^4)
n ≤ 100	O(n^3)
n ≤ 10^3	O(n^2)
n ≤ 10^5	O(n log n)
n ≤ 10^6	O(n)
n > 10^8	O(log n) or O(1)

Tips - Examples

- O(n!) [Factorial time]: Permutations of 1 … n
- O(2^n) [Exponential time]: Exhaust all subsets of an array of size n
- O(n^3) [Cubic time]: Exhaust all triangles with side length less than n
- O(n^2) [Quadratic time]: Slow comparison-based sorting (eg. Bubble Sort, Insertion Sort, Selection Sort)
- O(n log n) [Linearithmic time]: Fast comparison-based sorting (eg. Merge Sort)
- O(n) [Linear time]: Linear Search (Finding maximum/minimum element in a 1D array), Counting Sort
- O(log n) [Logarithmic time]: Binary Search, finding GCD (Greatest Common Divisor) using Euclidean Algorithm
- O(1) [Constant time]: Calculation (eg. Solving linear equations in one unknown)

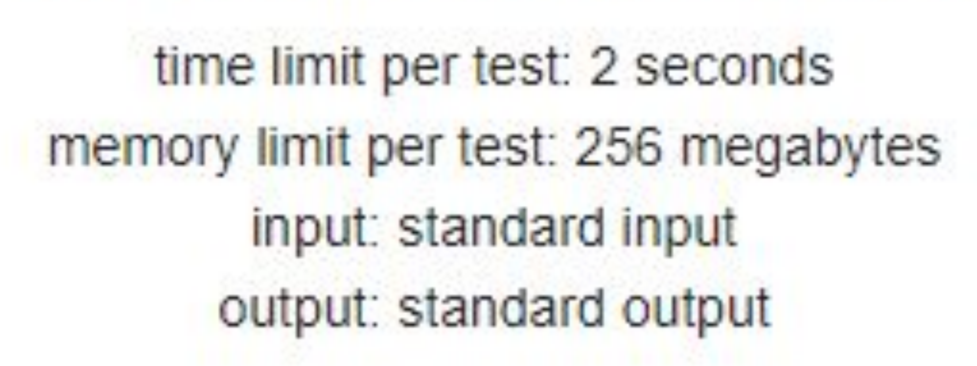
A2SV
Africa To Silicon Valley

Tips - Space Usage

Data Type	Size
char	1 byte
short	2 bytes
int	4 bytes
long	4 bytes
long long	8 bytes
Int[n] / long[n]	4*n bytes ??
str	49 + 1*n bytes

Tips - What does this mean?

It depends on the processor involved. For a processor running 4GHz, it would mean 4*10^9 cycles per-second. If each operation take one cycle, it means the computer can run 10^9 operations per second. (time complexity should be less than 10^9)



time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Tips - Common Errors

Time Limit Exceeded: The program hadn't terminated in time indicated in the problem statement

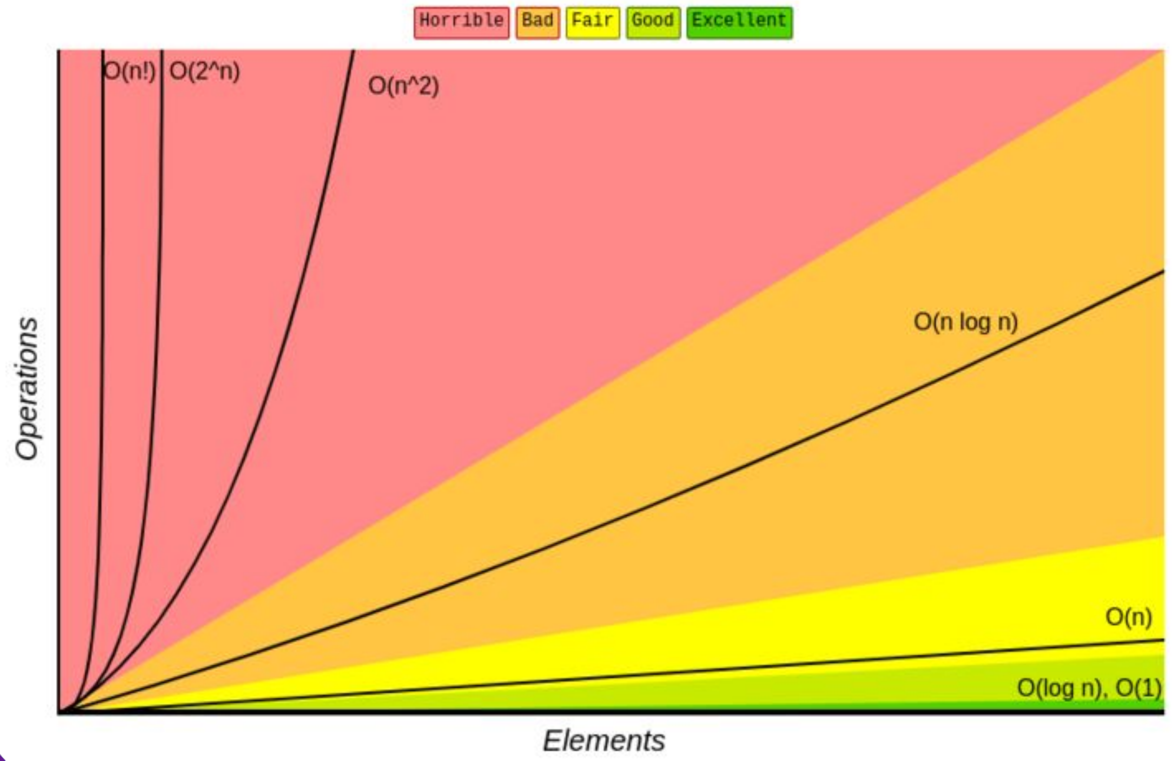
Memory Limit Exceeded: The program tries to consume more memory than is indicated in the problem statement

Runtime Error (): The program terminated with a non-zero return code (possible reasons: array out of bound error, division by zero, stack overflow, incorrect pointers usage, recursion limit exceeded, etc)

Idleness limit exceeded: The program didn't use the CPU time for considerable time

A2SV
Africa To Silicon Valley

Tips


Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

Pitfalls

- Don’t use the same input size notation for different inputs

```
for i in range(len(input_array)):
        for j in range(max_value):
            total_operations += 1
```

Time Complexity :

O(length of input_array*max_value)

Africa To Silicon Valley

Pitfalls

- Don't include the input space in your space analysis; focus on the auxiliary space unless specified otherwise.

  nums.sort() => Space complexity : O(1)
  sorted(nums) => Space Complexity : O(n)

- Expected memory is not considered when calculating space complexity. E.g If a question asks to build a list and return it, it doesn't count.

Pitfalls

- Don’t forget Big-Oh is an expression for the upper bound of an algorithm’s performance. It is possible for an algorithm’s performance to have different upper bounds.

- If an algorithm has O(n) runtime complexity, it is correct to say it also has O(n^2) time complexity.

- Note: Always pick the the upper bound that does not underestimate your algorithm’s performance in your analysis.

Resources

- How to determine the solution of a problem by looking at its constraints? - Codeforces
- A Time Complexity Guide - Codeforces
- Asymptotic notation (article) | Algorithms | Khan Academy
- Codeforces Contest Rules
- Gayle Laakmann McDowell - Cracking the Coding Interview_ 189 Programming Questions and Solutions-CareerCup (2015)

A2SV Africa To Silicon Valley

Quote of the Day

“The first principle is that you must not fool yourself — and you are the easiest person to fool.”

- Richard Feynman

A2SV
Africa To Silicon Valley