Advanced Programming

Analysis of Algorithms - overview

Ramaseshan Ramachandran

Algorithm Analysis

ANALYSIS OF ALGORITHM

- Algorithm analysis[1] involves evaluating the time and space complexities[1]
- Enables informed choices in algorithm selection

Key Questions

1. Does it give accurate results?
2. Predict the performance of an algorithm as the input size increases, making informed decisions about its suitability for large-scale tasks
3. Can we solve the same problem faster?

[1] Time complexity measures how long it takes to execute in terms of computational steps while space complexity measures the amount of memory it uses.

ASSEMBLY CODE FOR ANALYSIS

```
...
mov eax, 10   ; Assign the value 10 to the EAX register
add eax, 15
cmp eax, 5
ret
...
```

ASYMPTOTIC ANALYSIS OF ALGORITHMS

In asymptotic analysis, a formula can be simplified to a single term with coefficient 1. Such a term is called a growth term (rate of growth, order of growth, order of magnitude).
The most common growth terms can be ordered from fastest to slowest as follows:

$O(1)$	Constant time
$O(\log(n))$	Logarithmic time
$O((\log(n))^c)$	Polylogarithmic time
$O(n)$	Linear time
$O(n\log(n))$	Linearithmic time
$O(n^2)$	Quadratic time
$O(n^c)$	Polynomial time
$O(c^n)$	Exponential time
$O(n!)$	Factorial time

BASIC OPERATIONS FOR ANALYSIS

Single step operations $(O(1))$

- Assignment - `a = 5`
- Add two numbers - `b+c`
- Function call
- Return from a function
- Comparison
- `x[3]`

ANALYSIS OF A SIMPLE ALGORITHM - 1/4

Constant Time. Each operation below is considered to be a single operation. Irrespective of the size, they are take constant time to perform the operation

- Array Indexing `a[4]`
- Hash Table/Hash map Lookups - Hash maps are the backbone of dictionaries in Python, Java
- Bitwise Operations:`AND`, `OR`, `and` `XOR` are constant time, regardless of the size of the data 2 » 4
- Basic arithmetic operations `+`, `-` , `/` , `*`
- Comparison `if x > 2:`

ANALYSIS OF A SIMPLE ALGORITHM - 2/4

Let us consider an unsorted array USA of size n

Algorithm To Find an element X from USA

```
$i \leftarrow 0$
while ($i < n$ and $USA[i] \neq X$) do
    $i \leftarrow i + 1$
    if $i < n$ then
        return $i$
    else
        return $-1$
    end if
end while
```

Worst case - Element is not found or the element is the last one in the array - proportional to the size of the array

Average Case

ANALYSIS OF A SIMPLE ALGORITHM

```
txt = input("Enter a number: ")  # 2 operations
n = int(txt)  # 2 operations

# Check if the number is positive
if n > 0:  # 1 operation
    print("The number is positive.")  # 1 operation

for i in range(n):  # 2 operations (initialization,
   condition check)
    if i % 10 == 0:  # 2 operations (modulus, comparison)
        j = i * 100  # 2 operation (multiplication)
```

Maximum number of operations: $2+2+1+1+2n+2n+2n$

At least: $2+2+1+1+2n+2n$

ASYMPTOTIC ANALYSIS

The Big O notation does not consider the between multiplicative constants. The functions f(n) = 10n and g(n) = 100n are identical in O analysis.

Formal Definitions

- $f(n) = O(g(n))$ means $c \cdot g(n)$ is an upper bound on $f(n)$. Thus there exists some constant c such that $f(n) \leq c \cdot g(n)$, for large enough n (i.e. , $n \geq n_0$ for some constant n_0).
- $f(n) = \Omega(g(n))$ means $c.g(n)$ is a lower bound on $f(n)$. Thus there exists some constant c such that $f(n) \geq c \cdot g(n)$, for all $n \geq n_0$.
- $f(n) = \Theta(g(n))$ means $c_1 \cdot g(n)$ is an upper bound on $f(n)$ and $c_2 \cdot g(n)$ is a lower bound on $f(n), \forall n \geq n_0$. Thus there exist constants c_1 and c_2 such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.

RUNTIME ANALYSIS

function `find_max` - finds a maximum value given a list of numbers and returns the maximum value found in the list

```python
def find_max(vector:list) -> int:
    max_element = vector[0]
    for element in vector:
        if element > max_element:
            max_element = element

    return max_element
```

RUNTIME ANALYSIS

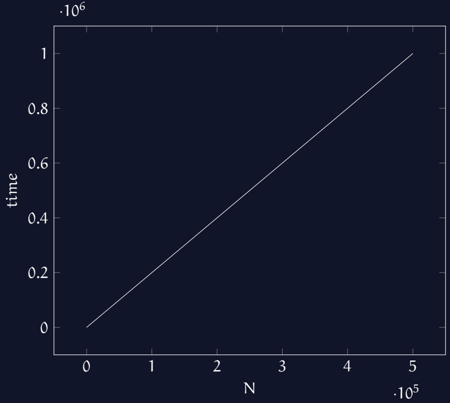
function `find_max` - finds a maximum value given a list of numbers and returns the maximum value found in the list

```
def find_max(vector:list) -> int:
    ~~max_element = vector[0]~~
    for element in vector:
        if element > max_element:
            max_element = element

    return max_element
```

RUNTIME ANALYSIS

The dominant factor in the time complexity is the loop. The number of loop operations depend on the number of elements in the data. Hence, the runtime of executing the loop is proportional to the number of elements in the vector(n), or $t \propto n$. We call this overall time complexity/runtime as $O(n)$ or on the order of n operations

```
def find_max(vector:list) ->
    int:
     ~~max_element = vector[0]~~
     for element in vector:
     ~~if element > max_element:~~
         ~~max_element = element~~

     ~~return max_element~~
```



TIME COMPLEXITY - $O(N^2)$

Another algorithm with runtime $O(N^2)$

```
def two_loops(items:list) -> list:
    items = []
    for j in range(1,len(items)):
        ...
        i = j -1
        while i >= 0 and items[i] > key:
            ...
            ...
```

TIME COMPLEXITY - $O(\log n)$

Runtime of this algorithm - $O(\log n)$

```
def binary_search(sorted_array, target):
    low, high = 0, len(sorted_array) - 1

    while low <= high:
        mid = (low + high) // 2
        mid_value = sorted_array[mid]

        if mid_value == target:
            return mid  # Found at mid(index)
        elif mid_value > target:
            #Target is in the left half
            high = mid - 1
        else:
            low = mid + 1   # Target is in the right half
return -1  # Target not found in the array
```

DOMINANT TERM

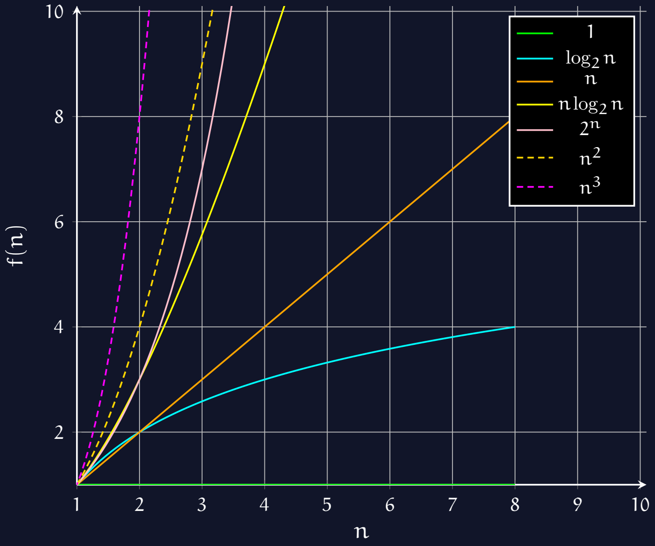
The total runtime is $O(n^2 + 800000)$. Here, 800K is significant in the presence of a quadratic term. Even if N is quite large, it is insignificant in this case. The relative performance of the algorithm with respect to N is important and significant in the analysis of algorithms. Hence, the run time complexity is $O(N^2)$

```
def quadratic_function(vector:list) -> int:
    The following loop is O(1)
    for i in range(800000):
        x = y
    The following code's runtime is O(N^2)
    for k in vector:
        l = k - 1
        while l < len(vector):
            ...
```

KEY RULES FOR DETERMINING TIME COMPLEXITY

- **Same indent Level**: Add the complexities at the same indent level
- **Focus on the most dominant term**: When adding terms, keep only the one that grows most significantly as data size increases. This is called the .
- **Ignore constant factors**: Terms that don't change with data size ($O(1)$) have minimal impact in the long run.
- **Multiply for nested code**: When code blocks are nested (like loops within loops), multiply their complexities to get the overall complexity.
- **Start from the inside out**: Analyze code's complexity starting with the innermost blocks and work your way outwards.
- **Consider individual statement costs**: Different code statements have varying costs. For example, calling a function with $O(N^2)$ complexity takes $O(N^2)$ time, even if it's just one line of code.

TIME COMPLEXITY GRAPH



COMNON ORDERS OF GROWTH

Name	Notation	Code Fragment
Constant	$O(1)$	`x = y`

Logarithmic	$O(\log n)$	`low = 0`
		`high = len(vector) - 1`
		`while low <= high:`
		`        mid = (low + high)`

Linear	$O(N)$	`for i in vector:`

Linearithmic	$O(n \log n)$	`low = 0`
		`high = len(vector) - 1`
		`for i in vector:`
		`    while low <= high:`
		`        mid = (low + high)`

COMNON ORDERS OF GROWTH[2]

Name	Notation	Code Fragment

Quadratic	$O(n^2)$	`for i in range(n):`
		`    for j  in range(n):`

Cubic	$O(n^3)$	`for i in range(n):`
		`    for j  in range(n):`
		`        for k in range(n):`

[2] CHEAT SHEET FOR ALGORITHM ANALYSIS

REFERENCES

[1] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison Wesley, 2006.