Chapter 1 Algorithmic Analysis

1.1 Correctness

A problem X is defined as an ordered (q,a) set where 'q' is a question and 'a' is an answer, written mathematically as:

$$X = ((q(x), a(x))|x \exists D)$$

With a 'D' arbitrary set of data.

Q(x) is the set of questions:

$$Qx = q|(q, a) \exists X$$

and a set of questions is solved by an algorithm 'A' if

$$(\forall q \exists Px|A[q] = a : (q, a) \exists X)$$

Besides all of this, the size of a question is written as *|q|=i, the difficulty of a question is usually proportional to its size.*

An algorithm is considered correct if it solves a problem given A input, resulting in a desired B output in all possible cases.

1.2 Complexity

Algorithmic complexity is the approximated cost of an algorithm when running on a machine capable of computing. This kind of analysis can be quantified in time, memory, and even operational costs.

> ☞ ***Remark***
> *The method to define and value algorithms must be independent of programming language, architecture, and even more broadly, of the specific machine the algorithm runs on. This implies we don't take into account computational problems, such as compilation or, in interpreted languages, the interpretation into bytecode of the algorithm, for this doesn't concern the algorithm, but rather factors external to it.*

1.2.1 R-Complexity

R is the class of decision problems solvable by a Turing machine. The main idea of it is defining a sum of the costs inside of the machine according to the operations realized by the algorithm. Resulting mathematically in the total cost of the algorithm's execution. However, R-Complexity is generally hard to compare between programming languages and individual machines because of the dependance of the result in constants.

given this definition, take as an example:

```
#Python pseudocode, take a list and copy it into another.
def Solution(list nums):
    answer = [] # <-- R = C1 + C2
    for i in list: # <-- R = nC9
        answer.append(i) # <-- <-- R = nC8
    return answer
```

And therefore, the R-Complexity of this equation is

$$R_n = C1 + C2 * n(C9 + C8)$$

However, even though this algorithm might have such R-Complexity, different implementations of this same Algorithm, where the solution is expressed even slightly different, could have completely different R-Complexity equations.

Take as an example:

```
#Python pseudocode, take a list and copy it into another.
def solution(List):
answer = []   # <-- R = C1 + C2
for i in range(0,len(List)): # <-- R = C1 + (n + 1)C3 + (n+1)C4 + nC5
    current = List[i]  # <-- R = nC1 + nC6
    answer.append(current)   # <-- R = nC7 + nC8
return answer
```

this implementation is a different algorithm that solves the exact same problem, and which R-Complexity can be defined as:

$$R_n = 2C_1 + C_2 + C_3 + C_4 + n(C_1 + C_3 + C_4 + C_5 + C_6 + C_7 + C_8)$$

This is a fairly different, much longer equation, which includes a lot more constants and a bit more complex to compute Given this, sometimes R-Complexity might end up being fairly dependant on implementation when different solutions are presented.

1.2.2 Complexity Order

be it 'f' and 'q' functions such as: f, g $: \mathbb{N} \rightarrow \mathbb{R}$

we can affirm that f is of equal or lesser order to g if:

$$(\exists c, k \exists \mathbb{N} | : (\forall m \exists \mathbb{N} | k \leq m : f(m) \leq cg(m))))$$

and we can write it as:

$$f = \mathbb{O}(g) \wedge f(n) = O(g(n))$$

'O' induces a partial order on functions such as

$$\mathbb{N} \rightarrow \mathbb{R}$$

and is an equivalence relation over that same group of functions.

Order Theorems

✿ *Constant rule:*

$$(\forall c \exists N | : cf = O(f))$$

✿ *Sum rule:*

$$f + g = O(max(f, g))$$

✿ *Product rule:*

$$f = O(r) \wedge g = O(s) \longrightarrow fg = O(rs)$$

Order classification examples

1.

$$2n^5 = O(n^5)$$

2.

$$2n^5 + 4n = O(n^5)$$

3.

$$(2n^5 + 4n)^2 = O(n^{10})$$

4.

$$(n + 1)^2 = O(n^2)$$

1.2.3 Complexity orders

Complexity orders are classified as follows, according to big O notation.

✿

$$O(1)$$

Constant: Execution time doesn't depend on the size of the input

$$O(log(n))$$

Logarithmic: Execution time grows slower than the input size, Ex: binary search

$$O(n)$$

Linear: Execution time grows at the same rate as the input size, Ex. Unordered array search

$$O(nlog(n))$$

Log-Linear: Execution time grows at the same rate as the input size but does O(log(n)) operations per item

$$O(n^2)$$

Quadratic: Execution time grows at a linear rate related to the input size, but requires a linear operation per item in the input.

$$O(2^n)$$

Exponential: Execution time grows exponentially compared to the input