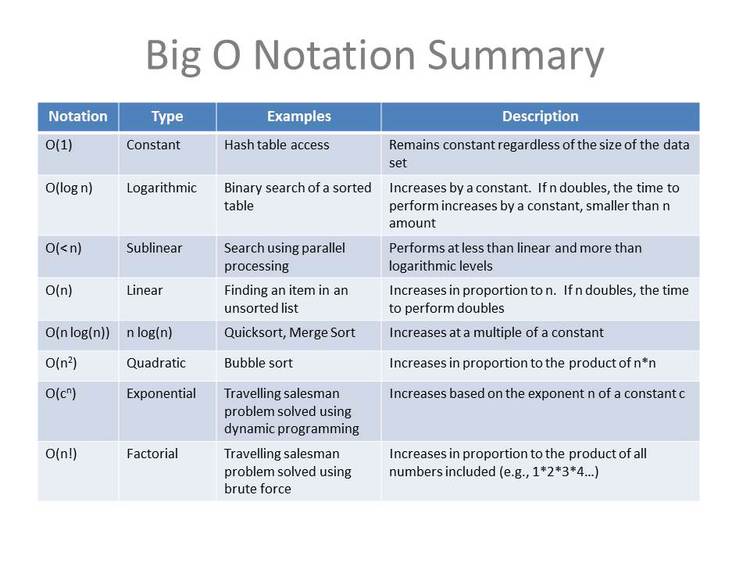
**The Notations Classifications Introduction. To understand the run time programs may take**

* Definition: It stands for a representation of a boundary
* As a good engineer you must give a classification for each program. The program may be executed using n^2, then is complied using n, and it can export using n log(n).

Table

Description automatically generated

**The Representation of Boundaries**

**Each one of these Greek symbols represents a type of a bound**

* Little o -> this means the program will always be faster than its bound

In this specific case. Say you have a program that executes using n^2, complied using n, and exported using nlog(n).

The notation that takes the longest is n^2

Therefore, little o means no matter what our program will be always faster than the worst notation we have

It tells us the worst-case scenario and it also tells us that our program will always run better than the worst-case scenario. Therefore, we know that our program cannot be at the worst case scenario but by HOW MUCH >. Therefore, it doesn’t benefit us too much while planning

* **big o** -> this means the program will always be faster or equal to than its bound

In this specific case. Say you have a program that executes using n^2, complied using n, and exported using nlog(n).

The notation that takes the longest is n^2

Therefore, big o means no matter what our program will be always faster or equal to the worst notation we have

It tells us that our program may be fast than a specific bad case or equal to it. It is similar to little o. But, it has an additional feature that tells us that our program could run at the worst case scenario. Therefore, the worst case scenario can be set as our limit.

* Equal too -> this means the program will always be equal to its bound

In this specific case. Say you have a program that executes using n^2, complied using n, and exported using nlog(n).

The notation that takes the longest is n^2

Therefore, “equal too” means no matter what our program will be always equal to the worst notation we have

Why is this useful > because if gives us exactly how fast the program will run. But the issue is that programs do not really run along a long a line constantly “It is too specific”

* Slower or equal too -> this means the program will always be slower or equal to than its bound

In this specific case. Say you have a program that executes using n^2, complied using n, and exported using nlog(n).

The notation that takes the longest is n^2

Therefore, ” Slower or equal too” means no matter what our program will be always slower or equal to the worst notation we have. Therefore, at best out program will run at n^2

Why is this useful > because it gives us very limited info, because there is an infinite side to it. It would go to infinite on its bad side

* Slower that -> this means the program will always be slower than its bound

In this specific case. Say you have a program that executes using n^2, complied using n, and exported using nlog(n).

The notation that takes the longest is n^2

Therefore, “Slower that” means no matter what our program will be always slower to the worst notation we have. Therefore, at best our program will always run slower than n^2

Why is this useful > because it gives us very limited info, because there is an infinite side to it. It would go to infinite on its bad side

**The big O The Greek letter is Omicron**

* This is the most important classification because it gives us the best possible algorithm
* It provides us with the worst-case scenario that our program might experience. Because it always is faster or equal to the worst-case scenario. Therefore, it works ideally as a benchmark notation
* It tells us that our program may be fast than a specific bad case or equal to it. It is similar to little o. But, it has an additional feature that tells us that our program could run at the worst case scenario. Therefore, the worst case scenario can be set as our limit.
* The most important notation above is the Omicron, or "Big O". The reason for this is because it gives us a worse case scenario. That means we know the absolute worse case scenario of our program, meaning if we plan for this, we won't have any surprises or weird corner cases that come up.
* Example:- if we have a Big O of nlog(n) -> O(nlog(n)) -> this means at worst case scenario it will be faster than nlog(n)
* Therefore, it helps us to plan for the worst-case scenario
* Example: configure the run time of a program that does the following:
* Loading -> O(n^2)
* Execute Step 1 -> O(nlogn)
* Execute Step 1 -> O(2n)
* Save -> O (1)
* Combine them -> n^2+nlong(n)+2n+(1)
* The constant 2 is useless. Then everything else will be crossed out and we will focus on the worst case scenario which is n^2
* The worst-case scenario is that program will run as n^2. Therefore, we should always plan that we are going to run at worst n^2. Because according to the big O notation we will always be running at or faster that n^2

**The Big O Notation Real World Examples “In Sudo Code”**

* 1. For Example 1

For each element in a list

{

Print each element

}

* + We are saying for each element in this set print the element
  + If we had 4 elements, then n = 4
  + For a single four loop you tend to have n number of operations.
  + In this case we had 4 data entries and ended up performing 4 operations. We have a O(n) type of an algorithm
* 2. For Example 2

For each data list in W

{

For data in W:

If elements\_W = elements\_W

Print elements

}

* + We are saying for each element in the list W, check it against all the other elements in the same list. Sometimes you can have repeated elements within a list. If elements match prints them
  + If W is 4. Therefore, you start with n = 4. However, to check each element against every element within a 4 elements list, you will perform 16 operations.
    - Basically, you will check:
      * element1 vs. element1, then element 1 vs element2, then element1 vs element3, then element1 vs element4, then element2 vs element1, then element2 vs element2, and so on all the way to element4 vs element4.
    - This equals 16 operations
  + Therefore, you started with n = 4 and ended up doing 16 operations.
  + 16 is 4^2 -> this means you started with n and ended up performing n^2 operations
  + Therefore, you are running an O(n^2)
* **General Rules:** 
  + When running a single for loop you tend to have O(n). then if you run a single nested for loop within that initial loop you will have O(n^2). Because you perform operations on every element from the first for loop
  + If you have another for loop inside the nested loop then you most likely will have O(n^3)