COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 4: JAN 30

Lecturer: Dr. Andrew Hines *Scribes: Miguel Martinez, Rafael Martins*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

4.1 Outline

Having looked at the Big-O notation on the previous lecture we here add to it by looking into the running times of the different types of function complexities. We proceed to delve into the rules that define the Big-O notation followed by coverage of the Big-Omega and Big-Theta.

4.2 Different Running Types

It is useful to highlight that the complexity of algorithms in general is not measured by the amount of lines of code that the algorithm is composed of. Therefore the running time is used as an aim to measure the complexity of an algorithm. You might come across an algorithm with 100s of lines of code that is less complex and is able to scale and take bigger sized inputs than an algorithm with 10 lines of code. It is now that we need to define the different running types.

4.2.1 Constant

The mathematical display of the Constant Running Time is: f(n) = c.

This is the simplest type of running time complexity that is found in data structure studies. The running time of this algorithm is not affected by the value of "n". With "n" being the input given to the algorithm. It is the count of the amount of operations that is taken into consideration.

Algorithm 1: Pseudo-code for Constant running type

Input : array with n items
Output: first element of the array

```
1 define function(array):
2     print array[0]
```

See the example above. We find that irrelevant of the amount of items that the array contains ("n") the number of operations performed will always be the same. In this case two operations:

line 2: print and accessing the array.

4.2.2 Linear

The mathematical display of the Linear Running Time is: f(n) = n.

This running time complexity occurs when we perform the same operation on a given amount of elements. The number of elements to be worked on takes the shape of "n" which again is the input. For all the complexities affected by "n", this structure provides one of the best compilation time of the complexities looked at. Logarithmic would be even better.

Algorithm 2: Pseudo-code for Linear running type

Input : array with n items
Output: each item in the array

```
1 define function(array):
2     for each in array:
3         print each
```

See example above. We find that depending on the amount of items in the array the number of operations to be performed will vary proportionally. Therefore the bigger the input the more time the algorithm takes to compute. In this example we have a complexity of 3n:

line 2: two operations per loop.
line 3: one operation per loop.

4.2.3 Quadratic

The mathematical display of the Quadratic Running Time is: f(n) = n^2.

This running time complexity occurs usually when nested loops are used. This is due to the fact that for each initial loop operation there will need to be a nested or child loop performed in turn. In other words there will be operations that will occur on a per loop per loop basis, making the algorithm more complex.

Algorithm 3: Pseudo-code for Quadratic running type

Input : range of size n
Output: all possible products between elements in the array

```
1 define function(array):
2     for i in range(n):
3         for j in range(n):
4             print i times j
```

See example above. We find that due to the fact that there is a loop nested within a loop, now we have a complexity of order n^2 where each item as "j" is multiplied by each item as "i". In this example we have a complexity of $4n^2$ + 2n:

line 2: two operations, one for loop and one for computing the range.
line 3: two operations per loop per loop, one for loop and one for computing the range.
line 4: two operations per loop per loop, one for printing and one for multiplying "i" by "j".

4.2.4 Logarithmic

The mathematical display of the Logarithmic Running Time is: f(n) = $\log_b n$.

Working out this running time complexity involves obtaining the number of times that we can divide "n" by "b" until we arrive to a less than or equal to one result. This approximation is used in order to avoid using calculus that otherwise would have been needed.

Algorithm 4: Pseudo-code for Logarithmic running type

Input : sorted array and an element
Output: Index of the element in the array or error message if item not found

```
1  define function(array, elem):
2      low = 0
3      high = n - 1
4      while (low ≤ high):
5          mid = (low + high) / 2
6          if array[mid] > elem:
7              high = mid - 1
8          else:
9              if array[mid] < elem:
10                 low = mid + 1
11             else:
12                 return mid
13     return error message
```

Let's assume we are returning the index of an item in a sorted array by using Binary search. The array will be sliced into two and the side of the array that does not contain the item required will be discarded. This process will be done repeatedly until the required item is found. With the worst case scenario being "n" divided by "b" until we obtain a number less or equal to one as described and shown above.

4.3 Big-O Rules

- Forget about lower-order terms.
- Forget about constant factors.
- Use the smallest possible degree.

In order to express the running time in Big-O notation we must drop constants, drop lower order terms and use the smallest possible degree. For example if f(n) = 2 we take Big-O as O(1), if f(n) = 2n + 8 we take Big-O as O(n) and finally if f(n) = n^2 + 2n + 5 we take big-O as O(n^2).

4.4 Big-Omega and Big-Theta

It is worth pointing out that having the Big-O notation representing the worst case scenario or upper bound of an algorithm we encounter the Big-Omega notation which shows the best case scenario or lower bound while the Big-Theta notation refers to both the higher and lower bound or best and worst case scenarios.