Algorithmic Analysis

Charl du Plessis and Robert Ketteringham

Basic Terminology

- Time Efficiency – this indicates the speed of an algorithm
- Space Efficiency – this indicates the amount of extra space an algorithm requires
- Mostly interested in Time Efficiency
- We will use some notation (O) Big-O for this

Starting Analysis

- Measure the input size.
- Describe its efficiency as a function on some parameter, say N.
- Usually, we let the number of data elements in the input be N.
- Note that some algorithms will require more than one function such as the number of vertices (V) and edges (E).

Starting Analysis

- Now let us put the time = F(N).
- Observe that we are not very interest in the running time for a program for small inputs.
- We will use what is called Big-Oh (O) as a way to compare the growth rates of functions

Big-Oh (O)

- Constant values are usually ignored.
- We also ignore smaller cases of input.
- Growth rate of algorithm:
  - Linear
  - Quadratic
  - Cubic
  - Logarithmic
- Since we ignore constant values, function takes upon most dominant term.

Big-Oh (O)

Function	Name
C	Constant
LogN	Logarithmic
Log^2N	Log-squared
N	Linear
NlogN	NlogN
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Is an upper-bound of the growth rate of the algorithm.

T(N) = O(F(N)) if there are positive constants M and c such that:

T(N) <= c.F(N) when N >= M

Other Terminology

Big-Omega: gives lower bound on the growth of T(N).

T(N) = Ω(F(N)) if there are positive constants c and M such that

T(N) >= c.F(N) when N >= M

Other Terminology

Big-theta: growth rate of $T(N)$ is equal to $F(N)$

$T(N) = \Theta(F(N))$

iff $T(N) = O(F(N))$

and $T(N) = \Omega(F(N))$

Other Terminology

Little-Oh: growth rate of T(N) is less than F(N)

T(N) = o(F(N)) if there are postive constants M and c such that:

$$T(N) < c.F(N) \text{ when } N >= M$$

Big-Oh Rules

Rule 1: If T(N) is a polynomial of degree k, then:

$T(N) = \Theta\ (N^k)$

Rule 2: If $T_1(N) = O(F(N))$ and $T_2(N) = O(G(N))$ then:

$T_1(N) + T_2(N) = \max\{O(F(N)),O(G(N))\}$

$T_1(N)*T_2(N) = O(F(N)*G(N))$

Rule 3:

$(\text{Log } N)^k = O(N)$

Big-Oh Disadvantages

- It gives the worst case run bound – and this might not happen often
- Sometimes it is better to use average case run bound, but is often difficult to calculate
- Not useful for small input – since only indicates how fast the algorithm grows
- Two different algorithms can have the same Big-Oh, but can run differently

Analysis – Sequential Search

- Given an array N elements and key, we search every given element for the key.
- Worst case: N
- Best case: 1
- Average case: Let p be the probability that the key is in the array.

Then the probability that the key will be in the ith position is p/N.

We will look at the average number of comparisons the program must make.

Analysis – Sequential Search

$$Avg = (1*p/N + 2*p/N + \ldots + N*p/N) + N*(1-p)$$
$$= p/N * (1+2+\ldots+N) + N*(1-p)$$
$$= p/N * N(N+1)/2 + N(1-p)$$
$$= p(N+1)/2 + N(1-p)$$

If $p = 1$ (the key will definitely be in the list), then the average number of comparisons is:

$$(N+1)/2$$

Analysis of Iterative Algorithms

- Choose parameter(s) to indicate the input's size
- Determine the algorithm's basic operation (operation in its most inner loop)
- Check if the number of times the basic operation depends only on the input size. If it depends on some other factors observe worst-case, best-case, avg-case seperately.
- Use the rules of Big-Oh and analyze the order of growth.

Analysis of Recursive Algorithms

- Choose parameter(s) to indicate the input's size
- Determine the algorithm's basic operation (operation in its most inner loop)
- Check if the number of times the basic operation depends only on the input size. If it depends on some other factors observe worst-case, best-case, avg-case seperately.
- Set up a recurrence relation for the number of times the that the basic operation is executed.
- Solve the recurrence or determine the order of growth of its solution.