Algorithm analysis - Textbook chapter 4.1 to 4.3

October 21, 2020 | 11:04 AM

4.1 Experimental Studies

- Primitive ways of measuring an algorithm's efficiency. Implement it and run with different test inputs. To do that we can use System.currentTimeMillis() before and after the algorithm and subtract to get an estimate of elapsed/running time of the method.

```
longstartTime=System.currentTimeMillis(); //recordthestartingtime
/*runthealgorithm*/
longendTime=System.currentTimeMillis(); //recordtheendingtime
longelapsed=endTime-startTime; //computetheelapsedtime
```

→ By using this method we would have to consider different input size, run on different machines. Can get different results on same machine, cuz processes share the CPU.

→ So we can't actually compare unless running in same hardware and software environments.

→ We also need to fully implement the algorithm to test it and plot results. Can be difficult.

Primitive operations are: (constant running time for all of these) 1) Assigning values to variables 2) Following an object reference. 3) Performing an arithmetic operation 4) Comparing numbers 5) Checking expression 6) Accessing an element by index 7) Calling a method 8) Returning from a method.

→ Average case is harder to calculate and compare so we use worst case as a function of input size. By making it the standard of success for algorithms (performing well in the worst case) then it will perform better in every case.

Theoretical Analysis: - Uses pseudocode instead of implementing the algorithm.
- Classifies algorithms running time as a function of input size n. Takes all possible inputs into account.
- Allows to evaluate performance independent of hardware/software.

Pseudocode: - Description of an algorithm. More structured than english prose, less detailed than a program.
- Hides the programs design or language specific issues/constraints.

4.2 The Seven Important functions used in algorithm analysis.

1) The constant function: $f(n) = c$

For any argument n (input) the constant function will assign a constant value c. We're only interested in integer functions so its always $f(n) = 1$. Any other constant function can be written like $f'(n) = c\,f(n) = c\,f(1)$. This is the simplest function and it describes the number of steps needed for a basic operation (Like primitives listed earlier)

Example Algorithms:
- Accessing Array's index (int a = Arr[5I;])
- Inserting a node in Linked list
- Insertion and removal from stack or queue.
- Moving to the next element in a doubly linked list.

2) The Logarithmic function: $f(n) = \log_b n$ for some constant $b > 1$

The most common base is 2 since computers store integers in binary.

* Some important identities:
  - $\log_b(ac) = \log_b a + \log_b c$
  - $\log_b(a/c) = \log_b a - \log_b c$
  - $\log_b(a^c) = c \log_b a$
  - $\log_b a = \frac{\log_d a}{\log_d b}$
  - $b^{\log_d a} = a^{\log_d b}$

Example Algorithms:
- Binary search (splits input into half every iteration)
- Calculating Fibonacci Number (Not using complete data and reducing size with every iteration)
- Some divide and conquer algos

3) The linear function: $f(n) = n$

Given an input n, the function assigns the value n itself. This is usually the best we can achieve for an algorithm that has to process each of n objects. Since reading n objects requires n operations. (unless stored in memory/cache already).

Example Algorithms:
- Traversing an Array or a LinkedList
- Linear Search (look at elements in a list till a match is found or no more elements)
- Deletion of a specific element in a LinkedList (that isn't sorted)
- Comparing two strings for equality or checking for palindrome
- Counting/Bucket Sort (Creates empty lists and fill with elements in a range. Sort the elements in that list/bucket then gather sorted result.) Scatter-Gather

!! Can go to n^2 depending on how u sort buckets and the distribution !!

4) The $n \log n$ function: $f(n) = n \log n$

Given an input n, the function assigns the value of n times the logarithm base-two of n. This grows faster than the

4) The N log N function: $f(n) = n \log n$

Given an input n, the function assigns the value of n times the logarithm base two of n. This grows faster than the linear function but less than the quadratic. The fastest possible algorithms for sorting n elements is nlogn

Example Algorithms:
- Merge sort (split array in 2(get the mid), Sort left (call itself till its 1, sort right; Merge in order)
- Heap sort
- Quick sort (picks a pivot, compare and sort based on pivot, split input around pivot, repeat)

Can get to n^2 if pivot is always smallest number

5) The quadratic function: $f(n) = n^2$

Given an input n, the function assigns the value of n x n. This can be seen in algorithms that have nested loops.
The inner loop will have to run n times and same for outer so $n \times n = n^2$

Example Algorithms:
- Bubble sort (Loop on array, compare and sort adjacents, loop again sort again, keep looping till 1 whole run with no swaps)
- Insertion sort (loop, compare current element to one on its left, if smaller sort by inserting it before. move to next, keep comparing and inserting at right place)
- Traversing a 2D array
- Selection sort (we have unsorted + sorted parts. Assign 1st element in unsorted to be minimum, loop over array looking for a smaller value to swap index, now 1 element in sorted, repeat)

6) The cubic function + polynomials: $f(n) = n^3$, n^4 etc.

Just keep adding more nested loops kek. They grow rapidly as input increases, but still not the worst.

Example: Naively solving a multi-variable equation

7) The exponential function: $f(n) = b^n$

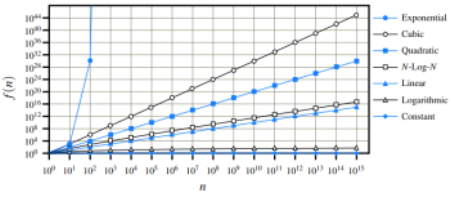
The most common base is 2 since computers store integers in binary. Given an input n, multiply b by itself n times.

So something that will double the size of its input on every iteration is 2^n

* Some important identities:
- $(b^a)^c = b^{ac}$
- $b^a b^c = b^{a+c}$
- $\frac{b^a}{b^c} = b^{a-c}$

Example Algorithms:
- Recursive Fibonacci (with 2 calls)
- Tower of Hanoi
- Finding powersets

Comparing growth rate of mentioned functions. 2^n isn't the worst, factorial grows even faster.



4.3 Asymptotic Analysis

① The Big O notation

Used to give an estimate of an algorithm's upper bound. To say this function will be greater or equal to this other function

Let f(n) and g(n) be functions mapping positive integers to positive real numbers.
We say that f(n) is O(g(n)) if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that:

$$f(n) \leq c \cdot g(n), \text{ for } n \geq n_0$$

Book Example: Function $2n+5$ is $O(n)$.

a) we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $2n+5 \leq cn$ for every integer $n \geq n_0$

→ Possible $c = 3$ and $n_0 = 5$ or $c = 13$ and $n_0 = 1$

b) Show that $3x^2 + 25$ is $O(x^2)$

(way 1)
- when $x = 5$, $3(5)^2 + 25 = 75 + 25 = 100$
  $c(5)^2 \geq 100$; $25c \geq 100$ so $c \geq 4$

let $c = 4$ and $k = 5$
so $3x^2 + 25 \leq 4(x^2)$ when $x > 5$

- when $x=6$, $3(6)^3+25 = 133$ $\quad 4(6)^2=144$

(way2) Bump everything to highest order in polynomial and find c and k

$3x^2+25 \rightarrow 3x^2+25x^2 \rightarrow 28x^2$

so $3x^2+25 \leq 28x^2$ when $x>1$

so $c=28$ and $k=1$

(c) Show that $4x^3+7x^2+12$ is $O(x^3)$.

$4x^3+7x^2+12 \leq 4x^3+7x^2+12x^2$

$4x^3+7x^2+12 \leq 23x^3$ for $x>1$

so $c=23$ and $k=1$

(d) show that x^3+5x is not $O(x^2)$.

$x^3+5x \leq c(x^2)$ where $x>k$

even if we let $c=x$, we will get equality at best, which will only work for 1 case like: $x^3+5x \leq x \cdot x^2$

$x^3+5x \leq x^3$ $\quad x \geq 0$. There is no c that will keep $x^3+5x \leq cx^2$ once $x>c$

(e) Exercise (6) from Assignment 1.

is $8000000n^2\log n + n^3 \in O(n^3 \log n)$

To show that we need to find $c>0$ and $k \geq 1$

$8000000n^2\log n + n^3 \leq 8000000n^3 \log n + n^3$

Properties of Big O

- We can ignore constant factors and lower order polynomials.
- Since its an upper bound. Saying $4n^3+3n^2$ is $O(n^5)$ or $O(n^4)$ is true. But we're trying to get to an accurate bound so $O(n^3)$ makes more sense
- Instead of saying $2n^2$ is $O(4n^2+6n\log n)$ which is true we want simplest form so $O(n^2)$

(2) Big Omega Ω

Similar to Big O, but estimates a lower bound to the algorithm. functions is lower than or equal to.

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers.
We say that $f(n)$ is $\Omega(g(n))$ if there is a real constant $c>0$ and an integer constant $n_0 \geq 1$ such that:

$$f(n) \geq c \cdot g(n), \text{ for } n \geq n_0$$

Book example: $3n\log n - 2n$ is $\Omega(n \log n)$

a) $\rightarrow 3n\log n - 2n = n\log n + 2n(\log n - 1) \geq n \log n$ for $n \geq 2$ so $c=1$ and $k=2$.

b) Prove $3n+1$ is $\Omega(n)$

(3) Big theta Θ

Means that 2 functions grow with the same rate up to a constant factor.
We say that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$ with $c'>0$ and $c''>0$ and $n_0 \geq 1$

$$c'g(n) \leq f(n) \leq c''g(n) \quad \text{for } n \geq n_0$$

Book example: $3n\log n + 4n + 5\log n$ is $\Theta(n \log n)$

$$3n\log n \leq 3n\log n + 4n + 5\log n \leq (3+4+5)n\log n \quad \text{for } n \geq 2$$