6/10/2020 - Alg. Analysis

Wednesday, June 10, 2020 4:04 PM

1. Base Conversion

   base of a number is the # of symbols used to represent numerals

   Our base we use is base 10, since we have 10 digits:
   0,1,2,3,4,5,6,7,8, and 9. (The name for our base is **decimal**.)

   Other bases that have names:

   base 2 is called **binary**. (0, 1)
   base 3 is called **ternary**. (0, 1, 2)
   base 8 is called **octal**. (0, 1, 2, 3, 4, 5, 6, 7)
   base 16 is called **hexadecimal** (0, 1, 2, …, 9, a, b, c, d, e and f)

   Let's say we have some number in base b with the following symbols:

   d(k), d(k-1), …, d(0)

   This number has the value

   $d(k)b^k + d(k-1)b^{k-1} + \ldots + d(0)b^0$

   If I have the number 6254 in octal, it's value is simply

   $6(8^3) + 2(8^2) + 5(8^1) + 4(8^0) = 3244$

   503 in base 8 converted to base 10 = $5(8^2) + 0(8^1) + 3(8^0) = 323$

   We typically represent the base as a subscript so I might write 503_8.

   How to convert from base 10 to base b:

   1723 in base 10, think about converting to base b:

   1723 = $d(k)b^k + d(k-1)b^k + \ldots d(1)b + d(0)$

   1723 % b = $(d(k)b^k + d(k-1)b^k + \ldots d(1)b + d(0)) \% b = d(0)$

   If I mod a number in base 10 by b, that will reveal the last digit of that number in base b.

   1723 / b = $(d(k)b^k + d(k-1)b^{k-1} + \ldots d(1)b + d(0))/b$ (int div)
   1723 / b = $d(k)b^{k-1} + d(k-1)b^{k-2} + \ldots d(1)1 + 0$

Integer divide by b lops off the last digit of the number in base b.

8 | 1723
8 | 215 R 3
8 | 26 R 7
8 | 3 R 2
0 R 3

$1723_{10} = 3273_8$

2. Big-Oh Intuition

O(f(n)) represents a class or set of functions, not just one function. Usually, this isn't the common use. Usually, will simply say that some function $f(n) = O(n^2)$, which simply means that f(n) belongs to the class of functions described by $O(n^2)$. Furthermore, what they mean is that f(n) is more complicated looking but the "important part of it" is n^2.

f(n) = O(g(n)) if only if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$

for some constant c.

(g(n) is an upper bound for f(n) within a constant)

f(n) can't exceed some constant times g(n) as n grows large.

What this really means is that if I am given an ugly complicated function, what I would like to do is find which Big-Oh class it belongs to, stripping the function of most of its ugliness, focusing on the portion of it that dominates the value of the function.

$f(n) = 6n^2 + 3n - 2 = O(n^2)$

$6n^2$ ← dominating term

In terms of big-oh, $3n^2$ and $17n^2$ are treated the same.

Here is a short chart of different functions in terms of big oh:

1, n, n^2, n^3, etc. Basically polynomial terms, the higher your exponent the "bigger" your big oh class is.

lg n grows slower than n^c for any positive constant c. In terms of big oh, all logs are the same regardless of base, which is why in textbooks, they leave out the base in logs for big oh.

1, lg (lg n), lg n, $(\lg n)^2$, sqrt(n), n, nlgn, nsqrt(n), n^2, $n^2 \lg n$.

Any exponential grows faster than any polyonomial, so n^k grows faster than c^n, provided that k is constant and $c > 1$. Different bases for exponentials means different big-ohs, so 8^n does NOT equal 2^n.

$f(n) = 3n + 17n\lg n + n^2 + 5(2^n) = O(2^n)$

Why do we use Big-Oh in CS?

We want a tool to judge how fast an algorithm might run.
But, counting exactly the number of tiny steps an algorithm might do is near impossible. So, instead, all we hope to do, is count the number of steps within a constant factor. The other thing is that different steps take different speeds on different computers, so it would be hopeless to make exact predictions.

The real goal of big-oh analysis is as follows:

1. Analyze an algorithm to figure out its big-oh.
2. Use that information to predict run times, to see if this solution is viable for us.
3. Perhaps use it to compare competing algorithms to solve a problem to decide which one to use.

Typically, we say that algorithms that run in a polynomial amount of time in the input size are "reasonable" while those that are exponential are not.

3. Summations - tool we need to figure out the big oh of an algorithm.

why we need summations:

Most code involves repetition. Counting the number of steps in repeating code requires us to add a bunch of numbers. Summations are the formal tool to add a bunch of numbers.

Simple example.

```
for (int i=0; i<n; i++) {
   scanf("%d", &array[i]);
   sum += array[i];
}
```

Detailed - counting # of simple steps:

1. i = 0
2. check if i<n, scanf, sum +=, i++ (repeat this n times)
3. check i<n one last time

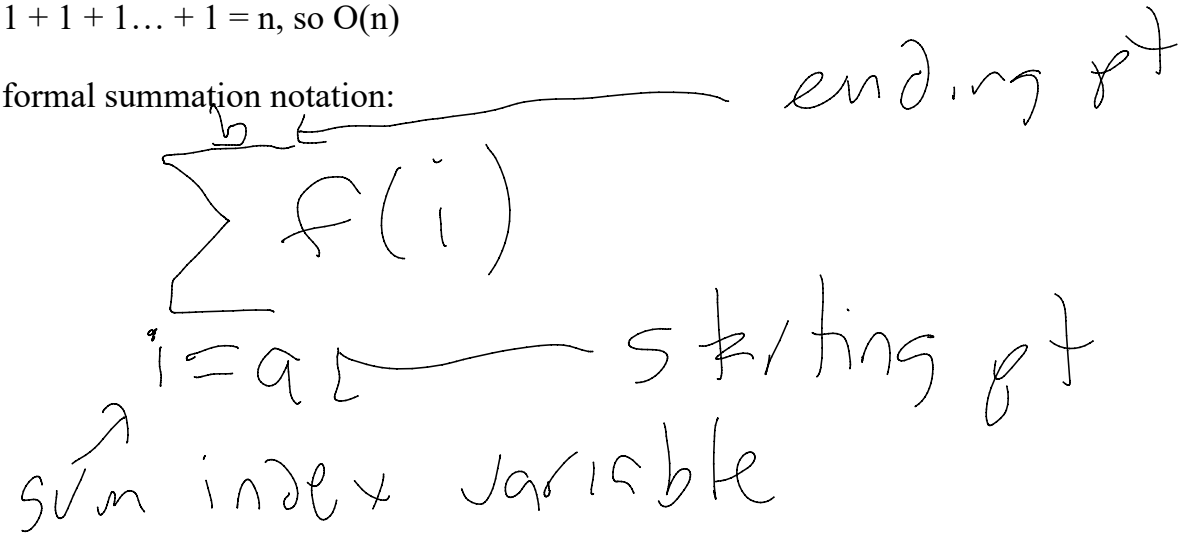
Counting the steps: 1 + 4*n + 1 = 4n + 2

Normally, we won't bother to go to this level of detail.

All we'll say is - hey, we're doing a constant amount of work in the loop, and we are repeating this n times.

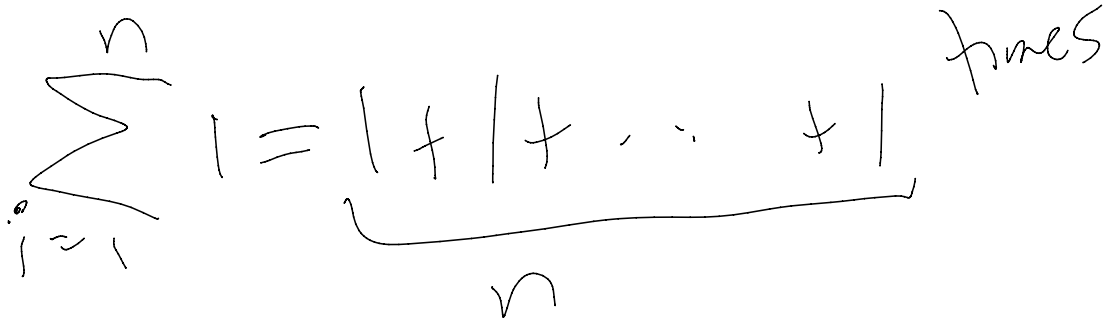
1 + 1 + 1 + 1… + 1 = n, so O(n)

So, formal summation notation:



This is short hand notation for: f(a) + f(a+1) + f(a+2) + … + f(b)

The sum that represents the run time of the code I just showed you is




```
for (int i=0; i<n; i++) {
```

```
for (int i=0; i<n; i++) {
    for (int j=0; j<=i; j++) {
        // Simple statement in here.
    }
}
```

$$\sum_{i=0}^{n-1}\left(\sum_{j=0}^{i} 1\right) = \sum_{i=0}^{n-1}(i+1) \quad \text{index shift}$$

$$= 1+2+3+\cdots+n = \sum_{i=1}^{n} i$$

Derive a few formulas:

1. $\sum_{i=a}^{b} c, \quad c = \text{const}$

2. $\sum_{i=1}^{n} i$

#1's $\underbrace{c+c+c+\cdots+c}_{b-a+1 \text{ times}}$

$b - a + 1$ times

$$\sum_{i=a}^{b} c = c(b-a+1), \quad b \geq a.$$

#2: $S = 1 + 2 + 3 + \cdots + n$ (+1 pairs)

$$S = n + (n-1) + (n-2) + \cdots + 1$$

$$2S = (1+n) + (1+n) + (1+n)$$

$$\frac{2S}{2} = \frac{n(1+n)}{2}$$

$$S = \frac{n(n+1)}{2}$$

Other formulas

$$\sum_{i=a}^{b} (f(i) + g(i))$$

$$\sum_{i=a}^{b} \quad \sum_{i=a}^{b} g(i)$$

$$\sum_{i=a}^{b} f(i) + \sum_{i=a}^{b} g(i)$$

Proof is based on commutative property of addition (adding same stuff in a different order)

$$\sum_{i=a}^{b} c f(i) = c \sum_{i=a}^{b} f(i)$$

Factoring…

if $1 < a < b$

$$\sum_{i=a}^{b} f(i) = \sum_{j=1}^{b} f(i) - \sum_{i=1}^{a-1} f(i)$$

LHS = f(a) + f(a+1) + f(a+2) + f(a+3) + …. + f(b)

RHS = (f(1) + f(2) ..+f(a-1)+f(a) + f(a+1) + … + f(b))
- (f(1) + f(2) ..+f(a-1))
- = f(a) + f(a+1) + … + f(b)

$$\sum_{i=n+1}^{2n} (3i+7) = \sum_{i=n+1}^{2n} 3i + \sum_{i=n+1}^{2n} 7$$

$$= \left(3\sum_{i=1}^{2n} i - 3\sum_{i=1}^{n} i\right) + 7(2n-(n+1)+1)$$

$$= \frac{3(2n)(2n+1)}{2} - \frac{3n(n+1)}{2} + 14n$$

$$= 6n^2 + 3n - \frac{3n^2}{2} - \frac{3n}{2} + 14n$$

$$= \boxed{\frac{9}{2}n^2 + \frac{31}{2}n}$$

Timing Problems

Making estimates for run time based on the big-oh run time of an algorithm.

Some algorithm has a run time of $O(n^2)$. For an input size of n = 4, the algorithm takes 10 ms. How long will it take for an input size of n = 16?

If a run-time is $O(n^2)$ we can model its run time as $f(n) = cn^2$, for some constant c. In order for this to be true, the Big Oh given has to be a tight upper bound. For this problems we will assume a tight upper bound.

$f(4) = c4^2 = 10$ ms --> so $c = 10/4^2$ ms

$f(16) = c16^2 =$ $\frac{10}{1.2} \times 16^2$ ms

$4^2 \times 10$

$= 10 \times \left(\frac{16}{4}\right)^2$ ms

$= 10 \times 4^2$ ms

$= 160$ ms

One thing to note: a run time can be based on more than one variable in the input.

Consider this problem:

An algorithm that processes an n by m array runs in O(nlgm) time. For an input with n = 2000 and m = 64, the algorithm takes 250 ms. How long do we expect the algorithm to take on an array with n = 500 and m = 2^{20}?

Let the runtime of the algorithm be f(n,m) = cnlgm for some const c.

f(2000,64) = c(2000)(lg 64) = 250 ms
c(2000)(6) = 250 ms
c = 250/12000 ms

f(500, 2^{20}) = (250/12000 ms)(500)(lg 2^{20})

$= \frac{250}{2000 \cdot 6} \cdot 500 \cdot 20 = \frac{2500}{12}$

$416\frac{2}{3}$ ms

$208\frac{1}{3}$ ms

Loop Examples - Sums

1. $\sum_{i=1}^{n} \left(\sum_{j=1}^{n} 1 \right) = \sum_{i=1}^{n} n = n^2$

1. $\sum_{i=1}^{n}\left(\sum_{j=1}^{n} 1\right) = \sum_{i=1}^{n} n = n^2$

2. $\sum_{i=1}^{n} 1 + \sum_{i=1}^{n} 1 = n + n = O(n)$

3. We start at n, and keep on dividing (int div) by 2 until we get to 0. So we have n, n/2, n/4, … after k steps we have $n/2^k$. We want to know the value of k for which $n/2^k = 1$
   $n = 2^k$ --> $k = \lg_2 n = O(\lg n)$.

4. Sums don't work here either, key is to realize that j++ runs a max of n times and same for i++ so the upper bound is n+n = O(n).

6. $\sum_{i=1}^{n}\left(\sum_{j=1}^{n-i} 1\right)$ or

$$\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2} = O(n^2)$$