Week 4 Notes

Prof Bill - Apr 2018

Week 4 notes covering:

A. Recursion

B. Sort and search algorithms

C. Algorithm analysis, Big-O

thanks… yow, bill

A. Recursion

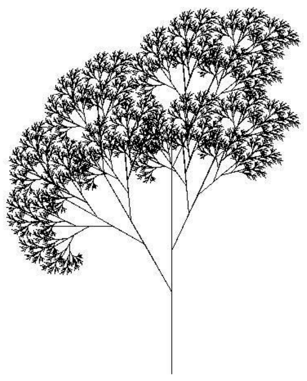
**** Book: Muganda Ch 16**

16.1 Intro

Java definition - a **recursive method** that calls itself
recursive data structure - self-referential data structures (like linked lists or trees); these structures can often be intuitively accessed with recursive code

More generally - **Recursion** occurs when a thing is defined in terms of itself or of its type, en.wikipedia.org/wiki/Recursion

"Tree (to the right) created using the Logo programming language and relying heavily on recursion. Each branch can be seen as a smaller version of a tree."
- en.wikipedia.org/wiki/Recursion_(computer_science)



We find recursion in mathematics and nature:

- Fractals are recursive objects, en.wikipedia.org/wiki/Fractal
- There's lots of recursion in nature… this is an interesting overview, www.nilsdougan.com/?page=writings/onfractals
- The most famous fractal is the Mandelbrot set, en.wikipedia.org/wiki/Mandelbrot_set
- In maths, recurrence relations are recursive equations, en.wikipedia.org/wiki/Recurrence_relation

16.2 Solving Problems

Recursive solution has two parts:

- ➢ **base case**: fixed value, end of the recursion
- ➢ **recursive case**: problem/function defined in terms of itself

So, solution steps: 1) identify base case (end of recursion), and recursive case (making problem successively smaller). Recursion must get smaller or it blows up.

Factorial example:

```
0! = 1     // base case
n! = n * (n-1)!     // recursive case
```

Other examples (Muganda Ch 16 challenges:

- Recursive multiplication: n*m = n + (n*(m-1))
- Recursive power: x^y = x * x^(y-1)

16.3 Examples

Recursive sum: value0 + sum(value1...valueN)   // base case? recursive case?
Fibonacci: again, base case? recursive case?

```
F0 = 0
F1 = 1
F(N) = F(N-1) + F(N-2)
```

16.4 Recursive Binary Search

BTW - binary search is O(log n).
Pseudocode:

```
// array must be sorted!
int binarySearch( array, value, first, last)
    if last < first
        return not found
    mid := (first + last) / 2
    if a[mid] = value
        return mid
    if value < a[mid]
        return binarySearch(a, value, first, mid-1)
    else
        return binarySearch(a, value, mid+1, last)
```

16.5 Towers of Hanoi

Yup.

Extra - Prof Bill

Recursion vs. iteration.

- Recursion often provides a more logical, elegant solution
- Iteration is faster! (see Homework #4)

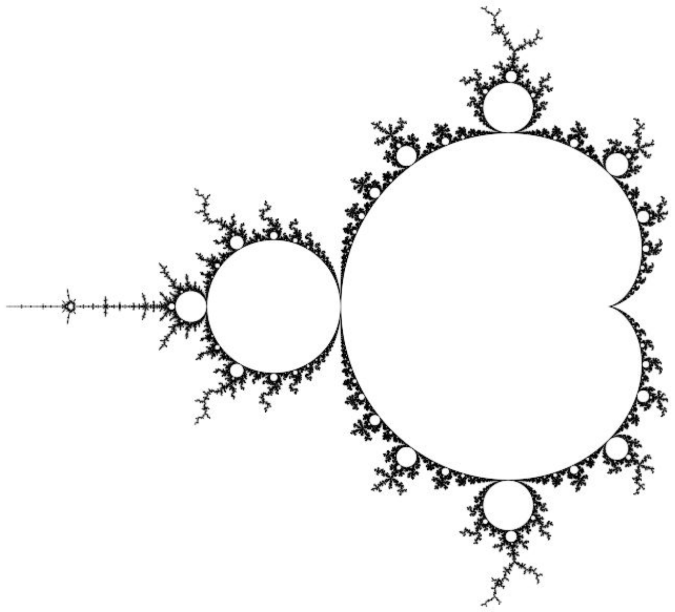
Why is recursion so much more expensive than iteration (220 students)?

tail recursion: when the recursive function call(s) is at the end
Tail recursion is important because 1) it’s pretty common, and 2) it’s usually easy to convert tail recursion to iteration. Motivation: iteration is faster than recursion!
Fibonacci is a good example of this.
Binary search too, www.codecodex.com/wiki/Binary_search#Pseudocode

```
function binarySearch(a, value, left, right)
    while left ≤ right
        mid := floor((right-left)/2)+left
        if a[mid] = value
            return mid
        if value < a[mid]
            right := mid-1
        else
            left  := mid+1
    return not found
```

thanks… yow, bill

PS - Next lecture, class field trip… we all get Mandelbrot set tattoos,
www.askideas.com/10-mandelbrot-tattoo-designs/



B. Sort and Search

**** Book: Muganda Ch 17.1-17.2**

17.1 Sorting Algorithms

Sorting:

- ➔ **Bubble sort** - lots of swaps
- ➔ **Selection sort** - one swap per pass
- ➔ **Insertion sort** - insert each item into position, swap to move others down

Here's a great way to watch the difference between these (pretty similar) sorting algorithms… our favorite animation/visualization site!

www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

The structure of these (similar, again) algorithms is a nested for loop:

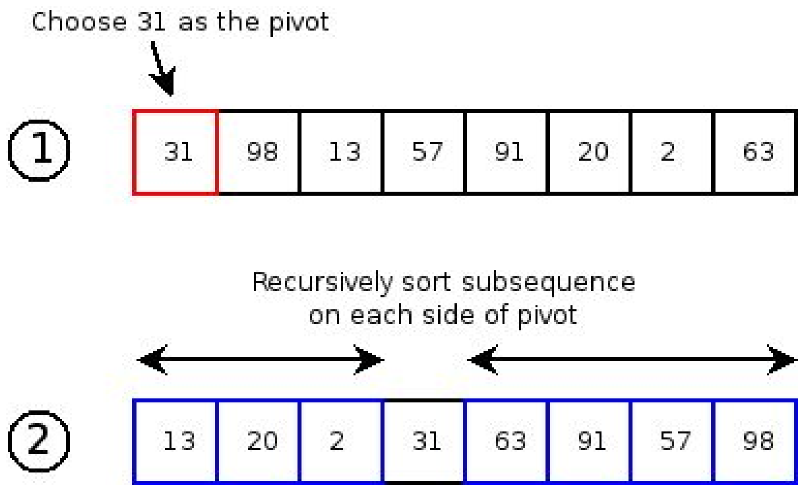
```
for( each element in the array)
   for( every other element in the array or so)
      do work: compare, swap, etc
```

Use Comparable to sort Objects.

As such, these sorting approaches are all $O(n^2)$. (yuk)
Question: How much slower is sorting for N1=10,000 versus N2=100?

Quicksort is the $$$; average performance is O(N log N).
Question: How much slower is Quicksort on N1=10,000 versus N2=100?

Partition, pivot - Quicksort places pivot in spot so that all values below pivot in the array are <= the pivot value, and all values above are > pivot value.



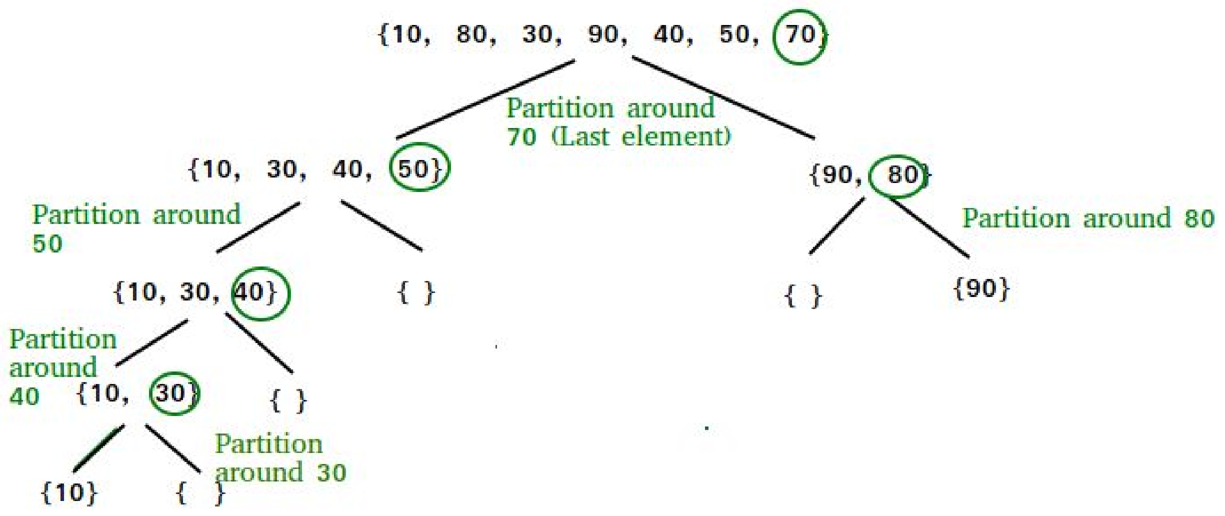
Source: faculty.ycp.edu/~dhovemey/fall2005/cs102/lecture/11-1-2005.html

I like the example and the pseudocode here: www.geeksforgeeks.org/quick-sort/

1) Quicksort pseudocode

```
// recursive quicksort algorithm
quickSort(arr[], low, high)
    if (low < high)
        // pi is partitioning index, arr[p] is now correct
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi


// swap within array range so everything below pivot is <= and above is >
// pivot index is returned
int partition (arr[], low, high)
    pivot = arr[high];    // pivot, last item
    i = (low - 1)    // index of smaller element

    for (j = low; j <= high- 1; j++)
        if (arr[j] <= pivot)    // if current is <= pivot
        i++;
        swap arr[i] and arr[j]

    swap arr[i + 1] and arr[high])
    return (i + 1)
```

2) Quicksort example



One problem with Quicksort. It's not **stable**. A stable sort is one where two equal objects are left in the same order in sorted output as they appear in the input array. www.geeksforgeeks.org/stability-in-sorting-algorithms/

17.2 Search Algorithms

Search this:

- ➔ Sequential search - O(N)
- ➔ Binary search - O(log N)

Binary search

- ❑ Array must already be sorted!
- ❑ Divide problem size in half with each iteration, hence the O(log N)

Pseudocode (source: www.codecodex.com/wiki/Binary_search):

```
// recursive binary search, returns item if found
binarySearch(a, value, left, right)
    if right < left, then return not found
    mid = (right + left)/2
    if a[mid] == value
        return mid
    if value < a[mid]   // smaller value means search left
        return binarySearch(a, value, left, mid-1)
    else   // larger value means search right
        return binarySearch(a, value, mid+1, right)
```

Since binary search is tail recursion, we can (pretty easily) make it iterative.

```
// iterative binary search
binarySearch(a, value, left, right)
    while left ≤ right
        mid = (right + left)/2
        if a[mid] == value
            return mid
        if value < a[mid]   // smaller value means search left
            right := mid-1
        else    // larger value means search right
            left  := mid+1
    return not found
```

C. Algorithm Analysis, Big-O

**** Book: Muganda Ch 17.3**

17.3 Algorithm Analysis

Big-O!

We can estimate the efficiency of an algorithm...

Our analysis is different than **benchmarking**, which involves running your algorithm and measuring it. Disadvantages to benchmarking... measurements will be impacted by many “real world” factors, including:

- ❖ Programming language and compiler choices
- ❖ Computer and operating system you’re using
- ❖ Load factor on your system while running
- ❖ Input values used during testing

Another disadvantage to benchmarking… Coding! Coding is work, and an algorithm must be implemented in code to benchmark it.

But look at the glass half full… without even coding up our algorithm, analysis gives us a measurement that is independent of any sticky issues like what computer you use or what data you run. This is powerful!

Algorithm analysis provides for a theoretical bounds for the performance of an algorithm in relation to the size of its inputs.
Here’s a fancier way of saying this:

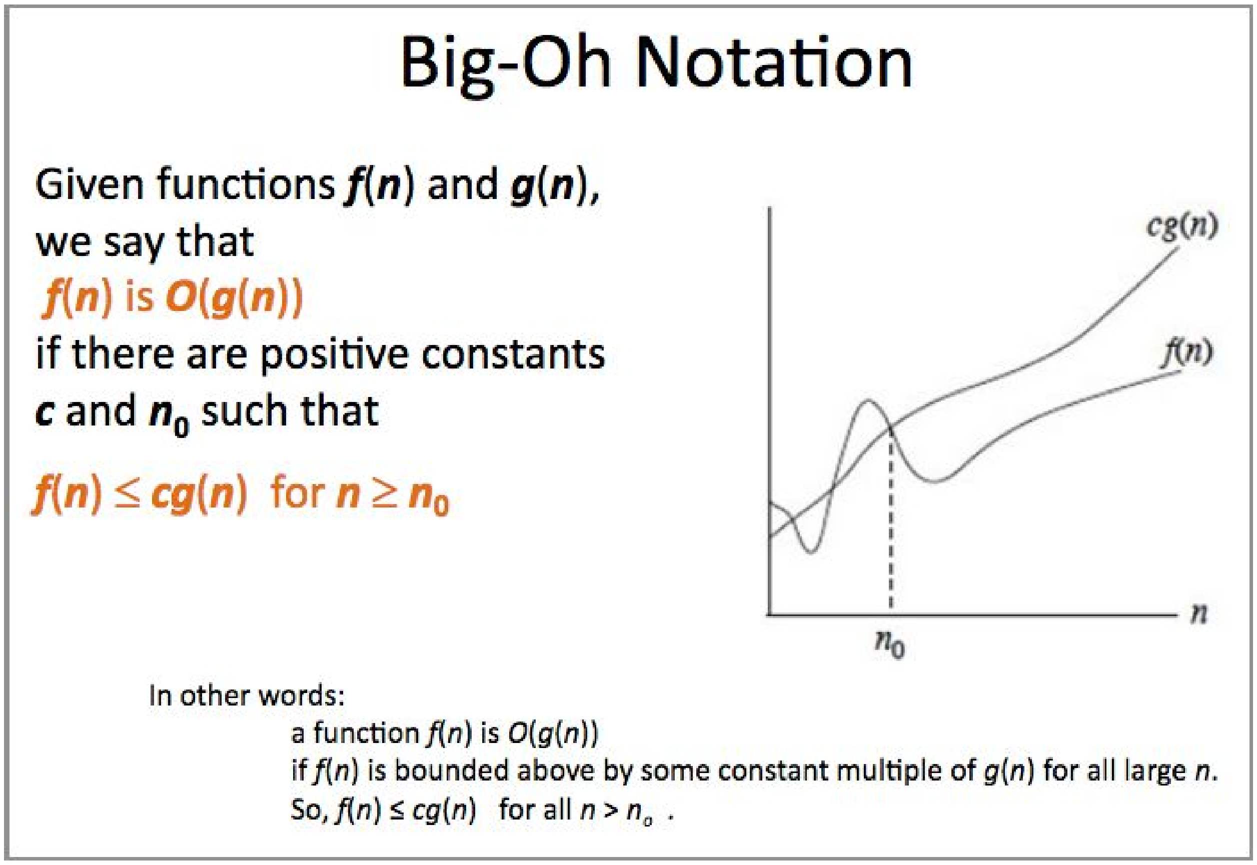
> the analysis of algorithms is the determination of the computational complexity of algorithms, that is the amount of time, storage and/or other resources necessary to execute them. Usually, this involves determining a function that relates the length of an algorithm's input to the number of steps it takes (its time complexity) or the number of storage locations it uses (its space complexity).
>
> - en.wikipedia.org/wiki/Analysis_of_algorithms

Worst case complexity vs. **average case complexity** - which is more important?
It depends. How common is the “worst case”. We see that hash table are O(N) in the worst case, but we take all measures (good hash function, resizing) to avoid worst case.

Try to categorize asymptotic performance of our algorithm with Big-O.
We say the complexity of an algorithm f(n) is

```
    O(g(n) if f(n) <= c*g(n), for n>n0
```

Maybe this is better.



Big-Oh Notation

Given functions $f(n)$ and $g(n)$,
we say that
$f(n)$ is $O(g(n))$
if there are positive constants
c and n_0 such that

$f(n) \leq cg(n)$ for $n \geq n_0$

In other words:
a function $f(n)$ is $O(g(n))$
if $f(n)$ is bounded above by some constant multiple of $g(n)$ for all large n.
So, $f(n) \leq cg(n)$ for all $n > n_0$.

Source: UW-Stout, http://slideplayer.com/slide/8348651/

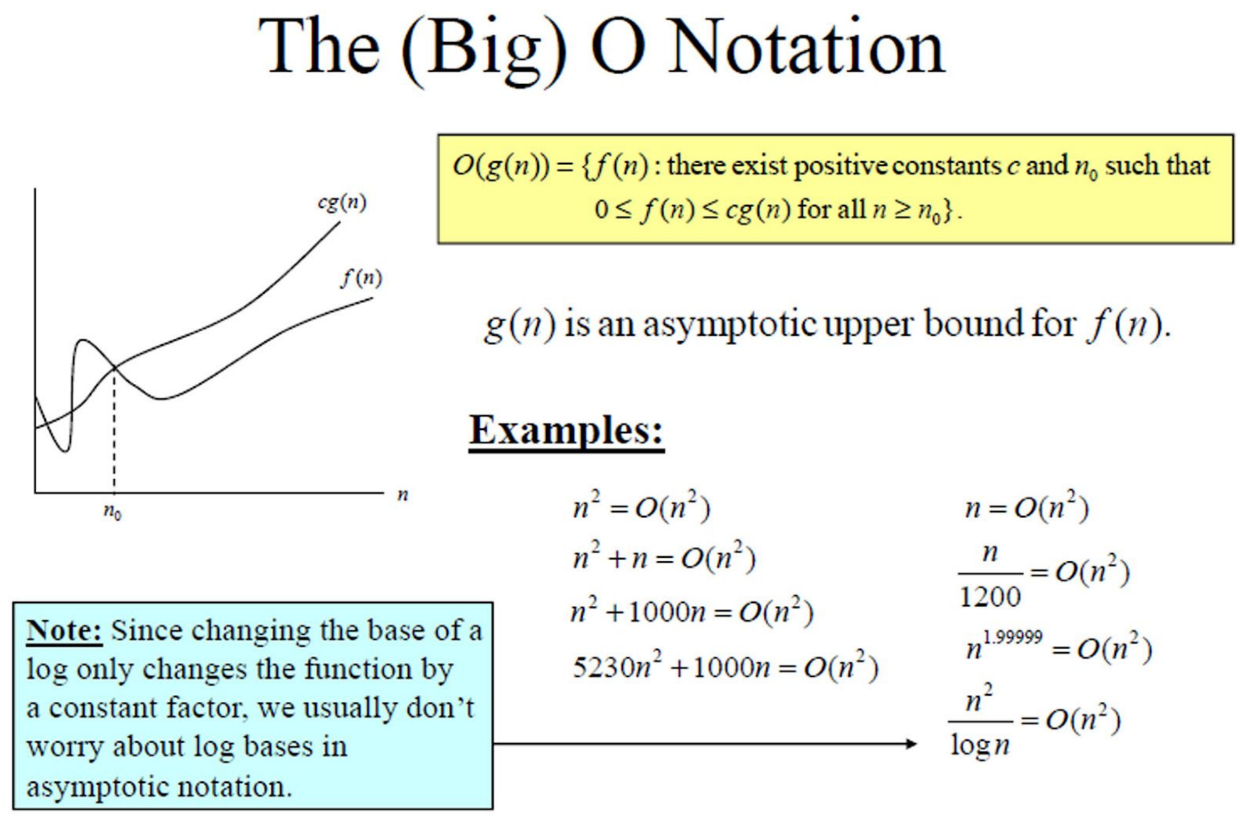
Big-O is an **upper bound** of our algorithm's performance! This is important, so that we can answer the question… performance will never be worse than this g(n).

Muganda: Another way... f(n)/g(n) becomes small or disappears as N gets large

Morin: "Big-oh notation allows us to reason at a much higher level" (yes!)

We can remove constants and multiples from our f(n) when categorizing using Big-O. These factors are insignificant when N is large.

- $32n^2 = O(n^2)$
- $n^2 + n\ 170 = O(n^2)$



The (Big) O Notation

$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$

$g(n)$ is an asymptotic upper bound for $f(n)$.

Examples:

$n^2 = O(n^2)$

$n^2 + n = O(n^2)$

$n^2 + 1000n = O(n^2)$

$5230n^2 + 1000n = O(n^2)$

$n = O(n^2)$

$\frac{n}{1200} = O(n^2)$

$n^{1.99999} = O(n^2)$

$\frac{n^2}{\log n} = O(n^2)$

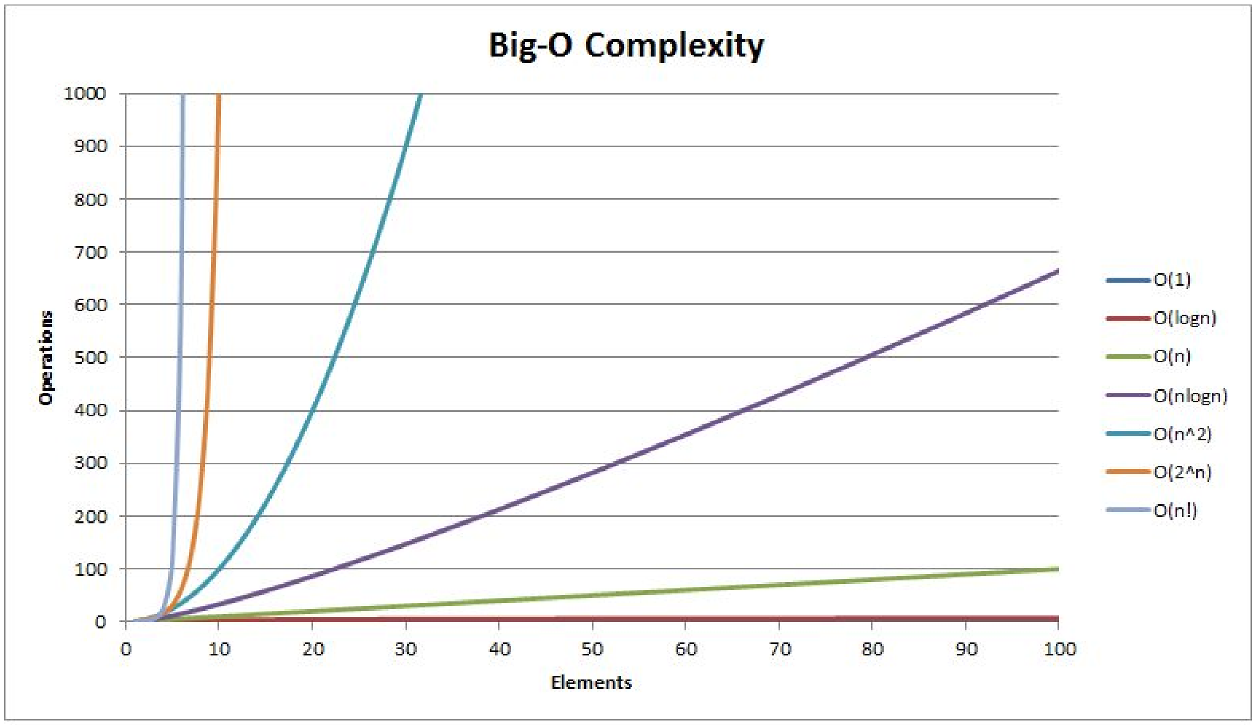
Note: Since changing the base of a log only changes the function by a constant factor, we usually don't worry about log bases in asymptotic notation.

Source: meherchilakalapudi.wordpress.com/category/data-structures-1asymptotic-analysis/

Big-O rate of growth functions, in order:

- **O(1) = constant time**; example is hash table get/put
- **O(log N) = logarithmic time**; example is binary search
- **O(N) = linear time**; example is sequential search
- **O(N log N) = "n log n time" or linearithmic**; example is Heapsort
- **$O(n^2)$ = quadratic time**; example is bubble sort
- **$O(n^3)$, $O(n^4)$, etc = polynomial time**; example is triple loop
- **$O(2^N)$ = exponential time**; example is recursive Fibonacci

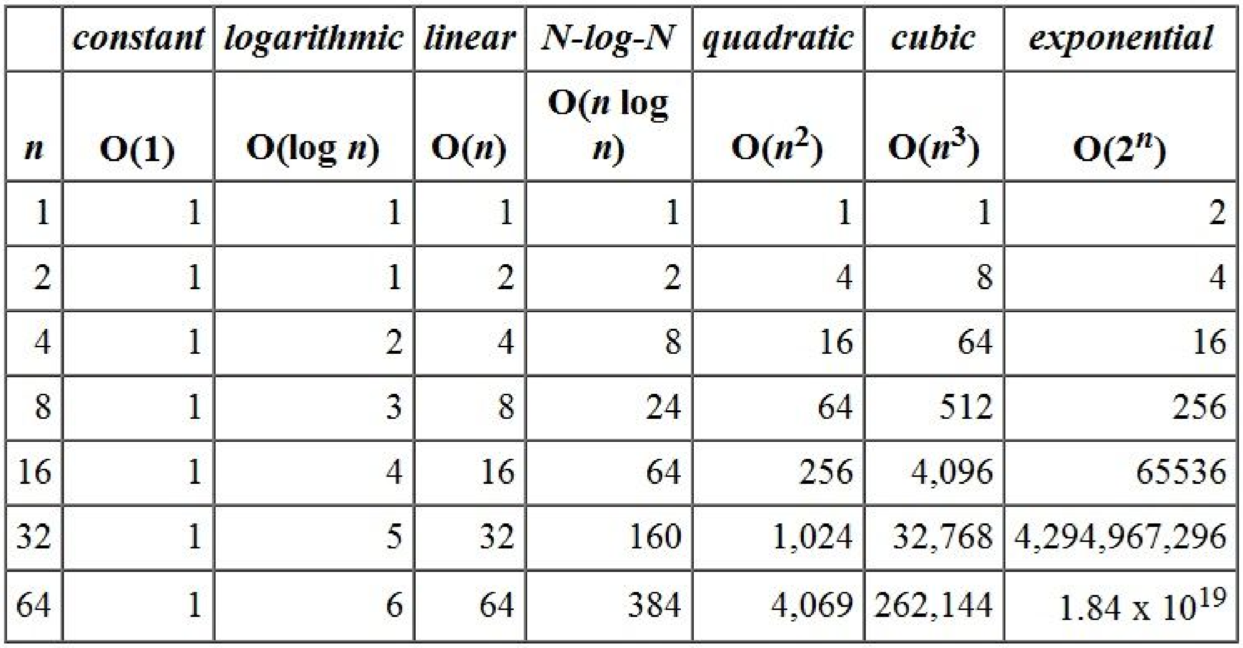
Always remember - our “most important chart of 210”... huzzah!



Source:www.hackerearth.com/practice/notes/big-o-cheatsheet-series-data-structures-and-algorithms-with-thier-complexities-1/

For all the hand-waving, Big-O essentially tries to fit an algorithm into one of these complexity buckets, from constant time to exponential. If we can do this, then we have a good upper bound and good, quick description of the performance of our algorithm. This is very valuable.

Big-O functions at various values for N.



	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Source: www.cpp.edu/~ftang/courses/CS240/lectures/img/alg-tab.jpg

This is fun too: bigocheatsheet.com/

Extras

These are other "Big" functions that are less important to us.

- **Big Omega, Ω(f(n))** - defines the lower bound of performance for an algorithm

- **Big Theta, Θ(n)** - provides a tighter (upper and lower) bound for the performance of an algorithm

You can read more here: www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation

Finally…
Big-O is a part of one of the most significant unsolved problems in math/computer science today. It’s known as the **P vs. NP Problem**. If you can publish a solution, these guys will give you a million bucks.

> www.claymath.org/millennium-problems/p-vs-np-problem
>
> *In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.*

P problems are those where solutions are easily found (polynomial time, hence the “P”).
NP problems are those that are easy to verify (in polynomial time), but hard to solve.
A large set of problems are called **NP-complete** as they can be easily verified, but no one has proven that they are hard to solve yet.

www.claymath.org