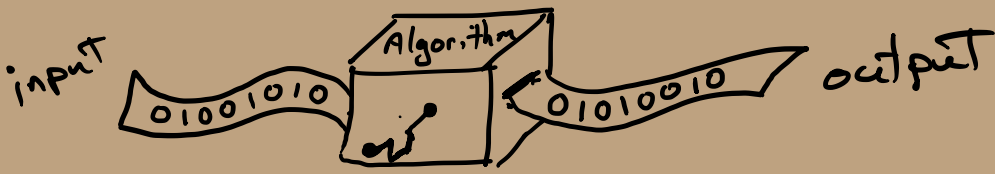
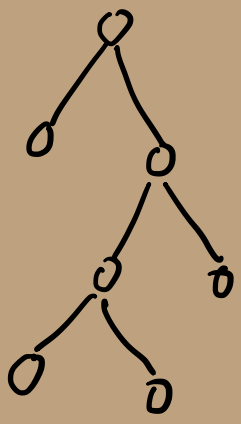
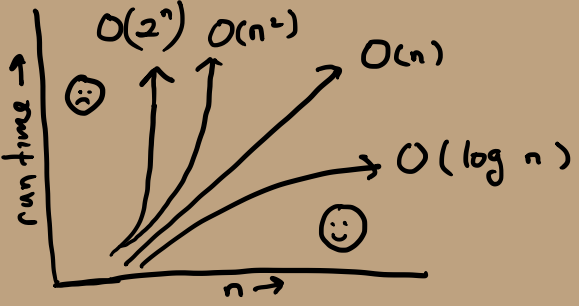


Algorithms: Search & Sort
and the growth of functions.





John Rachlin
CS 5200
Northeastern

① What is an <u>algorithm</u>?

a) A step-by-step procedure for solving a problem, or accomplishing some task.

b) A recipe for "cooking" a solution (output) from ingredients (input)

② Algorithms impact our lives in countless ways:

a) Choosing a route to follow to go from home to school.

b) When we search the internet, computer algoriths dictate what search results we see, and what advertisements we see as well!

c) Algorithms on "strings" drove the sequencing of the human genome

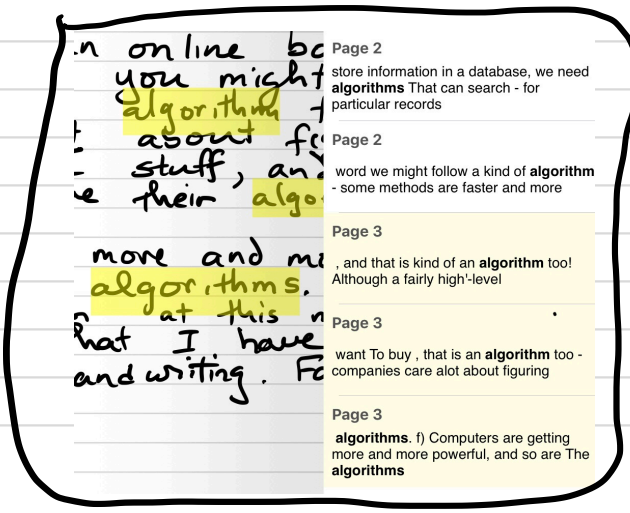
d) When we store information in a database, we need algorithms that can search for particular records efficiently or generate summaries by sorting, aggregating, and collating our data.

e) A dictionary is a kind of database too and when we look up a word we might follow a kind of algorithm - some methods are faster and more efficient than others - we will explore this further.

e) When we bake a cake, we may follow a recipe, and that is kind of an algorithm too! Although a fairly high-level one. It doesn't say: "Walk three steps to cupboard and get flour." Much is assumed and recipes work because you have a brain and can fill in lots of details. Computers have a sort of brain (the CPU), but it is mostly dumb and we have to explain every little detailed step and this is called "programming" or "Software engineering"

f) When a streaming movie service recommends a movie, or an online bookstore suggests a book you might want to buy, that is an algorithm too — companies care alot about figuring out how to sell more stuff, and may hire you to improve their algorithms.

f) Computers are getting more and more powerful, and so are the algorithms. The ipad app I am writing on at this moment can search for words that I have written — recognizing my handwriting!



Searching for the word "algorithm" in my notes. (I find this very impressive!)

③ Lots of factors impact the speed or "running time" of a program.

a) The size of the input. We expect it will take longer to sort 1 million words compared to say 10 words.

b) The algorithm: Some sorting algorithms, while simpler and more intuitive are provably slower (require more steps) all else being equal.

c) The programming language used: Languages have "trade offs" - interpreted scripting languages offer convenience, but run slower than compiled languages like C++

d) The coder's implementation: Quality code often runs faster than sloppy code - programming is an ART!

e) The hardware. The same program run on a supercomputer will run faster than on a laptop.

Computer Scientists mostly focus on the first two: How an algorithm works (does it do what it is supposed to - is it correct?) and how does its running time grow as a function of the input size.

④ Growth functions

In characterizing different algorithms as a function of input we generally ignore constants that reflect hardware, or quality of implementation. We focus on intrinsic properties of the algorithm.

The way the input data is presented to the algorithm might affect its running time too, so we sometimes distinguish:

Best Case: smallest number of operations possibly required. Sorting might be really fast if your data is already sorted. This case is usually trivial, rare, and uninteresting.

Average Case: Average # of operations assuming all inputs are equally likely.

- Requires assumptions
- More mathematically difficult to derive.
- Usually within a constant factor of worst case

our Focus ↓

Worst Case: Most operations required. Useful and more straight-forward.

So we characterize an algorithm by how its running time grows as a function of n, the input size.

function	interpretation		
1	constant		"efficient"
$\log n$	logarithmic		
n	linear	coined by	
$n \log n$	"linearithmic"	– Sedgewick	
n^2	quadratic	polynomial-time	
n^3	cubic		
$\vdots$			
n^k			
2^n	exponential		"intractable"
$n!$	factorial		

For some problems, like travelling salesman, we cant find an efficient solution, that produces a <u>guaranteed optimal</u>, such an algorithm probably doesnt exist, but we dont know for sure.

The differences between these numbers is significant as input size, n, grows.

Suppose 10,000 operations per second.

(For example, Twitter receives about this many tweets every second!)

function	$n = 10$	$n = 1000$	$n = 1{,}000{,}000$
$\log n$	1×10^{-4} sec	3×10^{-4} sec	6×10^{-4} sec
n	1 ms	100 ms	100 sec
$n \log n$	1 ms	3 sec	10 min
n^2	10 ms	100 sec	3.2 years
2^n	100 ms	3.4×10^{287} cent	Overflow

⑤ Search - Unordered Linear

i	1	2	3	4	5	6	7	8
A	19	23	1	4	77	12	38	42

↑ ↑ ↑ . . . .

- Search each element one at a time or until we reach the end.
- How many items do we have to inspect?

Best case: 1 item - its at the beginning?

Average Case: Assume all positions equally likely:

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

Doesn't consider Probability that item isn't in list.

Worst Case: We try all items: n

Note how worst case was simpler and with a factor of average. Both demonstrated that unordered linear search Running time is $\approx n$

Imagine looking up words in a dictionary this way!

⑥ Search - Ordered Linear

i	1	2	3	4	5	6	7	8
A	1	4	12	19	23	38	42	77

This helps a little - searching for 18, we can stop when we get to item 4.

But, overall, it is still:

Best : 1
Average : $(n+1)/2$
Worst : n - item bigger than last.

OPTIONAL

⑦ "Chunk" Search - (from Aslam book).

- Grab 1 chunk at a time.
- Test last element - is item maybe in chunk?
- If yes, linear search with that chunk
- If no, grab next chunk.

i	1	2	3	4	5	6	7	8
A	1	4	12	19	23	38	42	77

chunk 1 chunk 2

e.g Find 38

Not in chunk 1 (38 > 19)
Maybe in chunk 2 (38 < 77)

Linear search A[5...8]

A[5] ? No
A[6] ? Yes!

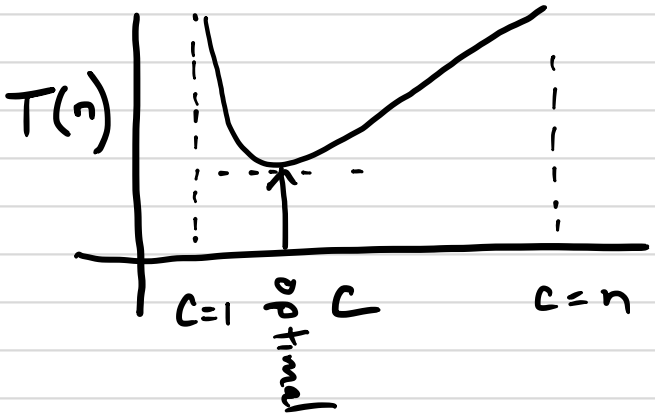
⑦ Chunk search (continued)

Let c be chunk size

$T(n)$ Running time.

$$T(n) = \underbrace{\frac{n}{c}}_{\text{find chunk}} + \underbrace{c}_{\text{linear search through chunk}}$$

What is optimal c?



Want: $T'(n) = \frac{d}{dc} T(n) = 0$

$$= \frac{d}{dc}\left(nc^{-1} + c\right) = \frac{-n}{c^2} + 1 = 0$$

$$1 = \frac{n}{c^2}$$

$$c^2 = n$$

$$c = \sqrt{n}$$

⑧ Binary Search

- We narrow our range by half with each iteration

$$n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots$$

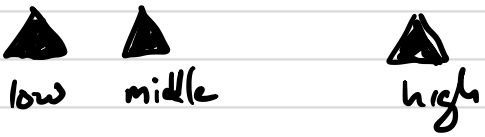
Search for "42"

i	1	2	3	4	5	6	7	8
A	1	4	12	19	23	38	42	77

Step 1

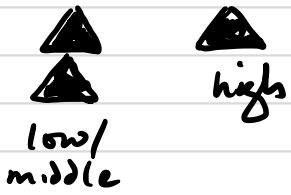
42 > 19

Step 2



42 > 38

Step 3



42 = 42 - found in 3 steps!

OPTIONAL

Pseudocode: Sketches of programs not written in a real language.

```
Binary-Search(A, key, low, mid, high)
    if (low > high) return NOT FOUND
else if A[mid] = key   return middle   // found!
else if A[mid] > key
        return Binary-Search(A, key, low, ⌊(low+mid)/2⌋, mid-1)
        // recurse on lower half
else    return Binary-Search(A, key, mid+1, ⌈(mid+high)/2⌉, high)
        // recurse on upper half.
```

(8) Binary Search - continued

Analysis: With each step we narrow our search range by half.

How many steps are required to narrow range down to one element?

$$n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{8}$$

1 step 2 steps 3 steps

So we want $\frac{n}{2^{steps}} = 1$

$$2^{steps} = n$$

$$steps = \log_2(n)$$

Search Algorithm	$n = 10^6$ # operations	Growth Func.
Linear	1,000,000	n
Chunk	1000	$\sqrt{n}$
Binary	20	$\log_2 n$

Logarithms (Inverse of Exponentiation)

$$\log_b(b^n) = n$$

$y = 10^x$ $\quad x = \log_{10} y$

$y = 10^{-3} = \frac{1}{1000}$ $\quad \log_{10} 10^{-3} = -3$

$\log(1) = 0$

$\log(1000) = 3$

$\log(10^6) = 6$

$\log(10^9) = 9$

$\log_2 216 = 8$

$\log_2 8 = 3$

$\log_2 1 = 0$

unsigned value x

$\lceil \log_2 x \rceil$ bits

In General

$y = b^x$ $\quad x = \log_b y$

$c = a \cdot b$ $\quad \log c = \log a + \log b$

$\log_b(a^n) = n \cdot \log_b a$

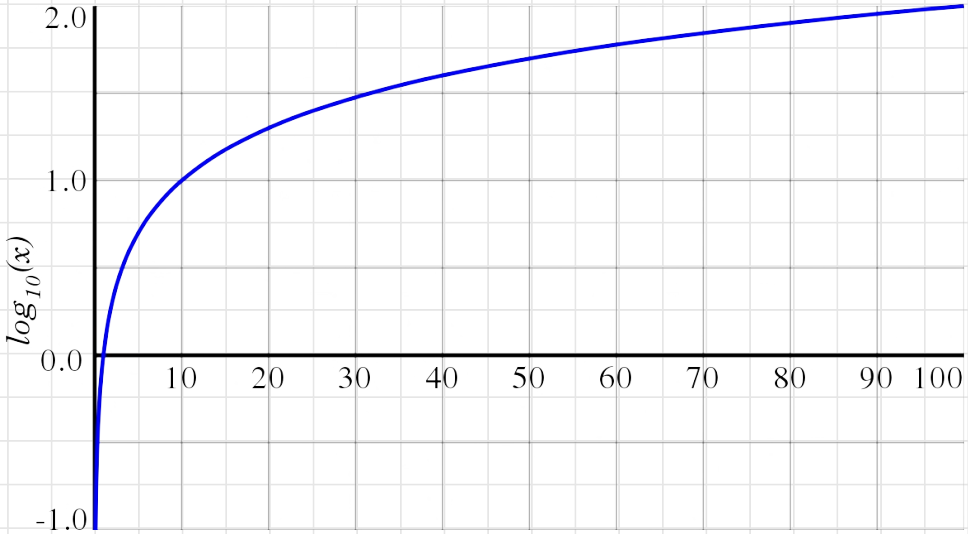
$\log_a x = \log_b x / \log_b a$

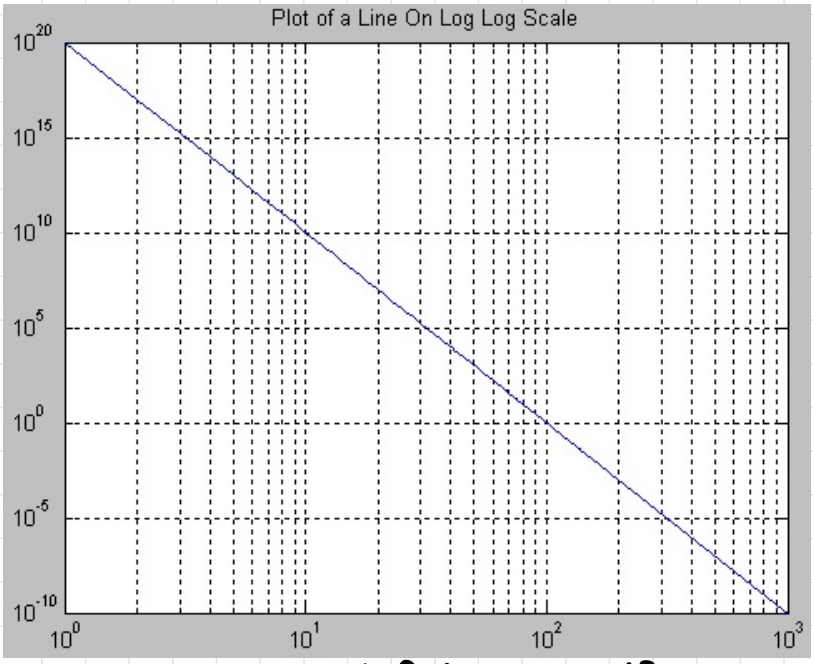
$$\log_2(256) = \frac{\log_{10}(256)}{\log_{10}(2)}$$

$$= \frac{2.408}{0.301}$$

$$= 8$$

$2^8 = 256$





facebook
instagram
twitter users

followers / friends / likes . . .

"scale-free"

$2^3 = 8 \iff \log_2 8 = 3$

$10^x = 1000 \iff \log_{10}(1000) = 3$

$10^x = 42 \quad \log_{10}(42) = 1.623$

$10^1 < 42 < 10^2$

$1 < x < 2$

$\log_b 1 = 0 \quad (b^0 = 1)$

$\log_2 \emptyset = x \quad$ (undefined)

because $2^x = 0$ has no solution

Natural Log : $\ln(x)$

Base: $e \approx 2.71828$

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = 1 + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \cdots$$

$$= \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n$$

n	$\approx e$
1	2
2	2.25
3	2.37
4	2.44
5	2.49
⋮	⋮
100	2.70

e.g. Account value w/ 100% interest compounded n times year.

Stirlings Approximation:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

$$\frac{d}{dx} e^x = e^x \quad \left(\text{slope @ } x = \text{value @ } x\right)$$

⑨ Sorting

Algorithm	Code complexity	Run-Time worst
insertion sort	easy	n^2
selection sort	easy	n^2
merge sort	moderate	$n \log n$
quick sort	complex	n^2 (worst-case) but $n \log n$ on average and very popular

- Bucket Sort, Counting sort, Radix sort achive linear performance but with special restrictions on input.

  (Non-comparison based)

- For comparison-based* methods, it can be proven that $n \cdot \log n$ is as good as it gets.

* Comparison-Based: Taking action based on comparing two values and deciding which is bigger or smaller

(10) Insertion sort.

a) Like sorting a hand of cards

b) Maintain sorted and unsorted.

c) For each unsorted element, insert into the right location by swapping with left neighbor until correct position found

19 | 23 1 4 77 12 38 42

19 | 23 1 4 77 12 38 42

19 23 | 1 4 77 12 38 42

19 23 1 4 77 12 38 42

swap

19 1 23

swap

1 19 23 | 4 77 12 38 42

1 4 19 23 | 17 12 38 42

1 4 17 19 23 | 12 38 42

1 4 12 17 19 23 | 38 42

1 4 12 17 19 23 38 | 42

1 4 12 17 19 23 38 42

(10) Insertion Sort - continued

Worst Case Performance: Items in reverse order

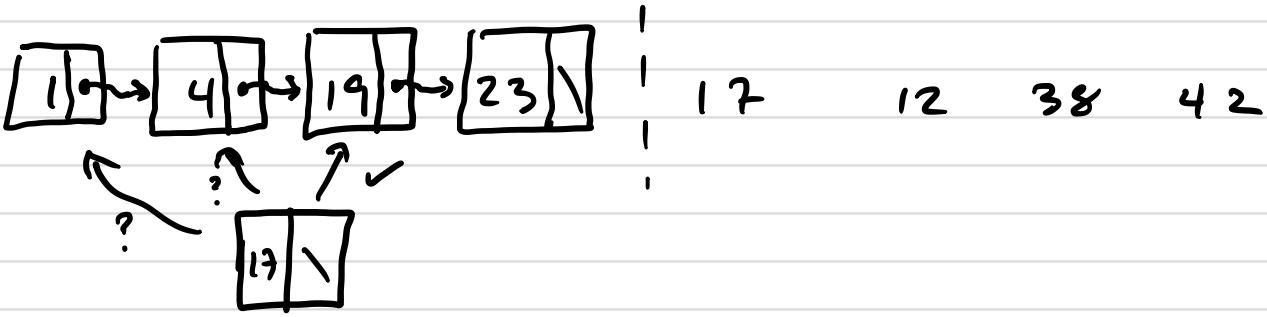
n items.

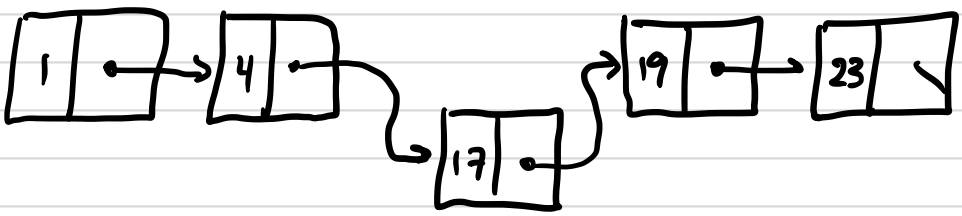
Total swaps $1 + 2 + 3 + \cdots + (n-1) = \frac{(n-1)n}{2}$

$\approx n^2$

We could avoid all the swaps by maintaining sorted part in a linked list.

But we still have to walk down a growing list to find the correct insertion point, so complexity is still $\approx n^2$





(11) Selection Sort

1. Find minimum element in list
2. Swap with 1st element
3. Find minimum in remaining $n-1$ elements
4. Swap with 2nd element
6. Repeat until all elements have been swapped into place.

$n=8$	19	23	(1)	4	77	12	38	42
Scan to find minimum — val:	19	19	1	1	1	1	1	1
Scan to find minimum — pos:	(1)	(1)	(3)	(3)	(3)	(3)	(3)	(3)
Scan $n-1=7$ elements	1	23 ↑	19	(4)	77	12	38	42
Scan $n-2=6$	1	4	19 ↑	23	77	(12)	38	42
$n-3=5$	1	4	12	23 ↑	77	(19)	38	42
$n-4=4$	1	4	12	19	77 ↑	(23)	38	42
$n-5=3$	1	4	12	19	23	77 ↑	(38)	42
$n-6=2$	1	4	12	19	23	38	77 ↑	(42)
	1	4	12	19	23	38	42	77

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

$$\approx n^2$$

(12) Merge Sort

A merge takes two sorted lists and combines them into one sorted list.

List A: 1 4 19 23

List B: 12 38 42 77

Merge (A, B) 1 4 12 19 23 38 42 77

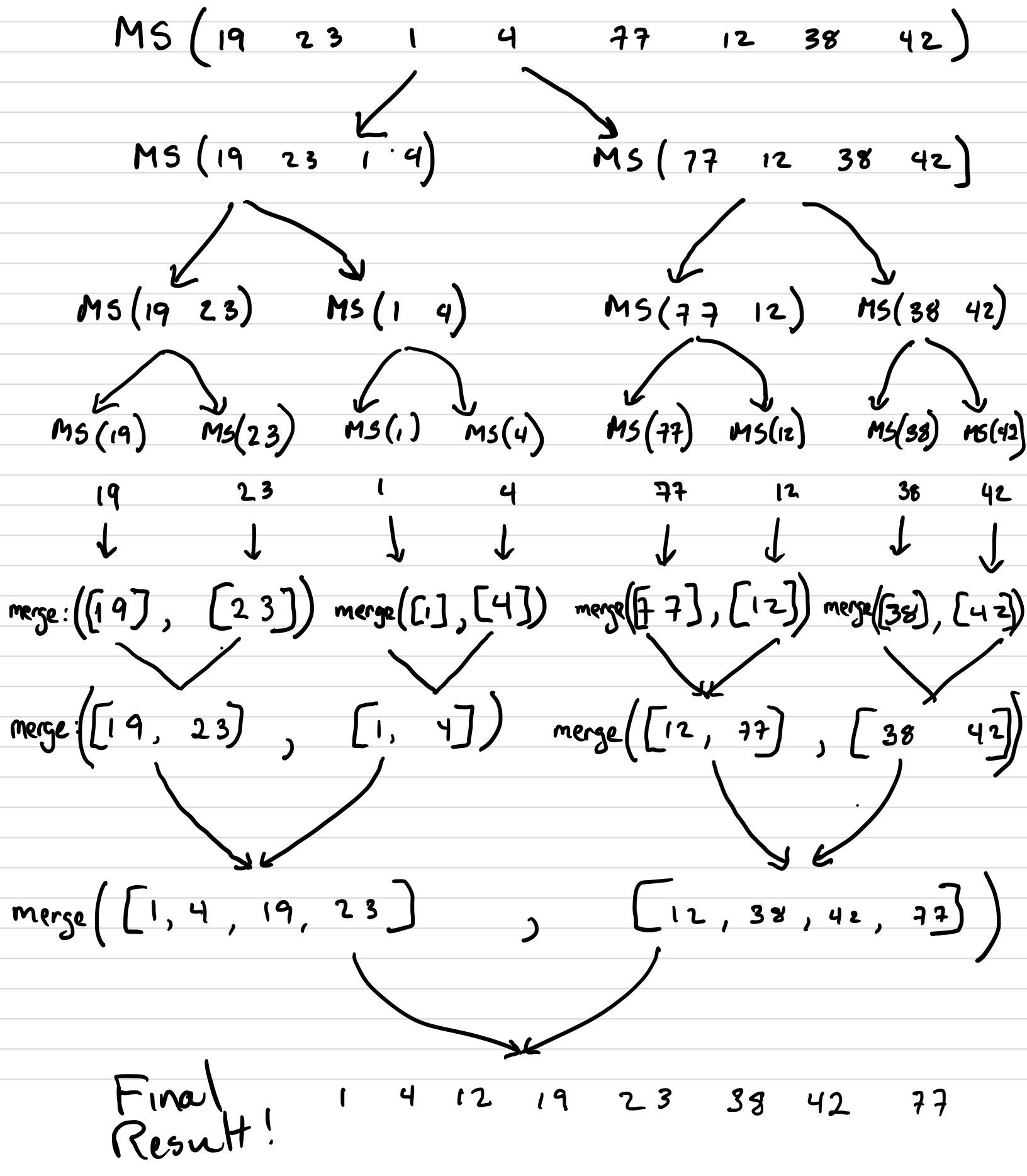
Method: Maintain two pointers/indexes, one on each list and walk down the lists.

Pick smallest value, advancing pointer for which ever list we pulled from

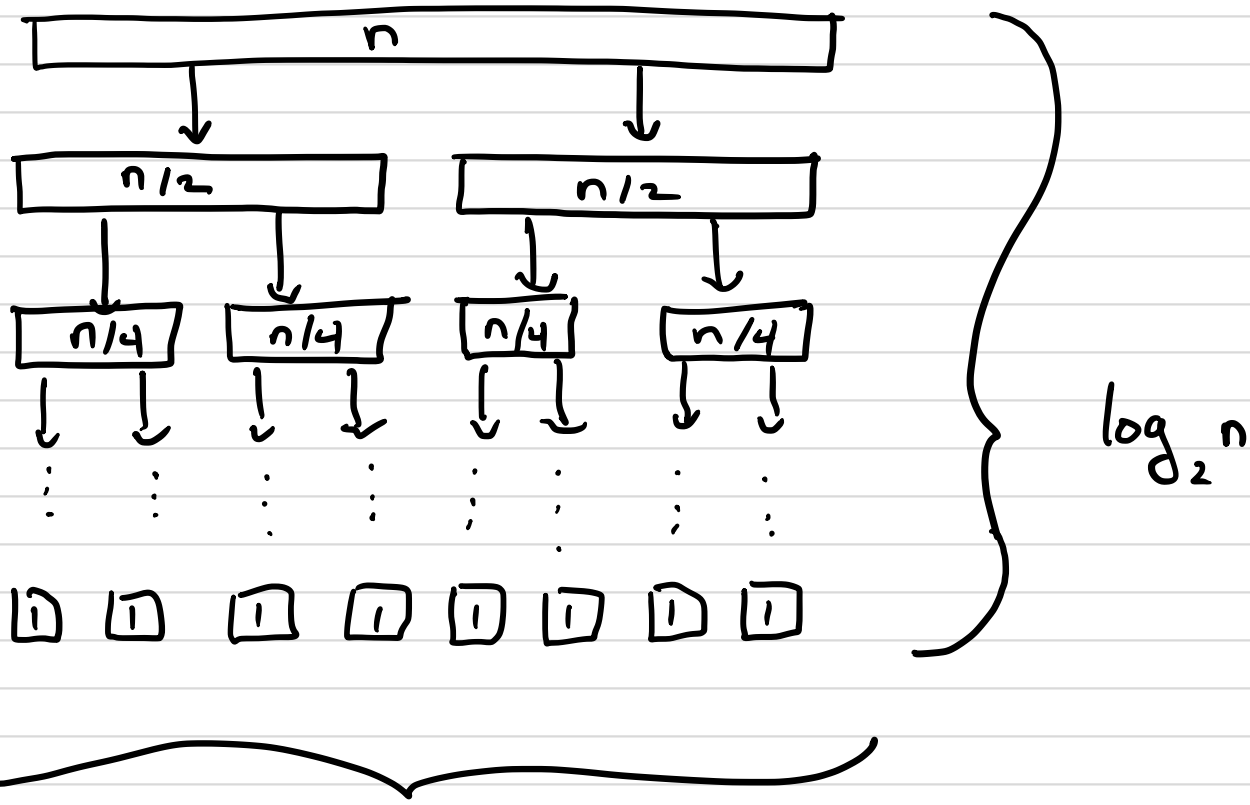
Pseudo code:

```
MergeSort (L):
    if length (L) ≤ 1 return L
    else
        A = MergeSort (left half of L)
        B = MergeSort (right half of L)
        return merge (A, B)
```

(13) Merge Sort - Demonstrated.



(14) Analysis of Merge Sort.



n nodes

$T(n)$ = running time.

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

calling mergesort twice on half of elements

total work required to merge at each level (n nodes)

$\simeq$ $n \log n$ (an informal argument)

Counting the operations in Merge Sort*

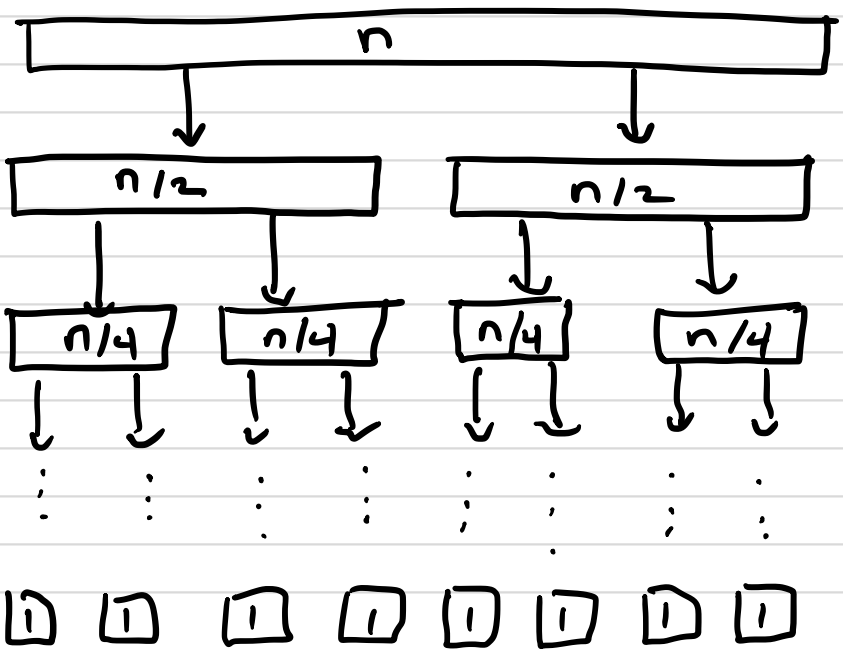
Lets take a closer look @ merge.

Input: A, B (length $n/2$) - sorted!
Output: M (length n)

```
1.  i = 1                                  } 2 operations
2.  j = 1                                  }
3.  for k = 1 to n                         } increment - 1 operation
4.      if A[i] < B[j] then                } compare - 1 operation
5          M[k] = A[i]                     }
6.         i = i + 1                       }   OR   2 operations
7.     else                                }        (5 & 6 or 8 & 9)
8          M[k] = B[j]                     }
9.         j = j + 1                       }
```

Total operations $\leq 4n + 2 \leq 6n$ an upper bound

$\leq 6n$ $(n \geq 1)$ also an upper bound!



	#sub-problems	size
Level 0	2^0	$n/2^0$
Level 1	2^1	$n/2^1$
Level 2	2^2	$n/2^2$
Level j	2^j	$n/2^j$

* See Roughgarden (2017) "Algorithms Illuminated"

Merge Sort Analysis - continued.

Level-j work = # of Level-j subproblems $\times$ work per level-j subproblem

$= 2^j \times 6n/2^j$

$= 6n$

Total work = number of levels $\times$ work/Level

$= \log_2 n + 1 \times \leq 6n$

$[0 \ldots \log_2]$

$\leq 6n \log_2 n + 6n = O(n \log n)$

(more on this soon)

Notes:

1) This is a worst-case analysis. It makes no assumptions about the input

2) We focus on the big picture: Don't worry about constant factors

   a) its more tractable mathematically

   b) constants depend on hardware & software "environment"

   c) We actually lose little predictive power.

(15) Asymtotic* notation

Goal: characterize the running time of an algorithm as a function of n, the input size, while:

a) suppressing constant factors (too system dependent)

b) and lower-order terms (irrelevant for large inputs)

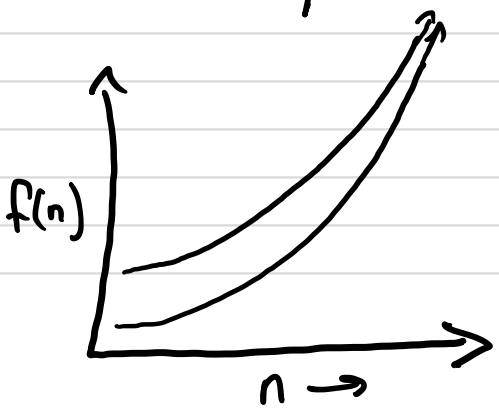
So $6n \log_2 n + 6n \Rightarrow O(n \log n)$

- $6n$: ignore constant
- $\log_2 n$: constant
- $6n$: ignore lower-order term

since: $\log_b(M) = \frac{\log_a(M)}{\log_a(b)}$

constant $\rightarrow \log_a(b)$

* Asymtotic: Approaching a value or a curve arbitrarily closely as some sort of limit is taken, e.g for large n.



Big-O Defined

Formally

Let f be a function of the natural #'s

$f = O(g(n))$ iff for some constants

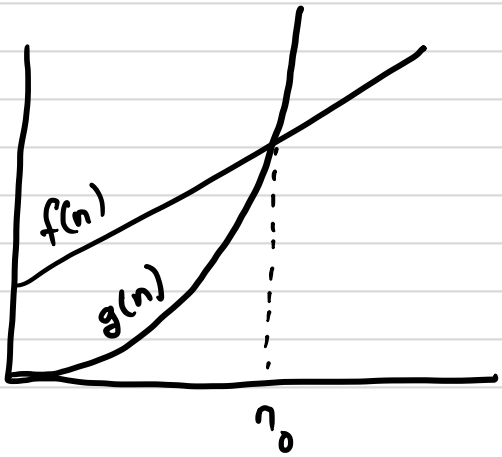
c and n_0

$$f(n) \leq c \cdot g(n) \text{ for all } n > n_0$$

In English:

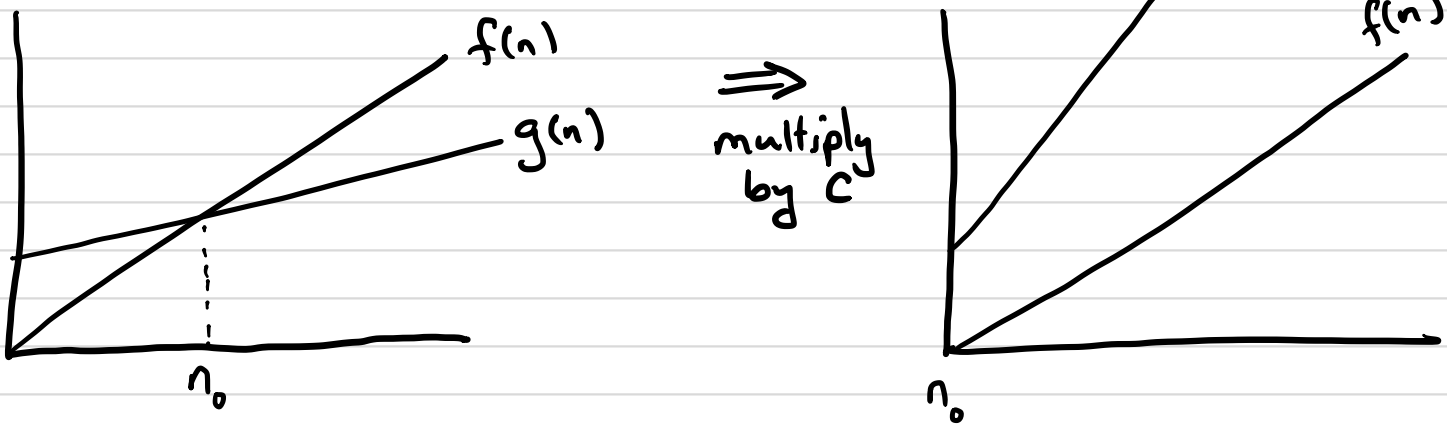
- $f(n)$ eventually grows no faster than $g(n)$ (ignoring constants)
- $f(n)$ is eventually bounded from above by a constant multiple of $g(n)$
- $O(g(n))$ describes an <u>upper bound</u>.

In Pictures



In this example, $g(n)$ eventually bounds $f(n)$ from above regardless of the slope of $f(n)$

More pictures



Two linear functions are big-O of each other.

$f(n) = O(g(n))$

$g(n) = O(f(n))$

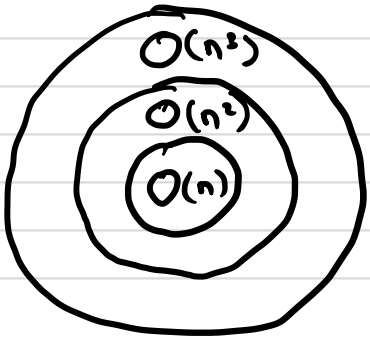
Technically

$5n^2 = O(n^2)$, but it is also true that

$5n^2 = O(n^3)$, but this is a looser bound and thus less informative.

By convention, we always specify the tightest upper bound that we know.

"=" expresses membership:



If $f(n) = O(n \log n)$ it is technically correct (but confusing) to say $f(n) = O(n^2)$.

(17) # Some Rules for determining O

a) **Summing terms.**

$f = O(h)$, $g = O(h) \Rightarrow f + g = O(h)$

e.g $f(n) = n^2 = O(n^2)$, $g(n) = n = O(n^2)$ $\Rightarrow f(n) + g(n) = n^2 + n = O(n^2)$

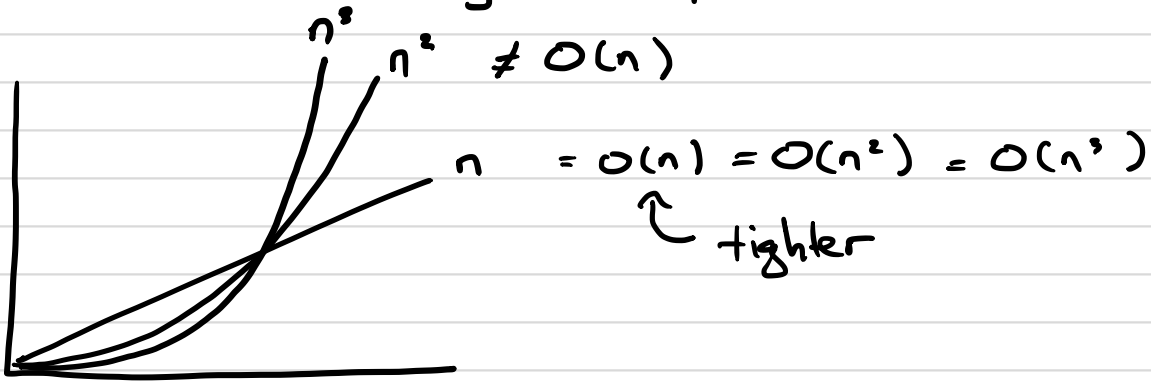
b) **A polynomial of degree d is $O(n^d)$**

$f(n) = 5n^2 - 3n + 6 = O(n^2)$

$\Downarrow \quad \Downarrow \quad \Downarrow$

$O(n^2) \quad O(n^2) \quad O(n^2)$

So we can always drop lower-order terms.



n^2 eventually surpasses $c \cdot n$ no matter how big we make c.

b) **Degree d polynomials are $O(n^d)$**

A more formal proof:

Suppose $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$

where $k \geq 0$ is a non-negative integer
a_i's are real #s (pos or neg)

Then:

$$f(n) \leq |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| n + |a_0|$$

$$\leq |a_k| n^k + |a_{k-1}| n^k + \cdots + |a_1| n^k + |a_0| n^k$$

(when $n \geq 1$)

$$\leq (|a_k| + |a_{k-1}| + \cdots + |a_1| + |a_0|) \cdot n^k$$

(when $n \geq 1$)

$$\leq c \cdot n^k \quad (n \geq n_0 = 1)$$

where $c = (|a_k| + |a_{k-1}| + \cdots + |a_1| + |a_0|)$

c) Degree-k polynomials are not $O(n^{k-1})$

e.g $n^3 \neq O(n^2)$

Proof (By contradiction):

Suppose $n^k = O(n^{k-1})$

Then there are constants c, n_0 such that

$$n^k \leq c\, n^{k-1} \quad \text{for all } n \geq n_0$$

$$\therefore \quad n \leq c \quad \text{(since } n \text{ is positive)}$$

which asserts that c is bigger than every positive integer.

But c is a constant and $\therefore$ cant be bigger than every positive integer.

For example, $n = \lceil c+1 \rceil > c$

This is our contradiction, so our supposition is false and our proposition that degree-d polynomials are not $O(n^{d-1})$ must be true.

■

(17) Rules - continued

d) Every logarithm grows slower than every polynomial.

$\log_b n = O(n^x)$ for all $b > 1$ and $x > 0$

e.g $\log_{10} n = O(n^{0.00001})$

e) Every polynomial grows slower than every exponential.

$n^k = O(r^n)$ for $r > 1$

e.g $n^{1000} = O(1.00001^n)$

(18)

Tractable: Algorithms are considered tractable if their run-time complexity is

$O(n^k)$ for some k.

- includes constant & logarithmic algorithms which are $O(n^k)$ too
- Exponential algorithms, e.g $\approx 2^n$, 3^n etc are "intractable".
- Factorial algorithms $O(n!)$ essentially require that we evaluate every possible permutation of input values, TSP (Travelling Salesman Problem) being a famous example.
- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^k) < O(k^n) < O(n!)$

tractable | intractable

(19)

Practice Problems

$f(n)$	$g(n)$	$f(n) = O(g(n))$?	$g(n) = O(f(n))$?
$2n$	$3n$	yes	yes
n	2^n	yes	no
n^2	n^2+n	yes	yes
$n^{0.0001}$	$\log(n)$	no	yes
n^{100}	1.1^n	yes	no
$n \log n$	n	no	yes
$\sin(n)$	$\cos(n)$	no	no

(20) Lower Bounds : Ω

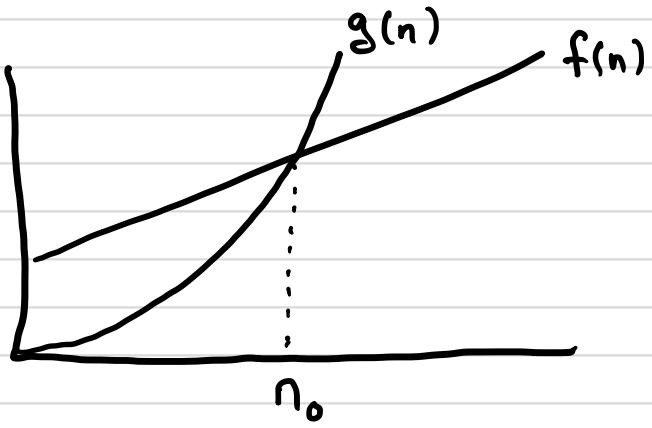
big-O : (upper bound) : The algorithm takes no more than this much time.

big-Ω : (lower bound) : The algorithm takes at least this much time.

a) mathematical definition

$f(n) = \Omega(g(n))$ iff there are positive constants c and n_0 such that $f(n) \geq c\, g(n)$ for all $n \geq n_0$

b) $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$



(21) Tight Bounds : Θ

a) If $f(n) = O(g(n))$ and
$f(n) = \Omega(g(n))$ then,
$f(n) = \Theta(g(n))$.

b) Saying $f(n) = \Theta(n^2)$ means that it really isn't growing any faster.

c) Formally:

$f(n) = \Theta(g(n))$ iff

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

for constants c_1, c_2, $n \geq n_0$

d) Algorithm designers focus on upper bounds, so will often say O when Θ holds.

MergeSort : $\Theta(n \log n)$
Binary Search : $\Theta(\log n)$

(22) Improving Running time

1. Use pre-processing to avoid repeated work:

$O(n^2)$:

```
for i = 1 to n
    sum = 0
    for j = 1 to n          // compute sum
        sum += A[j]
    print sum * i
```

$O(n)$:

```
sum = 0
for j = 1 to n
    sum += A[j]

for i = 1 to n
    print sum * i
```

2. Modify the algorithm somehow: This is where algorithms research focusses much effort.

3. Delete a line of code
4. switch Processor or language
5. Skip some inputs

} Constant factor improvements (small impact usually.)