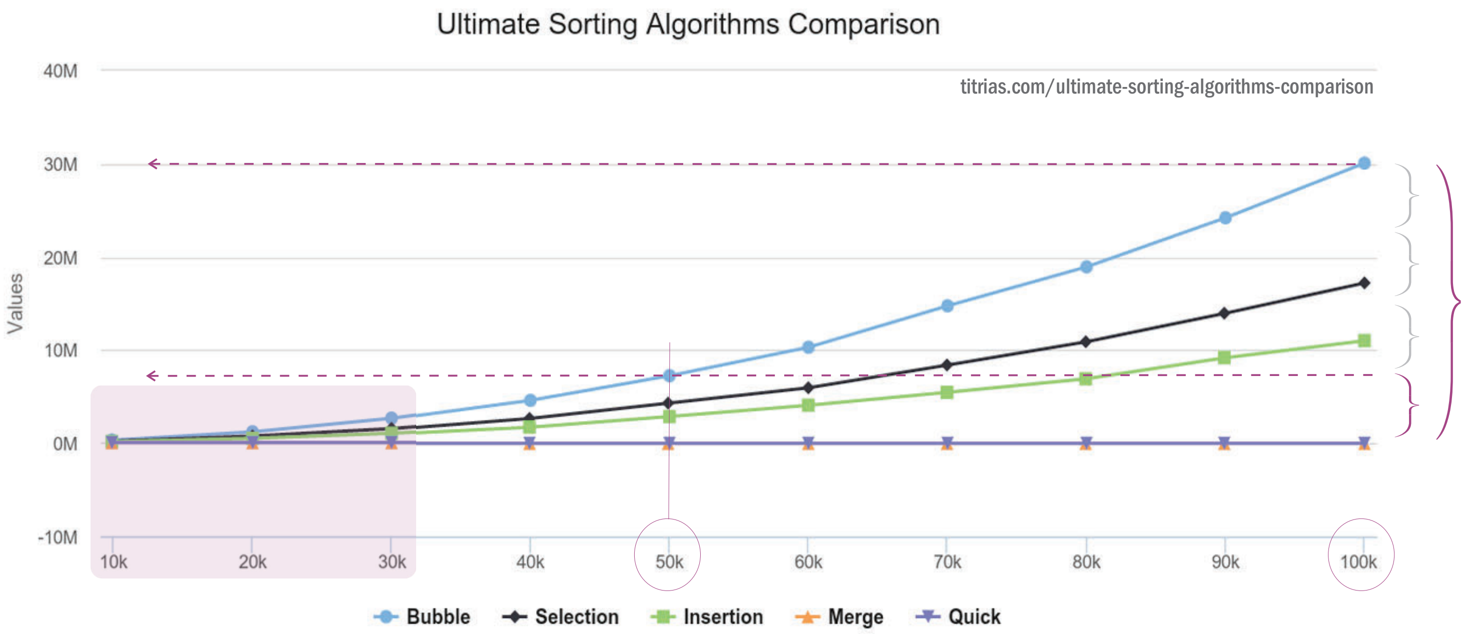
Run Time

➪ [Runtime] Growth

Ultimate Sorting Algorithms Comparison

by Fady S. Ghatas | Aug 5, 2015 | Algorithm Analysis | 0 comments



Review

Therefore, base efficiency on the growth of runtime rather than the runtime itself.

Algorithms run longer on larger inputs (run time growth varies widely)

Efficiency doesn't really matter for small inputs.

- ✎ Focus time efficiency for large inputs
- ✎ Need to isolate algorithm performance from that of machine and code

Most of the runtime is spent in the most frequently executed operation(s).

- ✎ Time efficiency may be measured by <u>growth</u> of basic operation count, $C(n)$, as input size n increases

Non-recursive Algorithms
Analysis Plan

➪ Order of growth

What property of the input that, if increased, causes run time to increase?

❶ Select suitable input size parameter, *n*

Examine inner-most loops for <u>most frequent</u> operations.

❷ Identify a suitable basic operation

Will the count (run time) vary for inputs (**instances**) of the same size? Check loops for early exits sometimes.

❸ Check dependancy of basic op

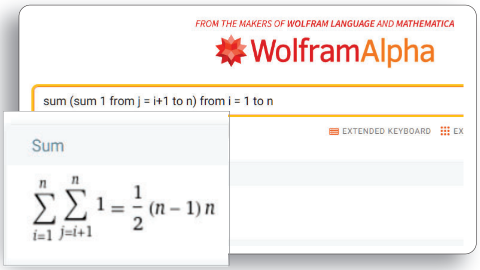
Summation and counting formulas are useful.

❹ Calculate count $C(n)$: write a sum

In practice, the dominant term in C(n) (one that grows faster as n tends to ∞) is enough.

❺ Determine *order of growth* of $C(n)$

Review Summations
A Useful Tool



➪ Basic properties

$$\sum_{i=l}^{u} c a_i = c \sum_{i=l}^{u} a_i \qquad \sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$$

Quiz
Which sum would you choose to solve Example 1 from previous slide?

➪ Basic sums (references)

$$\sum_{i=l}^{u} 1 = u - l + 1 \text{ where } l \leq u$$

Upper term times following one, divided by 2.

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Non-recursive Algorithms Example 1

- [Problem] Instance
- Pseudocode

One of the easiest non-trivial algorithms.

For each key, do a selection round to pick the smallest.

Selection Sort, efficiency?

Use a small **instance** to figure out/design steps before writing a **pseudo-code** *(mix of natural language, math, and programming-like expressions)*.

89 45 68 90 29 34 17

Selections are marked bold (for example, in the first round: 45, 29, 17); the last one to select is picked out.

The last may be swapped with the first of a selection round when sorting inside the input array (in-place).

Limits

89	**45**	68	90	**29**	34	**17**
17	45	68	90	**29**	34	89
17	29	68	90	**45**	**34**	89
17	29	34	90	**45**	68	89
17	29	34	45	90	**68**	89
17	29	34	45	68	90	**89**
17	29	34	45	68	89	90

Algorithm Selection Sort

Input Array of n keys $A[0..n-1]$

Output Sorted array

```
for i ← 0 to n − 2 do
    sel ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[sel] then
            sel ← j
    swap A[i], A[sel]
```

Non-recursive Algorithms
Example 1

➪ Basic operation

Examining a pseudo-code description of the algorithm could greatly facilitate analysis.

➪ Efficiency?

Algorithm Selection Sort

Input Array of n keys $A[0..n-1]$ ❶

Output Sorted array

```
1: for i ← 0 to n − 2 do
2:     sel ← i
3:  ❷  for j ← i + 1 to n − 1 do
4:         if A[j] < A[sel] then
5:             sel ← j
6:     swap A[i], A[sel]
```

❶ Select suitable input size parameter, *n*

❷ Identify a suitable basic operation

❸ Check dependancy of basic op

❹ Calculate count $C(n)$: write a sum

❺ Determine *order of growth* of $C(n)$

Quiz
What would be a <u>suitable</u> **basic operation**? Can we pick + or – in line 3? What about loop exit check or index increment (j++)?

Analysis of Algorithms
Effciency

➪ Observation

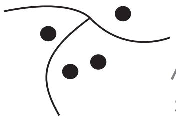
Time efficiency of most algorithms falls into a few categories of (runtime) growth

➪ A classification?

A system for classifying efficiency should avoid dealing individually with efficiency of potentially 1000s of algorithms

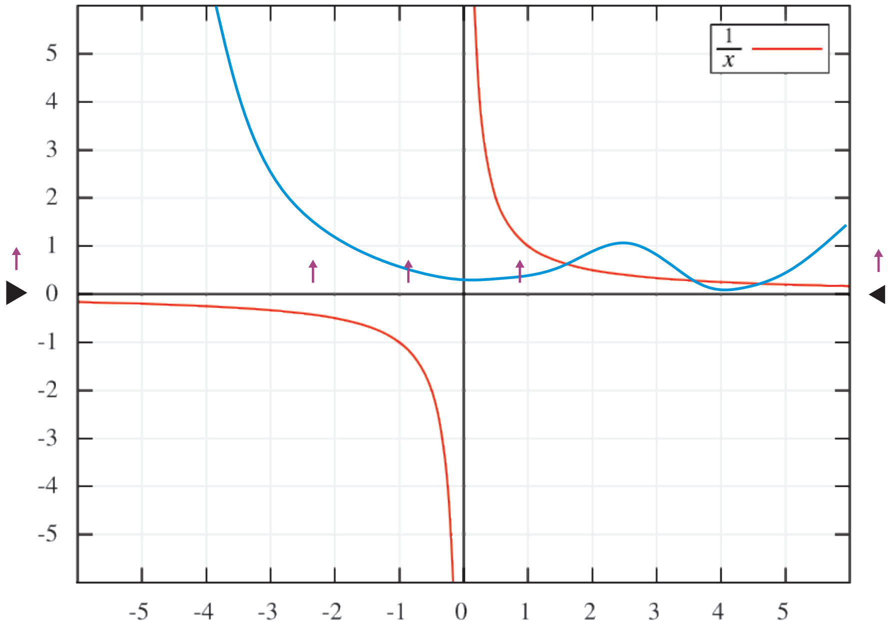
A Useful Tool from Math
Asymptotic Classification

➪ Asymptotes



A **boundary** that other curve(s) may come close to but never cross.

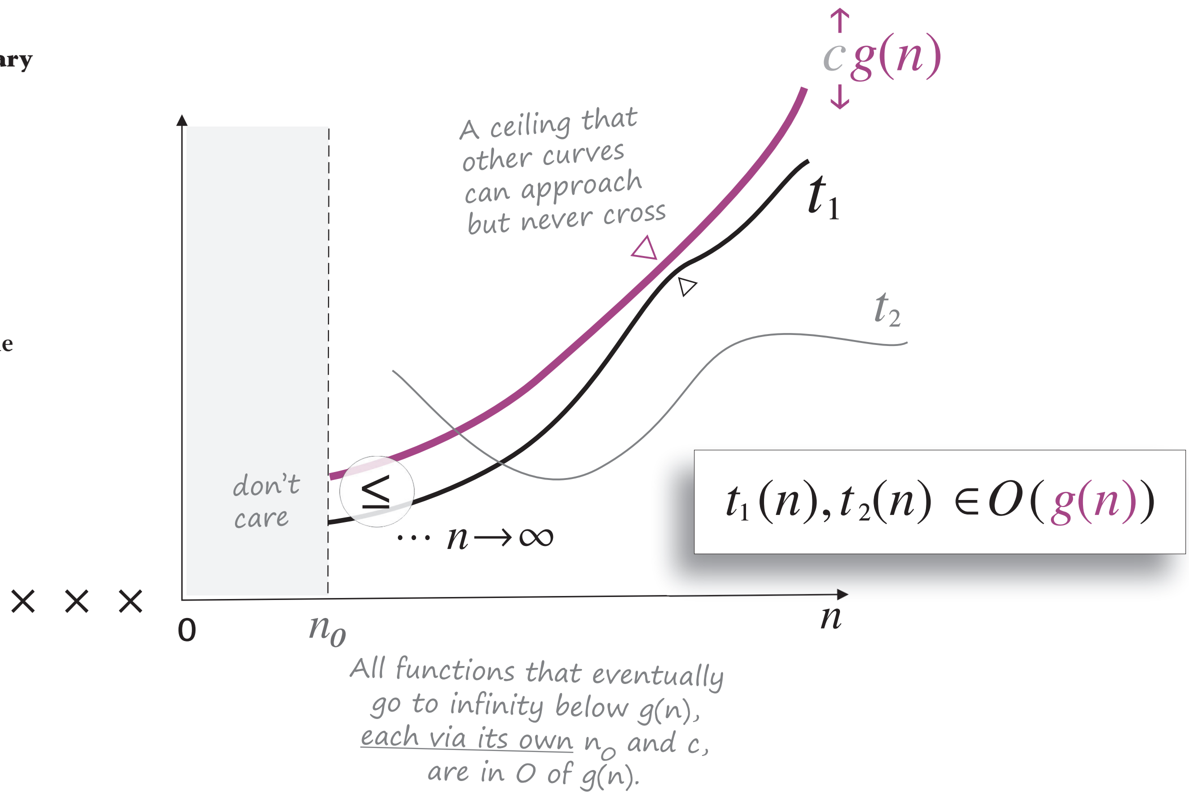
Any function can be used to set a boundary, in this case $y = f(x) = ?$



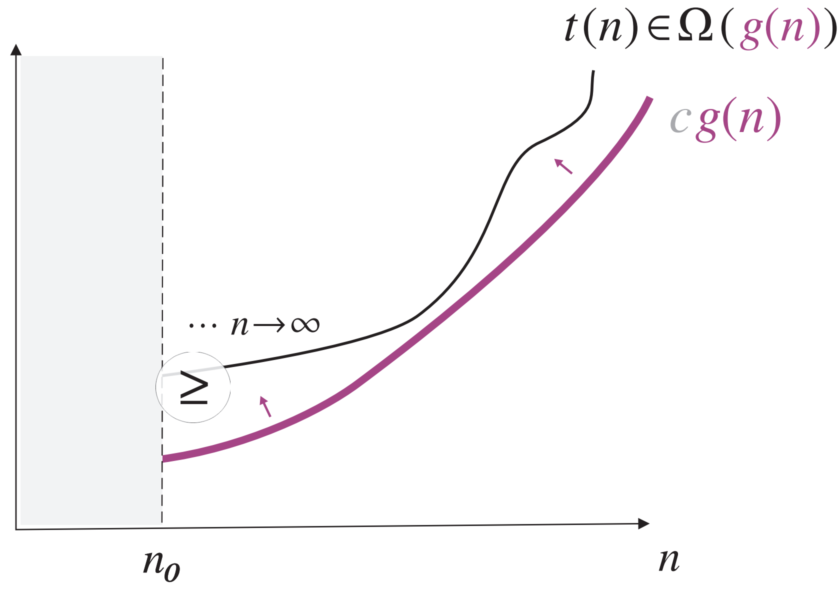
Asymptotic Classification
Setting Upper Boundary

Setup an upper **boundary** on order of growth.

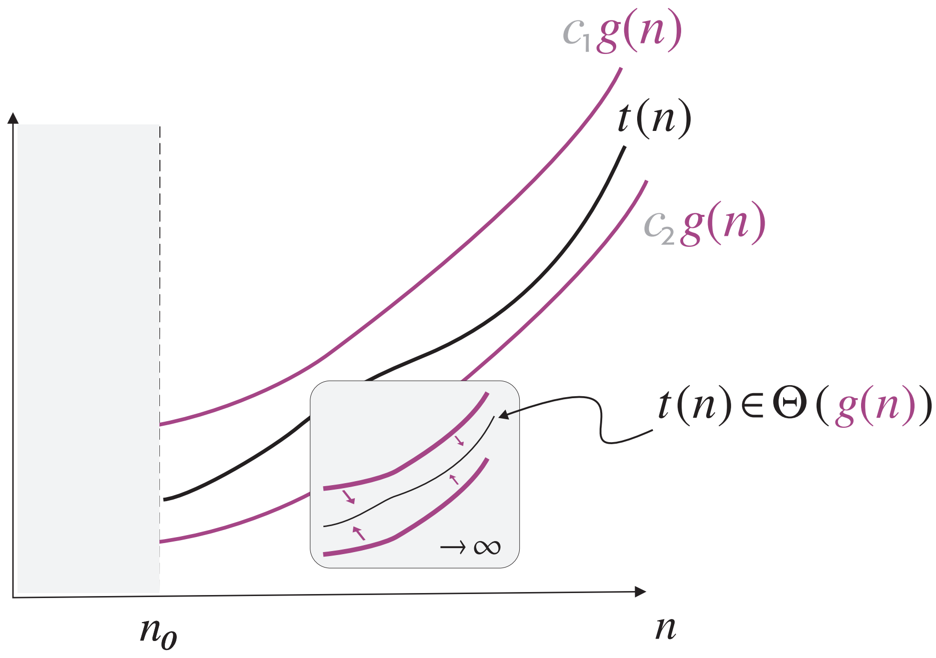
t_2 stays under after some $n > n_0$ of t_1.



Setting Lower Boundary

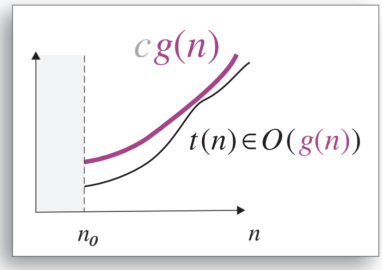


Asymptotic Classification
Θ – Similar Growth



Exercise
Write a formal (math) definition of O, Ω, Θ.

Asymptotic Classification Exercise



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

$c\,g(n)$

$$100n + 5 \in O(n^2) \qquad \frac{1}{2}n(n-1) \in O(n^2)$$

$$n^3 \notin O(n^2) \qquad 0.00001n^3 \notin O(n^2) \qquad n^4 + n + 1 \notin O(n^2)$$

$$n^3 \in \Omega(n^2) \qquad \frac{1}{2}n(n-1) \in \Omega(n^2) \qquad 100n + 5 \notin \Omega(n^2)$$

Algorithm Efficiency
Basic Classes

➪ Asymptotic efficiency

Using long-term (limiting) runtime behavior to classify efficiency as inputs become increasingly larger.

TABLE 2.2 Basic asymptotic efficiency classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 5.5). Note that a logarithmic algorithm cannot take into account all its input (or even a fixed fraction of it): any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	"*n-log-n*"	Many divide-and-conquer algorithms (see Chapter 4), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficie[n] two embedded loops (see the tary sorting algorithms and c[e] n-by-n matrices are standard e
n^3	*cubic*	Typically, characterizes efficie[n] three embedded loops (see the nontrivial algorithms from line class.

2^n	*exponential*	Typical for algorithn n-element set. Ofter in a broader sense tc growth as well.
$n!$	*factorial*	Typical for algorithn of an n-element set.

Non-recursive Algorithms
Example 2

➪ Worst-case efficiency

Recognize structure
- ✏ Conditionals reduce op frequency
- ✏ Loops (non-recursive) amplify frequency
- ✏ Look for early exit clues

How to read?

Algorithm $UniqueElements$
Input Array $A[0..n-1]$
Output Return true if elements in A distinct, otherwise false

```
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] then
            return false
return true
```

Quiz
Does the basic operation count depend on input size only? Hint: Try different instances of the same size.

❶ Select input size parameter, *n*
❷ Identify a suitable basic operation
❸ Check dependancy of basic op
❹ Calculate count *C*(*n*): write a sum
❺ Determine *order of growth* of *C*(*n*)

Asymptotic Classification
A Useful Property

✓

8
2
−1
3
12
5

✗

8	−1
2	2
−1	5 ◀
5	5
12	8
5	12

- **Revisit: decide if keys distict**
- **A solution: overall efficiency?**
  Sort, check consecutive pairs
- **Compare Example 2**
- **Basis (theorems-proofs)**

Of course, it is based on the time formula $T=T_1+T_2$, however, no sense in going to basic principles everytime.

Non-recursive Algorithms Exercise

Recognize structure

- ✏ Procedures nested vs. sequenced
- ✏ Conditionals reduce op frequency
- ✏ Loops (non-recursive) amplify frequency
- ✏ Look for early exit clues

How to read?

Algorithm *Factorial*
Input Integer $n \geq 0$
Output $n!$

$fact \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
    $fact \leftarrow fact \times i$
return $fact$

Algorithm *InsertionSort*
Input Array $A[0..n-1]$ of orderbale keys
Output Sorted array

for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

- ➪ **Write C(n)**
- ➪ **Determine efficiency**