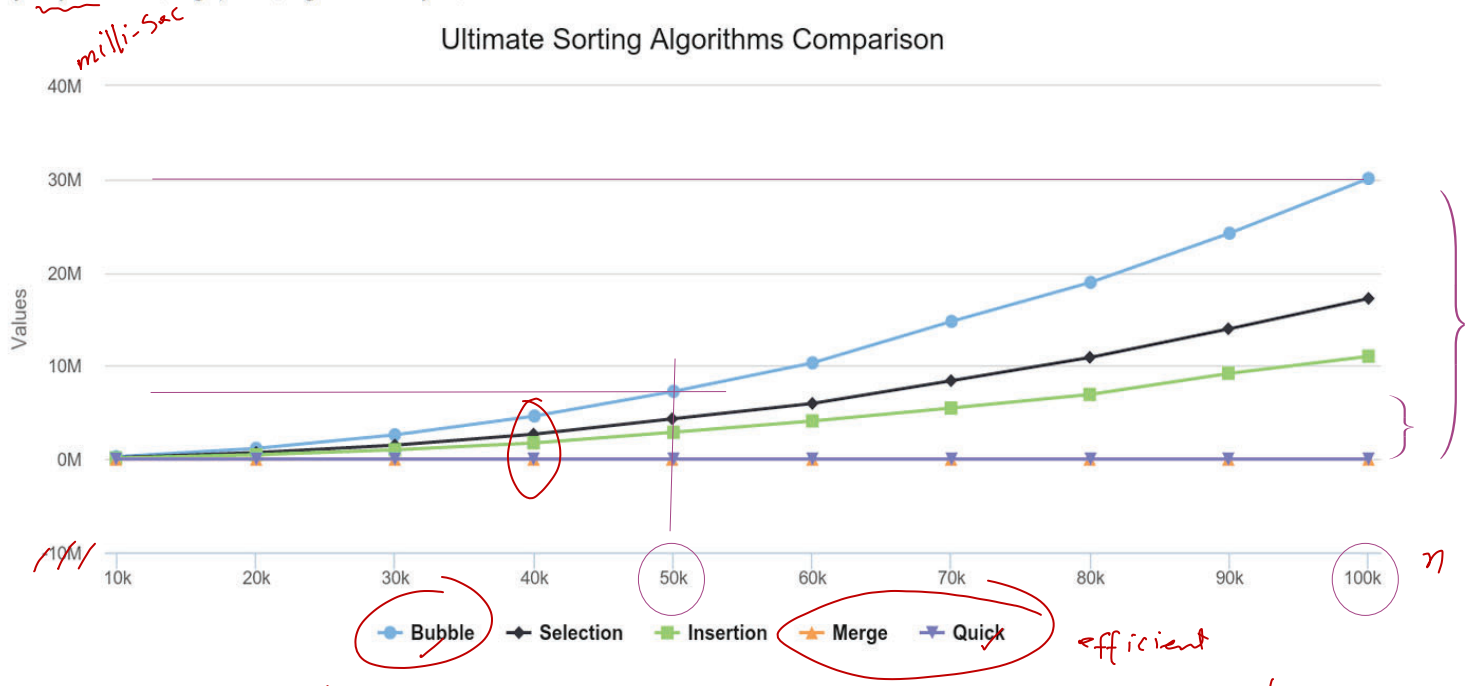
Run Time

➪ [Runtime] Growth

Ultimate Sorting Algorithms Comparison

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



Review

- Algorithms run longer on larger inputs
- Focus on time efficiency
- Need to isolate algorithm performance from that of machine and code
- Time eff. may be measured by growth of basic operation count, C(n), as input size n increases

Efficiency doesn't really matter for small inputs.

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

Non-recursive Algorithms
Analysis Plan

❶ Select suitable input size parameter, n

❷ Identify suitable basic operation

❸ Check dependancy of basic op

❹ Determine count $C(n)$: write a sum

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

Non-recursive Algorithms
Selection Sort

➪ **Key comparison**

Algorithm *SelectionSort*
Input ··· $A[0..n-1]$ ···
Output ···

1: **for** $i \leftarrow 0$ **to** $n-2$ **do**
2: $min \leftarrow i$
3: **for** $j \leftarrow i+1$ **to** $n-1$ **do**
4: **if** $A[j] < A[min]$ **then**
5: $min \leftarrow j$
6: swap $A[i]$, $A[min]$

➪ **Operation**

➪ **Efficiency?**

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

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

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?

❶ Select input size parameter *n*
❷ Identify basic operation
❸ Check basic op count dependancy
❹ Setup a sum or a recurrence for $C(n)$
❺ Determine order of growth of $C(n)$ (may need to solve sum or recurrence)

Review Summations
A Useful Tool

➪ Basic properties

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

➪ Basic sums (references)

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

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

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

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

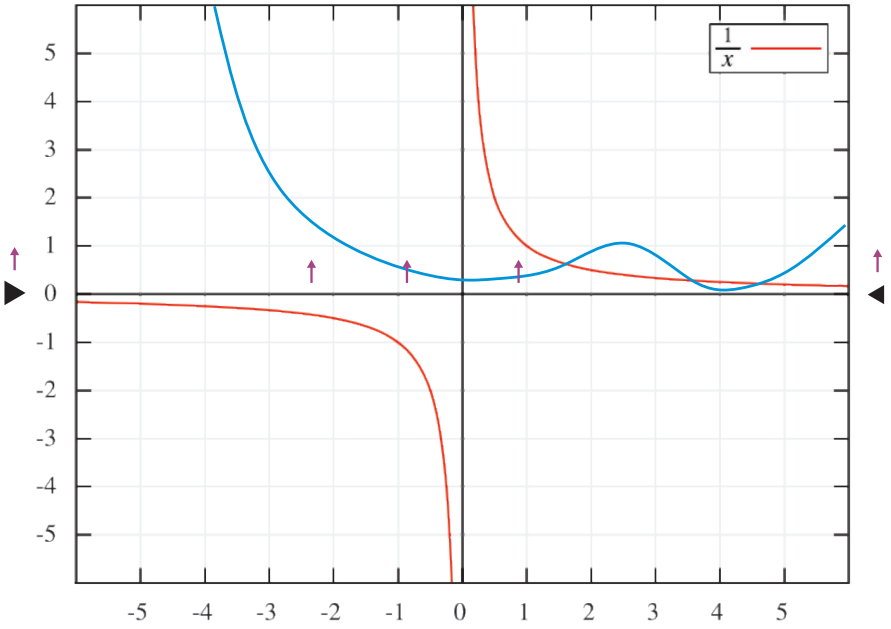
A Useful Tool from Math

Asymptotic Classification

➪ Asymptotes

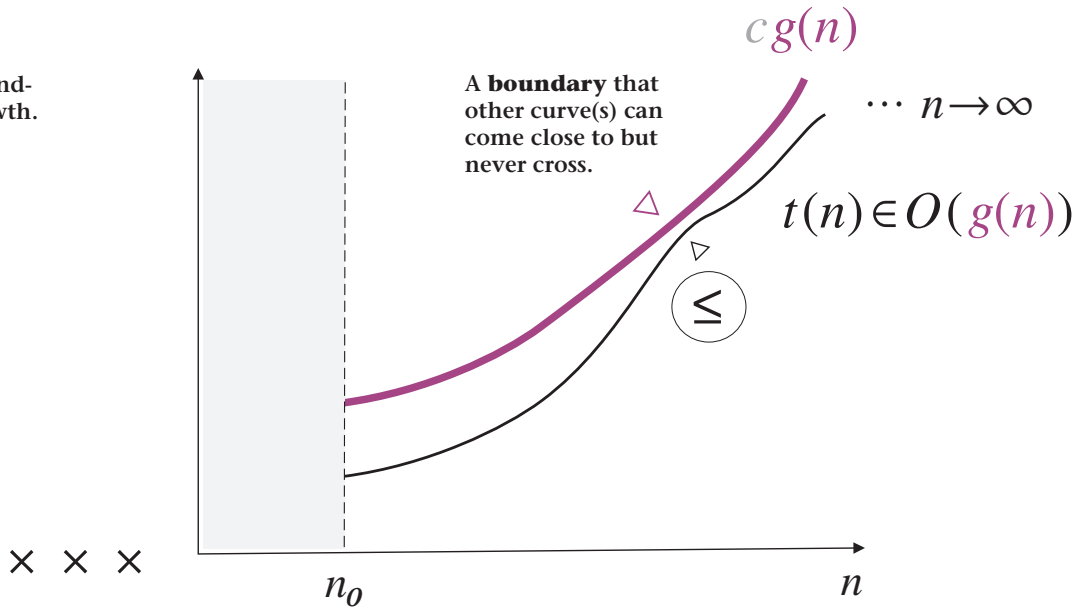
A **boundary** that other curve(s) may come close to but never cross.

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

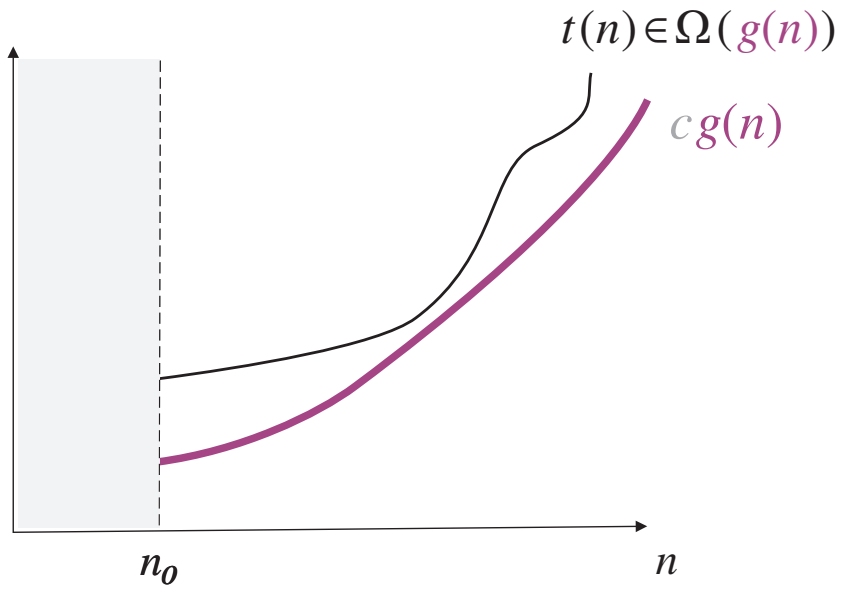


Asymptotic Classification
Setting Upper Boundary

Setup an upper boundary on order of growth.



Asymptotic Classification
Setting Lower Boundary



Asymptotic Classification
Θ – Similar Growth

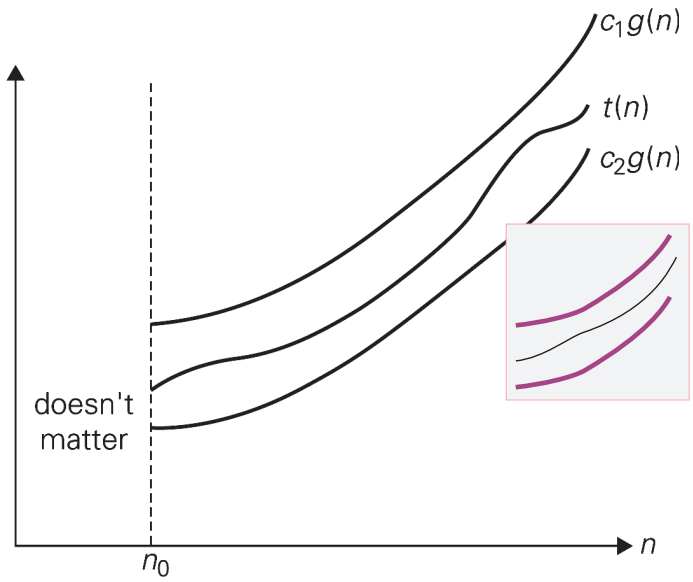
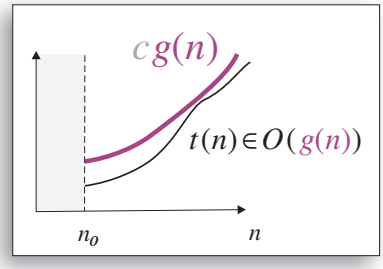


FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Asymptotic Classification Exercise



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

$100n + 5 \in O(n^2)$

$\frac{1}{2}n(n-1) \in O(n^2)$

$n^3 \notin O(n^2)$

$0.00001n^3 \notin O(n^2)$

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

$n^3 \in \Omega(n^2)$

$\frac{1}{2}n(n-1) \in \Omega(n^2)$

$100n + 5 \notin \Omega(n^2)$

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 efficien two embedded loops (see the tary sorting algorithms and c n-by-n matrices are standard e
n^3	*cubic*	Typically, characterizes efficien three embedded loops (see th nontrivial algorithms from lin class.
2^n	*exponential*	Typical for algorithn n-element set. Ofter in a broader sense t growth as well.
$n!$	*factorial*	Typical for algorithn of an n-element set.

Example 2

Exercise
Compare to solution discussed in previous lecture. What if a *quicksort* was used?

How to read?

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

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

Quiz
Does the basic operation count depend on input size only?