cs611fig1.cdr
Monday, January 29, 2024 12:42:34 PM
Color profile: Disabled

Composite Default screen

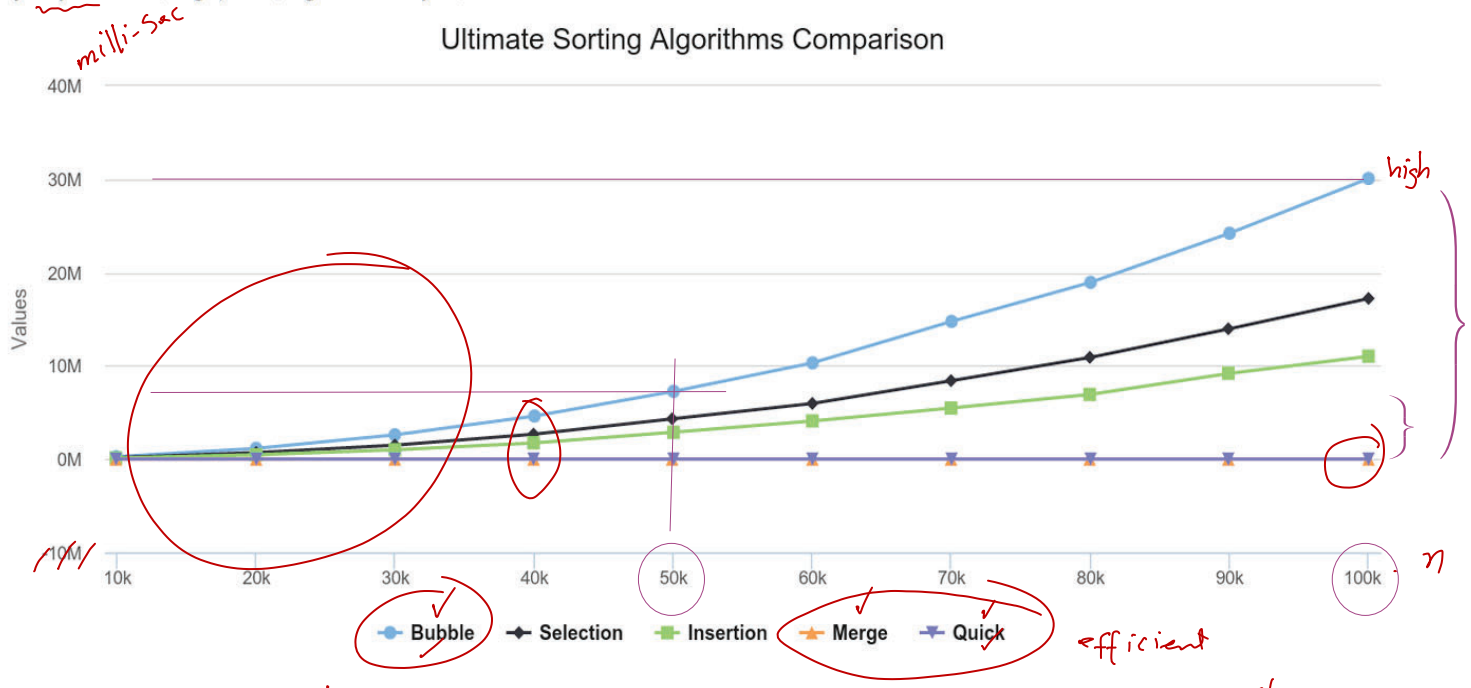
# Run Time

➪ [Runtime] Growth

## Ultimate Sorting Algorithms Comparison

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

milli-sec




high

not efficient

efficient

runtime growth as list size n increases is slow

# Review

- Algorithms run longer on larger inputs
- Focus on time efficiency

Efficiency doesn't really matter for small inputs.

- 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

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





Def. Basic op ≜ most frequent.

time ≜ Top

runtime ← T = Top × Count + Other — (1)

basic op time

# of times

from a machine: T ~ Top × Count — (2)

same ~

from alg.

time (seconds), is constant on the same machine

on the same machine Top const. — (3)

T ~ Count ⇒ same growth

ايش طلع.

of basic ops

If we can determine growth of count

⇒ same growth of run time i.e., the efficiency

# Non-recursive Algorithms Analysis Plan

Ex Sorting → list size

1. Select suitable input size parameter, $n$
2. Identify suitable basic operation(s) most frequent
3. Check dependancy of basic op
4. Determine count $C(n)$: write a sum ? How
5. Determine *order of growth* of $C(n)$

$C(n) \triangleq$ number of times (count) of basic ops execution as function of input size $n$

# Non-recursive Algorithms
# Selection Sort

∴ Conclusion Sel Sort has quadratic (like $n^2$) eff.

**Algorithm** *SelectionSort*
**Input** ··· $A[0..n-1]$ ··· (list size n)
Output ···

1: **for** $i \leftarrow 0$ **to** $n-2$ **do** (once)
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 $A[i]$, $A[min]$

j++ same freq.

➪ **Key comparison** (specific input)

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

**Limits** $n=7$ { 0, 7−1=6

| | | | | | | |
|---|---|---|---|---|---|---|
| 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 |

➪ **Operation**
➪ **Efficiency?**

**Quiz**
What would be a suitable **basic operation**? Can we pick + or − in line 3? What about loop exit check or index increment?

1. Select input size parameter *n*
2. Identify basic operation
3. Check basic op count dependancy
4. Setup a sum or a recurrence for $C(n)$
5. Determine order of growth of $C(n)$ (may need to solve sum or recurrence)



We identify 3 basic ops (equally freq.)

loops always run to end ⇒ basic ops count only depends on n
no dependence on instance −

Step 4 (loop limits: $n-2$, $n-1$)

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 3 = 3 \text{ basic ops (step 2)}$$

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

Step 5 order of growth is like $n^2$

Step 5 Facts
1. $n^2$ order of growth is the same for runtime.
2. $n^2$ is the same wether we count one or all basic ops.

$n^2$ - growth pattern investigate:

$n_1 = n$ $\quad n_2 = 2n$

$C(n) = n^2$ $\quad \therefore \frac{C_2(n)}{C_1(n)} = \frac{(2n)^2}{n^2} = \frac{4n^2}{n^2} = 4$ times

doubling input size from n to 2n lead to C(n) increasing 4 times

# 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

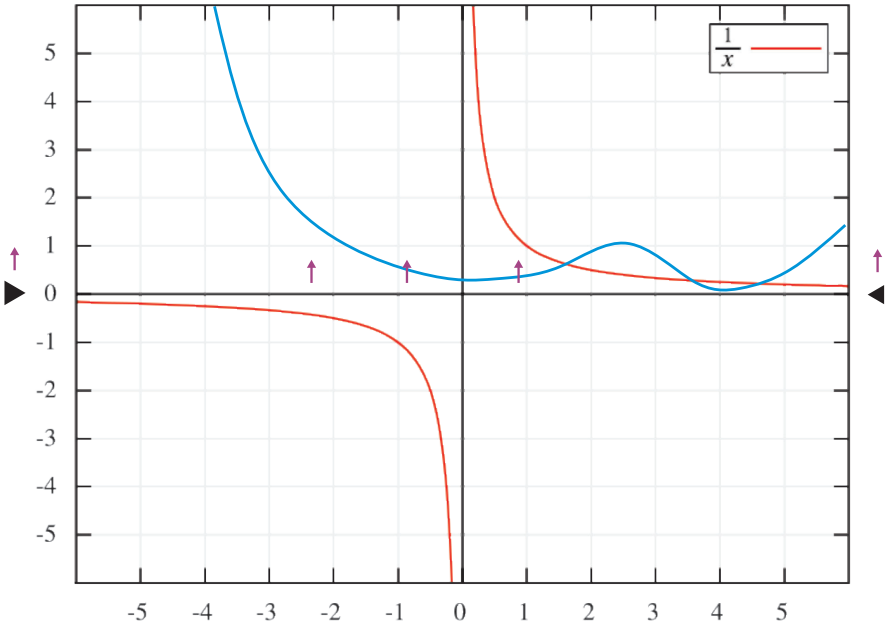
# A Useful Tool from Math

# Asymptotic Classification

## ➪ Asymptotes

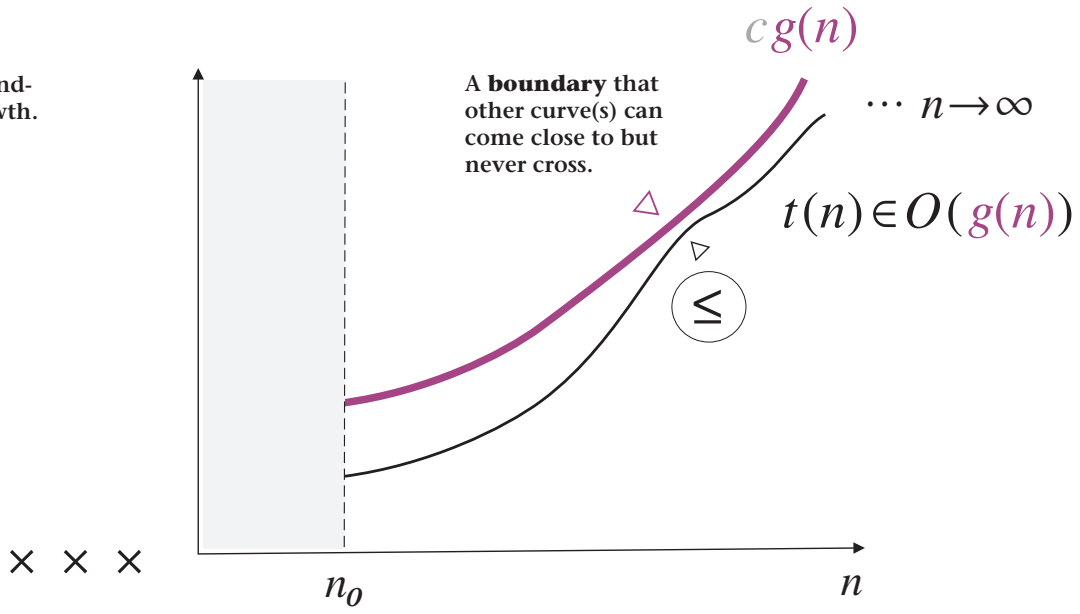
A **boundary** that other curve(s) may come close to but never cross.

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

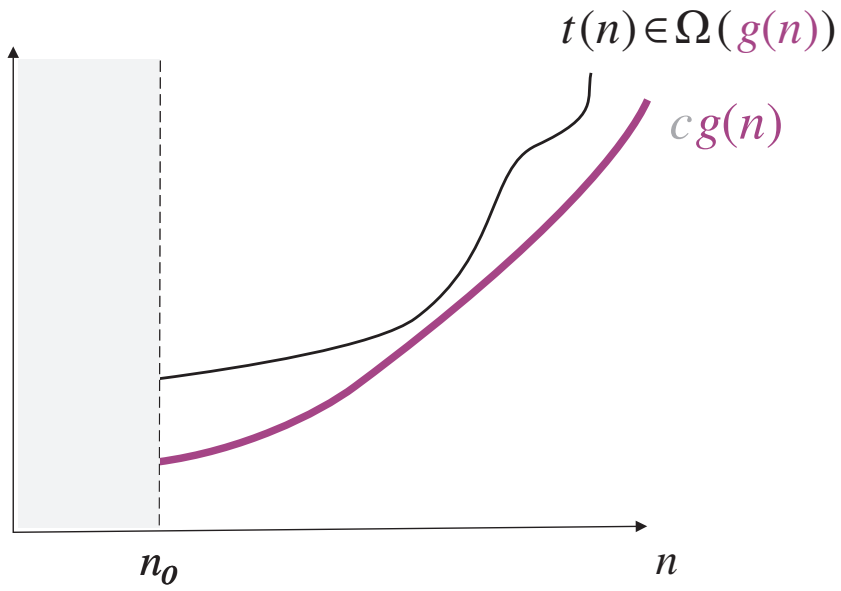


# Asymptotic Classification
# Setting Upper Boundary

Setup an upper boundary on order of growth.



# Asymptotic Classification
# Setting Lower Boundary



# Asymptotic Classification
# Θ – Similar Growth

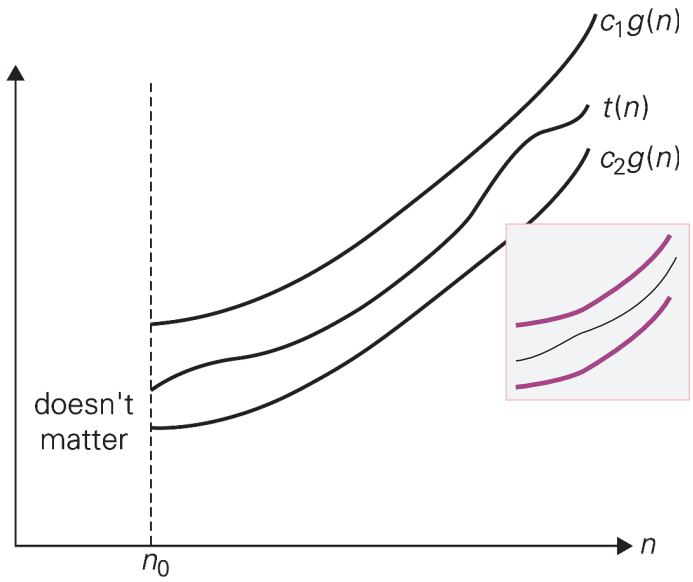
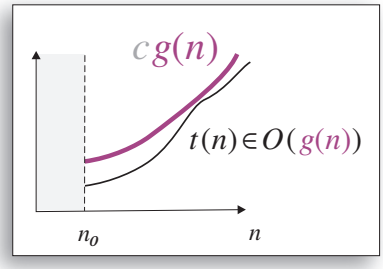



**FIGURE 2.3** Big-theta notation: $t(n) \in \Theta(g(n))$

# 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?**