Algorithm Analysis

- Efficiency of algorithms
  - growth rate
- Big-O

Determining efficiency

- How do we decide whether one program is more efficient (faster) than another?

- Time them?
- What if different processors?
- What if written in different languages?
- What if one uses a while loop and one uses recursion?

Determining efficiency

How do we determine the INTRINSIC efficiency of an ALGORITHM, independent from the processor, programming language, coding style, etc.?

Solution:
By determining how the running time of an algorithm depends on the size of the problem it's solving.

Algorithm Efficiency

- Analysis is difficult because
  - different way of coding the same algorithm
  - different computer architecture
  - different original status of the data

- It is not an exact science, just an approximation!

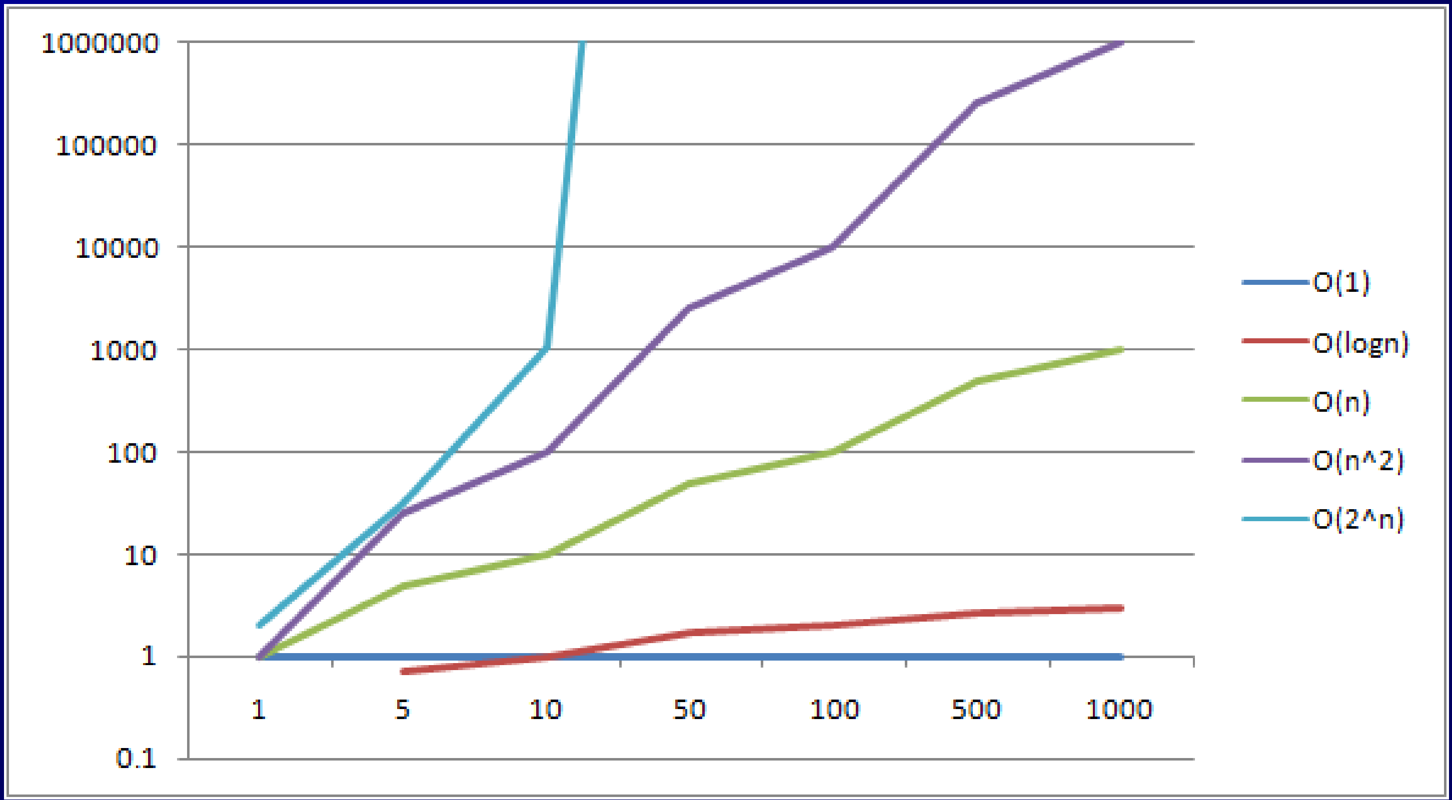
Efficiency

- Efficiency depends on
  - sorting algorithm
  - initial order of the data
  - number of elements
  - swaps are more costly than comparisons

Order of Magnitude

- How fast the algorithm grows as a function of the size of the data.
  - Growth rate functions (Big O)
  - constant: not proportional to n
  - logarithmic: proportional to $\log_2 n$
  - linear: proportional to n
  - quadratic: proportional to n^2
  - cubic: proportional to n^3
  - exponential: proportional to 2^n
- The “O” comes from the phrase “*on the order of*”

Growth rate functions



It does matter …

size	O(n2)	O(n $\log_2$ n)
10	93 millisecs	356 millisecs
100	8.46 secs	3.62 secs
1,000	13.91 min	36.91 secs
10,000	23.15 hrs	5.93 min
100,000	96.45 days	1 hr
1,000,000	26.41 years	10.23 hrs

Scalability and scenarios

- Any algorithm is efficient for a small set of data. Thus, when analyzing algorithms we need to think BIG.

- We need to consider three possible scenarios:
  - Best case – when the algorithm performs the fastest
  - Worst – when the algorithm performs the slowest
  - Average (or expected)

What does Big-Oh tell you?

- It does not tell you the numerical running time of an algorithm for a particular input or for small n.
- It tells you something about the rate of growth as the size of the input increases.
- At some point O(n) algorithm will be faster than an $O(n^2)$ algorithm, always. As the input size grows, the O(n) algorithm will get increasingly faster than an $O(n^2)$ algorithm. But it will not tell you for what values of n the O(n) algorithm is faster than the $O(n^2)$ algorithm.

The math of BigO()

$2n + 25 \quad => \quad O(n)$

$5n^2 - 2n + 5 \quad => \quad O(n^2)$

$\log_2(3n) \quad => \quad O(\log_2 n)$

and …

$O(n) + O(n) = O(n)$

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

$O(n) * O(\log_2 n) = O(n \log_2 n)$

$O(n) + O(\log_2 n) = O(n)$

Example 1 - Initializing an array

- Algorithm A

```
for (k = 0; k < size; k++)
        list[k] = 0;
```

- Algorithm B

```
list[0] = 0;
list[1] = 0;
list[2] = 0;

list[size-1] = 0;
```

Example 2 - cumulative sum

- Algorithm A

```
for (k = 0; k < size; k++)
    sum = sum + k;
```

- Algorithm B

```
sum = ((n+1) * n) / 2;
```

Hidden factors

- One must pay attention when calling Java methods. We need to understand how the method works.

```
private static boolean contains(int[] a, int value, int index) {
    if (index == a.length)
            return false;
    else if (a[index] == value)
            return true;
    else
            return contains(a, value, index+ 1);
}
```

Time is proportional to the size of the array. We say O(n).

Always?

Hidden factors


```
public static boolean hasDuplicates(int[] a) {
    for (int i = 0; i < a.length; i++){
            if (contains(a, a[i], i + 1))
                return true;
    }
    return false;
}
```

How many times do we call *contains*? __________

Time for each call to *contains* is proportional to __________

Hidden factors

If we double the length of the array, then we will call *contains* _____ as many times,AND each call to *contains* will take _____ as long. Therefore, *hasDuplicates* will take ____ times as long.

The running time for *hasDuplicates* is proportional to the ______ of a.length. We say $O(n^2)$.

Fibonacci numbers revisited …

- Fib(n) = Fib(n-1) + Fib(n-2)

Fib(10)

Fib(9) Fib(8)

Time for Fib(10) is almost twice the time for Fib(8) or almost twice the time for Fib(9). Each of those is O(n). Thus, Fib(n) is O(n)

Let's analyze some algorithms …(1)

- Retrieving an element in a list
  - array-based -- **O(1)**
  - reference-based -- **O(n)**

KEY: is the time proportional to the length of the list?

NOTE: we are <u>not</u> using the ArrayList or the LinkedList Java classes …

Let's analyze some algorithms … (2)

- Inserting an element at the beginning of a list
  - array-based -- **O(n)**
  - reference-based -- **O(1)**

KEY: is the time proportional to the length of the list?

NOTE: we are not using the ArrayList or the LinkedList Java classes …

Let’s analyze some algorithms … (3)

- Deleting last element in a list
  - array-based -- **O(1)**
  - reference-based -- **O(n)**

KEY: is the time proportional to the length of the list?

NOTE: we are <u>not</u> using the ArrayList or the LinkedList Java classes …

Homework

- Homework #4 (recursion) due tonight
- Quiz #4 (recursion) tomorrow
- Exam #1 on <u>Thursday Oct. 8</u>

Readings

- Read, as much as possible, the pdf on algorithm analysis
- Watch the video *Algorithm Analysis*