Complexity Analysis

Complexity Analysis

- As true computer scientists, we need a way to compare the efficiency of algorithms
- Should we just use a stopwatch to time our programs?
  - No, different processors will cause the same program to run a different speeds.
- Instead, we will count the **number of operations** that must run.

Counting Operations

n = number of elements in array a

```
public static double avg(int[] a, int n)
{
       double sum=0;
       for(int j=0; j<n; j++)
              sum+=a[j];
       return (sum/n);
}
```

There is one operation in the for-loop

Loop runs **n** times
1 loop comparison
1 loop increment
1 operation
3n operations

Total Operations:

3n + 3

Counting Operations

- With respect to n, how many operations does this code have?

```
public static double avg(int[] a, int n)
{
       double sum=0;
       for (int i=0; i<n; i++)
         for(int j=0; j<n; j++)
           sum+=a[j];
       return (sum/n);
}
```

Grouping Complexities

- So the run-time of our average function depends on the size of the array. Here are some possibilities:
  - Array size 1: runtime = 3(1) + 3 = 6
  - Array size 50: runtime = 3(50) + 3 = 153
  - Array size 1000: runtime = 3(1000) + 3 = 3003
- Notice that increasing the size of the array by a constant multiple has approximately the same effect on the runtime.

Grouping Complexities

- In order to compare runtimes, it is convenient to have some grouping schema.
- Think of runtimes as a function of the number of inputs.
- How do mathematicians group functions?

Functions

- How might you "describe" these functions? What "group" do they belong to?
  - $f(x) = 7$
  - $f(x) = \log(x + 3)$
  - $f(x) = 3x + 5$
  - $f(x) = 4x^2 + 15x + 90$
  - $f(x) = 10x^3 - 30$
  - $f(x) = 2^{(3x + 3)}$

Functions

- How might you "describe" these functions? What "group" do they belong to?
  - $f(x) = 7$ Constant
  - $f(x) = \log(x + 3)$ Logarithmic
  - $f(x) = 3x + 5$ Linear
  - $f(x) = 4x^2 + 15x + 90$ Quadratic
  - $f(x) = 10x^3 - 30$ Cubic
  - $f(x) = 2^{(3x + 3)}$ Exponential

Big-O Notation

- Instead of using terms like linear, quadratic, and cubic, computer scientists use big-O notation to discuss runtimes.
- A function with linear runtime is said to be of Order n.
- The shorthand looks like this: O(n)

Functions

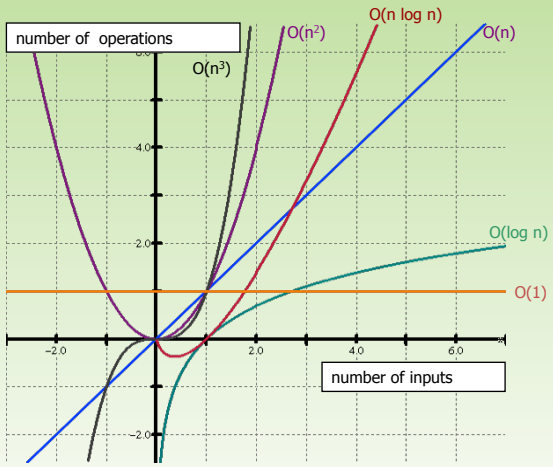
- If the following are runtimes expressed as functions of the number of inputs (n), we would label them as follows:
  - f(n) = 7
  - f(n) = log(n + 3)
  - f(n) = 3n + 5
  - f(n) = $4n^2 + 15n + 90$
  - f(n) = $10n^3 - 30$
  - f(n) = $2^{(3n + 3)}$

Functions

- If the following are runtimes expressed as functions of the number of inputs (x), we would label them as follows:
  - f(n) = 7 — O(1)
  - f(n) = log(n + 3) — O(log n)
  - f(n) = 3n + 5 — O(n)
  - f(n) = $4n^2 + 15x + 90$ — O(n^2)
  - f(n) = $10n^3 - 30$ — O(n^3)
  - f(n) = $2^{(3n + 3)}$ — O(2^n)

The common Big-O values

- O(1) : constant
- O(log n): Logarithmic
- O(n) : Linear
- O(n log n) : n log n
- O(n^2): Quadratic
- O(n^3): Cubic
- O(2^n): exponential

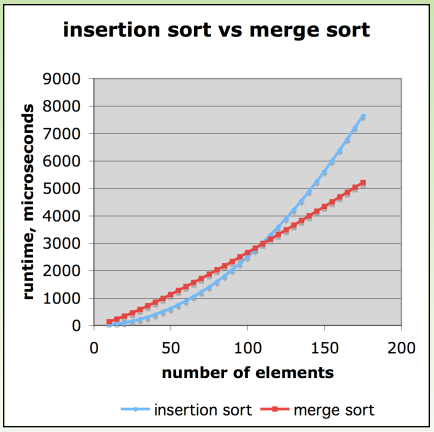


Algorithms are categorized by how their runtime increases in relation to the number of inputs.

Comparison of Two Algorithms

- Insertion Sort
  - Order: $n^2 / 4$
- Merge sort:
  - Order: 2n log n

Comparison of Two Algorithms

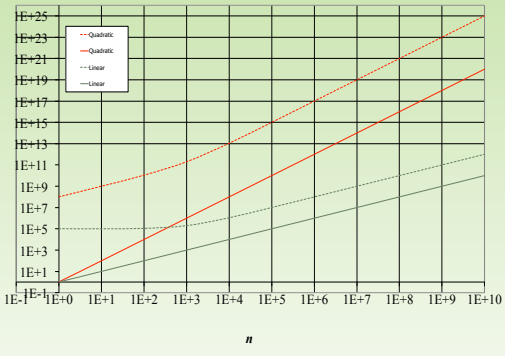


Slide by Matt Stallmann included with permission.

- Insertion Sort is $n^2 / 4$
- Merge sort is 2n log n
- Sorting a million items:
  - Insertion Sort takes roughly 70 hours
  - Merge Sort takes roughly 40 seconds
- This is a slow machine, but if it was 100 times faster then it would still be 40 minutes versus less than 0.5 of a second

Constant Factors Do Not Matter!

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function



Pseudocode

- Example: find max element of an array

```
Algorithm arrayMax(A, n)
  Input array A of n integers
  Output maximum element of A
  currentMax ← A[0]
  for i ← 1 to n - 1 do
     if A[i] > currentMax then
        currentMax ← A[i]
  return currentMax
```


Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition is not important
- Assumed to take a constant amount of time

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
Algorithm arrayMax(A, n)          # operations

 currentMax ← A[0]                2
  for i ← 1 to n - 1 do           2n
   if A[i] > currentMax then      2(n - 1)
      currentMax ← A[i]           2(n - 1)
  { increment counter i }         2(n - 1)
  return currentMax               1


                                  Total  8n - 2
```


General Guidelines

- The worst-case instructions determine worst-case behaviour, overall.
  - For example, a single n^2 statement means the whole algorithm is n^2.
- Instant recognition:
  - Assignments/arithmetic is O(1),
  - Loops are O(n),
  - Two nested loops are $O(n^2)$.
  - How about three nested loops?

```
for (int x=0; x<n; x++)
  for (int y=0; y<n; y++)
    for (int z=0; z<n; z++)
      sum = x + y + z;
```