Lecture 5a: Performance Analysis

CS 70: Data Structures and Program Development

Tuesday, February 18, 2020

Learning Goals for Today

1. I can describe strengths and weaknesses of timing code
2. I can describe strengths and weaknesses of counting operations
3. I can describe strengths and weaknesses of asymptotic analysis
4. I can contrast $o(g)$, $O(g)$, $\Theta(g)$, $\Omega(g)$, $\omega(g)$.
5. I can do asymptotic analyses for iterative (looping) functions.

Why Algorithms Matter

$$\frac{\text{age of the universe}}{\text{Planck time}} \approx 10^{61}$$

Why Algorithms Matter

$$\frac{\text{age of the universe}}{\text{Planck time}} \approx 10^{61}$$

$$\text{number of atoms in the observable universe} \approx 10^{80}$$

Why Algorithms Matter

$$\frac{\text{age of the universe}}{\text{Planck time}} \approx 10^{61}$$

$$\text{number of atoms in the observable universe} \approx 10^{80}$$

$$\text{max cpu cores} \times \text{max clock ticks} \approx 10^{141} \text{ steps} \quad \text{(universe, so far)}$$

(maximum *possible*: one core = one atom, one clock tick one Planck time)

Why Algorithms Matter

$$\frac{\text{age of the universe}}{\text{Planck time}} \approx 10^{61}$$

$$\text{number of atoms in the observable universe} \approx 10^{80}$$

$$\text{max cpu cores} \times \text{max clock ticks} \approx 10^{141} \text{ steps} \quad \text{(universe, so far)}$$

(maximum *possible*: one core = one atom, one clock tick one Planck time)

$$\text{Sorting 100 items by brute force} = 100! \text{ permutations} \approx 10^{158} \text{ steps}$$

But how can we *compare* different algorithms?

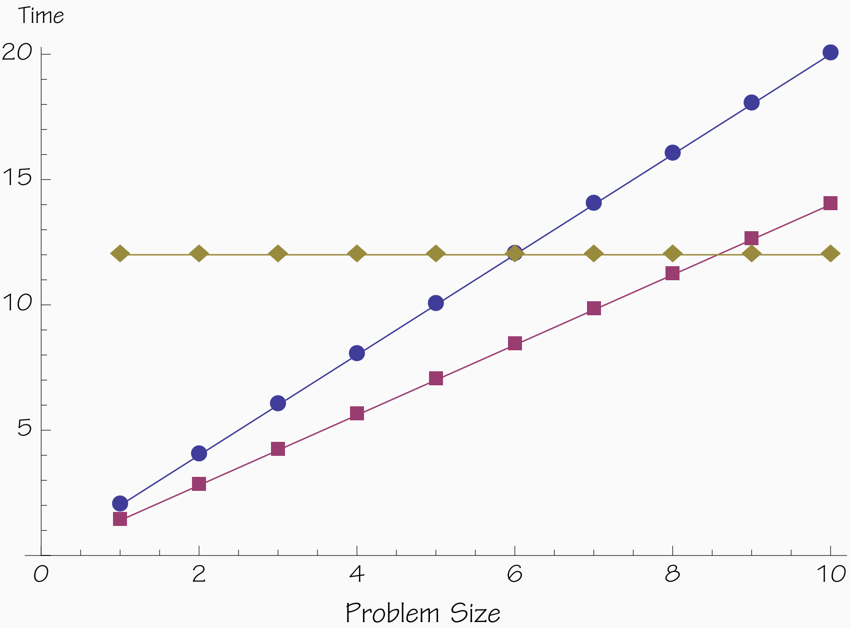
Benchmarking (Measuring Runtime)

Imagine this case study

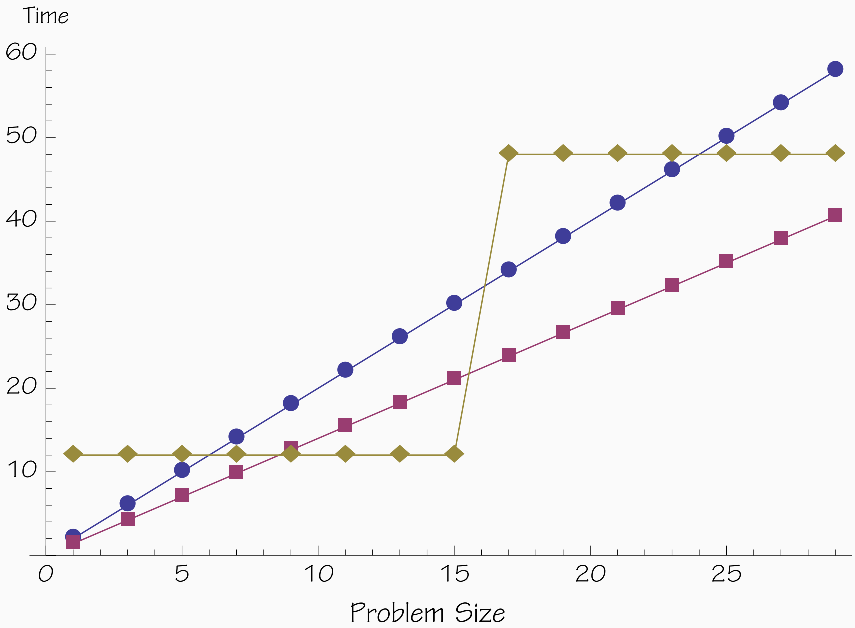
- At Xcomp '15, Professor Bauer of UCNY announces that the new sorting algorithm WildSort takes only 0.16 seconds to sort a list of 100,000 names.
- Later, at the same conference, Professor Taylor of SUNY SD reports the new algorithm SneakerSort takes only 0.03 seconds to sort a list of 100,000 names.

Your conclusion(s)? Which is better?

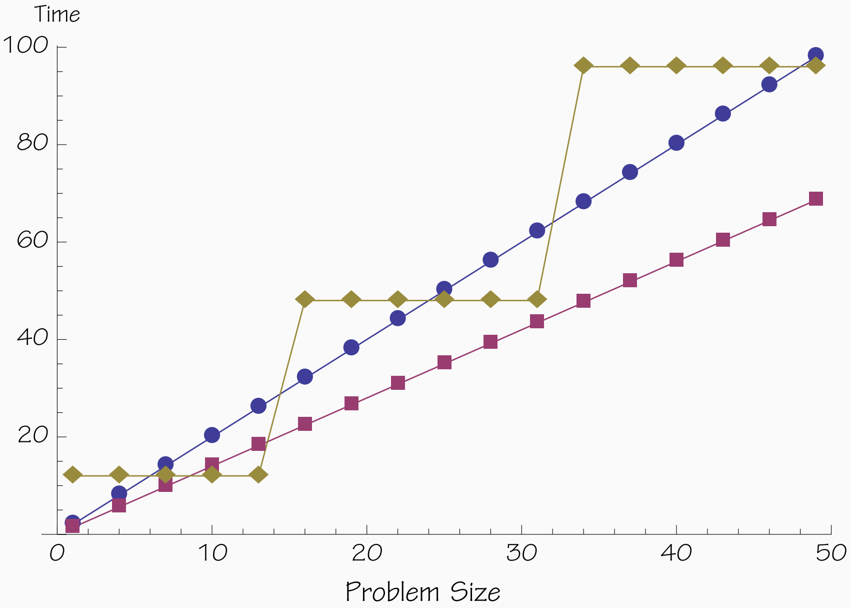
Which Program is “Fastest” ?



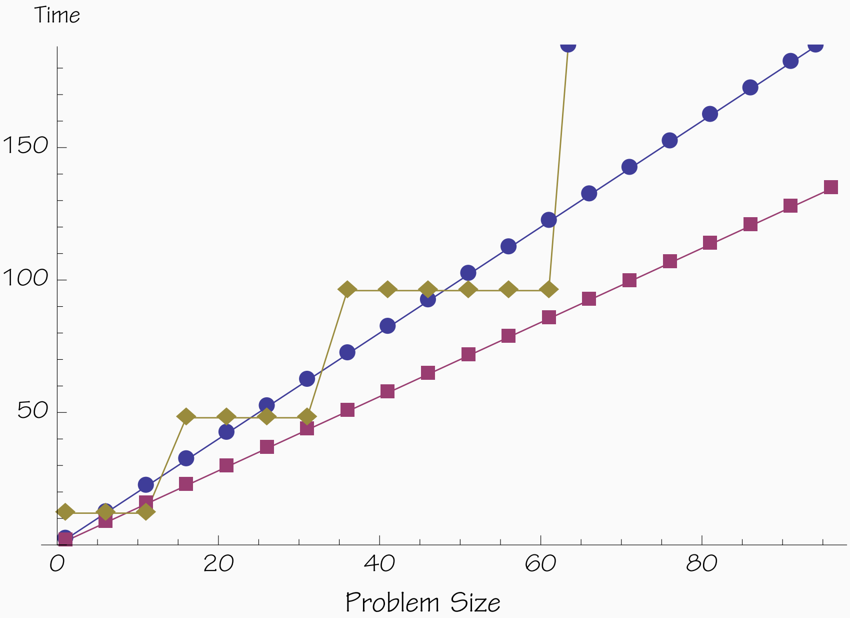
Which Program is “Fastest” ?



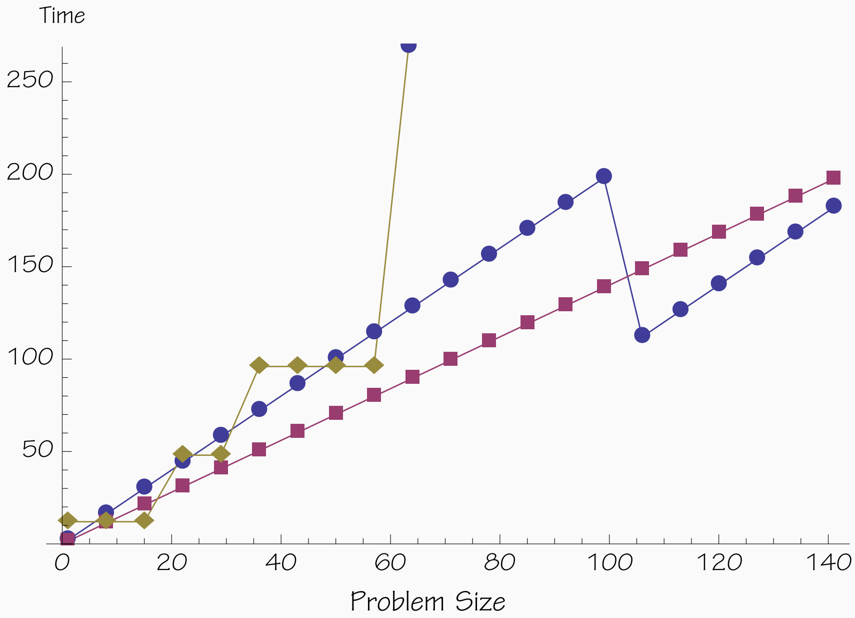
Which Program is “Fastest” ?



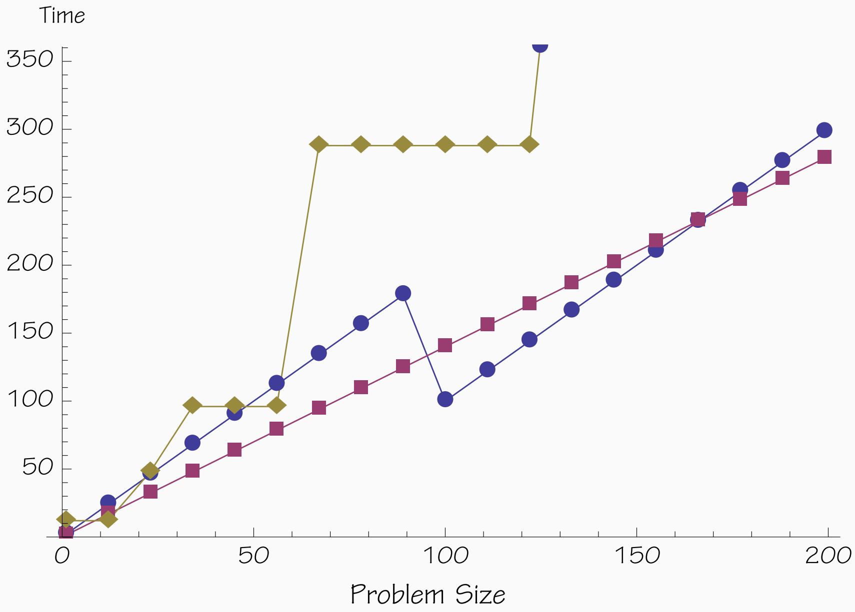
Which Program is “Fastest” ?



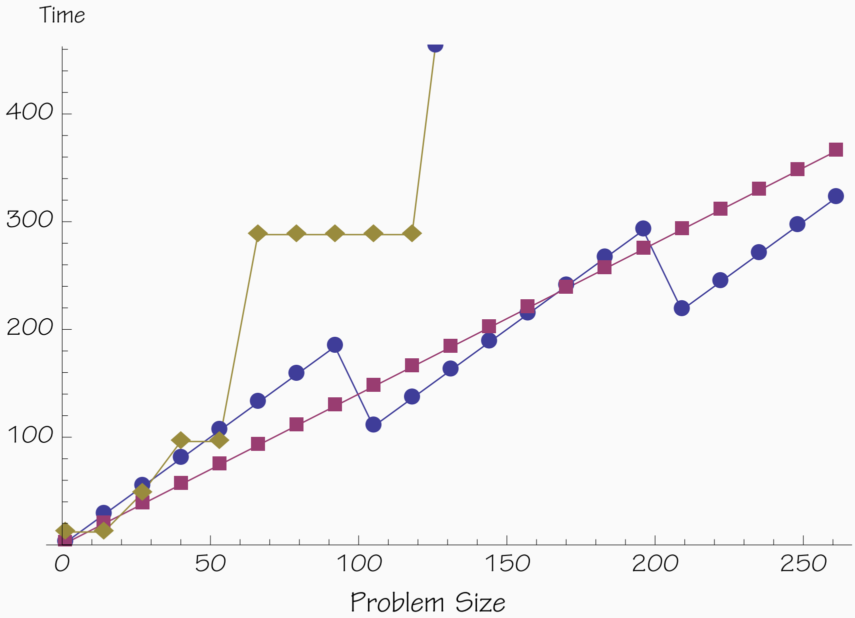
Which Program is “Fastest” ?



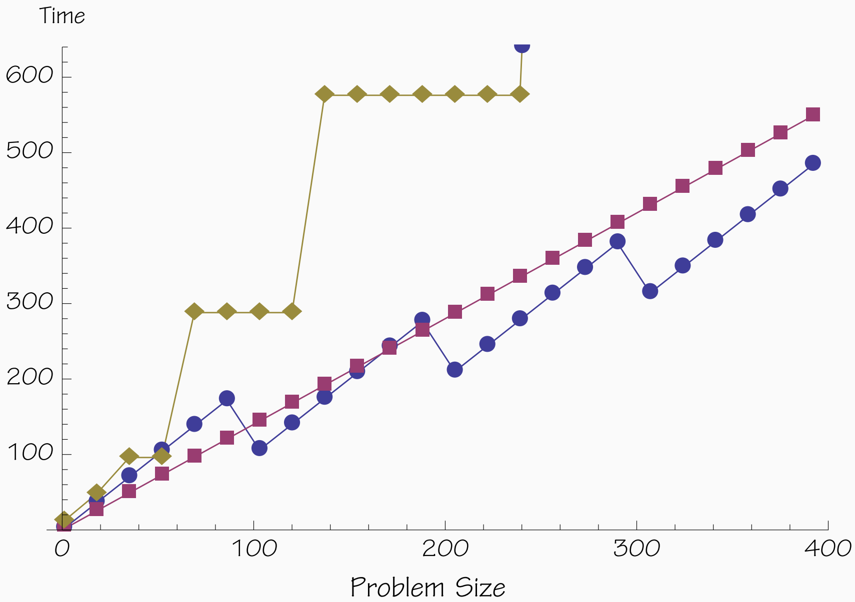
Which Program is “Fastest” ?



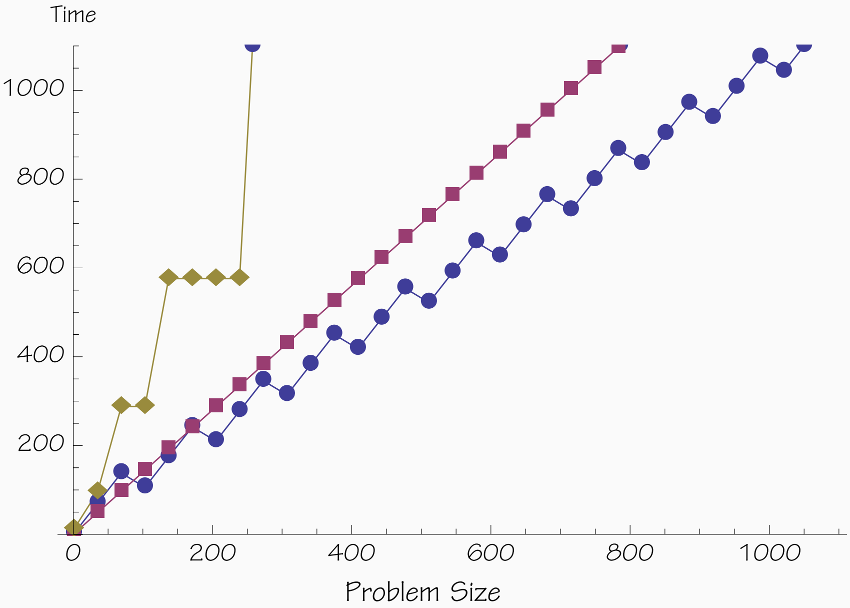
Which Program is “Fastest” ?



Which Program is "Fastest" ?



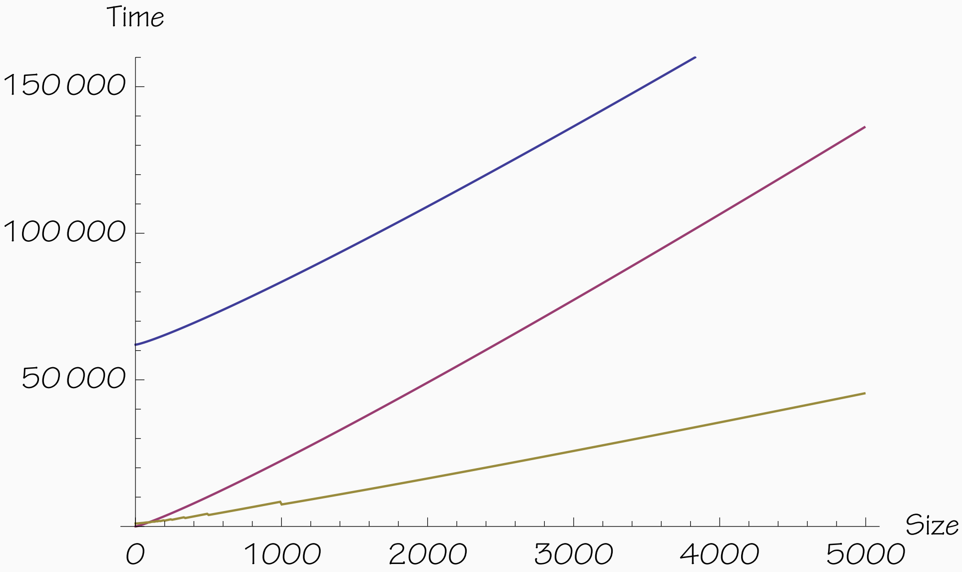
Which Program is “Fastest” ?



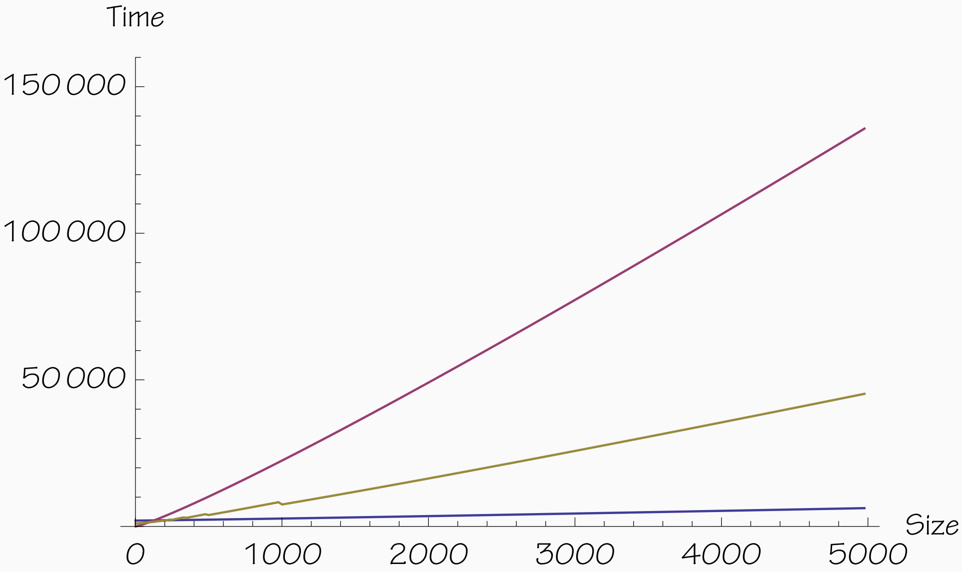
We can directly measure (benchmark)

- a specific algorithm
- written in a specific language
- as a specific program
- compiled using a specific version of a specific compiler
- with specific compiler flags settings (e.g., −g or not)
- running on a specific data set
- on a specific computer
- with a specific cpu (or cpus), memory, bus, hard drive, network card, ...
- under a specific version of a specific operating system
- while specific other programs run in the background.

Another example — oh, but blue is in Python



Switch blue to C++ for a 31x speedup:



Counting Steps

Donald Knuth, *Art of Computer Programing, Vol. 2*

[T]he determinant of almost all matrices ... can be computed with at most $(2n^3 - 3n^2 + 7n - 6)/6$ multiplications, $(2n^3 - 3n^2 + n)/6$ additions, and $(n^2 - n - 2)/2$ divisions.

Recall: 1D vs. 2D Arrays

Suppose we want to take in a sequence of characters like

10101110110111010001

and print these as 4 rows of 5 characters?

10101
11011
01110
10001

Outputs? comparisons? additions? multiplications?

```
int original[NUM_ROWS * NUM_COLS];

// ...load the digits into the array...

for (size_t row = 0; row < NUM_ROWS; ++row) {
    for (size_t col = 0; column < NUM_COLS; ++col) {
        cout << original[row * NUM_COLS + col];
    }
    cout << endl;
}
```

Concerns

Is incrementing the same as addition?

Concerns

Is incrementing the same as addition?

Does output buffering matter?

Concerns

Is incrementing the same as addition?

Does output buffering matter?

What about the implementation of indexing?

- Additional multiplications at run time?

Concerns

Is incrementing the same as addition?

Does output buffering matter?

What about the implementation of indexing?

- Additional multiplications at run time?

What about optimizations like “loop invariant hoisting” and “strength reduction”?

- No multiplications at run time?

Concerns

Is incrementing the same as addition?

Does output buffering matter?

What about the implementation of indexing?

- Additional multiplications at run time?

What about optimizations like “loop invariant hoisting” and “strength reduction”?

- No multiplications at run time?

Sum of operation times != total run-time

Michael Abrash, *The Zen of Assembly Language*

A few years back, I came across an article called "Optimizing for Speed". [The author] had clearly fine-tuned the code with care adding up cycles until he arrived at an implementation he calculated to be nearly 50% faster. There was, in fact, only one slight problem: it ran slower than the original version!

Asymptotic Analysis

Asymptotic Analysis (Big-O, Big-Θ, etc.)

Answers an abstract question about an algorithm:

- How do costs *scale* as input sizes become arbitrarily large?

Asymptotic Analysis (Big-O, Big-Θ, etc.)

Answers an abstract question about an algorithm:

- How do costs *scale* as input sizes become arbitrarily large?

Not

- How many seconds do we need for an input of size n
- How many bytes of memory are required?
- How many additions are performed?
- How easy is it to implement the algorithm?
- Is this the best algorithm for my problem?

Costs

Answers an abstract question about an algorithm:

- How do costs *scale* as input sizes become arbitrarily large?

"Cost" might refer to

- Time spent (today's focus)
- Bytes of memory required
- Bits transmitted over the network
- Watts of electricity consumed
- etc.

Asymptotic Analysis

Answers an abstract question about an algorithm:

- How do costs *scale* as input sizes become arbitrarily *large*?

Suppose a function with input size n takes $63n$ steps when it runs.

What is the ratio?

$$\frac{\#\text{steps on some input twice as big}}{\#\text{steps for some input}}$$

Asymptotic Analysis

Answers an abstract question about an algorithm:

- How do costs *scale* as input sizes become arbitrarily *large*?

Suppose a function with input size n takes $5n^3$ steps when it runs.

What is the ratio?

$$\frac{\#\text{steps on some input three times as big}}{\#\text{steps for some input}}$$

Asymptotic Analysis

Answers an abstract question about an algorithm:

- How do costs *scale* as input sizes become arbitrarily *large*?

Suppose a function with input size n takes $n^3 + 17$ steps when it runs.

What is the ratio

$$\frac{\#\text{steps on some input three times as big}}{\#\text{steps for some input}}$$

as n gets very large?

Consequences

If we only care about *scalability* for arbitrarily large inputs:

Constant factors don't matter.

- $0.01n$ and n and $9999n$ scale linearly
- $0.01n^2$ and n^2 and $9999n^2$ scale quadratically
- $\ln n$ and $\log_2 n$ and $\log_{10} n$ scale logarithmically

Consequences

If we only care about *scalability* for arbitrarily large inputs:

Constant factors don't matter.

- $0.01n$ and n and $9999n$ scale linearly
- $0.01n^2$ and n^2 and $9999n^2$ scale quadratically
- $\ln n$ and $\log_2 n$ and $\log_{10} n$ scale logarithmically

"Small" summands can be ignored

- $(n^2 + 100n + 1000000) \approx n^2$ for very large n
- $(n^n + n! + 2^n) \approx n^n$ for very large n

Comparing Growth Rates

	n	$n \log n$	n^2	n^3	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 18 min	10^{25} years
$n = 100$	< 1 sec	< 1 sec	1 sec	1s	10^{17} years	very long
$n = 1000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long

(adapted from [2], Table 2.1, pg. 34)

Simplify

1. $O(3n^2 + 2n + 2 + \cos(\pi n))$
2. $O(\log_{10}(n^3))$
3. $O(5n^{1.5} + 2n \log n)$
4. $O(n^2 + 2^n)$

More Consequences

We can count “steps” rather than “instructions” or “clock cycles”

- Both give the same asymptotic result
- A step can be a lot of work, as long as it’s bounded by a constant.

More Consequences

We can count "steps" rather than "instructions" or "clock cycles"

- Both give the same asymptotic result
- A step can be a lot of work, as long as it's bounded by a constant.

```
const int ARR_LENGTH = 80;
for (size_t i = 0; i < 50; ++i) {
  size_t arr[ARR_LENGTH]{i};
  for (size_t j = 0; j < ARR_LENGTH; ++j) {
    cout << arr[j] << " ";
  }
  cout << endl;
}
```

Be careful about identifying single steps!

```
string output;
for(size_t i = 0; i < n; ++i) {
    output += " " + to_string(i);
}
cout << output << endl;
```

Coarse-Grained Group: $O(1)$

- Takes 6 steps
- Takes 1 (big) step
- No more than 4000 steps
- Somewhere between 2 and $47,000$ steps, depending on the input

Coarse-Grained Group: $O(n)$

- Takes $100n + 3$ steps
- Takes $n/20 + 10,000,000$ steps
- Anywhere from 3 to 68 steps per item, for n items

Coarse-Grained Group: $O(n^2)$

- Takes $2n^2 + 100n + 3$ steps
- Takes $n^2/17$ steps.
- Somewhere between 1 and 40 steps per item, for n^2 items
- Anywhere between 1 and $7n$ steps per item, for n items

Making Life Simpler

If there's any one step that dominates (asymptotically), we can ignore everything else, e.g.,

```
for (int i = 0;  i < n;  ++i) {
    sum += 2;
}
```

Asymptotic notation, intuitively.

$f \in o(g)$ if f grows strictly less fast than g $\quad (<)$

$f \in O(g)$ if f grows no faster than g $\quad (<=)$

$f \in \Theta(g)$ if f grows at the same rate as g $\quad (=)$

$f \in \Omega(g)$ if f grows at least as fast as g $\quad (>=)$

$f \in \omega(g)$ if f grows strictly faster than g $\quad (>)$

What relationships hold between these classes?

$f \in o(g)$ if f grows strictly less fast than g $\quad (<)$

$f \in O(g)$ if f grows no faster than g $\quad (<=)$

$f \in \Theta(g)$ if f grows at the same rate as g $\quad (=)$

$f \in \Omega(g)$ if f grows at least as fast as g $\quad (>=)$

$f \in \omega(g)$ if f grows strictly faster than g $\quad (>)$

Asymptotic notation, more formally

$f \in o(g)$ if f stays below every multiple of g, eventually.

$f \in O(g)$ if f stays below some multiple of g, eventually.

$f \in \Theta(g)$ if f stays between two multiples of g, eventually.

$f \in \Omega(g)$ if f stays above some multiple of g, eventually,

$f \in \omega(g)$ if f stays above every multiple of g, eventually.

Asymptotic notation, even more formally

$f \in o(g)$ if $\forall c > 0.\ \exists N \geq 0.\ \forall n \geq N.\ f(n) \leq c \cdot g(n)$

$f \in O(g)$ if $\exists c > 0.\ \exists N \geq 0.\ \forall n \geq N.\ f(n) \leq c \cdot g(n)$

$f \in \Theta(g)$ if $\exists c_1, c_2 > 0.\ \exists N \geq 0.\ \forall n \geq N.\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

$f \in \Omega(g)$ if $\exists c > 0.\ \exists N \geq 0.\ \forall n \geq N.\ f(n) \geq c \cdot g(n)$

$f \in \omega(g)$ if $\forall c > 0.\ \exists N \geq 0.\ \forall n \geq N.\ f(n) \geq c \cdot g(n)$

Exercises

1. Is $3n \in O(n)$?
2. Is $3n \in \Omega(n)$?
3. Is $3n \in \Theta(n)$?
4. Is $3n \in O(n^2)$?
5. Is $3n \in \Omega(n^2)$?
6. Is $3n \in \Theta(n^2)$?

Exercises

1. Is $3n \in O(n)$?
2. Is $3n \in \Omega(n)$?
3. Is $3n \in \Theta(n)$?
4. Is $3n \in O(n^2)$?
5. Is $3n \in \Omega(n^2)$?
6. Is $3n \in \Theta(n^2)$?
7. Is $3n \in O(2^n)$?

Which should we avoid saying?

1. This algorithm isn't scalable because it takes $O(n^3)$ time.
2. This algorithm isn't scalable because it takes $\Theta(n^3)$ time.
3. This algorithm isn't scalable because it takes $\Omega(n^3)$ time.
4. Wow! I'm surprised we can sort n `ints` in $O(n)$ time with Radix Sort.
5. Wow! I'm surprised we can sort n `ints` in $\Theta(n)$ time with Radix Sort.
6. Wow! I'm surprised we can sort n `ints` in $\Omega(n)$ time with Radix Sort.

Warning!

Many programmers say $O(g)$ when they mean $\Theta(g)$

- "It's too slow; it's $O(n^3)$"

and further assume that hidden factors are always small

- "If you double the input size of an $O(n^2)$ algorithm, it will take four times as long."

Calculating Asymptotically

$$O(f) + O(g) = O(f + g).$$

$$O(f) \cdot O(g) = O(f \cdot g)$$

$$O(max\{f, g\}) = O(f + g).$$

Calculating Asymptotically

$O(f) + O(g) = O(f + g)$.

$O(f) \cdot O(g) = O(f \cdot g)$

$O(max\{f, g\}) = O(f + g)$.

(same for Θ and Ω)

A Very Common/Important Summation!

$$1 + 2 + 3 + \cdots + n \; = \frac{n(n+1)}{2} \in O(n^2)$$

Sample Calculations

Classify $T(n)$, the number of steps required for input n

```
for (int i = 0;  i < n;  ++i)
    ++sum;
```

Classify $T(n)$, the number of steps required for inputs n and m

```
for (int i = 0;  i < 2*n;  ++i)
    for (int j = 0;  j < m+1;  ++j)
        ++sum;
```

Classify $T(n)$, the steps required for input n

```
for (int i = 0;  i < n;  ++i)
    for (int j = 0;  j < i+1;  ++j)
        ++sum;
```

Classify $T(n, m)$, steps required for inputs n and m

```
for (int i = 0;  i < n;  ++i)
    ++sum;

for (int j = 0;  j < m;  ++j)
    ++sum;
```

Classify $T(n)$, the steps required for input n

```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < i; j += 2) {
        a[j+1] += 1;
        if (a[j+1] % 2 == 0) a[j] = 2*a[j];
    }
}
```