L16: Complexity

Scalability of computer codes to large problems

Lucas A. J. Bastien

E7 Spring 2017, University of California at Berkeley

February 24, 2017

Version: release

Announcements

Lab 06 is due on March 3 at 12 pm (noon)

Today:

- Complexity of algorithms (Chapter 7)

Next week:

- Monday: midterm review
- Wednesday: midterm (**bring your own scantron!**)
- Friday: Reading and writing data (Chapter 10)

On bCourses:

- Sample midterm with solutions

A few words about coding style

Keep it simple

- When faced with multiple options, go with the most simple approach

Use self-explanatory variable names. For example:

```
>> [n_rows, n_cols] = size(my_2d_array);
```

is preferred over:

```
>> [m, n] = size(my_2d_array);
```

Write short functions, each performing a specific task

- For example, if you have 4 levels or more of nested loops and/or `if`-statements, you may want to write separate (sub-)functions that perform smaller tasks

A few words about coding style (continued)

Include a detailed description of what your function does, in comments right below the function header

- Describe the function's input and output parameters
- Mention and/or describe the algorithms used by your function (cite relevant references)
- Describe any assumption made when coding the function
- Give examples of the use of your function

Use comments to document the rest of your code, but do not over-comment. Over-commenting includes describing what the code is doing where the code is self-explanatory. Examples of over-commenting:

```
>> % Assign the value 10 to the variable n
>> n = 10;
>> % Create a variable b that contains a n by n array of zeros
>> b = zeros(n, n);
```

Use comments to describe parts of your code that are difficult to understand by only looking at the code itself

Counting the number of operations (example 1)

```
>> my_sum = 0;
>> for i = 1:n
>>     my_sum = my_sum + 1/i;
>> end
```

How many operations of each kind are performed when executing the piece of code above (assume n is a positive integer of class `double`)?

Additions	n
Subtractions	0
Multiplications	0
Divisions	n
Variable assignments	$2n + 1$
Function calls	0
Total	$4n + 1$

Note: `my_sum` is assigned $n + 1$ times and `i` is assigned n times

Counting the number of operations (example 2)

```
>> my_sum = 0;
>> for i = 1:n
>>     for j = 1:n
>>         my_sum = my_sum + 1/i/j;
>>     end
>> end
```

How many operations of each kind are performed when executing the piece of code above (assume n is a positive integer of class `double`)?

Additions	n^2
Subtractions	0
Multiplications	0
Divisions	$2n^2$
Variable assignments	$2n^2 + n + 1$
Function calls	0
Total	$5n^2 + n + 1$

Note: `my_sum` is assigned $n^2 + 1$ times, `i` is assigned n times, and `j` is assigned n^2 times

Big-O notation

```
function [] = my_big_o_n(n)
my_sum = 0;
for i = 1:n
    my_sum = my_sum + 1/i;
end
end
```

```
function [] = my_big_o_n_square(n)
my_sum = 0;
for i = 1:n
    for j = 1:n
        my_sum = my_sum + 1/i/j;
    end
end
end
```

$4n + 1$ operations

$5n^2 + n + 1$ operations

$$\lim_{n \to \infty} 4n + 1 = 4n$$

$$\lim_{n \to \infty} 5n^2 + n + 1 = 5n^2$$

Complexity: $\mathcal{O}(n)$

Complexity: $\mathcal{O}(n^2)$

We drop the multiplicative constants in big-O notation

Big-O notation (continued)

Big-O notation describes how much more resources are needed to solve a problem as the size of the problem increases

$\mathcal{O}(n)$:

- If the size of the problem doubles, we need twice as much resources
- If the size of the problem triples, we need 3 times as much resources
- If the size of the problem quadruples, we need 4 times as much resources

$\mathcal{O}(n^2)$:

- If the size of the problem doubles, we need 4 times as much resources
- If the size of the problem triples, we need 9 times as much resources
- If the size of the problem quadruples, we need 16 times as much resources

$\mathcal{O}(n^3)$:

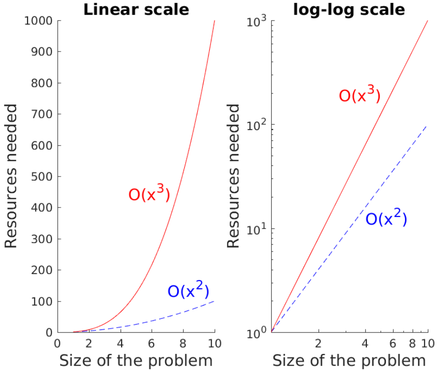
- If the size of the problem doubles, we need 8 times as much resources
- If the size of the problem triples, we need 27 times as much resources
- If the size of the problem quadruples, we need 64 times as much resources

Plotting big-O notation on log-log scales

Using log-log scales often makes it easier to visualize data that span a very broad range of values (several orders of magnitude)

$\mathcal{O}(n^p)$:

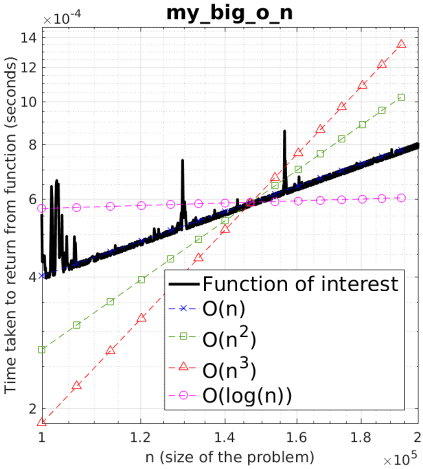
- resources needed $\approx$ constant $\times$ (size of problem)^p
- log(resources needed) $\approx$ log(constant) $+$ p $\times$ log(size of problem)
- $y = ax^p$: appears as a straight line of slope p on a log-log plot



Time complexity of my_big_o_n

Comparison of time complexity:

- **predicted** (theory) versus
- **observed** (black solid line, actually measured by Matlab)



Plot created using the functions my_big_o_n and my_time_complexity (both are posted on bCourses)

Time complexity of my_big_o_n_square

Comparison of time complexity:

- **predicted** (theory) versus
- **observed** (black solid line, actually measured by Matlab)

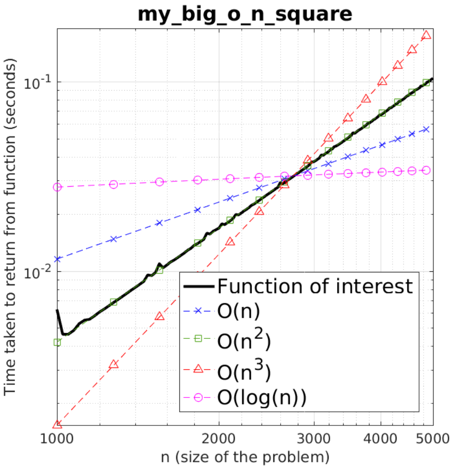


Figure created using the functions my_big_o_n_square and my_time_complexity (both are posted on bCourses)

Time complexity of my_big_o_n_cube

Comparison of time complexity:

- **predicted** (theory) versus
- **observed** (black solid line, actually measured by Matlab)

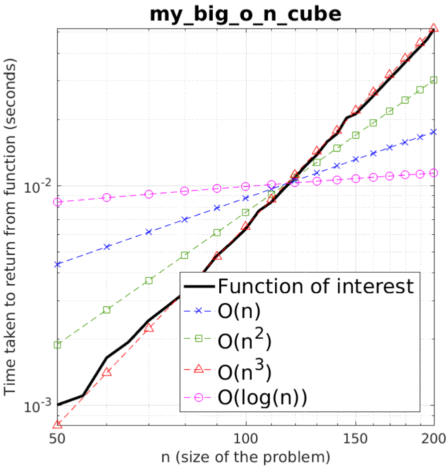


Figure created using the functions my_big_o_n_cube and my_time_complexity (both are posted on bCourses)

Log complexity

If the size of a problem decreases by a factor of 2 or more at each iteration, then the complexity of a problem in big-O notation is $\mathcal{O}(\log(n))$

For example, the following function has $\mathcal{O}(\log(n))$ complexity:

```
function [] = my_big_o_log_n(n)
while abs(n) > 1
    n = n / 3;
end
end
```

Why $\mathcal{O}(\log(n))$? Cut n in thirds k times until it is $\leqslant 1$ (assume $n > 0$):

$$\frac{n}{3 \times 3 \times \cdots \times 3} = \frac{n}{3^k} = 1 \quad \text{yields} \quad k = \log_3 n \quad \rightarrow \quad \mathcal{O}(\log(n))$$

The base of the log (2, 3, 10, e, ...) does not matter in big-O notation (the difference is just a multiplicative constant)

Time complexity of my_big_log_n

Comparison of time complexity:

- **predicted** (theory) versus
- **observed** (black solid line, actually measured by Matlab)

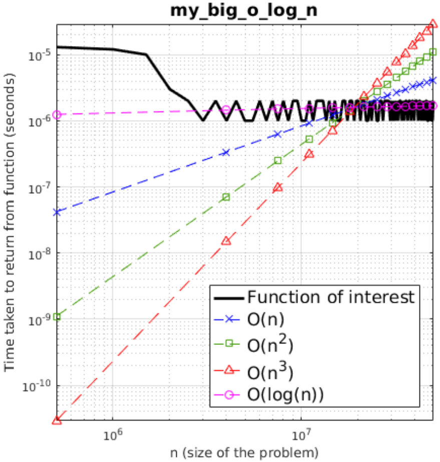


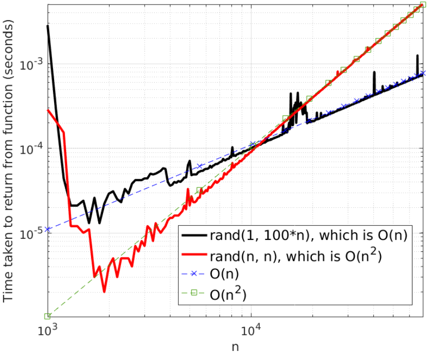
Figure created using the functions my_big_o_log_n and my_time_complexity (both are posted on bCourses)

Important distinction (example with O(n) versus O(n^2))

A $\mathcal{O}(n)$ function does not always execute faster than a $\mathcal{O}(n^2)$ function for the same value of n. However:

It is always possible to find a value N such that if $n > N$, then the $\mathcal{O}(n)$ function will execute faster than the $\mathcal{O}(n^2)$ function

In other words, the $\mathcal{O}(n)$ function eventually becomes faster than the $\mathcal{O}(n^2)$ function if n is sufficiently large



Different types of complexity

- **Time complexity:** How much more time will my function/program need to solve bigger and bigger problems?

- **Memory (RAM) complexity:** How much more memory (RAM) will my function/program need to solve bigger and bigger problems?

  For example, the following function call is $\mathcal{O}(n^2)$ in memory complexity:

```
>> % Create an n by n array of random numbers
>> rand(n, n);
```

- **Disc space complexity:** How much more hard drive space will my function/program need to solve bigger and bigger problems?
  - Either because it needs to **read data from disc**, and/or
  - Because it needs to **write data to disc**

Measuring computer memory and storage

Amount	Number of bytes	Equivalent number of double-precision numbers	Size of the corresponding $n \times n$ array of class `double`
1 Kilobyte	10^3	125	$\approx 11 \times 11$
1 Megabyte	10^6	$125,000$	$\approx 354 \times 354$
1 Gigabyte	10^9	$125,000,000$	$\approx 11,180 \times 11,180$
1 Terabyte	10^{12}	$125,000,000,000$	$\approx 353,553 \times 353,553$

Typical amounts in 2017's personal computers:

- Memory (RAM): 4 – 16 Gigabytes
- Hard drive: 250 Gigabytes – 3 Terabytes

Remember: 1 byte = 8 bits

External storage

- Up to the ≈2000's: floppy disks!

Floppy disks (from left to right):

- 8-inch (≈ 1 Megabyte)
- 5.25-inch (≈ 1 Megabyte)
- 3.5-inch (1.44 Megabyte)

- CD (compact disk): ≈ 700 Megabytes
- DVD: ≈ 4.7 Gigabytes (higher capacity exists)
- 2017's typical desktop external hard drive: 0.5 – 4 Terabytes

Image retrieved from https://en.wikipedia.org/wiki/File:Floppy_disk_2009_G1.jpg on February 24$^{\text{th}}$ 2017

Concluding remark(s)

Today, we saw $\mathcal{O}(n^p)$ complexities (with $p \geqslant 1$) and $\mathcal{O}(\log(n))$ complexities

There are other possible complexities, for example:

- $\mathcal{O}(2^n)$ (exponential, see recursive implementation of Fibonacci sequences in textbook page 115 for an example)
- $\mathcal{O}(n\log(n))$. For example:

```
function [] = my_function_nlogn(n)
for i = 1:n
    x = abs(n);
    while x > 10
        x = x / 2;
    end
end
end
```