CSE 503

Introduction to Computer Science for Non-Majors

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Day 33
Asymptotic Notation

Recap

- Two different sorting algorithms: **`SelectionSort`** and **`MergeSort`**
  - **`SelectionSort`** on list of size $\boldsymbol{N}$ requires $\boldsymbol{N^2}$ steps
  - **`MergeSort`** on a list of size $\boldsymbol{N}$ requires $\boldsymbol{N \log(N)}$ steps
  - As we try larger and larger inputs, $\boldsymbol{N^2}$ grows much faster than $\boldsymbol{N \log(N)}$

Recap

- Two different sorting algorithms: **`SelectionSort`** and **`MergeSort`**
  - **`SelectionSort`** on list of size $\boldsymbol{N}$ requires $\boldsymbol{N^2}$ steps
  - **`MergeSort`** on a list of size $\boldsymbol{N}$ requires $\boldsymbol{N \log(N)}$ steps
  - As we try larger and larger inputs, $\boldsymbol{N^2}$ grows much faster than $\boldsymbol{N \log(N)}$

How much faster is this really (in practice)?

Can we formalize this notion of "complexity"?

Sorting Comparison in Python

Tim Sort

The sorting algorithm used by Python is **Tim Sort**

It is a hybrid sorting algorithm: it uses a slower sorting algorithm (insertion sort) for small inputs, and a fast algorithm (merge sort) for large inputs

It requires ***N* log(*N*)** steps just like MergeSort, but has lower overheads

It is very fast…

Tim Sort

The sorting algorithm used by Python is **Tim Sort**

It is a hybrid sorting algorithm: it uses a slower sorting algorithm (insertion sort) for small inputs, and a fast algorithm (merge sort) for large inputs

It requires $\boldsymbol{N \log(N)}$ steps just like MergeSort, but has lower overheads

It is very fast…**so in your own programs, just use `.sort()`**

Sorting Custom Data

What if we want to sort a list that isn't numbers or strings?

Sorting Custom Data

The sort function can take an extra argument for determining how to sort the items of the list.

It is a function that takes a list item as input, and returns a key that can be sorted as output.

Sorting Custom Data

```
students = [
  { "fname": "Sally", "lname":"Smith", "pn":"342083", "age":"23" },
  { "fname": "Barb",  "lname":"Woods", "pn":"934850", "age":"21" },
  { "fname": "Bo",    "lname":"Meele", "pn":"393847", "age":"22" },
  { "fname": "Amy",   "lname":"Fable", "pn":"705834", "age":"21" }
]

def byFirstName(V): return V["fname"]
def byFirstNameLength(V): return len(V["fname"])

students.sort(key = byFirstName)
students.sort(key = byFirstNameLength)
```

Formalizing Complexity

Tactical Programming

Go from point A to point B

1. Move up 100 feet
2. Turn right, move forward 200 feet
3. Move north 10 feet then turn left
4. Move forward 20 feet
5. Move south 50 feet
6. Move west 150 feet, then turn left
7. Move forward 60 feet

We can optimize each individual step

- For example, taking a bike will speed up step 2 compared to walking

Strategic Programming

Look at the big picture

Design (not just implement) an algorithm

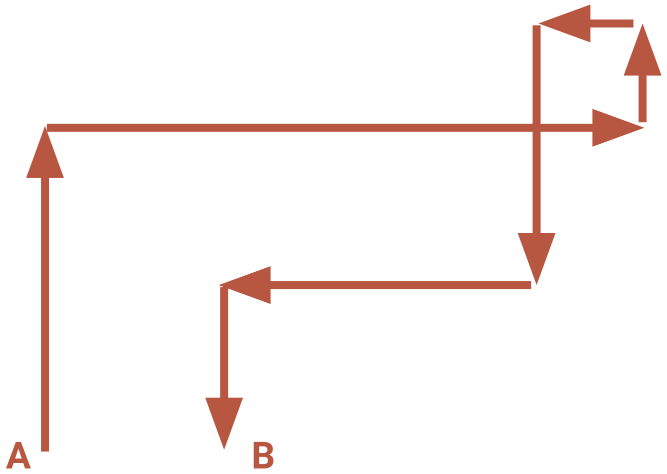
Focus on "complexity"

Strategic Programming

Look at the big picture

Design (not just implement) an algorithm

Focus on "complexity"

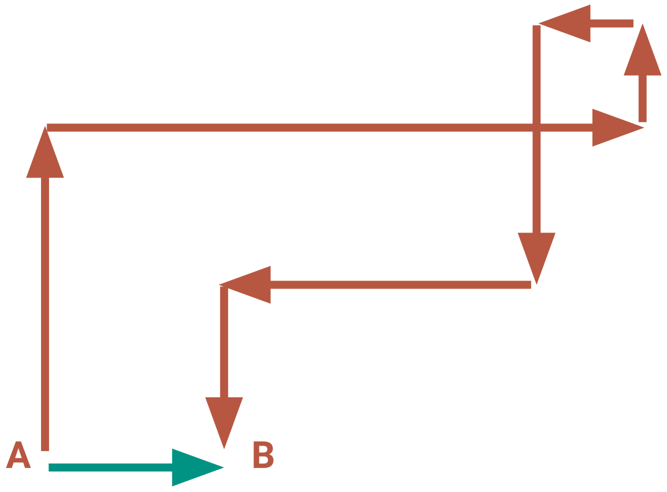


Strategic Programming

Look at the big picture

Design (not just implement) an algorithm

Focus on "complexity"



Why not just move east 30 feet...

Complexity

We don't want to spend time optimizing details of an algorithm that is more complex than it needs to be!

First and foremost, we want to choose the right algorithm. Then we can start optimizing the details later.

Big-O Notation

Let $f(n)$, $g(n)$ be non-negative, non-decreasing functions

f is said to be $O(g)$ if there exist *constants* c and k such that:

$$f(n) \leq c\, g(n) \text{ for all } n \geq k$$

Big-Ω Notation

Let $f(n)$, $g(n)$ be non-negative, non-decreasing functions

f is said to be $\Omega(g)$ if there exist *constants* c and k such that:

$$c\, g(n) \leq f(n) \text{ for all } n \geq k$$

Big-Θ Notation

Let $f(n)$, $g(n)$ be non-negative, non-decreasing functions

f is said to be $\Theta(g)$ if there exist *constants* c_1, c_2 and k such that:

$$c_1\, g(n) \le f(n) \le c_2\, g(n) \text{ for all } n \ge k$$

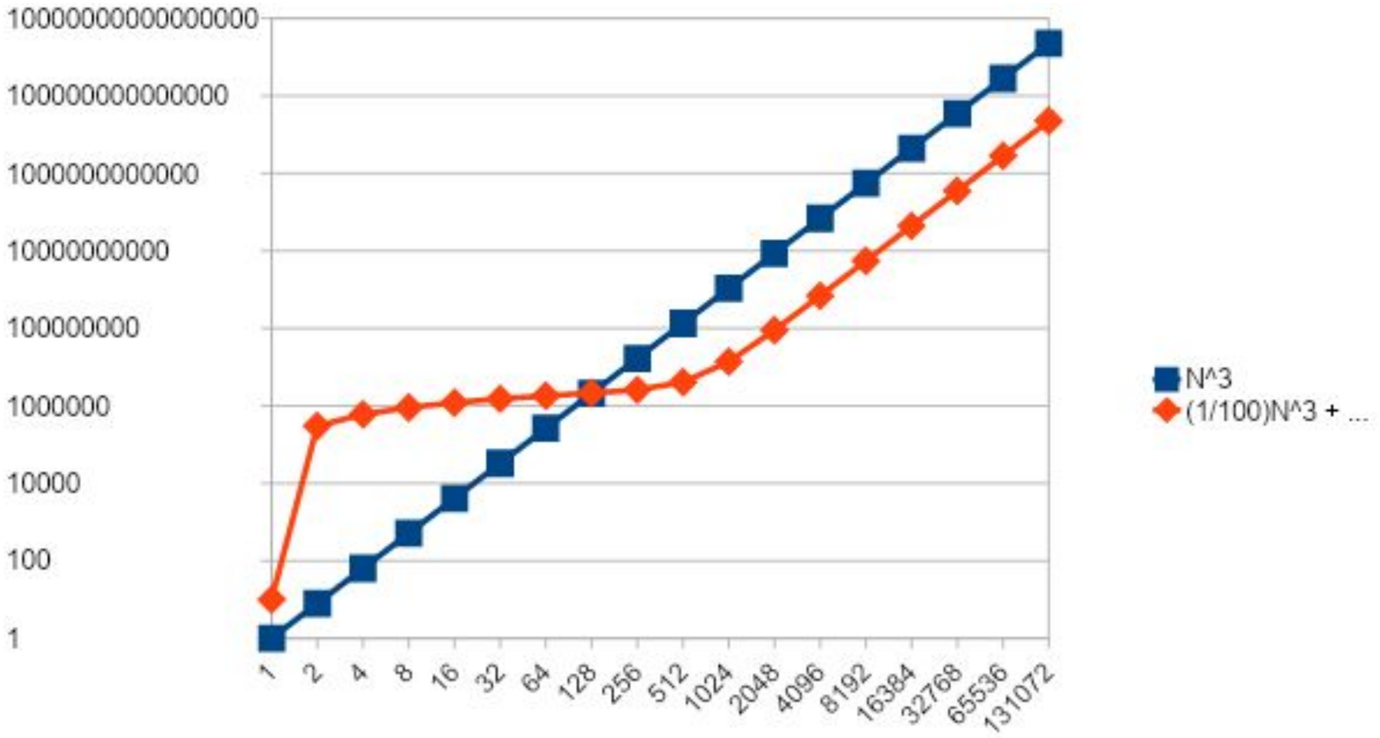
Example

Consider the following two functions:

$$f(n) = \frac{1}{100}n^3 + 10n + 1000000log(n)$$

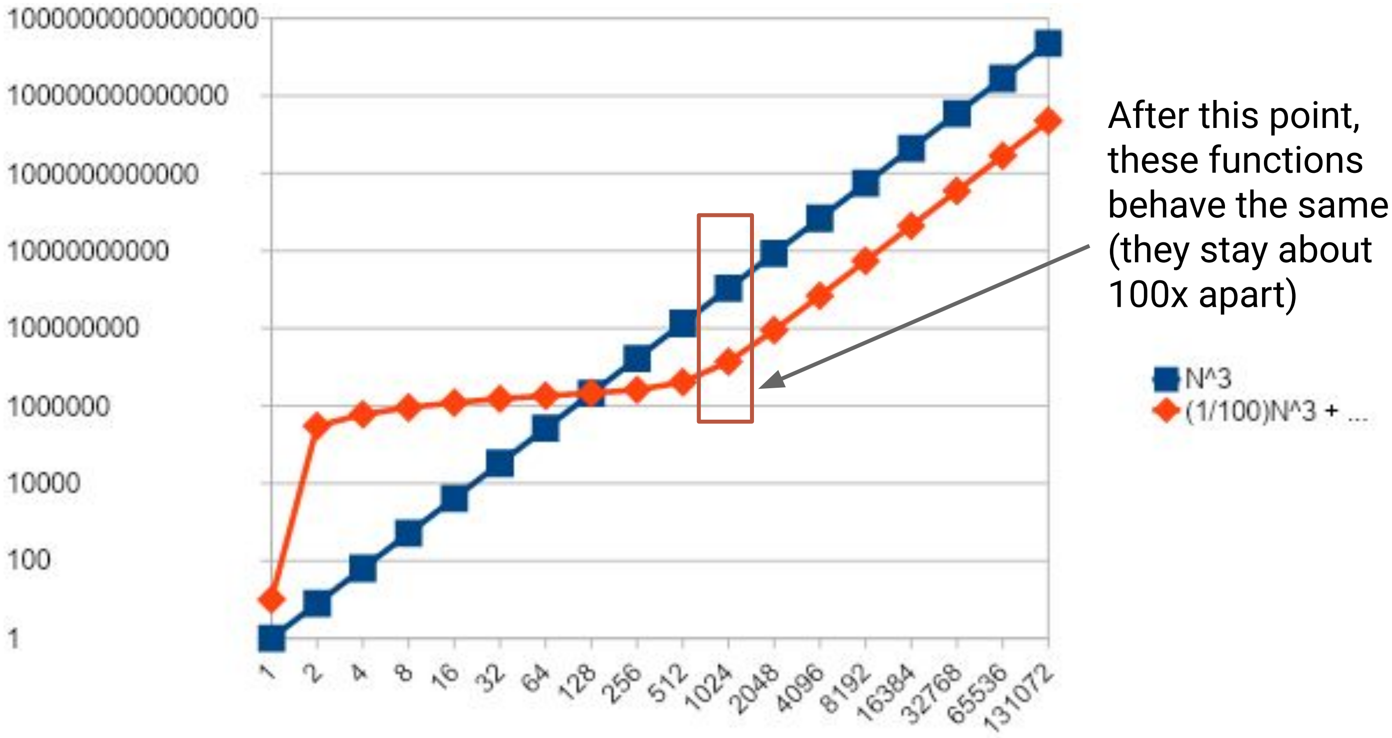
$$g(n) = n^3$$

Example



10000000000000000
1000000000000000
1000000000000
10000000000
100000000
1000000
10000
100
1
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
N^3
(1/100)N^3 + ...

Example



After this point, these functions behave the same (they stay about 100x apart)

Example

$$f(n) = \frac{1}{100}n^3 + 10n + 1000000log(n)$$

$$g(n) = n^3$$

Therefore:

$$f(n) \in O(g(n))$$

Example

$$f(n) = \frac{1}{100}n^3 + 10n + 1000000log(n)$$

$$g(n) = n^3$$

Therefore:

$$f(n) \in \Omega(g(n))$$

Example

$$f(n) = \frac{1}{100}n^3 + 10n + 1000000log(n)$$

$$g(n) = n^3$$

Therefore:

$$f(n) \in \Theta(g(n))$$

What does this mean/why does it matter?

Computer science can be used to solve HUGE problems:

- Simulating spread of disease across huge populations
- Rendering millions of points in 3D animation
- Simulating all the celestial bodies in the galaxy

…and plenty more

What does this mean/why does it matter?

Computer science can be used to solve HUGE problems:

- Simulating spread of disease across huge populations
- Rendering millions of points in 3D animation
- Simulating all the celestial bodies in the galaxy

…and plenty more

We need to know how our algorithms will perform on large inputs!

Complexity Classes

$f(n)$	**10**	**20**	**50**	**100**	**1000**
$log(log(n))$	0.43 ns	0.52 ns	0.62 ns	0.68 ns	0.82 ns
$log(n)$	0.83 ns	1.01 ns	1.41 ns	1.66 ns	2.49 ns
n	2.5 ns	5 ns	12.5 ns	25 ns	0.25 μs
$nlog(n)$	8.3 ns	22 ns	71 ns	0.17 μs	2.49 μs
n^2	25 ns	0.1 μs	0.63 μs	2.5 μs	0.25 ms
n^5	25 μs	0.8 ms	78 ms	2.5 s	**2.9 days**
2^n	0.25 μs	0.26 ms	**3.26 days**	**10^{13} years**	**10^{284} years**
$n!$	0.91 ms	**19 years**	**10^{47} years**	**10^{141} years**	🤯

Complexity Classes

$$2^n \gg n^c \gg n \gg log(n) \gg c$$

SelectionSort vs MergeSort

SelectionSort requires roughly N^2 steps to sort a list of size N

N^2 grows pretty fast…

MergeSort requires roughly $N \log(N)$ steps to sort a list of size N

$N \log(N)$ grows much slower

SelectionSort is $\Theta(n^2)$

MergeSort is $\Theta(n \log(n))$

SelectionSort vs MergeSort

SelectionSort requires roughly N^2 steps to sort a list of size N

N^2 grows pretty fast…

MergeSort requires roughly $N \log(N)$ steps to sort a list of size N

$N \log(N)$ grows much slower

SelectionSort is $\Theta(n^2)$

MergeSort is $\Theta(n \log(n))$

This is the fastest we can sort!

SelectionSort vs MergeSort

SelectionSort requires roughly N^2 steps to sort a list of size N

N^2 grows pretty fast…

MergeSort requires roughly $N \log(N)$ steps to sort a list of size N

$N \log(N)$ grows much slower

Tim Sort is also $\Theta(n \log(n))$

SelectionSort is $\Theta(n^2)$

MergeSort is $\Theta(n \log(n))$

This is the fastest we can sort!