ECE 250 *Algorithms and Data Structures*

An introduction to hash tables

Adapted for ECE650 by Arie Gurfinkel

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ece.uwaterloo.ca
dwharder@alumni.uwaterloo.ca

WATERLOO
ENGINEERING

Outline

Discuss storing unrelated/unordered data

- IP addresses and domain names

Consider conversions between these two forms

Introduce the idea of hashing:

- Reducing $\mathbf{O}(\ln(n))$ operations to $\mathbf{O}(1)$

Consider some of the weaknesses

Supporting Example

Suppose we have a system which is associated with approximately 150 error conditions where

- Each of which is identified by an 8-bit number from 0 to 255, and
- When an identifier is received, a corresponding error-handling function must be called

We could create an array of 150 function pointers and to then call the appropriate function….

9.1.1.1

Supporting Example

```
#include <iostream>

void a() {
    std::cout
        << "Calling 'void a()'"
        << std::endl;
}

void b() {
    std::cout
        << "Calling 'void b()'"
        << std::endl;
}
```

```
int main() {
    void (*function_array[150])();
    unsigned char error_id[150];

    function_array[0] = a;
    error_id[0] = 3;
    function_array[1] = b;
    error_id[1] = 8;

    function_array[0]();
    function_array[1]();

    return 0;
}
```

Output:

```
% ./a.out
Calling 'void a()'
Calling 'void b()'
```

Supporting Example

Unfortunately, this is slow—we would have to do some form of binary search in order to determine which of the 150 slots corresponds to, for example, error-condition identifier `id = 198`

This would require approximately 6 comparisons per error condition

If there was a possibility of dynamically adding new error conditions or removing defunct conditions, this would substantially increase the effort required…

Supporting Example

A better solution:

- Create an array of size 256
- Assign those entries corresponding to valid error conditions

```
int main() {
    void (*function_array[256])();
    for ( int i = 0; i < 256; ++i ) {
        function_array[i] = nullptr;
    }

    function_array[3] = a;
    function_array[8] = b;

    function_array[3]();
    function_array[8]();

    return 0;
}
```

Question:

- Is the increased speed worth the allocation of additional memory?

Keys

Our goal:

Store data so that all operations are $\Theta(1)$ time

Requirement:

The memory requirement should be $\Theta(n)$

In our supporting example, the corresponding function can be called in $\Theta(1)$ time and the array is less than twice the optimal size

Keys

In our example, we:

- Created an array of size 256
- Store each of 150 objects in one of the 256 entries
- The error code indicated which bin the corresponding function pointer was stored

In general, we would like to:

- Create an array of size M
- Store each of n objects in one of the M bins
- Have some means of determining the bin in which an object is stored

9.1.3.1

The hashing problem

The process of mapping an object or a number onto an integer in a given range is called *hashing*

Problem: multiple objects may hash to the same value

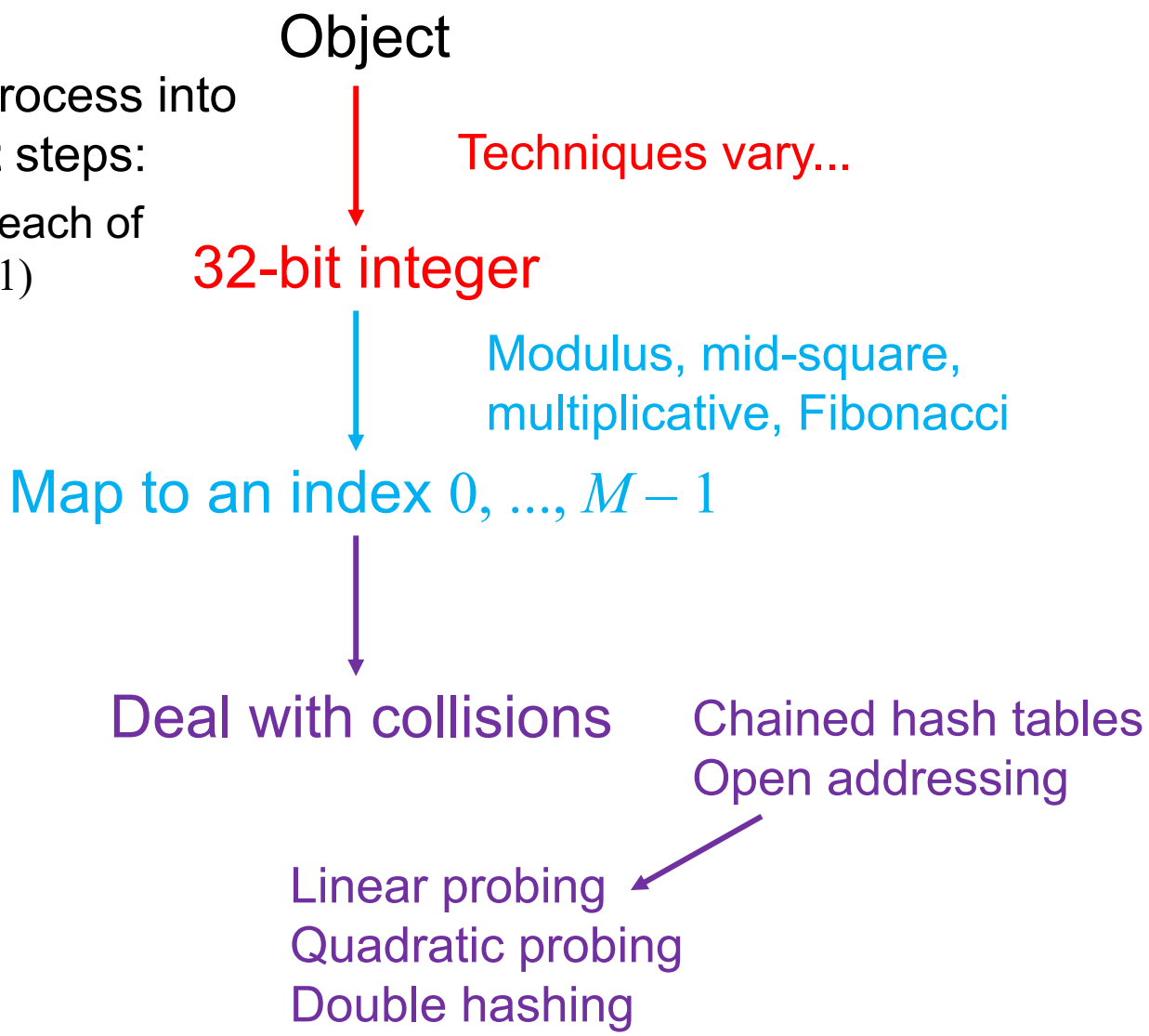
- Such an event is termed a *collision*

Hash tables use a hash function together with a mechanism for dealing with collisions

The hash process

We will break the process into three **independent** steps:

- We will try to get each of these down to $\Theta(1)$



In this talk, we will discuss

- Finding 32-bit hash values using:
  - Predetermined hash values
    - Auto-incremented hash values
    - Address-based hash values
  - Arithmetic hash values
- Example: strings

Properties

Necessary properties of such a hash function h are:

1a. Should be fast: ideally $\Theta(1)$

1b. The hash value must be *deterministic*

- It must always return the same 32-bit integer each time

1c. Equal objects hash to equal values

- $x = y \Rightarrow h(x) = h(y)$

1d. If two objects are randomly chosen, there should be only a one-in-2^{32} chance that they have the same hash value

9.2.3

Types of hash functions

We will look at two classes of hash functions

- Predetermined hash functions (explicit)
- Arithmetic hash functions (implicit)

9.2.4

Predetermined hash functions

For example, an auto-incremented static member variable

```
class Class_name {
    private:
        unsigned int hash_value;
        static unsigned int hash_count;
    public:
        Class_name();
        unsigned int hash() const;
};

unsigned int Class_name::hash_count = 0;
```

```
Class_name::Class_name() {
    hash_value = hash_count;
    ++hash_count;
}

unsigned int Class_name::hash() const {
    return hash_value;
}
```

9.2.4

Predetermined hash functions

Examples: All UW co-op student have two hash values:

- UW Student ID Number
- Social Insurance Number

Any 9-digit-decimal integer yields a 32-bit integer

$$\lg(\ 10^9\ ) = 29.897$$

9.2.4

Predetermined hash functions

If we only need the hash value while the object exists in memory, use the address:

```
unsigned int Class_name::hash() const {
   return reinterpret_cast<unsigned int>( this );
}
```

This fails if an object may be stored in secondary memory

- It will have a different address the next time it is loaded

9.2.4.1

Predetermined hash functions

Predetermined hash values give each object a unique hash value

This is not always appropriate:

- Objects which are conceptually equal:

```
Rational x(1, 2);
Rational y(3, 6);
```

- Strings with the same characters:

```
string str1 = "Hello world!";
string str2 = "Hello world!";
```

These hash values must depend on the member variables

- Usually this uses arithmetic functions

9.2.5

Arithmetic Hash Values

An arithmetic hash value is a deterministic function that is calculated from the relevant member variables of an object

We will look at arithmetic hash functions for:

- Rational numbers, and
- Strings

9.2.5.1

Rational number class

What if we just add the numerator and denominator?

```
class Rational {
    private:
        int numer, denom;
    public:
        Rational( int, int );
};

unsigned int Rational::hash() const {
    return static_cast<unsigned int>( numer ) +
        static_cast<unsigned int>( denom );
}
```

9.2.5.1

Rational number class

We could improve on this: multiply the denominator by a large prime:

```
class Rational {
    private:
        int numer, denom;
    public:
        Rational( int, int );
};

unsigned int Rational::hash() const {
    return static_cast<unsigned int>( numer ) +
        429496751*static_cast<unsigned int>( denom );
}
```

9.2.5.1

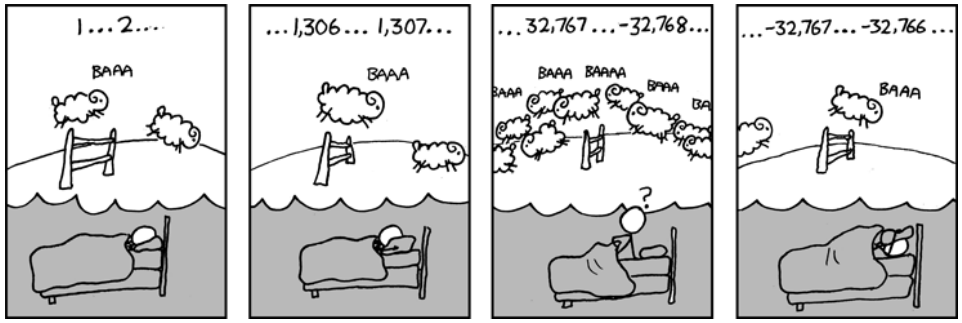
Rational number class

For example, the output of

```
int main() {
    cout << Rational(  0,   1 ).hash() << endl;
    cout << Rational(  1,   2 ).hash() << endl;
    cout << Rational(  2,   3 ).hash() << endl;
    cout << Rational( 99, 100 ).hash() << endl;

    return 0;
}
```

is

```
429496751
858993503
1288490255
2239
```



http://xkcd.com/571/

Recall that arithmetic operations wrap on overflow

9.2.5.1

Rational number class

This hash function does not generate unique values

- The following pairs have the same hash values:

0/1	1327433019/800977868
1/2	534326814/1480277007
2/3	820039962/1486995867

- Finding rational numbers with matching hash values is very difficult:
- Finding these required the generation of $1\ 500\ 000\ 000$ random rational numbers
- It is fast: $\Theta(1)$
- It does produce an even distribution

9.2.5.1

Rational number class

Problem:

- The rational numbers 1/2 and 2/4 have different values
- The output of

```
int main() {
    cout << Rational( 1, 2 ).hash();
    cout << Rational( 2, 4 ).hash();
    return 0;
}
```

is

```
858993503
1717987006
```

9.2.5.1

Rational number class

Solution: divide through by the greatest common divisor

```
Rational::Rational( int a, int b ):numer(a), denom(b) {
    int divisor = gcd( numer, denom );
    numer /= divisor;
    denom /= divisor;
}
```

```
int gcd( int a, int b) {
    while( true ) {
        if ( a == 0 ) {
            return (b >= 0) ? b : -b;
        }

        b %= a;

        if ( b == 0 ) {
            return (a >= 0) ? a : -a;
        }
        a %= b;
    }
}
```

9.2.5.1

Rational number class

Problem:

- The rational numbers $\frac{1}{2}$ and $\frac{-1}{-2}$ have different values
- The output of

```
int main() {
    cout << Rational(  1,  2 ).hash();
    cout << Rational( -1, -2 ).hash();
    return 0;
}
```

is

```
858993503
3435973793
```

9.2.5.1

Rational number class

Solution: define a normal form

- Require that the denominator is positive

```
Rational::Rational( int a, int b ):numer(a), denom(b) {
    int divisor = gcd( numer, denom );
    divisor = (denom >= 0) ? divisor : -divisor;
    numer /= divisor;
    denom /= divisor;
}
```

9.2.5.3

String class

Two strings are equal if all the characters are equal and in the identical order

A string is simply an array of bytes:

- Each byte stores a value from 0 to 255

Any hash function must be a function of these bytes

String class

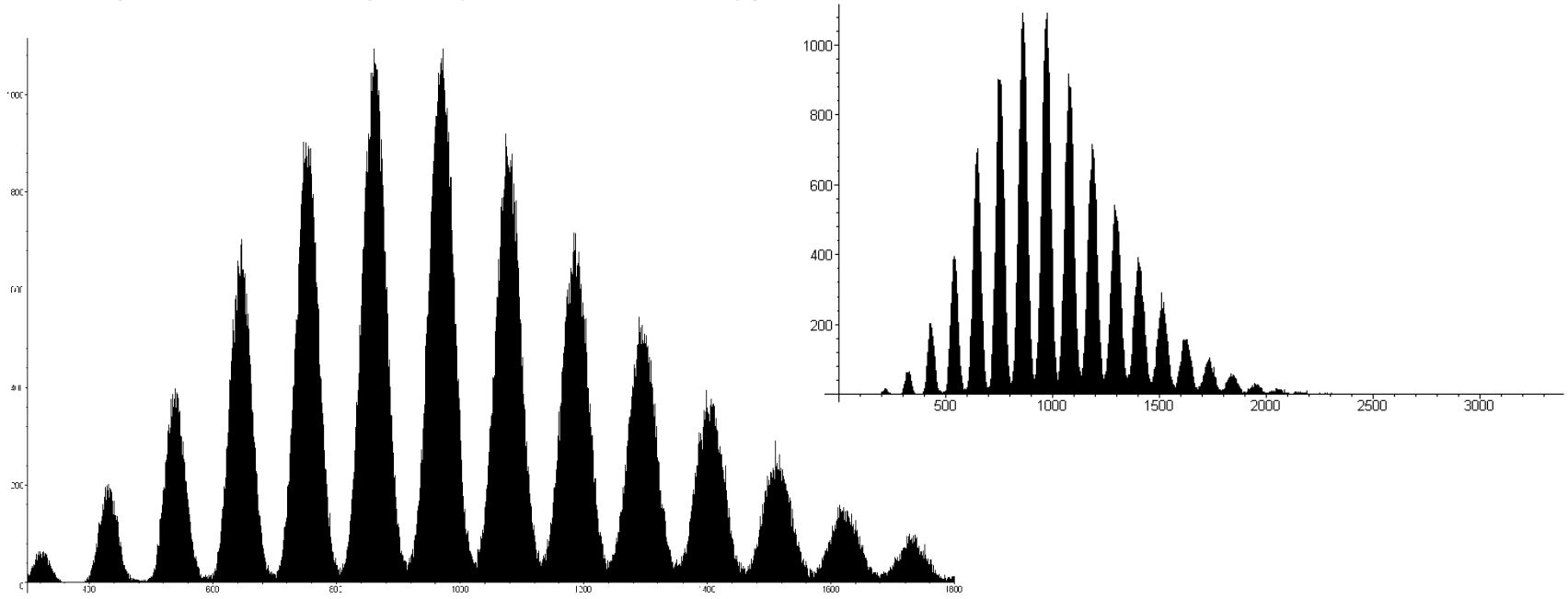
We could, for example, just add the characters:

```
unsigned int hash( const string &str ) {
    unsigned int hash_vaalue = 0;

    for ( int k = 0; k < str.length(); ++k ) {
        hash_value += str[k];
    }

    return hash_value;
}
```

String class

Not very good:

- Slow run time: $\Theta(n)$
- Words with the same characters hash to the same code:
  - `"form"` and `"from"`
- A poor distribution, *e.g.*, all words in Moby™ Words II by Grady Ward (`single.txt`) Project Gutenberg):



String class

Let the individual characters represent the coefficients of a polynomial in x:

$$p(x) = c_0 x^{n-1} + c_1 x^{n-2} + \cdots + c_{n-3} x^2 + c_{n-2} x + c_{n-1}$$

Use Horner's rule to evaluate this polynomial at a prime number, *e.g.*, $x = 12347$:

```
unsigned int hash( string const &str ) {
    unsigned int hash_value = 0;

    for ( int k = 0; k < str.length(); ++k ) {
        hash_value = 12347*hash_value + str[k];
    }

    return hash_value;
}
```

Horner's Rule

Polynomial evaluation and long division [edit]

Given the polynomial

$$p(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n,$$

where $a_0, \ldots, a_n$ are constant coefficients, the problem is to evaluate the polynomial at a specific value x_0 of x.

For this, a new sequence of constants is defined recursively as follows:

$$\begin{aligned} b_n &:= a_n \\ b_{n-1} &:= a_{n-1} + b_n x_0 \\ &\vdots \\ b_1 &:= a_1 + b_2 x_0 \\ b_0 &:= a_0 + b_1 x_0. \end{aligned} \qquad (1)$$

Then b_0 is the value of $p(x_0)$.

https://en.wikipedia.org/wiki/Horner%27s_method

9.2.5.3.3

Arithmetic hash functions

In general, any member variables that are used to uniquely define an object may be used as coefficients in such a polynomial

- The salary hopefully changes over time…

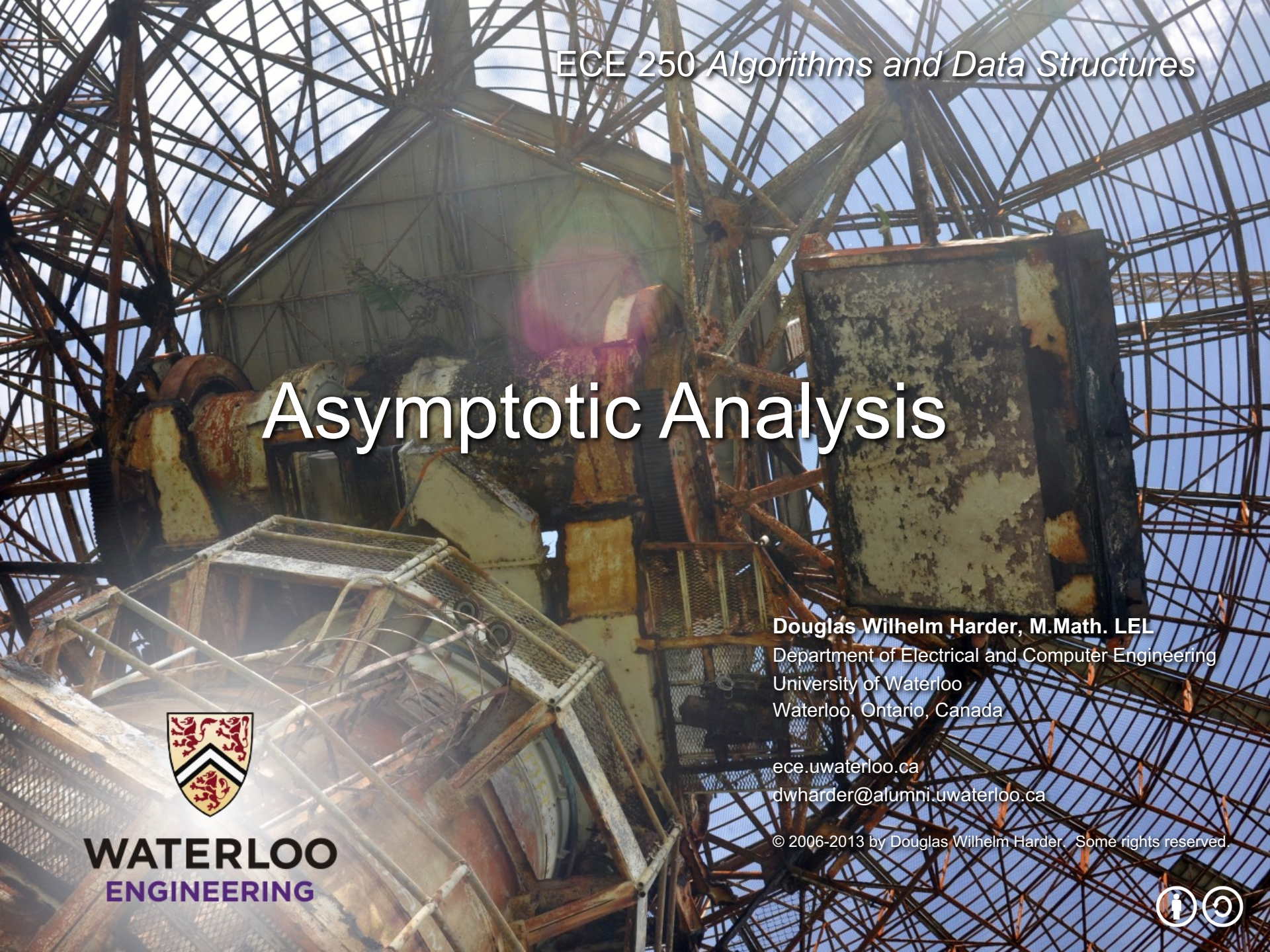
```
class Person {
    string surname;
    string *given_names;
    unsigned char num_given_names;
    unsigned short birth_year;
    unsigned char birth_month;
    unsigned char birth_day;
    unsigned int salary;
    // ...
};
```

Summary

We have seen how a number of objects can be mapped onto a 32-bit integer

We considered

- Predetermined hash functions
  - Auto-incremented variables
  - Addresses
- Hash functions calculated using arithmetic



ECE 250 *Algorithms and Data Structures*

Asymptotic Analysis

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ece.uwaterloo.ca
dwharder@alumni.uwaterloo.ca

WATERLOO
ENGINEERING

Background

Suppose we have two algorithms, how can we tell which is better?

We could implement both algorithms, run them both

- Expensive and error prone

Preferably, we should analyze them mathematically

- *Algorithm analysis*

2.3.1

Maximum Value

For example, the time taken to find the largest object in an array of n random integers will take n operations

```
int find_max( int *array, int n ) {
    int max = array[0];

    for ( int i = 1; i < n; ++i ) {
        if ( array[i] > max ) {
            max = array[i];
        }
    }

    return max;
}
```

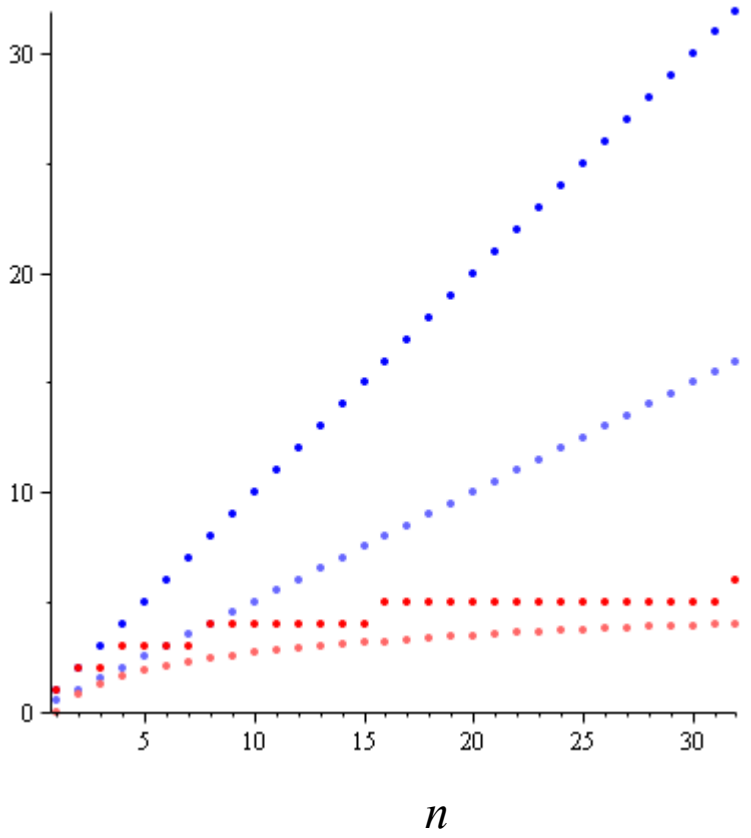
2.3.2

Linear and binary search

There are other algorithms which are significantly faster as the problem size increases

This plot shows maximum and average number of comparisons to find an entry in a sorted array of size n

- Linear search
- Binary search



2.3.3

Asymptotic Analysis

Given an algorithm:

- We need to be able to describe these values mathematically
- We need a systematic means of using the description of the algorithm together with the properties of an associated data structure
- We need to do this in a machine-independent way

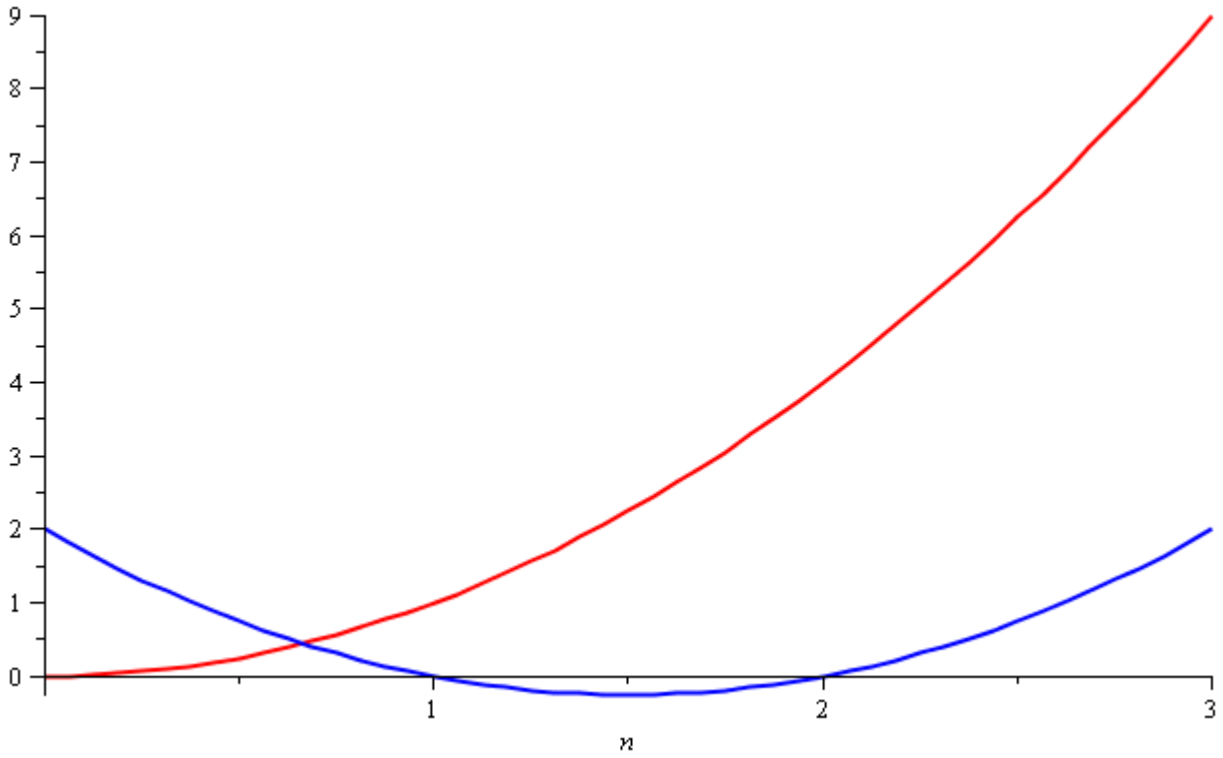
For this, we need Landau symbols and the associated asymptotic analysis

Quadratic Growth

Consider the two functions

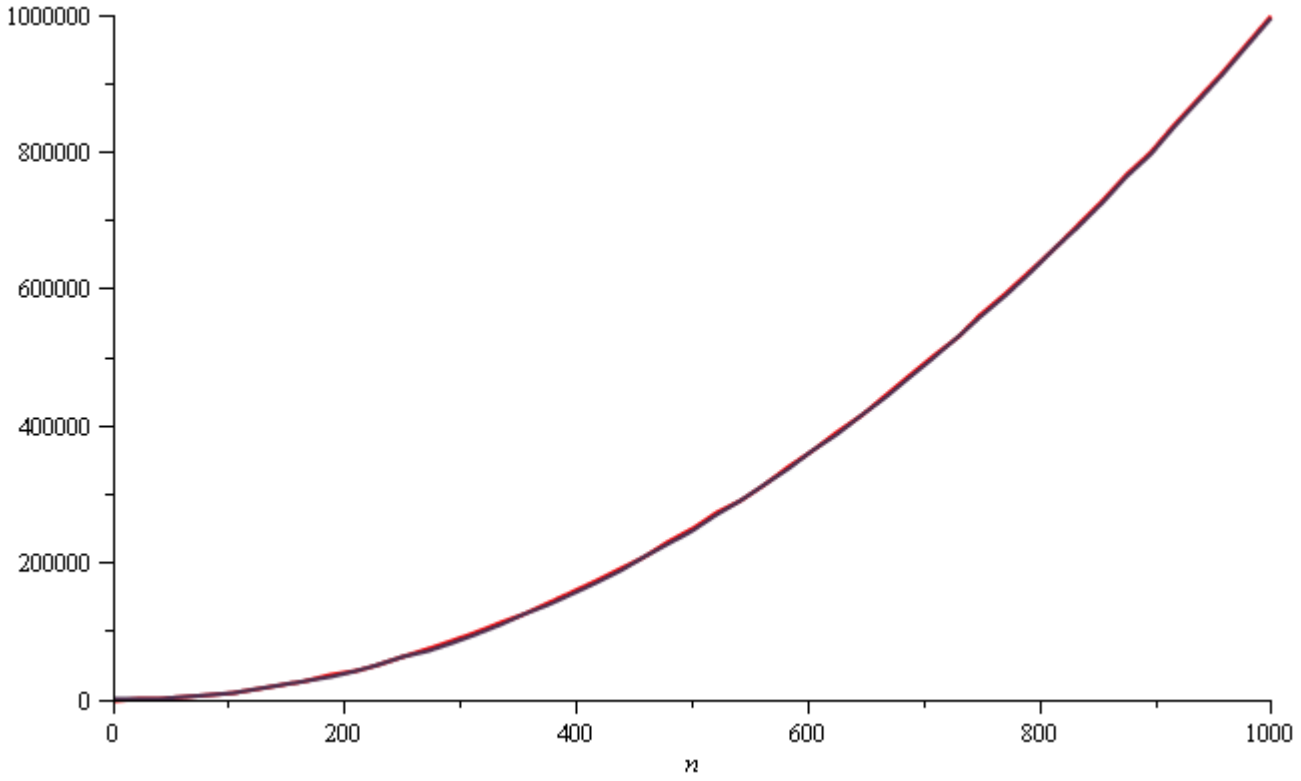
$f(n) = n^2$ and $g(n) = n^2 - 3n + 2$

Around $n = 0$, they look very different



Quadratic Growth

Yet on the range $n = [0, 1000]$, they are (relatively) indistinguishable:



Quadratic Growth

The absolute difference is large, for example,

$$f(1000) = 1\ 000\ 000$$

$$g(1000) = \ \ 997\ 002$$

but the relative difference is very small

$$\left|\frac{f(1000) - g(1000)}{f(1000)}\right| = 0.002998 < 0.3\%$$

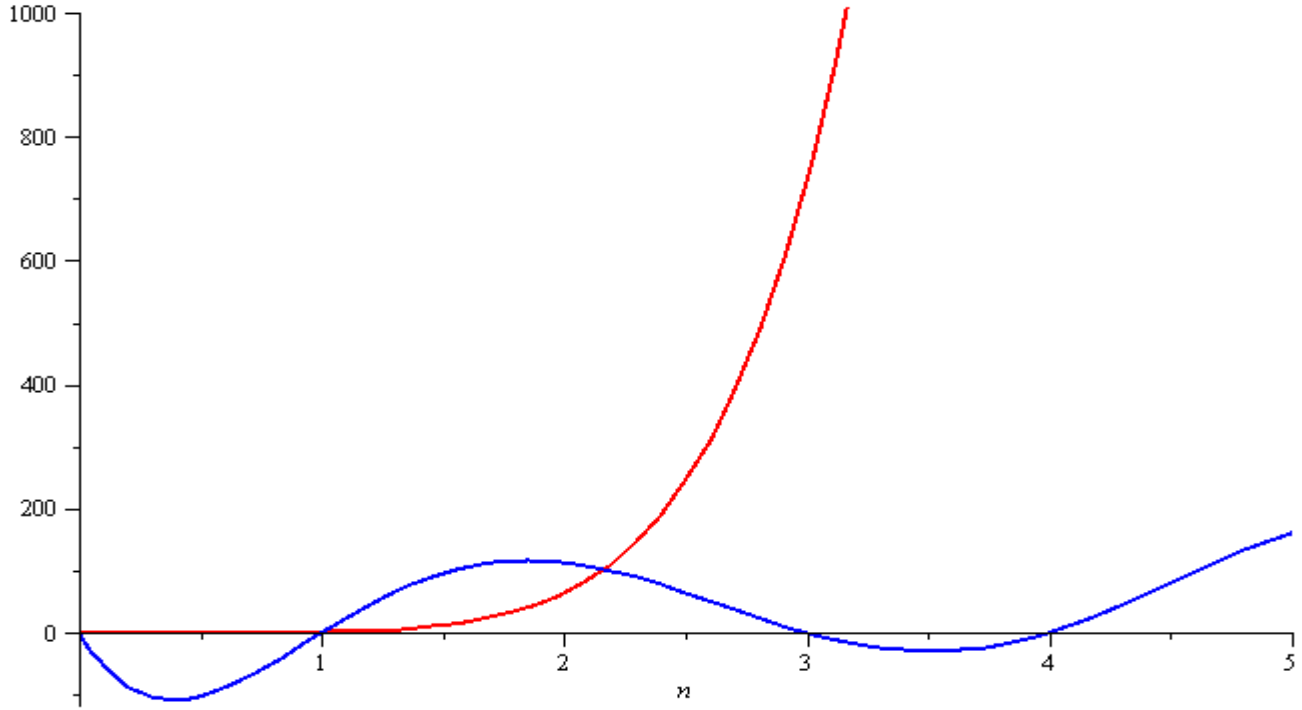
and this difference goes to zero as $n \to \infty$

Polynomial Growth

To demonstrate with another example,

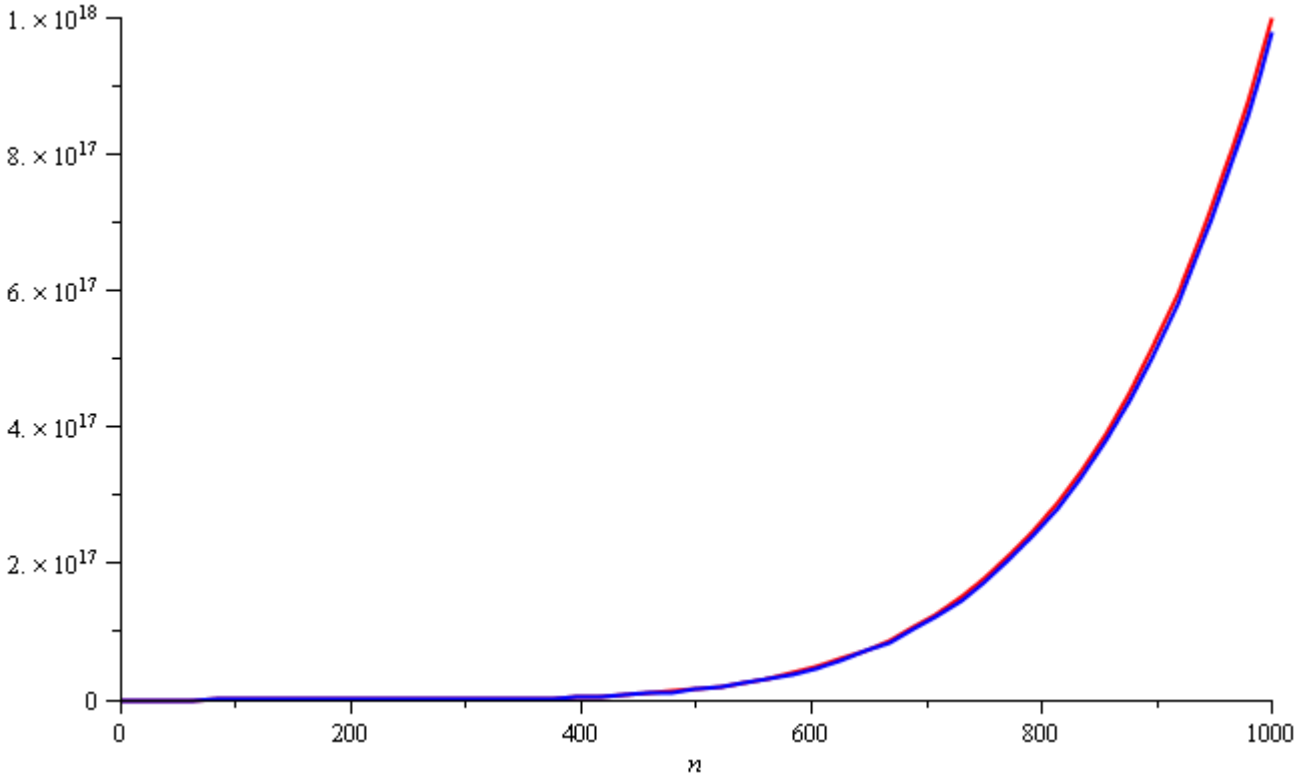
$f(n) = n^6$ and $g(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$

Around $n = 0$, they are very different



Polynomial Growth

Still, around $n = 1000$, the relative difference is less than 3%



2.3.3

Polynomial Growth

The justification for both pairs of polynomials being similar is that, in both cases, they each had the same leading term:

n^2 in the first case, n^6 in the second

Suppose however, that the coefficients of the leading terms were different

– In this case, both functions would exhibit the same rate of growth, however, one would always be proportionally larger

Counting Instructions

Suppose we had two algorithms which sorted a list of size n and the run time (in μs) is given by

$$b_{\text{worst}}(n) = 4.7n^2 - 0.5n + 5 \quad \text{Bubble sort}$$

$$b_{\text{best}}(n) = 3.8n^2 + 0.5n + 5$$

$$s(n) = 4n^2 + 14n + 12 \quad \text{Selection sort}$$

The smaller the value, the fewer instructions are run

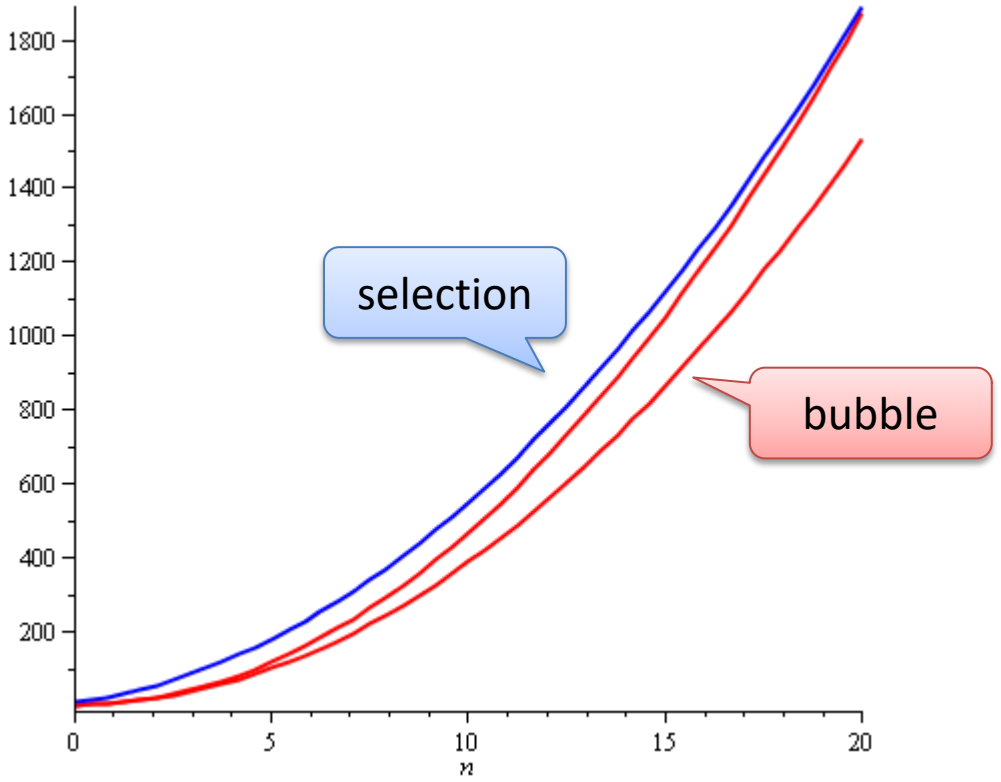
- For $n \leq 21$, $b_{\text{worst}}(n) < s(n)$
- For $n \geq 22$, $b_{\text{worst}}(n) > s(n)$

https://en.wikipedia.org/wiki/Selection_sort
https://en.wikipedia.org/wiki/Bubble_sort

2.3.4.1

Counting Instructions

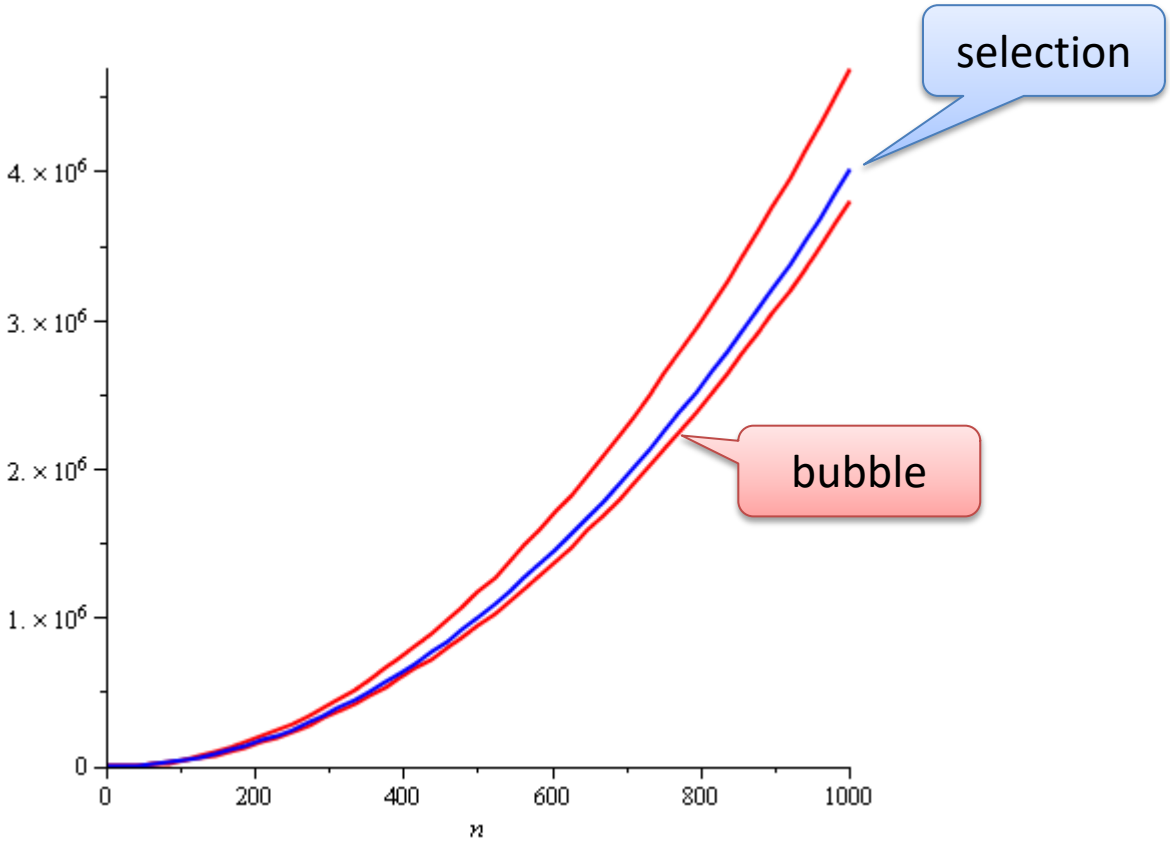
With small values of n, the algorithm described by $s(n)$ requires more instructions than even the worst-case for bubble sort



2.3.4.1

Counting Instructions

Near $n = 1000$, $b_{\text{worst}}(n) \approx 1.175\, s(n)$ and $b_{\text{best}}(n) \approx 0.95\, s(n)$



2.3.4.1

Counting Instructions

Is this a serious difference between these two algorithms?

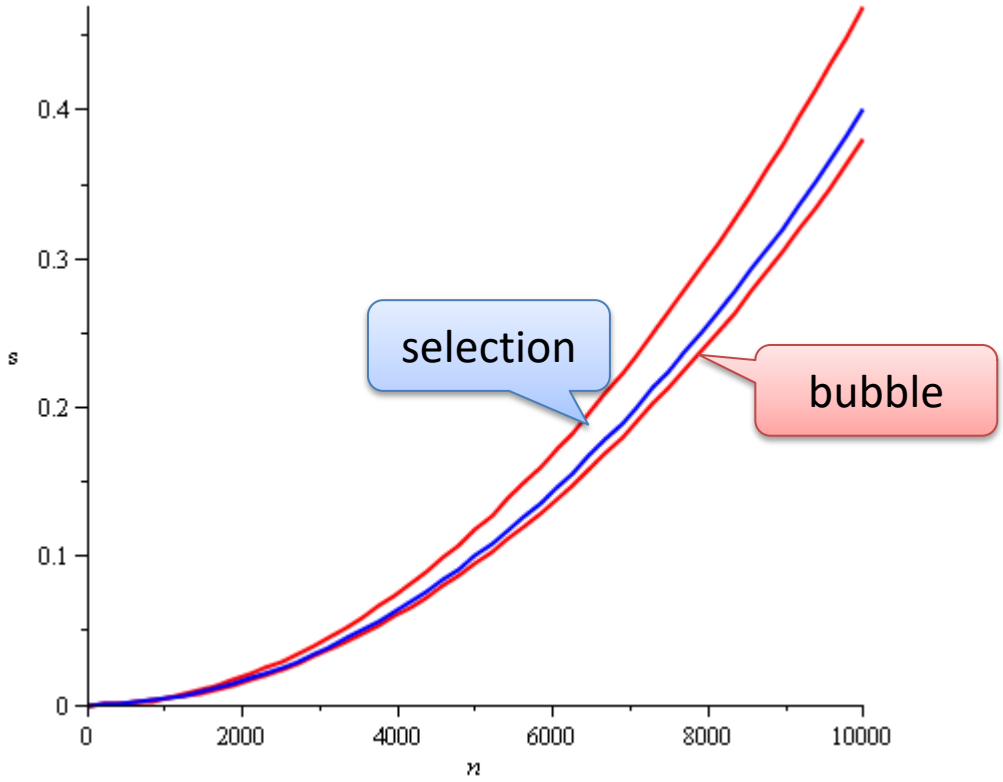
Because we can count the number instructions, we can also estimate how much time is required to run one of these algorithms on a computer

2.3.4.1

Counting Instructions

Suppose we have a $1\ \mathrm{GHz}$ computer

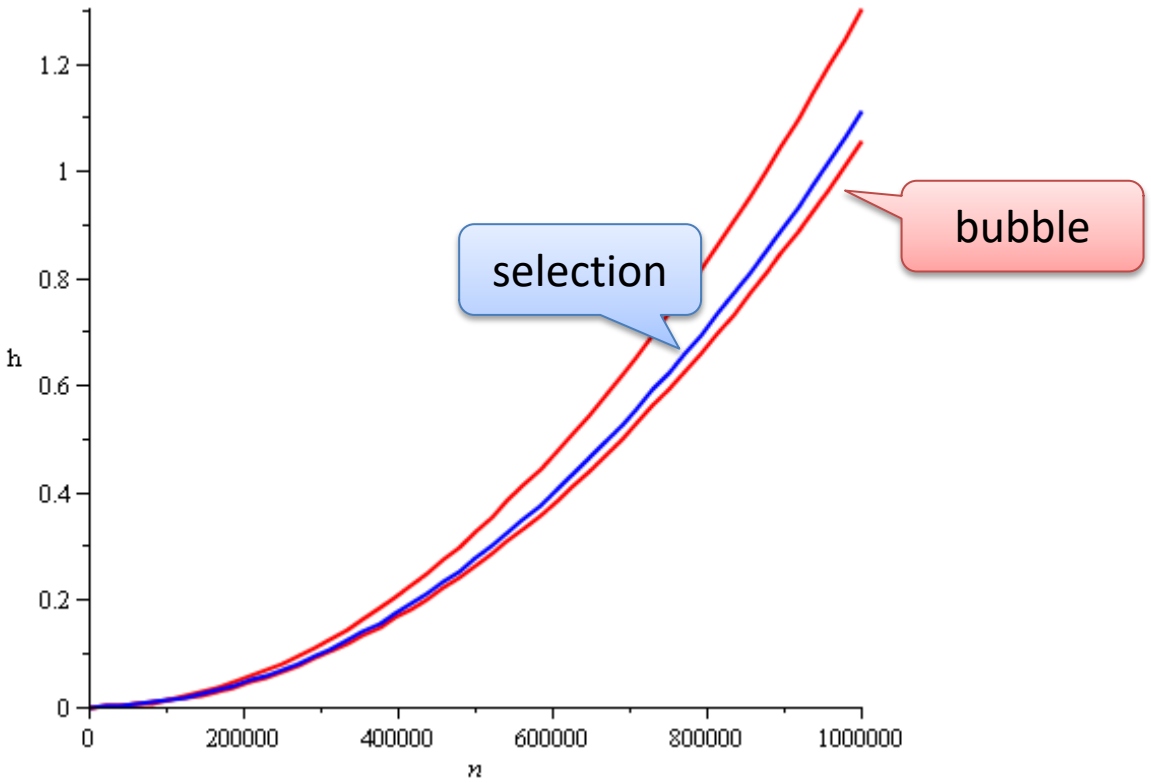
– The time (s) required to sort a list of up to $n = 10\,000$ objects is under half a second



2.3.4.1

Counting Instructions

To sort a list with one million elements, it will take about 1 h

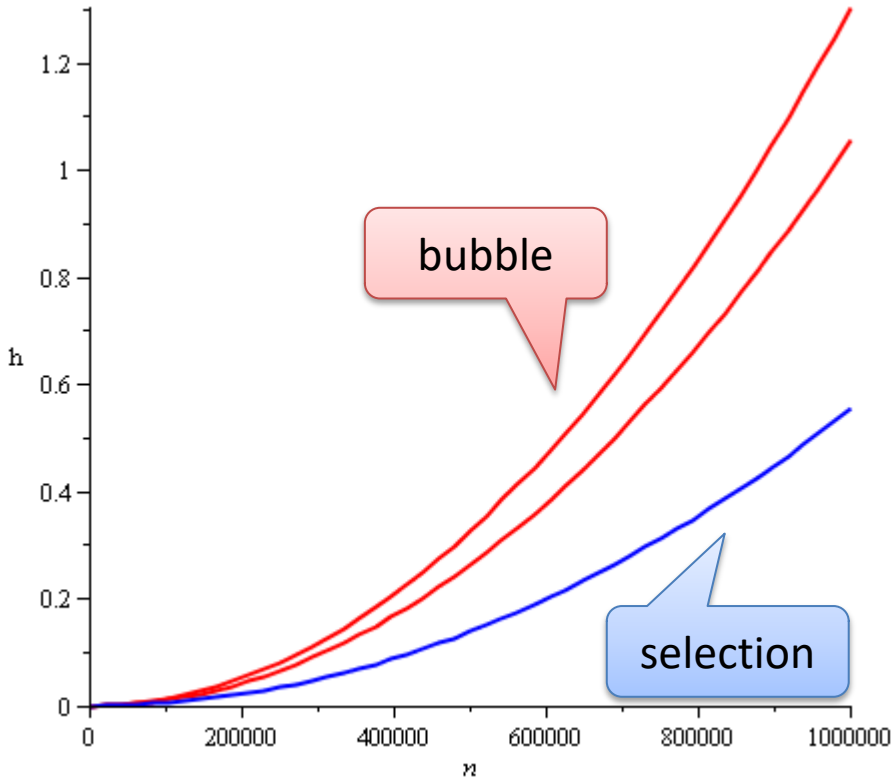


Bubble sort could, under some conditions, be 200 s faster

Counting Instructions

How about running selection sort on a faster computer?

– For large values of n, selection sort on a faster computer will always be faster than bubble sort



2.3.4.1

Counting Instructions

Justification?

- If $f(n) = a_k n^k + \cdots$ and $g(n) = b_k n^k + \cdots$,
  for large enough n, it will always be true that

$$f(n) < Mg(n)$$

  where we choose

$$M = a_k/b_k + 1$$

In this case, we only need a computer which is M times faster (or slower)

Question:

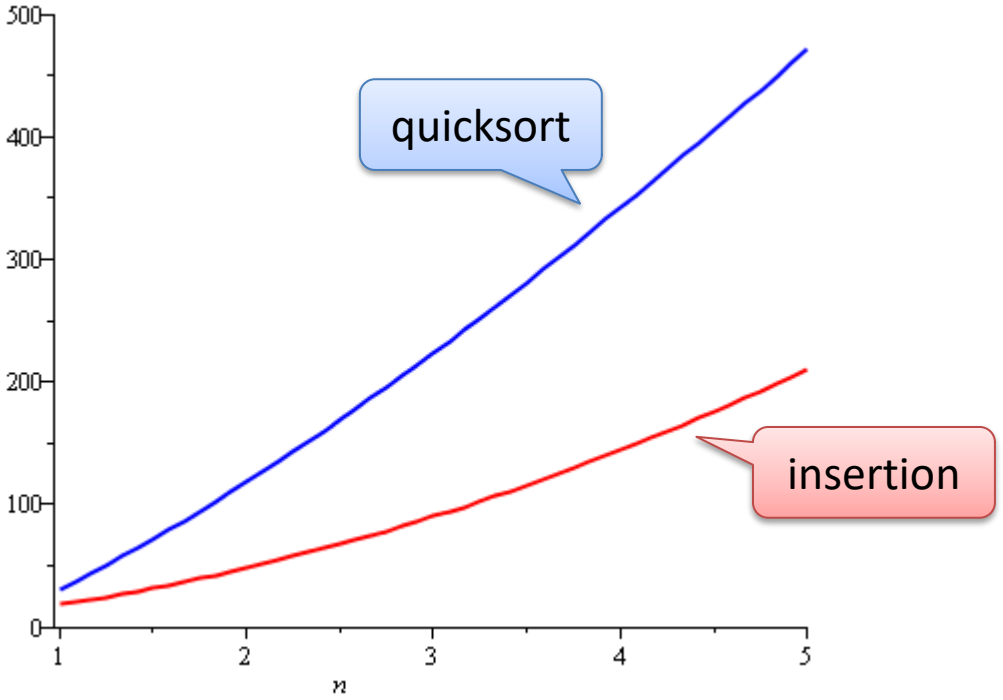
- Is a linear search comparable to a binary search?
- Can we just run a linear search on a slower computer?

2.3.4.2

Counting Instructions

As another example:

- Compare the number of instructions required for insertion sort and for quicksort
- Both functions are concave up, although one more than the other

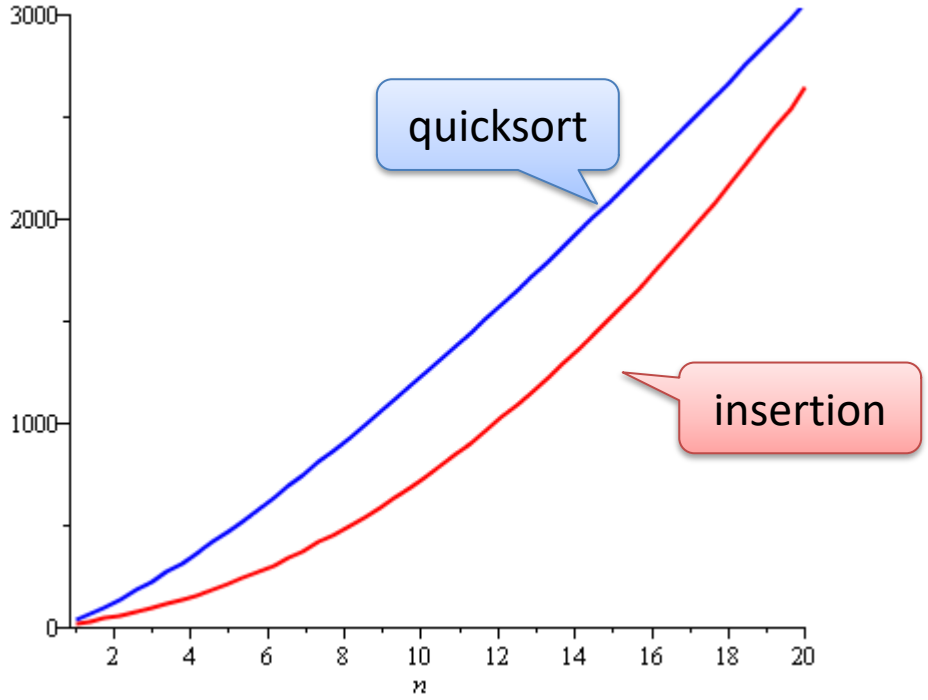


2.3.4.2

Counting Instructions

Insertion sort, however, is growing at a rate of n^2 while quicksort grows at a rate of $n \lg(n)$

- Never-the-less, the graphic suggests it is more useful to use insertion sort when sorting small lists—quicksort has a large overhead

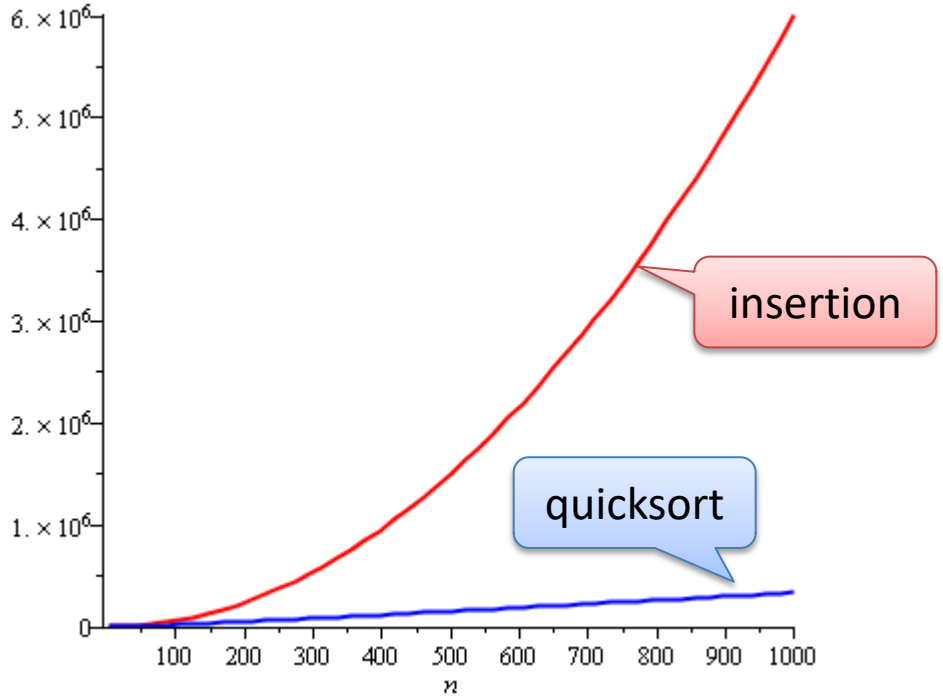


2.3.4.2

Counting Instructions

If the size of the list is too large (greater than 20), the additional overhead of quicksort quickly becomes insignificant

- The quicksort algorithm becomes significantly more efficient
- Question: can we just buy a faster computer?



Weak ordering

Consider the following definitions:

- We will consider two functions to be equivalent, $f \sim g$, if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \text{ where } 0 < c < \infty$$

- We will state that $f < g$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

For functions we are interested in, these define a weak ordering

Weak ordering

Let $f(n)$ and $g(n)$ describe either the run-time of two algorithms

- If $f(n) \sim g(n)$, then it is always possible to improve the performance of one function over the other by purchasing a faster computer
- If $f(n) < g(n)$, then you can never purchase a computer fast enough so that the second function always runs in less time than the first

Note that for small values of *n*, it may be reasonable to use an algorithm that is asymptotically more expensive, but these have to be considered on a one-by-one basis

2.3.5

Landau Symbols

A function $\mathrm{f}(n) = \mathbf{O}(\mathrm{g}(n))$ if there exists N and c such that

$$\mathrm{f}(n) < c\ \mathrm{g}(n)$$

whenever $n > N$

- The function $\mathrm{f}(n)$ has a rate of growth no greater than that of $\mathrm{g}(n)$

Landau Symbols

Before we begin, however, we will make some assumptions:

- Our functions will describe the time or memory required to solve a problem of size n
- Therefore, we are restricting ourselves to certain functions:
  - They are defined for $n \geq 0$
  - They are strictly positive for all n
    - In fact, $f(n) > c$ for some value $c > 0$
    - That is, any problem requires at least one instruction and byte
  - They are increasing (monotonic increasing)

Landau Symbols

Another Landau symbol is Θ

A function $f(n) = \Theta(g(n))$ if there exist positive N, c_1, and c_2 such that

$$c_1\, g(n) < f(n) < c_2\, g(n)$$

whenever $n > N$

- The function $f(n)$ has a rate of growth equal to that of $g(n)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \quad \text{where} \quad 0 < c < \infty$$

Landau Symbols

We have a similar definition for $\mathbf{O}$:

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where $0 \le c < \infty$, it follows that $f(n) = \mathbf{O}(g(n))$

There are other possibilities we would like to describe:

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, we will say $f(n) = \mathbf{o}(g(n))$

- The function $f(n)$ has a rate of growth less than that of $g(n)$

We would also like to describe the opposite cases:

- The function $f(n)$ has a rate of growth greater than that of $g(n)$
- The function $f(n)$ has a rate of growth greater than or equal to that of $g(n)$

Landau Symbols

We will at times use five possible descriptions

$$\mathrm{f}(n) = \mathbf{o}(\mathrm{g}(n)) \qquad \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} = 0$$

$$\mathrm{f}(n) = \mathbf{O}(\mathrm{g}(n)) \qquad \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} < \infty$$

$$\mathrm{f}(n) = \mathbf{\Theta}(\mathrm{g}(n)) \qquad 0 < \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} < \infty$$

$$\mathrm{f}(n) = \mathbf{\Omega}(\mathrm{g}(n)) \qquad \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} > 0$$

$$\mathrm{f}(n) = \boldsymbol{\omega}(\mathrm{g}(n)) \qquad \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} = \infty$$

2.3.7

Landau Symbols

For the functions we are interested in, it can be said that

$f(n) = \mathbf{O}(g(n))$ is equivalent to $f(n) = \mathbf{\Theta}(g(n))$ or $f(n) = \mathbf{o}(g(n))$

and

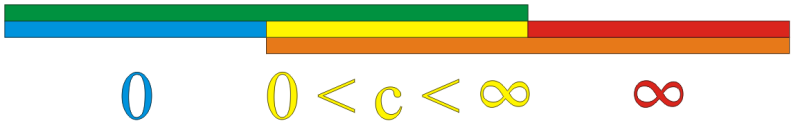
$f(n) = \mathbf{\Omega}(g(n))$ is equivalent to $f(n) = \mathbf{\Theta}(g(n))$ or $f(n) = \boldsymbol{\omega}(g(n))$

Landau Symbols

Graphically, we can summarize these as follows:

We say $f(n) =$ $\mathbf{O}(g(n))$ $\Omega(g(n))$ $\mathbf{o}(g(n))$ $\Theta(g(n))$ $\omega(g(n))$

if $\lim_{n \to \infty} \frac{f(n)}{g(n)} =$



2.3.8

Big-Θ as an Equivalence Relation

The most common classes are given names:

$\Theta(1)$	constant
$\Theta(\ln(n))$	logarithmic
$\Theta(n)$	linear
$\Theta(n \ln(n))$	"n log n"
$\Theta(n^2)$	quadratic
$\Theta(n^3)$	cubic
2^n, e^n, 4^n, ...	exponential

2.3.8

Logarithms and Exponentials

Recall that all logarithms are scalar multiples of each other

- Therefore $\log_b(n) = \Theta(\ln(n))$ for any base b

Alternatively, there is no single equivalence class for exponential functions:

- If $1 < a < b$, $\lim_{n\to\infty} \frac{a^n}{b^n} = \lim_{n\to\infty} \left(\frac{a}{b}\right)^n = 0$
- Therefore $a^n = \mathbf{o}(b^n)$

However, we will see that it is almost universally undesirable to have an exponentially growing function!

2.3.8

Logarithms and Exponentials

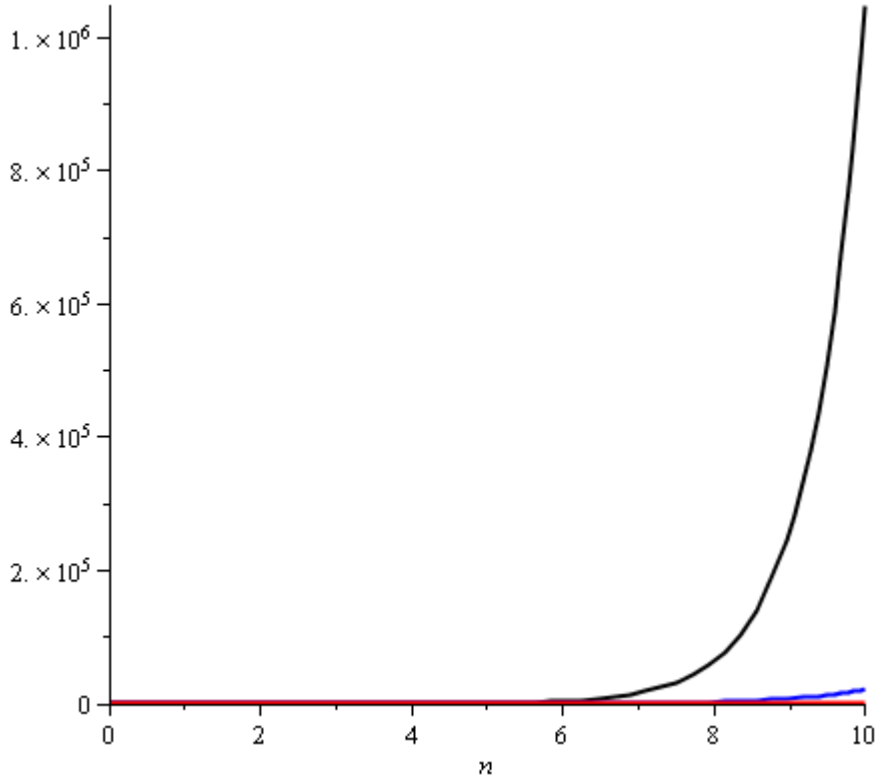
Plotting 2^n, e^n, and 4^n on the range $[1, 10]$ already shows how significantly different the functions grow

Note:

$2^{10} = 1024$

$e^{10} \approx 22\,026$

$4^{10} = 1\,048\,576$



Algorithms Analysis

We will use Landau symbols to describe the complexity of algorithms

- E.g., adding a list of n doubles will be said to be a $\Theta(n)$ algorithm

An algorithm is said to have *polynomial time complexity* if its run-time may be described by $\mathrm{O}(n^d)$ for some fixed $d \geq 0$

- We will consider such algorithms to be *efficient*

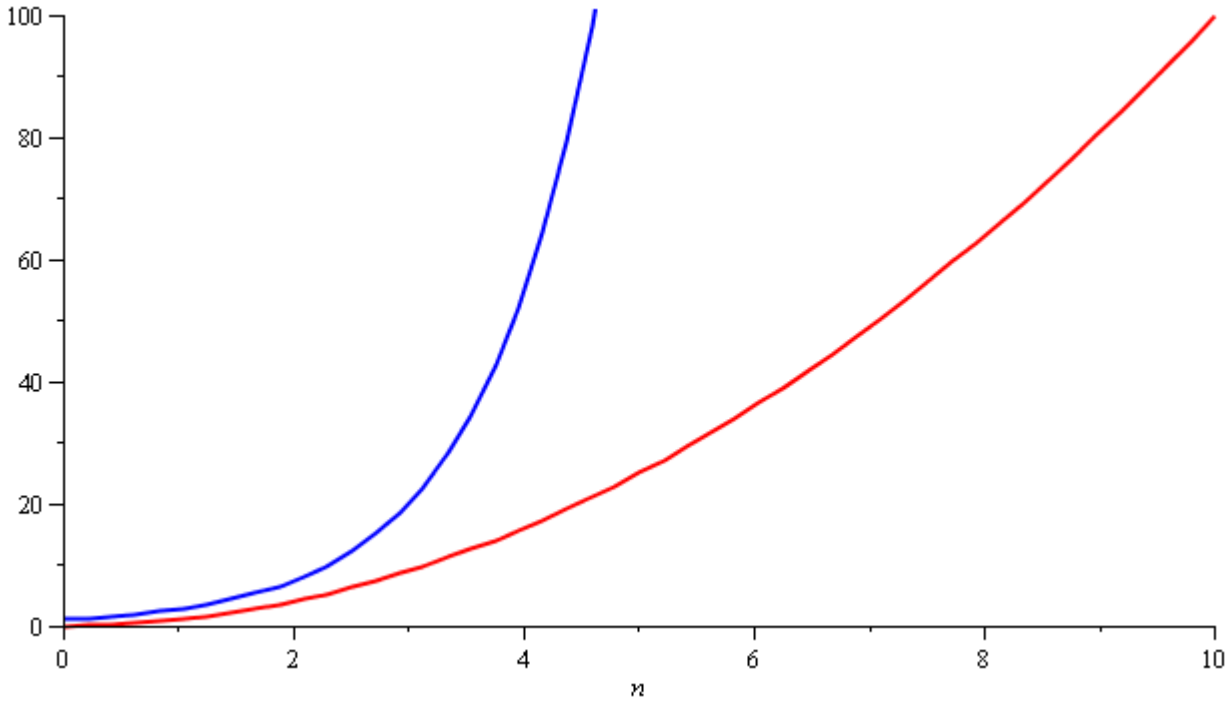
Problems that have no known polynomial-time algorithms are said to be *intractable*

- Traveling salesman problem: find the shortest path that visits n cities
- Best run time: $\Theta(n^2\, 2^n)$

Algorithm Analysis

In general, you don't want to implement exponential-time or exponential-memory algorithms

– Warning: don't call a quadratic curve "exponential", either...please



Summary

In this class, we have:

- Reviewed Landau symbols, introducing some new ones: o O Θ Ω ω
- Discussed how to use these
- Looked at the equivalence relations

References

Wikipedia, https://en.wikipedia.org/wiki/Mathematical_induction