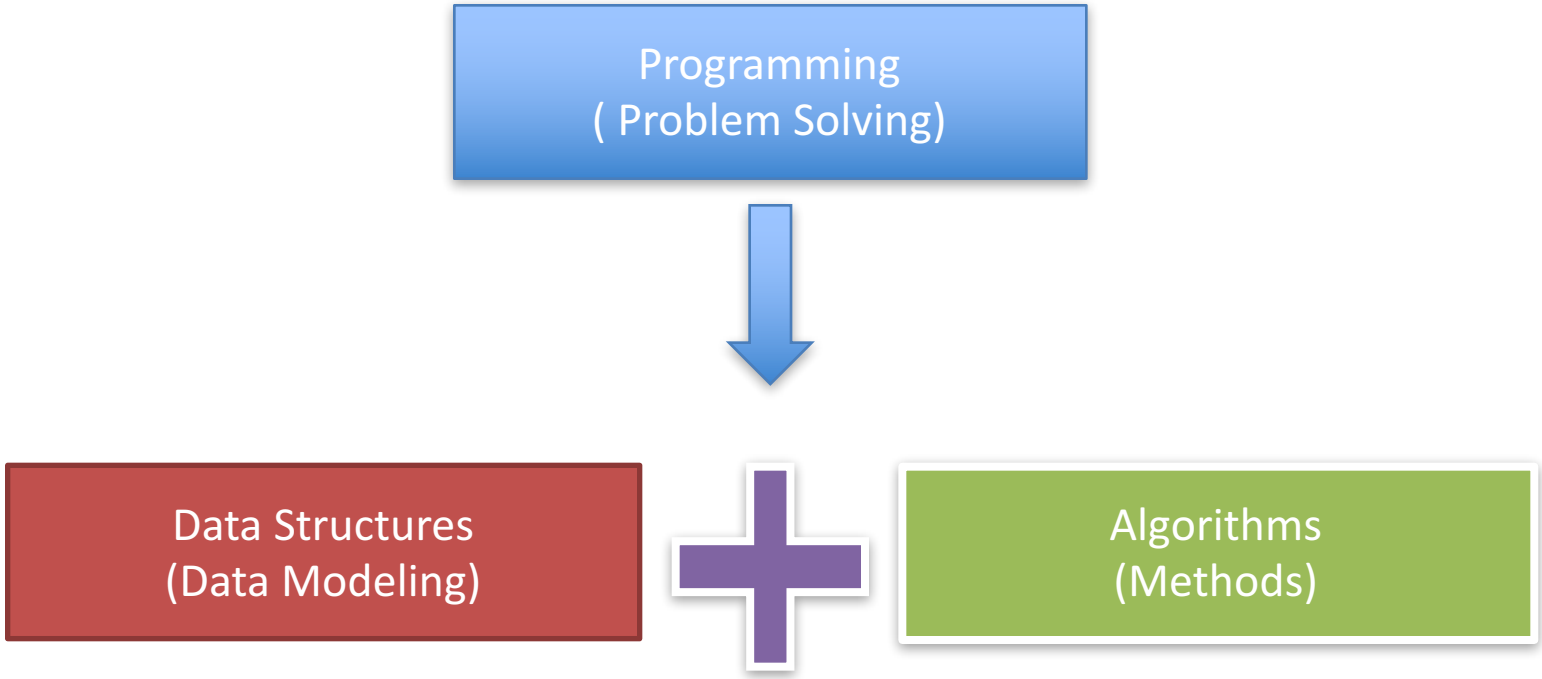
Welcome to Data Structures and Algorithms

Prof. Yi Fang
Dept. Electrical and Computer Engineering

Office: C1-156
Email: yfang@nyu.edu
Phone: 026284891

TA: Jing Zhu (jingzhu@nyu.edu)

Acknowledge Prof. Harder from University of Waterloo for the permission of using his course materials in algorithms and data structures.


Programming
(Problem Solving)
Data Structures
(Data Modeling)
Algorithms
(Methods)

Example

River-crossing puzzles

Solution:

1. https://en.wikipedia.org/wiki/River_crossing_puzzle
2. https://mark-borg.github.io/blog/2016/river-crossing-puzzles/

This example help us understand, to solve a real problem by computer, we need to select a data structure (here is graph) and a algorithm (here is dijkstra algorithm) . We will explain the details of the graph structure and dijkstra (shortest path) algorithm in this course)

Introduction and overview

ENGR-AD-202 Computer System Programming

AD-202 covered:

- The history of computing / objects / types / console I/O
- Operators / loops / methods / parameter passing
- Selection statements / arrays / strings
- Pointers / unsafe code / linked lists
- Collections / multi-dimensional arrays / search algorithms
- Sorting algorithms
- Object-oriented design / polymorphism / interfaces / inheritance

AD202 Computer System Programming

Basics of programming

- The ability to manipulate the computer to perform the required tasks

You saw data storage techniques:

- Arrays

You saw array accessing/manipulation techniques:

- Searching
- Sorting

UH-3510 Algorithms and Data Structures

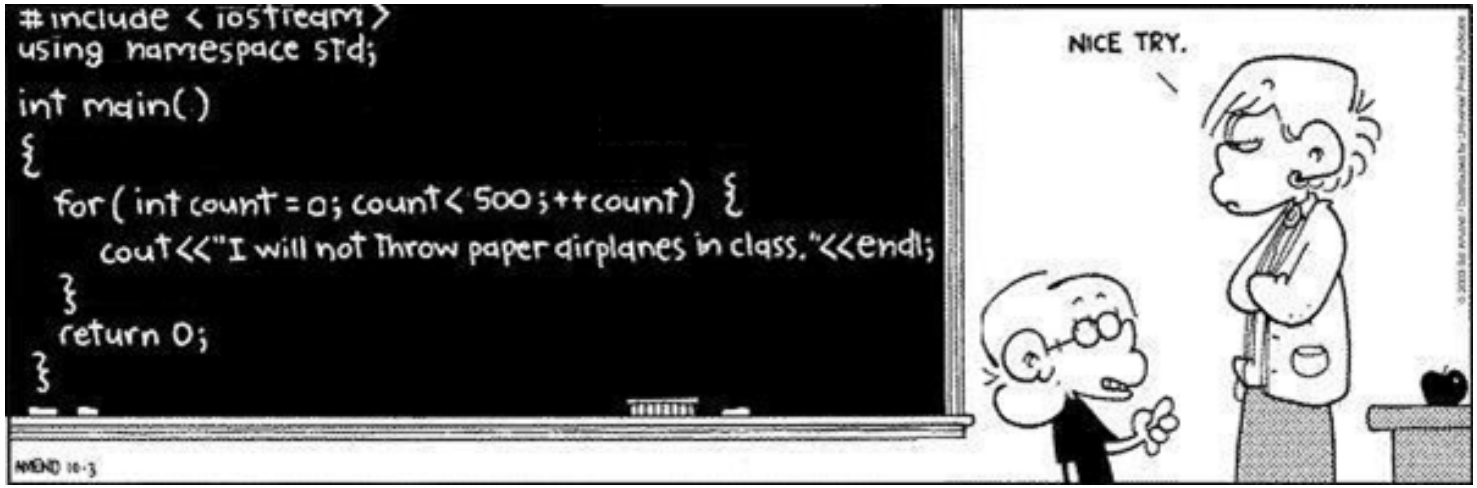
In this course, we will look at:

Algorithms for solving problems efficiently *Data structures* for efficiently storing, accessing, and modifying data

We will see that all data structures have trade-offs, there is no *ultimate* data structure. There is no unique algorithm solution for problems. The choice depends on our requirements

C/C++

You will be using the C/C++ programming language in this course



Modified for C++ from http://www.foxtrot.com/

C/C++

This course does not teach C++ programming

- You will use C/C++ to demonstrate your knowledge in this course

C/C++

Other sources of help in C++ are:

- The Project T.A.s,
- The lab instructor, and
- Other online tutorials: `http://www.cplusplus.com/`

Labs and projects are held every second week

- All laboratory material the course web site
- Laboratories is also associated with the project
- Bonus projects allows you to practice more engineering problems.

Course Evaluation

The course is divided into numerous topics

- Storing ordered and sorted objects
- Storing an arbitrary collection of data
- Sorting objects
- Graphs
- Algorithm Design Techniques

Evaluation

Your evaluation in this course is based on three components:

- Six equally weighted projects (5% each)
- Three mid-term examination (15% each)
- One final examination (25%)
- Bonus projects and exam problems

Evaluation

Commenting code is necessary for engineers:

- Engineers who do not comment code will not encourage employees and contracted programmers to comment their code. This will lead to significant additional costs.
- All of project code required well-documented comments to earn full credits.

Projects & Labs

For each of course labs, you will practice the data structures taught in class. You will be given two to three problems each week.

For each of the six projects, you will be required to implement one or more of the data structures taught in class.

You do not need to submit the course lab assignment. You will be required to submit the project by due time.

Container and Abstract Data Types

Outline

Any form of information processing or communication requires that data must be stored in and accessed from either main or secondary memory

- There are two questions we should ask:
  - What do we want to do?
  - How can we do it?
- Abstract Data Types:

  Models of the storage and access of information
- Data structures and algorithms:

  The concrete methods for organizing and accessing data in the computer

Containers

- The most general Abstract Data Type (ADT) is that of a *container*
  - The Container ADT
- A container describes structures that store and give access to objects
- The queries and operations of interest may be defined on:
  - The container as an entity, or
  - The objects stored within a container

Operations on a Container

The operations we may wish to perform on a container are:

- Create a new container
- Copy or destroy an existing container
- Empty a container
- Query how many objects are in a container
- Query what is the maximum number of objects a container can hold
- Given two containers:
  - Find the union (merge), or
  - Find the intersection

Operations on a Container

Many of these operations on containers are in the Standard Template Library

Constructor	`Container()`
Copy Constructor	`Container( Container const & )`
Destructor	`~Container()`
Empty it	`void clear()`
How many objects are in it?	`int size() const`
Is it empty?	`bool empty() const`
How many objects can it hold?	`int max_size() const`
Merge with another container	`void insert( Container const & )`

Operations on Objects Stored in a Container

Given a container, we may wish to:

- Insert an object into a container
- Access or modify an object in a container
- Remove an object from the container
- Query if an object is in the container
  - If applicable, count how many copies of an object are in a container
- Iterate (step through) the objects in a container

Operations on Objects Stored in a Container

Many of these operations are also common to the Standard Template Library

Insert an object	`void insert( Type const & )`
Erase an object	`void erase( Type const & )`
Find or access an object	`iterator find( Type const & )`
Count the number of copies	`int count( Type const & )`
Iterate through the objects in a container	`iterator begin() const`

Data Storage

Outline

This topic will describe:

- The concrete data structures that can be used to store information
- The basic forms of memory allocation
  - Contiguous
  - Linked
  - Indexed
- The prototypical examples of these: arrays and linked lists
- Other data structures:
  - Trees
  - Hybrids
  - Higher-dimensional arrays
- Finally, we will discuss the run-time of queries and operations on arrays and linked lists

Memory Allocation

Memory allocation can be classified as either

- Contiguous
- Linked
- Indexed

Prototypical examples:

- Contiguous allocation: arrays
- Linked allocation: linked lists

Memory Allocation

Contiguous, *adj.*

Touching or connected throughout in an unbroken sequence.

Meriam Webster

Touching, in actual contact, next in space; meeting at a common boundary, bordering, adjoining.

www.oed.com

Contiguous Allocation

An array stores n objects in a single contiguous space of memory

Unfortunately, if more memory is required, a request for new memory usually requires copying all information into the new memory

- In general, you cannot request for the operating system to allocate to you the next n memory locations

Linked Allocation

Linked storage such as a linked list associates two pieces of data with each item being stored:

- The object itself, and
- A reference to the next item
  - In C++ that reference is the address of the next node

Linked Allocation

This is a class describing such a node

```
template <typename Type>
class Node {
    private:
        Type element;
        Node *next_node;
    public:
            // ...
};
```

Linked Allocation

The operations on this node must include:

- Constructing a new node
- Accessing (retrieving) the value
- Accessing the next pointer

```
Node( const Type& = Type(), Node* = nullptr );
Type retrieve() const;
Node *next() const;
```

Pointing to nothing has been represented as:

C	`NULL`
Java/C#	`null`
C++ (old)	`0`
C++ (new)	`nullptr`
Symbolically	Ø

Linked Allocation

For a linked list, however, we also require an object which links to the first object

The actual linked list class must store two pointers

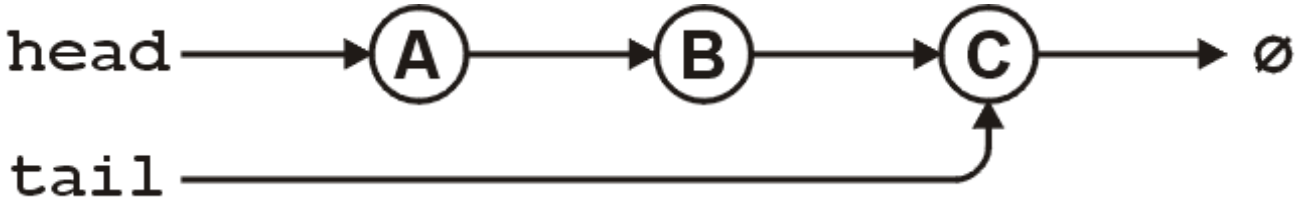
- A head and tail:

```
Node *head;
Node *tail;
```

Optionally, we can also keep a count

```
int count;
```

The `next_node` of the last node is assigned `nullptr`



Linked Allocation

The class structure would be:

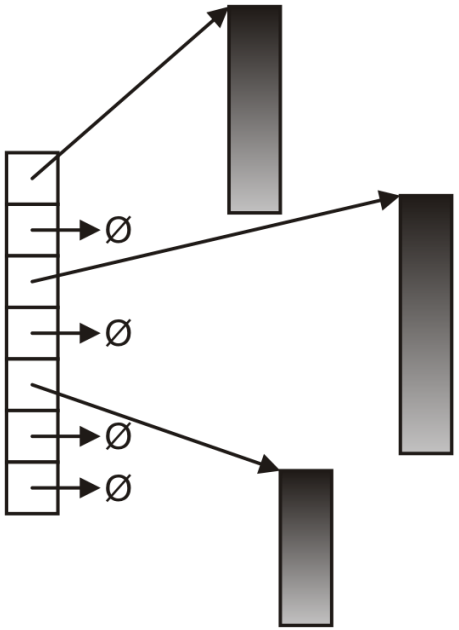
```
template <typename Type>
class List {
    private:
        Node<Type> *head;
        Node<Type> *tail;
        int count;
    public:
        // constructor(s)...
        // accessor(s)...
        // mutator(s)...
};
```

Indexed Allocation

With indexed allocation, an array of pointers (possibly NULL) link to a sequence of allocated memory locations

Used in the C++ standard template library

Computer engineering students will see indexed allocation in their operating systems course

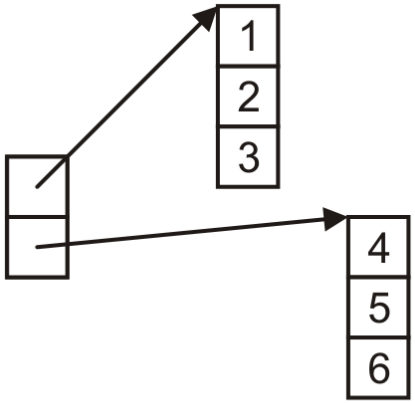


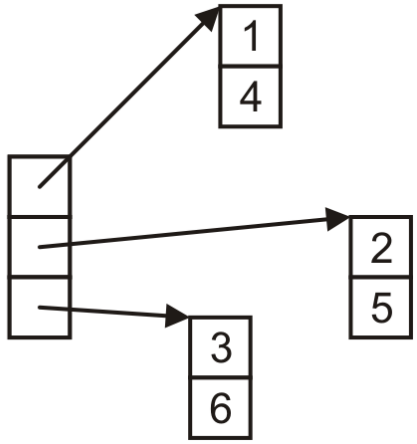
Indexed Allocation

Matrices can be implemented using indexed allocation:

1 2 3

4 5 6





2.2.1.3

Indexed Allocation

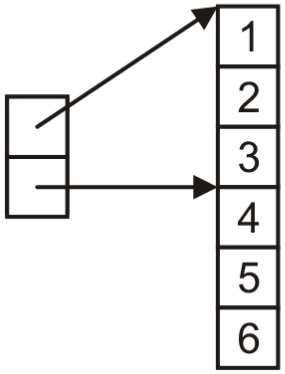
Matrices can be implemented using indexed allocation

- Most implementations of matrices (or higher-dimensional arrays) use indices pointing into a single contiguous block of memory

1 2 3

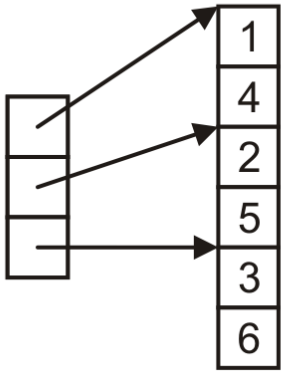
4 5 6

Row-major order



C, Python

Column-major order



Matlab, Fortran

Other Allocation Formats

We will look at some variations or hybrids of these memory allocations including:

- Trees
- Graphs
- Deques (linked arrays)
- inodes

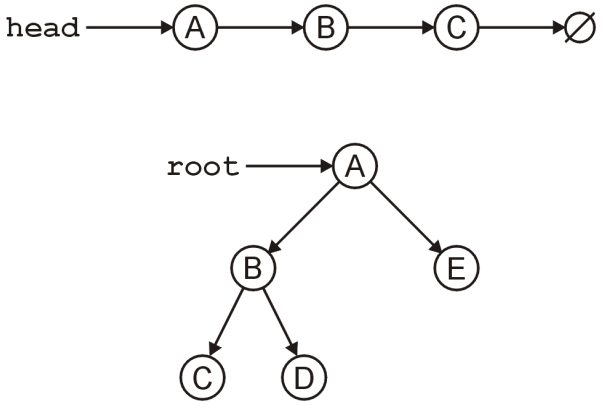
2.2.2.2

Trees

The linked list can be used to store linearly ordered data

- What if we have multiple *next* pointers?

A rooted tree (weeks 4-6) is similar to a linked list but with multiple next pointers



Trees

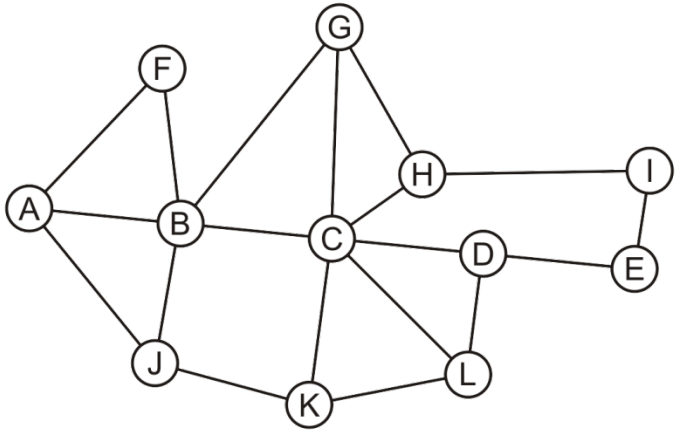
A tree is a variation on a linked list:

- Each node points to an arbitrary number of subsequent nodes
- Useful for storing hierarchical data
- We will see that it is also useful for storing sorted data
- Usually we will restrict ourselves to trees where each node points to at most two other nodes

Graphs

Suppose we allow arbitrary relations between any two objects in a container

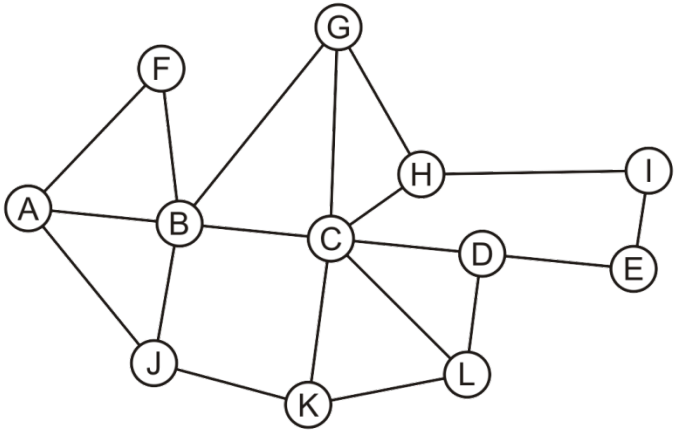
- Given n objects, there are $n^2 - n$ possible relations
  - If we allow symmetry, this reduces to $\frac{n^2 \quad n}{2}$
- For example, consider the network



Arrays

Suppose we allow arbitrary relations between any two objects in a container

- We could represent this using a two-dimensional array
- In this case, the matrix is *symmetric*

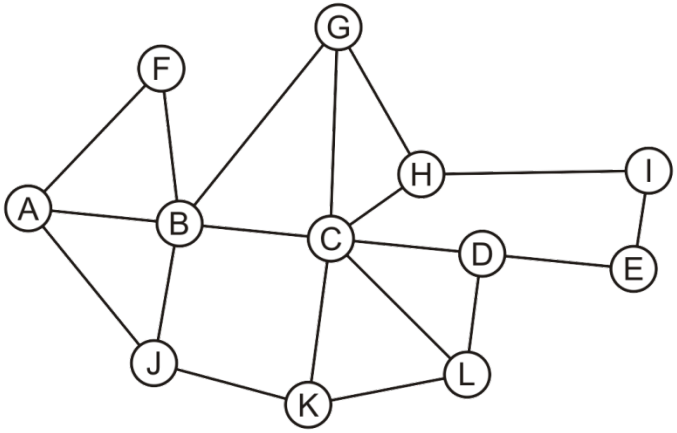


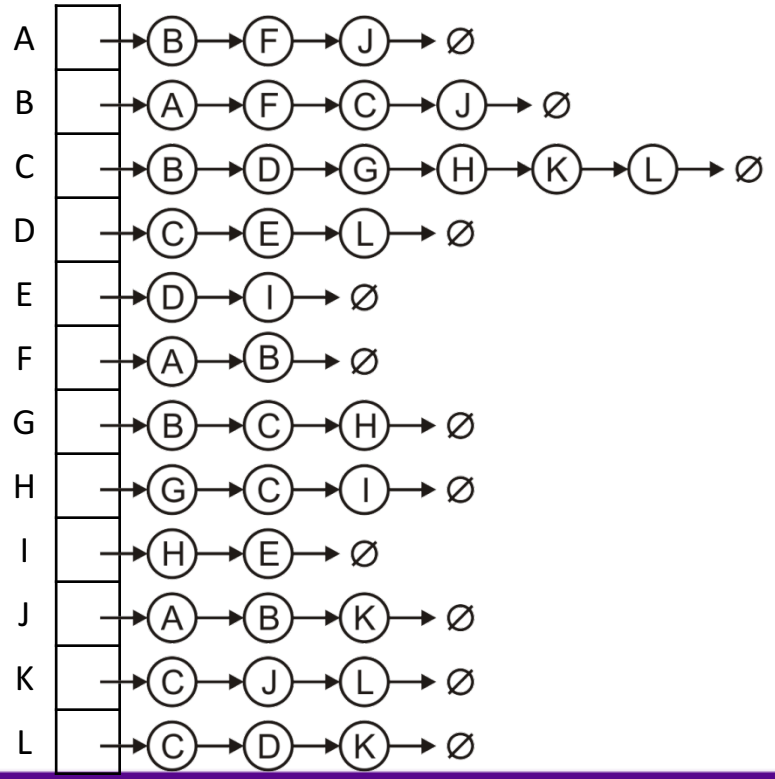
	A	B	C	D	E	F	G	H	I	J	K	L
A		×				×				×		
B	×		×			×	×			×		
C		×		×			×	×			×	×
D			×		×							×
E				×					×			
F	×	×										
G		×	×					×				
H			×				×		×			
I					×			×				
J	×	×									×	
K			×							×		×
L			×	×							×	

Array of Linked Lists

Suppose we allow arbitrary relations between any two objects in a container

- Alternatively, we could use a hybrid: an array of linked lists





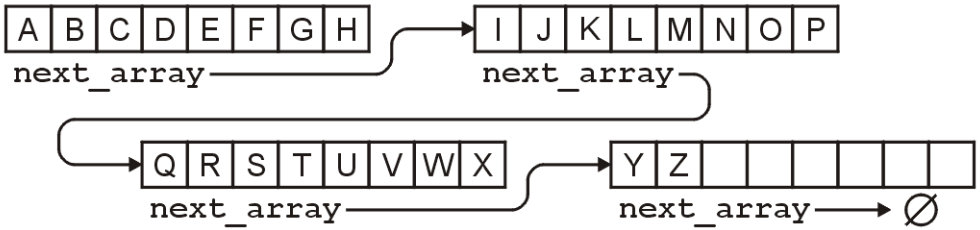
Linked Arrays

Other hybrids are linked lists of arrays

- Something like this is used for the C++ STL `deque` container

For example, the alphabet could be stored either as:

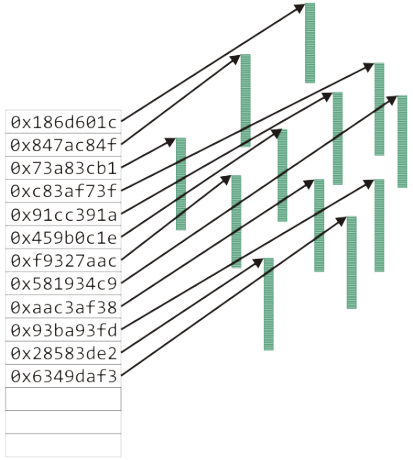
- An array of 26 entries, or
- A linked list of arrays of 8 entries



Hybrid data structures

The Unix inode was used to store information about large files

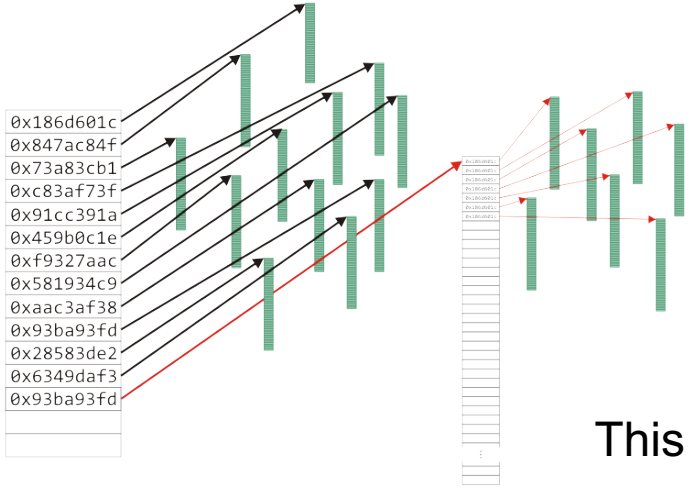
– The first twelve entries can reference the first twelve blocks (48 KiB)



Hybrid data structures

The Unix inode was used to store information about large files

- The next entry is a pointer to an array that stores the next 1024 blocks

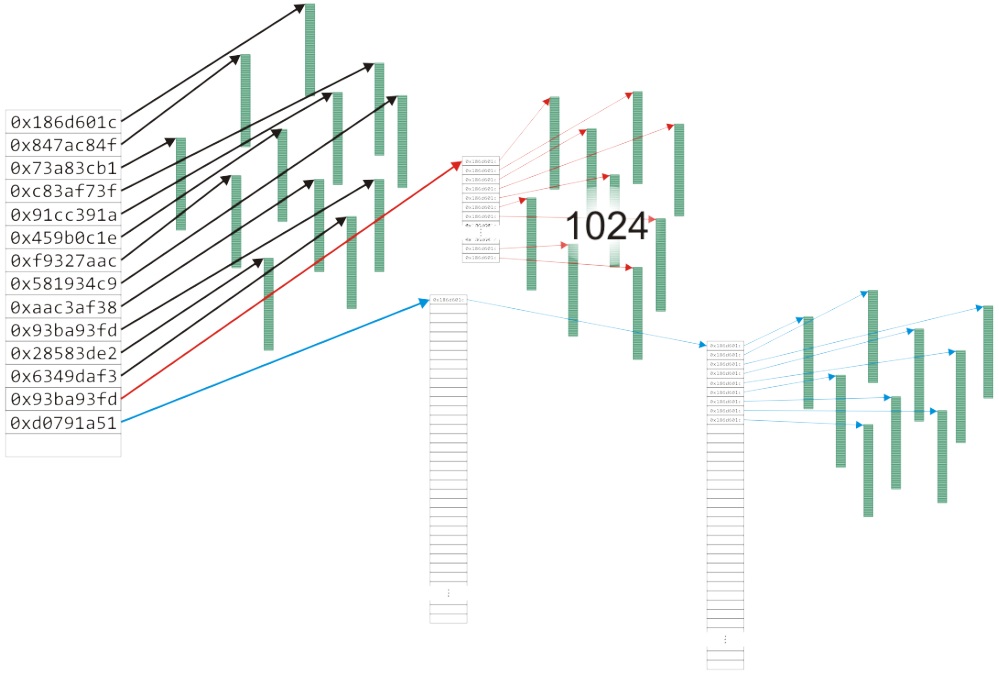


This stores files up to 4 MiB on a 32-bit computer

Hybrid data structures

The Unix inode was used to store information about large files

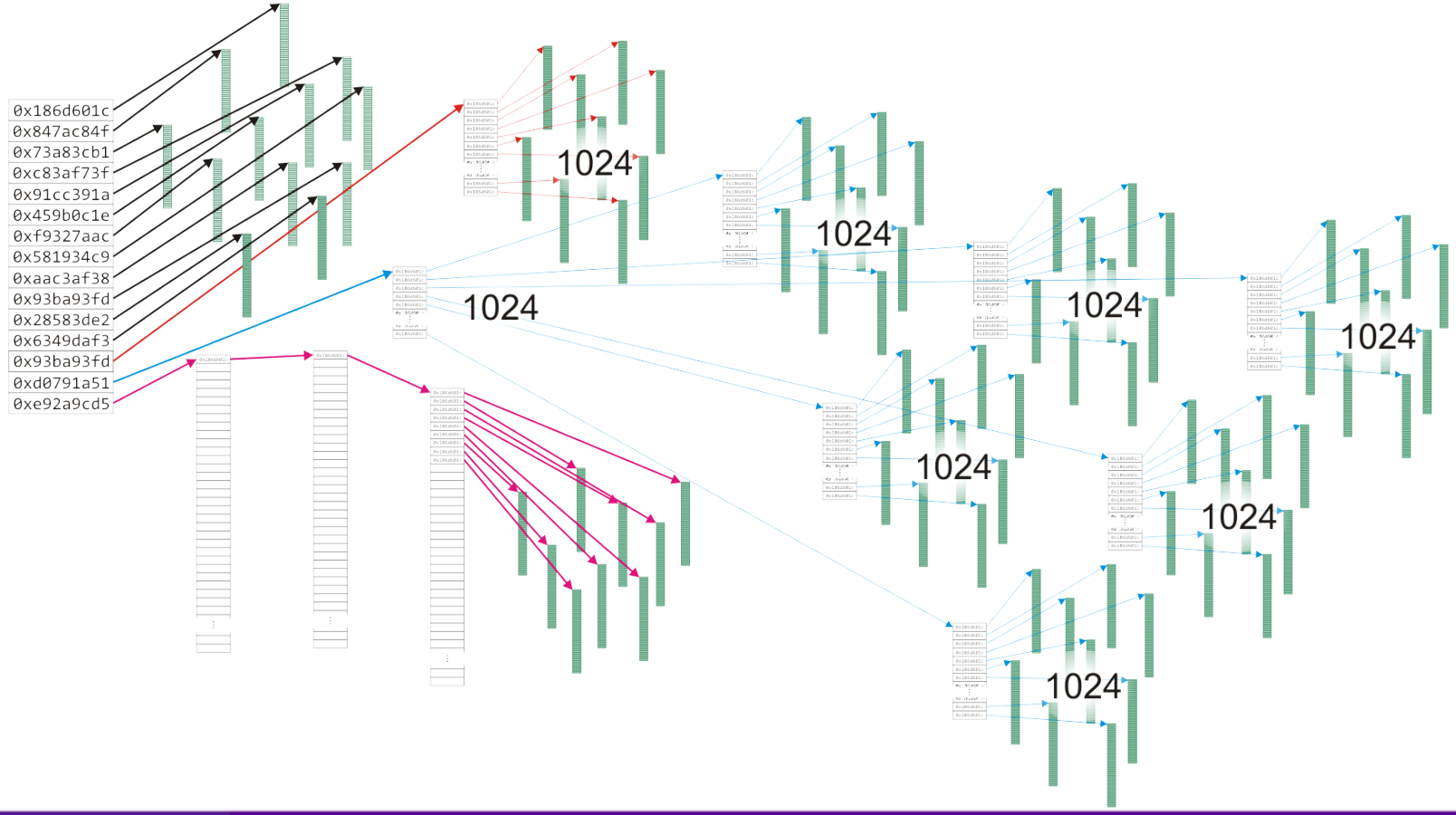
– The next entry has two levels of indirection for files up to 4 GiB



Hybrid data structures

The Unix inode was used to store information about large files

- The last entry has three levels of indirection for files up to 4 TiB



Algorithm run times

Once we have chosen a data structure to store both the objects and the relationships, we must implement the queries or operations as algorithms

- The Abstract Data Type will be implemented as a class
- The data structure will be defined by the member variables
- The member functions will implement the algorithms

The question is, how do we determine the efficiency of the algorithms?

Operations

We will us the following matrix to describe operations at the locations within the structure

	Front/1^{st}	Arbitrary Location	Back/n^{th}
Find	**?**	**?**	**?**
Insert	**?**	**?**	**?**
Erase	**?**	**?**	**?**

Operations on Sorted Lists

Given an sorted array, we have the following run times:

	Front/1^{st}	Arbitrary Location	Back/n^{th}
Find	Good	Okay	Good
Insert	Bad	Bad	Good* Bad
Erase	Bad	Bad	Good

* only if the array is not full

Operations on Lists

If the array is not sorted, only one operations changes:

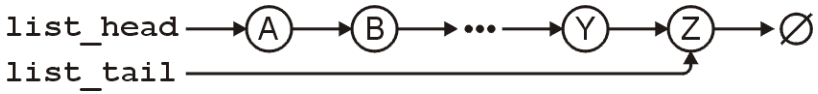
	Front/1^{st}	Arbitrary Location	Back/n^{th}
Find	Good	Bad	Good
Insert	Bad	Bad	Good* Bad
Erase	Bad	Bad	Good

* only if the array is not full

Operations on Lists

However, for a singly linked list where we a head and tail pointer, we have:

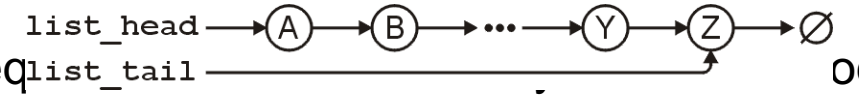
	Front/1^{st}	Arbitrary Location	Back/n^{th}
Find	Good	Bad	Good
Insert	Good	Bad	Good
Erase	Good	Bad	Bad



Operations on Lists

If we have a pointer to the k^{th} entry, we can insert or erase at that location quite easily

	Front/1^{st}	Arbitrary Location	Back/n^{th}
Find	Good	Bad	Good
Insert	Good	Good	Good
Erase	Good	Good	Bad

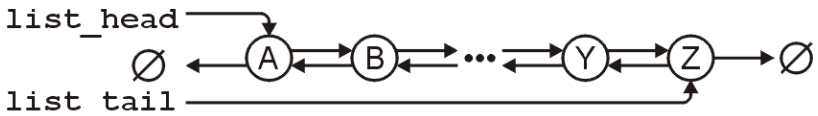


- Note, this req odify the value stored in the k^{th} node
- This is a common co-op interview question!

Operations on Lists

For a doubly linked list, one operation becomes more efficient:

	Front/1^{st}	Arbitrary Location	Back/n^{th}
Find	Good	Bad	Good
Insert	Good	Good	Good
Erase	Good	Good	**Good**



Following Classes

The next topic, asymptotic analysis, will provide the mathematics that will allow us to measure the efficiency of algorithms

It will also allow us the measure the memory requirements of both the data structure and any additional memory required by the algorithms

Following Classes

Following our discussion on asymptotic and algorithm analysis, we will spend

- 13 lectures looking at data structures for storing linearly ordered data
- One week looking at data structures for relation-free data
- Four lectures on sorting
- One week on partial orderings and adjacency relations, and
- Two weeks on algorithm design techniques

Summary

In this topic, we have introduced the concept of data structures

- We discussed contiguous, linked, and indexed allocation
- We looked at arrays and linked lists
- We considered
  - Trees
  - Two-dimensional arrays
  - Hybrid data structures
- We considered the run time of the algorithms required to perform various queries and operations on specific data structures:
  - Arrays and linked lists

Asymptotic Analysis of Algorithm

Outline

In this topic, we will look at:

- Justification for analysis
- Quadratic and polynomial growth
- Counting machine instructions
- Landau symbols
- Big-Θ as an equivalence relation
- Little-o as a weak ordering

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*

Asymptotic Analysis

In general, we will always analyze algorithms with respect to one or more variables

We will begin with one variable:

- The number of items n currently stored in an array or other data structure
- The number of items expected to be stored in an array or other data structure
- The dimensions of an $n \times n$ matrix

Examples with multiple variables:

- Dealing with n objects stored in m memory locations
- Multiplying a $k \times m$ and an $m \times n$ matrix
- Dealing with sparse matrices of size $n \times n$ with m non-zero entries

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;
}
```

Maximum Value

One comment:

- In this class, we will look at both simple C++ arrays and the standard template library (STL) structures
- Instead of using the built-in array, we could use the STL **`vector`** class
- The **`vector`** class is closer to the C#/Java array

Maximum Value

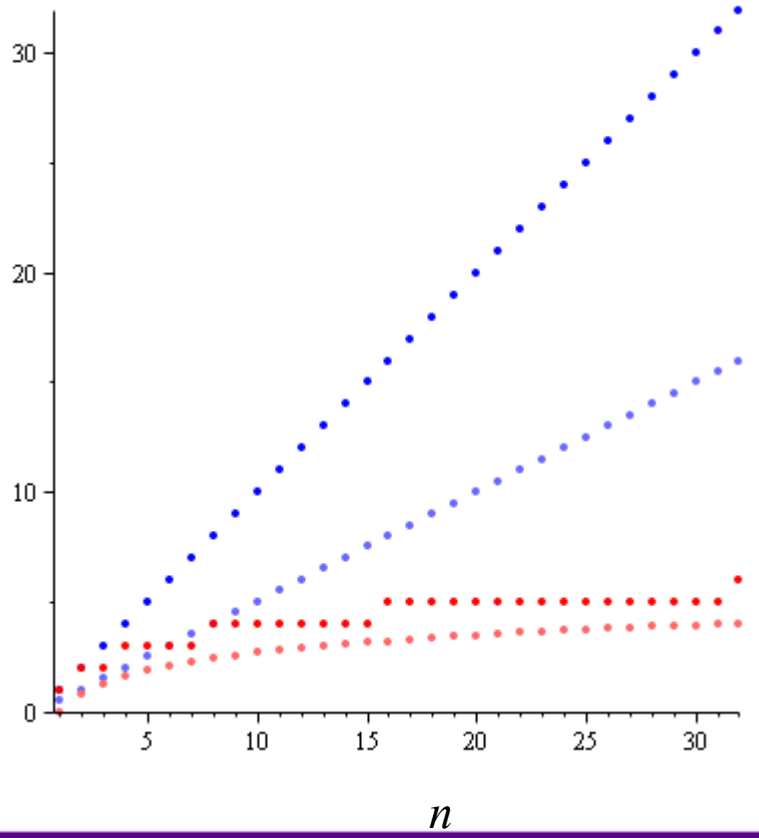
```
#include <vector>

int find_max( std::vector<int> array ) {
    if ( array.size() == 0 ) {
        throw underflow();
    }

    int max = array[0];

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

    return max;
}
```

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



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

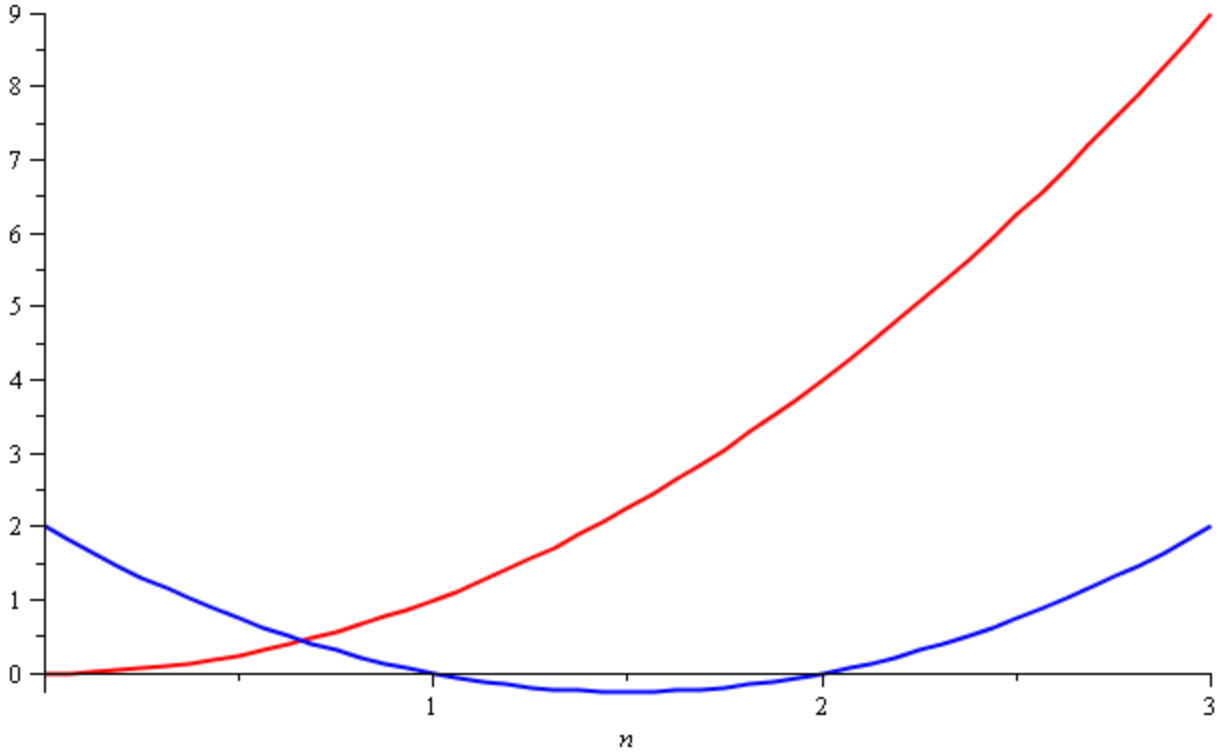
For this, we need Landau symbols and the associated asymptotic analysis

Quadratic Growth

Consider the two functions

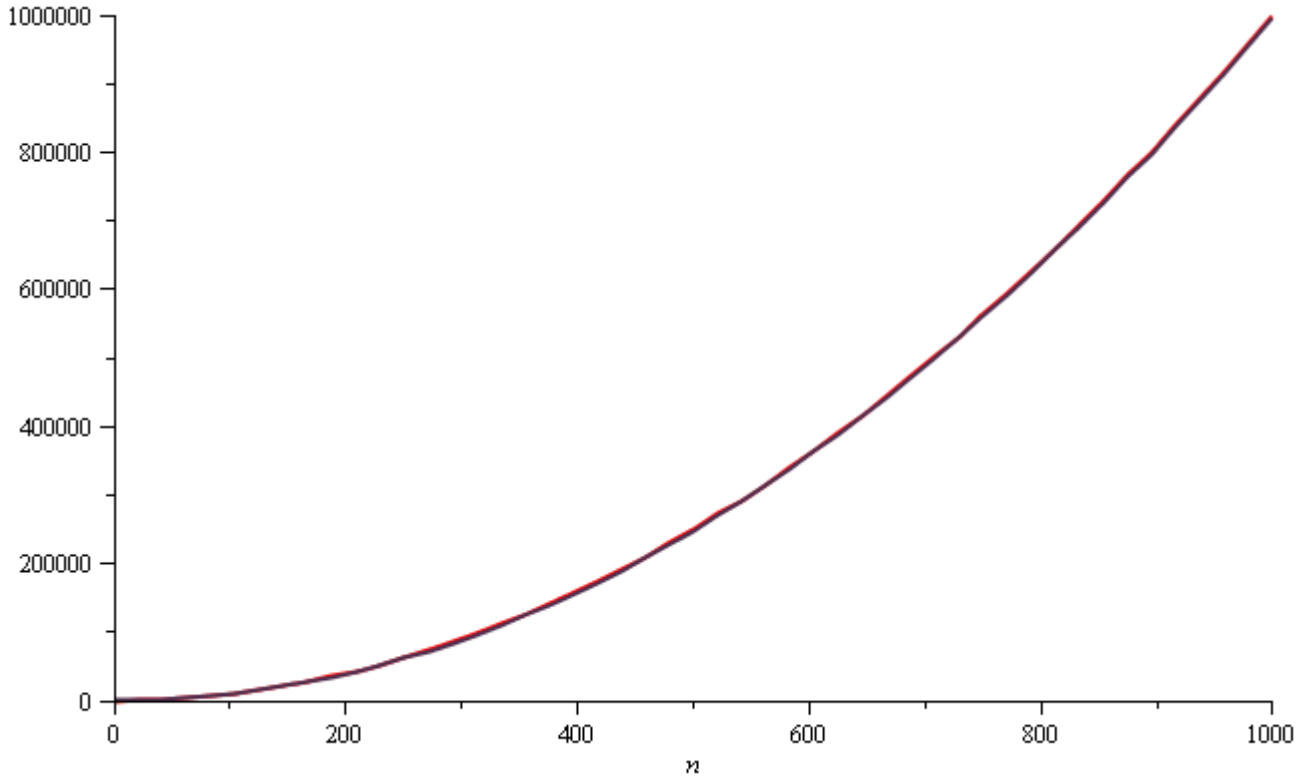
$f(n) = n^2$ and $g(n) = n^2 - 3n + 2$

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

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

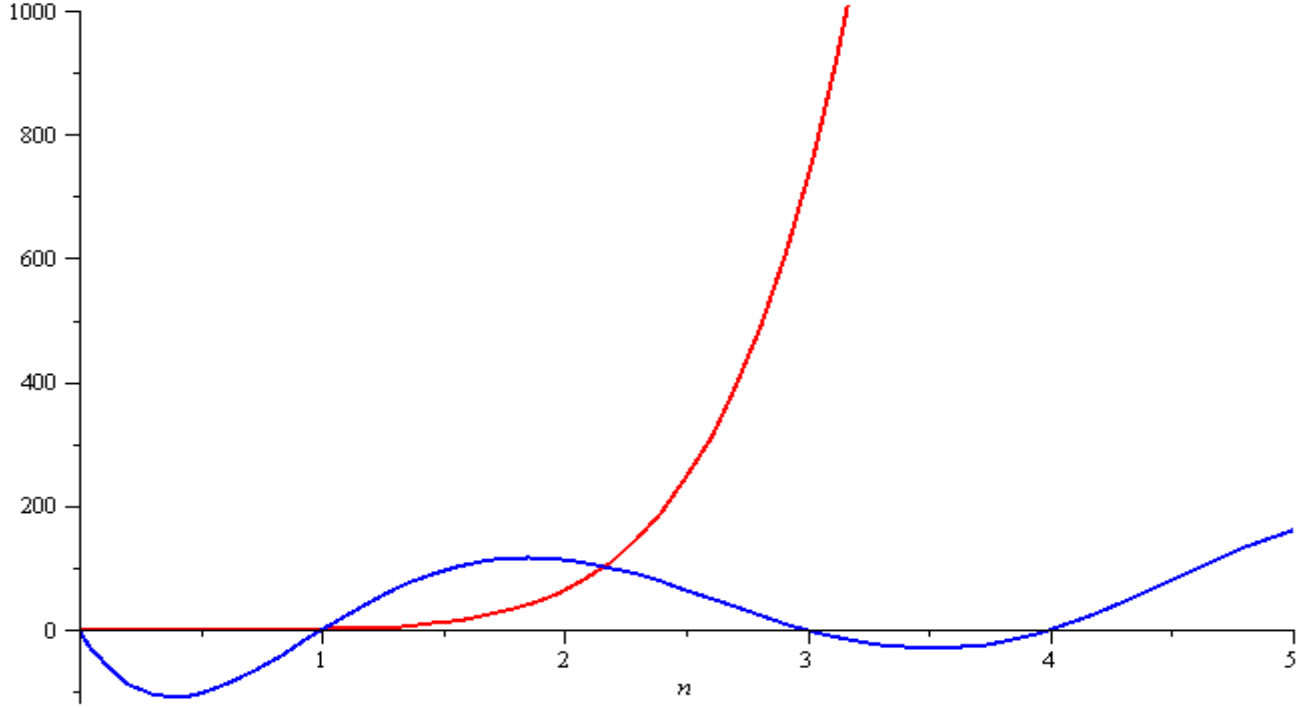
and this difference goes to zero as $n \to \infty$

Polynomial Growth

To demonstrate with another example,

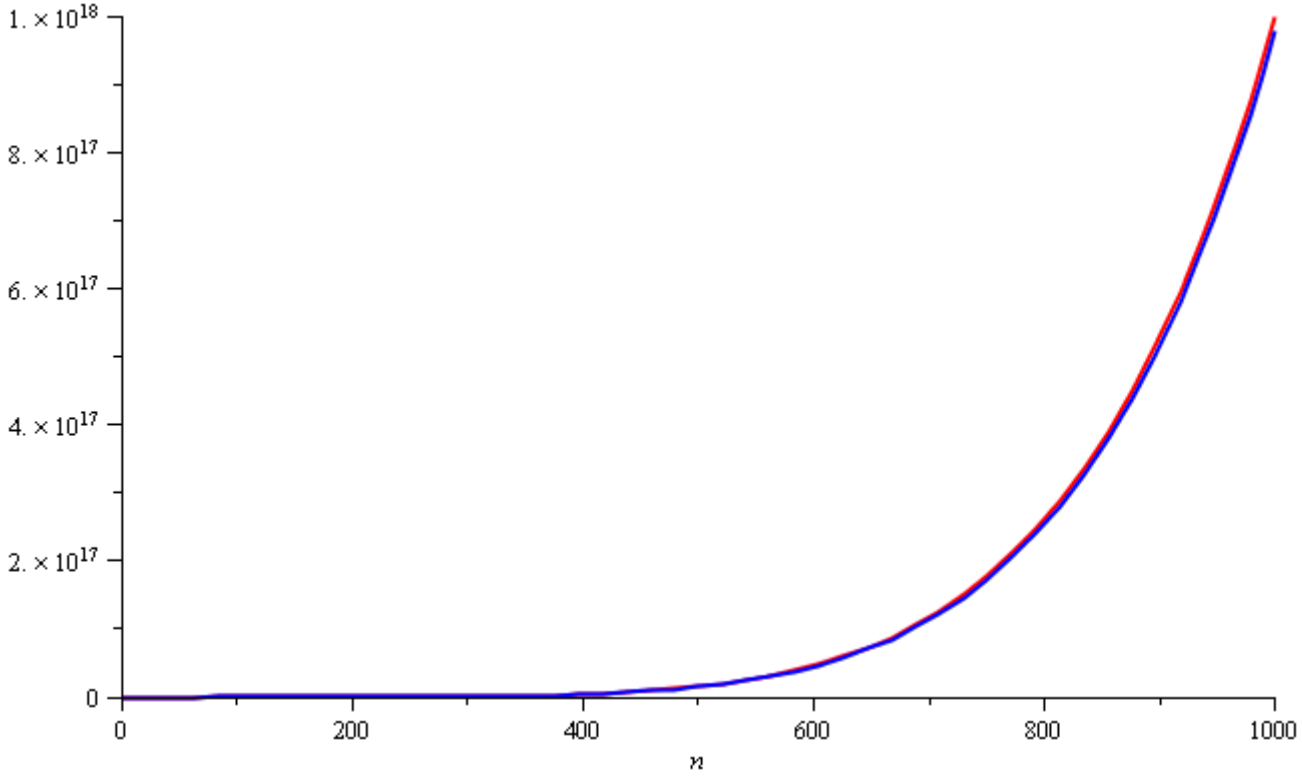
$f(n) = n^6$ and $g(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$

Around $n = 0$, they are very different



Polynomial Growth

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



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

Examples

We will now look at two examples:

- A comparison of selection sort and bubble sort
- A comparison of insertion sort and quicksort

Counting Instructions

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

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

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

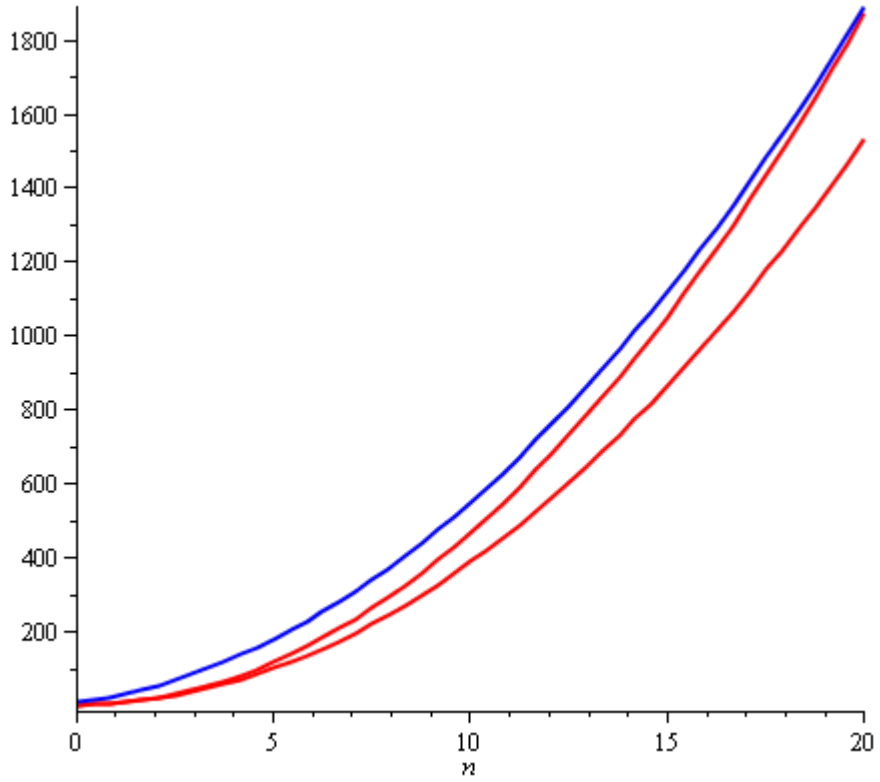
$$s(n) = 4n^2 + 14n + 12 \quad \text{Selection sort}$$

The smaller the value, the fewer instructions are run

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

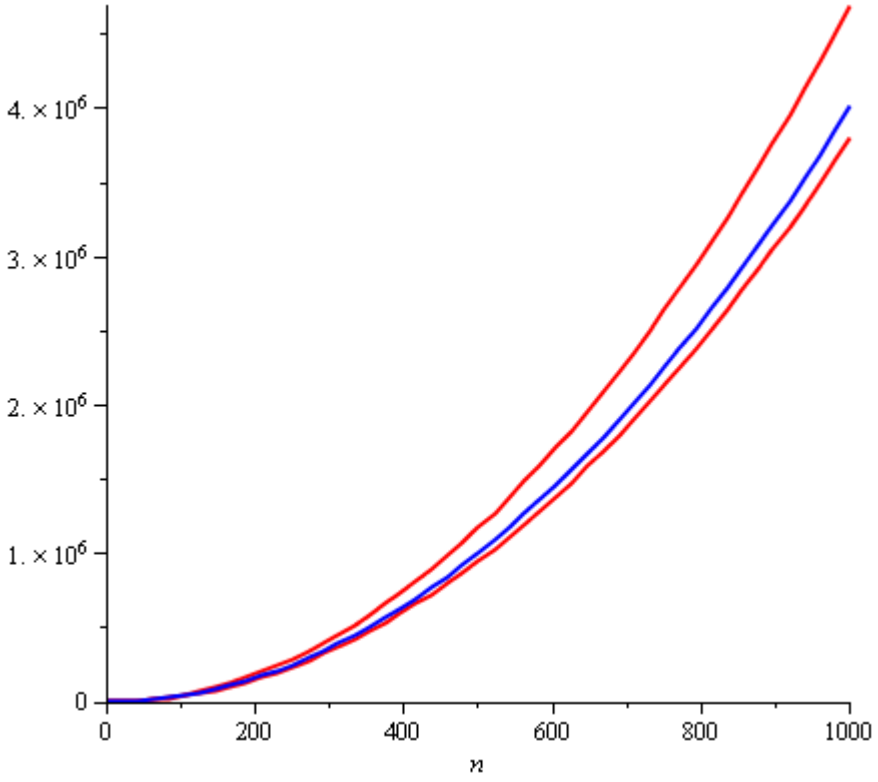
Counting Instructions

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



Counting Instructions

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



Counting Instructions

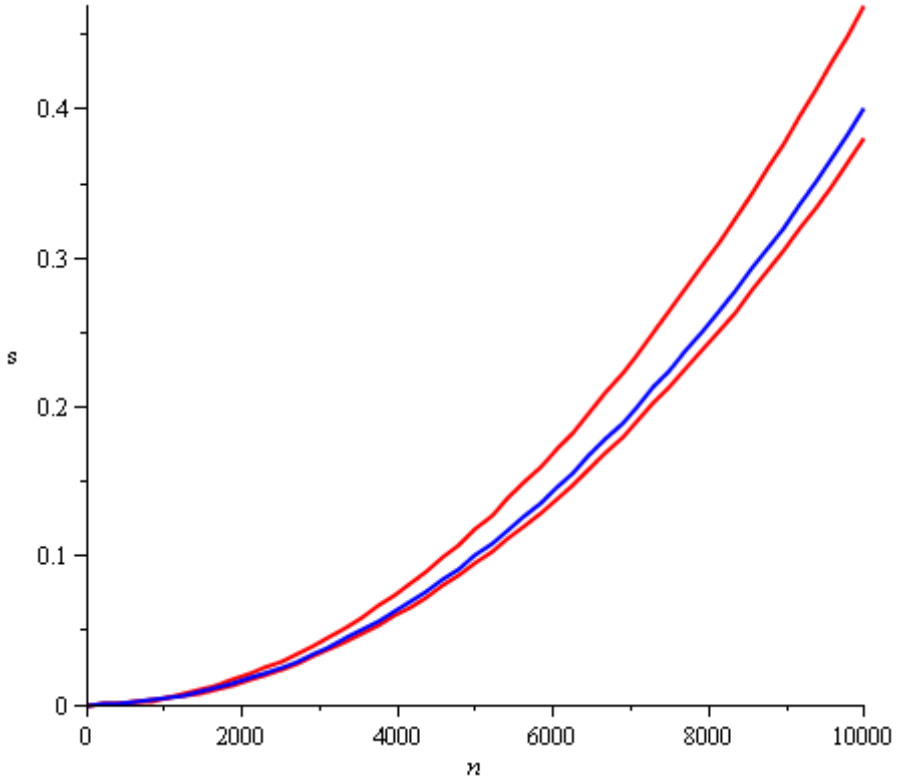
Is this a serious difference between these two algorithms?

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

Counting Instructions

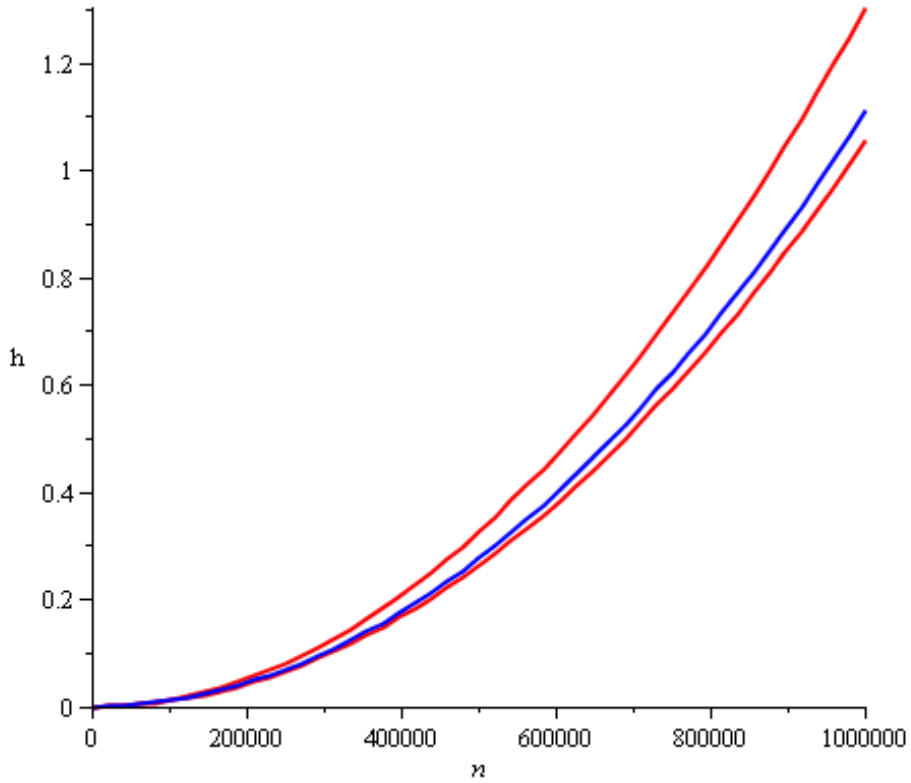
Suppose we have a $1\ \mathrm{GHz}$ computer

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



Counting Instructions

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

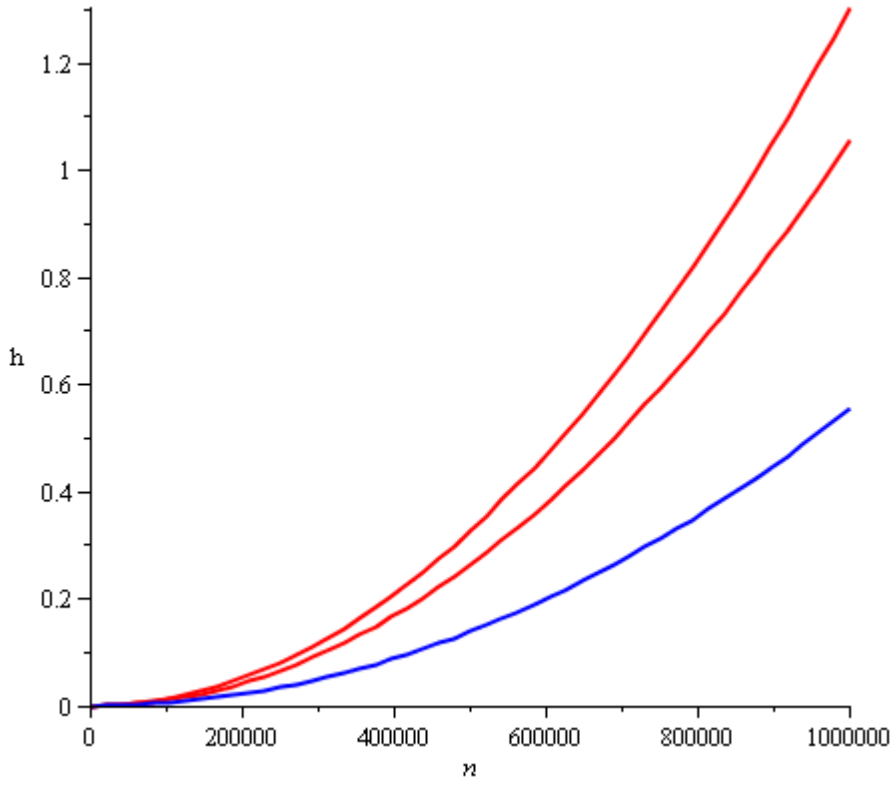


Bubble sort could, under some conditions, be 200 s faster

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



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)

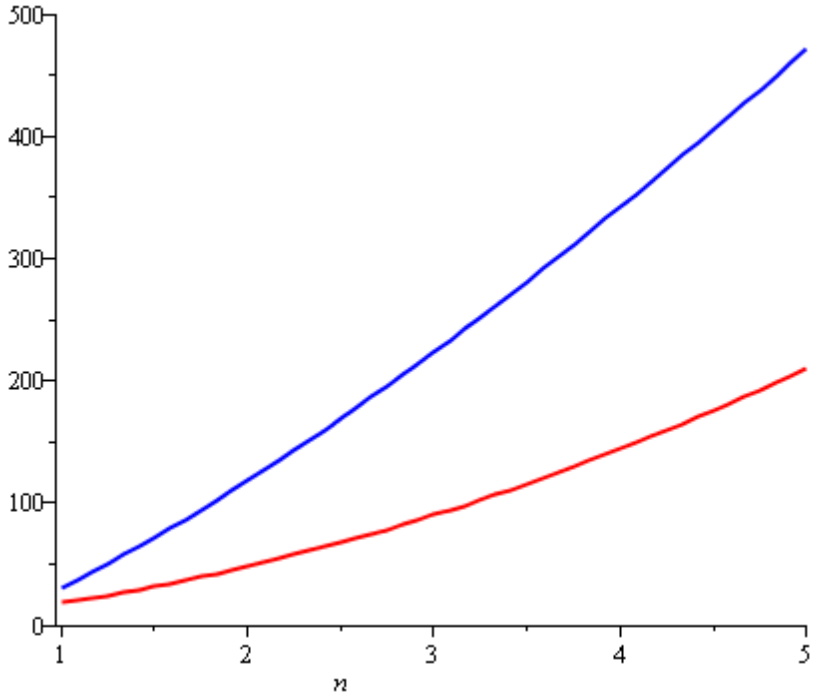
Question:

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

Counting Instructions

As another example:

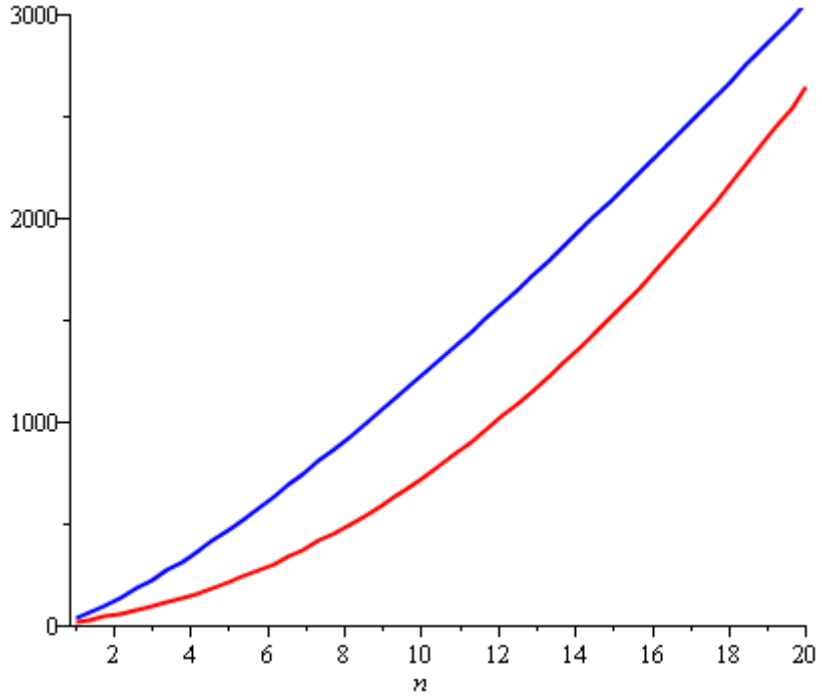
- Compare the number of instructions required for insertion sort and for quicksort
- Both functions are concave up, although one more than the other



Counting Instructions

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

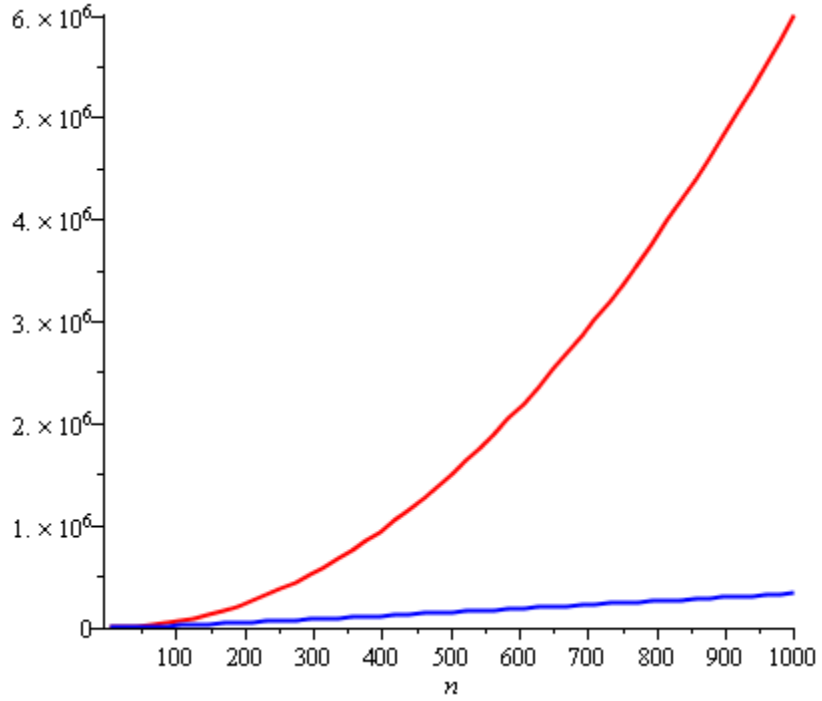
- Never-the-less, the graphic suggests it is more useful to use insertion sort when sorting small lists—quicksort has a large overhead



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

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

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

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 we will consider these on a one-by-one basis

Weak ordering

In general, there are functions such that

- 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 we will consider these on a one-by-one basis

Landau Symbols

Recall Landau symbols from 1st year:

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
- We conclude 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 $g(n)$

Landau Symbols

These definitions are often unnecessarily tedious

Note, however, that if $\mathrm{f}(n)$ and $\mathrm{g}(n)$ are polynomials of the same degree with positive leading coefficients:

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

Landau Symbols

Suppose that $\mathrm{f}(n)$ and $\mathrm{g}(n)$ satisfy $\lim_{n} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} = c$

From the definition, this means given $c > \varepsilon > 0$ there

exists an $N > 0$ such that $\left| \frac{\mathrm{f}(n)}{\mathrm{g}(n)} \quad c \right| <$ whenever $n > N$

That is,

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

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

Landau Symbols

However, the statement says that $f(n) = \Theta(g(n))$

$$g(n)(c \quad) < f(n) < g(n)(c + \quad)$$

Note that this only goes one way:

If $\lim_{n} \frac{f(n)}{g(n)} = c$ where $0 < c <$, it follows that $f(n) = \Theta(g(n))$

Landau Symbols

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

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

There are other possibilities we would like to describe:

If $\lim_{n} \frac{f(n)}{g(n)} = 0$, 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

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

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

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

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

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

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

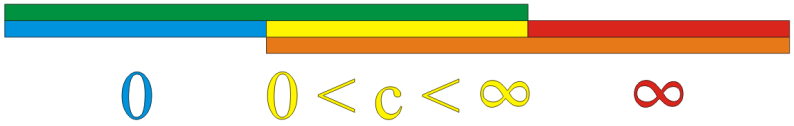
$f(n) = \mathbf{\Omega}(g(n))$ is equivalent to $f(n) = \mathbf{\Theta}(g(n))$ or $f(n) = \boldsymbol{\omega}(g(n))$

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



Landau Symbols

Some other observations we can make are:

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

$$f(n) = \mathbf{O}(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

$$f(n) = \mathbf{o}(g(n)) \Leftrightarrow g(n) = \omega(f(n))$$

Big-Θ as an Equivalence Relation

If we look at the first relationship, we notice that $f(n) = \Theta(g(n))$ seems to describe an equivalence relation:

1. $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
2. $f(n) = \Theta(f(n))$
3. If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, it follows that $f(n) = \Theta(h(n))$

Consequently, we can group all functions into equivalence classes, where all functions within one class are big-theta Θ of each other

Big-Θ as an Equivalence Relation

For example, all of

$$n^2 \qquad 100000\,n^2 - 4\,n + 19 \qquad n^2 + 1000000$$

$$323\,n^2 - 4\,n\ln(n) + 43\,n + 10 \qquad 42n^2 + 32$$

$$n^2 + 61\,n\ln^2(n) + 7n + 14\ln^3(n) + \ln(n)$$

are big-Θ of each other

E.g., $42n^2 + 32 = \Theta(\,323\,n^2 - 4\,n\ln(n) + 43\,n + 10\,)$

Big-Θ as an Equivalence Relation

Recall that with the equivalence class of all 19-year olds, we only had to pick one such student?

Similarly, we will select just one element to represent the entire class of these functions: n^2

- We could chose any function, but this is the simplest

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

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

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

Logarithms and Exponentials

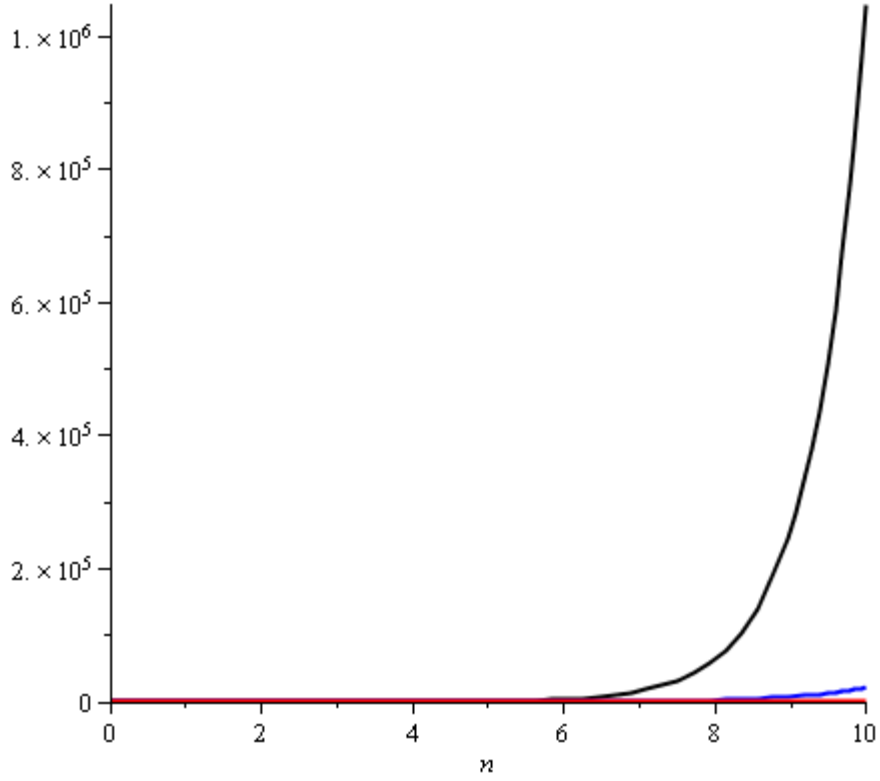
Plotting 2^n, e^n, and 4^n on the range $[1, 10]$ already shows how significantly different the functions grow

Note:

$2^{10} = 1024$

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

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



Little-o as a Weak Ordering

We can show that, for example

$$\ln(n) = \mathbf{o}(n^p)$$

for any $p > 0$

Proof: Using l'Hôpital's rule, we have

$$\lim_n \frac{\ln(n)}{n^p} = \lim_n \frac{1/n}{pn^{p-1}} = \lim_n \frac{1}{pn^p} = \frac{1}{p}\lim_n n^{-p} = 0$$

Conversely, $1 = \mathbf{o}(\ln(n))$

Little-o as a Weak Ordering

Other observations:

- If p and q are real positive numbers where $p < q$, it follows that

$$n^p = \mathbf{o}(n^q)$$

- For example, matrix-matrix multiplication is $\Theta(n^3)$ but a refined algorithm is $\Theta(n^{\lg(7)})$ where $\lg(7) \approx 2.81$

- Also, $n^p = \mathbf{o}(\ln(n)n^p)$, but $\ln(n)n^p = \mathbf{o}(n^q)$
  - n^p has a slower rate of growth than $\ln(n)n^p$, but
  - $\ln(n)n^p$ has a slower rate of growth than n^q for $p < q$

Little-o as a Weak Ordering

If we restrict ourselves to functions $f(n)$ which are $\Theta(n^p)$ and $\Theta(\ln(n)n^p)$, we note:

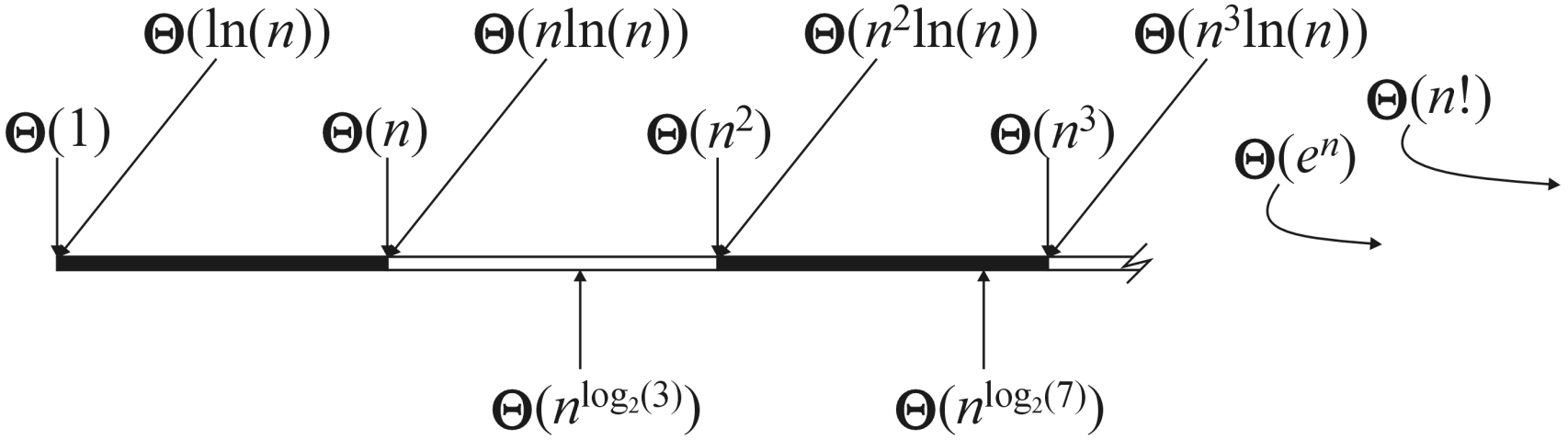
- It is never true that $f(n) = \mathbf{o}(f(n))$
- If $f(n) \neq \Theta(g(n))$, it follows that either

  $$f(n) = \mathbf{o}(g(n)) \text{ or } g(n) = \mathbf{o}(f(n))$$
- If $f(n) = \mathbf{o}(g(n))$ and $g(n) = \mathbf{o}(h(n))$, it follows that $f(n) = \mathbf{o}(h(n))$

This defines a weak ordering!

Little-o as a Weak Ordering

Graphically, we can shown this relationship by marking these against the real line



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*

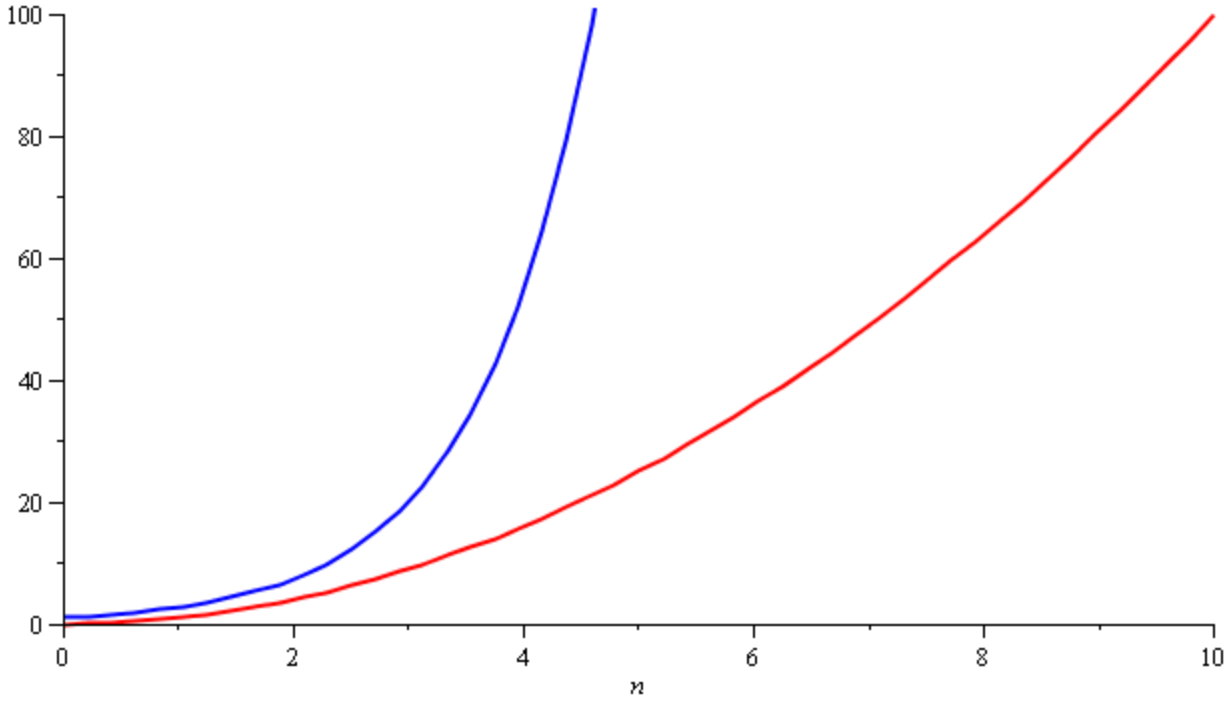
Problems that have no known polynomial-time algorithms are said to be *intractable*

- 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