Data Structures and Algorithms

Alice E. Fischer

January 22, 2016

The Course Website

Write this down: **http://eliza.newhaven.edu/datastr/**
Posted on this website will be:

- The complete syllabus and emergency announcements.
- My two email addresses.
- Reading assignments and lecture notes, when available.
- Written homework assignments.
- Programming assignments.

Learning C++

We will spend some time in class each day on C++.

- Most of the class already knows this language, and the rest know Java.
- If you want some extra instruction, come to my Monday night class for a couple of weeks. Almost none of the graduate students know C++, so class will start each night with C++ material.
- Try the programming assignment and ask questions freely in class, in person, or by email.
- I will introduce parts of the language as they are needed, over the first few weeks.

The First Programming Assignment

Run-Length Encoding

- Due in one week
- In C++
- A very simple program to introduce everyone to C++, the required IDE, and the UNH environment.
- Compress or decompress a file according to the run-length algorithm.
- Replace every run of 4 or more copies of the same letter by a triplet consisting of an escape code, the letter, and a count. Replace 1 or more copies of the escape character by a triplet.

The C++ Compilation Model

Header files
Implementation files
Compilation and type checking
Linking

.hpp and .cpp

A program is a collection of modules Each module has a header file with a name ending in .hpp.

- A header contains
  - #include commands for headers of other modules.
  - type declarations.
  - class declarations.
  - function prototypes for the functions in this module.
  - inline function definitions for short functions.
- Most modules also have a separate implementation file with a file name ending in .cpp
  - ONE #include command for the matching header
  - The code for all the non inline functions in the class.
  - defining declarations of static class members.

Header files.

- A header file should begin with *an include guard*. The purpose is to prevent the same declaration from being included twice in a module. The compiler cannot deal with duplicate definitions.
- An include guard is three lines of the form:

```
#ifndef MODNAME_H
#define MODNAME_H
... // the header declarations go here
#endif
```

- This tells the compiler to include this header if it is not already defined.
- If you fail to #include a needed header, or get your include guards wrong, or have a circular #include, compilation will end with a "missing file" or "undefined symbol" error.

Why do we #include header files?

- The #include command tells the compiler to bring in the type declarations and prototypes for another module.
- They are used at compile time to type-check every expression and function call.
- The type of every part of an expression analyzed. The two operands of an operator must match. The arguments in a function call must match the parameter types in the function.
- When the types do not match perfectly, the compiler will look for a conversion that it can apply to make them match. If no conversion is found, the compiler stops with a type error.
- Prototype information is used in this comparison.
- The declared return type of the function is used to check the surrounding context.

Each module is compiled seperatly.

- Only .cpp files are compiled. Each one includes its own header, and each header includes all the other headers needed to compile the .cpp file.
- Inline functions (written in the .hpp files) are compiled when they are included by a .cpp file.
- Typically, one .hpp file will be included in at least two .cpp files: its own module and the client module(s).
- Compiling a .cpp file produces a .o file, one per module. Each .o file starts with a list of the entry points to the functions it provides, and a list of the “foreign” functions it calls.
- These .o files are linked together by the *loader*, which matches the needs to the functions provided.
- If the match cannot be made, a long, strange, and confusing error comment is displayed

Two kinds of linker errors are possible.

- Missing function definition. This happens when you failed to put one of the modules into the list on the command line of modules to be compiled.
  If you are using an IDE, it happens when you fail to tell the IDE to put a .cpp file into the project.
- Multiply defined symbol. This happens when you have an object or variable declaration in a header file. All variables and constants must be in the .cpp file.

Data Structures

What is a Data Structure?
Relationships in the Data
Application Requirements
Efficiency

What is this course about?

Data Structures a required course in every CS curriculum.

We look three main issues:

- Representing the relationships among a set of related objects,
- In a way that is efficient to process,
- Given the intended application and its requirements.

We design or select an appropriate data structure for an application based on the answers to these questions.

Relationships

When an application works with a set of objects, those objects may be related in several ways:

- Maybe they have some natural order based on the data in the object.
- Maybe the order depends on timing, not on the data.
- Maybe the order does not matter.

Application Requirements

Applications that work with sets of objects must all do this:

- Put a new object into the set.
- Remove an object from the set.
- Find a given object stored somewhere in the set
- Traverse all the objects in the set.

Applications can differ greatly on these issues:

- The number of data objects in the set.
- Whether the objects are large or small.
- How often objects are added or removed.
- How often the whole set needs to be searched.
- Whether there is a need to traverse the set in a specified order.

Efficiency

We need to consider three kinds of efficiency:

- How much time does it take to process N operations on the data set?
- How much space will it take to store those items and their relationships?
- How much work will it be for a programmer to implement the data structures and the algorithms that operate on them?

All three of these issues are important and play a role in program design

Big-O Notation

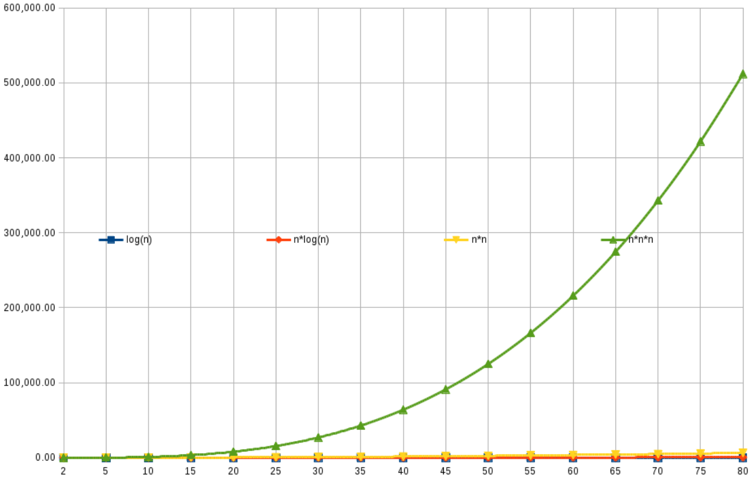
Algorithms that are not practical.
Algorithms that are slow
Good algorithms and better ones
What Big-O cannot tell us. Big-O Comes from the Program Structure

Big-O Notation

We use Big-O notation to talk about time efficiency.

- The Big-O of a function, $f()$ is the upper bound of the time it can take to execute $f(n)$ on a set of n data items.
- We write $f(n) = O(n^2)$ if the upper bound of the execution time is expressed by a function with an n^2 term, and no term with a power higher than 2.
- Big-O rules. In the long run, as n grows larger, the Big-O measure determines which algorithms are practical, and which are not.

$n, \quad n \times log(n), \quad n^2, \quad n^3$

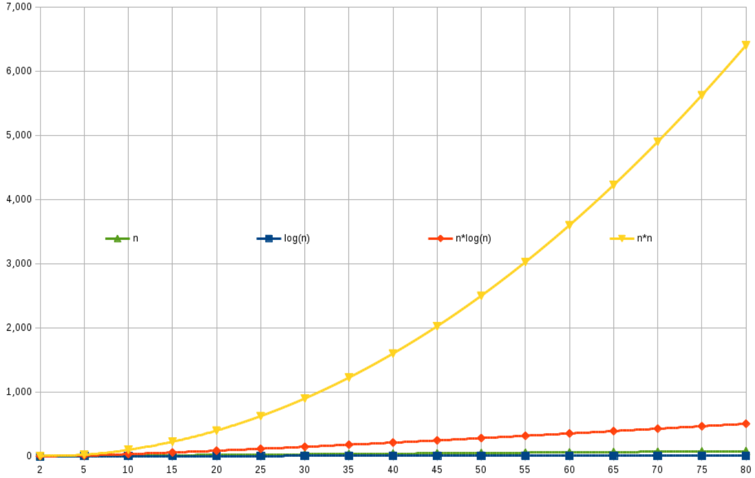


n^3 algorithms

- An n^3 algorithm is not scalable. Even at $n = 80$, there are over half a million operations.
- Examples: Matrix multiplication, graph isomorphism, transitive closure of a graph, context-free parsing.
- Compared to n^3, an n^2 algorithm is a dream.

$log(n), \quad n, \quad n \times log(n), \quad n^2, \quad n^3$

Here, we exclude the n^3 curve to see some detail in the others.

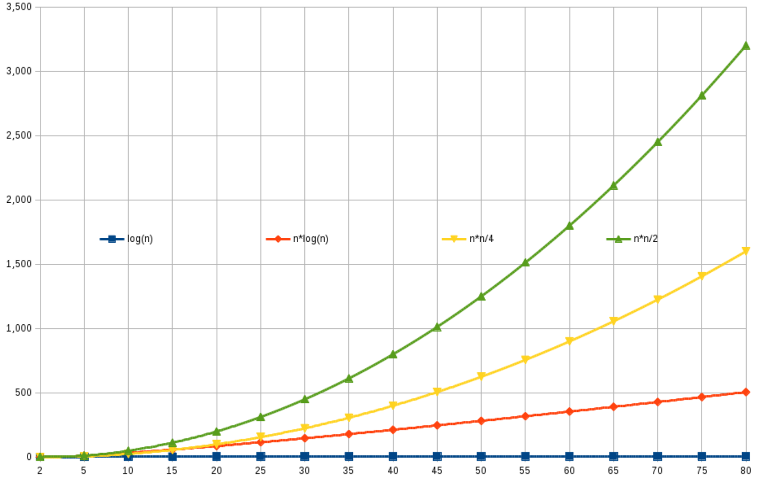


n^2 algorithms

- An n^2 algorithm is useful when n is small.
- At $n = 30$, $n^2 = 900$, whereas $n^3 = 27,000$
- Examples: Bubble sort, Selection sort, Insertion sort.
- But there is a huge difference between the n^2 curve and the $n \times log(n)$ curve

$log(n), \quad n \times log(n), \quad n^2/4, \quad n^2/2$

Here, we exclude the n^3 curve to see some detail in the others.

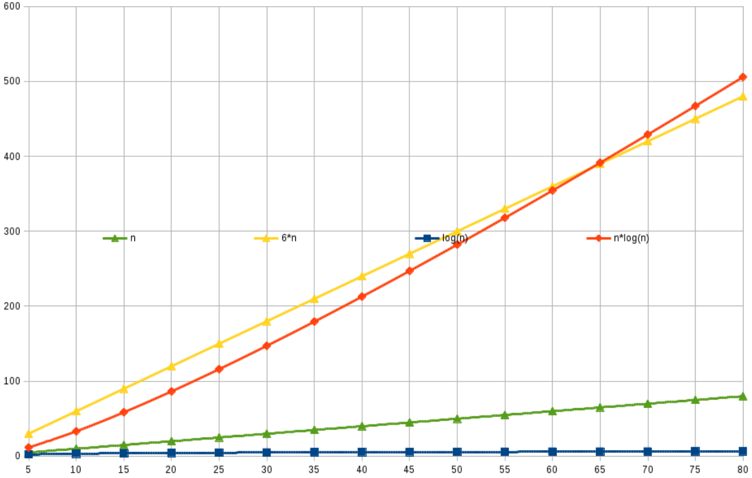


Common n^2) Sorting Algorithms

- A bubble sort uses a nested for/for loop and goes through the inner loop $n \times (n-1)/2$ times (green).
- A selection sort also uses a nested for/forloop but is faster because it does less swapping (green).
- Both are $O(n^2)$ – Big-O does not capture this kind of difference between algorithms.
- An insertion sort uses a nested for/while loop and, on the average, leaves the inner loop mid-way through it. So the running time is $n \times (n-1)/4$ times (yellow).
- So a bubble sort of 80 items would take time proportional to 3160 trips through the inner loop, while insertion sort would take 1580 loop executions.
- This is why you should never use bubble sort.

$log(n)$, n, $6 \times n$, $n \times log(n)$

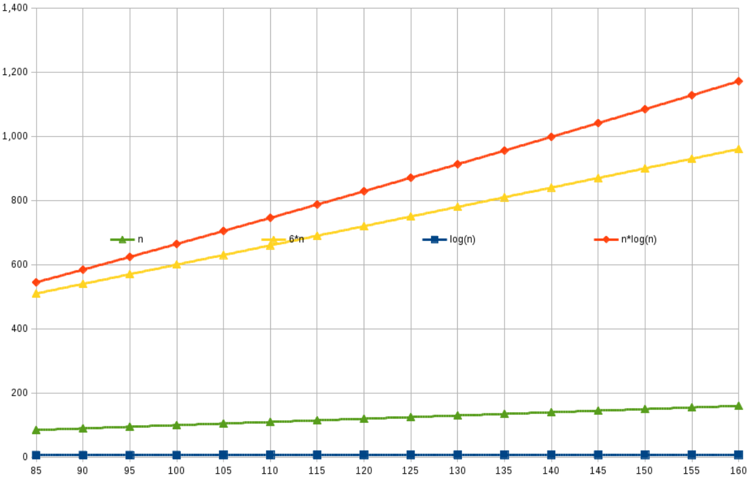
Now we exclude the n^2 curves and focus on the others



Common n^2) Sorting Algorithms

- The best sorting algorithms we have are $O(n \times log(n))$ (orange).
- These sorts include mergesort, quicksort, and heapsort.
- You can see that, for $n < 80$, this approximates $6 \times n$ (yellow, a straight line).
- After 80 (next slide) these lines slowly diverge.
- The curves for $log(n)$ grows very slowly (blue). This represents the running time of a binary search or finding a value in a tree-map.

$log(n), \quad n, \quad 6 \times n, \quad n \times log(n)$

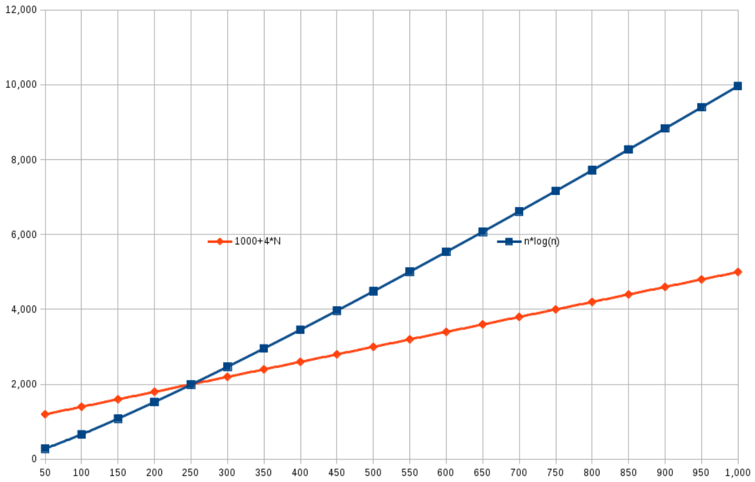


The effect of constant terms.

Big-O describes the performance in the long run.

- The graph for mergesort is blue ($n \times log(n)$),
- An approximation to the the graph for radix sort is orange.
- Radix sort is $O(n)$ but it starts out with an extensive setup phase, so a constant term is added to its running time.
- The Radix sort algorithm makes several passes over the data. For 4 passes, the running time is proportional to $c + 4 * n$.
- For a while, the ($n \times log(n)$) sort "wins" the race.
- Then the lines cross and stay crossed, and the O(n) algorithm is clearly better.

$log(n)$, vs. $c + n$


12,000
10,000
8,000
6,000
4,000
2,000
0
50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850 900 950 1,000
1000+4*N
n*log(n)

What Big-O Cannot Tell Us

Big-O does not tell the whole story.

- Big-O is a gross generalization. It separates algorithms into general categories.
- It does not give us a way to compare algorithms within the same category.
- It ignores constant terms and constant multipliers and focuses only on the skeleton of the way in which a function operates.
- One basic algorithm may have many implementations: some are easier to analyze, others include optimizations that make them more efficient. The quality of the coding can make a major difference in performance.

Big-O Comes from the Program Structure

The Big-O of a C function is determined by the code that defines the function.

- If there is a loop from 0 to N in the code, then the Big-O has an n term. A double loop would give an n^2 term, etc.
- If the code repeatedly divides the data in half, processes each half, then combines the results, there will be a $log_2 n$ term.
- Lines of code that are not inside loops or recursions contribute a constant term.