

Visualizing Data Structures

Daniel Petrisko

December 11, 2013

Abstract

To say algorithms are a major component of computer science is a gross understatement. Even as computers become more and more powerful, data sets expand faster still, dictating the need for the most efficient processing possible. The goal of my project is to illustrate the major properties of several algorithms for comparison data sets of varying size and complexity. Because of their relative simplicity and ease of illustration, I will focus on rudimentary sorting, searching and selecting.

1 Basics of Algorithmic Analysis

1.1 Algorithmic Notation

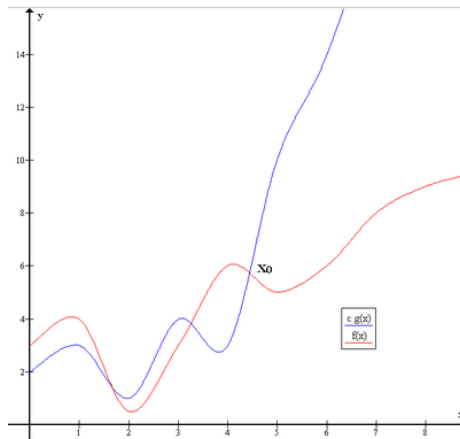


Figure 1: A graph where $f(x)$ is $O(g(x))$

Let's examine the running time of a simple algorithm in its worst and best case. "Worst case" means the data set that will take the longest to be sorted with this particular algorithm. "Best case" means the opposite. The notation generally accepted for basic algorithmic analysis is called Big-O notation.

A mathematical definition of Big-O is $f(x)$ is $O(g(x))$ if there exists $c \geq 0$ and k such that $f(x)$ is less than $cg(x)$ whenever $x \geq k$. Thus, the Big-O running time represents an upper bound in computational complexity based on the number of elements being processed. In order to prove these upper bounds,

we will use a technique called unrolling a recurrence relation. This is essentially reversing an inductive proof, where we take a general form of running time and trace to a base case. $T(n)$ is a function that takes the size of the data and returns the running time (in arbitrary computational units).

1.2 Sample Analysis - Bubble Sort

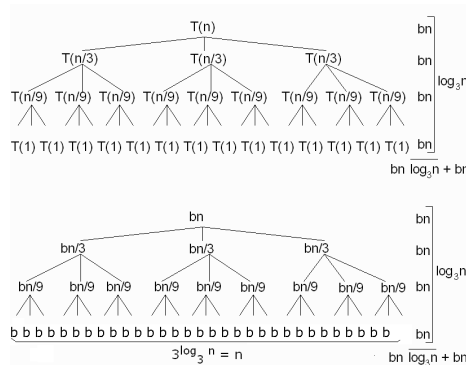


Figure 2: An example of unrolling a recurrence (not Bubble Sort)

Bubble Sort is a basic algorithm that runs linearly through the data set and swaps adjacent elements that are out of order with respect to each other. I claim that Bubble Sort's running time is:

$$T(n) = T(n-1) + n$$

$$T(0) = 1$$

This makes sense because for each pass through the array, the largest element will bubble up to the highest position in the set. This guarantees its position is correct, and thus does not require additional swapping, which essentially reduces the size of the remaining data set by one. The base case is that a set with no data is defined to be sorted. If we substitute T for itself in this equation, we get:

$$T(n) = T(n-1) + n = T(n-2) + n + n = T(n-3) + 3n$$

Until we reach the base case of:

$$T(n) = T(0) + n * n = 1 + n^2$$

We say that Bubble Sort is $O(n^2)$. That is, the running time of the algorithm is proportional to the number of elements squared. However, I also claim that in the best case, when the array is already sorted, bubble sort behaves differently. If you implement bubble sort to be intelligent it can detect that no swaps were done in the immediate pass, which means that the array is sorted. The recurrence becomes:

$$T(n) = 1 + n$$

And we say that bubble sort in best case is $O(n)$. That is, the running time is only proportional to the number of elements - much better than the previous case. This is the lower bound of Bubble Sort.

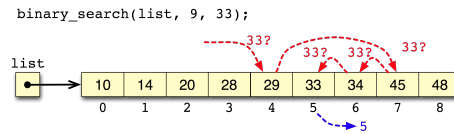


Figure 3: An illustration of binary search on a sorted array

1.3 Sample Analysis - Binary Search

Binary Search is a basic algorithm that finds the element with a certain value in a sorted data structure. It accomplishes this efficiently by repeatedly checking the middle element of the set, then dividing the data set into a smaller set of elements that are greater than or less than depending on the desired values relation to the middle element. I claim that Binary Search's running time is:

$$T(n) = T(n / 2) + 1$$

$$T(1) = 1$$

This makes sense because for each iteration, you inspect the middle element (the constant factor), and then divide the size of the data in half. The base case is running the algorithm on a single element - either it is the element (in the set), or it is not (not in the set). If we substitute T for itself in this equation, we get:

$$T(n) = T(n / 4) + 2 = T(n / 8) + 3$$

Until we reach the base case of:

$T(n) = T(0) + \log(n) = 1 + \log(n)$. And we say that binary search is $O(\log(n))$, so proportional to the log of the number of elements. This is clearly a better case than a linear search $O(n)$ time, which grows much faster.

2 Factors of a Basic Comparison Algorithm

2.1 Sections

Use section and subsection commands to organize your document. \LaTeX handles all the formatting and numbering automatically. Use ref and label commands for cross-references.

2.2 Computational Complexity

The number of element comparisons made by the algorithm, generally given as a function of the size of the input list. There is a generally accepted lower bound of average performance on the order of $O(n \log n)$ comparisons for comparison based sorting algorithms.

2.3 Memory Usage

In particular, there are two major categories of algorithms: in place and out of place. Out of place algorithms use additional memory than the original data. For many situations, this extra memory use is unacceptable either because of restricted space and resources, or the additional processing time required to access data between sections of memory. In addition, in place algorithms can be

optimized to be as efficient in most cases. All the sorting algorithms illustrated in my project will be in place algorithms.

2.4 Stability and Adaptability

Stable sorting algorithms maintain the existing relative order of the elements of the data structure. If a sort breaks the existing relations between elements, it is acting counter productively. In some structures can break the ability of the data set to be processed while it is being modified, a crucial property in systems such as databases. An algorithm is said to be adaptive if the algorithm takes advantage of whether or not the list is presorted, nearly sorted or completely random. This is an important metric since we clearly do not want to process the entire list repeatedly if it is already sorted.

3 Applications of Graph Theory

3.1 Graph definitions

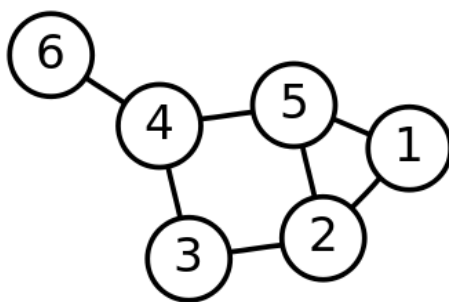


Figure 4: An illustration of a simple connected graph

A graph is an ordered pair of vertices and edges connecting those vertices. The edges can be either undirected or directed, indicating a binary relationship between two vertices. A cycle is a closed walk that begins and ends at the same vertex. A graph is acyclic if it contains no cycle. Because the algorithms we are examining are comparison sorts, there are only two possible relations between elements: less than or equal to, and greater than. Two vertices are connected when there is a path between them, and graph is called connected if there exists a path between any two given nodes, or identically if the graph consists of only one component

3.2 Array

An array is a section of contiguous data in memory, indexed from a starting point. This means that an unsorted array is equivalent to a indexed sequence of unconnected graphs. Each element can be accessed individually in constant time, but there is no inherent connection between elements. Searching for a particular value requires a check on each element which in its worst case will take $O(n)$ time as the entire sequence of vertices is traversed.

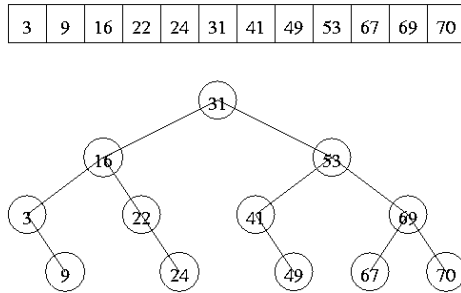


Figure 5: A illustration of a sorted array being treated as a binary search tree

When an array is sorted, it is analogous to transforming this graph into a directed, acyclic graph where each vertices has a max degree of 3, (in degree of 1 and out degree of 2) also sometimes called a binary search tree in computer science terminology. This enables easier searching, since the k th element can be found in constant time $O(1)$ as an offset from the base, and the element with value m can be found in $O(\log(n))$ using a divide and conquer approach, binary search.

3.3 Linked Lists

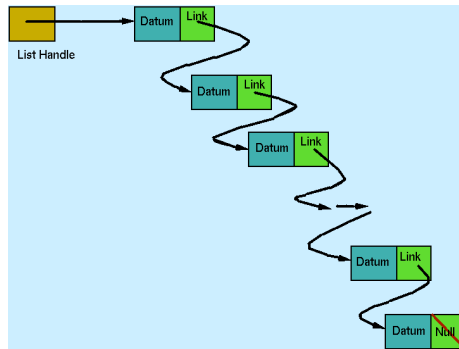


Figure 6: A illustration of a singly linked list data structure. A doubly linked list, as you may imagine, has an additional arrow pointing in the other direction

A doubly linked list is a data structure wherein the data is stored in a series of elements, containing a pointer to the next element in addition to the data itself. This likens an unsorted linked list to an undirected, connected graph with max degree of 2, where the user only has access to the first and last elements of the structure, commonly called the head and tail. Searching for a particular value requires $O(n)$ time for the same reason as array traversal.

When a linked list is sorted, it is analogous to transforming this graph to a directed, connected graph still with max degree 2, with the same user access of the head and tail. This actually does not provide any feasible benefits in searching by value or index, since you must still traverse every element in order to ensure the element is found or not.

4 Bibliography

http://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_man.htm

<http://mathworld.wolfram.com/RecurrenceEquation.html>

http://en.wikipedia.org/wiki/Sorting_algorithm

http://en.wikipedia.org/wiki/Graph_theory