***K*eywords** programming, sequences, sorting

.

Abstract

This paper is an experiment in presenting programming algorithms as recursive functions, without pseudo-code or genuine code. The algorithms presented are the standard basic sorting algorithms with some computational complexity analysis. The style is that of ordinary working mathematics although the case statements of the recursive functions look a lot like pattern matching.

An introduction to sorting algorithms using ordinary algebra

A Preprint

September 8, 2020

1. Introduction

The standard algorithms for sorting lists are usually specified using code or pseudo code and a mix of diagrams and natural language. This note is an experiment in using ordinary elementary algebra as an alternative. Algorithms are defined with (primitive) recursive functions. There are no programming language constructs or "formal methods" of any variety. Control flow involves just case statements, nesting, and recursion. There is no variable assignment, no "loops" or gotos, no statement order. Types are just ordinary mathematical ones. Sorting algorithms are specially suited for this type of experiment because the "state" of the computation is almost entirely captured in the list itself, but the method seems applicable more widely. Algorithms are, as Knuth put it, " recipes" for calculations[Knu97]. The primitive recursive functions used here to specify algorithms are recipes for calculating how to reorder lists.

1.1. Basic concepts

Define a *list* or finite sequence of length n to be a map $\sigma : \{1, \dots n\} \to X$, for some set of values X. Write $Len(\sigma)$ for the length of list σ. The empty list with length zero is written *Nil*. A list is *sorted* if the elements are in order according to some inequality, so that for any $0 < i < j \leq Len(\sigma), \sigma(i) \leq \sigma(j)$. Although I will just write $\leq$, list elements are not always numbers so the "inequality" could be, for example, on the time a sample was obtained, on the lexicographic order of words, on the numerical code for an RNA molecule, or the length of a video combined with its title or anything else[1].

There are many reasons for sorting lists, but a common motivation is to make *searching* faster. Let's start by looking at the (trivial) algorithm for searching a potentially out of order list for a matching element. I'll define an algorithm $F(\sigma, x)$ that compares each element of the list to x and either provides the least i so that $\sigma(i) = x$ or 0 if there is no match. It's common in recursive specifications of these algorithms to have an "outer" part that sets up the arguments to an "inner" component that does most of the work and needs some additional arguments used to track progress through the list.

$$F(\sigma, x) = InF(\sigma, 1, x)$$

[1]The only requirement is that the partial order has the usual properties. That is, for all elements x,y and z in X, the relation is reflexive $x \leq x$, anti-symmetric: $x \leq y$ and $y \leq x$ implies $x = y$, and transitive $x \leq y$ and $y \leq z$ implies $x \leq z$.

The inner component is passed an argument that tells it where on the list to start looking.

$$InF(\sigma, i, x) = \begin{cases} InF(\sigma, i+1, x) & \text{if } 0 < i < Len(\sigma) \text{ and } \sigma(i) \neq x \\ i & \text{if } 0 < i \leq Len(\sigma) \text{ and } \sigma(i) = x \\ 0 & \text{otherwise} \end{cases}$$

.

Computation time can be approximated by the number of comparisons. In the worst case, when x is the last element in the list or is not in the list, it takes $Len(\sigma)$ comparisons to complete the search. If lists are randomly populated, the average case is $Len(\sigma)/2$ comparisons and that number grows linearly with the length of the list. Linear algorithms are pretty good in programming, but we can do better.

If the list is ordered, *binary search* can split the part of the list to be searched in half each time the comparison fails. When searching for x in σ, check the middle element of σ at position $m = \lfloor Len(\sigma)/2 \rfloor$ (rounded down division). If $x \leq \sigma(m)$ then x can only be in the left (lower numbered) half of σ, between 1 and m. Otherwise, x can only be in the right (higher numbered) half of σ, between $m+1$ and $Len(\sigma)$. The outer part of the algorithm sets up the inner component with arguments for the start and length of the subsequence to search: $InB(\sigma, x, i, n)$ searches for x in the sublist starting at i and containing n elements. The middle position of the sublist of n elements starting at position i is $(i-1) + \lfloor n/2 \rfloor$ (when the algorithm starts, $i = 1$ so the middle position is $0 + \lfloor n/2 \rfloor$).

Definition 1.1. Binary Search.

$$B(\sigma, x) = InB(\sigma, x, 1, Len(\sigma))$$

$$InB(\sigma, x, i, n) = \begin{cases} InB(\sigma, x, 1, \lfloor n/2 \rfloor)) & \text{if } n > 1 \text{ and } (x \leq \sigma((i-1) + \lfloor n/2 \rfloor)) \\ InB(\sigma, x, i + \lfloor n/2 \rfloor, \lceil n/2 \rceil) & \text{if } n > 1 \text{ and } (x > \sigma((i-1) + \lfloor n/2 \rfloor)) \\ i & \text{if } n = 1 \text{ and } \sigma(i) = x \\ 0 & \text{otherwise} \end{cases}$$

Binary search takes $\log_2 n$ comparisons, worst case. Each recursion divides the length of the list that still needs to be searched by 2. If the list length is n then k recursions divides it by 2^k so $\log_2(n)$ recursions gets to a list of length 1. For a list of one billion RNA samples, with a comparison time of 1 millionth of a second per comparison, F will take, worst case, about 20 minutes but binary search will take 30 microseconds. If we are searching thousands of samples for matches, then this could be the difference between a program that is useful and one that is not.

In this paper, algorithm analysis will be approximate (or even maybe "hand wavy") because the goal here is intuition, not rigor, and also because while algorithmic analysis is an indispensable tool, these algorithms are only approximations of programs and it's sensible not to analyze the algorithms past the level of detail they provide.

1.2. **Outline**

The paper starts with some list operations and properties, and then the usual elementary sorting algorithms: insertion and selection sort. These are good algorithms, but don't do well for long lists where we need to turn to faster methods like the Quicksort and Mergesort algorithms that come in sections 3 and 4.

2. Sorting and Insertion and Selection sort

Definition 2.1. A list σ is *in order* for some given partial order $\leq$ if and only if $\sigma(i) \leq \sigma(j)$ for all $0 < i \leq j \leq Len(\sigma)$.

The empty list and lists with a single element are always in order.

A *permutation* of a set $\{1, \ldots n\}$ is a map $\pi : \{1, \ldots n\} \to \{1, \ldots n\}$ so that for every $0 < i \leq n$ there is a unique $0 < j \leq n$ with $\pi(j) = i$ (or, equivalently, π is both onto and one-to-one).

Definition 2.2. A list σ' is an *list permutation* of a list σ if and only if $Len(\sigma) = Len(\sigma')$ and there is a permutation π of the set $\{1, \ldots Len(\sigma)\}$ so that $\sigma'(i) = \sigma(\pi(i))$ for all $0 < i \leq Len(\sigma)$.

So the goal is to find a permutation of σ that is in order. From algebra it's known that every permutation on $\{1, \ldots n\}$ can be constructed by composing 2 element cycles that map a pair of the elements to each other and map all other elements to themselves. As a result, every permutation of a list can be constructed by swapping pairs of elements.

Definition 2.3. $swap(\sigma, k, j)$ interchanges the elements at index k and i.

$$\text{If } \sigma' = swap(\sigma, k, j) \text{ then } \sigma'(i) = \begin{cases} \sigma(k) & \text{if } i = j \\ \sigma(j) & \text{if } i = k \\ \sigma(i) & \text{otherwise} \end{cases}$$
$$\text{and } Len(\sigma') = Len(\sigma)$$

Note that $swap(\sigma, i, j)$ is a permutation of σ and, $swap(\sigma, i, i) = \sigma$. Swap could be defined recursively, but in most cases computers can swap two items on a list in a small, fixed number of steps that does not depend on list length.

Swapping is the only operation that is necessary to order a sequence and it is pretty fast, but it often takes a lot of swaps to order a list — depending on how lists are represented in the computer. This paper is, like most introductions to sorting, focused on the case where lists are represented in a straightforward way by just being laid out in consecutive fixed size blocks of memory (arrays). For intuition, let's write lists out: so $\sigma = [xyz]$ is a list of length 3 with $\sigma(2) = y$. For this representation of lists, "indexing" (finding the i^{th} element is so low cost we can generally ignore it. Swapping two elements takes some fixed time that depends on the size of the elements not the length of the list. But inserting an element in the list can be time consuming. Suppose we are given the list of 9 digits [123567894] and then need to put the 4 in the right place. If swapping is the only permitted operation on the list then it takes 5 swaps

$$[123567894] \mapsto [123567849] \mapsto [123567489] \mapsto [123564789] \mapsto [123546789] \mapsto [123456789]$$

It's common in algorithm analysis to ignore the time it takes to setup the algorithm, calculate the additional arguments for the "inner" part and other operations that are done only a few times so that we can focus on the "asymptotic" time complexity of the algorithm: how time to complete increases as the length of the list grows large. This approach turns out to provide a great deal of the most important information — figures 2.1 and 3.1 show that they can predict runtime closely (if we can fit the constants). However, different implementations of even the same algorithm can be 2 or 3 or 100 times slower and have follow the same asymptotic curve. That difference may or may not matter, depending on the size of the lists, the constants, and the application.

If the list is represented by an array in memory making this permutation in one step is not possible[2]. There are alternative data structures, like "linked lists" (see section 6.1) that have different tradeoffs, but this note follows tradition to focus on the common and important cases of lists implemented as arrays.

2.1. Insertion sort

Since lists of length 1 are already in order, any list already has an ordered sublist of length one on the left (low numbered) side. For illustration, we can put a comma in the written out list like: $[x, yxq]$. The comma is just there to remind us we are thinking of this list as being in two parts – it's all the same list. Insertion sort then shuffles the second element into the right place place and then the third and so on. For a list of digits, if $\sigma = [7235]$ then the sorted list starts out with only the element 7, so $[7, 235]$.

$$[7, 235] \mapsto [72, 35] \mapsto [27, 35] \mapsto [237, 5] \mapsto [2357]$$

Insertion sort has the familiar form of an outer part setting up parameters for the inner component that does the work. In this case $InI(\sigma, i, j)$ swaps the element at position i left (to a lower number) until it is in the correct place (or $i = 1$), remembers that it started in position j, and restarts at $j + 1, j + 1$ for the next element. Since the first element is already in order we can start at position 2.

Definition 2.4. Insertion Sort.

$$\mathbf{I}(\sigma) = InI(\sigma, 2, 2).$$

[2]Current (2020) era high end processors have single instructions that can arbitrarily shuffle short memory sequences. See e.g. the Intel "packed shuffle bytes" instruction. So in some cases, we might see a speedup by a constant factor.

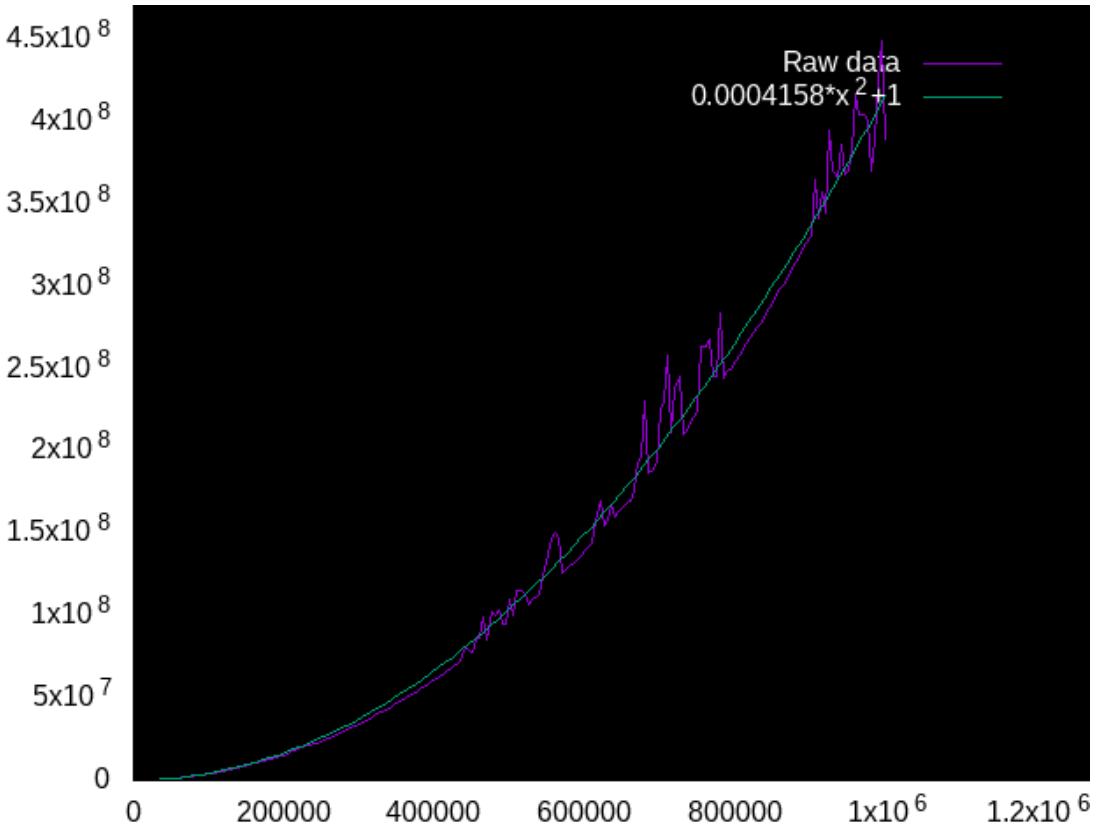

FIGURE 1. Insertion sort time versus list length and quadratic fit

$$InI(\sigma,i,j) = \begin{cases} InI(swap(\sigma,i,i-1),i-1,j) & \text{if } 1 < i \leq j \leq Len(\sigma) \\ & \text{and } \sigma(i-1) > \sigma(i))) \\ InI(\sigma,j+1,j+1) & \text{if } j < Len(\sigma) \\ & \text{and } (i = 1 \text{ or } (i > 1 \text{ and } \sigma(i-1) \leq \sigma(i))) \\ \sigma & \text{otherwise} \end{cases}$$

Worst case for insertion sort is if the list is in reverse order. Moving element 2 into place takes one swap, then element 3 needs 2 and so on up to element $Len(\sigma)$ which needs $Len(\sigma)-1$ swaps. Each swap is preceded by a compare. For a list of length n there are

$$\Sigma_{i=1}^{(n-1)} i = (n^2 - n)/2$$

swaps and the same number of compares. If the list is in random order, that decreases to about $n^2/4$ — in either case the search time is quadratically proportional to the list length. On the other hand, for a list that starts out ordered or nearly ordered, insertion sort will be very efficient. In tests, sorting a list of integers on a current (2020) mediocre desktop computer with insertion sort takes 20 microseconds for 100 elements, 9000 microseconds for 5000 elements, 54000 for 10000 elements and to nearly 2 seconds when we get to 100000 elements . A list of that length initially in reverse order takes 3.5 seconds. Figure 2.1 show a graph of measurements and a fit to a quadratic curve. The raw measurements show some jitter that is mostly caused by the operating system interrupting long lasting computations to let some other process run.

This is *worse*, a lot worse, than the linear time for searching an unordered list. In the common case, however, the list is sorted rarely and searched often. So one relatively time consuming sort operation may be worth it to make a lot of searches fast. Even so, really long lists may be too long to be practically sorted with an quadratic time method like insertion sort (or selection sort, below). Fortunately, we also have some $n \log_2 n$ sorting methods available (sections 3,4).

2.2. **Selection sort**

Selection sort searches the list for the "least" element (according to $\leq$) and then swaps it with the element in position 1. Then it searches for the "least" element starting at position 2 and swaps that element with the element in position 2 and so on until there's nothing left to "select". At that point the list is sorted. As usual, $\mathbf{S}(\sigma)$ sets up an inner algorithm $InS(\sigma, i, j, k)$ where the object is to swap the element at position i with the least element in the sublist starting at position i and going to the end of the list. The index k is the position of the least element on this sublist that has been found so far, and j is the next element to compared to $\sigma(k)$. When all elements from $i \ldots Len(\sigma)$ have been inspected, we swap $\sigma(i)$ and $\sigma(k)$ and start the search again from the next position $InS(swap(\sigma, i, k), i+1, i+2, i)$. When $i > Len(\sigma)$ we are done.

Definition 2.5. Selection sort.

$$\mathbf{S}(\sigma) = InS(\sigma, 1, 2, 1)$$

where:

$$InS(\sigma, i, j, k) = \begin{cases} InS(\sigma, i, j+1, k) & \text{if } i \leq Len(\sigma) \text{ and } j \leq Len(\sigma) \\ & \text{and } \sigma(k) \leq \sigma(j) \\ InS(\sigma, i, j+1, j) & \text{if } i \leq Len(\sigma) \text{ and } j \leq Len(\sigma) \\ & \text{and } \sigma(k) > \sigma(j) \\ InS(swap(\sigma, i, k), i+1, i+2, i+1) & \text{if } i \leq Len(\sigma) \text{ and } j > Len(\sigma) \\ \sigma & \text{otherwise} \end{cases}$$

When $i = 1$ it takes $Len(\sigma) - 1$ steps to find the least element, when $i = 2$ it takes $Len(\sigma) - 2$ steps to find the least element and so on for $\Sigma_{i=1}^{n}(Len(\sigma) - i) = (n^2 - n)/2$ comparisons. On the other hand, there are few swaps — only $Len(\sigma)$ worst case.

In practice, on random lists insertion and selection sort perform about the same, but on lists that start in reverse order, insertion sort is nearly 2 times slower.

3. **Quicksort.**

Quicksort usually takes only $n \log_2 n$ steps to sort a list of length n. The basic idea is to pick a "pivot" value and then scan through the list pushing elements that are "less" than or equal to the pivot to the left and all other others on the right. Then the list is cut into three parts: left, and right with the pivot in the middle.

$$[left, p, right]$$

The left and right are partitioned the same way until we end up with partitions that are in order. A single element list is in order already and a 2 element list will be in order after partitioning. If the pivots are chosen well, each left and right part will be about half the length of the list being partitioned, so after $\log_2 n$ splits the partitions are length 1.

The main algorithm picks a pivot element and swaps it to be at the front of the list and then sets up the inner computation with indexes that are at the beginning and end of the list. I'll discuss the pivot algorithm below, but let's suppose it chooses a pivot element and produces either the index of that element or 0 if one is not needed (for example if the list length is less than 2).

I'm going to show two different quicksort algorithms here. The first one, to me, is more intuitive but needs operations to slice the list and then reglue it. The second method uses more index variables and just relies on swap.

3.1. **Quicksort 1**

To track the partitions, we slice the list: $slice(\sigma, j, k)$ is the sublist of σ begining at position j and ending at position k inclusive.

Definition 3.1. If $\sigma' = slice(\sigma, j, k)$ then

$$\sigma'(i) = \sigma(j + i - 1)$$

and

$$Len(\sigma') = \begin{cases} (k - j) + 1 & \text{if } 0 < j \leq k \leq Len(\sigma) \\ 0 & \text{otherwise} \end{cases}$$

The operator *glue* reverses slicing, if $\sigma = glue(s_1, s_2)$ then $Len(\sigma) = Len(\sigma_1) + Len(\sigma_2)$ and

$$\sigma'(i) = \begin{cases} \sigma_1(i) & \text{if } i \leq Len(\sigma_1) \\ \sigma_2(i - Len(\sigma_1)) & \text{otherwise} \end{cases}$$

Gluing three lists together $glue(\sigma_1, \sigma_2, \sigma_3)$ is just shorthand for $glue(glue(\sigma_1, \sigma_2)), \sigma_3)$. The main work is done in $InQ(\sigma, l, r)$ where initially $l = 1, r = Len(\sigma)$. Until $l > r$ we keep sweeping "smaller" elements left and larger ones right and preserve the properties that

$$\text{for } 0 < i \leq l, \sigma(i) \leq \sigma(1)$$

$$\text{for } r < i \leq Len(\sigma), \sigma(i) > \sigma(1)$$

When, eventually $r < l$ the list is has been completely partitioned: so that for $i \leq l, \sigma(i) \leq \sigma(1)$ and for for $i > l, \sigma(i) > \sigma(1)$. At that point the list can be sliced at l and recursively sorted. We put the pivot in the middle, it doesn't need to be sorted again, and this makes sure that at least some progress is made.

Definition 3.2. Quick Sort.

$$\mathbf{Q}(\sigma) = \begin{cases} \sigma & \text{if } pivot(\sigma) = 0 \\ InQ(swap(\sigma, 1, pivot(\sigma)), 1, Len(\sigma)) & \text{otherwise.} \end{cases}$$

$$InQ(\sigma, l, r) = \begin{cases} InQ(\sigma, l+1, r-1) & \text{if } \sigma(l+1) \leq \sigma(1) \text{ and } \sigma(r) > \sigma(1) \\ & \text{and } l \leq r \\ InQ(\sigma, l+1, r) & \text{if } \sigma(l+1) \leq \sigma(1) \text{ and } \sigma(r) \leq \sigma(1) \\ & \text{and } l \leq r \\ InQ(\sigma, l, r-1) & \text{if } \sigma(l+1) > \sigma(1) \text{ and } \sigma(r) > \sigma(1) \\ & \text{and } l \leq r \\ InQ(swap(\sigma, l+1, r), l+1, r-1)) & \text{if } \sigma(l+1) > \sigma(1) \text{ and } \sigma(r) \leq \sigma(1) \\ & \text{and } l \leq r \\ glue(\mathbf{Q}(slice(\sigma, 2, l)), & \\ \quad slice(\sigma, 1, 1), & \\ \quad \mathbf{Q}(slice(\sigma, l+1, Len(\sigma)) & \\ \quad) & \text{if } l > r \end{cases}$$

Quicksort is designed so that lists can be split and glued (concatenated) just by changing perspective. The lists are adjacent and never move (although their elements are swapped around). For this reason, we can ignore splits and concatenation in analyzing run time (see section 3.2). If the pivots are chosen well, the whole process can take around $n \log_2 n$ steps where $n = Len(\sigma)$ to complete quicksort. In this case, the list is repeatedly split in half (or something close) so that after $\log_2 Len(\sigma)$ splits the sublists are 1 or 2 elements. Think of this phase of the algorithm as building layers of splits - first the list is n elements, then there is a layer of two lists of (we hope) something like $n/2$ elements each, then a layer of four lists of hopefully $n/4$ elements each . The sweeps to partition take no more steps than the length of the partitions - which sums to n for each layer. If the pivots are chosen badly the process can take n^2 steps. Imagine, for example that the pivot is the maximum element of σ, so that the left partition contains all of σ except the pivot. In practice, maybe surprisingly, even simple methods of chosing the pivot are almost always good enough. There are many variations on quicksort in terms of how the pivot is chosen and where it stays during partition. Sometimes, an average of a couple of randomly chosen elements is used as the pivot. In my test implementation, I used $pivot(\sigma) = \sigma(\lfloor Len(\sigma)/2 \rfloor)$.

While insertion sort is up to over 6 seconds on a randomly ordered list of 200,000 elements, quick sort is still 16 milliseconds — somewhere around 375 times faster. At 300000 elements, insertion sort is up to 14 seconds, 538 times slower than quicksort. By the time we get to a list of 1 million elements, insertion sort is 3 1/2 minutes and quick sort is 98 milliseconds. Figure 3.1 has the same graph we saw above for insertion sort - $n \log_2 n$ is pretty close to linear until n gets enormous.

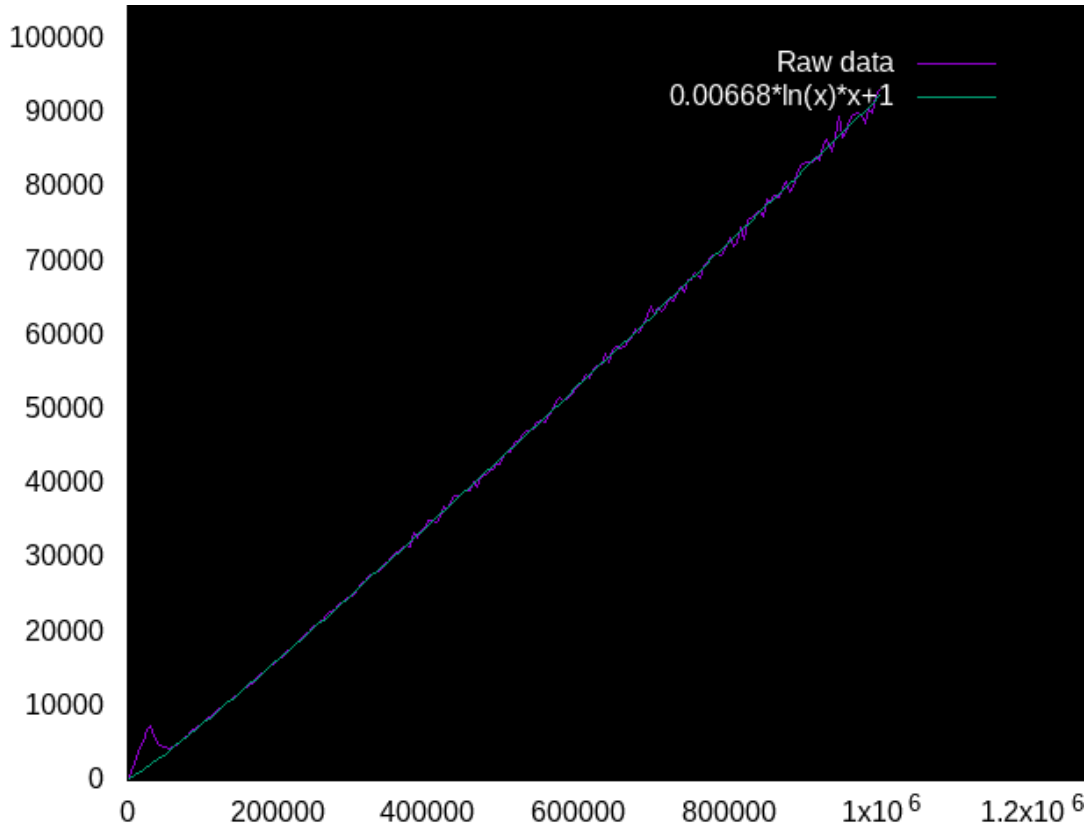

FIGURE 2. Quick sort runtime versus list length

3.2. **Quicksort 2**

This version of quicksort doesn't split or concatenate the list so it must carry around two additional indexes, a and b which give the starting index and length, respectively, of the sublist being partitioned. These are initially the same as l and r and are not changed until the two partitions have to be recursively partitioned, at which point they are adjusted to the lengths of the two partitions.

$$\mathbf{Q_2}(\sigma) = \begin{cases} \sigma & \text{if } pivot(\sigma) = 0 \text{ or } Len(\sigma) < 2 \\ InQ_2(swap(\sigma, 1, pivot(\sigma)), 1, Len(\sigma), 1, Len(\sigma)) & \text{otherwise.} \end{cases}$$

$$
\begin{aligned}
&InQ_2(\sigma, l, r, a, b) = \\
&\begin{cases}
InQ_2(\sigma, l+1, r-1, a, b) & \text{if } a \neq b \text{ and } l \leq r \\
& \text{and } \sigma(l) \leq \sigma(1) \text{ and } \sigma(r) > \sigma(1) \\
InQ_2(\sigma, l+1, r, a, b) & \text{if } a \neq b \text{ and } l \leq r \\
& \text{and } \sigma(l) \leq \sigma(1) \text{ and } \sigma(r) \leq \sigma(1) \\
InQ_2(\sigma, l, r-1, a, b) & \text{if } a \neq b \text{ and } l \leq r \\
& \text{and } \sigma(l) > \sigma(1) \text{ and } \sigma(r) > \sigma(1) \\
InQ_2(swap(\sigma, l, r), l+1, r-1, a, b) & \text{if } a \neq b \text{ and } l \leq r \\
& \text{and } \sigma(l) > \sigma(1) \text{ and } \sigma(r) \leq \sigma(1) \\
InQ_2(& \\
\quad InQ_2(swap(\sigma, 1, l), a, l-1, a, l-1), & \\
\quad l+1, b, l+1, b) & \text{if } a \neq b \text{ and } l > r \\
\sigma & \text{if } a = b
\end{cases}
\end{aligned}
$$

4. **Mergesort**

Quicksort has worst case much worse than $n \log n$, but "merge sort" is never worse than $n \log n$ comparisons. Merge sort has a different tradeoff though. Mergesort is based on the observation that merging two ordered lists into a single ordered list can be carried out in time that is linearly proportional to the sum of the lengths. Start with two sorted sublists to merge and an empty list that will become the merged list. Compare the first elements of the two lists, remove the lesser element from the start of its list and append that element to the end of the merged list. Repeat until the two lists are emptied.

Write $x \cdot \sigma$ for the sequence obtained by *appending* x to σ so $[abc] \cdot x = [abcx]$ (definition 6.3). Then assuming σ_1 and σ_2 are in order $Mrg(Nil, \sigma_1, \sigma_2)$ will merge the second two sequences by appending elements to the nil sequence until both subsequences have been fully merged. As usual, there is an inner algorithm with indexes tracking progress: $InMrg(\sigma_0, \sigma_1, i, \sigma_2, j)$ is done when $i > Len(\sigma_1)$ and $j > Len(\sigma_2)$. Otherwise it will append either $\sigma_1(i)$ to σ_0 or append $\sigma_2(i)$ to σ_0, increment i or j depending — until both lists have been copied to σ_0) in order.

Definition 4.1.

$$
\begin{aligned}
&Mrg(\sigma_1, \sigma_2) = InMrg(Nil, \sigma_1, Len(\sigma_1), Len(\sigma_2)) \\
&InMrg(\sigma_0, \sigma_1, i, \sigma_2, j) = \begin{cases}
InMrg(\sigma_0 \cdot \sigma_1(i), \sigma_1, i+1, \sigma_2, j) & \text{if } (Len(\sigma_1) \geq i \text{ and } Len(\sigma_2) \geq j \\
& \quad \text{and } \sigma_1(i) \leq \sigma_2(j)) \\
& \text{or } Len(\sigma_1) \geq i \text{ and } Len(\sigma_2) < j \\
InMrg(\sigma_0 \cdot \sigma_2(j), \sigma_1, i, \sigma_2, j+1) & \text{if } (Len(\sigma_1) \geq i \text{ and } Len(\sigma_2) \geq j \\
& \quad \text{and } \sigma_1(i) > \sigma_2(j)) \\
& \text{or } Len(\sigma_2) \geq j \text{ and } Len(\sigma_1) < i \\
\sigma_0 & \text{if } Len(\sigma_1) < i \text{ and } Len(\sigma_2) < j
\end{cases}
\end{aligned}
$$

So

$$
\begin{aligned}
InMrg(Nil, [135], 3, [24], 2) &= InMrg([1], [135], 2, [24], 1) \\
&= InMrg([12], [135], 2, [24], 2) \\
&= InMrg([123], [135], 3, [24], 2) \\
&= InMrg([1234], [135], 3, [24], 3) \\
&= InMrg([12345], [135], 4, [24], 3) \\
&= [12345]
\end{aligned}
$$

Ordered merge takes $Len(\sigma_1) + Len(\sigma_2)$ steps to complete. Each step takes one comparison and one $\cdot$ and we can make appending fast by using some additional memory (see below).

Merge sort is then simple because lists of length 1 are already in order. Keep splitting the list until we have sublists of length 1, merge those, merge the result and so on, until we've remerged everything.

Definition 4.2. Mergesort.

$$\mathbf{M}(\sigma) = \begin{cases} \sigma & \text{if } Len(\sigma) < 2 \\ Mrg(\mathbf{M}(\sigma_1), \mathbf{M}(\sigma_2)) & \text{for } \sigma_1 = slice(\sigma, 1, \lfloor Len(\sigma)/2 \rfloor) \\ & \text{and } \sigma_2 = slice(\sigma, \lfloor (Len(\sigma)/2 \rfloor) + 1, Len(\sigma)) \\ & \text{otherwise} \end{cases}$$

Ordered merge takes $Len(\sigma_1) + Len(\sigma_2)$ steps to

For a list of length n, if each merge phase takes only $n = Len(\sigma)$ steps, then mergesort has worst case $n \log_2 n$ compute time.

4.1. Implementation considerations

For lists that are implemented as arrays in memory, inserting elements can be time consuming. For example, suppose we have two lists of alphabetic characters $[aceg, bdf]$ and want to merge. The first step is easy: $[a, ced, bdf]$, just a change of perspective. But to get to the next step $[ab, ceg, df]$ would take 3 swaps. Merging in alphabet order $[efghijklmnop, abc]$ takes 36 swaps, if I counted correctly. The number of swaps may be n^2 if n is the length of the left hand ordered sublist. That would not be useful. The trick in merge sort is to provide merge with an additional "scratch" sublist which we can think of as filled with blanks _. If the blank list is adjacent to the two lists that are to be merged [_____, 1357, 246] append can be implemented as a single swap[3].

$$[\_\_\_\_\_, 1357, 246] \mapsto [1\_\_\_\_\_, \_357, 246] \mapsto [12\_\_\_\_\_, 357, \_46] \dots$$

The scratch list doesn't need to be actually blank, its contents don't matter. When the lists are merged, the sublist on the left is what is retained. When the algorithm is coded up, each phase of merging can fill up a scratch sublist and then start the next phase switching to use the two earlier source sublists as the scratch list.

In most cases, mergesort is fine, because computers have a lot of memory for the extra list. There can be problems for really huge lists, or in embedded systems, or when the possibility of sorting failing because the system is out of memory is not acceptable (Rich Felker explained this to me). In those cases pick a different sort method or a different representation of lists.

5. Bucket and Radix sort

TBD?

6. Tools

It's common to represent lists in computers as arrays — which are just sequences of adjacent memory blocks of fixed length.

$$\sigma = [abcdefg]$$

A "swap"operation just interchanges two blocks

$$swap(\sigma, 1, 3) = [cbadefg]$$

But inserting or deleting an element may depend on the length of the list: these may require repeated swapping like in the insertion sort algorithm.

[3] In fact, we don't even need a "swap" here: we just need to assign a value to a position in the destination list and the source element can remain unchanged.

6.1. **Linked lists**

Linked list representations are outside the scope of this note, but make indexing the list more expensive (order of $Len(\sigma)$) for the advantage of moving elements around faster. Merging two in order linked lists does not need additional storage because operations like

$$[abcdef] \mapsto [fabcde]$$

are constant time. A linked list can be represented by a triple (h, σ_d, σ_l) where $Len(\sigma_d) = Len(\sigma_l) = n$ each element of σ_l is a number between 1 and n, inclusive, and h is some number between 1 and n inclusive. Define $g(1) = h$ and $g(n+1) = \sigma_l(g(n))$. Then σ so that $\sigma(i) = \sigma_d(g(i))$ is the standard list implemented by the triple. For example, if:

$$\sigma_l = [35401], h = 2, \sigma_d = [abcde]$$

then

$$\sigma(2) = \sigma_d(g(2)) = \sigma_d(\sigma_l(g(2))) = \sigma_d(\sigma_l(g(1))) = \sigma_d(\sigma_l(h)) = \sigma_d(5)$$

6.2. **Tools**

Definition 6.1. Splitting

$$\begin{array}{l} \text{If } \sigma_1 = slice(\sigma, j, k) \\ \text{and } 0 < j \leq k \leq Len(\sigma) \\ Len(\sigma_1) = (k - j) + 1 \\ \sigma_1(i) = \sigma(i + j) \end{array}$$

Definition 6.2. Glueing (Concatenation).

$$\begin{array}{l} \text{if } \sigma = glue(\sigma_1, \sigma_2, Len(\sigma) = Len(\sigma_1) + Len(\sigma_2)) \\ \sigma(i) = \begin{cases} \sigma_1(i) & \text{if } 0 < i \leq Len(\sigma_1) \\ \sigma_2(i - Len(\sigma_1)) & \text{if } Len(\sigma_1) < i \leq Len(\sigma) \end{cases} \end{array}$$

Note that if the two lists to be glued together are adjacent, gluing is just a perspective change.

Definition 6.3. append

$$\begin{array}{l} \text{If } \sigma' = \sigma \cdot x \text{ then} \\ Len(\sigma') = Len(\sigma) + 1 \\ \text{and } \sigma'(i) = \begin{cases} x & \text{if } i = Len(\sigma') \\ \sigma(i) & \text{otherwise} \end{cases} \end{array}$$

$$\begin{array}{l} M(\sigma) = m(appendblank(\sigma), 1, 1, 1, N, N, 2N + 1, 1, p, p) \\ m(\beta, s, p, i, j, k, l, d, q, r) \\ = \begin{cases} m(set(\beta, k + d, \beta(i)), s, p, i + d, j, d, q - 1, r) & \text{if } k \neq l \text{ and } q > 0 \text{ and } (r = 0 \text{ or } \beta(i) \leq \beta(j)) \\ m(set(\beta, k + d, \beta(j)), s, p, i, j + d, d, q, r - 1) & \text{if } r > 0 \text{ and } (q = 0 \text{ or } \beta(i) > \beta(j)) \\ m(\beta, s, p, i, j, k, l, d, q, r) & \text{if } r = 0 \text{ and } q = 0 \end{cases} \end{array}$$

References

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Third. Reading, Mass.: Addison-Wesley, 1997.

Austin Texas.

Email address: victor.yodaiken@gmail.com