How inefficient can a sort algorithm be?

Miguel A. Lerma

November 2, 2012

Abstract

Here we find large lower bounds for a certain family of algorithms, and prove that such bounds are limited only by natural computability arguments.

1 Introduction

Here we study algorithms intended to sort a list of n integers. It is well known that optimal sort algorithms such as *mergesort* have a run time $\Theta(n \log n)$ (see [2]). *Bublesort*, with a worst-case run time of $\Theta(n^2)$, is considered "inefficient". But, are there any sort algorithms that perform even worse?

This paper is inspired on a discussion found in Internet about inefficient sort algorithms. The summary of such discussion can be found (at the time of this writing) in the following page:

`http://home.tiac.net/~cri_d/cri/2001/badsort.html`

That discussion contains details on how to design sort algorithms with larger than quadratic run time. The record holder for such kind of inefficient algorithm among the ones mentioned in that page is called *EvilSort*, with a run time $\Omega((n^2)!)$. Here we show how to break that record and produce basically boundless inefficient sort algorithms.

2 A hierarchy of inefficient sort algorithms

Before we start stepping up the slope of inefficiency we want to make sure that we don't do it in a trivial way, such as inserting useless loops just with the purpose of "wasting" time by adding delays in an artificial way. The sort algorithms described here will always contain only steps directed to the final goal of obtaining a sorted list of elements.

The basic task of our algorithms will be to sort a list of integers $L = [a_1, a_2, \ldots, a_n]$ in increasing order. The size of the input will be given by the number n of elements in the list, and time will be measured by the number of integer comparisons performed.

A particularly inefficient way to sort the given list of integers consists of generating a random permutation of it and check if such permutation contains the elements correctly sorted. That is the so called *bogosort* algorithm (see e.g [1]), performing asymptotically $(e-1)n!$ integer comparisons and $(n-1)\cdot n!$ swaps in average. This kind of algorithm however has several problems. First, it requires a random generator. Then, the best case run time is very low, just $n-1$ integer comparisons and no swaps if the given list is already sorted. Finally, the worst case run time is unbounded.

A variation of *bogosort* that eliminates randomness consists of generating all $n!$ permutations of the given list and then search for the one that contains the elements correctly sorted. This keeps the average run time in $\Omega(n!)$ integer comparisons, but still produces a low $n-1$ number of comparison in the best case.

In order to keep the best case run time high, we will change the strategy to find the correctly sorted permutation. Instead of performing a linear search on the list of permutations, we will *sort* all $n!$ permutations in lexicographical order, an return the first one of them. For instance, if the given list is $L = [2, 3, 1]$, we generate a list of lists consisting of all possible permutations of the given list:

$$P = [[2,3,1],[2,1,3],[3,1,2],[3,2,1],[1,2,3],[1,3,2]]$$

and then sort them in lexicographical order:

$$P_{\text{sorted}} = [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]\,.$$

The first element of this list of integer lists is the sorted integer list $[1, 2, 3]$.

The sorting of the list of integer lists can be performed with any standard algorithm such as *bublesort*, which runs in $\Theta(n^2)$ time. The lexicographical order of integer lists is defined so that $L_1 <_{\text{lex}} L_2$ precisely when the first index $k \in [1, \ldots, n]$ for which they differ verifies $L_1[k] < L_2[k]$. So, comparing two integers lists requires at least one integer comparison, and the total time (number of integer comparisons) required to sort $n!$ permutations of n elements in lexicographical order using bublesort will be $\Omega((n!)^2)$.

That is still less than the run time of the EvilSort algorithm mentioned above, but soon we will see how to do better—I mean, worse.

One obvious way consists of replacing bublesort with another instance of the algorithm just described, i.e., instead of using bublesort to sort the $n!$ permutations of the original list of n integers, generate the $(n!)!$ permutations of the list of $n!$ permutations, and then sort lexicographically the list of permutations of permutations. The first element will be a list of permutations of integers lists, and the first element of it will be the original list of n integers sorted in increasing order. The number of integer comparisons performed will be now $\Omega(((n!)!)^2)$.

This finally breaks the record hold by EvilSort, but we want go further, break our own record, and in fact any record ever set by anybody in the past or in the future. To do so we can repeat what we just did, i.e., replace the final application of bublesort with an instance of the latest version of the kind of algorithm described here, so that the run time will keep growing to $\Omega((((n!)!)!)^2)$, $\Omega(((((n!)!)!)!)^2)$, and so on, but how far can we go?

In the next section we will develop these ideas in a more precise way, and also will look at what the limit of this strategy might be. In particular we will answer the following question: given any (rapidly) increasing computable function $f : \mathbb{N} \to \mathbb{N}$, is there a sort algorithm with run time $\Omega(f(n))$?

3 Worstsort: the final solution

As stated, our algorithm will take as its input a list `L` with n integer elements and return the same list with its elements sorted in increasing order. In the intermediate steps we will be handling general lists whose elements can be of any type, in particular the elements of a list can also be lists.

The following is assumed about lists:

1. The number of elements of a list `L` is available and represented as `length(L)`.
2. Elements are indexed with an index that runs from 1 to `length(L)`.
3. It is possible to access/retrieve/modify the element at a particular index without affecting any other elements. In particular it is possible to swap two elements of a list.
4. It is possible to insert an element at a particular index. The indices of higher elements at that are increased by 1.
5. It is possible to remove an element at a particular index. The indices of higher elements at that are decreased by 1.
6. It is possible to append two lists. Here we represent `L1 + L2` = result of appending list `L1` and `L2`, e.g. $[a, b, c] + [d, e] = [a, b, c, d, e]$.

The usual assignment operator '`:=`' between lists makes the list in the left hand side identical to the list on the right hand side, i.e., `L1 := L2` makes `L1` into another name for list `L2`. If after the assignment list `L2` is modified, list `L1` is also modified because they in fact represent the same list.

We can make a copy of a list in such as way that the original list and its copy have the same elements, but remain different lists, so that changes in the copy do not affect the original list. The following is an implementation of a list copy function (using Pascal-like pseudocode):

```
1: procedure copy(A,B)
2:   for i := 1 to length(A) do
3:     B[i] := A[i]
4:   od
5: end
```

The length and indexing of `B` are adjusted to fit those of `A`.

Variables are supposed to be local to the procedure where they occur, and created as needed if they do not exist. The types of variables will be 'integer', 'list of integers', 'list of list of integers', and so on. The type of a variable

is determined by context. Integer arguments are passed by value, and lists are passed by reference.

Since the algorithms to be precisely defined here will require not only integer comparison, but also lexicographical comparisons of list of integers, of lists of list of integers, etc., we need a function `lt` that is able to perform that operation to any level. The following code fulfills this requirement:

```
 1: procedure lt(A,B) // is A less than B?
 2:   if type(A) = integer then // the arguments are integers
 3:     return(A<B) // return integer comparison
 4:   else
        // otherwise the arguments are lists,
        // perform lexicographic comparison
 5:     for k := 1 to length(A) do
 6:       if lt(A[k],B[k]) then
 7:         return(true) // A[k] < B[k], hence A < B
 8:       elsif lt(B[k],A[k]) then
 9:         return(false) // A[k] > B[k], hence A > B
10:       else
            // otherwise A[k] = B[k], keep going
11:       fi
12:     od
13:     return(false) // all elements are equal, hence A = B
14:   fi
15: end lt
```

The following is the version of `bublesort` that we will be using here. The algorithm modifies the original list `L`, and performs $\Theta(n^2)$ '`lt`' comparisons.

```
1: procedure bublesort(L)
2:   for i:=2 to length(L) do
3:     for j:=1 to length(L)-i+1 do
4:       if lt(L[j+1],L[j]) then
5:         swap(L[j],L[j+1])
6:       fi
7:     od
8:   od
9: end bublesort
```

The procedure `permutations` takes a list as its argument and returns a list of lists with all permutations of the elements of the original list. The following code is one among many possible ways of generating all the permutations of a list `L`:

```
 1: procedure permutations(L)
 2:   if length(L) =< 1 then
        // in this case there is only one permutation
 3:     copy(L,L0) // this is to preserve original list
 4:     return([L0]) // return the only permutation
 5:   else
 6:     P := [] // the list of permutations is initially empty
 7:     for i:=1 to length(L) do
 8:       copy(L,L1) // make copy of original list
 9:       remove(i,L1) // remove i-th element from the copy
10:       P0 := permutations(L1) // generate its permutations
          // put removed element at the beginning
          // of each permutation of L1 and add the
          // result to the list of permutations
11:       for j:=1 to length(P0) do
12:         P := P + [[L[i]] + P0[j]]
13:       od
14:     od
15:     return(P)
16:   fi
17: end permutations
```

The following is the code for the multilevel version of the sort algorithm described in section 2:

```
 1: procedure multilevelsort(L,k)
 2:   if k = 0 then // last level, just perform bublesort
 3:     bublesort(L)
 4:   else
 5:     P := permutations(L) // generate permutations
 6:     multilevelsort(P,k-1) // sort them lexicographically
 7:     copy(P[1],L) // copy first element into L
 8:   fi
 9: end multilevelsort
```

For $k = 0$, `multilevelsort` performs just bublesort on the given list of elements, run time $\Omega(n^2)$. For $k > 0$, `multilevelsort` performs k recursive self-calls before using bublesort. Its run time is $\Omega(((\cdots(n!)\cdots!)!)^2)$, with k nested factorials. Using the *multifactorial* notation $n!^{(k)}$ = take the factorial of n k times, then the lower bound for the run time of `multilevelsort` will be $\Omega((n!^{(k)})^2)$.

We finally answer the question of how inefficient a sort algorithm can be. To do so we define the following sort algorithm, that takes a list of integers L, and an increasing computable function $f : \mathbb{N} \to \mathbb{N}$ as its arguments:

```
1: procedure worstsort(L,f)
2:   multilevelsort(L,f(length(L)))
3: end worstsort
```

The run time for this algorithm is now $\Omega((n!^{(f(n))})^2) \geq \Omega(f(n))$, showing that a sort algorithm can be made as inefficient as we wish, with its run time growing at least as fast as any given fix computable function. Since worstsort is itself computable, the growth rate of its run time will still be asymptotically bounded above by rapidly growing uncomputable functions such as a *busy beaver* (which is known to grow faster than any computable function—see [3]). But given any fix rapidly growing *computable* function, we can make the run time of worstsort grow faster just by feeding that function as its second argument.

4 Conclusion

We have shown that there is no computable limit to the inefficiency of a sort algorithm, even when respecting the rule of not using useless loops and delays unrelated to the sorting task. The run time of such algorithm can growth at least as fast as any given fix computable function.

References

[1] H.xi Gruber, M. Holzer, and O. Ruepp. Sorting the slow way: an analysis of perversely awful randomized sorting algorithms. *Lecture Notes in Computer Science*, 4475:183–197, 2007.

[2] Donald Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 2nd edition, 1998.

[3] Tibor Radó. On non-computable functions. *Bell System Tech. J.*, 41:877–884, 1962.