# Pessimal Algorithms and Simplexity Analysis

ANDREI BRODER and JORGE STOLFI
*DEC Systems Research Center*
*130 Lytton Avenue, Palo Alto CA 94301*

**Abstract:** The twin disciplines of Pessimal Algorithm Design and Simplexity Analysis are introduced and illustrated by means of representative problems.

## 1. Introduction

Consider the following problem: we are given a table of $n$ integer keys $A_1, A_2, \ldots, A_n$ and a query integer $X$. We want to locate $X$ in the table, but we are in no particular hurry to succeed; in fact, we would like to delay success as much as possible.

We might consider using the trivial algorithm, namely test $X$ against $A_1, A_2$, etc. in turn. However, it might happen that $X = A_1$, in which case the algorithm would terminate right away. This shows the naïve algorithm has $O(1)$ best-case running time. The question is, can we do better, that is, worse?

Of course, we can get very slow algorithms by adding spurious loops before the first test of $X$ against the $A_i$. However, such easy solutions are unacceptable, partly because any fool can see that the algorithm is just wasting time (which would be very embarrassing to its author), but mostly because we need something to fill the rest of this paper with. Therefore, we must look for an algorithm that does indeed progress steadily towards its stated goal even though it may have very little enthusiasm in (or even a manifest aversion to) actually getting there.

We can get an algorithm that satisfies this criterion and is much better (that is, worse) than the naive one if we keep the table $A$ sorted in ascending order. Then we can use the *reluctant search* procedure below:

```
procedure research ($X$, $i$, $j$: integer): integer =
  { Result is the index $k$ such that $A_k = X$, or $-1$ if no such $k$ exists. }
  if $i > j$ then return $-1$ fi
  if $i = j$ then if $X = A_i$ then return $i$ else return $-1$ fi
  $m \leftarrow \lfloor \frac{i+j}{2} \rfloor$
  if $X \leq A_m$ then
    $k \leftarrow$ research ($X$, $m+1$, $j$)
    if $k = -1$ then return research($X$, $i$, $m$) else return k fi
  else
    $k \leftarrow$ research ($X$, $i$, $m$)
    if $k = -1$ then return research ($X$, $m+1$, $j$) else return k fi
  fi
erudecorp
```

The number of probes performed by this algorithm is independent of $X$ and the $A_i$, and is given by the recurrence

$$T(n) = \begin{cases} 1 + T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

which has the solution[†]

$$T(n) = \Theta\left(n + \frac{n}{1+n} \frac{\log(n+1)^{3.14159+}}{\log n \log\log n}\right).$$

(The proof will be given in a future report.) This represents a desimprovement by a factor of $n$ over the naive algorithm. Observe that the lack of enthusiasm of the reluctant search algorithm is not at all evident from its behavior, since it performs a $X = A_i$ test every $O(1)$ operations, never repeats a test, and stops as soon as it finds the answer. Few search algorithms, honest or not, can match this performance.

## 2. Generalizations

The ***research*** procedure is a prototypical example of an entirely new branch of Computer Science, the design and analysis of *reluctant algorithms*. Intuitively, a reluctant algorithm for a problem $P$ is one which wastes time in a way that is sufficiently contrived to fool a naive observer. We can make this concept mathematically precise by saying that $A$ is a reluctant algorithm for $P$ iff

$$(\exists W) \bigwedge_{o \in N} w(A, t, W) \wedge \mathcal{F}^*(W, o), \tag{1}$$

where $N$ is the set of all naive observers, $t$ is time, $w(A, t, W)$ is the predicate "$A$ wastes $t$ in the way $W$", and $\mathcal{F}^*(W, o)$ is the boolean test "$W$ is sufficiently contrived to fool $o$".

In the study of reluctant algorithms, the performance of an algorithm $A$ is better expressed by its *inefficiency* or *best-case time*, the minimum (as a function of $n$) over all inputs of size $n$ of the running time of $A$. The *simplexity* of a problem is the maximum inefficiency among the algorithms that solve $P$. An algorithm is said to be *pessimal* for a problem $P$ if the best-case inefficiency of $A$ is asymptotically equal to the simplexity of $P$.

Reluctant algorithms have plenty of important practical applications. For example, the reluctant search algorithm is particularly applicable to the case of real keys (real not in the mathematical sense, but rather in the sense that they can be used to open doors and drawers). The reluctant search algorithm is the only one known so far that accurately emulates the behavior of bundles of such keys [SB].

## 3. Path problems in pleasant graphs

Table search can be viewed as a special case of the following more general problem. We are given a "maze", i.e. an undirected graph $G$ with $n$ nodes, and an "entry" node $u$ in it. Our task is to find a path from $u$ to a specified "exit" node $v$, by walking on the maze one edge at at time. In the spirit of classical Analysis of Algorithms, we would immediately think of using one of the efficient shortest-path or graph traversal methods. However, suppose the maze is actually quite agreeable, so much that we wouldn't mind spending a few extra cycles in the search for $v$, in fact we vaguely hope, nay, decidedly wish, that the search will take as long as possible, and even though our sense of duty prevents us from giving up the search altogether, we are not that insensitive to the primeval necessities of our human nature, and besides what is wrong with taking a more relaxed attitude to the problem, as long as we do what we are supposed to do, since

[†] We use "log" for logarithms in base two, and "ln" for natural logarithms.

we have always been told that haste makes waste, and no one needs to be perfect anyway, and so forth. With these assumptions, the problem falls squarely within the domain of our theory.

This problem has been extensively studied by graph theorists, who call it the *sloppiest path problem.* The important branch of operations research that goes by the name of *anemic programming* is entirely devoted to the study of inefficient methods for solving this problem. What do we know about its simplexity? Early on it was shown by R. Wagner [Wa] that if we have no information about the location of $v$, the best-case running time may be as low as $O(1)$: at every single step — even the very first one — we risk stumbling upon $v$ and falling out of the maze, no matter how much we would like to avoid it. However, H. Homer [Ho] showed that, if the graph is embedded in the plane (or in a flat globe), and we are given an oracle that reveals the general direction of our goal, it is possible to delay getting there until after most or all of the graph has been traversed. In fact, in this situation the delay is limited not by the inherent simplexity of the problem, but by its monotonicity.[†] Homer's algorithm has $\Omega(n)$ inefficiency, and this is a lower bound for the simplexity of the sloppiest path problem.

The reluctant search method and Homer's sloppiest-path algorithm are both based on the same idea, variously known as the *method of reverse gradient* or *method of feeblest descent.* We mention in passing that another important paradigm for reluctant algorithm design was described by Homer in the same paper. It was given by its inventor the colorful name of *Penelope's stratagem*, and relies in the use of a `for` loop whose step oscillates between positive and negative values at each iteration. Unfortunately, this technique (which is presently called the *backtrack method*), has become so well-known that even naive observers can spot it at first sight, and is now of historical interest only.

## 4. Backwards-first search

A somewhat similar problem is that of enumerating all $n$ vertices of a connected graph $G$ in a systematic fashion. This problem has been extensively studied in the framework of classical theory of algorithms, and it is usually solved by the well-known depth-first [V1] or breadth-first [V2] algorithms, which exhibit $\Omega(n)$ best-case time.

This was for a long time thought to be an upper bound to the simplexity of the problem, but on October 4, 1984 at 2:17 p.m. the reluctant algorithmics community was shaken by the discovery of a search strategy exhibiting $\Omega(n^2)$ inefficiency for an important class of graphs. The *backwards first searching* method, as it was called by its inventor, is described below. Like its predecessors, it is best thought of as a method for assigning to the vertices $v_1, v_2, \ldots, v_n$ of $G$ the integer labels $\lambda(v_1), \lambda(v_2), \ldots, \lambda(v_n)$, in the range 1 to $n$. The algorithm is expressed by the recursive procedure *bwfs* below. The procedure assumes all labels $\lambda(v)$ are initially zero; the recursion is started by the call *bwfs*$(v_1, 1)$.

```
procedure bwfs ($v$: vertex, $i$: integer) =
  $\lambda(v) \leftarrow i$
  for each neighbor $u$ of $v$ do
    if $0 < \lambda(u) < i$ then bwfs ($u$, i) fi
  rof
  for each neighbor $u$ of $v$ do
    if $\lambda(u) = 0$ then bwfs ($u$, $i+1$) fi
  rof
erudecorp
```

We leave to the reader as an enlightening exercise the task of proving the correctness of this algorithm, and establishing that its inefficiency is indeed $\Theta(n^2)$ for straight line graphs. Its

[†] Also known as *boredom*.

inefficiency on general graphs is an open problem, but it seems that it never does worse (that is, better) than $O(n\sqrt{n})$.

The *backwards first numbering* of the vertices of a graph is by definition the labels $\lambda(v)$ assigned by this algorithm. Like the depth-first and breadth-first numberings, this one has several interesting properties. Due to lack of space, we will mention only a couple of them here. If the edges are arbitrarily oriented so as to produce an acyclic graph, then $\lambda(\text{head}(e)) \geq \lambda(\text{tail}(e))$ for every edge $e$, or $\lambda(\text{head}(e)) \leq \lambda(\text{tail}(e))$ for every $e$. Furthermore, if the maximum degree of the graph is $d$, for any pair of adjacent vertices $u, v$ we will have $|\lambda(u) - \lambda(v)| \leq d \log \min\{\lambda(u), \lambda(v)\}$. These and other properties make the backwards-first numbering to be of prime combinatorial importance.

## 5. Slowsort

No other problem shows more clearly the power and elegance of reluctant algorithmics than the sorting of $n$ given numbers. This problem has a long and rich history, whose beginnings can be traced far back in the past, almost certainly to a time before the establishment of reluctant algorithmics as a recognized discipline in the second half of last Wednesday. Thanks to the efforts of many industrious pioneers, the inefficiency of sorting algorithms was steadily raised in those heroic days from the modest $\Omega(n \log n)$ of the merge sort algorithm to the $\Omega(n\sqrt{n})$ of Shell's sort (with appropriate increments), to the $\Omega(n^2)$ of bubble sort, and finally to the clever $\Omega(n^3)$ sorting routine recently described by Bentley [B].

One of the most important results of modern simplexity theory is the proof that the sorting problem can be solved in $\Omega(n^{\log(n)/(2+\varepsilon)})$ best-case time. This was the first problem to be shown to have non-polynomial simplexity. An elegant recursive algorithm that attains this inefficiency is the *slowsort* method below.

The *slowsort* algorithm is a perfect illustration of the *multiply and surrender* paradigm, which is perhaps the single most important paradigm in the development of reluctant algorithms. The basic multiply and surrender strategy consists in replacing the problem at hand by two or more subproblems, each slightly simpler than the original, and continue multiplying subproblems and subsubproblems recursively in this fashion as long as possible. At some point the subproblems will all become so simple that their solution can no longer be postponed, and we will have to surrender. Experience shows that, in most cases, by the time this point is reached the total work will be substantially higher than what could have been wasted by a more direct approach.

To get a firmer grasp of the multiply and surrender method, let us follow step by step the development of the *slowsort* algorithm. We can decompose the problem of sorting $n$ numbers $A_1, A_2, \ldots, A_n$ in ascending order into (1) finding the maximum of those numbers, and (2) sorting the remaining ones. Subproblem (1) can be further decomposed into (1.1) find the maximum of the first $\lfloor n/2 \rfloor$ elements, (1.2) find the maximum of the remaining $\lceil n/2 \rceil$ elements, and (1.3) find the largest of those two maxima. Finally, subproblems (1.1) and (1.2) can be solved by sorting the specified elements and taking the last element in the result. We have thus multiplied the original problem into three slightly simpler ones (sort the first half, sort the second half, sort all elements but one), plus some overhead processing. We continue doing this recursively until the lists have at most one element each, at which point we are forced to surrender.

```
procedure slowsort ($A$, $i$, $j$) =
  { This procedure sorts the subarray $A_i, A_{i+1}, \ldots, A_j$. }
  if $i \geq j$ then
    return
  else
    $m \leftarrow \lfloor \frac{i+j}{2} \rfloor$
    slowsort ($A$, $i$, $m$)
    slowsort ($A$, $m+1$, $n$)
    if $A_m > A_j$ then $A_m \leftrightarrow A_j$ fi
```

```
      slowsort (A, i, j − 1)
    fi
  erudecorp
```

The recursion characterizing the running time of *slowsort* will look familiar to readers of Volume Three. It is essentially $T(n) = 2T(n/2) + T(n-1)$. The Hamming distance between this and the well known $T(n) = 2T(n/2) + cn$ recurrence of merge-sort is only 5, but a simple argument about finite differences shows that this is sufficient to make the first equation have no polynomially-bound solution. In fact it can be shown that the solution satisfies

$$C_a n^{\log(n)/(2+\varepsilon)} \le T(n) \le C_b n^{\log(n)/2}$$

for any fixed $\varepsilon > 0$ and some constants, $C_a$ and $C_b$. The idea of the proof (we were told that we need at least one proof to get published) is to assume that $T(n) = C_1 n^{C_2 \ln n}$ for some constants. Then

$$\frac{2T(n/2) + T(n-1)}{T(n)} = 1 - \frac{2C_2 \ln n}{n} + \frac{2^{1+C_2 \ln 2}}{n^{2C_2 \ln 2}} + O\left(\frac{(\ln n)^2}{n^2}\right).$$

Making $C_2 = 1/(2\ln 2)$ we can show that $T(n) \le C_b n^{\log(n)/2}$ and making $C_2 = 1/\big((2+\varepsilon)\ln 2\big)$ we show that $T(n) \ge C_a n^{\log(n)/(2+\varepsilon)}$ for sufficiently large $n$. (The constants $C_a$ and $C_b$ are fudge factors to get the induction going.) The details will available from the authors in the near future, on 7-track odd-parity EBCDIC tapes, containing rasterized punched card images of the proof written in EQN.

For practical applications, it is obvious that *slowsort* is *the* eminently suitable algorithm whenever your boss sends you to sort something in Paris. Among other nice properties, during the execution of *slowsort* the number of inversions in $A$ is nonincreasing. So, in a certain sense (if you are in Paris, all expenses paid, this sense is clear) *slowsort* never makes a wrong move.

## 6. Conclusions and open problems

The analysis of *slowsort* led to the following conjecture known as the *raising hypothesis* (RH): If the complexity of a problem is $O(gf)$ where $g$ and $f$ are functions of the length of the input and $f = o(g)$ then the simplicity of this problem is $\Theta(g^f)$.

The *extended raising hypothesis* (ERH) states that if the complexity of a problem is $O(g+f)$, then its simplicity is $\Theta(gf)$. It is obvious that ERH implies RH.

The proof or disproof of RH is one of the greatest open problems in Simplexity. However we must end on the sad note that it might be impossible to prove RH due to the well known incompleteness of Peano arithmetic.

## 5. References

[B] Bentley, J. L., **Programming pearls.** Somewh. in Comm. ACM.

[Ho] Homer, H., **The Odissey.**

[Wa] Wagner, R., **The Tannhäuser.**

[SB] Stolfi, J., and Broder, A. Personal experience.

[V1] Verne, J., **Journey to the Center of the Earth.**

[V2] Verne, J., **Around the World in 80 Days.**