29.5 years of Maple: how many of the design principles of the system paid dividends

Gaston H. Gonnet

Informatik, ETH, Zurich

Joachim von zur Gathn 60th
May 27, 2010
Bonn, Germany

Abstract

Abstract

Most of the original literature about Maple described it as a "compact and efficient computer algebra system". It was partly our goal to be able to run in small desktop computers and even on a pocket computer (the term "pocket symbolic" was also used). This talk will concentrate on four aspects of the early design that went in this direction and were the cornerstones of the design. These are the use of the language C, the S^2T measure of complexity, option remember and its implication which is the unique representation of subexpression and the systematic elimination of quadratic algorithms.

The S^2T measure

The S^2T measure comes from lower bounds of computational problems: e.g. when S is the auxiliary storage available and T is the time used, every algorithm must use $S^2T = \Omega(n^2)$

The S^2T measure

The S^2T measure comes from lower bounds of computational problems: e.g. when S is the auxiliary storage available and T is the time used, every algorithm must use $S^2T = \Omega(n^2)$

Usually there are severe restrictions on the computational model, for example, the input is available on a Turing machine tape.

The S^2T measure (II)

Given that such a measure is an invariant across algorithms, it makes sense as a measure of optimization for different algorithms.

The S^2T measure (II)

Given that such a measure is an invariant across algorithms, it makes sense as a measure of optimization for different algorithms.

That is, an algorithm which performs better under this measure is usually a **better algorithm**, not just a fluke due to operating on a different complexity space

The S^2T measure (III)

The S^2T measure (III)

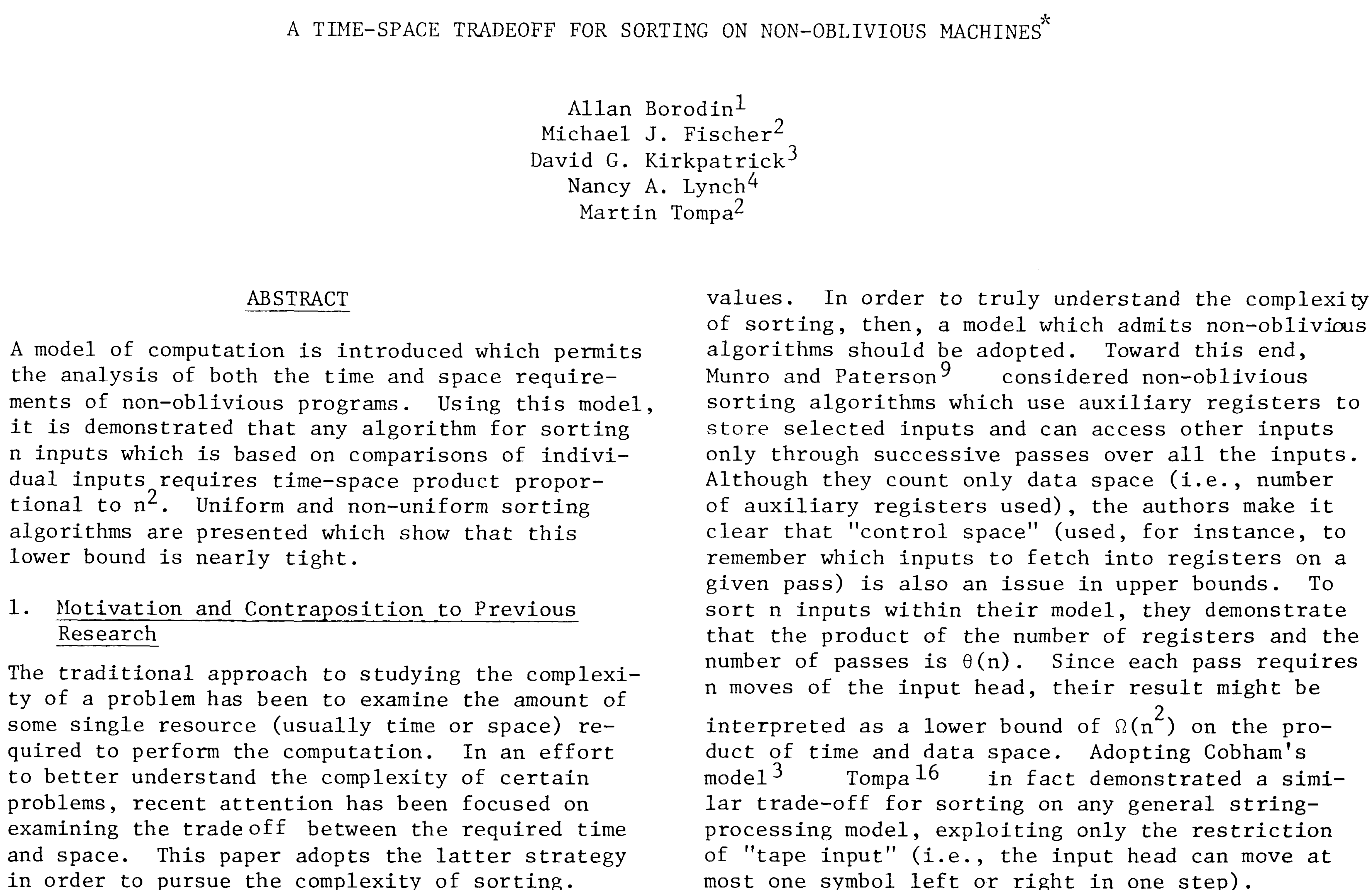
The S^2T measure is also very meaningful for computer algebra, as it weighs storage more than time and computer algebra has suffered more from lack of storage than from time.

The S^2T measure (III)

The S^2T measure is also very meaningful for computer algebra, as it weighs storage more than time and computer algebra has suffered more from lack of storage than from time.

In any case, it is good to have a measure so that algorithms can be benchmarked and the best can be selected. It is the basis of scientific software development.

The S^2T measure (the paper)



A TIME-SPACE TRADEOFF FOR SORTING ON NON-OBLIVIOUS MACHINES*

Allan Borodin[1]
Michael J. Fischer[2]
David G. Kirkpatrick[3]
Nancy A. Lynch[4]
Martin Tompa[2]

ABSTRACT

A model of computation is introduced which permits the analysis of both the time and space requirements of non-oblivious programs. Using this model, it is demonstrated that any algorithm for sorting n inputs which is based on comparisons of individual inputs requires time-space product proportional to n^2. Uniform and non-uniform sorting algorithms are presented which show that this lower bound is nearly tight.

1. Motivation and Contraposition to Previous Research

The traditional approach to studying the complexity of a problem has been to examine the amount of some single resource (usually time or space) required to perform the computation. In an effort to better understand the complexity of certain problems, recent attention has been focused on examining the trade off between the required time and space. This paper adopts the latter strategy in order to pursue the complexity of sorting.

values. In order to truly understand the complexity of sorting, then, a model which admits non-oblivious algorithms should be adopted. Toward this end, Munro and Paterson[9] considered non-oblivious sorting algorithms which use auxiliary registers to store selected inputs and can access other inputs only through successive passes over all the inputs. Although they count only data space (i.e., number of auxiliary registers used), the authors make it clear that "control space" (used, for instance, to remember which inputs to fetch into registers on a given pass) is also an issue in upper bounds. To sort n inputs within their model, they demonstrate that the product of the number of registers and the number of passes is $\theta(n)$. Since each pass requires n moves of the input head, their result might be interpreted as a lower bound of $\Omega(n^2)$ on the product of time and data space. Adopting Cobham's model[3] Tompa[16] in fact demonstrated a similar trade-off for sorting on any general string-processing model, exploiting only the restriction of "tape input" (i.e., the input head can move at most one symbol left or right in one step).

The S^2T measure (the paper)

This is the paper that I had in mind:

A TIME-SPACE TRADEOFF FOR SORTING ON NON-OBLIVIOUS MACHINES*

Allan Borodin[1]
Michael J. Fischer[2]
David G. Kirkpatrick[3]
Nancy A. Lynch[4]
Martin Tompa[2]

ABSTRACT

A model of computation is introduced which permits the analysis of both the time and space requirements of non-oblivious programs. Using this model, it is demonstrated that any algorithm for sorting n inputs which is based on comparisons of individual inputs requires time-space product proportional to n^2. Uniform and non-uniform sorting algorithms are presented which show that this lower bound is nearly tight.

1. Motivation and Contraposition to Previous Research

The traditional approach to studying the complexity of a problem has been to examine the amount of some single resource (usually time or space) required to perform the computation. In an effort to better understand the complexity of certain problems, recent attention has been focused on examining the trade off between the required time and space. This paper adopts the latter strategy in order to pursue the complexity of sorting.

values. In order to truly understand the complexity of sorting, then, a model which admits non-oblivious algorithms should be adopted. Toward this end, Munro and Paterson[9] considered non-oblivious sorting algorithms which use auxiliary registers to store selected inputs and can access other inputs only through successive passes over all the inputs. Although they count only data space (i.e., number of auxiliary registers used), the authors make it clear that "control space" (used, for instance, to remember which inputs to fetch into registers on a given pass) is also an issue in upper bounds. To sort n inputs within their model, they demonstrate that the product of the number of registers and the number of passes is $\theta(n)$. Since each pass requires n moves of the input head, their result might be interpreted as a lower bound of $\Omega(n^2)$ on the product of time and data space. Adopting Cobham's model[3] Tompa[16] in fact demonstrated a similar trade-off for sorting on any general string-processing model, exploiting only the restriction of "tape input" (i.e., the input head can move at most one symbol left or right in one step).

The *ST* surprise

The *ST* surprise

The big surprise is that this paper proves, that for sorting, ST = $\Omega(n^2)$

The $O(n^2)$ hidden bugs

I call a $O(n^2)$ hidden bug to any part of the system (kernel or library) which uses a quadratic time/space algorithm when a linear one is possible.

The $O(n^2)$ hidden bugs

I call a $O(n^2)$ hidden bug to any part of the system (kernel or library) which uses a quadratic time/space algorithm when a linear one is possible.

E.g. adding one term at a time to a list (or to a set) is a typical example of this problem

The $O(n^2)$ hidden bugs

I call a $O(n^2)$ hidden bug to any part of the system (kernel or library) which uses a quadratic time/space algorithm when a linear one is possible.

E.g. adding one term at a time to a list (or to a set) is a typical example of this problem

These bugs go often unnoticed until a production-type problem is submitted.

The $O(n^2)$ hidden bugs (II)

The $O(n^2)$ hidden bugs (II)

```
res := NULL;
for i to n do res := res, f(i) od;
```

The $O(n^2)$ hidden bugs (II)

```
res := NULL;
for i to n do res := res, f(i) od;
```

Internal kernel support was introduced very early (append), but unfortunately not extended to the users

The $O(n^2)$ hidden bugs (II)

```
res := NULL;
for i to n do res := res, f(i) od;
```

Internal kernel support was introduced very early (append), but unfortunately not extended to the users

```
res = New(EXPSEQ);
for( i=0; i<n; i++ )
    res = append(res,f(i));
```

The $O(n^2)$ hidden bugs (III)

The $O(n^2)$ hidden bugs (III)

Before the release of version 4.0 Mike Monagan combed patiently the entire kernel and removed most (all?) of the $O(n^2)$ bugs.

The $O(n^2)$ hidden bugs (III)

Before the release of version 4.0 Mike Monagan combed patiently the entire kernel and removed most (all?) of the $O(n^2)$ bugs.

For those in the internal mailing groups, it is still common to see reports of $O(n^2)$ new bugs

The $O(n^2)$ hidden bugs (IV)

The $O(n^2)$ hidden bugs (IV)

The only answer that I have to this problem is to create a "Quadratic Police"

The $O(n^2)$ hidden bugs (IV)

The only answer that I have to this problem is to create a "Quadratic Police"

"Cubic or higher Army" and "Exponential nuclear deterrent"

Option remember and unique representation

"Option remember" is the term that we use to describe the ability of a function of remembering previous arguments/results and avoid/save computation.

Option remember and unique representation

"Option remember" is the term that we use to describe the ability of a function of remembering previous arguments/results and avoid/save computation.

```
F := proc( n::integer )
option remember;
if n < 2 then n else F(n-1)+F(n-2) fi
end:
```

Remember and unique representation (II)

```
diff( tan(x), x$100 );
```

Remember and unique representation (II)

```
diff( tan(x), x$100 );
```

It may change an algorithm from exponential to linear in time (and/or space) required.

Remember and unique representation (II)

```
diff( tan(x), x$100 );
```

It may change an algorithm from exponential to linear in time (and/or space) required.

(At one point I wrote a program to create the largest expression ever (a Guiness-type record). This expression was so large that any linear function would never return, only remembering functions had a chance).

Remember and unique representation (III)

The rationale for remembering is that computer algebra (manipulated mathematical expressions) contain highly repetitive parts

Remember and unique representation (III)

The rationale for remembering is that computer algebra (manipulated mathematical expressions) contain highly repetitive parts

The kernel maintains unique representation with the same principle. The lowest-level expression simplifier uses option remember.

Remember and unique representation (III)

The rationale for remembering is that computer algebra (manipulated mathematical expressions) contain highly repetitive parts

The kernel maintains unique representation with the same principle. The lowest-level expression simplifier uses option remember.

All expressions are simplified and unified recursively. Duplicates are discarded.

Remember and unique representation (IV)

This is not the panacea, it has excellent consequences but also produces strange behaviours. It is a tough design decision.

Remember and unique representation (IV)

This is not the panacea, it has excellent consequences but also produces strange behaviours. It is a tough design decision.

But, there is no question in my mind that it makes for the largest space/time economy in the early Maple.

Remember and unique representation (V)

Remembering required hash signatures, and signatures were soon quite ubiquitous in Maple. That is the mapping of expressions to integers or integers mod n.

Remember and unique representation (V)

Remembering required hash signatures, and signatures were soon quite ubiquitous in Maple. That is the mapping of expressions to integers or integers mod n.

It is not a surprise, in this context, that the heuristic GCD algorithm became a leading example of several similar algorithms and a cornerstone of Maple's efficiency.

Memory and Ghz are cheap

How many times did I hear: “why do you bother about memory, memory is cheaper every day” ?

Memory and Ghz are cheap

How many times did I hear: “why do you bother about memory, memory is cheaper every day” ?

Answer: as many as #(fools) $\times$ #(encounters)

Memory and Ghz are cheap

How many times did I hear: "why do you bother about memory, memory is cheaper every day" ?

Answer: as many as #(fools) $\times$ #(encounters)

A system which uses memory wisely will always be ahead of one who doesn't. (paging, garbage collection, compaction, etc. will also cost time)

Use of the C language

A predecessor of Maple (Wama) and Maple were initially coded in B (B is a successor of BCPL and a predecessor of C). Later Maple was converted to allow compilation both in B and C (through a pre-processor called Margay).

Use of the C language

A predecessor of Maple (Wama) and Maple were initially coded in B (B is a successor of BCPL and a predecessor of C). Later Maple was converted to allow compilation both in B and C (through a pre-processor called Margay).

The architecture of Maple was strongly influenced by the language (B and later very simple C). The language itself was influenced by Algol68.

Use of the C language (II)

The CA community did not believe in this approach, personally I received several critiques for not using Lisp (or one of its dialects)

Use of the C language (II)

The CA community did not believe in this approach, personally I received several critiques for not using Lisp (or one of its dialects)

Once I asked one of the top system builders of the time: what do I get from Lisp? After an extremely long pause the answer was: “garbage collection for free”

Use of the C language (II)

The CA community did not believe in this approach, personally I received several critiques for not using Lisp (or one of its dialects)

Once I asked one of the top system builders of the time: what do I get from Lisp? After an extremely long pause the answer was: "garbage collection for free"

We do not need to defend this decision, just observe the language in which the current top CA systems are written.

Fast to start

Dennis Ritchie tried Maple once in the very early stages of its history (Maple's kernel used to be one of the benchmark programs to compile inside ATT). He liked the look-and-feel of Maple, and in particular he was very positive about its rapid response.

Fast to start

Dennis Ritchie tried Maple once in the very early stages of its history (Maple's kernel used to be one of the benchmark programs to compile inside ATT). He liked the look-and-feel of Maple, and in particular he was very positive about its rapid response.

We did not think very highly of this remark at the time. Maybe this was a very good compliment that we did not appreciate enough.

Fast to start (II)

We now know that it does make a difference to start in 0.1 secs as opposed to starting in 30 secs. It hurts the "calculator" use of Maple

Fast to start (II)

We now know that it does make a difference to start in 0.1 secs as opposed to starting in 30 secs. It hurts the "calculator" use of Maple

The speed of starting is a combination of various aspects: small kernel, load-on-demand library, efficient language, among others

Conclusion

Natural selection, if applicable to software, shows that some subset of these features is very good, as Maple survived 29.5 years

the END