'Runsort'—An Adaptive Mergesort for Prolog

Mike Brady
Department of Computer Science
University of Dublin
Trinity College Dublin
Ireland
brady@cs.tcd.ie

March 31, 2005

Abstract

This note describes a novel list-sorting method for Prolog which is stable, has $O(n \log n)$ worst-case behaviour and $O(n)$ best-case behaviour. The algorithm is an adaptive variant of bottom-up mergesort using so-called *long runs* of preexisting order to improve efficiency; accordingly we have called it '`runsort`'. `Runsort` compares favourably with `samsort`, and a modification to `samsort` is suggested.

1 Introduction

The objective of this work was to develop a general-purpose sorting predicate for Prolog with good time and memory performance. The standard `quicksort/3` predicate[1] listed below has order $O(n^2)$ worst-case run time behaviour. Furthermore, `quicksort` has a very large memory requirement in the worst case.

```
quicksort([A|B],C,D) :-
        partition(B,A,E,F),
        quicksort(F,G,D), %not a tail-recursive call
        quicksort(E,C,[A|G]).
quicksort([],A,A).

partition([A|B],C,[A|D],E) :-
        A@=<C,
        !,
        partition(B,C,D,E).
partition([A|B],C,D,[A|E]) :-
        partition(B,C,D,E).
partition([],A,[],[]).
```

The order $O(n^2)$ worst case time behaviour of quicksort is well known, and in this implementation it occurs if the complete list is already in non-descending or descending order. The worst-case memory requirement arises as follows: Given a list of n items in ascending order, the `partition/4` predicate will partition the

list into a newly-constructed sublist of $n-1$ items and an empty list. The sublist of $n-1$ items will then be sorted by a recursive call—which is not tail-recursive—to `quicksort/3`. This call will generate another newly-constructed list of $n-2$ items followed by another non tail-recursive call to `quicksort/3` to sort a sublist of $n-2$ items, and so on. Thus, a list of n items in ascending order will give rise to a stack of $n-1$ simultaneously extant non tail-recursive calls, each represented by a stack frame and each containing a newly-constructed sublist of the list constructed in the previous call. Hence, there will be an $O(n)$ requirement for stack frame space and an order $O(n^2)$ requirement for space to represents all the sublists.

1.1 Mergesort

Mergesort is another well known sorting algorithm which has order $O(n \log n)$ worst-case run time behaviour. A straightforward implementation of mergesort in Prolog splits the list of n elements to be sorted into n individual sublists. The n sublists thus generated are merged into $n/2$ ordered sublists, which in turn are merged into $n/4$ ordered sublists, and so on until just one ordered sublist—the result—remains (see Figure 1 for a pictorial representation). The predicates used are tail-recursive and so memory usage should be relatively constant.

Figure 1: Mergesort of a small list. The split phase generates a sublist for each element in the list to be sorted. In the merge phase, successive pairs of sublists are repeatedly merged until just one sublist remains.

The code for this implementation of mergesort is listed below:

```
%split list into n sublists and merge...
simple_mergesort([],[]).
simple_mergesort(X,Y) :-
        nonvar(X),
        simple_split_phase(X,Fragments),!,
        merge_phase(Fragments,Y).

%split incoming list of n elements into n sublists
simple_split_phase([],[]).
simple_split_phase([H|T],[[H]|R]) :-
        simple_split_phase(T,R).

%merge sublists until just one sublist remains
merge_phase([],[]) :- !.
merge_phase([X],X) :- !.
merge_phase(X,Y) :-
        merge_pass(X,A),!,
        merge_phase(A,Y).

merge_pass([],[]).
merge_pass([X],[X]) :- !.
```

```
merge_pass([X,Y|R],[Z|A]) :-
        merge(X,Y,Z),!,
        merge_pass(R,A).

merge([],X,X) :- !.
merge(X,[],X) :- !.
merge([X|Y],[A|B],[X|R]) :- X@<A,!,merge(Y,[A|B],R).
merge(Y,[A|B],[A|R]) :- merge(Y,B,R).
```

2 An Improved Mergesort

It will be observed that an n element list is initially split into a list of n one-element sublists and these sublists are immediately merged into a list of $n/2$ sublists before being merged further. These two steps can be amalgamated as follows: Instead of splitting a list of n elements into one-element sublists, the split phase can be modified to produce $n/2$ sublists of ordered pairs of elements (see example in Figure 2).

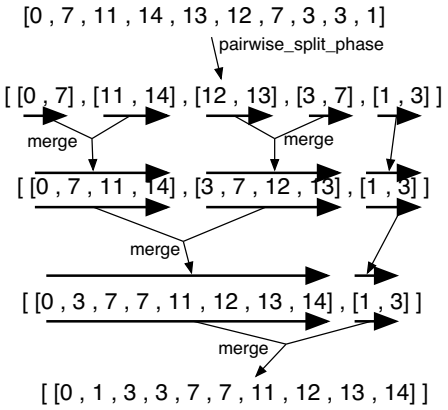

Figure 2: An improved mergesort of a small list. The improvement is that the split phase generates a sublist for each *pair* of elements in the list to be sorted. One less merge pass is needed during the merge phase.

During the split phase, pairs of elements are compared and put in relative order in new sublists. The $n/2$ extra comparisons needed to put the each pair of elements in order are offset by the fact that one less merge pass—which would need the same number of comparisons—is needed. The modified code for this version of mergesort is listed below (the merge phase code is the same as before):

```
%split list into n/2 sublist and merge...
mergesort([],[]).
mergesort(X,Y) :-
        nonvar(X),
        pairwise_split_phase(X,Fragments),!,
```

```
        merge_phase(Fragments,Y).

%split incoming list of n elements into n/2 ordered sublists
pairwise_split_phase([X,Y|R],[[X,Y]|Z]) :-
        X@=<Y,!,
        pairwise_split_phase(R,Z).
pairwise_split_phase([X,Y|R],[[Y,X]|Z]) :-
        !,
        pairwise_split_phase(R,Z).
pairwise_split_phase([X],[[X]]).
pairwise_split_phase([],[]).
```

3 Runsort—An Adaptive Mergesort

Runsort is an extension of the improved mergesort of the previous section. The use of a comparison operator in the split phase is extended to identify existing order in sequences of elements, making the split phase adaptive (see [2] for a survey of adaptive sorting programs). Where a sequence is in non-descending order, the sublist representing the sequence is extended by appending each new element to the end of it. If the sequence is in descending order, the sublist is extended by prepending each new element to the start of it. Either way, sublists are generated in non-descending order during the split phase and are then merged in the normal way. Figure 3 illustrates the process.

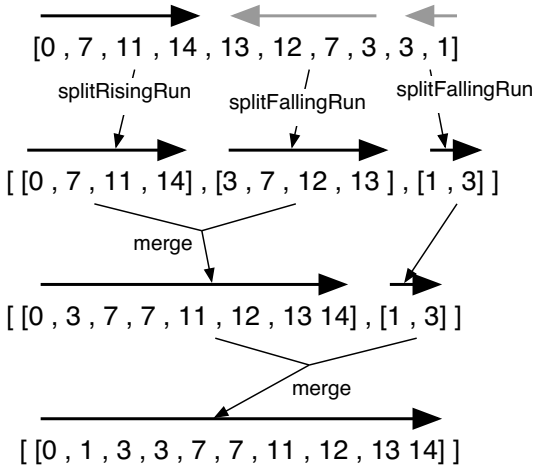

Figure 3: `Runsort` working on a small list. The split phase identifies sequences of elements that are already in non-descending or descending order. Each sequence becomes a sublist. The sublists are then merged in the normal way.

Here is the code for the split phase of `runsort`:

```
runsort([],[]).
runsort(X,Y) :-
        nonvar(X),
        runsort_split_phase(X,Fragments),!,
        merge_phase(Fragments,Y).

runsort_split_phase([],[]).
runsort_split_phase([X,Y|R],[[X,Y|L]|Z]) :-
        X@=<Y,
        splitRisingRun(R,S,L,Y),!,
        runsort_split_phase(S,Z).
runsort_split_phase([X,Y|R],[T|Z]) :-
        splitFallingRun(R,S,[Y,X],T,Y),!,
        runsort_split_phase(S,Z).
runsort_split_phase([X],[[X]]).

splitRisingRun([X|R],S,[X|L],K) :-
        X@>=K,
        !,
        splitRisingRun(R,S,L,X).
splitRisingRun(R,R,[],_).

splitFallingRun([X|R],S,Ti,To,K) :-
        X@<K,
        !,
        splitFallingRun(R,S,[X|Ti],To,X).
splitFallingRun(L,L,B,B,_).
```

The code operates as follows:

- If the incoming list is empty (clause 1), or contains only one element (clause 4), then the empty list, or a list of one sublist, is returned;
- If the incoming list's first two elements are in non-descending order (clause 2), they are made the start of an output sublist, and `splitRisingRun` is called to append each element in the rest of the non-descending sequence to the sublist;
- otherwise (i.e. if the first two elements are in descending order, clause 3) they are made the end of a sublist, and `splitFallingRun` is called to prepend each element in the rest of the descending sequence into the sublist. At the end of the falling run, the sublist contains the reversed falling sequence, now in ascending order.

Thus, in one pass, the list has been broken into maximal-length non-descending sublists. The rest of the process, i.e. the merge of the sublists to produce a final single sublist, is conventional—the number of passes made by `merge_pass` will be $\lceil \log_2 \rceil$ of the number of sublists.

4 Discussion

The three sorting programs presented above have similar structures: a split phase followed by a merge phase.

4.1 The Split Phase

The split phase is 'straight' (i.e. non-adaptive) for the two mergesort programs; sublists of one or two elements are always formed. The split phase of `runsort` is adaptive because it generates sublists of the *long runs* [4, Section 5.1.3, Exercise 23] in the input list. Long runs are either increasing or decreasing, depending on the order of their first two elements. An increasing long run is where successive elements are greater or equal to their predecessors; a decreasing long run is where successive elements are strictly less than their predecessors.[1] The best case situation for runsort is when the list comprises one long run and one sublist is formed by the split phase. The worst case is when none of the long runs contain more than two elements, so the extra $n/2$ comparisons performed do not result in a smaller number of sublists being formed. The average case, where the list contains a sequence of random numbers, the number of sublists generated should be about $n/2.4202 = 0.41n$ [4, Section 5.1.3, Exercise 23] or about 19% fewer than generated by the improved mergesort, though at the cost of $n/2$ extra comparisons.

4.2 The Merge Phase

All three programs use the same code for the merge phase and it exhibits some adaptive behaviour. Consider the two lists `[0,1,2,3]` and `[4,5,6,7]`. It takes four comparisons—`<0:4, 1:4, 2:4, 3:4>`—to merge them into the new list `[0,1,2,3,4,5,6,7]`. Now consider the lists `[0,2,4,6]` and `[1,3,5,7]`. Seven comparisons—`<0:1, 2:1, 2:3, 4:3, 4:5, 6:5, 6:7>`—are needed to achieve the same result. Where two lists of length n are merged, therefore, between n and $2n - 1$ comparisons are needed.

4.2.1 Best Case

Assuming lists of equal length, the best case situation for the merge phase is where the ranges of values of the sublists to be merged don't overlap. For example, if the original list is `[0,1,2,3,4,5,6,7]`, the pairs of sublists are `[0,1] & [2,3]`, `[4,5] & [6,7]` and `[0,1,2,3] & [4,5,6,7]`.

4.2.2 Worst Case

The worst case situation for the merge phase is where the ranges of values of the sublists overlap to the maximum extent possible. For example, if the original list is `[0,4,2,6,1,5,3,7]`, the pairs of sublists are

[1] This definition of a long run ensures that runsort is stable, because it implies that a descending long run can never contain identical elements. From this, it follows that reversing descending long runs 'in place' can not alter the relative order of identical elements in the original list.

`[0,4] & [2,6]`, `[1,5] & [3,7]` and `[0,2,4,6] & [1,3,5,7]`[2]. Lists like this can be generated from lists of ascending positive integers by reversing the order of the bits in the binary representation of the numbers. For example, given the list `[0,1,2,3,4,5,6,7]`, or, in binary, `[000,001,010,011,100,101,110,111]`, by reversing the order of the bits, we obtain
`[000,100,010,110,001,101,011,111]` or `[0,4,2,6,1,5,3,7]`.

5 Related Work

Another adaptive mergesort called `samsort` (for **s**mooth **a**pplicative **m**erge**sort**) was developed by Richard O'Keefe and is part of the Edinburgh Prolog Library [7]. `Samsort` will add the next element in the list to the front of the current run if it is less than the head element or it will add the element to the end of the run if if is not less than the last element. Failing this, the current run will be closed and a new run begun. `Samsort` can therefore identify longer runs than `runsort` at the cost of some extra comparisons and at the cost of having to manage the head and tail of a list while it is being formed. An interesting feature of `samsort` is that it interleaves splitting the incoming list with merging the resulting fragments using an auxiliary predicate called `samfuse`.

As will be seen from the empirical results section, `samsort` performs less well by comparison with `runsort`, and in an effort to identify the causes, a modification was developed which retains the distinctive way that runs are identified by `samsort` but which uses the same merge-phase predicates as all the other programs. This modified `samsort`, called `new_samsort`, turned out to be as fast or faster than `samsort` in every case and was very slightly faster than `runsort` in a number of cases. The code for `new_samsort` (apart from the `merge_phase/2` code which it shares with `mergesort`, `runsort` etc.) follows:

```
%new_samsort splits the list a la samsort, uses the standard merge phase
new_samsort([],[]).
new_samsort(X,Y) :-
        nonvar(X),
        sam_split(X,Fragments),!,
        merge_phase(Fragments,Y).

sam_split([],[]).
sam_split([H|T],[Run|Rest]) :-
        sam_run(T,[H|Q],[H|Q],Run,More),
        sam_split(More,Rest).

%sam_run comes unaltered from the original samsort.
sam_run([], Run, [_], Run, []) :- !.
sam_run([Head|Tail], QH, QT, Run, Rest) :-
        sam_run(QH, QT, Head, Tail, Run, Rest).
```

[2]Even though number of comparisons would be the same, program execution would be slower if the higher number in a comparison appears in the first sublist rather than the second, as it results in the failure of the predicate `X@<A` in the third clause of the `merge/3` predicate and this forces a backtrack to the fourth clause, causing extra computation.

```
sam_run([Ah|At], Qt, H, T, Run, Rest) :-
        H @< Ah, !,
        sam_run(T, [H,Ah|At], Qt, Run, Rest).
sam_run(Qh, [Qt], H, T, Qh, [H|T]) :-
        H @< Qt, !.
sam_run(Qh, [_,H|Nt], H, T, Run, Rest) :-
        sam_run(T, Qh, [H|Nt], Run, Rest).
```

6 Empirical Results

The programs `simple_mergesort` (SM), `mergesort` (M), `runsort` (R), `quicksort` (Q), `samsort` (S) and `new_samsort` (NS) were tested on lists of varying lengths and types. Testing was performed on a 550 MHz Macintosh PowerBook G4 with 512 MB RAM running Mac OS X 10.2.6, with Open Prolog[3] running in a memory allocation of 70,000 KBytes in the 'Classic' environment. The machine was disconnected from all networks, no other programs were running, sharing services were turned off, and all times are elapsed times, measured using the `runtime` selector of the `statistics/2` built-in predicate. This actually measures *elapsed* time in Open Prolog, as the Mac OS does not have a reliable runtime clock. Times were averaged over ten runs and the results rounded to the nearest millisecond.

Two sets of results are presented. The first set is best-, average- and worst-case performance timings for lists of numbers. For average case tests, lists of pseudo random numbers were prepared using the built-in predicate `random/3` which is an implementation of Wichman and Hill's pseudo random number generator [9, 8]. For worst case tests, lists of numbers were generated as outlined in the previous section: each number was generated by reversing the bit representation of its zero-based position number in the list. For best case tests, lists of pseudo-random numbers were sorted into non-descending order. Table 1 gives execution times for the four sorting programs, and Figure 4 depicts this information graphically.

[3]Open Prolog is available at `http://www.cs.tcd.ie/open-prolog/` . The version used here is 1.1b15.

	Worst			Average						Best				
Length	SMw	Mw	Rw	Qa	SMa	Ma	Ra	Sa	NSa	SMb	Mb	Rb	Sb	NSb
500				19	18	16	17	21	18					
1000				43	41	36	38	48	41					
2000				97	90	87	87	105	93					
5000				268	266	248	261	288	254					
10000	585	551	573	608	571	532	561	617	550	322	289	31	83	83
20000	1258	1187	1229	1288	1228	1160	1225	1337	1196	700	621	61	169	168
50000	3431	3246	3341	3522	3320	3141	3190	3604	3250	1831	1651	152	416	417

Table 1: Test results for `simple_mergesort` (SM-), `mergesort` (M-), `runsort` (R-), `quicksort` (Q-), `samsort` (S-) and `new_samsort` (NS-). Each row give results for list of the indicated length. Each column give execution time in milliseconds for the program on a list of the size denoted by the row. Worst, average and best figures of given, indicated by the -w, -a and -b suffixes. Worst-case figures are for lists of numbers that cause the maximum number of initial sublists to be generated in the split phase and the maximum number of comparisons in the merge phase. Average-case figures are for lists of pseudo-random numbers. Best-case figures are for lists already in ascending order. Worst case figures were not calculated for `quicksort`, `samsort` or `new_samsort`.

The second set of results are for sorting lists of pseudo-random numbers where the second half of the list is sorted into ascending order. The intention here was to simulate a situation where items would be prepended to an existing ordered list and then sorted to bring the full list into order. Table 2 gives execution times for the four sorting programs on lists of 10,000, 20,000 and 50,000 numbers, and Figure 5 depicts this information graphically.

Length	M	R	S	NS
10000	436	345	533	449
20000	947	785	1121	991
50000	2525	2212	2974	2653

Table 2: Test results for `mergesort` (M), `runsort` (R), `samsort` (S) and `new_samsort` (NS) on lists of pseudo-random numbers where the second half of the list is sorted into ascending order. The idea is to simulate a situation that might arise in practice where items are haphazardly prepended to a list of items that is already in order, and the whole list is then re-sorted. `Runsort` is best, and `samsort` and `new_samsort` are both worse than straight `mergesort`.

7 Conclusions

A novel list-sorting algorithm—`'runsort'`—has been presented, which is an adaptive, stable implementation of `mergesort`. The algorithm has good worst-case behaviour and takes advantage of long runs of preexisting order to give very good best-case behaviour. Though its average case and worst case performance is a little slower than `mergesort`, its adaptive features make it an attractive candidate for general-purpose sorting routines, such as might be built in to a Prolog implementation.

`Runsort` compares favourably with `samsort`, being faster in every situation tested. There are some cases where `samsort` would possibly be faster, for example if a list was composed of an ascending list interleaved

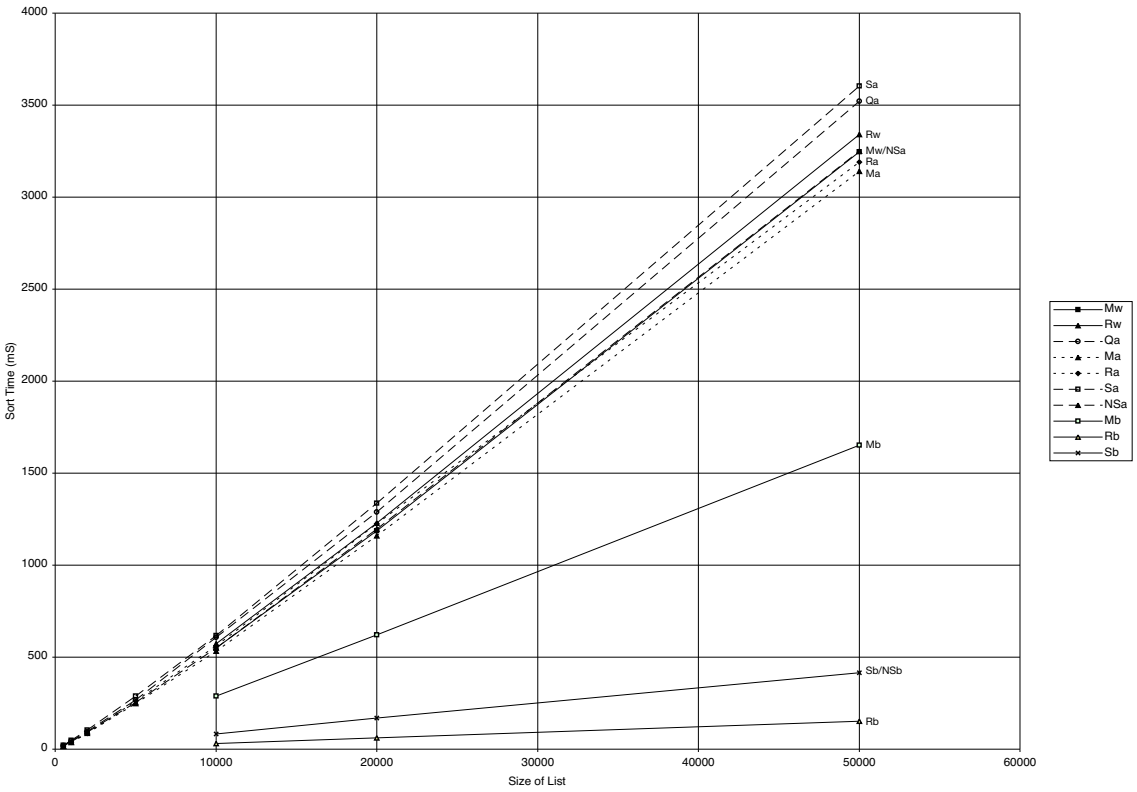

Figure 4: Graph of the best-case, average and worst-case execution times given in Table 1. `Runsort` is fastest in the best case (Rb), followed by `samsort` and `new_samsort` in the best case (Sb/NSb). Next is the best case performance of `mergesort` (Mb). The average (-a) and worst case (-w) performances of `runsort` (R-), `new_samsort` (NS-) and `mergesort` (M-) are bunched together, with `mergesort` slightly faster than the other two. The average performances of `runsort` and `new_samsort` are very close, with `new_samsort` having a slight advantage for list sizes of about 5,000 to 45,000 elements and `runsort` being faster for shorter and longer lists. The average case performance of `samsort` (Sa) is the worst depicted. `Simple_mergesort` is always slightly slower than `mergesort` and is omitted from the graph.

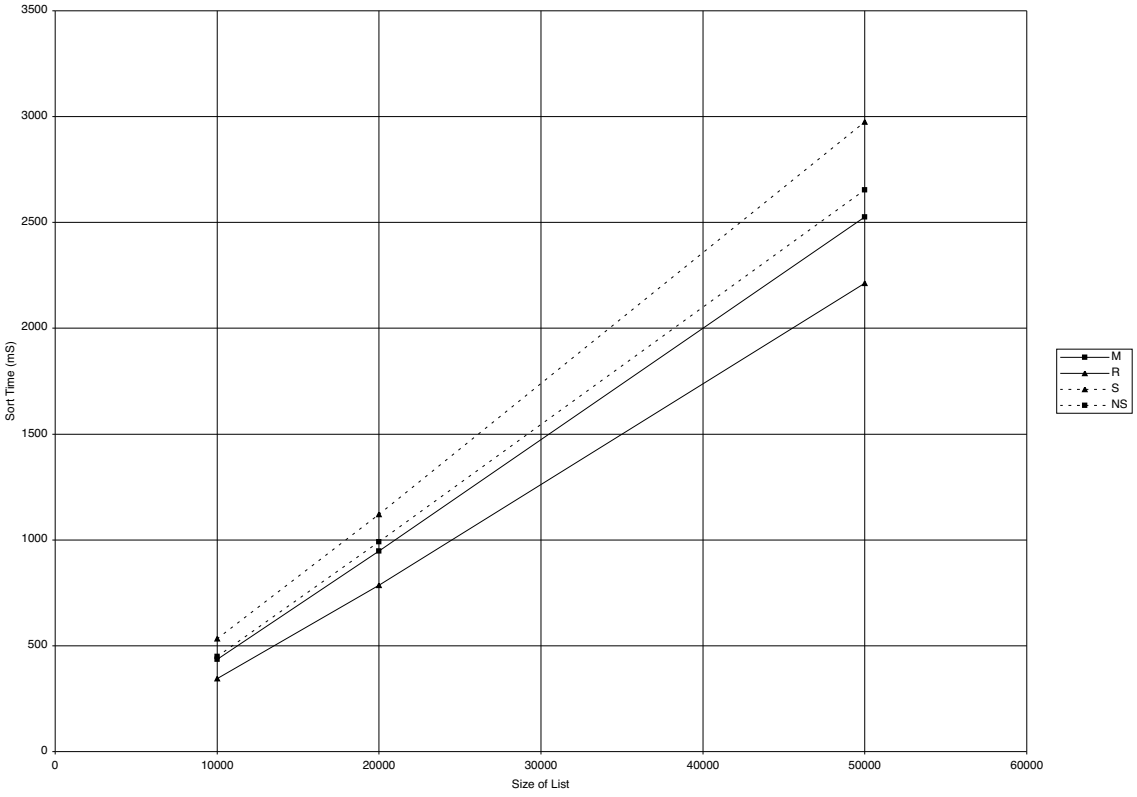

Figure 5: Graph of the data presented in Table 2, showing execution times in mS for `mergesort` (M), `runsort` (R), `samsort` (S) and `new_samsort` (NS) sorting lists of pseudo-random numbers where the second half of the list is presorted into ascending order. `Runsort` is best, and `samsort` and `new_samsort` are both worse than straight `mergesort`.

with a descending list, where the first item of the descending list sorts before the first item of the ascending list.

`New_samsort` appears to offer a definite advantage over `samsort` and is actually slightly faster than `runsort` is some cases. Paradoxically, `samsort` and `new_samsort` are both slower than [straight] `mergesort` in lists where the first half is disordered and second half is ascending.

8 Future Work

It would be interesting to make a meticulous analysis of `runsort`, in the style of Katajainen and Traff [3]. The main problem with attempting to do this with even a subset of Prolog would be identifying and characterising the primitive operations of the language. For example, while most Prologs use structure copying techniques, some, including Open Prolog, use structure sharing. Thus, for instance, constructing a new list from two existing lists could generate more operations in a structure copying than in a structure sharing Prolog. Perhaps a better approach would be to avoid Prolog altogether and analyse an implementation of `runsort` written in C and using linked lists.

A second area for future work is the status of long runs, which `runsort` is adaptive to, as measures of presortedness. Mannila [5] concludes that long runs do not fully qualify as measures of presortedness, "as [such a measure] ignores the reversal operations needed." In addition, long runs violate one other general condition considered necessary for measures of presortedness to satisfy. However, the split phase of `runsort` shows that the reversal operations can be accommodated at little or no cost beyond identifying the long runs in the first place. It would be very interesting to see if the characterisation of measures could be meaningfully extended to include long runs.

References

[1] D. L. Bowen, L. Byrd, L. M. Pereira, and Warren D. H. D. DECsystem-10 Prolog User's Manual. Occasional Paper 27, Dept. of AI, University of Edinburgh, Nov 1982.

[2] Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.

[3] Jyrki Katajainen and Jesper Larsson Traff . A meticulous analysis of mergesort programs. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1997.

[4] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[5] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, 34:318–325, 1985.

[6] A. Ian McLeod. Statistical algorithms: Remark AS R58: A remark on Algorithm AS 183. an efficient and portable pseudo-random number generator. *Applied Statistics*, 34(2):198–200, 1985. See [8, 10].

[7] Richard A. O'Keefe. Samsort (Smooth Applicative Mergesort). `http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/code/tools/edinbrgh/` , 1984.

[8] B. A. Wichmann and I. D. Hill. Statistical algorithms: Algorithm AS 183: An efficient and portable pseudo-random number generator. *Applied Statistics*, 31(2):188–190, June 1982. See remarks [6, 10].

[9] Brian A. Wichmann and I. D. Hill. Building a random-number generator: A Pascal routine for very-long-cycle random-number sequences. *Byte Magazine*, 12, March 1987.

[10] H. Zeisel. Statistical algorithms: Remark ASR 61: A remark on Algorithm AS 183. an efficient and portable pseudo-random number generator. *Applied Statistics*, 35(1):89–89, 1986. See [8, 6].