Processing List Elements in Reverse Order

Irène Durand and Robert Strandh
University of Bordeaux
Science and Technology College
LaBRI, 351, Cours de la Libération
33405 Talence Cedex, France
robert.strandh@u-bordeaux.fr
irene.durand@u-bordeaux.fr

ABSTRACT

The Common Lisp sequence functions and some other functions such as `reduce` accept a keyword parameter called `from-end`. In the case of `count` and `reduce`, when the value of that parameter is *true*, it is required that the elements are processed in reverse order. Some implementations, in particular SBCL, CCL, and LispWorks, implement the reverse-order traversal of a list by non-destructively reversing the list and then traversing the reversed version instead. This technique requires $O(n)$ additional heap space (where n is the length of the list), and increases the amount of work required by the garbage collector.

In this paper, we present a technique that only uses additional *stack space*. To avoid stack overflow, our technique traverses parts of the list multiple times when the list has more elements than the available stack space can handle. We show that our technique is fast, in particular for lists with a large number of elements, which is also the case where it is the most important to avoid allocating heap space.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Run-time environments*

General Terms

Algorithms, Languages

Keywords

Common Lisp, List processing

1. INTRODUCTION

The Common Lisp *sequence* functions are defined to work on lists as well as vectors. Furthermore, many of these sequence functions accept a keyword argument `from-end` that alters the behavior, in that elements toward the end of the sequence are favored over elements toward the beginning of the sequence. Other functions, in particular `reduce`, also accept this keyword argument.

Most sequence functions are not required to process the elements from the end of the sequence even though the value of the `from-end` keyword argument is *true*. For example, it is allowed for `find` to compare elements from the beginning of the sequence and return the *last* element that *satisfies the test*[1] even if the test has side effects. There is one exception, however: The `count` function is required by the standard to test the elements from the end of the sequence.[2] In addition to the function `count`, the function `reduce` also requires processing from the end of the list when `from-end` is *true*.

Processing list elements from the beginning to the end could, however, have a significant additional cost associated with it when processing from the end would require fewer executions of the test function, and the additional cost increases with the complexity of the test.

In this paper, we will concentrate on the functions that are required by the standard to process list elements from the end, and we will use only the function `count` in our test cases.

There are of course some very simple techniques for processing elements from the end of a list. One such technique would be to start by reversing the list[3] and processing the elements from the beginning in the reversed list. This technique is used by several implementations, including SBCL, CCL, and LispWorks. A major disadvantage of this technique is that it requires $O(n)$ additional heap space, and that it requires additional execution time by the memory allocator and the garbage collector.

Another simple technique would be to traverse the list *recursively* and testing the elements during the *backtracking*

[1]The phrase *satisfy the test* has a precise meaning in the Common Lisp standard as shown in section 17.2 in that document.

[2]Though if the test has no side effects and cannot fail, as is the case of functions such as `eq` or `eql`, testing from the beginning is arguably conforming behavior.

[3]By *reversing the list* we do not mean modifying the list as `nreverse` would do, but creating a new list with the elements in reverse order as `reverse` would do. The reason for excluding modifications to the list is that doing so might influence the semantics of other functions, including perhaps the test function or the view of the list by other threads.

phase of the recursion.[4] Again, $O(n)$ extra space is required, even though this time it is *stack space* rather than heap space, so that the memory allocator and the garbage collector are not solicited, at least in most implementations. Worse, many implementations have a fairly small call stack, in particular in multi-threaded implementations where each thread must have a dedicated stack. Aside from these disadvantages, this technique is however fairly efficient in terms of execution time, because a simple function call is quite fast on most modern processors. For that reason, we will use recursion as the basis of the technique described in this paper, but with fairly few recursive calls so that the additional extra space is modest.

Throughout this paper, we assume that the lists to be processed have a large number of elements, for several reasons:

- We do not want the list to be small enough to fit in the cache, because cache performance depends on other workloads as well.
- For short lists, performance may be dominated by the overhead of calling a few functions, or by loop prologues and epilogues. By using long lists, we make sure that performance is dominated by traversing the list and computing the test.
- If the list is too short, it can be processed by a simple recursive technique. In order to avoid this possibility, we want the lists to have orders of magnitude more elements than the size of the stack.

Furthermore, throughout this paper, we will assume that the *test* to be performed on the elements of the list is the function `eq`. By making this assumption, we expose the worst case for our technique, because the execution time will then be dominated by the overhead of traversing the list, as opposed to by executing the test function.

It should be noted that the difficulty of processing list elements in reverse order is due to the way Common Lisp practically imposes the representations of such lists. Other representations are, of course, possible. For instance, Hughes [2] suggested a representation of lists as first-class functions. Similarly, in his talk on parallelism in 2009,[5] Guy Steele proposed a representation of lists for parallel processing, based on using the four operations `item`, `split`, `list`, and `conc`. Clearly, such alternative representations could be devised that facilitate processing elements in reverse order.

In this paper, we use the international convention [1] for writing logarithms. Hence, we write $\mathsf{lb}\, n$ for the logarithm in base 2. We use log only when the base is unimportant. Given a real number n, the notation $\lfloor n \rfloor$ represents the *floor* of n and $\lceil n \rceil$ *ceiling*. For example, $\lfloor 1.5 \rfloor = 1$ and $\lceil 1.5 \rceil = 2$.

[4]Despite considerable research, we have not been able to find any original reference to this technique, and it seems too trivial for standard text books to even discuss. We must conclude that this technique must be so obvious that it was probably discovered independenlty by several people.

[5]http://groups.csail.mit.edu/mac/users/gjs/6.945/readings/MITApril2009Steele.pdf

2. PREVIOUS WORK

We will frequently refer to techniques used by SBCL because of its reputation as a high-performance implementation. We will however also use other high-performance implementations for comparison when we have information on the techniques used by those implementations, or when we can reasonably guess these techniques from other evidence.

For its implementation of `find`, SBCL takes advantage of the freedom given by the standard, by processing elements from the beginning, and remembering the last element that satisfies the test. For implementations where the technique is unknown, it suffices to write a test function that counts the number of times it is invoked and run it on a list where only the last element satisfies the test.

For its implementation of `count`, SBCL uses the simple technique of reversing the list first and then processing the elements of the reversed list from the beginning.

As we already mentioned, we use recursion as the basis of our technique, because it is quite fast. We devised the following test to verify this hypothesis:

```
(defun recursive-count (x list)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (if (endp list)
      0
      (+ (recursive-count x (cdr list))
         (if (eq (car list) x) 1 0))))
```

On SBCL executing this function on a list with 50000 elements where no element satisfies the test takes around 4ns per element compared to around 1.5ns for an explicit loop from the beginning of the list, and around twice as fast as calling `count`. This result indicates that we should use recursion whenever the size of the stack allows it, though there is of course no portable way of testing how much stack space is available. However, each implementation might have a specific way, which would then be good enough.

3. OUR TECHNIQUE

3.1 Basic technique

To illustrate our technique, we first show a very simple version of it in the form of the following code:[6]

```
(defun count-from-end (x list)
  (labels ((aux (x list n)
             (cond ((= n 0) 0)
                   ((= n 1)
                    (if (eq x (car list)) 1 0))
                   (t (let* ((n/2 (ash n -1))
                             (half (nthcdr n/2 list)))
                        (+ (aux x half (- n n/2))
                           (aux x list n/2)))))))
    (aux x list (length list))))
```

This function starts by computing the length of the list and then calling the auxiliary function with the original arguments and the length. The auxiliary function calls `nthcdr`

[6]Throughout this paper, we rely on the left-to-right evaluation order mandated by the Common Lisp standard.

in order to get a reference to about half the list it was passed. Then it makes two recursive calls, first with the second half of the list and then with the first half of the list. The recursion terminates when the list has a single element or no element in it. When it has no element in it, clearly the count is 0. When it has a single element in it, the element is compared to the argument `x` and if they are the same, the value 1 is returned, otherwise 0 is returned.

The main feature of our technique is that it trades fewer recursive calls for multiple traversals of the list. The maximum number of simultaneous active invocations of this simple function is around $\mathsf{lb}\, n$, where n is the length of the list. The maximum value of this number is quite modest. On a 64-bit processor, it can never exceed 60 and it is significantly smaller in practice of course. The number of `cdr` operations can be approximately expressed as $n\,(1 + \frac{1}{2}\mathsf{lb}\, n)$. In Section 3.3 we analyze this result in greater depth.

The best case for this function is very efficient indeed. The worst case is unacceptably slow. Even for a list of some reasonable length such as a million elements, the execution time is a factor 6 slower than for the best case.

The remainder of this section is dedicated to ways of improving on the performance of the basic technique.

3.2 Using more stack space

By far the most important improvement to the basic technique is to take advantage of the available stack space to decrease the number of multiple list traversals required by the basic technique.

The following example illustrates this technique by using the simple recursive traversal if there are fewer than 10000 elements in the list.[7] If there are more elements, then it divides the list in two, just like the basic technique shown in Section 3.1.

```
(defun count-from-end-2 (x list)
  (labels ((recursive (x list n)
             (if (zerop n)
                 0
                 (+ (recursive x (cdr list) (1- n))
                    (if (eq x (car list)) 1 0))))
           (aux (x list n)
             (if (<= n 10000)
                 (recursive x list n)
                 (let* ((n/2 (ash n -1))
                        (half (nthcdr n/2 list)))
                   (+
                    (aux x half (- n n/2))
                    (aux x list n/2))))))
    (aux x list (length list))))
```

With this improvement, the number of `cdr` operations required can now be expressed as approximately

$$n\,(1 + \frac{1}{2}\mathsf{lb}\,\frac{n}{10000})$$

which is significantly better than the corresponding value for the basic technique.

However, there is no particular reason to divide the list into 2 equal-sized parts when there are too many elements for the basic technique. Section 4 gives a more complete explanation of the parameters involved and how they influence the execution time of the resulting code.

3.3 Analyses

In this section we give approximate formulas for the performance of our technique. The basic measure we are interested in is the number of `cdr` operations that must be performed as a function of the number of elements of the list. We will denote the number of elements of the list by N and the number of `cdr` operations required by $F(N)$. Since our technique always starts by traversing the entire list in order to compute N, we can always write $F(N)$ as $N + f(N)$, were $f(N)$ is the number of `cdr` operations required in the subsequent step.

For the basic technique where the list is divided into two equal-size sublists, we obtain the following recursive relation:

$$f(N) = \begin{cases} 0 & \text{if } N = 1 \\ \lfloor \frac{N}{2} \rfloor + f(\lfloor \frac{N}{2} \rfloor) + f(\lceil \frac{N}{2} \rceil) & \text{otherwise} \end{cases}$$

In order to obtain an approximate solution to this relation, we can solve for N being a power of 2, i.e., $N = 2^n$. In that case, for $N > 1$ we obtain:

$$f(N) = \frac{N}{2} + 2f(\frac{N}{2})$$

The details of the approximate resolution of this recursive equation is given in the appendix. This solution yields

$$f(N) = \frac{N}{2}\mathsf{lb}\ N + Nf(1) = \frac{N}{2}\mathsf{lb}\ N$$

Including the traversal to compute the number of elements of the list, we obtain:

$$F(N) = \frac{N}{2}\mathsf{lb}\ N + N = N(1 + \frac{1}{2}\mathsf{lb}\ N)$$

which is clearly $O(N\mathsf{log}\ N)$. More importantly, for a list with around 16 million elements (which fills the default heap of most implementations we have tested), we have $N \approx 2^{24}$ which gives $F(N) \approx 13N$ which is probably unacceptably slow.

Let us now consider what happens when we are able to handle more than a single element with the basic recursive technique, as shown in Section 3.2. We denote the number of

[7]The number 10000 was chosen to be a significant part of a typical per-thread default stack while still leaving room for stack space required by callers and callees of this function. In a real production implementation, the number would be chosen based on the remaining space left on the stack when the function is called.

elements that the basic recursive technique can handle by K, and again, in order to simplify the analysis, we assume that both N and K are powers of 2, i.e., $N = 2^n$ $K = 2^k$, and also that $N \geq K$. The recursion relation now looks as follows:

$$f(N) = \begin{cases} N - 1 & \text{if } N \leq K \\ \frac{N}{2} + 2f(\frac{N}{2}) & \text{otherwise} \end{cases}$$

The resolution of this equation is given in the appendix (Part B). It yields:

$$f(N) \approx N(1 + \frac{1}{2}\mathsf{lb}\ \frac{N}{K})$$

With the best portable version of our technique and a typical stack being able to handle $K = 2^{14}$ we are now looking at a performance for $N = 2^{24}$ of $F(N) \approx 6N$. Comparing this result to the technique of reversing the list, it is fair to say that the overhead of allocating and subsequently garbage-collecting a `cons` cell can very well be comparable to 6 times the time taken by the `cdr` operation. In other words, the performance of our portable version is already comparable to an implementation based on first creating a reversed copy of the list and then traversing that reversed copy.

Finally, instead of using more stack space for the base case, let us analyze what happens if we divide the original list into more than two parts. For this analysis, let us assume that we divide the list into M equal parts, and that M also is a power of 2 so that $M = 2^m$. We then obtain the following relation:

$$f(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - \frac{N}{M} + Mf(\frac{N}{M}) & \text{otherwise} \end{cases}$$

The resolution of this equation is given in the appendix (Part C). It yields:

$$F(N) \approx N(1 + \frac{\mathsf{lb}\ N}{\mathsf{lb}\ M})$$

While it may appear that we can get very good performance when M is chosen to be large, in practice, using large values of M introduces a different kind of overhead, namely large stack frames, making the gain less than what the formula suggests.

3.4 Implementation-specific solutions

So far, we have explored techniques that can mostly be implemented in portable Common Lisp. In this section, we explore a variation on our technique that requires access to the control stack of the implementation.

Recall that at the lowest level of our technique, there is a recursive function that is used for traversing the list when the number of elements is small compared to the stack size. At each invocation, this function does very little work.

With direct access to the control stack, we can convert the recursive function to an iterative function that pushes the elements of the list on the control stack, and then processes them in reverse order. This technique has several advantages:

- A single word is used for each element, whereas the recursive function requires space for a return address, a frame pointer, saved registers, etc. As a result, this technique can be used for lists with more elements than would be possible with the recursive technique, thereby further decreasing the number of times a list is traversed.
- There is no function-call overhead involved. The only processing that is needed for an element is to store it on the stack and then comparing it to the item.

We illustrate this technique in a notation similar to Common Lisp:

```
(defun low-level-reverse-count (item list length)
  (loop for rest = list then (cdr rest)
        repeat length
        do (push-on-stack (car rest)))
  (loop repeat length
        count (eq item (pop-from-stack))))
```

We implemented this technique in SBCL. In order not to have to recompile SBCL with our additional function, we used the implementation-specific foreign-function interface and wrote the function using the language C. Rather than pushing and popping the control stack, we used the built-in C function `alloca` to allocate a one-dimensional C array on the control stack to hold the list elements.

In SBCL, the default stack size is 2MBytes, or around 250k words on a 64-bit processor. We tested our technique using 100000 words on the stack. The result is that for a list with 10 million elements, our technique processes the list in reverse order as fast as an ordinary loop from the beginning of the list.

This surprising result can be explained by a few factors:

- Presumably in order to speed up the functions `car` and `cdr`, SBCL uses the same tag for `cons` cells and for the symbol `nil`. As a result, in order to traverse a list, SBCL must make *two* tests for each element, namely one to check whether the putative list is something other than a list altogether, and another to check whether it is a `cons` cell. When our technique traverses a list for which the number of elements is known, there is no need to make any additional tests, simply because when the length of the list is positive, the first element must be a `cons` cell.

- The SBCL compiler can not determine that the return value of `count` must always be a `fixnum`.[8] When the function is implemented in C, this problem disappears.

If we put this technique in the perspective of the analyses in Section 3.3, we can also see that the number of `cdr` operations remains quite modest, even for lists with a very large number of elements.

There are several variations on this implementation-specific technique. Some implementations might allocate a vector or a list declared to be `dynamic-extent` on the stack, thus giving essentially the same advantage as the version we implemented in C. However, such a technique would still be implementation specific, given that it is permitted for the compiler to ignore `dynamic-extent` declarations. In the case of SBCL, using such a declaration, we were able to obtain performance almost as good as our C version. However, as it turns out, SBCL only allocates a vector on the stack under certain circumstances thereby making this technique impossible to apply in general.

4. BENCHMARKS

We implemented ten different versions of `reverse-count`, a function that counts elements of a list from the end. The difference between these versions can be expressed in terms of two different numeric parameters, namely:

1. the minimum number of elements for which we apply the logarithmic method, consisting of dividing the list into two equal-size halves, and
2. into how many chunks do we cut the list when the number of elements is smaller than the first parameter, but larger than the number of elements that can be handled by the simple recursive technique.

In all of our versions, when the number of elements is less than 10000, we process the elements using the purely recursive technique where the element is processed in the backtracking phase.

For the purpose of this article, we have retained the one with the best experimental behavior (`v7`) and compared it to two more traditional versions (`v0` and `v1`).

These three versions can be characterized as follows:

0. Version `v0` uses the native `count` function called with the `from-end` keyword argument set to `t`,

```
;; standard version
(defun reverse-count-0 (x list)
  (count x list :from-end t :test #'eq))
```

[8]On a byte-addressed processor where n word-aligned bytes are needed to represent a `cons` cell, the number of elements in a list can be at most N/n where N is the maximum number of possible addresses. In a system that uses at most $\operatorname{lb} n$ tag bits for a fixnum, the value that `count` returns must be a fixnum. While some systems might use 8 tag bits, SBCL on a 64-bit platform uses a single tag bit for fixnums. As a consequence, `count` must then return a fixnum.

1. Version `v1` is the naive version consisting in reversing the list before counting; this version uses the heap space and no stack space.

```
(defun reverse-count-1 (x list)
  (declare (optimize
             (speed 3) (debug 0) (safety 0)
             (compilation-speed 0)))
  (loop for e in (reverse list)
        count (eq x e)))
```

7. Version `v7` divides the list in 2 parts if it has more than one hundred million elements. Otherwise, if it has more than 10000 elements, it divides it into chunks that have 10000 elements each. Finally, if it has no more than 10000 elements, then it uses the standard recursive method.

   We think this method is faster than the others, at least for lengths no more than one hundred million elements, because then it is guaranteed to traverse the list at most 3 times + 1 time for computing the length. It could be improved for lengths greater than one hundred million by using a better division than 2 in this case, but we have not attempted that improvement. The code is given below.

```
(defun count-from-end-with-length-7 (x list length)
  (declare (optimize (speed 3) (safety 0) (debug 0)
           (compilation-speed 0)))
  (declare (type fixnum length))
  (labels (;; AUX1 is the recursive traversal
           ;; by CDR.
           (aux1 (x list length)
             (declare (type fixnum length))
             (if (zerop length)
                 0
                 (+ (aux1 x (cdr list) (1- length))
                    (if (eq x (car list))
                        1
                        0))))
           ;; AUX2 recursive traversal
           ;; by (NTHCDR 10000 ...).
           ;; used when the length of the list is
           ;; less than 100000000.
           (aux2 (x list length)
             (declare (type fixnum length))
             (if (<= length 10000)
                 (aux1 x list length)
                 (+ (aux2 x
                          (nthcdr 10000 list)
                          (- length 10000))
                    (aux1 x list 10000))))
           ;; AUX3 recursive traversal
           ;; by half the size of the list.
           ;; used for lists that have more than
           ;; 100000000 elements.
           (aux3 (x list length)
             (declare (type fixnum length))
             (if (< length 100000000)
                 (aux2 x list length)
                 (let* ((n (ash length -1))
                        (middle (nthcdr n list)))
```

```
                  (+ (aux3 x middle (- length n))
                     (aux3 x list n))))))
    (aux3 x list length)))

(defun reverse-count-7 (x list)
  (count-from-end-with-length-7
   x list (length list)))
```

Thanks to the help of the Lisp community, we could test the behavior of these three versions on several implementations and architectures. In Figure 1, we summarize the results of tests that worked for a list of size up to 10^7. In many cases, the details of the implementation are unknown or not shown. However, the purpose of the Figure 1 is not to compare performance between different systems, but to compare the performance of different versions of `count` on each system. For that reason, the exact details of the system are unimportant; we are only interested in whether `v7` compares favorably to the other versions on most systems. Furthermore, for some implementations, we had to change the `optimize` settings and some other parameters in order to get our code to work.[9] For that reason, it is not possible to compare the performance on different implementations in Figure 1, even when the processor and the clock frequency are the same.

To get a better idea of the difference in performance between the three versions of the `count` function, we selected the table entry corresponding to the non-commercial implementation that resulted in the greatest advantage of our `v7` compared to the other versions (`Clozure CL 1.10-dev`), and we rendered the performance in the form of a graph. The result is shown in Figure 2.

As Figure 2 shows, the performance of `v7` is significantly better than that of the other versions. Furthermore, the fact that the curve for `v7` is smoother than the curves for other versions indicates that the performance of `v7` is more predictable. We attribute this behavior to the garbage collector, which occasionally has to run when heap allocation is required. Since `v7` does not require any heap allocation, the garbage collector is not solicited.

5. CONCLUSIONS AND FUTURE WORK

We have presented a general technique for processing elements of a list from the last element to the first. The implementation-specific version of our technique is comparable in speed to traversing the list from the first to the last element for all reasonably-sized lists. For very long lists, the performance of our technique degrades modestly and gracefully.

Even the implementation-independent version of our technique performs well enough that it is preferable to the existing technique of reversing the list that is used in some implementations.

We have presented our technique in the context of the function `count` because, together with `reduce`, processing the elements from the end is required by the HyperSpec, whereas, for other functions accepting the keyword argument `from-end`, it is explicitly allowed to process the elements from the beginning to the end.

However, our technique is potentially even more interesting to use with functions such as `find` and `position` for which it is not required to process the elements from the end to the beginning. The reason is that when elements are processed form the beginning to the end in these functions, *all* elements must be tested. When the combination of the `test` and the `key` functions has non-trivial computational cost, a significant amount of work may be wasted. However, when elements are processed in reverse order, a result can be returned when the test is satisfied the first time, thereby avoiding such wasted work.

Since the performance of our technique is not significantly worse than processing from the beginning to the end, it is very likely that our technique will be faster in almost all cases. Only when the last element of the list to satisfy the test is close to the beginning of the list will our technique apply the test as many times as when processing is done from the beginning to the end. We conjecture that, on the average, our technique will be faster whenever the cost of applying the test is at least that of a function call. If so, our technique should be used in all cases except for those using a very inexpensive test functions such as `eq`, and when the implementation then uses a special version of the sequence function where this test is inlined, so as to avoid a function call.

Further research is required in order to verify our conjecture. In order to determine the result with some accuracy, additional parameters have to be taken into account. In particular, the position of the last element in the list to satisfy the test must be taken into account, as well as the cost of calling the test function. As usual, benchmarks will have to be performed on a variety of implementations and processors, further complicating the verification of our conjecture.

6. ACKNOWLEDGMENTS

We would like to thank Pascal Bourguignon for reading and commenting on an early draft of this paper. We would also like to thank Alastair Bridgewater for helping us with the foreign-function interface of SBCL. Finally, we would like to thank Pascal Bourguignon, Alastair Bridgewater, James Kalenius, Steven Styer, Nicolas Hafner, and Eric Lind for helping us run benchmarks on platforms that are unavailable to us. Finally, we would like to thank Martin Simmons at LispWorks technical support for giving us the information we needed in order to explain the performance of our technique on the LispWorks Common Lisp implementation.

[9] In particular, LispWorks has a much smaller default stack than for instance SBCL (16k words, compared to 250k words) resulting in stack overflow of our benchmark with default parameters. For that reason, we ran the LispWorks benchmark with a smaller stack and with a higher value of `safety`. The combination of these factors is the likely explanation to the absence of performance improvement for LispWorks. However, we are told that on 32-bit LispWorks our technique gives a factor 10 improvement.

System characteristics				Time in seconds		
Implementation	Version	Processor	Frequency	v0	v1	v7
LispWorks	6.1.1	Intel Core	?	0.20	0.18	0.14
Clozure CL	1.10	Intel Xeon	3.33GHz	1.93	1.79	0.15
Clozure CL	1.10-dev	AMD FX	?	1.77	1.63	0.15
SBCL	1.2.8	Intel Xeon	3.33GHz	0.51	0.27	0.22
ABCL	1.3.1	Intel Xeon	3.33GHz	1.13	0.22	0.34
CLISP	2.49	X86_64	?	1.15	1.14	0.87
ECL	13.5.1	?	?	0.69	0.41	0.36
SBCL	1.2.7	Intel Core	2.53GHz	0.36	0.38	0.25

Figure 1: Performances of the three versions on several systems with a list of 10^7 elements

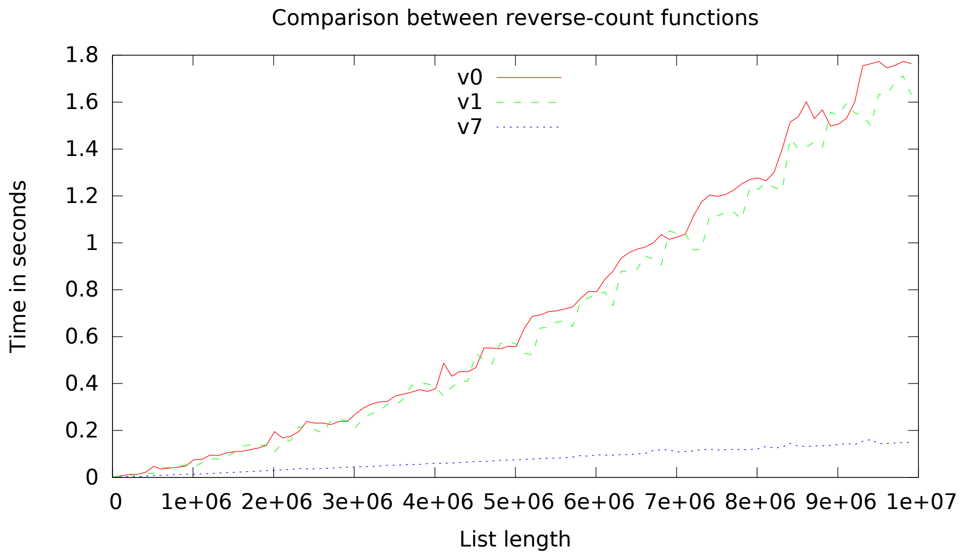

Figure 2: Comparison of the behavior of the three versions on a single system

APPENDIX

Part A

In this part, we develop the details of the approximate solution of the recursive equation defined in Section 3.3.

$$f(N) = \frac{N}{2} + 2f(\frac{N}{2})$$

Developing the second term of this equation one step, we obtain:

$$f(N) = \frac{N}{2} + 2f(\frac{N}{2}) =$$
$$= \frac{N}{2} + 2(\frac{N}{4} + 2f(\frac{N}{4})) =$$
$$= \frac{N}{2} + \frac{N}{2} + 4f(\frac{N}{4}) =$$
$$= 2\frac{N}{2} + 4f(\frac{N}{4})$$

Developing the second term of this equation one more step, we obtain:

$$f(N) = 2\frac{N}{2} + 4f(\frac{N}{4}) =$$
$$= 2\frac{N}{2} + 4(\frac{N}{8} + 2f(\frac{N}{8})) =$$
$$= 2\frac{N}{2} + \frac{N}{2} + 8f(\frac{N}{8}) =$$
$$= 3\frac{N}{2} + 8f(\frac{N}{8})$$

After developing the second term $p - 1$ times, we obtain:

$$f(N) = p\frac{N}{2} + 2^p f(\frac{N}{2^p})$$

When $p = n = \mathsf{lb}\ N$, this equation turns into:

$$f(N) = \frac{N}{2}\mathsf{lb}\ N + Nf(1) = \frac{N}{2}\mathsf{lb}\ N$$

Part B

In this part, we develop the details of the approximate solution of the recursive equation defined in Section 3.3.

$$f(N) = \begin{cases} N-1 & \text{if } N \le K \\ \frac{N}{2} + 2f(\frac{N}{2}) & \text{otherwise} \end{cases}$$

Since the second equation is the same as for the basic technique, developing the second equation $p-1$ times, we again obtain:

$$f(N) = p\frac{N}{2} + 2^p f(\frac{2^n}{2^p})$$

When $p = n - k = \mathsf{lb}\ \frac{N}{K}$ we get:

$$f(N) = \frac{N}{2}\mathsf{lb}\ \frac{N}{K} + \frac{N}{K} f(K)$$

Substituting $K-1$ for $f(K)$ and factoring out N, we obtain:

$$f(N) = N(\frac{1}{2}\mathsf{lb}\ \frac{N}{K} + \frac{K-1}{K})$$

Or:

$$f(N) = N(1 - \frac{1}{K} + \frac{1}{2}\mathsf{lb}\ \frac{N}{K})$$

Clearly, the term $\frac{1}{K}$ can be ignored, giving:

$$f(N) \approx N(1 + \frac{1}{2}\mathsf{lb}\ \frac{N}{K})$$

Part C

In this part, we develop the details of the approximate solution of the recursive equation defined in Section 3.3.

$$f(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - \frac{N}{M} + Mf(\frac{N}{M}) & \text{otherwise} \end{cases}$$

Solving as before, after developing the last term once, we obtain:

$$f(N) = N - \frac{N}{M} + Mf(\frac{N}{M}) =$$

$$= N - \frac{N}{M} + M(\frac{N}{M} - \frac{N}{M^2} + Mf(\frac{N}{M^2})) =$$

$$= N - \frac{N}{M} + N - \frac{N}{M} + M^2 f(\frac{N}{M^2}) =$$

$$= 2(N - \frac{N}{M}) + M^2 f(\frac{N}{M^2})$$

Developing the last term a second time, we obtain:

$$f(N) = 2(N - \frac{N}{M}) + M^2 f(\frac{N}{M^2}) =$$

$$= 2(N - \frac{N}{M}) + M^2(\frac{N}{M^2} - \frac{N}{M^3} + Mf(\frac{N}{M^3})) =$$

$$= 2(N - \frac{N}{M}) + N - \frac{N}{M} + M^3 f(\frac{N}{M^3}) =$$

$$= 3(N - \frac{N}{M}) + M^3 f(\frac{N}{M^3})$$

After developing the last term $p-1$ times, we obtain:

$$f(N) = p(N - \frac{N}{M}) + M^p f(\frac{N}{M^p}) =$$

Setting $p = \frac{n}{m} = \frac{\mathsf{lb}\ N}{\mathsf{lb}\ M}$ so that $M^p = N$, we get:

$$f(N) = \frac{\mathsf{lb}\ N}{\mathsf{lb}\ M}(N - \frac{N}{M}) + Nf(1) = \frac{\mathsf{lb}\ N}{\mathsf{lb}\ M}(N - \frac{N}{M})$$

Factoring out N, we obtain:

$$f(N) = N(1 - \frac{1}{M})\frac{\mathsf{lb}\ N}{\mathsf{lb}\ M}$$

and thus:

$$F(N) = N(1 + (1 - \frac{1}{M})\frac{\mathsf{lb}\ N}{\mathsf{lb}\ M})$$

and again:

$$F(N) \approx N(1 + \frac{\mathsf{lb}\ N}{\mathsf{lb}\ M})$$

7. REFERENCES

[1] *ISO 80000-2:2009 Quantities and Units – part 2: Mathematical signs and symbols to be used in the natural sciences and technology.* International Organization for Standardization, 2009.

[2] R. J. M. Hughes. A novel representation of lists, and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.