In-Situ, Stable Merging by way of the Perfect Shuffle. *

John Ellis and Minko Markov
Department of Computer Science
University of Victoria, P.O. Box 3055
Victoria, British Columbia, V8W 3P6, Canada

October 28, 1999

Abstract

We introduce a novel approach to the classical problem of in-situ, stable merging, where "in-situ" means the use of no more than $O(\log^2 n)$ bits of extra memory for lists of size n. Shufflemerge reduces the merging problem to the problem of realising the "perfect shuffle" permutation, that is, the exact interleaving of two, equal length lists. The algorithm is recursive and so uses a logarithmic number of variables. Thus it uses more than "absolutely minimum storage", i.e., $O(\log n)$ bits.

A simple method of realising the perfect shuffle uses one extra bit per element, and so is not in-situ. We show that the perfect shuffle can be attained using absolutely minimum storage and in linear time, at the expense of doubling the number of moves, relative to the simple method.

We note that there is a worst case for Shufflemerge requiring time $\Omega(n \log n)$, where n is the sum of the lengths of the input lists. We also present an analysis of a variant of Shufflemerge which uses a generalised shuffle and which has a provable average case time complexity of $O(n \log \log m)$, where m is the length of the shortest input list. It is unlikely that the generalised shuffle can be achieved in-situ.

Linear time, in-situ, stable merging has previously been demonstrated. We present experimental evidence indicating that Shufflemerge, although almost certainly not asymptotically linear, might be of value in practice.

The relative simplicity of the basic method, particularly with respect to stability, also recommends it.

*This work was supported by the Natural Sciences and Engineering Research Council of Canada

1 Introduction

We describe a novel merging algorithm with certain properties of theoretical interest and possibly of practical value. The merging problem is to produce one sorted list from an input of two sorted lists. The properties of interest are execution time, memory usage and "stability".

An "in-situ" merge, sort or other permutation, rearranges the subject elements within the space that they occupy, in contrast to the standard merge algorithm which duplicates the input space. We use the precise but tight definition of "in-situ" [Knu73] (Chapter 5, Section 5, Exercise 3) which allows the use of no more than $O(\log^2 n)$ bits over and above that space occupied by the elements themselves, where n is the number of elements. This definition permits recursion, so long as the stack depth is restricted to $O(\log n)$, but does not allow the use of extra space proportional to n. It also implies that the lists be represented by arrays, or parts of arrays. The extra space used by the pointers in a linked list representation would be at least (*pointer size* $\times$ n), and each pointer would require at least $\log n$ bits. Originally, what we are here referring to as "in-situ", was called "minimum storage", e.g., in the citation just given [Knu73]. An even tighter restriction, sometimes referred to as "absolutely minimum storage", permits only $O(\log n)$ bits, i.e., a constant number of variables.

Stability is the property of a merge or sort which guarantees that the order of equal elements in the input is preserved in the output. A finer distinction is sometimes made between "weak" and "strong" stability. The former permits a pair of equal elements from different lists to be arranged in either order. The latter requires that priority be guaranteed for elements from one designated list. Stability is essential when one wants to sort successively on more than one key.

The standard merge algorithm is simple, stable, and uses only linear time but it duplicates the input space. An in-situ merge is desirable in circumstances such as merging or sorting large external files, where the saving in space can permit the entire process to complete faster since larger files can be merged at each step. Further, in-situ, stable, linear time merging immediately yields in-situ, stable sorting in (asymptotically) optimal time, by way of a sort by repeated merging.

The problem of constructing an algorithm with all three desirable properties has a long history. It was first posed in [Knu73]. At that time, a linear time, in-situ but unstable merge was known, [Kro69]. It was based on the idea of dividing up the lists into about $\sqrt{n}$ blocks of size about $\sqrt{n}$. Almost all subsequent developments, [Hor78] is one exception, have been based on this principle. Early solutions, [Par77, Hor78], are generally considered to be both complex and, although linear time, to have impractically large constants of proportionality, i.e., greater than $\log n$ for "practical" n. Further, [Hor78] modifies keys.

A fairly straightforward algorithm using $O(\sqrt{n})$ extra space can be based on the $\sqrt{n}$ block method by using two extra $\sqrt{n}$ buffers for the elements plus space for $\sqrt{n}$ pointers. This method is described in [TB95] and is likely faster than any other known method using less than linear extra space.

A practical solution should be both programmable within reasonable time and with ordinary skill, and should have a time penalty, relative to the standard algorithm, that is not so great as to make the space saving irrelevant. The first, in-situ, linear time solution, that is arguably practical is given in [HL88], where an unstable method is described, and in [HL92], where a stabilising technique is added. This method uses absolutely minimum storage. In this paper we report some experimental results obtained using both our method and this one, which, for convenience, we call the "HL-merge".

More recent work [Sym95], besides satisfying the three basic requirements, achieves an optimal $O(m\log(p/m+1))$ number of comparisons (where m and p are the input list lengths and $m \leq p$). The authors say that the stabilising techniques used in the paper are "powerful enough to make stable all the existing linear in-place unstable algorithms we are aware of". However, in that paper no estimates for the constants of proportionality are given and no experimental results are presented.

Probably the most recent results in this area are reported in [Pas99]. In that thesis a variety of absolutely minimum storage, stable/unstable, merging and sorting algorithms are presented and analysed. For unstable merging $3(n+m)+o(n)$ moves and $m(t+1)+n/2^t+o(m)$ comparisons are used, where $m \leq n$ are the input list lengths and $t = \lfloor\log(n/m)\rfloor$. For stable merging $5n + 12m + o(m)$ moves and the same number of comparisons are used. Many of these results have been, or are about to be, published, for example see [KP94], [KPT96] and [GKP].

The interesting and useful features of the new algorithm are:

- The method is novel and further development may be possible. It reduces the problem of merging to the problem of realising the permutation known as the "perfect shuffle".
- The best upper bound on average time complexity that we can prove analytically is $O(n\log\log n)$ and this is not for the basic algorithm but for a variant of the basic algorithm which can not use the in-situ shuffle method, but must use the one extra bit per element. There is an $\Theta(n\log n)$ worst case. However, we present empirical evidence indicating that our method might be of practical value.
- Stability falls out very naturally and simply in our method, and imposes no time penalty (in terms of element comparisons and movements). In [HL92], [Sym95] and in [Pas99] stability is achieved at the expense of extra program complexity and a significant time penalty.

- The method is distinctly simpler, and hence easier to program, than any of those so far cited.

Definition 1.1 *The perfect shuffle is defined for two equal length lists, say L and R. The effect is to intersperse the elements of one list amongst the other, in a perfectly even manner, and such that the first element of R, which we can imagine to be the "right" input list, becomes the first element of the result. For example, if $L = a, b, c, d$ and $R = 1, 2, 3, 4$, then the result of Shuffle(L, R) is the list:* $1, a, 2, b, 3, c, 4, d$.

We reduce merging to perfect shuffling, which we refer to as just "shuffling" for convenience. The time and space complexity of Shufflemerge is clearly dependent on the time and space complexity of the shuffling algorithm. We describe a simple shuffling algorithm which uses linear time, but which requires one extra bit per list element. Thus that method is not in-situ, though, in practice, one bit per element may often be immaterial .

We go on to show that the extra bits can be dispensed with, at the expense of approximately doubling the number of moves required. Thus the perfect shuffle itself, can be realised in absolutely minimum space and in linear time, which we do not know to have been demonstrated previously.

2 The Algorithm

The algorithm reduces the problem of merging to the problem of realising the "perfect shuffle" permutation and its inverse. A novel feature of the method is that a large number, in some cases all, of the list elements are moved before any comparisons are made. All movement of elements is effected by way of the "perfect shuffle" permutation, or its inverse. Shuffling was defined at the end of the introduction. How shuffling can be realized is discussed in Section 7. We refer to the shuffling procedure as "Shuffle" and to its inverse as "Unshuffle".

It follows immediately from the definition that, if L and R are ordered, then the list resulting from Shuffle is *2-ordered*, i.e., for all i, the i^{th} element is less than or equal to the $(i+2)^{th}$ element, if the latter exists. Likewise, the effect of Unshuffle on a 2-ordered list is to produce two, ordered lists.

A 2-ordered list resulting from a Shuffle of ordered lists is not necessarily itself ordered. Picturing the constituent, ordered lists as being in increasing order from left to right, we use the term *d-string* (d for disordered) to denote a maximal, even length, 2-ordered list segment in which the leftmost element is greater than the rightmost. The maximality is defined algorithmically. A *leftwards* d-string is maximal with respect to a leftwards scan from its rightmost element, i.e., starting from the i^{th} element the algorithm scans the $(i-1)^{th}$, the $(i-3)^{th}$ etc. until an element less than or equal to the i^{th} is found. If that element is the j^{th}, then the d-string extends from the $(j+2)^{th}$ to the i^{th}. If the $(i-1)^{th}$ is less than or equal to the i^{th}, then there is no leftwards d-string starting at the i^{th} element. Likewise, a *rightwards* d-string is maximal with respect to a rightwards scan from its leftmost element.

First we give an informal description of the algorithm. Let the left and right lists, input to a merge be called L and R. They are not necessarily of equal lengths, whereas the shuffling operation is defined only on equal length lists. Assume that $|L| \leq |R|$. If not, use the mirror image of this description.

- Cut off a "tail" from the right end of R, such that the remainder is the same length as L. Apply Shuffle to the *interior* elements of the list comprising the concatenation of L and this remainder. Because the goal is to produce a 2-ordered string and because of the way we have defined Shuffle, there is no need to move the end elements.
- Scan the shuffled segment plus the end elements from right to left, delineating leftwards d-strings.
- Unshuffle the d-strings, which are by definition of even length.
- Recursively merge adjacent segments resulting from the unshuffling of adjacent d-strings. Include a merge of the rightmost unshuffled segment, if any, with the "tail" of R, which was cut off in step 1.

Figure 1 illustrates the method. In this example, $|L| < |R|$. The figure shows one application of the shuffle/scan/unshuffle process. A few particular points should be noted.

- We obtain a 2-ordered string by shuffling just the interior elements of the list formed by the concatenation of the two lists to be shuffled. It is not necessary to include the end elements.
- In this example the scan proceeds from right to left. In the opposite direction, different d-strings would be obtained.
- Unshuffling the leftmost d-string yields the list 6, 7 as the right list and unshuffling the adjacent d-string yields 4, 5, 8 as the left list. But we do not need to include 8 in the recursive merge. We already know that the 8 element is greater than the 7 element because the scan ensured that the d-string starting at 8 is maximal to the left.

Figure 2 is a Pascal realisation of the algorithm. We omit the code for the case in which the right list is shorter than the left list. That code is just the mirror image of that given.

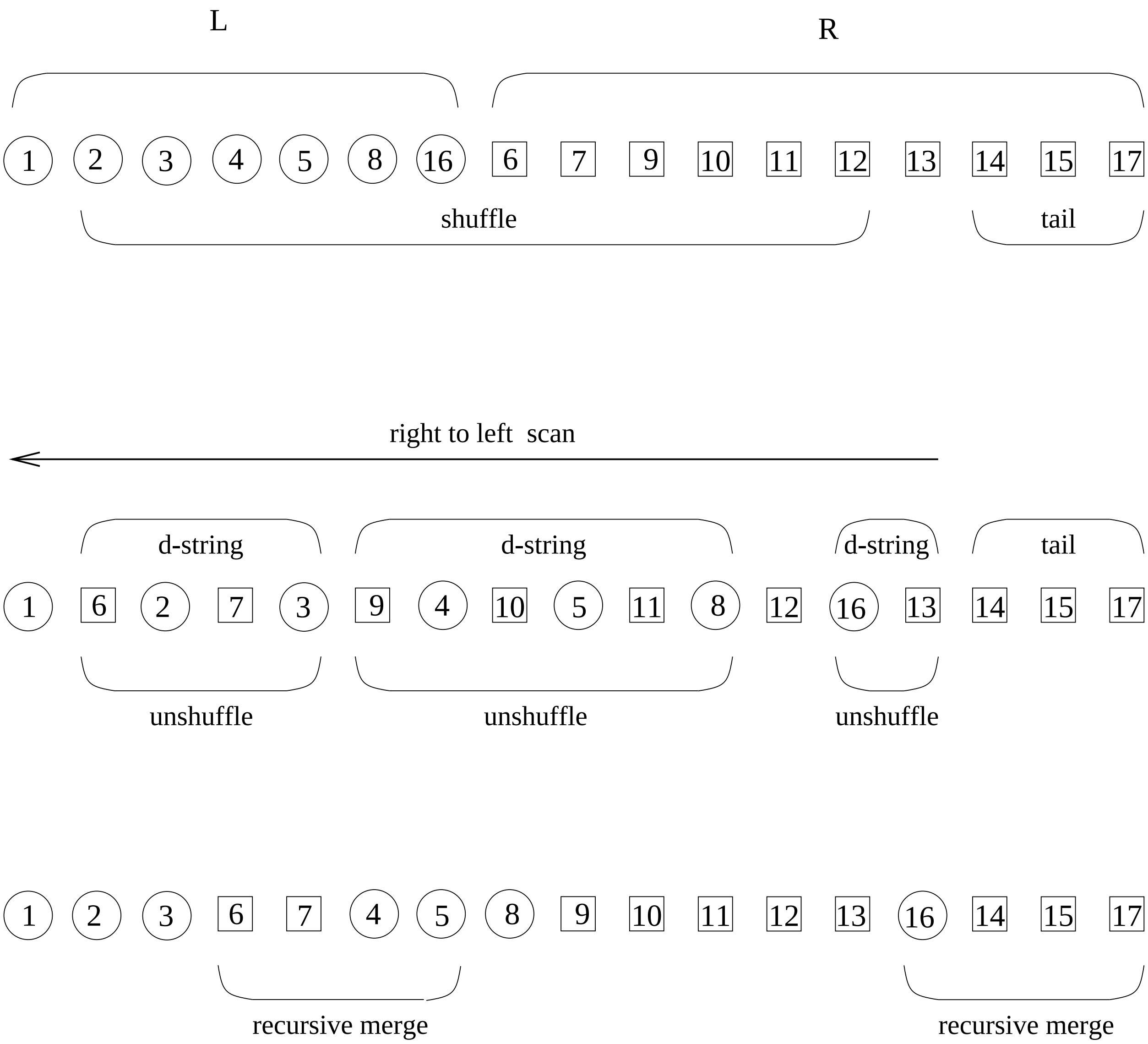

Figure 1: The Operation of Shufflemerge

```
procedure Shufflemerge (first1, last1, last2 : integer);

{ Merges that  segment of an array A from A[first1] through A[last1] with
  the adjacent segment A[last1+1] through A[last2].  We assume that both
  segments are ordered. }

var  high, mid, low : integer;

begin
      {Check that both lists are non-empty}
      if (last1 - first1 >= 0) and (last2 - last1 >= 1) then
      if (last1 - first1) < (last2 - last1 - 1) then
      begin
         {Left list is shorter than right list}
         mid := 2 * last1 - first1 + 1;  low := mid - 1; high := last2 + 1;

         Shuffle (first1 + 1, mid - 1);

         while mid > first1 do
         begin
            {Scan, right to left, for next d-string}
            while (low >= first1) and (A[low] > A[mid]) do low := low - 2;

            if low < mid - 1 then {a d-string exists}
            begin  {Unshuffle d-string}
               Unshuffle (low + 2, mid);
               {Merge adjacent parts of unshuffled, adjacent d-strings}
               Shufflemerge ((mid + low + 3) div 2, mid, high - 1);
               high := (mid + low + 1) div 2; mid := low + 1
            end
            else {A[mid] is in correct position}
            begin  mid := mid - 1; low := low - 1; high := mid  end
         end
      end
      else  {Left list is longer than, or same length as, right list.
             Mirror image of previous code }
end;
```

Figure 2: The Merge Program

The way this procedure works can be understood with the help of Figure 3, which illustrates the relationships between the various indices used in the Pascal procedure. Correctness can be verified by showing that the following proposition is a loop invariant for the outer *while* loop, and by applying induction on the length of the lists. We note that the argument is valid whether or not equal elements exist. We use $[x, y]$ to denote that segment of the array between and including $A[x]$ and $A[y]$. If $y < x$, $[x, y]$ denotes the empty segment. The first item in the proposition guarantees that $[first1, last2]$ is ordered at exit from the loop, since then $mid \leq first1$.

Proposition 2.1

- $[mid, last2]$ *is ordered and*
- *no element in* $[high, last2]$ *is less than any element in* $[first1, high - 1]$ *and*
- $[first1, mid]$ *is 2-ordered*

Termination is guaranteed by noting that *mid* is necessarily decreased at each loop iteration and that the recursive calls are on lists necessarily smaller than the input.

3 Stability

The algorithm as it stands is not "strongly" stable. However, since it never transposes or even compares elements from the same list, it is "weakly" stable. To achieve strong stability it is necessary and sufficient to know, whenever a comparison occurs between equal elements, which element was originally from the left list and which from the right.

It is very easy in Shufflemerge to keep track of which lists the current elements being compared ($A[low]$ and $A[mid]$) came from. Suppose we maintain a Boolean flag, call it *low-left*, which will be true iff $A[low]$ came originally from the left list.

As the shuffled, 2-ordered segment is scanned by the inner *while* loop *low-left* should be unchanged, because *mid* is constant and *low* is decremented by 2. As the outer *while* loop is iterated, whether or not *low-left* should be complemented depends on whether the *then* block or the *else* block was executed in the body of the loop. If the *then* block was executed *low-left* should not be changed because *low* is not changed inside the *then* block. However, in the *else* block, *low* is decremented by one, and so *low-left* should be complemented.

Of course the value of *low-left* must be passed down to lower levels of recursion. When a recursive call is made, the left list input was from a right list at the current level of recursion,

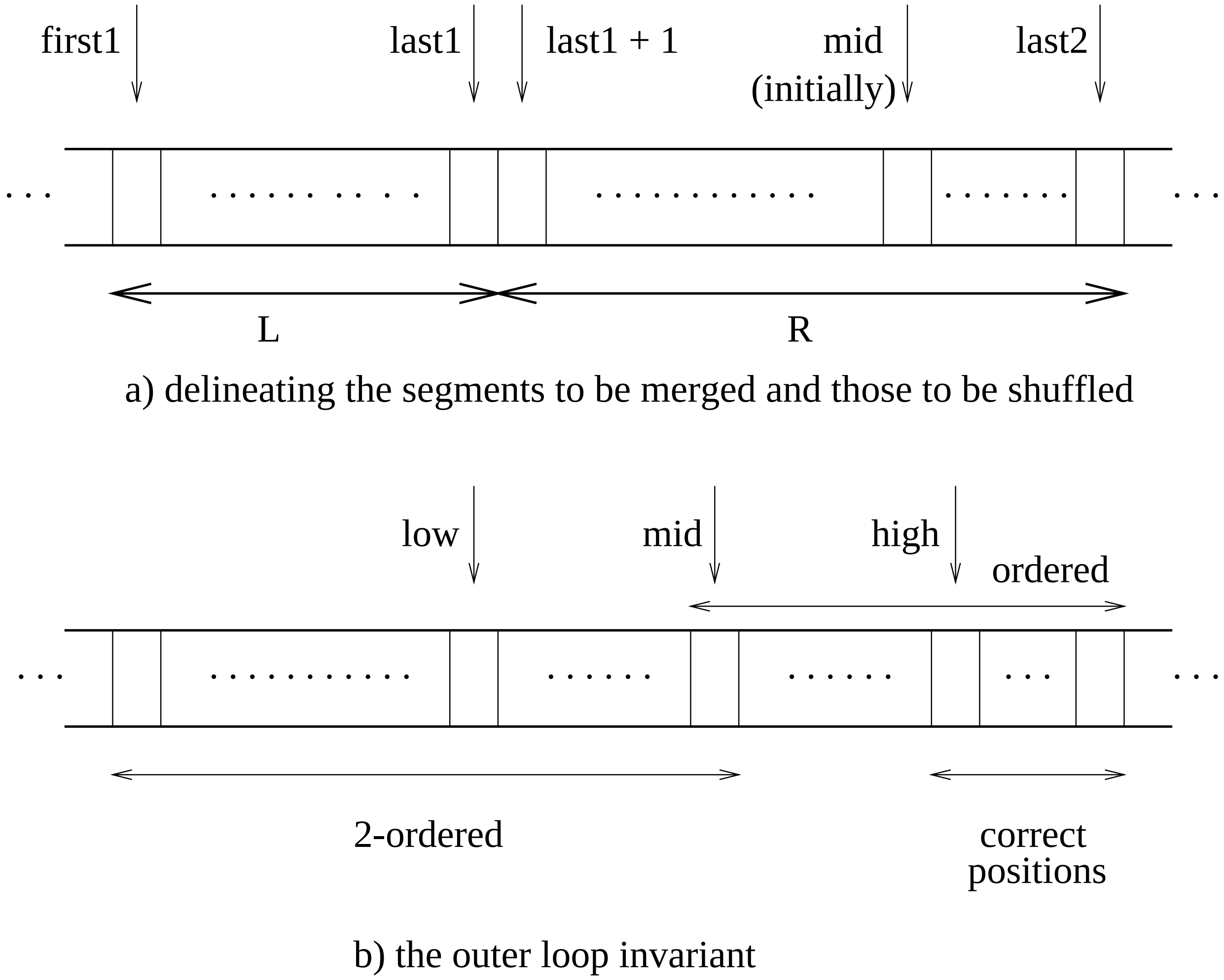

Figure 3: The Meaning of the Program Variables

and *vice versa*. Hence *low-left* should be complemented. This is all that is necessary to achieve stability.

Note also that, although the flag now needs to be checked at every element comparison, no more element comparisons are required. The flag check is used to choose between a "greater than" operator, if *low-left* is true, or a "greater than or equal to" otherwise. Alternatively, one could use a "state programming" approach in which the program would flip between code for the "greater than" state and the "greater than or equal to" state, thus further reducing flag tests.

By contrast, adding stability to the $\sqrt{n}$ block methods has proved challenging. For example, the relatively simple block merge in [HL88] is unstable, a stable version was described later in [HL92]. Stability is there obtained at a noticeable cost in both program complexity and time, the authors estimating that the number of moves required is more than doubled relative to the unstable version. The same observation applies to the results obtained in [Pas99] and quoted in the introduction.

4 Space Complexity

We note that, for the algorithm as presented, run-time stack depth is limited only by the length of the input. Consider the case where the left list contains one element which is greater than anything in the right list. The algorithm will progress by repeatedly comparing the single left list element with elements from the right list. The single left list element will progress in unit steps through the right list until it reaches the right hand end. But this is done by a sequence of recursive calls of length equal to the length of the right list. In this instance extra space used on the run time stack is *list length* $\times$ *space required for procedure variables*, which contravenes the in-situ definition.

We recall that, unmodified, Quicksort has a similar worst case. To ensure that stack depth is limited to $O(\log n)$ one must explicitly test segment lengths and stack the longer of the two partitions produced by the partition procedure. Otherwise, the same problem can arise.

Likewise, a simple modification can ensure that Shufflemerge never requires a stack depth exceeding $O(\log n)$. Consider the code in Figure 2. Note that there is exactly one recursive call to Shufflemerge, in the middle of the loop which scans for d-strings. Note also that for all these recursive calls, *except* possibly the first, the input lists are both parts of unshuffled d-strings. The length of an input derived from a d-string can not exceed half the length of the shortest original input list, by definition of a d-string. So a sequence of recursive calls on these inputs can not generate worse than a logarithmic stack depth. This is not necessarily true for the first recursive call, which may involve the "tail" cut off before the shuffle. That is the only possibility that can lead to a recursion depth worse than logarithmic.

Consequently, we can avoid that bad possibility by not invoking the procedure recursively on the tail. This can be done by combining iteration and recursion. We replace the if statements at the head of the code by:

while (both lists are not empty) **do**

The rest of the code remains unchanged *except* that must we ensure that no recursive call is made involving the tail. Rather, at the end of the loop, we reset the indices defining the lists so as to define the tail and the list to be merged into it. Of course, if either is empty, the iteration terminates. The result is that the merge into the tail, if necessary, is accomplished by iteration rather than by recursion.

We can see that this modification does not change the time complexity of the algorithm by noting that we are not changing the actions of the algorithm, just the order in which they are done.

Because recursion depth is limited to $O(\log n)$ and the procedure uses a fixed number of variables, each requiring $O(\log n)$ bits, the procedure uses $O(\log^2 n)$ bits, plus whatever is used by the Shuffle/Unshuffle procedures. In Section 7 we show that shuffling can be achieved either by the use of an extra bit/element, or in absolutely minimum space at the expense of more moves.

5 Time Complexity

An analysis of the time complexity of Shufflemerge, as defined in Figure 2, is not trivial. We show that the worst case time complexity of the algorithm is bounded by $O(n \log n)$ and that a family of instances with this behaviour exists. We are unable to analyse the average behaviour of the basic algorithm but we do establish an average case time complexity of $O(n \log \log n)$ on a modified version of the basic algorithm. A better bound might be obtainable by a more sophisticated analysis. We recall that Quicksort has an $\Omega(n^2)$ worst case performance, but is excellent in the average case and in practice.

5.1 An Upper Bound on the Worst Case

We establish an $O(n \log n)$ upper bound on the worst case time complexity, where n is the sum of the input list lengths. We work with the unmodified algorithm, as defined in Figure 2.

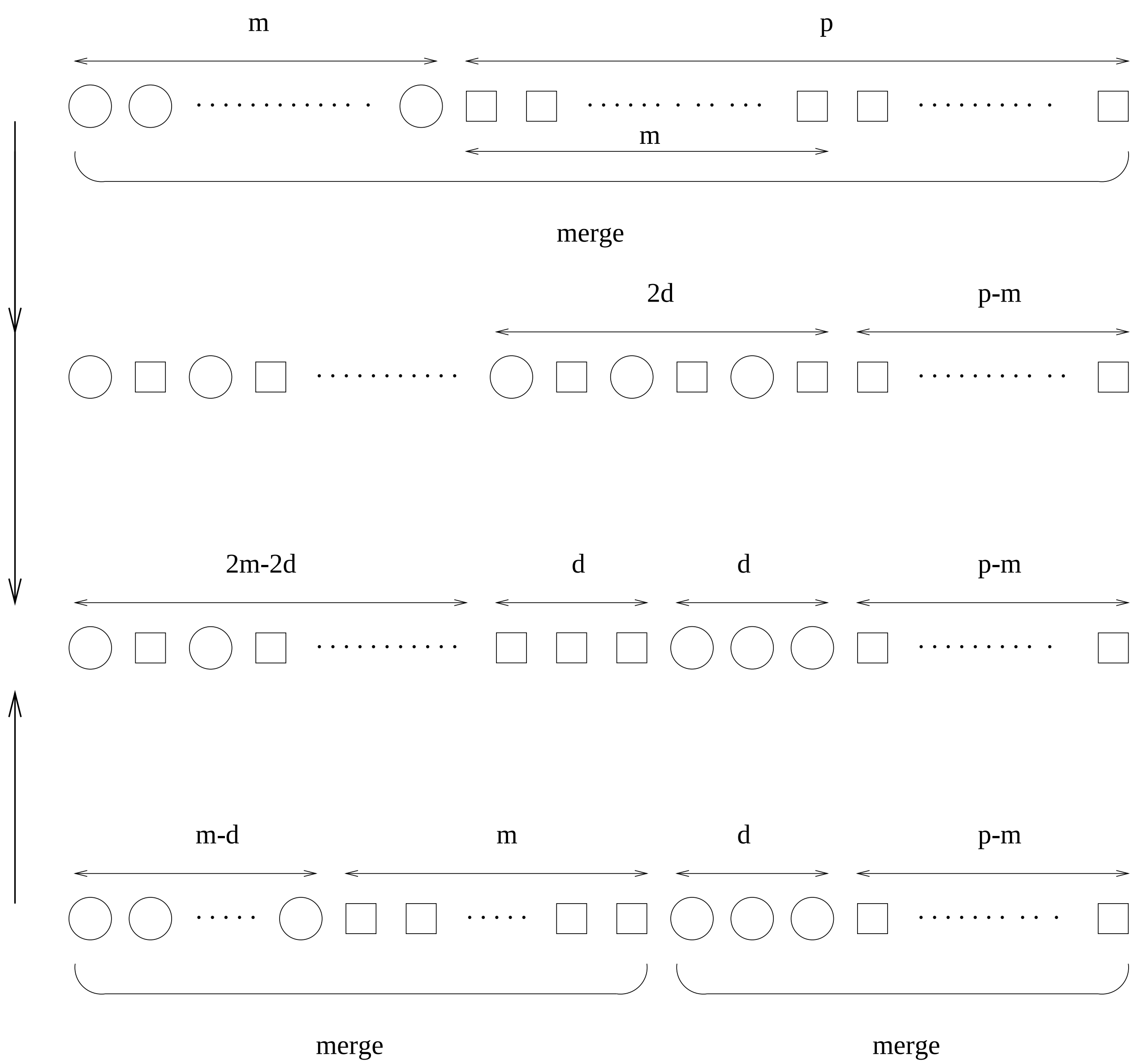

Figure 4: Setting up the Recurrence Relation for the Worst Case

Consider Figure 4. Without loss of generality, let the length, m, of the left list be less than the length, p, of the right list. The first line in the diagram represents the input lists. The second line represents the situation after the left list is shuffled with the leftmost m elements of the right list. Suppose the rightmost d-string in the second line is of length $2d$, the third line represents the situation after the unshuffling of that d-string.

Now consider the bottom line. This represents two separate merges on lists comprising undisturbed segments from the original lists. In the left merge we have the first $m - d$ elements from the original left list and the leftmost m elements from the original right list. In the right merge we have the rightmost d elements from the original left list and the rightmost $p - m$ elements from the original right list. Consider the effect of the first step of the left merge, namely a shuffle on the $2m - 2d$ leftmost elements. Note that we arrive again at line 3.

The difference in the work done in going from line 1 to line 3 and going from line 4 to line 3 is proportional to the length of the d-string. We see this by noting that from line 1 to line 2 we shuffle $2m$ elements, using say bm time for some constant b. From line 2 to line 3 we scan and unshuffle a d-string, using say cd time, for some constant c. To go from line 4 to line 3 we shuffle $2m - 2d$ elements, using time $b(m - d)$. So the difference is $(b + c)d$ From line 3 onwards, the process would continue in the same manner, no matter the different starting points. This observation justifies the following recurrence relation, where $T(m,p)$ is an upper bound on the time required to merge a total of $m + p$ elements.

$$T(m,p) \leq c_1 d + T(m - d, m) + T(d, p - m) \tag{1}$$

for some constant c_1. Assume that the time complexity of the algorithm is $O(n \log n)$, where $n = m + p$, for smaller n, so that, for some c_2 and all sufficiently large n:

$$T(m,p) \leq c_2 n \log n$$

Substituting in Equation 1 yields:

$$T(m,p) \leq c_1 d + c_2(2m - d)\log(2m - d) + c_2(p - m + d)\log(p - m + d) \tag{2}$$

Consider positive numbers a and b such that $a \leq b$. By writing $b = ka$ for some $k \geq 1$ we can show that:

$$a \log a + b \log b < (a + b)\log(a + b) - a \tag{3}$$

Take a to be the smaller of $(2m - d)$ and $(p - m + d)$, and b to be the larger. Then Equation 2 can be written:

$$T(m,p) \leq c_1 d + c_2 a \log a + c_2 b \log b$$

Using Equation 3 yields:

$$T(m,p) \le c_1 d - c_2 a + c_2(m+p)\log(m+p)$$

Each of $(2m-d)$ and $(p-m+d)$, is at least d, since $m \ge d$ and $p \ge m$. Hence if we choose $c_2 \ge c_1$, then we ensure that $T(n) \le c_2 n \log n$, and the hypothesis is confirmed.

5.2 Worst Case Instances

We define an infinite family of instances that require time $\Omega(n \log n)$, were $n = m + p$. Suppose the right list is exactly twice the length of the left list, i.e., $p = 2m$, and let m be a power of two.

The first invocation of shuffle will be on the left list and the leftmost m elements of the right list. Suppose that the lengths of the d-strings resulting from the shuffle are, from right to left: $m, m/2, m/4, \cdots$. Then, after unshuffling, the recursive calls will be on lists for which, again, the left list is always one half the length of the right list. Further, since no other information is known about the relationships between elements in opposing lists, we may assume that the same thing happens at all lower levels of recursion.

The following recurrence relation is therefore justified:

$$T(m, 2m) \ge cm + \sum_{i=1}^{\log_2 m} T(2^{i-1}, 2^i)$$

for some constant c.

The solution is $T(m, 2m) \ge bm \log_2 m$ for some constant b and for sufficiently large m. This can be justified by noting that the substitution of the alleged solution into the summation yields: $b((k-2)2^k + 2)$, when $m = 2^k$. Hence the right hand side of the above equation is at least $bm \log m = bk2^k$ so long as we choose b so that $c - 2b > 0$. Hence the algorithm has time complexity $\Omega(n \log n)$ on the family of instances considered.

We conclude then that no instance requires more than $O(n \log n)$ and that instances causing this behaviour do exist.

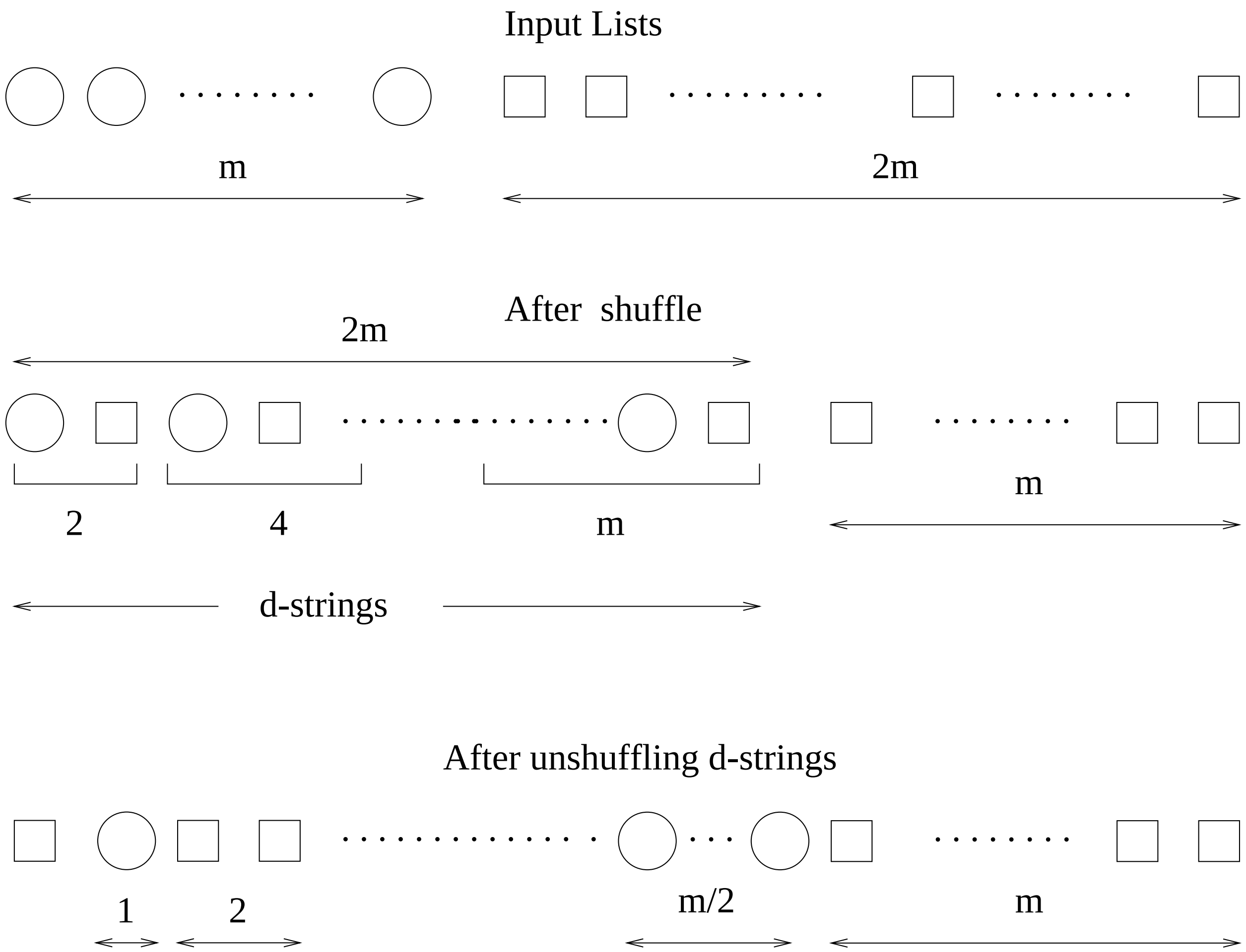

Figure 5: A Worst Case Instance

5.3 Average Case

By average we mean the arithmetic mean over all instances of the problem. There are $\binom{m+p}{m}$ possible ways of arranging a merge input characterised by the lengths, m and p, of the input lists. We establish that the average case time complexity of a modified version of the algorithm is $O((m+p)\log\log m)$. This result does not preclude the possibility that a better upper bound may be obtainable by a more sophisticated analysis.

We do not know how to analyse the average behaviour of the basic algorithm so far described. One reason is that the behaviour is dependent on the ratio of the input list lengths. Although we present, in the next section, some experimental evidence showing that the average behaviour is relatively good on lists of approximately equal length, we have just seen that the behaviour on lists of lengths m and $2m$ can be expected to be poor, because the most likely distribution is exactly the one with worst case behaviour. In order to get some analytical insight into the algorithm's behaviour we propose a variation of the basic algorithm which can be analysed. We are not necessarily recommending this method as having practical value.

Let an (m,p)-merge denote a merge instance where $m \leq p$. Instead of using of the standard shuffle, we use what we will call a "generalised" shuffle. This latter, no matter the ratio m/p, will distribute the elements of the shorter list evenly among the elements of the longer list. Let us accomplish this by specifying that element i in the left m-list is placed at position $\lceil i(p+m)/m \rceil$ and that the elements from the right p-list are placed in input order in the gaps. Thus the elements of the shorter list can be viewed as dividing up the longer list into segments of length either $\lfloor (p/m) \rfloor$ or $\lceil (p/m) \rceil$. We discuss further the feasibility of such a procedure in Section 7. For the purposes of this section, let us assume that such a process exists and requires linear time. Likewise, we assume that a "generalised unshuffle" exists.

A modified scan process need only scan in one direction. If the first element in the scan is from the shorter input, then blocks from the longer input are scanned until an element greater than the first is found. This may be in the middle of a block. If the first element in the scan is from the longer input, then elements from the shorter input are scanned until an element greater than the first is found, as in the basic algorithm.

A "generalised" d-string will now consist of individual elements from the shorter list interspersed among blocks from the longer. The first or last block may be partial. The *m-length* of a generalised d-string will be taken to be the number of elements from the shorter input list in the d-string, or, equivalently, the number of blocks of elements from the longer input list.

Recursion will terminate in this modified merge when the length of one list is reduced to one. At that point, a simple procedure such as repeated exchanges can be used to bring the

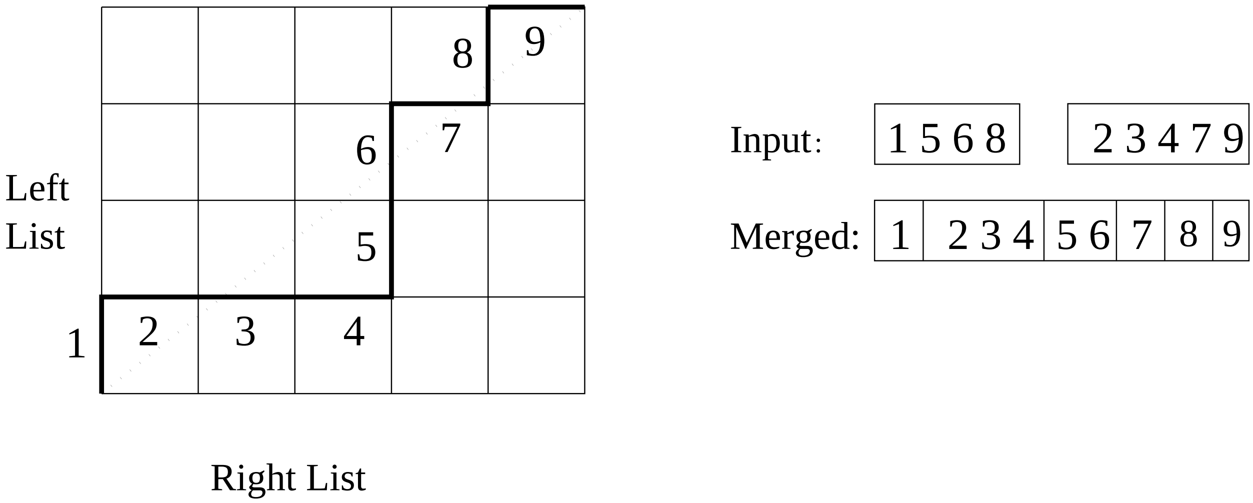

Figure 6: Correspondence between Lattice Paths and Merges

single element to its correct position in the longer list.

We first put an upper bound on the expected length of generalised d-strings resulting from the application of the generalised shuffle. We use the fact that there is a bijection between instances of merging two lists of length m and p and monotonic paths between the point (0,0) and the point (m, p) in the m by p lattice. Figure 6 shows the path in the 4 by 5 lattice corresponding to one instance of merging lists of length 4 and 5. Each of the $\binom{m+p}{m}$ possible paths corresponds to one of the $\binom{m+p}{m}$ possible outcomes of the merge.

Suppose that an (m, p)-merge instance, where $m \leq p$, is represented by an $m \times p$ lattice where m is the vertical dimension. Let the *diagonal* of the lattice have the obvious geometrical interpretation and let the *height* of a point on a lattice path be the vertical distance of the point from the diagonal. By *almost all paths* having some property, we mean that the fraction of paths having the property tends to one as the list lengths tend to infinity. We want to put an upper bound on the maximum height of any point on a lattice path which holds for almost all paths.

Lemma 5.1 *The probability that a path deviates, for at least one of its points, from the diagonal by a vertical distance greater than $m^{0.5+\epsilon}$, for any $\epsilon > 0$, tends to zero as m and p tend to infinity in such a way that m/p remains constant.*

Proof

We use a result due to Smirnov [Smi39],which is quoted in many texts on statistics. It gives the asymptotic distribution of $D_{m,p}$, where, if d is the absolute deviation of the path from the diagonal, $D_{m,p} = d/m$. By asymptotic is meant: as m and p tend to infinity in such a way that m/p remains constant. The Smirnov formula for the probability P is:

$$\lim_{m,p\to\infty} P\left(\sqrt{\frac{mp}{m+p}} D_{m,p} \le z\right) = L(z)$$

where

$$L(z) = 1 - 2\sum_{i=1}^{\infty}(-1)^{i-1}\exp(-2i^2z^2) \tag{4}$$

If $D_{m,p} \le m^{0.5+\epsilon}/m$ then $z \ge m^{\epsilon}\sqrt{\frac{p}{m+p}}$. But $p/(m+p) > 1/2$, so $z > m^{\epsilon}/\sqrt{2}$. Hence

$$L(z) > 1 - 2\left(\frac{1}{\exp(m^{2\epsilon})} - \frac{1}{\exp(4m^{2\epsilon})} + \frac{1}{\exp(9m^{2\epsilon})} + \cdots\right)$$

Thus $L(z) \to 1$ as $m \to \infty$. $\square$

Lemma 5.2 *For almost all instances of generalised shuffles, the m-length of the longest generalised d-string resulting from the shuffle is* $O(m^{0.5+\epsilon})$ *for any* $\epsilon > 0$.

Proof

Consider the effect of applying the basic shuffle to lists of equal length. Suppose, without loss of generality, that the result of the shuffle is a sequence of contiguous d-strings $d_1, d_2, \cdots, d_t$, of lengths $2l_1, 2l_2, 2l_3, \cdots, 2l_t$ respectively. The contiguity implies that the corresponding lattice path does not touch the diagonal between the end points. The scan/unshuffle phases will unshuffle the d-strings and invoke merges as follows: an (l_1, l_2)-merge, an (l_2, l_3)-merge, $\cdots$, an (l_{t-1}, l_t)-merge. The notation is illustrated in Figure 7. The height of the lattice path corresponding to a particular merge at a path point corresponding to the p^{th} element in the merged list is the absolute value of the number of left input elements minus the number of right input elements up and including the p^{th} element.

Consider in particular those p_j^{th} elements in the merged input where:

$$p_j = 2\sum_{i=1}^{j} l_i + l_{j+1}$$

for any $0 \le j \le t-1$. That is, consider those portions of the merged list up to and including the rightmost element resulting from any of the $t-1$ recursive merges just listed. The height

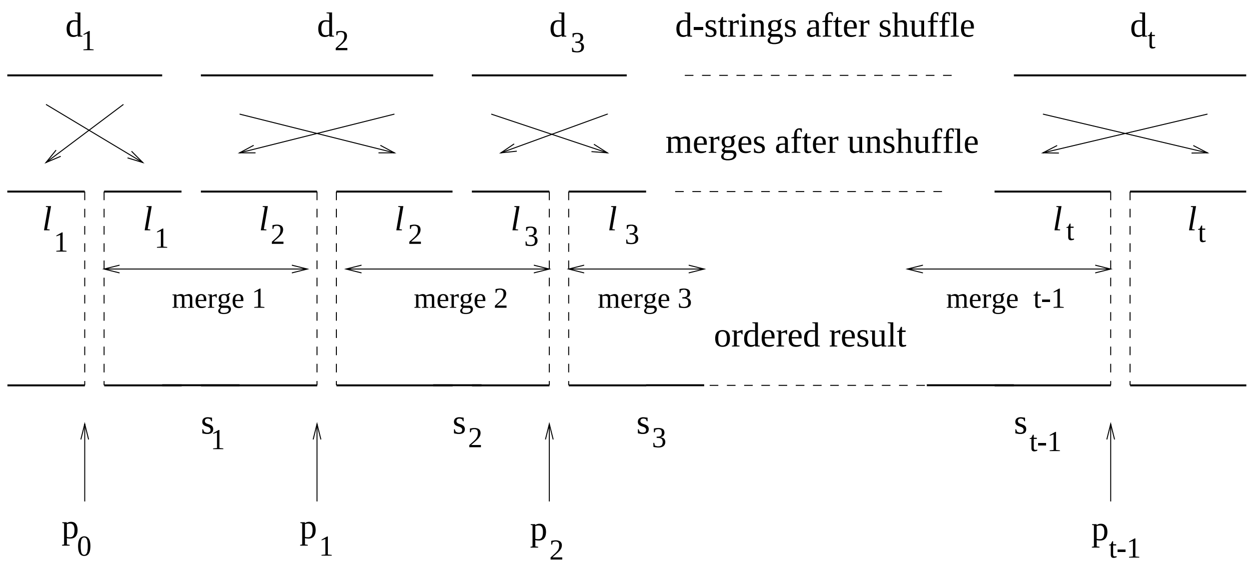

Figure 7: Notation used in Lemma 5.2

of these points is l_{j+1} because an equal number of elements from the left and right inputs is contributed by all d-strings except d_{j+1}. If d_j is the longest d-string in the sequence, then there is a point of height l_j.

The *m-length* of a generalised d-string has been defined to be the number of elements in that d-string derived from the shorter input list. The height of a point in an (m, p) lattice path has been defined to be the vertical distance of that point from the diagonal. To expand the argument to cover the generalised case, it is sufficient to note that the two components of a generalised d-string are no longer of equal length. If the part belonging to the m-list is of length l_i, the part belonging to the p-list is of length $l_i p/m$. But, since the slope of the diagonal is now m/p instead of one, the summation $\sum_{i=1}^{j}(1 + p/m)l_i$, still yields points on the diagonal. Hence the height of the corresponding p_j points is still l_j.

Inserting the upper bound on the height of almost all paths given in Lemma 5.1 produces the desired result. □

We are going to prove Theorem 5.1 below by induction and it will be necessary to apply the inductive hypothesis to the sub-instances of the problem input to the recursive calls. In analysing average case behaviour it is customary to assume that all initial instances are equally likely. This does not necessarily imply that all sub-instances are also equally likely. The next lemma classifies sub-instances in a way such that each type is equally likely.

Let I denote the set of all (m, p)-merge instances, for some m, p, where $m \leq p$. Let R be the multi-set (we need to be able to count the number of appearances of each element) of all merge instances defined by the recursive calls generated by all instances $i \in I$. Let elements in R be characterised by the lengths a and b of the input lists and by j and k, the positions in the original input lists of the first elements in the a-list (the one with a elements) and in the b-list, (the one with b elements) respectively. Thus any element in R can be characterised as an (a, b, j, k)-merge instance.

Lemma 5.3 *If any particular (a, b, j, k)-merge instance exists in R then all (a, b, j, k)-merge instances exist in R and each particular instance occurs the same number of times.*

Proof

For any particular set of values a, b, j, k, let $I' \subset I$ be the set of (m, p)-merge instances that generate a recursive (a, b, j, k)-merge instance. Now consider subsets of I' such that all instances in the subset yield identical merged outcomes, except for the outcome of the (a, b, j, k)-merge instance. We observe that all possible (a, b, j, k)-merge instances must exist exactly once in each of these subsets of I', else some (m, p)-merge instance is absent from I. Consequently, if an (a, b, j, k)-merge instances exists in R, then all possible instances exist and each appears the same number of times. $\square$

Theorem 5.1 *The average time complexity of the generalised Shufflemerge over all (m, p)-instances is $O((m + p) \log \log m)$.*

Proof

We interpret the O notation as: $T_{av}(m, p) \leq b(m + p) \log \log m$ for some constant b and all sufficiently large m, and proceed by induction on m. Suppose the statement is true for all instances of length $< m$.

Consider the set I of all (m, p)-instances, for some m, p, $m \leq p$. Let N denote the number of distinct instances, so that $N = \binom{m+p}{m}$. Let R be the set of all recursive calls generated by all instances $i \in I$. The time T_0 taken at the top level to shuffle/scan/unshuffle is linear in the number of elements, i.e. $T_0 \leq cN(m + p)$, for some constant c.

Let us consider the set R to be partitioned into sub-sets, R_2 through R_{m-1}, one per possible value of m_i, the length of the shorter list input to a recursive call. We remember that if $m = 1$ no recursive call is generated and that m_i must be less than m.

Consider the application of the algorithm to all the instances in R. We note that each R_i can be divided into subsets of instances with identical values for the length of the longer list. These subsets can be further divided into sub-subsets with identical j and k values, as defined in the preamble to Lemma 5.3. By Lemma 5.3 we are entitled to apply the inductive hypothesis across each of these sub-subsets of R_i. Let T_R be the time required to complete computation on all instances in R. Let m_r and p_r denote the list lengths input to a recursive instance $r \in R$, and let l_r denote $m_r + p_r$. Then, by the inductive hypothesis:

$$T_R \leq \sum_{i=2}^{m-1} \left(b \log \log m_i \sum_{r \in R_i} l_r \right)$$

For any $\epsilon, 0 < \epsilon < 0.5$ we divide the sum into those R_i for which $m_i \leq m^{0.5+\epsilon}$ and the rest. Hence:

$$T_R \leq b \log \log m^{0.5+\epsilon} \sum_{i=2}^{m^{0.5+\epsilon}} \sum_{r \in R_i} l_r + b \log \log m \sum_{i=m^{0.5+\epsilon}}^{m-1} \sum_{r \in R_i} l_r$$

Let k be the number of instances in R that fall within the left summation divided by $|R|$, so that $0 \leq k \leq 1$. We note that the sum of the lengths of the recursive instances generated by a single instance in I is $\leq m + p$. Consequently:

$$\sum_{i=2}^{m-1} \sum_{r \in R_i} l_r \leq (m+p)N$$

we have:

$$T_R \leq kNb(m+p) \log \log m^{0.5+\epsilon} + (1-k)Nb(m+p) \log \log m$$

Consequently, the total time is:

$$T_0 + T_R \leq c(m+p)N + kNb(m+p) \log \log m^{0.5+\epsilon} + (1-k)Nb(m+p) \log \log m$$

We note that for $p > 1$, $\log_p \log_p n^{1/p} = \log_p \log_p n - 1$. So let the logarithms be taken to base $1/(0.5 + \epsilon)$. Then:

$$T_0 + T_R \leq N(m+p)(c + k(b \log \log m - b) + (1-k) b \log \log m)$$

The definition of k implies that k is the probability that an instance in R has m-length $\leq m^{0.5+\epsilon}$. From the proofs of Lemmas 5.1 and 5.2, $k = L(z)$ in Equation 4, Lemma 5.1.

Since, $1-k$ tends to zero faster than $\log\log(m)$ increases, there exist a b and an m_0 such that, for all $m \geq m_0$: $c - kb + (1-k)b \log\log m < 0$. Hence, for these values of b and m, the average total time is

$$T_{av} = (T_0 + T_R)/N \leq b(m+p)\log\log m$$

confirming the inductive hypothesis. $\square$

6 Experimental Results

We performed experiments to obtain some measure of the actual performance of both Shufflemerge and HL-merge. We used that version of Shuffle which uses the one extra bit/element. As noted in Section 7, saving the extra bit is possible, at the expense of doubling the number of moves. We also experimented with using these two algorithms as the basis for sorting by repeated merging. In the latter case, we also ran a standard Quicksort implementation which permits some measure of the cost of requiring stability in an in-situ sort.

We chose to use numbers of element comparisons and element moves as the time complexity measures. We did not record CPU times, since these are data, machine and program dependent. Many other factors, such as sequences of memory references that cover wide spans of memory, can influence execution time. So we claim only that these results are an initial indication of the possible practical value of Shufflemerge.

In all these experiments we used lists of integers, randomly selected by our system's random number generator. Since the generator is designed to cycle through 2^{31} integers without repetition, duplicate elements did not occur. Lists of up to one million elements were processed. For the merging tests, the initial lists were of equal length. Each point in the graphs below records the average of four independent experiments.

The reader should also note that whereas in [HL88] and [HL92] it is element "exchanges" that are counted, we count an exchange as 3 moves. Further, for HL-merge, we only counted work done during the "main" phase of the algorithm, ignoring that done during the "preprocessing". The reason for that was that we used a simpler but slower set up process than that described in [HL88]. The faster, but somewhat intricate, preprocessing described in [HL88] requires time $O(\sqrt{n})$ which is asymptotically negligible. In comparing the two merges one must also note that we only ran the unstable HL-merge. The stable version would have been noticeably slower to execute and more complex to program. We consider estimates, taken from [HL92], of the work done by the stable version of HL-merge at the end of this section.

We provide four figures which summarise our results. Figure 8 compares the number of comparisons used by the two merge algorithms as a function of input size. The y-axis is (number of comparisons / number of elements). Figure 9 compares the number of moves used. The y-axis is (number of moves/ number of elements). Figure 10 compares the number of comparisons used when the merges are used in a standard sort by repeated merging process. The same information obtained from a standard implementation of Quicksort is included. The y-axis is (number of comparisons / $n \log_2 n$). Figure 11 compares the number of moves used when the merges are used in a standard sort by repeated merging process. Quicksort is also included. The y-axis is (number of moves) / ($n \log_2 n$).

Figure 8 and Figure 9 show Shufflemerge using relatively fewer moves for n up to about 2^{14} and somewhat more comparisons over the range of the experiments. The results for HL-merge are consistent with the expected linear time performance, namely 6n moves and 1.5n comparisons. Those constants are derived as follows. The main phase of HL-merge proceeds by first ordering $\sqrt{n}$ blocks, each of size $\sqrt{n}$, on the first element in the block and then merging pairs of blocks via the so called buffer block. During the block sorting process, a selection sort would expect to use about $n/2$ comparisons in total, and about $3\sqrt{n}$ moves per block, i.e., $3n$ moves in total. During block merging each element must be compared and moved by way of an exchange, yielding about n comparisons and $3n$ moves.

The results for Shufflemerge show a slowly increasing "constant" of proportionality. Since each level of recursion can perform a shuffle/unshuffle on all elements, and includes a scan across all elements, between $2n$ and $3n$ moves and $n/2$ comparisons are expected per recursion level.

We used the two merges to achieve sort by repeated merge procedures and compared their performances with that of a standard Quicksort. For the latter we used the method which brings two pointers from the ends of the list in towards each other. The pivot was chosen from the mid-point of the list. We note that Quicksort's behaviour is consistent with $O(n \log n)$ time complexity, whereas the behaviour of Shufflemerge clearly is not. The behaviour of Shufflemerge and the unstable HL-merge are very similar up to the maximum size (a million elements) of the experiments. The increasing number of moves used by HL-merge sort, relative to expected $O(n \log n)$ behaviour, is perhaps explained by the earlier graphs showing that it does better on shorter lists, even though its behaviour is undoubtedly linear asymptotically.

The merge sorts are using up to five times as many moves as Quicksort, but are, perhaps surprisingly, using fewer comparisons, in the range of the experiments. This gives some idea of the cost of obtaining a stable, in-situ sort by using these plausibly practical merges, although, to get the cost of using a stable version of HL-merge, one must consider the extra cost discussed at the end of this section.

In evaluating these results one should remember that they are representative of special

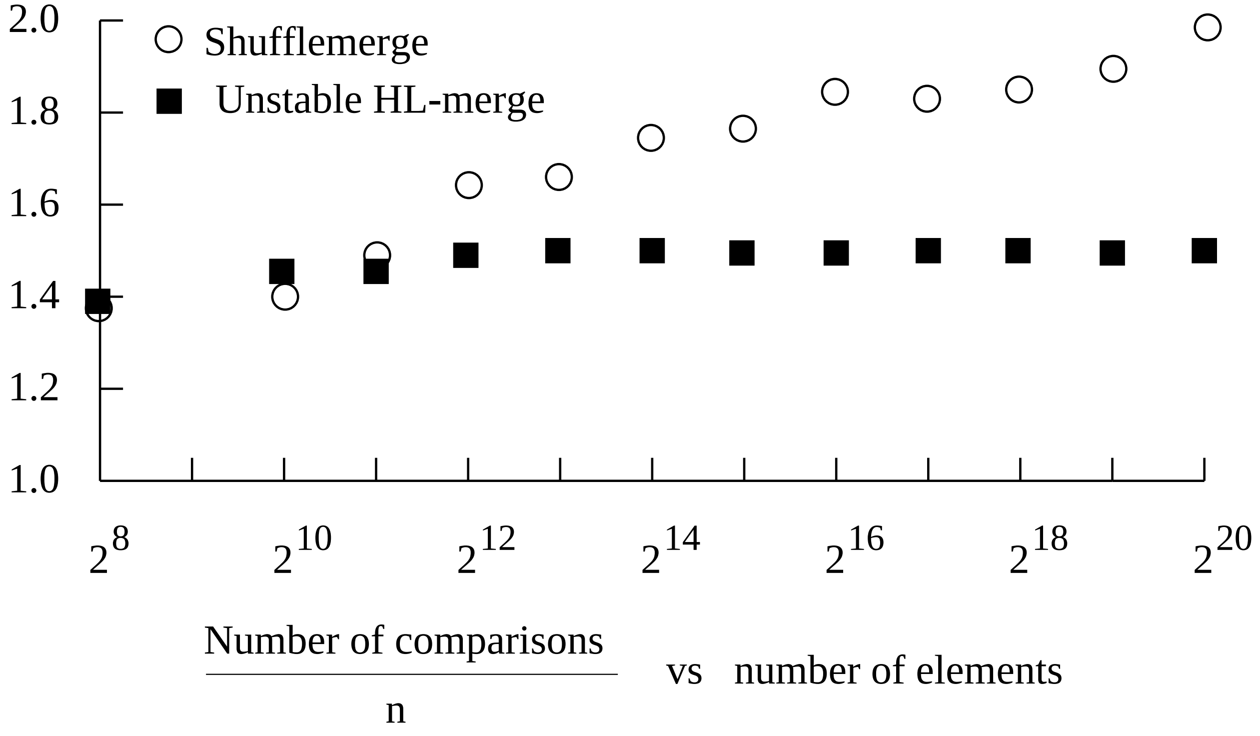

Figure 8: Merging - Numbers of Comparisons

instances of the problem, namely equal length lists (such as one would expect in external sorting applications) of elements approximately uniformly distributed over a range larger than the list size. They should only be taken as approximate indicators of the relative performance one might expect from carefully tuned realisations of the algorithms.

Finally we remind the reader that we experimented with the unstable version of HL-merge, and with the extra bit per element version of Shufflemerge. The simple analysis just given leads us to expect about $1.5n$ comparisons and $6n$ moves for the main phase of the unstable HL-merge. The experimental results accord nicely with expectations. In [HL92] an estimate is given of the number of comparisons and exchanges used by the stable version of HL-merge. Converting each of the authors' exchanges to 3 moves we obtain not more than $2n$ comparisons and about $13.5n$ moves. Using the in-situ version of the Shuffle procedure would again about double the number of moves used by Shufflemerge.

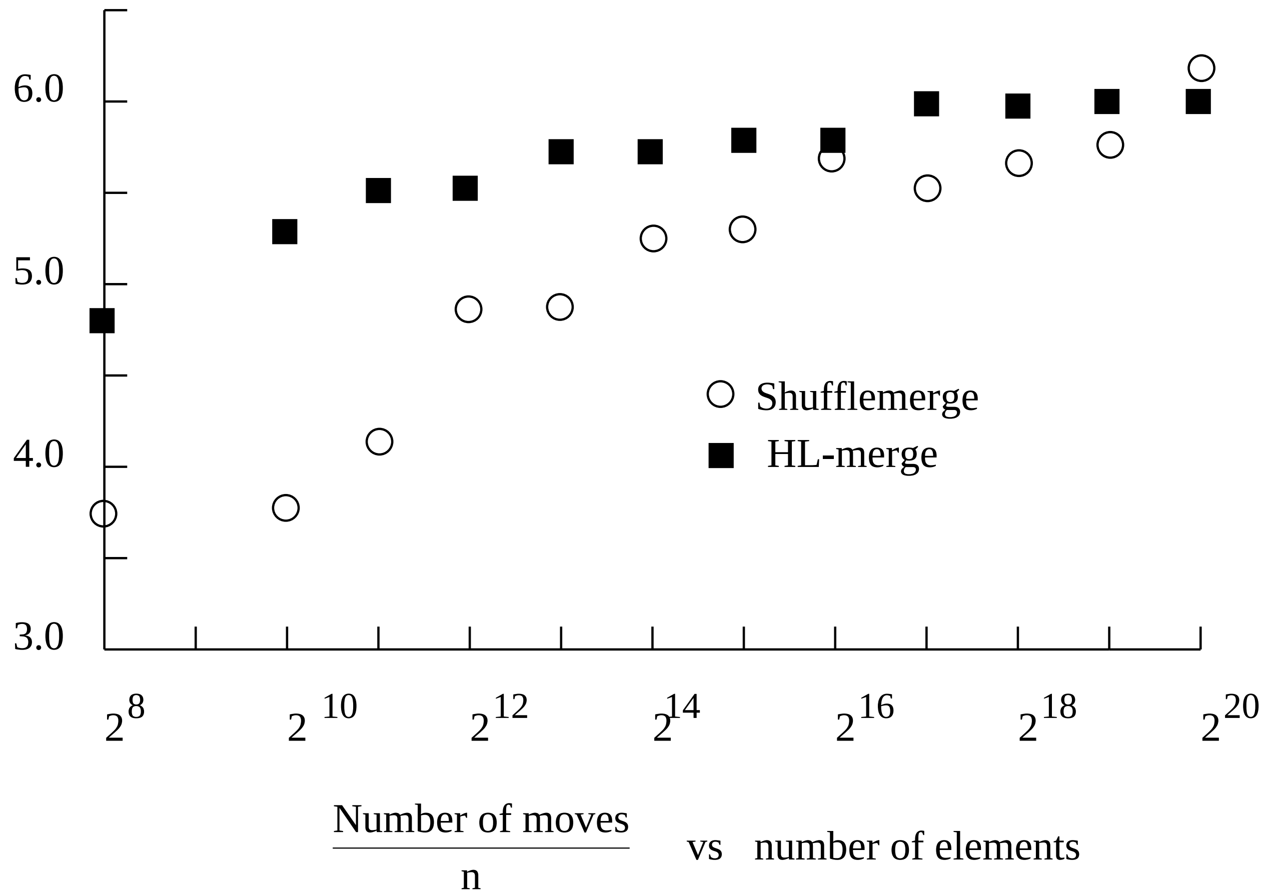

Figure 9: Merging - Numbers of Moves

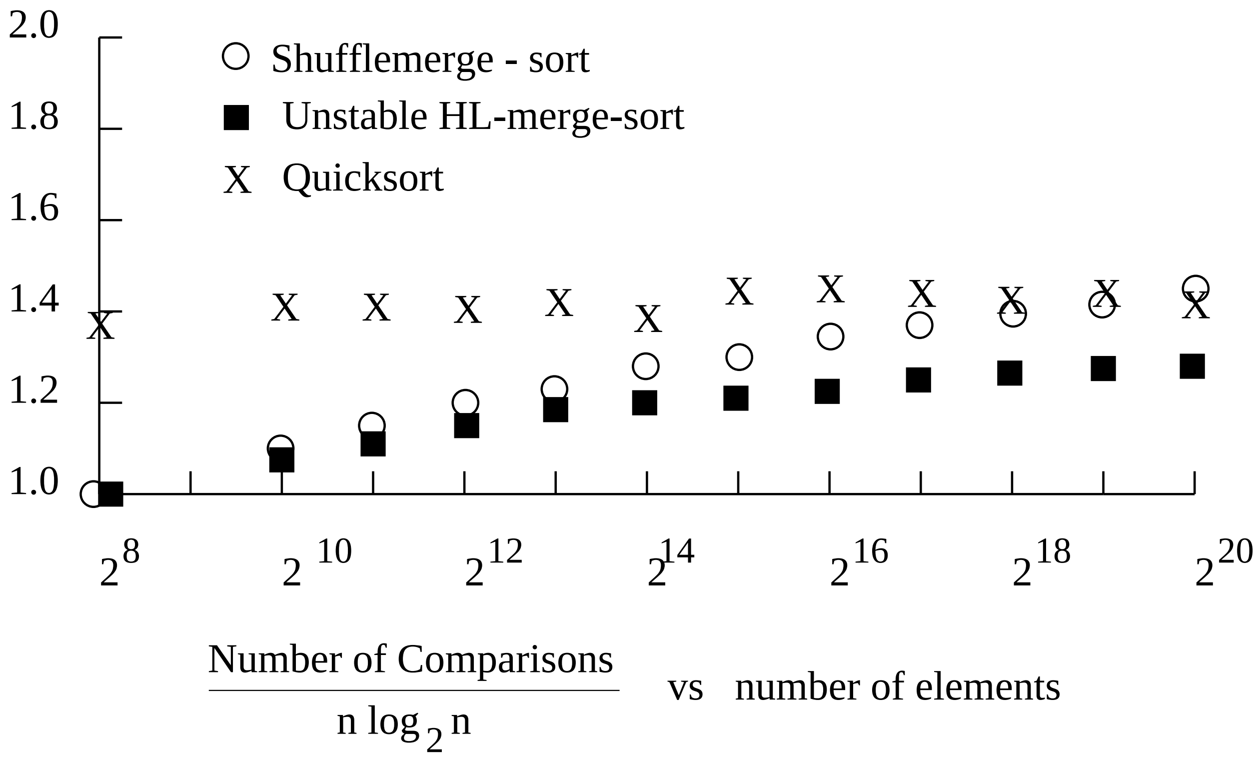

Figure 10: Sorting - Numbers of Comparisons

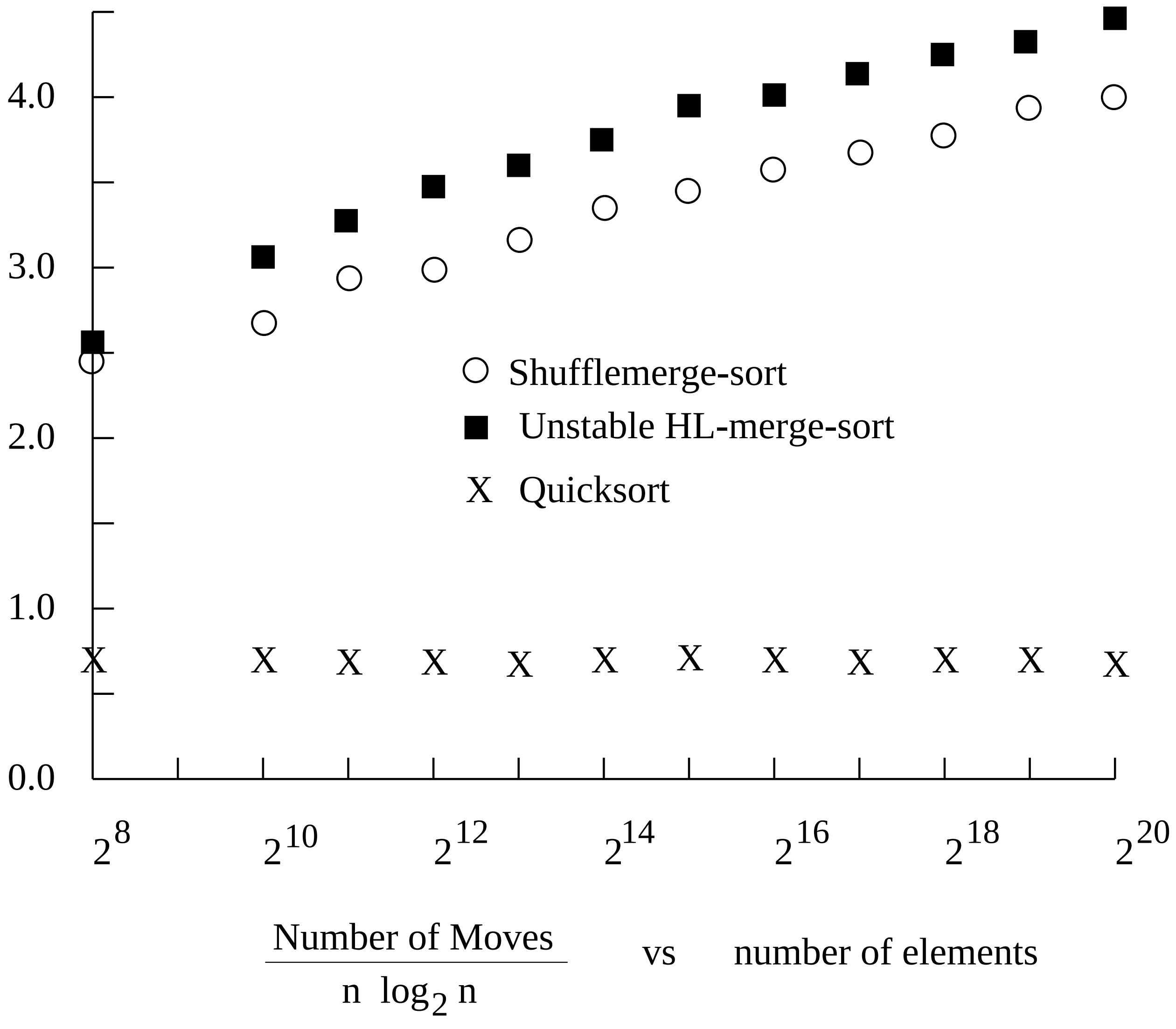

Figure 11: Sorting - Numbers of Moves

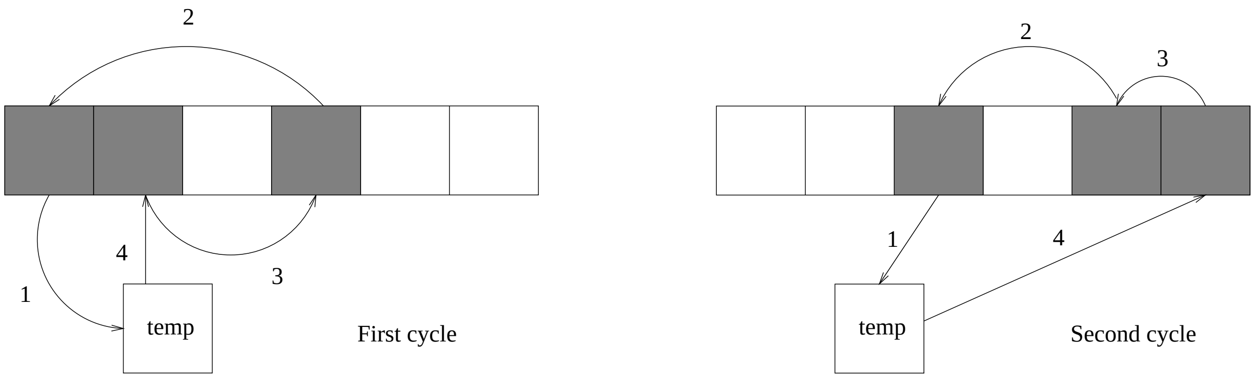

Figure 12: Realising the Perfect Shuffle

7 Shuffling

Certain permutations can be achieved in-situ and in linear time without difficulty. Examples are the reversal or cyclic shift of a list, represented by an array. Permutations are made up of cycles and it is easy to move all the elements of one cycle, using just one extra location, by a “cycle leader” algorithm. A cycle leader algorithm proceeds by repeatedly making a space in the list, computing the index of the element that belongs in that space and moving that element, so creating a new space. See for example [FMP95]. The movement of elements during a six element shuffle is illustrated in Figure 12. In the case of the perfect shuffle it is not obvious how to compute the beginning of a new cycle when a given cycle returns to the starting point.

We show that, even so, the perfect shuffle can be achieved in-situ and in linear time. First we observe that an obvious method achieves the perfect shuffle in linear time, at the expense of one bit per element which is used to mark each element as moved/not moved. When we need to start a new cycle, we can simply scan for an unmoved element. Of course the use of these extra bits contravenes the definition of in-situ.

In Figure 13 we give the code for computing the next index in a cycle sequence. Note that the computation of indices involves only addition, subtraction and multiplication or division by 2. In the case of unshuffle, note that the mod operator is just a conditional subtraction. This minimises the possibility of arithmetic overflow and avoids slower operations.

Next we show that the extra bits can be dispensed with, thus attaining an absolutely minimum use of storage, at the expense of approximately doubling the number of moves. We note that the general problem of shuffling n elements can be reduced to the special case of shuffling $2^k - 2 = p$ elements. This is done by letting k be the largest integer such that $p \leq n$ and exchanging the rightmost $(n-p)/2$ elements of the left list with the leftmost $p/2$

```
{Let first and last be the indices of the first and last
elements in the segment to be shuffled.  Let space be the index
of the current location to be filled.  Let belongs be the index
of the location containing the element that should be  moved to
the location indexed by space.  Then belongs is computed as follows.}

{to shuffle:}
 if odd(space - first + 1)
 then belongs := num div 2 + (space - first) div 2 + first
 else belongs := (space - first + 1) div 2 + first - 1;

{to unshuffle:}
 belongs := 2 * (space - first + 1) mod (num + 1) + first - 1;
```

Figure 13: The Computation of Shuffle/Unshuffle Locations

elements from the right list. If we can solve the leader generation problem for the special case of $n = 2^k - 2$, then the general problem can be solved by recursive application of the procedure to the unshuffled, rightmost $n - p$ elements. The exchange itself is easily done by a cyclic shift using between m and $3m/2$ moves to shift m elements. The sum of these moves over the recursion is between n and $3n/2$.

Finally we note that the shuffle (or unshuffle) is essentially defined by the iterated assignment: $i \leftarrow 2 * i \bmod (n+1)$. If $n + 1 = 2^k - 1$ the right hand side is equivalent to a left cyclic shift of the binary representation of i. Consider the equivalence classes generated by the cyclic shift operation, i.e., each class is the set of binary strings such that, for any pair, one can be obtained from the other by a sequence of cyclic shifts. In other contexts, these equivalence classes are called "necklaces". Each necklace corresponds to one of our permutation cycles. A simple algorithm for generating a representative element from each necklace is given in [FM78] and [FK86]. These elements are of course exactly what we need as cycle leaders.

The algorithm generates a sequence of bit strings of length k, 0^k being the first and 1^k the last in the sequence, where the notation $(\textit{any string})^k$ denotes the repetition of the string k times. Let the bits be numbered 1 through k, from left to right. The successor, $\mathrm{succ}(\alpha)$, of a string $\alpha = a_1 a_2 \cdots a_k$ is formed by applying the following definition.

For $\alpha < 1^k$, $\mathrm{succ}\ (\alpha) = (a_1 a_2 \cdots a_{i-1} 1)^t a_1 \cdots a_j$, *where* i *is the largest integer* $1 \leq i \leq k$ *such that* $a_i = 0$ *and* t, j *are such that* $ti + j = k$ *and* $j < i$.

$\mathrm{succ}(\alpha)$ is a cycle leader iff the i in the definition is a divisor of k. Figure 14 shows the

0 0 0 0 0 0	0 0 0 1 1 0	0 0 1 1 0 1 *	0 1 1 0 1 1 *
0 0 0 0 0 1 *	0 0 0 1 1 1 *	0 0 1 1 1 0	0 1 1 1 0 1
0 0 0 0 1 0	0 0 1 0 0 1 *	0 0 1 1 1 1 *	0 1 1 1 1 0
0 0 0 0 1 1 *	0 0 1 0 1 0	0 1 0 1 0 1 *	0 1 1 1 1 1 *
0 0 0 1 0 0	0 0 1 0 1 1 *	0 1 0 1 1 0	1 1 1 1 1 1
0 0 0 1 0 1 *	0 0 1 1 0 0	0 1 0 1 1 1 *	

Figure 14: Generating Necklaces of Length 6

sequence of bit strings of length 6 generated by this algorithm. The elements marked by an asterisk (1, 3, 5, 7, 9, 11, 13, 15, 21, 23, 27, 31) are the cycle leaders needed to shuffle $2^6 - 2 = 62$ elements.

In [RSW92] it is shown that the algorithm generates these representative elements in constant amortized time. Here "time" counts each bit operation and we count the division test for cycle leadership as one operation. Hence, the total time is proportional to the number of cycles. Both the cycle leader algorithm and the necklace generator use only a constant number of variables and thus together constitute an absolutely minimum space shuffling algorithm.

Finally we observe that the "generalised" shuffle used in Section 5.3 is attainable by elaborating on the basic method just described. We now require that every element i in the left m-list is placed at position $\lceil i(p+m)/m \rceil$ and that the elements from the right p-list are placed in input order in the gaps, i.e., that m individual elements be shuffled with m blocks. This can be done by the elementary method using the extra bits as moved/not moved flags, but at the expense of noticeably more elaborate arithmetic to compute the element positions.

We note that the more sophisticated method does not generalise. The cycles in the new permutation bear no obvious relationship to those in the basic shuffle. We remember that the generalised shuffle was only proposed as an aid in analysing the algorithm. We do not suggest that it be considered as a practical possibility.

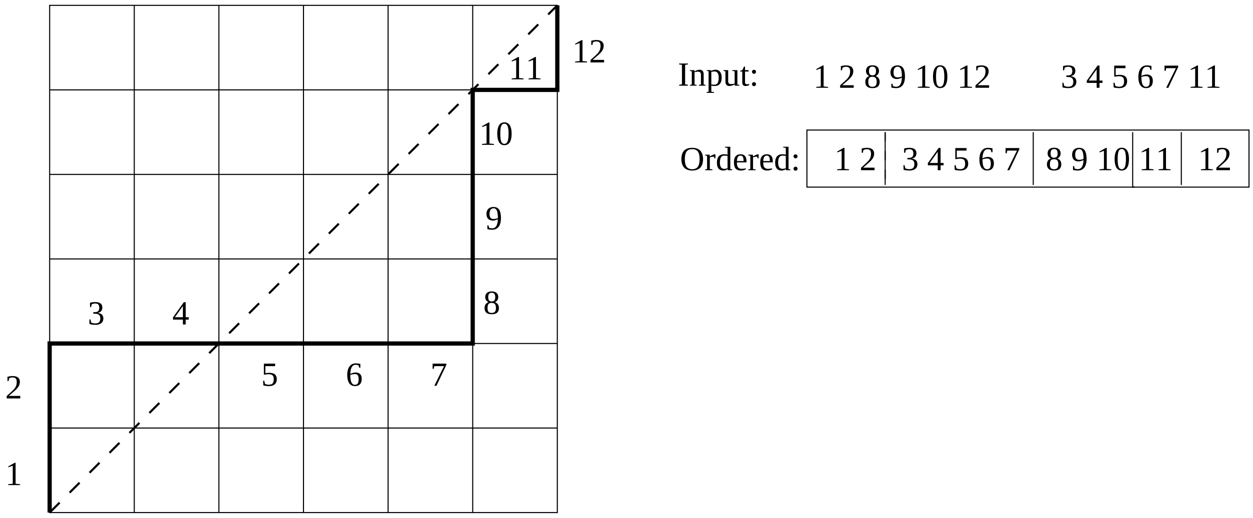

Figure 15: Balanced Input

8 Balanced Inputs

We point out an interesting property of Shufflemerge which indicates that the average case analysis just given may be too pessimistic and also that there may be variations on this algorithmic theme with even better properties.

We have described the correspondence between lattice paths and merge instances. Consider those points in the lattice path at which either the point is not on the diagonal and the path changes direction or the point is on the diagonal. Let us call these two types of point *critical points*. We define a lattice path, in a square lattice, to be *balanced* if points in the sequence of critical points are alternately on and off the diagonal. A merge instance is balanced iff the input lists are of equal length and its corresponding lattice path is balanced. Figure 15 shows an example of a balanced lattice path.

Lemma 8.1 *Shufflemerge merges any balanced input in linear time.*

Proof

Consider the simplest balanced merge instance, i.e., equal length lists such that all elements in one list are less than all elements in the other. One shuffle/scan/unshuffle is sufficient for each of the two possibilities, left list greater than right or *vice versa.* Since shuffling, scanning and unshuffling are all accomplished in linear time, the entire process is linear time.

Now note that any balanced input consists of a sequence of one or more of these simple instances. One shuffle/scan/unshuffle is sufficient to order the entire list. The algorithm, as it stands, may invoke one recursive merge on adjacent segments resulting from the previous unshuffle, but, since these merges are on the simplest balanced instances, the algorithm terminates at this point. □

The upper bound on average case complexity used the $O(\sqrt{n})$ upper bound on the average height of lattice paths corresponding to the inputs. This is valid, but pessimistic. If the input is balanced, the height is irrelevant. Rather, the amount of recursion is determined by the deviation from balance. It is possible that an analysis based on some measure of unbalance could justify a better upper bound on the average time complexity.

The lattice path model also suggests that some kind of "preprocessing", perhaps using minor rearrangements to bring the input closer to balance, might improve performance. We note that this can be done by exchanging and cyclic shifting segments. We have also considered the possibility of "shifting the diagonal" so that the input is closer to balance about the new diagonal.

9 Conclusions

We believe we have described an algorithm which is both interesting, because it tackles a classic problem with a novel technique, attractive in its simplicity, and which might be of practical value. We suggest that further developments based on this new method may be possible.

We have given an average case time analysis which, although it does not apply to the basic algorithm, nevertheless aids our understanding. Obtaining a better bound, or proving that $O(n \log\log n)$ is the best that can be done, is an interesting open problem. Likewise, whether or not a better bound than $O(n \log n)$ exists for the average case for the basic algorithm remains to be established.

We have shown that the perfect shuffle is attainable in linear time and using absolutely minimum storage.

We have pointed out an interesting property of Shufflemerge, namely that so called "balanced" inputs are merged in linear time, without recursion if so desired, using $n/2$ comparisons and about 2n moves. A possibly fruitful variation on our approach could perhaps be to try to balance unbalanced inputs by some kind of preprocessing. It remains possible that this or some other variation on the theme could remove the bad worst case or improve the average case performance.

Finally we note that certain hypercube like parallel processing architectures can achieve a perfect shuffle in one step. We therefore suggest that it may be worth considering the translation of this method onto such a machine.

Acknowledgment

We are indebted to Ian Munro for the suggestion that we try reducing the general in-situ shuffling problem to the special case and to the several referees for very thorough readings and for their many invaluable suggestions and corrections.

References

[FK86] H. Fredricksen and I.J. Kessler. An algorithm for generating necklaces of beads in two colors. *Discrete Mathematics*, 61:181–188, 1986.

[FM78] H. Fredricksen and J. Maiorana. Necklaces of beads in k colors and k-ary de Bruijn sequences. *Discrete Mathematics*, 23:207–210, 1978.

[FMP95] Faith E. Fich, J. Ian Munro, and Patricio V. Poblete. Permuting in place. *SIAM Journal on Computing*, 24(2):266–278, 1995.

[GKP] Viliam Geffert, Jyrki Katajainen, and Tomi Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science*, to appear.

[HL88] B.-C. Huang and M. A. Langston. Practical in-place merging. *Communications of the ACM*, 31(3):348–352, 1988.

[HL92] B.-C. Huang and M. A. Langston. Fast stable merging and sorting in constant extra space. *The Computer Journal*, 35(6):643–650, 1992.

[Hor78] Edward C. Horvath. Stable sorting in asymptotically optimal time and extra space. *Journal of the ACM*, 25(2):177–199, 1978.

[Knu73] Donald Knuth. *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Addison-Wesley, 1973.

[KP94] Jyrki Katajainen and Tomi Pasanen. Sorting multisets stably in minimum space. *Acta Informatica*, 31(4):301–313, 1994.

[KPT96] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. Practical in-place mergesort. *Nordic Journal of Computing*, 3:27–40, 1996.

[Kro69] M. A. Kronrod. Optimal ordering algorithm without operational field. *Dokladi Akademi Nauk SSSR*, 186:203–208, 1969.

[Par77] Luis Trabb Pardo. Stable sorting and merging with optimal space and time bounds. *SIAM Journal on Computing*, 6(2):351–372, 1977.

[Pas99] Tomi Pasanen. *In-place Algorithms for Sorting Problems.* PhD thesis, University of Turku, Finland, 1999.

[RSW92] F. Ruskey, C. Savage, and T. Wang. Generating necklaces. *Journal of Algorithms*, 13:414–430, 1992.

[Smi39] N. Smirnov. On the estimation of the discrepancy between empirical curves of distribution for two independent samples. *Bulletin Mathématique de l'Université de Moscou*, 2, 1939.

[Sym95] A. Symvonis. Optimal stable merging. *The Computer Journal*, 38(8):681–690, 1995.

[TB95] Andrew Tridgell and Richard Brent. A general purpose parallel sorting algorithm. *International Journal of High Speed Computing*, 7(2):285–301, 1995.