MergeShuffle: a very fast, parallel random permutation algorithm

Axel Bacher
Université Paris 13
bacher@lipn.univ-paris13.fr

Olivier Bodini
Université Paris 13
bodini@lipn.univ-paris13.fr

Alexandros Hollender
Université Paris 13

Jérémie Lumbroso
Princeton University
lumbroso@cs.princeton.edu

Abstract

This article introduces an algorithm, MergeShuffle, which is an extremely efficient algorithm to generate random permutations (or to randomly permute an existing array). It is easy to implement, runs in $n \log_2 n + O(1)$ time, is in-place, uses $n \log_2 n + \Theta(n)$ random bits, and can be parallelized across any number of processes, in a shared-memory PRAM model. Finally, our preliminary simulations using OpenMP[1] suggest it is more efficient than the Rao-Sandelius algorithm, one of the fastest existing random permutation algorithms.

We also show how it is possible to further reduce the number of random bits consumed, by introducing a second algorithm BalancedShuffle, a variant of the Rao-Sandelius algorithm which is more conservative in the way it recursively partitions arrays to be shuffled. While this algorithm is of lesser practical interest, we believe it may be of theoretical value.

1 Introduction

Random permutations are a basic combinatorial object, which are useful in their own right for a lot of applications, but also are usually the starting point in the generation of other combinatorial objects, notably through bijections.

The well-known Fisher-Yates shuffle [FY48, Dur64] iterates through a sequence from the end to the beginning (or the other way) and for each location i, it swaps the value at i with the value at a random target location j at or before i. This algorithm requires very few steps—indeed a random integer and a swap at each iteration—and so its efficiency and simplicity have until now stood the test of time.

There have been two trends in trying to improve this algorithm. First, the algotithm initially assumed some source of randomness that allows for discrete uniform variables, but there has been a shift towards measuring randomness better with the random bit model. Second, with the avent of large core clusters and GPUs, there is an interest in making parallel versions of this algorithm.

In: L. Ferrari, M. Vamvakari (eds.): Proceedings of the GASCom 2018 Workshop, Athens, Greece, 18–20 June 2018, published at http://ceur-ws.org

[1] Full code available at: `https://github.com/axel-bacher/mergeshuffle`

Algorithm 1 The classical Fisher-Yates shuffle [FY48] to generate random permutations, as per Durstenfeld [Dur64].

```
1: procedure FisherYatesShuffle(T)
2:     for i = n − 1 to 0 do
3:         j ← random integer from {0, ..., i}
4:         Swap(T, i, j)
5:     end for
6: end procedure
```

The random-bit model.

Much research has gone into simulating probability distributions, with most algorithms designed using infinitely precise *continuous uniform random* variables (see [Dev86, II.3.7]). But because (pseudo-)randomness on computers is typically provided as 32-bit integers—and even bypassing issues of true randomness and bias—this model is questionable. Indeed as these integers have a fixed precision, two questions arise: when are they not precise enough? when are they too precise? These are questions which are usually ignored in typical fixed-precision implementations of the aforementioned algorithms. And it suggests the usefulness of a model where the unit of randomness is not the uniform random variable, but the *random bit*.

This random bit model was first suggested by Von Neumann [Neu51], who humorously objected to the use of fixed-precision pseudo-random uniform variates in conjunction with transcendant functions approximated by truncated series. His remarks and algorithms spurred a fruitful line of theoretical research seeking to determine *which* probabilities can be simulated using only random bits (unbiased or biased? with known or unknown bias?), with which complexity (expected number of bits used?), and which guarantees (finite or infinite algorithms? exponential or heavy-tailed time distribution?). Within the context of this article, we will focus on designing practical algorithms using unbiased random bits.

In 1976, Knuth and Yao [KY76] provided a rigorous theoretical framework, which described generic optimal algorithms able to simulate any distribution. These algorithms were generally not practically usable: their description was made as an infinite tree—infinite not only in the sense that the algorithm terminates with probability 1 (an unavoidable fact for any probability that does not have a finite binary expansion), but also in the sense that the description of the tree is infinite and requires an infinite precision arithmetic to calculate the binary expansion of the probabilities.

In 1997, Han and Hoshi [HH97] provided the *interval algorithm*, which can be seen as both a generalization and implementation of Knuth and Yao's model. Using a random bit stream, this algorithm amounts to simulating a probability p by doing a binary search in the unit interval: splitting the main interval into two equal subintervals and recurse into the subinterval which contains p. This approach naturally extends to splitting the interval in more than two subintervals, not necessarily equal. Unlike Knuth and Yao's model, the interval algorithm is a concrete algorithm which can be readily programmed... as long as you have access to arbitrary precision arithmetic (since the interval can be split to arbitrarily small sizes). This work has recently been extended and generalized by Devroye and Gravel [DG15].

We were introduced to this problematic through the work of Flajolet, Pelletier and Soria [FPS11] on *Buffon machines*, which are a framework of probabilistic algorithms allowing to simulate a wide range of probabilities using only a source of random bits.

One easy optimization of the Fisher-Yates algorithm (which we use in our simulations) is to use a recently discovered optimal way of drawing discrete uniform variables [Lum13].

Prior Work in Parallelization.

There has been a great deal of interest in finding efficient parallel algorithms to randomly generate permutations, in various many contexts of parallelization, some theoretical and some practical [Gus03, Gus08, San98, Hag91, AS96, CB05, CKKL98, And90].

Most recently, Shun et al.[SGBFG15] wrote an enlightening article, in which they looked at the intrinsic parallelism inherent in classical sequential algorithms, and these can be broken down into independent parts which may be executed separately. One of the algorithms they studied is the Fisher-Yates shuffle. They considered the insertion of each element of the algorithm as a separate part, and showed that the dependency graph, which provides the order in which the parts must be executed, is a random binary search tree, and as such, is well

known to have on average a logarithmic height [Dev86]. This allowed them to show that the algorithm could be distributed on $n/\log n$ processors, by using linear auxiliary space to track the dependencies.

Our contribution takes a different direction to provide another algorithm with similar guarantees. Our algorithm is completely in-place, and can be parallelized to give a $\log n$ speedup given enough processors. It runs, in practice, extremely fast even when run sequentially (presumably due to better cache performance compared with Fisher-Yates). We hope, in future work, to paralellize it further to approach the performance of Shun et al.'s algorithm.

Algorithm 2 The MergeShuffle algorithm.

```
1: procedure MergeShuffle(T, k)        ▷ k is the cut-off threshold at which to shuffle with Fisher-Yates.
2:     Divide T into 2^k blocks of roughly the same size
3:     Shuffle each block independently using the Fisher-Yates shuffle
4:     p ← k
5:     repeat
6:         Use the Merge procedure to merge adjacent blocks of size 2^p into new blocks of size 2^(p+1)
7:         p ← p + 1
8:     until T consists of a single block
9: end procedure
```

Splitting Processes.

Relatively recently, Flajolet et al. [FPS11] formulated an elegant random permutation algorithm which uses only random bits, using the *trie* data structure, which models a splitting process: associate to each element of a set $x \in S$ an *infinite* random binary word w_x, and then insert the key-value pairs (w_x, x) into the trie; the ordering provided by the leaves is then a random permutation.

This general concept is elegant, and it is optimized in two ways:

- the binary words thus do not need to be infinite, but only long enough to completely distinguish the elements;
- the binary words do not need to be drawn *a priori*, but may be drawn one bit (at each level of the trie) at a time, until each element is in a leaf of its own.

This algorithm turns out to have been already exposed in some form in the early 60's, independently by Rao [Rao61] and by Sandelius [San62]. Their generalization extends to the case where we split the set into R subsets (and where we would then draw random integers instead of random bits), but in practice the case $R = 2$ is the most efficient. The interest of this algorithm is that it is, as far as we know, the first example of a random permutation algorithm which was written to be parallelized.

2 The MergeShuffle algorithm

The new algorithm which is the central focus of this paper was designed by progressively optimizing a splitting-type idea for generating random permutation which we discovered in Flajolet et al.[FPS11]. The resulting algorithm closely mimics the structure and behavior of the beloved MergeSort algorithm. It gets the same guarantees as this sorting algorithm, in particular with respect to running time and being in-place.

To optimize the execution of this algorithm, we also set a cut-off threshold, a size below which permutations are shuffled using the Fisher-Yates shuffle instead of increasingly smaller recursive calls. This is an optimization similar in spirit to that of MergeSort, in which an auxiliary sorting algorithm is used on small instances.

2.1 In-Place Shuffled Merging

The following algorithm is the linchpin of the MergeShuffle algorithm. It is a procedure that takes two arrays (or rather, two adjacent ranges of an array T), both of which are assumed to be randomly shuffled, and produces a shuffled union.

Importantly, this algorithm uses very few bits. Assuming a two equal-sized sub-arrays of size k each, the algorithm requires $2k+\Theta(\sqrt{k}\log k)$ random bits, and is extremely efficient in time because it requires no auxiliary space.

Lemma 2.1. *Let A and B be two randomly shuffled arrays, respectively of sizes n_1 and n_2. Then the procedure* Merge *produces a randomly shuffled union C of these arrays, of size $n = n_1 + n_2$.*

Algorithm 3 In-place shuffled merging of two random sub-arrays.

```
1: procedure MERGE(T, s, n_1, n_2)
2:     i ← s                          ▷ i, j, n are the beginning, middle, and end position considered in the array.
3:     j ← s + n_1
4:     n ← s + n_1 + n_2
5:     loop
6:         if FLIP() = 0 then         ▷ Flip a coin to determine which sub-array to take an element from.
7:             if i = j then break
8:         else
9:             if j = n then break
10:            SWAP(T, i, j)
11:            j ← j + 1
12:        end if
13:        i ← i + 1
14:    end loop
15:    while i < n do                 ▷ One list is depleted; use Fisher-Yates to finish merging.
16:        Draw a random integer m ∈ {s, ..., i}
17:        SWAP(T, i, m)
18:        i ← i + 1
19:    end while
20: end procedure
```

Proof. Since A and B are assumed to be initially randomly shuffled, we may assume that they consist of identical symbols, say a's and b's respectively. It suffices to prove that, after the execution of the procedure, all words with n_1 occurrences of a and n_2 of b appear with the same probability. We may reinterpret the procedure as first randomly drawing a's and b's by flipping coins, stopping when we would write the $n_1 + 1$-st a or the $n_2 + 1$-st b, and then writing down the missing b's or a's and using the Fisher-Yates shuffle to swap them into random locations.

The correction of the procedure is based on the following fact: after the execution of the first loop (lines 5–14), the elements of T from 0 to $i - 1$ are randomly shuffled. Indeed, these elements consist either of n_1 a's and $n_2 + i - n$ b's (if the array A was depleted first) and $n_1 + i - n$ a's and n_2 b's (if B was depleted first), which can appear in every permutation with equal probability.

The rest of the proof then mimics that of the Fisher-Yates shuffle, based on the loop invariant: after every iteration of the second loop (lines 15–19), the first i elements of T are randomly shuffled. This shows that the whole array is randomly shuffled after the procedure. □

Lemma 2.2. *Assuming that* $|n_1 - n_2| = \mathcal{O}(\sqrt{n})$*, the procedure* MERGE *produces a shuffled array* C *of size* n *using* $n + \Theta(\sqrt{n} \log n)$ *random bits on average.*

In practice, it is always possible to set up the arrays so that $|n_1 - n_2| \leq 1$, which more that satisfies the conditions of this result.

Proof. There are two places where the procedure consumes randomness: in the first loop at line 6 (one bit per element) and in the second loop at line 16 (on average, $\Theta(\log n)$ bits per element since we need to draw a random integer). The number of iterations of the second loop is equal to the number of elements remaining after one of the arrays A or B is depleted. If n_1 and n_2 are not too far apart (at most on the order of $\sqrt{n}$), this number will have an expected value of $\Theta(\sqrt{n})$. This gives the result. □

2.2 Average number of random bits of MergeShuffle

We now give an estimate of the average number of random bits used by our algorithm to sample a random permutation of size n. Let $\text{cost}(k)$ denote the average number of random bits used by a merge operation with an output of size k. For the sake of simplicity, assume that we sample a random permutation of size $n = 2^m$. The average number of random bits used is then

$$\sum_{i=1}^{m} 2^{m-i} \, \text{cost}(2^i).$$

We have seen that $\text{cost}(k) = k + \Theta(\sqrt{k}\log k)$. Thus, the average number of random bits used to sample a random permutation of size $n = 2^m$ is

$$\sum_{i=1}^{m} 2^{m-i}\left(2^i + \Theta\left(\sqrt{2^i}\log(2^i)\right)\right) = m2^m + \Theta\left(2^m \sum_{i=1}^{m} \frac{i}{2^{i/2}}\right)$$

which finally yields

$$m2^m + \Theta(2^m) = n\log_2 n + \Theta(n).$$

3 BalancedShuffle

For theoretical value, we also present a second algorithm, which introduces an optimization which we believe has some worth.

Algorithm 4: The BalancedShuffle algorithm.

Input: an array T
Result: T is randomly shuffled
Main Function `BalancedShuffle(T)`
  n = length(T)
  if $n > 1$ **then**
    `BalancedShuffle(`T$[0{:}\frac{n}{2}]$`)`
    `BalancedShuffle(`T$[\frac{n}{2}{:}n]$`)`
    `BalancedMerge(T)`
  end

Procedure `BalancedMerge(T)`
  n = length(T)
  w = uniformly sampled random balanced word of size n
  $i = 0$
  $j = n/2$
  for $k = 0$ *to* $n - 1$ **do**
    if $w[k] = 1$ **then**
      swap elements at positions i and j in T
      $j = j + 1$
    end
    $i = i + 1$
  end

3.1 Balanced Word

Inspired by Rémy [Rem85]'s now classical and efficient algorithm to generate a random binary tree of exact size from the repeated drawing of random integers, Bacher et al. [BBJ14] produced a more efficient version that uses, on average $2k + \Theta((\log k)^2)$. Binary trees, which are enumerated by the Catalan numbers [Sta15], are in bijection with Dyck words, which are balanced words containing as many 0's as 1's. So Bacher et al.'s random tree generation algorithm can be used to produce a balanced word of size $2k$ using very few extra bits.

Rationale.

The idea behind using a balanced word is that it is more efficient, in average number of bits.

Indeed, splitting processes (repeatedly randomly partition n elements until each is in its own partition), are well known to require $n\log_2 n + O(n)$ bits on average—this is the path length of a random trie [FS09]. The

linear term comes from the fact that when processes are partitioned in two subsets, these subsets are not of equal size (which would be the optimal case), but can be very unbalanced; furthermore, with small probability, it is possible that all elements remain in the same set, especially in the lower levels.

On the other hand, if we are able to partition the elements into two equal-sized subsets, we should be able to circumvent this issue. This idea is useful here, and we believe, would be useful in other contexts as well.

Disadvantages.

The advantage is that using balanced words allows to for a more efficient and sparing use of random bits (and since random bits cost time to generate, this eventually translates to savings in running time). However this requires a linear amount of auxiliary space; for this reason, our BalancedShuffled algorithm is generally slower than the other, in-place algorithms.

3.2 Correctness

Denote by $\mathfrak{S}_n$ the symmetric group containing all permutations of size n. Let $C(n,k)$ be the set of all words of length n on the alphabet $\{0,1\}$ containing k 0's and $n-k$ 1's. We have $|\mathfrak{S}_n| = n!$ and $C(n,k) = \binom{n}{k}$.

The algorithm works at follows: to shuffle a list of length $m+n$, we first shuffle recursively the first m and the last n elements (we sample an element of $\mathfrak{S}_m$ and one of $\mathfrak{S}_n$ independently), then we use Bacher et al.'s algorithm to sample an element of $C(m+n,m)$. We combine those three elements to produce an element of $\mathfrak{S}_{m+n}$. Since this combination is bijective, the correctness follows by induction.

3.3 Average number of random bits

We now give an estimate of the average number of random bits used by our algorithm to sample a random permutation of size n. Let $\mathrm{cost}(2k)$ denote the average number of random bits used to sample a random balanced word of length $2k$ (an element of $C(2k,k)$). For the sake of simplicity, assume that we sample a random permutation of size $n = 2^m$. The average number of random bits used is then

$$\sum_{i=1}^{m} 2^{m-i}\,\mathrm{cost}(2^i).$$

For the algorithm we have $\mathrm{cost}(2k) = 2k + \Theta(\log^2 k)$ [BBJ14]. Thus, the average number of random bits used to sample a random permutation of size $n = 2^m$ is

$$\sum_{i=1}^{m} 2^{m-i}\left(2^i + \Theta\left(\log^2(2^{i-1})\right)\right) = m2^m + \Theta\left(2^m \sum_{i=1}^{m} \frac{i^2}{2^i}\right) = m2^m + \Theta(2^m)$$

which can be rewritten as

$$n\log_2 n + \Theta(n).$$

4 Simulations

The simulations were run on a computing cluster with 40 cores. The algorithms were implemented in C, and their parallel versions were implemented using the OpenMP library, and delegating the distribution of the threading entirely to it.

n	10^5	10^6	10^7	10^8
Fisher-Yates	1 631 434	19 550 941	229 329 728	2 628 248 831
MergeShuffle	1 636 560	19 686 051	231 641 075	2 650 387 993
Rao-Sandelius	1 631 519	19 550 449	229 327 120	2 628 251 036
BalancedShuffle	1 889 034	22 046 574		

Table 1: Average number of random bits used by our implementation of various random permutation algorithms over 100 trials. (The current implementation of BalancedShuffle were in Python rather than C, and are prohibitively slow on larger permutations, but preliminary results show that it converges to an improved number of random bits.)

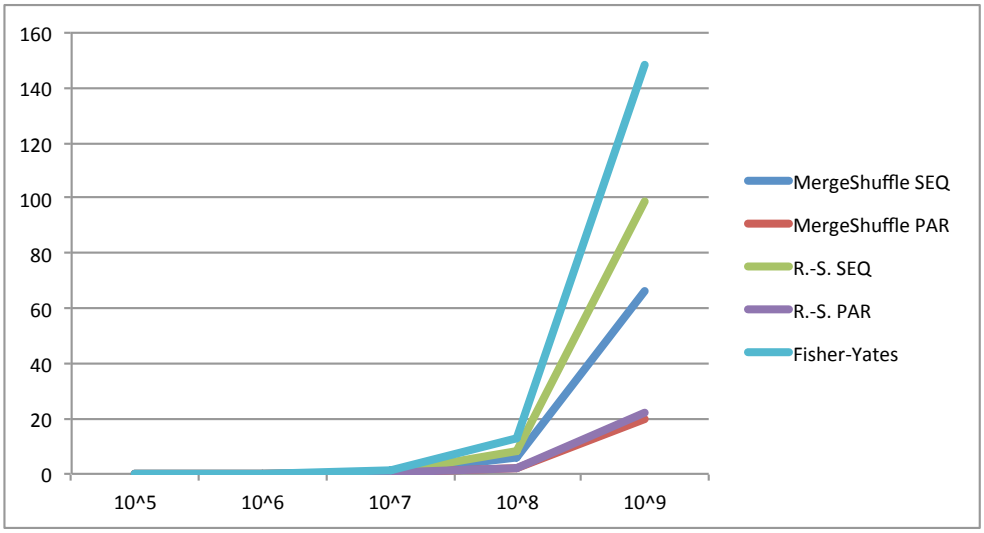

Figure 1: Running times of several random permutation algorithms. Fisher-Yates shuffle, while extremely fast, gets slowed down once permutations are very large. Our parallel MergeShuffle algorithm is consistently faster than all algorithms, although the lead is not yet much compare to the Rao-Sandelius algorithm.

Acknowledgements

This research was partially supported by the ANR MetACOnc project ANR-15-CE40-0014.

References

[AS96] L. Alonso and R. Schott. A parallel algorithm for the generation of a permutation and applications. *Theoretical Computer Science*, 159(1):15–28, 1996.

[And90] R. Anderson. Parallel algorithms for generating random permutations on a shared memory machine. In: *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '90, pp. 95–102, New York, NY, USA, 1990. ACM.

[BBJ14] A. Bacher, O. Bodini and A. Jacquot. Efficient random sampling of binary and unary-binary trees via holonomic equations. At *https://arxiv.org/abs/1401.1140*, 2014.

[Cha06] R. Chandra. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[CB05] G. Cong and D. A Bader. An empirical analysis of parallel random permutation algorithms on SMPs. In: *Proc. 18th ISCA International Conference on Parallel and Distributed Computing Systems* (PDCS 2005), Las Vegas, NV, 2005.

[CKKL98] A. Czumaj, P. Kanarek, M. Kutylowski and K. Lorys. Fast generation of random permutations via networks simulation. *Algorithmica*, 21(1):2–20, 1998.

[DE98] L. Dagum and R. Enon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[Dev86] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986.

[DG15] L. Devroye and C. Gravel. Sampling with arbitrary precision. At *https://arxiv.org/abs/1502.02539*, 2015.

[Dur64] R. Durstenfeld. Algorithm 235: Random permutation. *Communications of the ACM*, 7(7):420, 1964.

[FY48] R. A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research. 3rd Edition*. London : Oliver and Boyd, 1948.

[FPS11] P. Flajolet, M. Pelletier and M. Soria. On Buffon Machines and Numbers. In: D. Randall (Ed.): *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011*, pp. 172–183. SIAM, 2011.

[FS09] P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[Gus03] J. Gustedt. Randomized permutations in a coarse grained parallel environment. In: *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, pp. 248–249, New York, NY, USA, 2003. ACM.

[Gus08] J. Gustedt. Engineering parallel in-place random generation of integer permutations. In: *Proceedings of the 7th International Conference on Experimental Algorithms*, WEA'08, pp. 129–141, Berlin, Heidelberg, 2008. Springer-Verlag.

[Hag91] T. Hagerup. Fast parallel generation of random permutations. In: *Proceedings of the 18th International Colloquium on Automata, Languages and Programming*, pp. 405–416, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[HH97] T. S. Han and M. Hoshi. Interval Algorithm for Random Number Generation. *IEEE Transactions on Information Theory*, 43(2):599–611, 1997.

[KY76] D. E. Knuth and A. C. Yao. The complexity of nonuniform random number generation. *Algorithms and Complexity: New Directions and Recent Results*, pp. 357–428, 1976.

[Lum13] J. Lumbroso. Optimal discrete uniform generation from coin flips, and applications. At *https://arxiv.org/abs/1304.1916*, 2013.

[Rao61] C. R. Rao. Generation of random permutation of given number of elements using random sampling numbers. *Sankhya A*, 23:305–307, 1961.

[Rem85] J.-L. Remy. Un procédé itératif de dénombrement d'arbres binaires et son application à leur génération aléatoire. *RAIRO - Theoretical Informatics and Applications*, 19(2):179–195, 1985.

[San62] M. Sandelius. A simple randomization procedure. *Journal of the Royal Statistical Society. Series B (Methodological)*, 24(2):472–481, 1962.

[San98] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Information Processing Letters*, 67(6):305–309, 1998.

[SGBFG15] J. Shun, Y. Gu, G. E. Blelloch, J. T. Fineman and P. B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In: *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '15, pp. 431–448. SIAM, 2015.

[Sta15] R. P. Stanley. *Catalan Numbers*. Cambridge University Press, 2015.

[Neu51] J. von Neumann. Various techniques used in connection with random digits. *Applied Mathematics Series*, 12:36–38, 1951.

A Code Listing for the MergeSort algorithm

We reproduce here the most part of our algorithm, with some OpenMP [DE98, Cha06] hints. The full code can be obtained at `https://github.com/axel-bacher/mergeshuffle`

A.1 The merge procedure

```
// merge together two lists of size m and n-m
void merge(unsigned int *t, unsigned int m, unsigned int n) {
    unsigned int *u = t;
    unsigned int *v = t + m;
    unsigned int *w = t + n;

    // randomly take elements of the first and second list according to flips
    while(1) {
        if(random_bit()) {
            if(v == w) break;
            swap(u, v ++);
        } else
            if(u == v) break;
        u ++;
    }

    // now one list is exhausted, use Fisher-Yates to finish merging
    while(u < w) {
        unsigned int i = random_int(u - t + 1);
        swap(t + i, u ++);
    }
}
```

A.2 The MergeSort algorithm itself

```
extern unsigned long cutoff;

void shuffle(unsigned int *t, unsigned int n) {
    // select q = 2^c such that n/q <= cutoff
    unsigned int c = 0;
    while((n >> c) > cutoff) c ++;
    unsigned int q = 1 << c;

    unsigned long nn = n;

    // divide the input in q chunks, use Fisher-Yates to shuffle them
    #pragma omp parallel for
    for(unsigned int i = 0; i < q; i ++) {
        unsigned long j = nn * i >> c;
        unsigned long k = nn * (i+1) >> c;
        fisher_yates(t + j, k - j);
    }

    for(unsigned int p = 1; p < q; p += p) {
        // merge together the chunks in pairs
        #pragma omp parallel for
        for(unsigned int i = 0; i < q; i += 2*p) {
            unsigned long j = nn * i >> c;
            unsigned long k = nn * (i + p) >> c;
```

```
            unsigned long l = nn * (i + 2*p) >> c;
            merge(t + j, k - j, l - j);
        }
    }
}
```