Cardchoose: simple, fast sampling without replacement

Paul Crowley

Anthropic PBC

9 November 2026

Abstract

We present Cardchoose, a new algorithm for choosing k distinct natural numbers below n such that every set of k is equally likely. It draws a multiset of k values below $n-k+1$ with a Pólya urn, then sorts it and converts it to a set by the stars and bars bijection. It is a few lines of integer arithmetic and a call to a standard sort, needs no memory beyond the output array, and takes $\mathcal{O}(k \log k)$ time. We benchmark C++ and Rust implementations of it and eleven other algorithms over a wide range of n and k, for both sorted and random order output. No algorithm is fastest for every n, k, language and output order, but Cardchoose is fastest or close to it in many cases, and is by far the simplest algorithm to perform acceptably across the whole range tested.

1 Introduction

Sampling without replacement means choosing k distinct numbers from $\{0, 1, \ldots, n-1\}$ such that every k-element subset is equally likely to be returned. Many programming languages include facilities for this in their standard libraries, for example Python's `random.sample()` or Rust's `rand::seq::index::sample`; Ting [Tin21b] lists the algorithms that several of them use.

We present Cardchoose, a new algorithm for this problem, which we first proposed in October 2018. It chooses a multiset with a Pólya urn and turns it into a set with a sort, so that it is a few lines of integer arithmetic and a call to a standard sort. It needs no memory beyond the array in which the result is written, and takes $\mathcal{O}(k \log k)$ time. Section 2 explains it and proves it fair, and Section 2.6 places it relative to earlier work.

Many other algorithms solve this problem, with different strengths. Some are fast only for certain values of k, n, for example where k is small or where $\frac{n}{k}$ is small. Some update a hash-based data structure such as a set or dictionary for each value generated, which adds memory and a significant constant factor to the runtime. Some need extra memory proportional to n, or floating-point

arithmetic. We describe eleven such algorithms in Section 3, and in Section 4 we compare the performance of C++ and Rust implementations of each. Along the way, in Section 3.4, we propose what appears to be a novel optimisation of the quadratic variant of Floyd's F2 for generating a fair random order.

No single algorithm is close to the fastest for all n, k, and libraries such as Python's and Rust's implement several algorithms and choose between them based on n and k. In Section 5, we find the small sets of algorithms that between them minimise the worst time per number output. Three algorithms need no extra memory and always perform reasonably, and of these Cardchoose is the fastest in many cases, and by far the simplest.

2 Cardchoose

Cardchoose first chooses a multiset uniformly at random, then sorts it and converts it to a set. A Python implementation follows. It returns k distinct integers below n in sorted order, with every set of k equally likely.

```
def sorted_choose(n, k):
    "k distinct integers 0 <= x < n, sorted"
    t = n - k + 1
    d = [None] * k
    for i in range(k):
        r = random.randrange(t + i)
        if r < t:
            d[i] = r
        else:
            d[i] = d[r - t]
    d.sort()
    for i in range(k):
        d[i] += i
    return d
```

The rest of this section explains why it works, proves it fair, and places it relative to earlier work.

2.1 Notation

Let $\mathbb{N}_{<n} \stackrel{\text{def}}{=} \{i \in \mathbb{N} : i < n\}$, the set of natural numbers less than n where $\mathbb{N} = \{0, 1, 2, \ldots\}$. Let $\binom{S}{k} \stackrel{\text{def}}{=} \{s \subseteq S : |s| = k\}$, the set of all k-element subsets of S; note that $\left|\binom{S}{k}\right| = \binom{|S|}{k}$.

A *multiset* is an extension of a set in which elements can appear more than once; we use brackets to delimit the elements of a multiset, so $[0, 1, 1]$ is the same multiset as $[1, 0, 1]$ but distinct from $[0, 0, 1]$. Define $\left(\!\binom{U}{k}\!\right)$ to be the set of multisets of cardinality k whose elements are drawn from U, and $\left(\!\binom{n}{k}\!\right) \stackrel{\text{def}}{=} \left|\left(\!\binom{\mathbb{N}_{<n}}{k}\!\right)\right|$ the number of distinct k-cardinality multisets over an n-element universe.

2.2 Subsets as stars and bars

Write the numbers $0, \ldots, n-1$ in a row, marking each chosen number as a star and each unchosen one as a bar. For $n = 7$, $k = 3$ and the subset $\{1, 2, 5\}$:

0	1	2	3	4	5	6
|	⋆	⋆	|	|	⋆	|

A row of k stars and $n-k$ bars is exactly a k-element subset of $\mathbb{N}_{<n}$: the stars' positions are the chosen numbers. Following Feller [Fel68, Chapter II §5], think of the spaces between bars, including the space at each end, as *cells*, and number them from 0. Instead of its position, we describe each star by its *cell number*, the number of bars to its left:

|	⋆	⋆	|	|	⋆	|
	1	1			3	

A cell number can be anything from 0 (before all the bars) to $n-k$ (after all of them), so there are $t = n - k + 1$ cells, and stars can share a cell. The cell numbers of the stars therefore form a multiset in $\left(\!\binom{\mathbb{N}_{<t}}{k}\!\right)$. This is the well-known stars and bars bijection between $\binom{\mathbb{N}_{<n}}{k}$ and $\left(\!\binom{\mathbb{N}_{<t}}{k}\!\right)$. Going one way, read off each star's cell number. Going back, the multiset says how many stars go in each cell, and that rebuilds the row. The two directions undo each other, so each subset corresponds to exactly one multiset and each multiset to exactly one subset. The bijection is usually used for counting, to show that $\left(\!\binom{t}{k}\!\right) = \binom{t+k-1}{k}$; here we use it to turn a random multiset into a random subset.

In code, the way back is: sort the multiset, then add each star's index to its cell number, because a star's position is its bars to the left plus its stars to the left:

sorted cell numbers	1	1	3
+ index	0	1	2
positions	1	2	5

That is the last three lines of the algorithm. Because the correspondence is a bijection, choosing a multiset uniformly at random and converting it gives a subset chosen uniformly at random: each subset is reached from exactly one multiset. So what remains is to choose an element of $\left(\!\binom{\mathbb{N}_{<t}}{k}\!\right)$ fairly.

2.3 Choosing a multiset fairly

Putting each of k stars in a cell chosen independently from $\mathbb{N}_{<t}$ does not give a uniform multiset. With $t = 2$, $k = 2$, the multiset $[0, 1]$ comes up half the time, because it can arrive as $0, 1$ or as $1, 0$, while $[0, 0]$ and $[1, 1]$ get a quarter each. Independent draws favour variety; we need to favour repeats, by exactly the right amount.

A classical probability model favours repeats in just this way: the Pólya urn [Fel68, Chapter V §2, pp. 119–120]. Feller's urn holds balls of two colours; we use its generalisation to t colours [LP17, §2.4], which starts with t balls, one of each colour in $\mathbb{N}_{<t}$. Each step draws a ball uniformly at random, records its colour, and puts it back along with a new ball of the same colour. A colour drawn c times so far has $c+1$ balls in the urn, so it is $c+1$ times as likely to be drawn next as a colour not yet drawn. Cardchoose runs this urn as an algorithm.

The loop in the listing runs exactly this urn, numbering the balls in the order they enter it: balls $0, \dots, t-1$ are the originals, ball $b < t$ having colour b, and ball $t+i$ is the one added at step i. At step i the urn holds $t+i$ balls, so `randrange(t + i)` draws one. If $r < t$, it is an original, of colour r. Otherwise it is the ball added at step $r-t$, whose colour is the one drawn at that step, `d[r - t]`. So each step takes one random number, one comparison and one write, with no search, and the urn needs no storage beyond `d` itself.

The same process can be pictured on the row of Section 2.2: starting from the $t-1$ bars alone, insert each new star in a uniformly random place in the row so far, between two neighbouring symbols or at either end; a cell holding c stars has $c+1$ such places, one for each ball of its colour [Cro26].

2.4 Proof

Consider a run of the urn that draws the colours $x_0, \dots, x_{k-1}$, and let c_j be the number of times colour j appears. At step i the urn holds $t+i$ balls, and one more ball of colour x_i than the number of earlier steps that drew x_i. Over the whole run, the numerators for colour j are therefore $1, 2, \dots, c_j$, and the probability of this sequence is

$$\frac{\prod_j c_j!}{t(t+1)\cdots(t+k-1)} = \frac{(t-1)!\prod_j c_j!}{(t+k-1)!},$$

which depends only on the counts, not on their order, as Feller observes for two colours [Fel68, p. 120]. The multiset with these counts is drawn by $k!/\prod_j c_j!$ distinct sequences, so its probability is

$$\frac{k!}{\prod_j c_j!} \cdot \frac{(t-1)!\prod_j c_j!}{(t+k-1)!} = \frac{k!\,(t-1)!}{(t+k-1)!} = 1 \Big/ \binom{t+k-1}{k},$$

the same for every multiset in $\left(\!\binom{\mathbb{N}_{<t}}{k}\!\right)$. This is a standard lemma about the urn [LP17, Lemma 2.7]. By Section 2.2, converting the multiset then gives every subset in $\binom{\mathbb{N}_{<n}}{k}$ with the same probability.

2.5 Implementation

The multiset choosing procedure in Python, taking integers n, k and returning a list of k integers in $\mathbb{N}_{<n}$, with repeats allowed, uniformly among all such multisets:

```
def choose_multiset(n, k):
    m = []
    for i in range(k):
        r = random.randrange(n + i)
        if r < n:
            m.append(r)
        else:
            m.append(m[r - n])
    return m
```

Applying it with universe $\mathbb{N}_{<n-k+1}$ and converting the result as in Section 2.2 returns a sorted list of k distinct integers in $\mathbb{N}_{<n}$, uniformly among all such subsets:

```
def choose_binom(n, k):
    m = choose_multiset(n - k + 1, k)
    m.sort()
    for i in range(k):
        m[i] += i
    return m
```

The listing at the start of this section is these two procedures merged into one.

2.6 Prior art

This subsection mentions several of the other algorithms, which Section 3 describes.

Physicists have names for the three distributions in this story [Fel68, Chapter II §5]. Distinguishable particles placed independently in cells, the biased way of picking a multiset in Section 2.3, is Maxwell–Boltzmann statistics. Indistinguishable particles with every occupancy pattern equally likely, a uniform multiset, is Bose–Einstein. At most one particle per cell with every pattern equally likely, a uniform subset, is Fermi–Dirac. Stars and bars is the standard way to count the Bose–Einstein configurations, by matching them with Fermi–Dirac ones. So the one-line summary of Cardchoose is: generate a Bose–Einstein configuration with a Pólya urn, then map it to Fermi–Dirac by stars and bars.

The mathematics of both halves is classical. The Pólya urn is a textbook object of study, and that with one ball of each colour its counts are uniform, which is to say Bose–Einstein, is a standard lemma about it [LP17, Lemma 2.7]; Niven and Grendar [NG09] map all three statistics onto the urn, with Maxwell–Boltzmann as replacement without a copy, Bose–Einstein as replacement with a copy, and Fermi–Dirac as no replacement. But the urn is presented as a model, not as a way to generate anything. The nearest thing to running an urn as a sampler in earnest is in Bayesian statistics, where the Blackwell–MacQueen urn [BM73] draws an exchangeable sequence from a Dirichlet process, which is a different purpose. The combinatorial generation literature does generate random

multisets, but through stars and bars in the other direction: Nijenhuis and Wilf's RANCOM [NW78, Chapter 6] chooses a random $(k-1)$-subset of $\{1,\ldots,n+k-1\}$ and differences it. Nor do surveys of sampling without replacement describe anything like it: Devroye's [Dev86, Chapter XII] has no urn and no stars and bars, and Ting [Tin21b], reworking the classical algorithms into simpler forms, gives a sparse Fisher–Yates sampler, essentially HSEL, and a sorted sampler that draws each skip from a beta-binomial distribution, which needs floating point. What is new in Cardchoose is the urn used as a generator for uniform multisets, and its composition with stars and bars into a sampler for sampling without replacement. Needing no set, map or extra memory is not new in itself: Nijenhuis and Wilf's RANKSB [NW78, Chapter 4] also uses nothing but the output array, and so do Vitter's Algorithm D and, more recently, Shekelyan and Cormode's Hidden Shuffle [SC21], though they need floating-point arithmetic where Cardchoose and RANKSB use only exact integer draws.

Among earlier algorithms, Gupta and Bhattacharjee's GSEL [GB84] also takes $\mathcal{O}(k \log k)$ time and can return sorted output, but works quite differently. Like SELECT, it draws its ith value uniformly from the $n-i+1$ values not yet chosen, as a rank r among them; that value is $r+u$, where u is the number of values already chosen below it, and GSEL finds u in a balanced binary tree of the chosen values that records the size of each subtree, as Ernvall and Nevalainen [EN82] had suggested in passing to bound HSEL's worst case. Rouzankin and Voytishek [RV99] do the same with a sorted array, in $\mathcal{O}(k^2)$ time, and with an unbalanced binary search tree, in $\mathcal{O}(k \log k)$ expected time; they also split the range in half recursively, drawing how many values fall in each half from a hypergeometric distribution, which with stored tables of its probabilities takes $\mathcal{O}(k)$ expected time, and is the division Sanders et al. [San+18] use to sample in parallel. Cardchoose's last step also adds to each value a count of chosen values below it, but the resemblance ends there. GSEL's ranks count the values not chosen at the time of each draw, a set that shrinks with every draw, so it must turn each rank into a value as soon as it is drawn, and that needs the tree. Cardchoose's cell numbers count the values that are never chosen, so they never change, and the counting can wait for a single sort at the end, in the output array. We have not benchmarked GSEL: besides needing memory that Cardchoose does not, it follows pointers through the tree for every value drawn, where Cardchoose makes one sort over a flat array, and we see no way for it to be faster.

Sorted output without a sort has a classic answer for real numbers: Bentley and Saxe [BS80] generate k sorted uniform reals in $\mathcal{O}(k)$ time, drawing the largest as $U^{1/k}$ and repeating for the rest below it, and credit Friedman [Fri71] with an essentially equivalent earlier program. This cannot stand in for the urn: sorted independent draws, even rounded to integers, follow the Maxwell–Boltzmann distribution over multisets, not the Bose–Einstein one. Its closest analogue for distinct integers is Vitter's Algorithm D, whose continuous approximation is the same step.

3 Benchmarked algorithms

We compare Cardchoose with eleven other algorithms. Every implementation fills an array of k elements. Some applications require that the returned elements be in sorted order, while others need the order of the elements to be a fair random draw. Both Python's `random.sample()` and Rust's `rand::seq::index::sample` return in a fair random order.

An algorithm that returns one order can be made to return the other with a sort or a Fisher–Yates shuffle. But each algorithm returns one order naturally, and the other costs that extra step, so which algorithm is fastest can depend on which order the caller wants. In our benchmarks we time both a sorted and a random variant of every algorithm.

For each algorithm we give how it works, the order it returns, its expected time, and any memory it needs beyond the output array. All but two draw only random integers below a bound; Vitter's Algorithm D and Hidden Shuffle also need floating-point arithmetic.

Algorithm	Time	Order	Extra memory	Floats
Quadratic rejection sampling	k^2	Random		
Set-based rejection sampling	k	Random	Set	
Floyd's F2	k	None	Set	
Quadratic F2	k^2	Random		
Iterative choosing	n	Sorted		
Reservoir sampling	n	Random		
SELECT	n	Random	n-element array	
HSEL	k	Random	Dictionary	
RANKSB	k	Sorted		
Vitter's Algorithm D	k	Sorted		Yes
Hidden Shuffle	k	Sorted		Yes
Cardchoose	$k \log k$	Sorted		

The times are expected times for $k \leq n/2$; the algorithms that reject draws slow down as k approaches n.

3.1 Quadratic rejection sampling

```
def random_choose(n, k):
    d = []
    while len(d) < k:
        x = random.randrange(n)
        if x not in d:
            d.append(x)
    return d
```

Draw a value below n, check whether it is already in the output by scanning the output so far, and append it if not; repeat until the output has k values. The output is in random order and no extra memory is needed. Each check takes

$\mathcal{O}(k)$ time, so the whole takes $\mathcal{O}(k^2)$ expected time for $k \leq n/2$, and more as k approaches n and more draws are rejected, as for set-based rejection sampling below.

3.2 Set-based rejection sampling

```
def random_choose(n, k):
    d = []
    r = set()
    while len(d) < k:
        x = random.randrange(n)
        if x not in r:
            d.append(x)
            r.add(x)
    return d
```

As above, but check each draw against a set of the values chosen so far, with an efficient membership test [GH77]. Teuhola and Nevalainen [TN82] and Ahrens and Dieter [AD85] keep the set in a hash table, and Devroye [Dev86, Chapter XII §2] compares the data structures that can hold it. The output is in random order, and the set takes $\mathcal{O}(k)$ memory. With a hash table, this takes $\mathcal{O}(k)$ expected time for $k \leq n/2$, where each draw is rejected with probability at most a half; Jakobsson [Jak85] shows that the expected number of draws is then less than $2k \ln 2 \approx 1.39k$. As k approaches n, rejections mount: the expected number of draws is $n(H_n - H_{n-k})$, where H_m is the mth harmonic number, so at $k = n$ it takes $\mathcal{O}(n \log n)$ time.

3.3 Floyd's F2

```
def set_choose(n, k):
    d = set()
    for m in range(n - k, n):
        j = random.randrange(m+1)
        if j in d:
            d.add(m)
        else:
            d.add(j)
    return d
```

Build a set of the values chosen so far, one value per step, with no rejection [BF87]. Before the step for m, running from $n - k$ to $n - 1$, the set is a uniformly random subset of $\mathbb{N}_{<m}$. Draw j uniformly from $\mathbb{N}_{<m+1}$, and add j if it is not in the set, or m if it is; m is never in the set before its own step, so the set grows by one, and it is then a uniformly random subset of $\mathbb{N}_{<m+1}$. This makes exactly k draws and takes $\mathcal{O}(k)$ expected time with a hash set, which takes $\mathcal{O}(k)$ memory, and the result has no order. For random order output, our C++ and Rust implementations do not shuffle afterwards. Instead, as in Ting's random-order

version of Floyd's algorithm [Tin21a], when the chosen value is already present, they move a randomly chosen earlier output to the end and put the new value in its place. This keeps the order uniformly random, and costs an extra random number only on collisions.

3.4 Quadratic F2

```
def random_choose(n, k):
    d = []
    for i in range(k):
        m = n - k + i
        r = random.randrange(m + 1)
        for j in range(i):
            if d[j] == r:
                d[j] = m
                break
        d.append(r)
    return d
```

Floyd's F2 with the output array as its set, scanned to check each draw, so that it needs no extra memory and takes $\mathcal{O}(k^2)$ time. When the draw r is already present, it replaces that entry with m and appends r, which keeps the output in uniformly random order without a shuffle or any extra draw: the scan has already found where r is, which Floyd's F2 with a set does not know. Rust's `rand` crate used Floyd's F2 with a scan in the same way [Ran18], but kept the order random either by inserting m before the matching entry, which shifts every entry after it, or by shuffling the whole output afterwards, which costs k more draws. It has used our version since 2023; the pull request that made that change [Cro23] is the first publication of this way of keeping the order random that we know of. For sorted output, our implementations append m instead, since before the sort only the set matters, and then sort.

3.5 Iterative choosing (Algorithm S)

```
def sorted_choose(n, k):
    d = []
    for i in range(n):
        if random.randrange(n - i) < k - len(d):
            d.append(i)
    return d
```

Go through the values of $\mathbb{N}_{<n}$ in order, and choose each with the probability that it is in the sample given how many values are still to choose and how many remain to consider [FMR62, Method 1, p.391]; this is Knuth's Algorithm S [Knu97]. The output is sorted, no extra memory is needed, and it makes one draw for every value it considers, so it takes $\mathcal{O}(n)$ time.

3.6 Reservoir sampling (Algorithm R)

```
def random_choose(n, k):
    d = [None] * k
    for i in range(k):
        r = random.randrange(i + 1)
        d[i] = d[r]
        d[r] = i
    for i in range(k, n):
        r = random.randrange(i + 1)
        if r < k:
            d[r] = i
    return d
```

Fill the output with the elements of $\mathbb{N}_{<k}$ in random order. Then for each i from k to $n-1$, draw r from $\mathbb{N}_{<i+1}$, and if $r < k$, replace the rth output with i; this is Knuth's Algorithm R [Knu97]. The output is in random order, no extra memory is needed, and it makes one draw for every value, so it takes $\mathcal{O}(n)$ time. Vitter's Algorithm Z [Vit85] instead draws how many elements to skip, which takes $\mathcal{O}(k(1+\log(n/k)))$ expected time but, like Algorithm D below, needs floating point; we time only Algorithm R.

3.7 SELECT

```
def random_choose(n, k):
    d = [None] * k
    e = list(range(n))
    for i in range(k):
        j = random.randrange(i, n)
        d[i] = e[j]
        e[j] = e[i]
    return d
```

Fill an array with the elements of $\mathbb{N}_{<n}$, and run the first k steps of a Fisher–Yates shuffle on it: at step i, swap a random element from position i onwards into position i [GH77]. The first k elements are then the sample, in random order. It makes only k draws, but the array takes $\mathcal{O}(n)$ memory and $\mathcal{O}(n)$ time to fill.

3.8 HSEL

```
def random_choose(n, k):
    d = [None] * k
    e = {}
    for i in range(k):
        j = random.randrange(i, n)
        d[i] = e.get(j, j)
        e[j] = e.get(i, i)
    return d
```

SELECT with a "virtual" array: a hash table stores only the elements that have changed, so it needs no n-element array [EN82]. The output is in random order, and it takes $\mathcal{O}(k)$ expected time and $\mathcal{O}(k)$ memory for the hash table. Teuhola and Nevalainen [TN82] give a version that needs $2k$ storage locations rather than about $5k$, which draws all k random numbers first and then works out which value each one stands for, from the last back. Our C++ and Rust implementations use a faster variant, which keeps the first k entries of the virtual array in the output array itself, so only swaps that reach beyond them touch the hash table. This variant was described by rici [ric15]; we thank Mathias Weller for bringing it to our attention.

3.9 RANKSB

Divide $\mathbb{N}_{<n}$ into k bins of nearly equal size. First decide how many of the chosen values fall in each bin, by simulating set-based rejection sampling while keeping only a count for each bin: a draw is rejected with the probability that it would duplicate a value already accepted in its bin, which depends only on that bin's count. Then choose that many values within each bin; bins hold about one value each, so a quadratic method suffices. The output is sorted, the expected time is $\mathcal{O}(k)$ for $k \leq n/2$, and the program in [NW78, Chapter 4] needs no memory beyond the output array. Our C++ and Rust implementations follow that program, except that they fill each bin as soon as they know where its values go, rather than first marking out every bin's slots and then scanning for the marks; our Python implementation keeps the counts in a separate array, which is easier to follow.

3.10 Vitter's Algorithm D

Like iterative choosing, produce the output in sorted order, but instead of deciding about each value in turn, draw the number of values to skip before the next chosen one directly. The skip has a known discrete distribution, which Algorithm D samples by rejection from a continuous approximation, the smallest of k uniform reals drawn in one step as in [BS80], in $\mathcal{O}(1)$ expected time per output, at the cost of a few floating-point exponentiations, and with no extra memory. Once $n \leq 13k$ it falls back on Vitter's Algorithm A, which draws each skip in time proportional to its length. Vitter introduced Algorithm D in [Vit84]; our Python, C++ and Rust implementations follow the improved code in the appendix of [Vit87], which is too long to list here. Ahrens and Dieter [AD85] give other sequential methods that take $\mathcal{O}(k)$ time, and Ting [Tin21b] one that draws each skip from a beta-binomial distribution; like Algorithm D, these draw their skips in floating point, and we take Algorithm D and Hidden Shuffle to represent them.

3.11 Hidden Shuffle

Shekelyan and Cormode's Hidden Shuffle [SC21] considers a Fisher–Yates shuffle of $\mathbb{N}_{<n}$ that stops after k steps, so that the sample is whatever ends up in the first k positions; call these the low positions, and the rest the high positions. Each step reaches a high position independently, with known probability, and which high position it reaches is uniform and independent of everything else. So first count the steps that reach a high position, then draw that many high positions with replacement in sorted order as in [BS80]. The first draw of a high position adds its value to the sample; a repeat finds a low value swapped there earlier, so the sample gains a low value instead, much as a collision in Floyd's F2 adds a different value. Given their number, the low values are a uniformly random subset of the low positions, chosen in order by Vitter's Algorithm A. The output is sorted, the time is $\mathcal{O}(k)$ in the worst case, and no extra memory is needed; like Algorithm D, it relies on floating-point arithmetic, with an exponentiation for each high draw. The code printed in the paper can divide by zero, and its last step has an off-by-one error that biases the output; our Python, C++ and Rust implementations apply the corrections from the errata in [She]. We thank metasj for bringing this algorithm to our attention.

3.12 Cardchoose

Described in Section 2: choose a multiset of k cell numbers below $n - k + 1$ with a Pólya urn, then sort it and add each one's index to turn it into a set. The output is sorted, and the only memory needed is the output array.

4 Benchmarks

The accompanying repository (`https://github.com/ciphergoth/sansreplace`) includes C++ and Rust implementations of these algorithms, which were benchmarked for a variety of values of n and k, for both sorted output and random output, on an AWS r8a.xlarge, whose four vCPUs are each a whole physical core, so nothing shares the core a benchmark runs on:

- CPU: AMD EPYC 9R45, four cores
- Cache: 48kiB L1 data and 1MiB L2 per core, 16MiB L3 shared
- Memory: 32GiB
- OS: NixOS 25.11, Linux 6.12
- C++ compiler: clang 21.1 with libc++, `-O3 -march=x86-64-v3`
- Rust compiler: rustc 1.98, release profile, `-C target-cpu=x86-64-v3`

We tested values of n and k that are Fibonacci numbers, with n up to 701,408,733 (the 44th Fibonacci number) and k up to 102,334,155 (the 40th).

4.1 Method

A Python harness times each algorithm in a process of its own, calling the C++ and Rust implementations through a C interface. At each point, it times passes of more and more calls, until a pass takes up what is left of 0.2 seconds, and records the mean time per call over that last pass; a point where one call takes longer than that is timed with a single call. The loop over calls runs in C++ or Rust, and every call writes to the same output array, allocated outside the timing, aligned to a 64-byte cache line and zeroed. The overhead of a call, measured by timing a function that does nothing, is about 1.1 ns, and is not subtracted. Every process seeds its random number generator with the same fixed value, and the C++ and Rust generators start in the same state, so running the same code again makes the same draws. Timing the same points on four separate boots of the machine, which may come back on a different host each time, every algorithm agreed to within about 1%. We describe what we found and changed to make these timings trustworthy, such as aligning arrays that were otherwise slower to fill depending on where earlier allocations left them, in `measurement.md` in the repository.

4.2 The two languages

The C++ and Rust implementations are the same algorithms, written the same way, and apart from random number generation each uses only its own language's standard library, so the results show the performance available from standard components. Some of those components differ a great deal.

Hash tables. The C++ uses libc++'s `std::unordered_set` and `std::unordered_map`, which are node-based: each entry is allocated separately and reached through a linked list. Rust's `HashSet` and `HashMap` are flat open-addressing tables based on SwissTable. The hash-based algorithms take about 0.3 times as long in Rust as in C++.

Sorting. libc++'s `std::sort` is an introsort, and Rust's `sort_unstable` is ipnsort. Sorting the same arrays of random numbers, `std::sort` takes 1.15 to 1.84 times as long (Table 1), and Cardchoose takes about 0.5 times as long in Rust as in C++. RANKSB, Algorithm D and Hidden Shuffle need neither a hash table nor a sort, and take about the same time in both languages for sorted output.

Random numbers. Both use the Xoshiro256++ generator, and both turn its output into a number below a bound with a multiplication. But the C++ uses its own code for this, which takes two 32-bit numbers from each 64-bit output, while the `rand` crate's `random_range` uses a whole output for each. The shuffle that turns the sorted output of RANKSB, Algorithm D and Hidden Shuffle into random order is the same loop in both languages, but adds 1.1 to 1.5 times as much time per number in Rust, at the median in each decade of k; so for random

Length	C++ `std::sort`	Rust `sort_unstable`
144	4.41	3.39
10,946	7.06	6.12
121,393	18.1	9.82
1,346,269	20.2	11.5

Table 1: Time to sort an array of random numbers, in ns per number

output, Algorithm D and Hidden Shuffle take 4–8% longer in Rust at the median. The `rand` crate's own `SliceRandom::shuffle` is slower than our loop.

Searching an array. The quadratic algorithms search the output so far on every draw. The Rust uses `slice::contains`, and the C++ a loop of its own that the compiler vectorises, since libc++'s `std::ranges::contains` calls `wmemchr`, which is slower for the short arrays where these algorithms are worth using.

4.3 Results

In Figure 1 to Figure 6 we graph time/k in ns against k for two different values of n for random order output, and a single value of n for sorted output, in each language. Animated graphs showing many values of n are in `results.md` in the repository, and the complete benchmark data in `results/latest`.

No algorithm is the fastest for all n, k. Quadratic rejection and quadratic F2 are the fastest for small k, and SELECT, reservoir sampling and iterative choosing for small n/k, but each of these is very slow elsewhere. The other algorithms run in $\mathcal{O}(k)$ or $\mathcal{O}(k \log k)$ time, and differ by smaller factors.

Section 5 compares the algorithms by their worst time per number output in each decade of k. The rest of this section describes regions of n and k, by the range of one algorithm's time over another's.

For sorted output in C++, Hidden Shuffle is the fastest algorithm wherever $n \geq 16k$ and $k \geq 1000$; there, Cardchoose takes up to 2.4 times as long. In Rust, Cardchoose and Hidden Shuffle divide the range by k: wherever $k < 1000$ and $n \geq 2k$, Cardchoose is the fastest algorithm or within 24% of it, and Hidden Shuffle takes up to 3.7 times as long, while for larger k they trade places: wherever $k \geq 10^5$ and $n \geq 16k$, each is within 27% of the fastest, and Hidden Shuffle is the fastest wherever $k \geq 10^7$ and $n \geq 16k$. Hidden Shuffle takes about the same time in both languages, so the difference is in Cardchoose, most likely in its sort. Hidden Shuffle is also faster than RANKSB wherever $n \geq 16k$ and $k \geq 1000$ in both languages, with RANKSB taking 1.1 to 2.7 times as long. As k approaches n, RANKSB slows down, taking up to 530 times as long as the fastest algorithm at $k = n$ in C++, and 870 times in Rust; [NW78] pairs it with iterative choosing (which Nijenhuis and Wilf refer to as RKS2) for $k \geq n/2$. Vitter's Algorithm D, with its fallback to Algorithm A, has the smallest worst

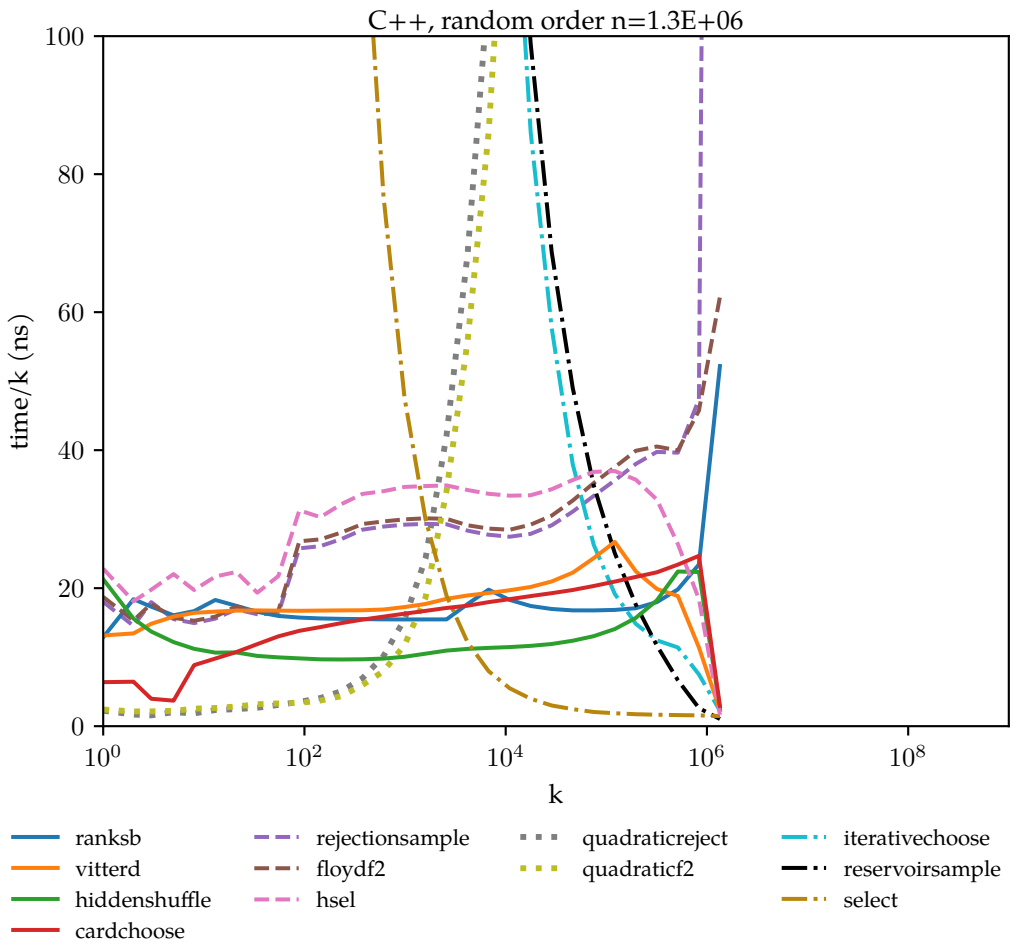


Figure 1: C++, random order, $n = 1.3 \times 10^6$

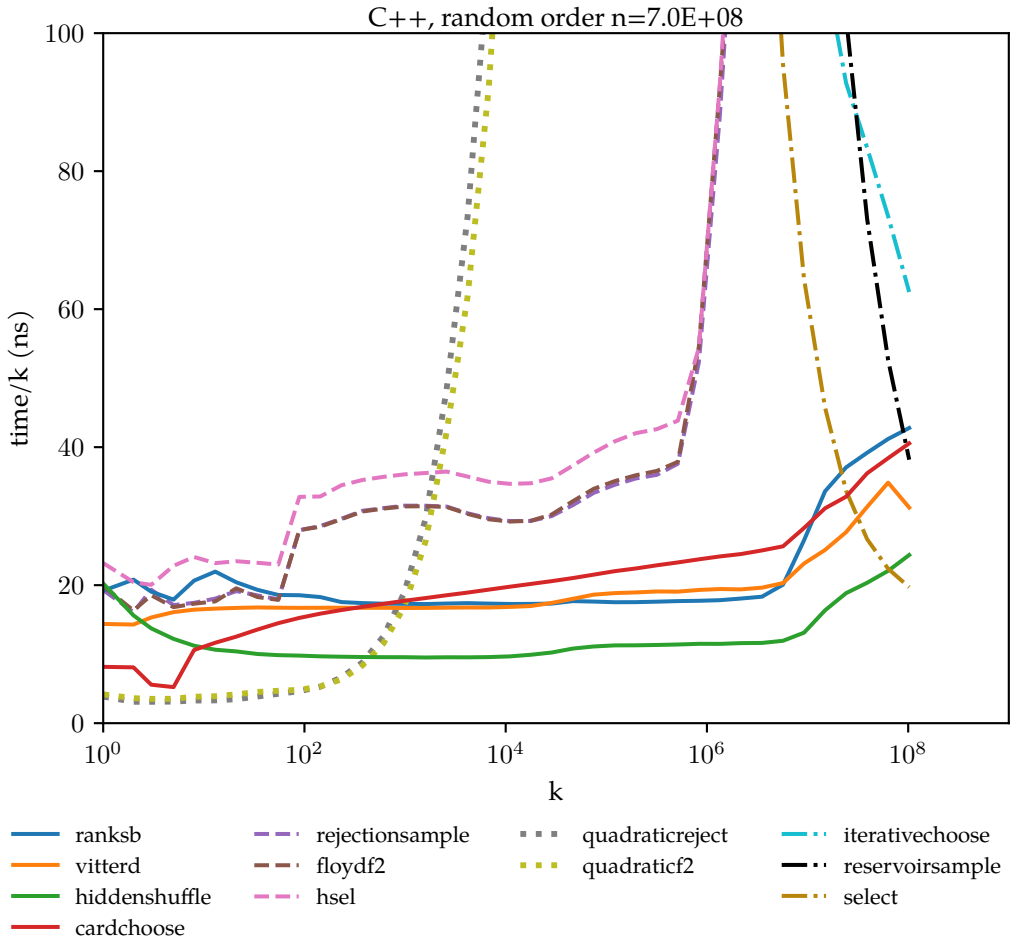


Figure 2: C++, random order, $n = 7.0 \times 10^8$

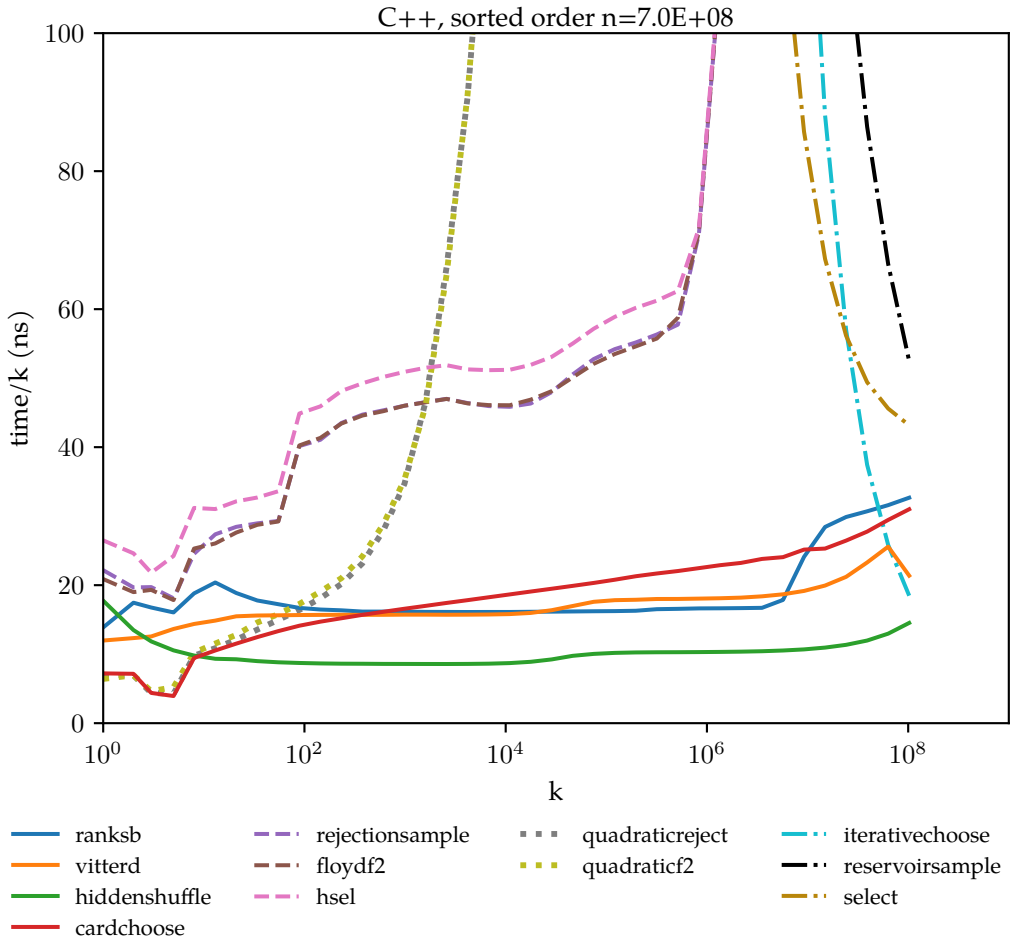


Figure 3: C++, sorted order, $n = 7.0 \times 10^8$

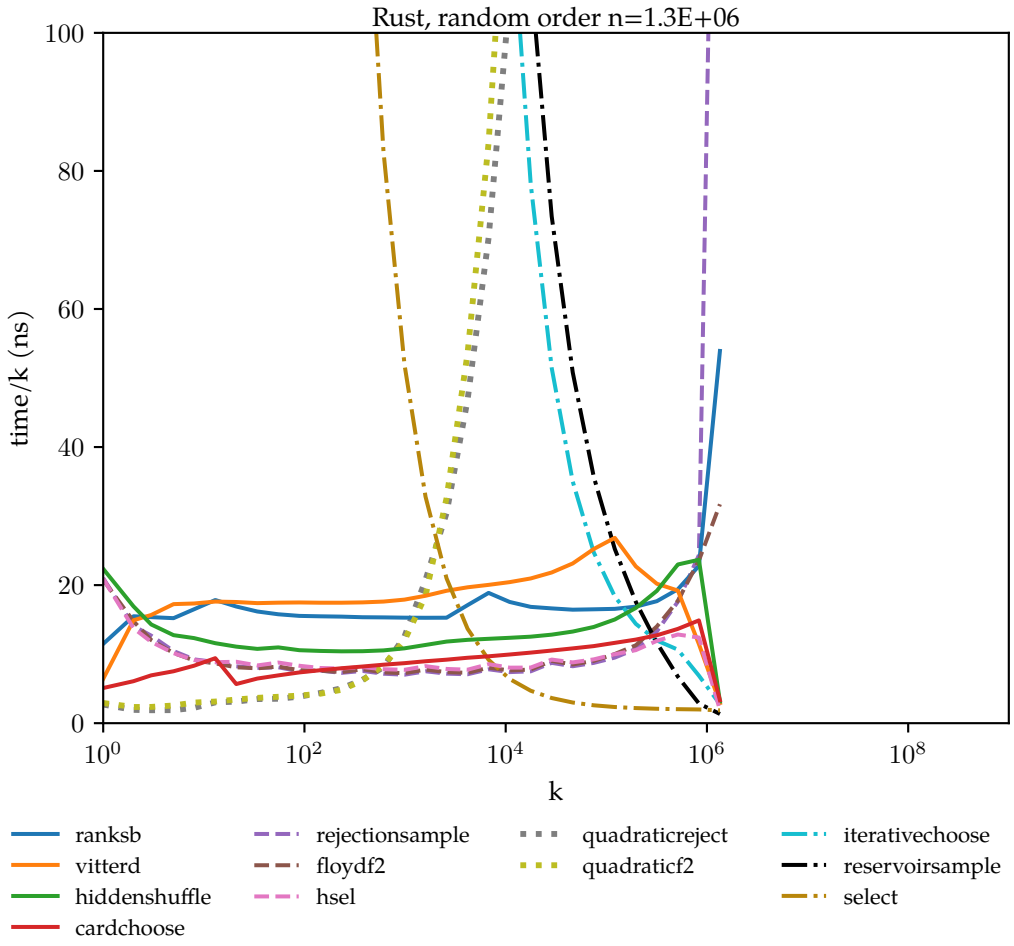


Figure 4: Rust, random order, $n = 1.3 \times 10^6$

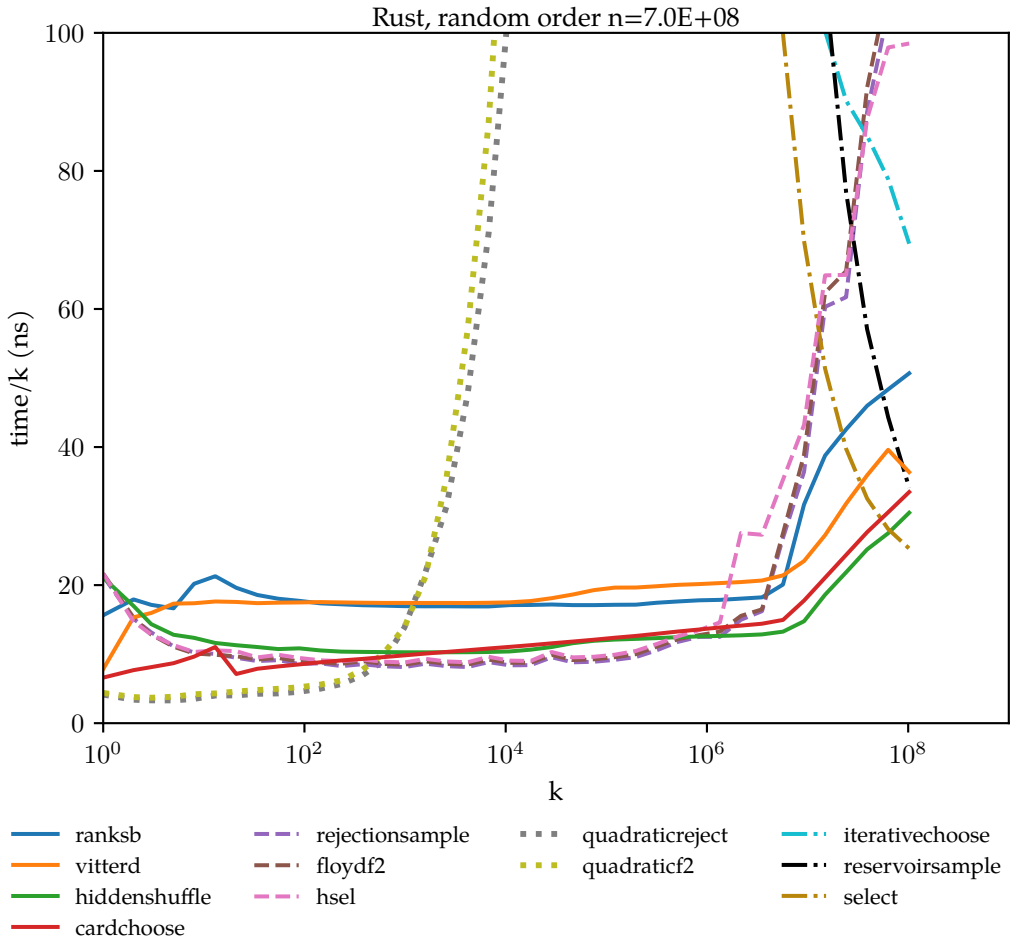


Figure 5: Rust, random order, $n = 7.0 \times 10^8$

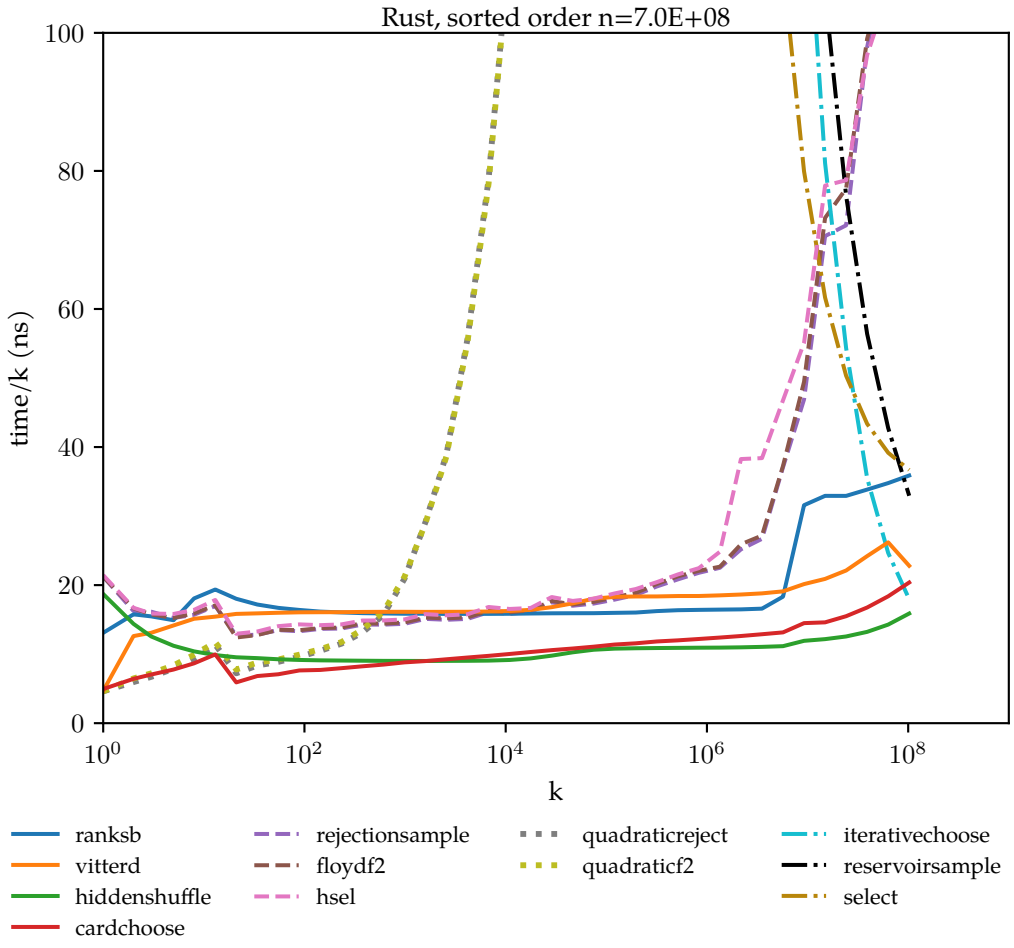


Figure 6: Rust, sorted order, $n = 7.0 \times 10^8$

case of any single algorithm for sorted output in Rust, and in C++ only Hidden Shuffle's is smaller. Where $n \geq 4k$, it takes 0.59 to 2.5 times as long as Cardchoose in C++ and 0.92 to 5.0 times as long in Rust; where $n < 2k$ and $k \geq 10^5$, it is faster than Cardchoose in both.

For random output, the hash-based algorithms are competitive only with a fast hash table. With C++ standard library hash tables, Cardchoose is faster than set-based rejection sampling, Floyd's F2 and HSEL at every point with $k \leq n/2$, where they take 1.10 to 8.9 times as long, though Hidden Shuffle followed by a shuffle is faster still wherever $n \geq 16k$ and $k \geq 1000$. With Rust's flat hash tables, HSEL is faster than Cardchoose wherever $1000 \leq k < 10^5$ and $n \geq 2k$, and Cardchoose is faster wherever $k \geq 10^7$ and $n \geq 2k$; it is never more than 1.38 times slower than the fastest of these four for $k \leq n/2$. In both languages, SELECT is the fastest algorithm wherever $2k \leq n < 16k$ and n is under 8 million; for larger n, its array is large enough that glibc's `malloc` maps it afresh on every call, and faulting its pages in again makes it up to 2.1 times slower than the fastest. The hash-based algorithms slow down at large k, once the hash table no longer fits in cache, and as k approaches n. In theory, their $\mathcal{O}(k)$ time should eventually beat Cardchoose's $\mathcal{O}(k \log k)$, but not at any k we tested: in Rust, wherever $k \geq 10^7$ and $n \geq 2k$, Cardchoose takes 0.23 to 0.39 times as long as each of them. For very large k, a parallel algorithm [RGG89; San+18] should generally be used; that of Sanders et al. calls a sequential algorithm at the leaves of the parallel tree it builds.

5 Choosing algorithms

We score each algorithm by its worst time per number output in each decade of k: for each band, the most its time per call divided by k reaches at any point with k in the band, over every n tested. This says what it costs at worst, per number, for any n and any k in that range. The bands start at $k = 10$, 100, 1,000 and so on, the last running from 10^7 to the largest k tested; below 10, per-call overhead swamps everything. Each band also has a floor: the score of choosing the fastest of all twelve algorithms separately at every point, which no algorithm or set of algorithms can beat. We give worst cases rather than averages, since any average depends on which values of n and k we chose to test, and score each band of k separately since otherwise the largest k, where everything is slowest per number, would decide every ranking.

Table 2 to Table 5 give, for each band, the best algorithm, with any within 3% of it, which is within the noise between runs, marked ≈; where any of those needs floating point (Hidden Shuffle and Algorithm D), the best algorithm that does not; then Cardchoose, and the floor. Each algorithm was timed only until a call took more than 30 seconds; only the quadratic algorithms reached that, from $k = 1.3 \times 10^6$.

Three algorithms are never far from the fastest per number output at any k: Cardchoose, Hidden Shuffle and Algorithm D are never worse than 44 ns per

k from	Best	Best integer-only	Cardchoose	Floor
10	quadratic F2 4.83 ns		16.4 ns	4.63 ns
100	quadratic F2 17.0 ns		18.8 ns	10.8 ns
1,000	Cardchoose 20.9 ns		20.9 ns	11.2 ns
10^4	Cardchoose 23.1 ns ≈ Hidden Shuffle 23.1 ns	Cardchoose 23.1 ns	23.1 ns	11.8 ns
10^5	Hidden Shuffle 22.4 ns	Cardchoose 24.7 ns	24.7 ns	14.2 ns
10^6	Hidden Shuffle 24.7 ns	Cardchoose 29.4 ns	29.4 ns	17.7 ns
10^7	Hidden Shuffle 31.7 ns	Cardchoose 43.8 ns	43.8 ns	22.1 ns

Table 2: C++, random order: the best algorithms by worst time per number output in each band of k

k from	Best	Best integer-only	Cardchoose	Floor
10	Cardchoose 15.8 ns		15.8 ns	13.2 ns
100	Cardchoose 17.9 ns		17.9 ns	13.9 ns
1,000	Cardchoose 19.5 ns		19.5 ns	13.9 ns
10^4	Hidden Shuffle 21.4 ns ≈ Cardchoose 21.5 ns	Cardchoose 21.5 ns	21.5 ns	13.9 ns
10^5	Hidden Shuffle 21.5 ns	Cardchoose 23.4 ns	23.4 ns	13.9 ns
10^6	Hidden Shuffle 21.7 ns	Cardchoose 26.8 ns	26.8 ns	14.3 ns
10^7	Hidden Shuffle 22.1 ns	Cardchoose 33.8 ns	33.8 ns	14.7 ns

Table 3: C++, sorted order: the best algorithms by worst time per number output in each band of k

k from	Best	Best integer-only	Cardchoose	Floor
10	quadratic F2 5.21 ns		11.8 ns	4.55 ns
100	HSEL 10.0 ns		11.2 ns	8.24 ns
1,000	HSEL 9.77 ns		12.1 ns	8.84 ns
10^4	HSEL 10.4 ns		13.2 ns	9.55 ns
10^5	HSEL 13.5 ns		14.9 ns	12.7 ns
10^6	Cardchoose 20.0 ns		20.0 ns	18.3 ns
10^7	Hidden Shuffle 37.6 ns ≈ Cardchoose 38.1 ns	Cardchoose 38.1 ns	38.1 ns	27.5 ns

Table 4: Rust, random order: the best algorithms by worst time per number output in each band of k

k from	Best	Best integer-only	Cardchoose	Floor
10	Cardchoose 11.1 ns		11.1 ns	9.87 ns
100	Cardchoose 10.1 ns		10.1 ns	8.81 ns
1,000	Cardchoose 11.1 ns		11.1 ns	9.15 ns
10^4	Cardchoose 12.2 ns		12.2 ns	10.7 ns
10^5	Cardchoose 13.3 ns		13.3 ns	11.5 ns
10^6	Cardchoose 17.2 ns		17.2 ns	14.5 ns
10^7	Hidden Shuffle 22.5 ns	Cardchoose 24.8 ns	24.8 ns	15.9 ns

Table 5: Rust, sorted order: the best algorithms by worst time per number output in each band of k

number in any case, while every other algorithm is worse than 100 ns per number somewhere it was tested. Cardchoose is the simplest of the three: it is a few lines of integer arithmetic, while the other two need floating point and are considerably more involved. Algorithm D switches to Algorithm A once $n \leq 13k$, and Hidden Shuffle runs three stages in turn, the last of them Algorithm A.

With sorted output, Cardchoose is the best algorithm at every k below 10^7 in Rust, and Hidden Shuffle above that; in C++, Cardchoose is the best below 10^4, tied with Hidden Shuffle from 10^4, and Hidden Shuffle is the best from 10^5. With random output, quadratic F2 is the best for small k, below 1,000 in C++ and 100 in Rust, where it needs neither a sort nor a shuffle; in Rust, with its fast hash tables, HSEL is the best from there to 10^6. Wherever the best algorithm needs floating point, the best one that does not is Cardchoose.

An algorithm is dominated by bands if another scores no worse in any band and better in at least one. Eight algorithms are dominated by bands in every case: Cardchoose dominates seven of them in every case, quadratic rejection, set-based rejection, RANKSB, Floyd's F2, iterative choosing, reservoir sampling and SELECT, and Hidden Shuffle dominates the eighth, Algorithm D. Because a band's score is the worst over every n, that does not make them useless: iterative choosing, reservoir sampling and SELECT are among the fastest where n/k is small, and iterative choosing belongs to the best sets for sorted output below. No algorithm is at least as fast as another at every point timed in every case, but in each case Cardchoose is at least as fast as some of the others at every point: in Rust with sorted output, as fast as six of the other eleven.

Since no algorithm is the best at every k, an implementation might implement several, and choose which one to use based on the values of n and k. Table 6 shows, for each language and order, the best set of one, two or three algorithms, if each point uses the fastest algorithm in the set. A set's score in a band is its worst time per number output there, as for a single algorithm, and the table gives the most that exceeds the band's floor, as a ratio, in the set's worst band: $1.00\times$ means the set is as good as all twelve algorithms together in every band. It also gives the most each set is slower than the best set of three, in time per number, in any band. Where the best set needs floating point, it also gives the best set that does not, or says that a third such algorithm is no better than two. This assumes the best choice is made at every point; an implementation has to choose by a rule, such as thresholds on k and n/k.

With sorted output, the best partner is iterative choosing, which wins where k is a large fraction of n; with random output, it is quadratic F2, which wins at small k. In C++, three algorithms including Hidden Shuffle are as good as all twelve together; in sorted order, no third integer-only algorithm improves on the best integer-only pair.

If an implementation has only one algorithm, Cardchoose is a good choice. It is the best single algorithm in three of the four cases; in the fourth, C++ with

Set	Algorithms	Against floor	Most over best three
C++, random order			
best 1	Cardchoose	3.53×	21.7 ns
best 2	quadratic F2 + Cardchoose	1.98×	21.7 ns
best 3	quadratic rejection + Hidden Shuffle + SELECT	1.00×	
best 3 integer only	quadratic rejection + RANKSB + SELECT	1.76×	11.6 ns
Cardchoose alone	Cardchoose	3.53×	21.7 ns
C++, sorted order			
best 1	Hidden Shuffle	1.68×	9.00 ns
best 1 integer only	Cardchoose	2.29×	19.0 ns
best 2	iterative choosing + Hidden Shuffle	1.10×	1.26 ns
best 2 integer only	iterative choosing + Cardchoose	1.89×	13.0 ns
best 3	iterative choosing + Hidden Shuffle + SELECT	1.00×	
best 3 integer only	no better than 2		
Cardchoose alone	Cardchoose	2.29×	19.0 ns
Rust, random order			
best 1	Cardchoose	2.58×	9.98 ns
best 2	quadratic F2 + Cardchoose	1.39×	9.98 ns
best 3	quadratic rejection + SELECT + Cardchoose	1.27×	
Cardchoose alone	Cardchoose	2.58×	9.98 ns
Rust, sorted order			
best 1	Cardchoose	1.56×	8.94 ns
best 2	iterative choosing + Cardchoose	1.16×	2.49 ns
best 3	iterative choosing + Hidden Shuffle + Cardchoose	1.03×	
best 3 integer only	no better than 2		
Cardchoose alone	Cardchoose	1.56×	8.94 ns

Table 6: The best sets of up to three algorithms, by how far their worst time per number output in a band of k exceeds the band's floor in their worst band, with the most they are slower than the best three in any band

sorted output, only the two floating-point algorithms beat it, because libc++'s `std::sort` is slower than Rust's sort. It is the best integer-only algorithm in every case. Implemented alone, it is never more than 9 to 22 ns per number slower than the best three algorithms together, in any case and at any k. Its largest ratios to the floor are where calls are cheap: with random output, it is 3.53 times the floor in C++ and 2.58 times in Rust for $k < 100$, where quadratic F2 needs neither a sort nor a shuffle, but no call to Cardchoose there takes more than 1.5 µs. With sorted output, it is 2.29 times the floor in C++ and 1.56 times in Rust for $k \geq 10^7$, where Hidden Shuffle wins.

References

[AD85] J. H. Ahrens and U. Dieter. "Sequential Random Sampling". In: *ACM Trans. Math. Softw.* 11.2 (June 1985), pp. 157–169. ISSN: 0098-3500. DOI: 10.1145/214392.214402.

[BF87] Jon Bentley and Bob Floyd. "Programming Pearls: A Sample of Brilliance". In: *Commun. ACM* 30.9 (Sept. 1987), pp. 754–757. ISSN: 0001-0782. DOI: 10.1145/30401.315746.

[BM73] David Blackwell and James B. MacQueen. "Ferguson distributions via Pólya urn schemes". In: *The Annals of Statistics* 1.2 (1973), pp. 353–355. DOI: 10.1214/aos/1176342372.

[BS80] Jon Louis Bentley and James B. Saxe. "Generating Sorted Lists of Random Numbers". In: *ACM Transactions on Mathematical Software* 6.3 (1980), pp. 359–364. DOI: 10.1145/355900.355907.

[Cro23] Paul Crowley. *Simpler and faster implementation of Floyd's F2*. rust-random/rand pull request 1277. 1st Jan. 2023. URL: https://github.com/rust-random/rand/pull/1277.

[Cro26] Paul Crowley. *Cardchoose: dealing a hand of cards is subtler than it looks*. YouTube. 25th Oct. 2026. URL: https://youtu.be/5br30I2trKU.

[Dev86] Luc Devroye. *Non-Uniform Random Variate Generation*. New York: Springer-Verlag, 1986. URL: https://luc.devroye.org/rnbookindex.html.

[EN82] Jarmo Ernvall and Olli Nevalainen. "An Algorithm for Unbiased Random Sampling". In: *The Computer Journal* 25.1 (Feb. 1982), pp. 45–47. ISSN: 0010-4620. DOI: 10.1093/comjnl/25.1.45.

[Fel68] William Feller. *An Introduction to Probability Theory and Its Applications*. 3rd ed. Vol. 1. John Wiley and Sons, Inc., 1968.

[FMR62] C. T. Fan, Mervin E. Muller and Ivan Rezucha. "Development of Sampling Plans by Using Sequential (Item by Item) Selection Techniques and Digital Computers". In: *Journal of the American Statistical Association* 57.298 (1962), pp. 387–402. DOI: 10.1080/01621459.1962.10480667.

[Fri71] Jerome H. Friedman. "Random event generation with preferred frequency distributions". In: *Journal of Computational Physics* 7.2 (1971), pp. 201–218. DOI: 10.1016/0021-9991(71)90085-4.

[GB84] P. Gupta and G. P. Bhattacharjee. "An efficient algorithm for random sampling without replacement". In: *Foundations of Software Technology and Theoretical Computer Science*. Vol. 181. Lecture Notes in Computer Science. Springer, 1984, pp. 435–442. DOI: 10.1007/3-540-13883-8_89.

[GH77] Seymour Evan Goodman and Stephen T. Hedetniemi. *Introduction to the Design and Analysis of Algorithms*. McGraw-Hill Inc., 1977. ISBN: 0-07-023753-0.

[Jak85] Matti Jakobsson. "Sampling without Replacement in Linear Time". In: *The Computer Journal* 28.4 (1985), pp. 412–413. DOI: 10.1093/comjnl/28.4.412.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896842.

[LP17] David A. Levin and Yuval Peres. *Markov Chains and Mixing Times*. 2nd ed. American Mathematical Society, 2017. URL: https://pages.uoregon.edu/dlevin/MARKOV/mcmt2e.pdf.

[NG09] Robert K. Niven and Marian Grendar. "Generalized classical, quantum and intermediate statistics and the Pólya urn model". In: *Physics Letters A* 373.6 (2009), pp. 621–626. arXiv: 0808.2102.

[NW78] Albert Nijenhuis and Herbert S. Wilf. *Combinatorial Algorithms for Computers and Calculators*. 2nd ed. Academic Press, 1978.

[Ran18] The Rand Project Developers. *rand. src/seq/index.rs*. GitHub. 2018. URL: https://github.com/rust-random/rand/blob/19897e53c1908c5a193e2aacff59170e3e72b8de/src/seq/index.rs#L231.

[RGG89] Vijayan Rajan, R.K. Ghosh and P. Gupta. "An efficient parallel algorithm for random sampling". In: *Information Processing Letters* 30.5 (1989), pp. 265–268. ISSN: 0020-0190. DOI: 10.1016/0020-0190(89)90206-8.

[ric15] rici. *Answer to "Randomly selecting k different numbers in a range"*. Stack Overflow. 31st May 2015. URL: https://stackoverflow.com/a/30559195.

[RV99] P.S. Rouzankin and A.V. Voytishek. "On the cost of algorithms for random selection". In: *Monte Carlo Methods and Applications* 5.1 (1999), pp. 39–54. DOI: 10.1515/mcma.1999.5.1.39.

[San+18] Peter Sanders et al. "Efficient Parallel Random Sampling—Vectorized, Cache-Efficient, and Online". In: *ACM Transactions on Mathematical Software* 44.3 (Jan. 2018), pp. 1–14. DOI: 10.1145/3157734. arXiv: 1610.05141.

[SC21] Michael Shekelyan and Graham Cormode. "Sequential Random Sampling Revisited: Hidden Shuffle Method". In: *Proceedings of the 24th International Conference on Artificial Intelligence and Statistics (AISTATS)*. Vol. 130. Proceedings of Machine Learning Research. 2021. URL: https://proceedings.mlr.press/v130/shekelyan21a.html.

[She] Michael Shekelyan. *sampleiterator*. GitHub. URL: https://github.com/shekelyan/sampleiterator.

[Tin21a] Daniel Ting. *Fun with Optimal Random Sampling*. Tableau Engineering Blog. 12th Apr. 2021. URL: https://www.tableau.com/blog/fun-optimal-random-sampling.

[Tin21b] Daniel Ting. *Simple, Optimal Algorithms for Random Sampling Without Replacement*. 2021. DOI: 10.48550/ARXIV.2104.05091. URL: https://arxiv.org/abs/2104.05091.

[TN82] J. Teuhola and O. Nevalainen. "Two efficient algorithms for random sampling without replacement". In: *International Journal of Computer*

Mathematics 11.2 (1982), pp. 127–140. DOI: 10.1080/00207168208803304.

[Vit84] Jeffrey Scott Vitter. "Faster Methods for Random Sampling". In: *Commun. ACM* 27.7 (July 1984), pp. 703–718. ISSN: 0001-0782. DOI: 10.1145/358105.893.

[Vit85] Jeffrey S. Vitter. "Random Sampling with a Reservoir". In: *ACM Trans. Math. Softw.* 11.1 (Mar. 1985), pp. 37–57. ISSN: 0098-3500. DOI: 10.1145/3147.3165.

[Vit87] Jeffrey Scott Vitter. "An Efficient Algorithm for Sequential Random Sampling". In: *ACM Trans. Math. Softw.* 13.1 (Mar. 1987), pp. 58–67. ISSN: 0098-3500. DOI: 10.1145/23002.23003.