A new algorithm for sampling without replacement

Paul Crowley

Google LLC

December 28, 2022

Abstract

The literature contains several algorithms for fairly choosing k distinct natural numbers below n. However, they either require a hash-based data structure such as a set or dictionary, or show asymptotically poor performance for some values of k, n. We present here an algorithm that requires no such data structure or auxiliary storage, only an integer sort taking $\mathcal{O}(k \log k)$ time; the algorithm is based on a method of fair multiset choosing. In our benchmarks the algorithm outperforms all we compare it to wherever $k > 100$ and $n > 100k$, and has acceptable performance for all values of k, n.

1 Introduction

The problem of choosing a subset of $\{0, 1, \ldots, n-1\}$ such that every k-element subset is equally likely arises in several contexts. Several programming languages include facilities for this in their standard libraries, for example Python's `random.sample()` or Rust's `rand::seq::index::sample`. In section 2 we discuss several known algorithms to this end; these all have their strengths, but all suffer from one of two disadvantages:

- Some are fast for certain values of k, n, e.g. where k is small or where $\frac{n}{k}$ is small, but very slow for other values.
- Others achieve much better asymptotic performance, but at the cost of updating a hash-based data structure such as a set or dictionary for each of the k values generated, adding a significant constant multiplier to the overall runtime as well as memory cost.

We here present an algorithm which requires no storage beyond the array in which the result is written but achieves a $\mathcal{O}(k \log k)$ runtime; the extra $\log k$ term is due to an integer sort over the results array.

In Python the algorithm works as follows:

```python
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
```

2 Comparison algorithms

Let $\mathbb{N}_{<n}$ denote the n-element set $\{0, 1, \ldots, n-1\}$. Someone who calls a routine that promises a fairly drawn k-element subset of $\mathbb{N}_{<n}$ might be expecting it to return one of three different things:

- an array of integers in sorted order;
- an array of integers in fair random order;
- a set of integers in some kind of efficient set structure (e.g. a hash table).

Both Python's `random.sample()` and Rust's `rand::seq::index::sample` return in a fair random order.

A routine which returns one of these can straightforwardly be converted into a routine that returns another, using e.g. a sort or a Fisher-Yates shuffle as appropriate. However each of these algorithms returns one of the above types "naturally", and so returning a different one will take an extra step, which means that which algorithm is most efficient can depend on which of the above three the caller wants to receive.

Algorithm	Order	Data structure	Time
Quadratic rejection sampling	Random		k^2
Set-based rejection sampling	Random	Set	k
Algorithm S	Sorted		n
Algorithm R	Random		n
SELECT	Random	n-element array	n
HSEL	Random	Dictionary	k
Floyd's F2		Set	k
Quadratic F2			k^2
Our work	Sorted		$k \log k$

2.1 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
```

Start with an empty array of integers. Generate an integer $0 \leq x < n$, and check if x is already present in the array by iterating through the array checking every element. If it is not present, append it to the array. Repeat this process until the array is of the desired length.

2.2 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 use a data structure representing sets with an efficient membership test. [GH77]

2.3 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
```

Iterate in order through each candidate to add to the list, calculate for each the probability it should be part of the list given the number of items added so far and the number remaining, and add it with that probability. [Knu97] [FMR62, Method 1, p.391]

2.4 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
```

Initialize a k-element array with the elements of $\mathbb{N}_{<k}$ in random order. Iterate over the elements of $\mathbb{N}_{<n} \setminus \mathbb{N}_{<k}$, and replace a random element of the array with each element with the appropriate probability. [Knu97]

2.5 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
```

Use a Fisher-Yates shuffle to randomize only the first k items of an array of the elements of $\mathbb{N}_{<n}$. [GH77]

2.6 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
```

As above, except that instead of allocating and initializing an n-element array, we use a “virtual” array, with a hash table storing only the modified elements. [EN82]

2.7 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
```

A twist on set-based rejection that uses compensatingly biased samples to avoid multiple tests. [BF87]

2.8 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
```

Quadratic variant on Floyd's F2 which needs no external data structure. Inspired by the implementation in [Har18] but with some optimizations.

3 Notation

Let $\mathbb{N}_{<n} \overset{\text{def}}{=} \{i \in \mathbb{N} : i < n\}$, the set of natural numbers less than n where $\mathbb{N} = \{0, 1, 2, \ldots\}$. Let $\binom{S}{k} \overset{\text{def}}{=} \{s \subseteq S : |s| = k\}$, 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 $[0, 0, 1]$. We represent multisets as functions $m : U \to \mathbb{N}$; the set U is the *universe*, and in what follows we consider only finite universes. For any $y \in U$ we call $m(y)$ the *multiplicity* of y in m. A multiset has a *cardinality* $|m| \overset{\text{def}}{=} \sum_{x \in U} m(x)$ and a *support* set $\text{Supp}(m) \overset{\text{def}}{=} \{x \in U : m(x) > 0\}$.

Where U is clear from context, for any set $S \subseteq U$ we consider $\overline{S}$ to be S viewed as a multiset, i.e. the multiset such that $\text{Supp}(\overline{S}) = S$ and $|\overline{S}| = |S|$:

$$\overline{S}(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$$

The sum of multisets $m_1 \uplus m_2$ is the multiset such that $(m_1 \uplus m_2)(x) = m_1(x) + m_2(x)$ for all $x \in U$. $m = m_1 \ominus m_2$ is the unique multiset such that $m_1 = m \uplus m_2$, and is defined only if this exists.

We define random sampling from a multiset to be analogous to drawing from a set, where each element's probability of being drawn is proportional to its multiplicity: $\Pr[x = y | x \leftarrow\$\, m] = \frac{m(y)}{|m|}$.

Define $\left(\!\binom{U}{k}\!\right) \overset{\text{def}}{=} \{m \in U \to \mathbb{N} : |m| = k\}$ the set of multisets over universe U of cardinality k, and $\left(\!\binom{n}{k}\!\right) \overset{\text{def}}{=} \left| \left(\!\binom{\mathbb{N}_{<n}}{k}\!\right) \right|$ the number of distinct k-cardinality multisets over an n-element universe.

4 Intuition behind the algorithm

Following [Fel57], we can represent an element of $\left(\!\binom{\mathbb{N}_{<7}}{8}\!\right)$ such as $[0, 0, 0, 1, 5, 5, 5, 5]$ using six bars and eight stars:

$$\star \ \star \ \star \ | \ \star \ | \ | \ | \ | \ \star \ \star \ \star \ \star \ |$$

The six bars divide the line into seven "bins", one for each element of $\mathbb{N}_{<7}$; the number of stars in each bin indicates the multiplicity of that element in our multiset. With this technique, we define a bijection between $\left(\!\binom{\mathbb{N}_{<n}}{k}\!\right)$ and $\binom{\mathbb{N}_{<n+k-1}}{k}$ wherever $n > 0$, from which we infer that $\left(\!\binom{n}{k}\!\right) = \binom{n+k-1}{k}$ where $n > 0$.

Suppose now that we want to choose 6 integers from the range $\mathbb{N}_{<11}$ at random. As per the above we see that this is equivalent to choosing a multiset from $\left(\!\binom{\mathbb{N}_{<6}}{6}\!\right)$ or a way of arranging 6 stars and 5 bars into a sequence. To choose fairly, we start with a sequence of five bars $|||||$ and insert six stars, one after another, in randomly chosen positions.

For the first star, there are six possible places it can go, and we choose one at random: $||| \star ||$. We record that it has three bars to its left $[3]$.

There are now six items in the sequence, and thus seven possible places to place the second star. In two of those seven cases—before the existing star, and after it—it will have three bars to its left. Let's suppose we choose the first position: $\star||| \star ||$. We append the number of bars to the left of the new star to our record, which becomes $[3, 0]$. The positions have changed from $\{3\}$ to $\{4, 0\}$ but because we're not recording positions, only bars to the left, we don't need to update the first entry.

We place three more stars in random positions, ending up with $\star \star \star| \star || \star ||$ and a record of $[3, 0, 0, 1, 0]$. Now there's one star left to place; there is one position it can be placed after the last bar, but four before the first bar, so it is four times more likely to be placed before the first bar than after the last. Let's suppose it's placed at the fifth position: $\star \star \star| \star \star|| \star ||$, $[3, 0, 0, 1, 0, 1]$ We now want to know the position of each star; we find this by sorting the list $[0, 0, 0, 1, 1, 3]$ and adding to each entry its index so that the value reflects the stars as well as the bars to its left, returning the answer $\{0, 1, 2, 4, 5, 8\}$

$$\begin{array}{ccccccccccc} 0 & 0 & 0 & & 1 & 1 & & & 3 & & \\ \star & \star & \star & | & \star & \star & | & | & \star & | & | \\ 0 & 1 & 2 & & 4 & 5 & & & 8 & & \end{array}$$

Thus to get a sequence without duplicates, we start with a procedure that deliberately biases towards duplicates.

5 Proof

5.1 Multiset choosing

We consider the problem of choosing an element from $\left(\!\!\binom{U}{k}\!\!\right)$ fairly. For example, $\left(\!\!\binom{\mathbb{N}_{<2}}{3}\!\!\right) = \{[0, 0, 0], [0, 0, 1], [0, 1, 1], [1, 1, 1]\}$; for each of these four, our algorithm should output it with probability $\frac{1}{4}$. If we choose three independent elements from U and add them together to make a multiset, our answer will favour multisets with lower multiplicities; in accordance with the binomial theorem, $[0, 0, 0]$ will be drawn with probability $\frac{1}{8}$, while $[0, 0, 1]$ will be drawn with probability $\frac{3}{8}$, reflecting the three ways this multiset can be written as a sequence.

Algorithm 1: Fair multiset choosing

```
procedure ChooseMultiset($(U, k)$)
    if k $= 0$ then
        return $\overline{\varnothing}$
    else
        $m' \leftarrow$ ChooseMultiset$(U, k - 1)$
        $x \leftarrow\!\$\ \overline{U} \uplus m'$
        return $m' \uplus \overline{\{x\}}$
    end if
end procedure
```

To address this, in ChooseMultiset we introduce a counter-bias in the selection of x, which favours duplicates. ChooseMultiset is trivially fair for $k = 0$, so we assume it is fair for $k - 1$ and proceed by induction. For a multiset $m \in \left(\!\!\binom{U}{k}\!\!\right)$:

$$
\begin{aligned}
&\Pr\left[m' = m | m' \leftarrow \textsc{ChooseMultiset}(U, k)\right] \\
=&\Pr\left[m' \uplus \overline{\{x\}} = m | m' \leftarrow \textsc{ChooseMultiset}(U, k-1), x \leftarrow\!\!\$\ \overline{U} \uplus m'\right] \\
=&\Pr\left[m' \uplus \overline{\{x\}} = m | m' \leftarrow\!\!\$\ \left(\!\!\binom{U}{k-1}\!\!\right), x \leftarrow\!\!\$\ \overline{U} \uplus m'\right] \\
=&\sum_{y \in \mathrm{Supp}(m)} \Pr\left[m' = m \ominus \overline{\{y\}} | m' \leftarrow\!\!\$\ \left(\!\!\binom{U}{k-1}\!\!\right)\right] \Pr\left[x = y | x \leftarrow\!\!\$\ \overline{U} \uplus (m \ominus \overline{\{y\}})\right] \\
=&\sum_{y \in \mathrm{Supp}(m)} \frac{1}{\left(\!\!\binom{|U|}{k-1}\!\!\right)} \frac{(U \uplus (m \ominus \overline{\{y\}}))(y)}{|U \uplus (m \ominus \overline{\{y\}})|} \\
=&\frac{1}{\left(\!\!\binom{|U|}{k-1}\!\!\right)} \sum_{y \in \mathrm{Supp}(m)} \frac{1 + (m \ominus \overline{\{y\}})(y)}{|U| + |m \ominus \overline{\{y\}}|} \\
=&\frac{1}{\binom{|U|+k-2}{k-1}} \sum_{y \in \mathrm{Supp}(m)} \frac{m(y)}{|U| + k - 1} \\
=&\frac{k}{(|U| + k - 1)\binom{|U|+k-2}{k-1}} \\
=&\frac{1}{\binom{|U|+k-1}{k}} \\
=&\frac{1}{\left(\!\!\binom{|U|}{k}\!\!\right)}
\end{aligned}
$$

This algorithm is straightforward to implement. This Python implementation takes integers n, k and returns a sorted list of integers in $\mathbb{N}_{<n}$.

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

5.2 Multisets and choices

To generate a random k-element subset of $\mathbb{N}_{<n}$, we can apply this method to generate a random multiset from the universe $\mathbb{N}_{<n-k+1}$ and use the "stars and bars" bijection to convert to the desired subset.

Our implementation of ChooseMultiset represents its result in $\left(\!\!\binom{\mathbb{N}_{<n-k+1}}{k}\!\!\right)$ as a sorted list of integers. In "stars and bars" representation, each entry in the list

represents a star, and the integer is the number of bars to its left. Converting this to a sorted k-element subset of $\mathbb{N}_{<n}$ simply means adding to each the number of stars to its left, which is equal to its position in the sequence; the Python code below returns a sorted list of k distinct integers in $\mathbb{N}_{<n}$ fairly among all ways of doing so.

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

6 Benchmarks

(https://github.com/ciphergoth/sansreplace/cpp) includes C++ implementations of these algorithms, which were benchmarked for a variety of values of n and k, for both sorted output and random output, on a Hewlett-Packard Z840 workstation:

- CPU: Intel Xeon E5-2690 v3
- Frequency: 3.5GHz
- Cache: 768kiB L1, 3 MiB L2, 30MiB L3
- Compiler: gcc 12.2, `-Ofast`

In Figure 1 and Figure 2 we graph time/k in ns against k for two different values of n for random order output, and in Figure 3 a single value of n for sorted output. Animated graphs showing many values of n up to 701,408,733 (the 44th Fibonacci number) can be seen at https://github.com/ciphergoth/sansreplace/blob/master/results.md, and complete benchmark data at https://github.com/ciphergoth/sansreplace/blob/master/results/latest

Our algorithm outperforms all hash- and set-based algorithms for all values of n and k tested. In theory, the $\mathcal{O}(k)$ performance of these algorithms compared to the $\mathcal{O}(k \log k)$ performance of our algorithm should mean that there is a value of k at which these algorithms perform better. However, the poor cache coherence of these algorithms means that their performance more closely resembles $\mathcal{O}(k \log k)$ in practice. For very large k, a parallel algorithm such as [San+18] should generally be used; this algorithm calls a sequential algorithm at the leaves of the parallel tree it builds.

Quadratic rejection and quadratic F2, both outperform our algorithm for small k, while SELECT, reservoir sampling, and iterative choosing all outperform our algorithm for small n/k. However all of these behave pathologically for other values of n, k; ours is the fastest of all of the algorithms that perform reasonably for all values of n, k.

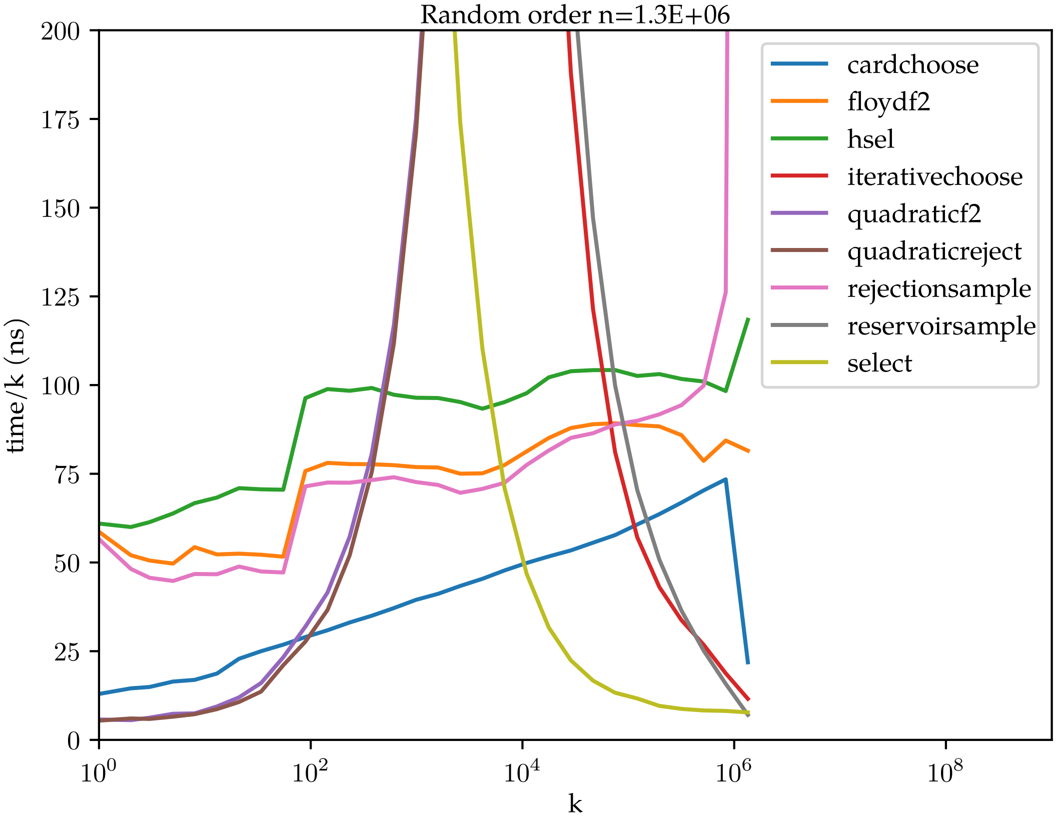

Figure 1: Random order, $n = 1.3 \times 10^6$

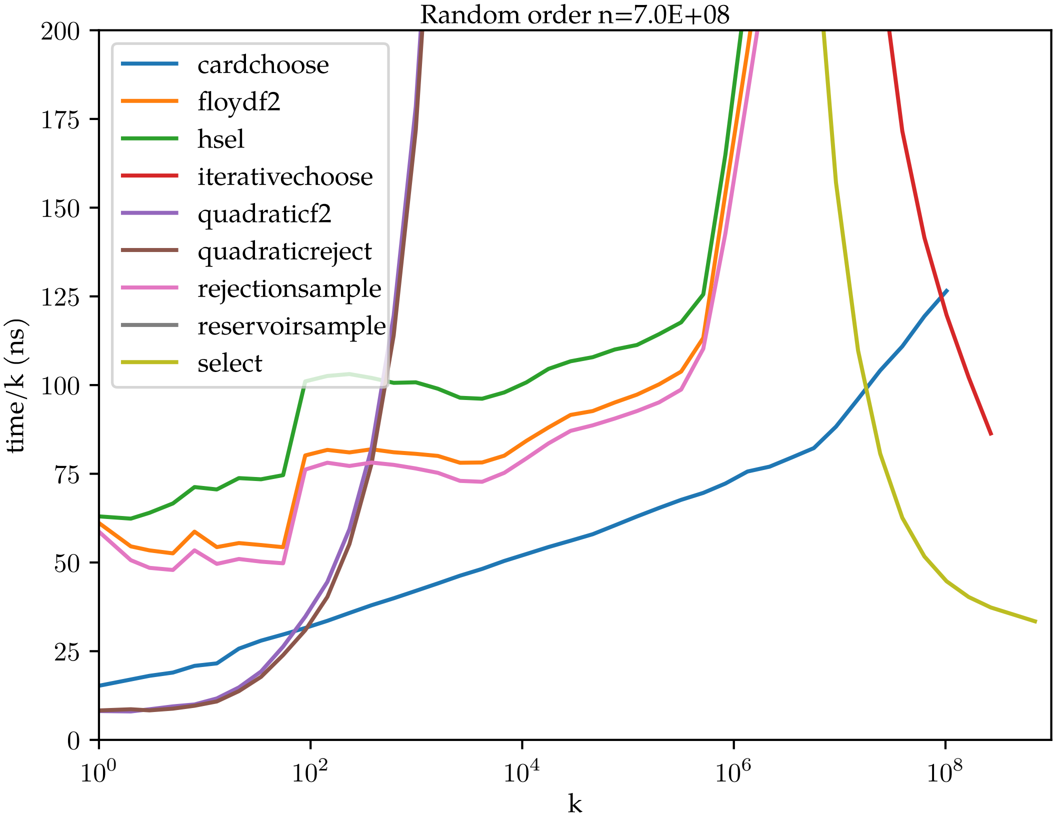

Figure 2: Random order, $n = 7.0 \times 10^9$

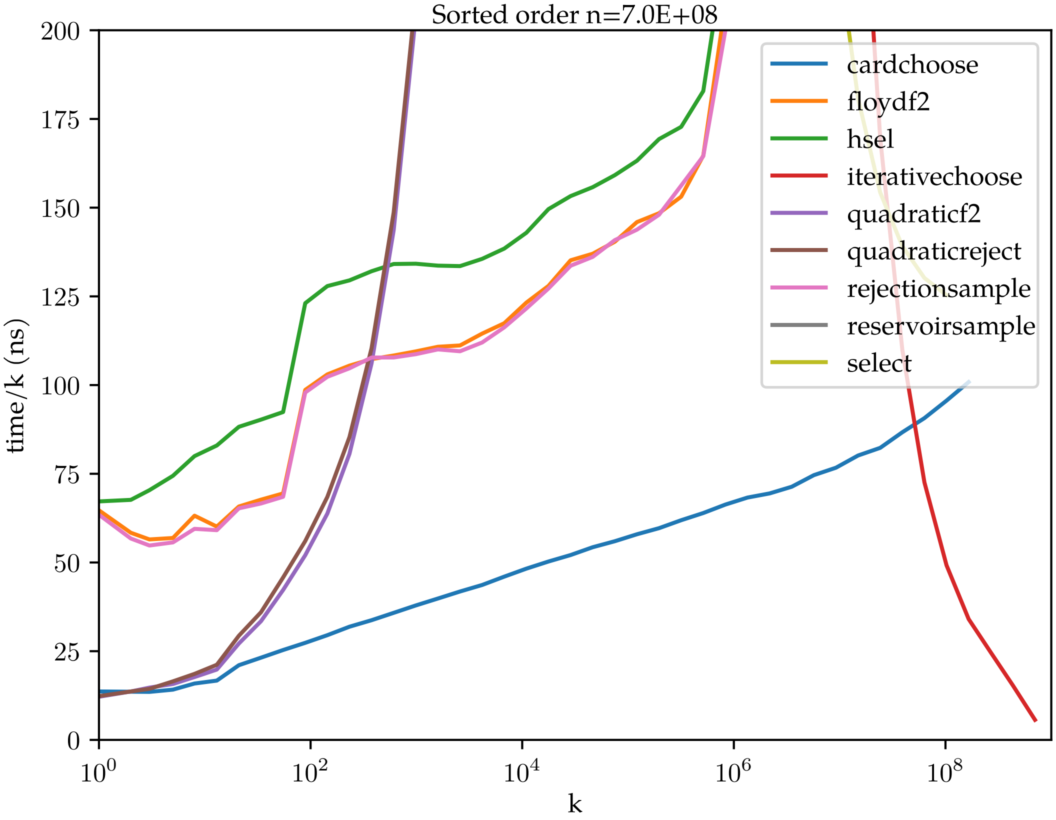

Figure 3: Sorted order, $n = 7.0 \times 10^9$

Quadratic F2	$(1.551 \times 10^{-10})k^2 + (1.617 \times 10^{-8})k - 1.195 \times 10^{-7}$
Our algorithm	$(4.675 \times 10^{-9})(k \log k) + (8.123 \times 10^{-9})k - 3.299 \times 10^{-9}$
SELECT	$(3.204 \times 10^{-10})n + (4.683 \times 10^{-9})k - 2.895 \times 10^{-8}$
Reservoir sampling	$(4.372 \times 10^{-9})n + (5.517 \times 10^{-9})k - 2.982 \times 10^{-8}$

Table 1: Time estimates for random order algorithms

Our algorithm	$(4.360 \times 10^{-9})(k \log k) + (5.911 \times 10^{-9})k - 2.552 \times 10^{-8}$
Iterative choosing	$(4.077 \times 10^{-9})n + (8.378 \times 10^{-9})k + 4.224 \times 10^{-9}$

Table 2: Time estimates for sorted order algorithms

A good implementation will implement multiple algorithms, and choose which one to use based on the values of n and k. For random order output, a combination of quadratic F2, our algorithm, and SELECT is likely to be near-optimal for all values, or if allocating extra memory on demand is not an option then reservoir sampling can be substituted for SELECT. For sorted order output, a combination of our algorithm and iterative choosing should suffice. Based on a least-squares curve fitting, the runtime estimates from Table 1 and Table 2 can be used to select which algorithm to use.

References

[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.

[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. eprint: https://academic.oup.com/comjnl/article-pdf/25/1/45/1410156/25-1-45.pdf.

[Fel57] William Feller. "An Introduction to Probability Theory and Its Applications". In: 3rd ed. Vol. 1. John Wiley and Sons, Inc., 1957. Chap. 2, pp. 38–39.

[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.

[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.

[Har18] Diggory Hardy. *Rand*. https://github.com/rust-random/rand/blob/master/src/seq/index.rs. 2018.

[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.

[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.