Proceedings of the 2025 Winter Simulation Conference
E. Azar, A. Djanatliev, A. Harper, C. Kogler, V. Ramamohan, A. Anagnostou, and S. J. E. Taylor, eds.

COMPUTING ESTIMATORS OF A QUANTILE AND CONDITIONAL VALUE-AT-RISK

Sha Cao[1], Truong Dang[1], James M. Calvin[1], and Marvin K. Nakayama[1]

[1]Dept. of Computer Science, New Jersey Institute of Technology, Newark, NJ, USA

ABSTRACT

We examine various sorting and selection methods for computing quantile and the conditional value-at-risk, two of the most commonly used risk measures in risk management scenarios. We study the situation where simulation data is already pre-generated, and perform timing experiments on calculating risk measures on the existing datasets. Through numerical experiments, approximate analyses, and existing theoretical results, we find that selection generally outperforms sorting, but which selection strategy runs fastest depends on several factors.

1 INTRODUCTION

The p-quantile of a random variable Y with a continuous distribution is a constant ξ such that exactly p of the distribution's mass lies below ξ. For example, the median is the 0.5-quantile. Many application areas, such as finance (McNeil et al. 2015) and manufacturing, employ a p-quantile with $p \approx 0$ or $p \approx 1$ as a risk measure of undesirable future outcomes. For example, if Y denotes the loss over the next year of a bank's credit portfolio, the bank may employ the p-quantile (also known as the *value-at-risk* (VaR) with $p \approx 1$ to specify how much capital to keep on hand to be able to absorb large unexpected losses with high probability (McNeil et al. 2015 Section 2.3.3).

Another popular risk measure is the *conditional value-at-risk* (CVaR), denoted by γ and defined as the conditional expectation of Y given that $Y > \xi$. (In some contexts, it is more appropriate to condition on $Y \leq \xi$ instead, but we focus here on conditioning on $Y > \xi$.) CVaR also goes by other names, including *tail conditional expectation*, *tail VaR*, and the *expected shortfall*.

For Y representing the output of a complicated stochastic model, analytically or numerically computing ξ and γ is typically not possible, so we often instead estimate them via *Monte Carlo* (MC) simulation. Commonly applied MC methods include *simple random sampling* (SRS), which is the focus of this paper, and *variance-reduction techniques* (Asmussen and Glynn 2007 Chapter V).

We examine different methods to compute SRS estimators of a quantile and CVaR. In particular, given a sample of SRS-generated data of size n (where n represents the number of runs of the simulation model), we study the efficiencies of algorithms for computing the SRS estimators from the observations for fixed n or as $n \to \infty$. To do this, we define notation: For two positive functions h_1 and h_2, we write "$h_1(n) = O(h_2(n))$" (resp., "$h_1(n) = \Omega(h_2(n))$") as $n \to \infty$ if there exist constants $c_0 > 0$ and $n_0 \geq 0$ such that $h_1(n) \leq c_0 h_2(n)$ (resp., $h_1(n) \geq c_0 h_2(n)$) for all $n \geq n_0$; also, "$h_1(n) = \Theta(h_2(n))$ as $n \to \infty$" means that both $h_1(n) = O(h_2(n))$ and $h_1(n) = \Omega(h_2(n))$ as $n \to \infty$. An approach sometimes suggested for computing a quantile estimator involves *sorting*, resulting in a worst-case running time of $\Omega(n \log n)$ as the sample size $n \to \infty$ (Cormen et al. 2022 Part II). But for SRS, a quantile estimator can instead be computed in worst-case linear time (i.e., $\Theta(n)$ as $n \to \infty$) using a *selection algorithm* (e.g., see Cormen et al. 2022 Section 9.3 or Goodrich and Tamassia 2015 Section 9.2.2). We investigate the efficiencies of sorting and selection methods through numerical experiments, approximate analyses, and existing theoretical results.

In many application settings, running a complicated simulation model to obtain a single output can take a large amount of time, and in the case of a transient simulation problem, this is repeated *n independent and identically distributed* (i.i.d.) times. Given the resulting n outputs, the additional time to compute the quantile or CVaR estimator can be quite small compared to the overall time to execute the n runs of the simulation model, in which case the user may not be concerned about employing an inefficient method for computing the risk measure. But there are situations in which it is important to have a fast technique for risk-measure computation given the data. Such settings arise in real-time applications with streaming data, such as self-driving vehicles and *digital twins* (National Academies of Sciences, Engineering, and Medicine 2024). Here, we have an actual operational physical system (such as a website serving specific information tailored to each visitor, which is studied, e.g., in Keslin et al. 2024), and simultaneously, a simulation of the system is running in the background, generating outputs based on a model of the current environment. At certain (possibly random) points in time (e.g., a new visitor arrives to the website), the real-time system must make an operational decision based on a risk measure, and this is to be done with only the most recent simulated outputs because of rapidly changing circumstances. To reduce the computational costs of constantly recomputing the risk measure, which is not needed all the time, it may be more efficient instead to compute the risk measure only when necessary using just the n most recent generated simulated data. In our web server example, providing a small response time requires being able to compute the risk measure quickly given the data. As another motivating (nonsimulation) example, a financial portfolio manager may design a trading strategy that makes buying and selling decisions based on only the most recent historical data because of ever-fluctuating economic conditions. This requires speedy algorithms to compute risk measures, such as the VaR and CVaR, from the given latest data. Other applications arise, e.g., in machine learning and databases (Chen and Guestrin 2016; Masson et al. 2019); e.g., see Section 6.

The rest of our paper unfolds as follows. Section 2 describes the mathematical framework for studying quantiles and CVaR. Sections 3 and 4 discuss algorithms for computing quantile and CVaR estimators, respectively, given the SRS-generated data. Section 5 presents empirical results from numerical experiments, and Section 6 concludes with future work including streaming applications discussed above.

2 MATHEMATICAL FRAMEWORK

Let F denote the *cumulative distribution function* (CDF) of the output Y of a (transient) simulation model, which we denote as $Y \sim F$, so $F(y) = \mathbb{P}(Y \leq y)$, where $\mathbb{P}$ is the probability measure. To simplify the discussion, we assume that F is absolutely continuous (with respect to Lebesgue measure) with density f. For many simulation models, Y has the form

$$Y = v(\mathbf{X}) \text{ for a random vector } \mathbf{X} = (X_1, X_2, \ldots, X_d) \sim G \text{ and function } v : \Re^d \to \Re, \tag{1}$$

where both v and G are assumed to be known. (Our notation follows the convention that boldfaced random variables are vectors, whereas nonbold quantities are scalars.) Thus, v transforms the input random vector $\mathbf{X}$ into an output Y having CDF F. While v and G are known, the complexity of v often renders F analytically intractable in practice. For a fixed $p \in (0,1)$, the p-quantile $\xi \equiv \xi_p$ of F (or of Y) is

$$\xi = F^{-1}(p) = \inf\{y : F(y) \geq p\},$$

so the median is the 0.5-quantile. When F is continuous, as we assume, exactly p of the mass of F lies below ξ_p, but $F(\xi_p) \geq p$ in general, where $F(\xi_p) > p$ may occur when F has discontinuities. We will assume throughout that $f(\xi_p) > 0$, ensuring that the equation $F(y) = p$ has a unique root $y = \xi_p$. In finance, ξ_p is called the *value-at-risk* (VaR), typically for $p \approx 0$ or $p \approx 1$ (McNeil et al. 2015 Section 2.3.2).

For a quantile (or risk) level $p \in (0,1)$, we define the CVaR as the conditional expectation

$$\gamma \equiv \gamma_p = \frac{1}{1-p} \int_p^1 F^{-1}(u)\,\mathrm{d}u = \mathbb{E}[Y \mid Y > \xi_p], \tag{2}$$

which satisfies $\gamma_p \geq \xi_p$, where $\mathbb{E}$ denotes the expectation operator. We will assume that $\mathbb{E}[|Y|] < \infty$, ensuring that γ is finite.

3 SIMPLE RANDOM SAMPLING FOR ESTIMATING A QUANTILE

As F usually cannot be computed analytically or numerically, the same applies to ξ_p and γ_p, so we instead estimate them via MC, where we focus here on the estimation of ξ_p. (Section 4 considers estimating CVaR.) A typical approach (Dong and Nakayama 2019) to estimating ξ_p via an MC method $\mathfrak{M}$ is to first use $\mathfrak{M}$ to construct an estimator $\widehat{F}_{\mathfrak{M},n}$ of F based on a sample size n, and invert it to obtain $\widehat{\xi}_{\mathfrak{M},n} = \widehat{F}^{-1}_{\mathfrak{M},n}(p)$ as the method-$\mathfrak{M}$ estimator of $\xi_p = F^{-1}(p)$. We now explain this process when $\mathfrak{M} = \mathrm{SRS}$. Note that $F(y) = \mathbb{P}(Y \le y) = \mathbb{E}[I(Y \le y)]$, where $I(\cdot)$ denotes the indicator function, which equals 1 (resp., 0) when its argument is true (resp., false). To estimate F via SRS, we run our simulation model n i.i.d. times to get n i.i.d. observations $Y_1, Y_2, \dots, Y_n$ from F. Next we estimate F by the *empirical distribution* $\widehat{F}_{\mathrm{SRS},n}$ with

$$\widehat{F}_{\mathrm{SRS},n}(y) = \frac{1}{n}\sum_{i=1}^{n} I(Y_i \le y), \tag{3}$$

which is the fraction of the n outputs that are less than or equal to y. We finally obtain the SRS estimator

$$\widehat{\xi}_{\mathrm{SRS},n} = \widehat{F}^{-1}_{\mathrm{SRS},n}(p) \tag{4}$$

of the p-quantile $\xi = F^{-1}(p)$. In the special case when Y has the form in (1), we generate $\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_n$ as n i.i.d. copies of $\mathbf{X} \sim G$, and set $Y_i = v(\mathbf{X}_i)$ for each $i = 1, 2, \dots, n$.

Given n observations $Y_1, Y_2, \dots, Y_n$, we can compute $\widehat{\xi}_{\mathrm{SRS},n}$ in (4) as follows. Let $Y_{1:n} \le Y_{2:n} \le \cdots \le Y_{n:n}$ be the *order statistics* (i.e., the sorted values) of the sample, so $Y_{k:n}$ is the kth smallest observation, which has *rank k*. We then have that

$$\widehat{\xi}_{\mathrm{SRS},n} = Y_{\lceil np \rceil:n}, \tag{5}$$

where $\lceil \cdot \rceil$ is the ceiling function.

One approach to compute $\widehat{\xi}_{\mathrm{SRS},n}$ through (5) is to *sort* the n observations $Y_1, Y_2, \dots, Y_n$ to obtain $\widehat{\xi}_{\mathrm{SRS},n}$ as the $\lceil np \rceil$-th smallest observation. Given the sample of size n, sorting can be accomplished in $\Theta(n \log n)$ worst-case running time as $n \to \infty$ using deterministic algorithms such as MERGESORT or HEAPSORT (e.g., Cormen et al. 2022 Section 2.3.1 and Chapter 6). Another approach is QUICKSORT (Cormen et al. 2022 Section 7.3), which has a $\Theta(n \log n)$ *expected* running time, but a $\Theta(n^2)$ *worst-case* running time.

Remark In computational complexity theory, there are equivalent ways (Cormen et al. 2022 Chapter 5) of analyzing the expected performance of an algorithm, such as QUICKSORT. One approach assumes that the input is n distinct deterministic elements, and the first step of *randomized* QUICKSORT applies a uniform random permutation of the data (each of the $n!$ permutations is equally likely); the complexity analysis then computes the expected number of steps to complete the task, where the expectation is taken with respect to the random permuation. Alternatively, since we assume i.i.d. data from a continuous distribution F, the n observations $Y_1, \dots, Y_n$ are distinct with probability 1, and each permutation of the data is equally likely, so we do not need to further permute the data to obtain the same expected behavior, where now the expectation is computed with respect to the joint distribution of the n i.i.d. observations.

3.1 Selection Algorithms to Compute $\widehat{\xi}_{\mathrm{SRS},n}$

If we are interested in estimating the p-quantile ξ_p for only a single value of p (or just a small number of them), then we can avoid sorting the sample and instead apply a *selection algorithm* (Cormen et al. 2022 Section 9.3). In particular, Algorithm 1 presents DETERMINISTICSELECT, which is taken from Algorithm 9.7 of Goodrich and Tamassia (2015) and originally developed by Blum et al. (1973). The deterministic algorithm applies a divide-and-conquer (or prune-and-search) approach based on the "median of medians". The goal of DETERMINISTICSELECT is to find the kth smallest element (i.e., the element of rank k) from an unordered sequence (or set) $\mathscr{S}$, and the algorithm accomplishes this by pruning away parts of the set that do not contain the desired element and recursing on what remains. The algorithm is guaranteed to prune a substantial proportion of elements in each recursive call through a clever choice of pivot m in lines 4–8, which divides the data into groups of 5, computes the median of each group, and then defines m as the median of the group medians. The pivot m guarantees pruning at least $3/10$ of the remaining elements, enough to ensure that the algorithm has $\Theta(n)$ worst-case running time (Goodrich and Tamassia 2015 Theorem 9.4, or Cormen et al. 2022 Theorem 9.3). Calling DETERMINISTICSELECT$(\{Y_1,Y_2,\ldots,Y_n\},\lceil np\rceil)$ then computes $\widehat{\xi}_{\mathrm{SRS},n}$ in (4) for any fixed $p\in(0,1)$ in $\Theta(n)$ worst-case running time.

While the $\Theta(n)$ worst-case running time of DETERMINISTICSELECT indicates that it can do better than $\Theta(n\log n)$ sorting algorithms as $n\to\infty$, the $\Theta(n)$ notation hides the leading constant c_0, which is large. Specifically, the complexity of the version by Blum et al. (1973) does not exceed $15n-163$ for $n>32$. Because the leading constant $c_0=15$ is quite large, DETERMINISTICSELECT may be outperformed by a sorting algorithm for practical sample sizes n. Alexandrescu (2017) considers variants of DETERMINISTICSELECT with better performance. Schönhage et al. (1976) give another worst-case $\Theta(n)$ deterministic selection approach with leading constant $c_0=3$, but that method is much more complicated.

Rather than employing a deterministic selection algorithm, the user can instead apply a *randomized* selection method (Cormen et al. 2022 Section 9.2). For example, Algorithm 2 presents QUICKSELECT, taken from Goodrich and Tamassia (2015), p. 271, and originally due to Hoare (1961), who calls it FIND and who also developed QUICKSORT. Recall that QUICKSORT works by choosing a pivot from among the current elements, splitting the current set of elements into two subsets—those less than or equal to the current pivot, and those greater than the pivot—and recursively calling QUICKSORT on each of those subsets. QUICKSELECT instead recurses on only one of those subsets, the one that contains the desired element, and this typically leads to QUICKSELECT being faster than QUICKSORT to find the kth order statistic. QUICKSELECT has $\Theta(n)$ *expected* running time (Cormen et al. 2022 Theorem 9.2), but a $\Theta(n^2)$ *worst-case* running time (Cormen et al. 2022 p. 231). We can compute $\widehat{\xi}_{\mathrm{SRS},n}$ by calling QUICKSELECT$(\{Y_1,Y_2,\ldots,Y_n\},\lceil np\rceil)$. Because we assume that $Y_1,Y_2,\ldots,Y_n$ are i.i.d. from a continuous distribution, each of the $n!$ permutation of the n observations is equally likely. In this case, line 4 in Algorithm 2 does not need to randomly choose the pivot, but rather any arbitrary element (e.g., the first or last) may be used as a pivot.

HEAPSELECT in Algorithm 3 is a sequential selection method for finding the kth smallest element from a collection of n comparable elements $Y_1,Y_2,\ldots,Y_n$ when the elements are generated one at a time. A slight modification of an approach (Hettinger 2011) based on the function `heapq.nlargest` in the python library, the method stores the $s=n-k+1$ largest elements generated so far in a priority queue PQ implemented as a binary *min-heap* (Goodrich and Tamassia 2015 Section 5.3) of size s. A min-heap is a data structure that can be viewed as a complete binary tree (i.e., all levels are full except for possibly the last, which is filled from left to right) in which the value of the element at a node is less than or equal to the value of each of its children, so the root contains the minimum value. We can implement a min-heap using an array (with starting index 1) in which each entry i has children at entries $2i$ and $2i+1$ (assuming the children exist). HEAPSELECT employs the operations PQ.insert, PQ.min, and PQ.replaceMin. This last operation replaces the minimum (at the root) with a new element and then reheapifies (sifts or bubbles down) to maintain the min-heap properties. This entails following a single path from the root to a leaf, always choosing to move towards the smaller child (ties broken arbitrarily). If the current node is larger than at least one of its children, we swap the current node with the smaller child, recursively repeating the process on the smaller child, and stopping once both children are larger than the current node or reaching the path's leaf. Since the height of a tree with s nodes is about $\lg(s)$, where $\lg(\cdot)$ denotes log with base 2, sifting down travels down a path of at most roughly $\lg(s)$ nodes. PQ.insert operates similarly by adding the new element to the first open spot in the last level of the tree and sifting up (rather than sifting down) to maintain the min-heap property. PQ.min simply returns the value of the root (the minimum) without removing it, so this takes constant time (in an array-implementation of the min-heap).

Algorithm 1 DETERMINISTICSELECT($\mathscr{S}, k$): A deterministic selection algorithm

Input: Sequence $\mathscr{S}$ of n comparable elements, and an integer $k \in [1, n]$

Output: The kth smallest element of $\mathscr{S}$

```
1: if $n = 1$ then
2:     return the (first) element of $\mathscr{S}$
3: end if
4: Divide $\mathscr{S}$ into $g = \lceil n/5 \rceil$ groups, $\mathscr{S}_1, \ldots, \mathscr{S}_g$, such that each of groups $\mathscr{S}_1, \ldots, \mathscr{S}_{g-1}$ has exactly 5
   elements, and group $\mathscr{S}_g$ has at most 5 elements
5: for $i \leftarrow 1$ to $g$ do
6:     Find the baby median $m_i$ in $\mathscr{S}_i$ (using any method)
7: end for
8: $m \leftarrow$ DETERMINISTICSELECT($\{m_1, \ldots, m_g\}, \lceil g/2 \rceil$)
9:  remove all the elements from $\mathscr{S}$ and put them into three sequences:
       •$\mathscr{L}$, storing the elements in $\mathscr{S}$ less than $m$
       •$\mathscr{E}$, storing the elements in $\mathscr{S}$ equal to $m$
       •$\mathscr{G}$, storing the elements in $\mathscr{S}$ greater than $m$
10: if $k \le |\mathscr{L}|$ then          ▷ $m$ is too large to be the desired element, so recurse on $\mathscr{L}$
11:     DETERMINISTICSELECT($\mathscr{L}, k$)
12: else if $k \le |\mathscr{L}| + |\mathscr{E}|$ then
13:     return $m$                          ▷ each element in $\mathscr{E}$ is equal to $m$
14: else                                    ▷ $m$ is too small to be the desired element, so recurse on $\mathscr{G}$
15:     DETERMINISTICSELECT($\mathscr{G}, k - |\mathscr{L}| - |\mathscr{E}|$)
16: end if
```

Algorithm 2 QUICKSELECT($\mathscr{S}, k$): A randomized selection algorithm

Input: Sequence $\mathscr{S}$ of n comparable elements, and an integer $k \in [1, n]$

Output: The kth smallest element of $\mathscr{S}$

```
1: if $n = 1$ then
2:     return the (first) element of $\mathscr{S}$
3: end if
4: Pick a random element $x \in \mathscr{S}$ as a pivot
5: Remove all the elements from $\mathscr{S}$ and put them into three sequences:
       •$\mathscr{L}$, storing the elements in $\mathscr{S}$ less than $x$
       •$\mathscr{E}$, storing the elements in $\mathscr{S}$ equal to $x$
       •$\mathscr{G}$, storing the elements in $\mathscr{S}$ greater than $x$
6: if $k \le |\mathscr{L}|$ then           ▷ $x$ is too large to be the desired element, so recurse on $\mathscr{L}$
7:     QUICKSELECT($\mathscr{L}, k$)
8: else if $k \le |\mathscr{L}| + |\mathscr{E}|$ then
9:     return $x$                           ▷ each element in $\mathscr{E}$ is equal to $x$
10: else                                    ▷ $x$ is too small to be the desired element, so recurse on $\mathscr{G}$
11:     QUICKSELECT($\mathscr{G}, k - |\mathscr{L}| - |\mathscr{E}|$)
12: end if
```

While Algorithm 3 can work for any k and n, it is most efficient when k is close to n (i.e., when computing the p-quantile estimator for p close to 1, which corresponds to the rank $k = \lceil np \rceil$ order statistic). For example, the size $s = n - k + 1$ of the data structure is then substantially smaller than n; Section 3.2 provides an approximate analysis to obtain a rough upper bound of the average complexity of HEAPSELECT. While Algorithm 3 is presented with each element Y_i being generated one at a time, the approach can also be applied when all n observations have already been generated, and then just examined one at a time, but then it does not have the benefit of reduced memory required. (When k/n is close to 0, we can modify Algorithm 3 to be efficient by replacing the min-heap with a max-heap of size k, changing line 8 to check if $Y_i <$ PQ.max(), using PQ.extractMax() in line 9, and returning PQ.max() in line 12.)

Algorithm 3 HEAPSELECT(n,k): A sequential selection algorithm using a priority queue implemented as a min-heap

Input: Sample size n of n comparable elements, and an integer $k \in [1,n]$
Output: The kth smallest element from the sequence $Y_1, Y_2, \ldots, Y_n$ generated one at a time

```
1: Let $s = n - k + 1$
2: for $i \leftarrow 1$ to $s$ do
3:     Generate $Y_i$
4:     PQ.insert($Y_i$)
5: end for
6: for $i \leftarrow s+1$ to $n$ do
7:     Generate $Y_i$
8:     if $Y_i >$ PQ.min() then
9:         PQ.replaceMin($Y_i$)
10:    end if
11: end for
12: return PQ.min()
```

Some other randomized algorithms are not guaranteed to find the kth order statistic from a set of n observations, but instead provide a high-probability guarantee of succeeding. For example, Section 3.3 of Motwani and Raghavan (1995) presents a randomized algorithm that randomly picks (i.i.d. with replacement) a subset of size $n^{3/4}$ from the original set, and from the subset, identifies two values a and b ($a < b$) that are likely to enclose the true kth order statistic of the original set. When this is successful, the algorithm then can identify the desired kth order statistic of the original set by sorting a constructed set of size $O(n^{3/4})$ (determined by a or b) of the original elements, and the sorting takes $O(n^{3/4}\log(n^{3/4})) = O(n)$. But the approach has a $O(n^{-1/4})$ probability of failing to find the desired order statistic in any single attempt, so the procedure keeps repeating independently until it is successful (geometric trials), leading to the expected running time being linear in n. We do not further consider this algorithm here.

3.2 An Approximate Analysis of HEAPSELECT and Comparison with QUICKSELECT

We now want to provide an approximate analysis of the behavior of HEAPSELECT and contrast it with the known expected behavior of QUICKSELECT. In complexity theory, analyzing an algorithm's running time requires adopting a particular computational model (Cormen et al. 2022 Section 2.2). One approach is the *random-access machine* (RAM) model, which considers the number of basic operations (e.g., add, multiply, store, conditional statements, subroutine calls) used by an algorithm. But the RAM model does not account for some features in actual computers, such as memory hierarchy (e.g., caches and virtual memory), which may significantly affect actual performance on a real computer. An alternative analysis employs the *comparison-based model*, which simply counts only the number of comparisons made ($<$, $\leq$, $>$, $\geq$) (Cormen et al. 2022 Chapter 8).

Let $C^{\mathrm{QS}}_{n,k}$ be the (random) total number of comparisons used by QUICKSELECT on an input sequence of n elements to identify the kth smallest element. Then modifying a result from Knuth (1972) for selecting the t-th largest element to instead find the k-th smallest for $k = n-t+1$ leads to

$$\mathbb{E}[C^{\mathrm{QS}}_{n,k}] = 2\left[(n+1)H_n - (k+2)H_k - (n-k+3)H_{n-k+1} + n + 3\right], \tag{6}$$

where $H_m = \sum_{i=1}^{m} 1/i$ is the mth harmonic number. As $n \to \infty$, using that $H_n/\ln(n) \to 1$, where $\ln(\cdot)$ denotes log base-e, and taking $k = \lceil np \rceil$ for fixed $p \in (0,1)$, we can show (Grübel and Rösler 1996) that

$$\lim_{n\to\infty} \frac{1}{n}\mathbb{E}[C^{\mathrm{QS}}_{n,\lceil np\rceil}] = 2 - 2p\ln p - 2(1-p)\ln(1-p), \tag{7}$$

so that $\mathbb{E}[C^{\mathrm{QS}}_{n,\lceil np\rceil}] = \Theta(n)$ for each fixed $p \in (0,1)$.

Let $C^{\mathrm{HS}}_{n,k}$ be the number of comparisons used by HEAPSELECT(n,k) in Algorithm 3 when the sequence $Y_1, Y_2, \ldots, Y_n$ is i.i.d. from a continuous distribution, so that all elements are distinct with probability 1. We next carry out a rough analysis to obtain an approximate upper bound for $\mathbb{E}[C^{\mathrm{HS}}_{n,k}]$.

- Lines 2–5 of Algorithm 3 build an initial min-heap of size $s = n-k+1$ from the first s elements, Y_i, $i = 1,2,\ldots,s$. The expected number of comparisons to do this is at most $2s$ (Doberkat 1984). This upper bound can be improved somewhat (Carlsson and Chen 1995), but we use $2s$ for simplicity.
- Next we analyze lines 6–11 of Algorithm 3, which is a loop to go through the remaining $n-s$ elements, Y_i, $i = s+1, s+2, \ldots, n$, sequentially checking if each Y_i should replace the current minimum in the min-heap (line 8). In each iteration of the loop, this is done by comparing Y_i to the current minimum in the min-heap, where returning the minimum requires no comparisons because it is always at the top of the min-heap. For $i > s$, let $J_{i,s} = I(Y_i$ is among the s largest of the first i elements$)$ be the indicator function of a required replacement, which line 9 performs by replacing the current minimum with Y_i. Because $E[J_{i,s}] = s/i$, the expected total number of replacements from the loop is $\sum_{i=s+1}^{n} s/i = s(H_n - H_s)$. Each replacement takes at most roughly $2\lg(s)$ comparisons: each step of sifting down entails 2 comparisons (comparing the current node with the minimum of its two children), and the number of steps down the path from root to leaf until finding the appropriate location is at most the tree's height, which is about $\lg(s)$. Thus, the expected number of comparisons for lines 6–11 of Algorithm 3 is at most about $n - s + 2s\lg(s)(H_n - H_s)$. Combining this with the upper bound $2s$ for building the initial min-heap from the item above and substituting $s = n-k+1$ leads to an approximate upper bound for $\mathbb{E}[C^{\mathrm{HS}}_{n,k}]$ as

$$v^{\mathrm{HS}}_{n,k} \equiv 2n - k + 1 + 2(n-k+1)\lg(n-k+1)(H_n - H_{n-k+1}). \tag{8}$$

For large n and fixed $p \in (0,1)$, using that $H_n \approx \ln(n)$ and again writing $s = n-k+1$ for $k = \lceil np \rceil$ show that $v^{\mathrm{HS}}_{n,k} \approx n + s + 2s\lg(s)[\ln(n) - \ln(s)] = n + s + 2s\lg(s)\ln(n/s)$. Thus, taking $s \approx (1-p)n$ yields

$$\frac{1}{n}v^{\mathrm{HS}}_{n,k} \approx 2 - p - 2(1-p)[\lg(1-p) + \lg(n)]\ln(1-p). \tag{9}$$

Equating the right side of (9) to (7) and solving for n leads to

$$n = 2^{a_p} \equiv n_p, \qquad \text{where} \qquad a_p = 1 - \lg(1-p) - \frac{p - 2p\ln(p)}{2(1-p)\ln(1-p)}. \tag{10}$$

For fixed $p \in (0,1)$, (10) suggests that when $n \le n_p$ (resp., $n > n_p$), the expected number of comparisons for HEAPSELECT should be roughly no greater (resp., greater) than that for QUICKSELECT.

Table 1 gives the value of n_p for various p approaching 1. As p increases, n_p grows, with the crossover point n_p for $p = 0.99$ and 0.999 exceeding most sample sizes n used in practice. The last column of Table 1

Table 1: For $p \in (0,1)$, n_p from (10) is approximately the threshold such that a sample size $n < n_p$ leads to HEAPSELECT outperforming QUICKSELECT. Also, the same conclusion is roughly true when $r_{n,p} < 1$, where $r_{n,p} = v^{\mathrm{HS}}_{n,\lceil np \rceil} / \mathbb{E}[C^{\mathrm{QS}}_{n,\lceil np \rceil}]$, which is given for $n = 10^5$.

p	n_p	$r_{10^5,p}$
0.90	103.1	2.72
0.95	451.5	1.97
0.97	1978.5	1.53
0.99	4.00e+05	0.91
0.999	1.29e+25	0.54

presents $r_{n,p} = v^{\mathrm{HS}}_{n,\lceil np \rceil} / \mathbb{E}[C^{\mathrm{QS}}_{n,\lceil np \rceil}]$ for the same values of p for fixed $n = 10^5$. Thus, $r_{n,p} \le 1$ (resp., > 1) suggests that the expected number of comparisons for HEAPSELECT should be roughly no greater (resp., greater) than that for QUICKSELECT. These results hint that HEAPSELECT will mainly be beneficial when $p \approx 1$ but not for $p \ll 1$, which is in line with our numerical experiments (Section 5).

4 COMPUTING AN SRS ESTIMATE OF CVAR

A plug-in estimator for the CVaR γ in (2) replaces F with $\widehat{F}_{\mathrm{SRS},n}$ from (3) to get

$$\widehat{\gamma}_{\mathrm{SRS},n} = \frac{1}{1-p} \int_p^1 \widehat{F}^{-1}_{\mathrm{SRS},n}(u)\,\mathrm{d}u = \frac{1}{n(1-p)} \sum_{k=\lceil np \rceil + 1}^{n} Y_{k:n} = \frac{1}{n(1-p)} \sum_{i=1}^{n} Y_i I(Y_i > \widehat{\xi}_{\mathrm{SRS},n}), \tag{11}$$

where $\widehat{\xi}_{\mathrm{SRS},n}$ is the SRS p-quantile estimator in (4); e.g., see McNeil et al. (2015) Section 9.2.6, and references therein, some of which consider slight variations of $\widehat{\gamma}_{\mathrm{SRS},n}$. One way of computing $\widehat{\gamma}_{\mathrm{SRS},n}$ based on the penultimate representation in (11) first sorts the sample to obtain the order statistics $Y_{k:n}$, $k = 1,2,\ldots,n$, and then averages the last $n - \lceil np \rceil$ of them. When performing the sort via HEAPSORT or MERGESORT, this approach takes $\Theta(n \log n)$ worst-case running time as $n \to \infty$ for any fixed $p \in (0,1)$. If QUICKSORT is applied for sorting, then the approach has expected runtime that is $\Theta(n \log n)$ but with worst-case $\Theta(n^2)$.

An alternative approach for computing $\widehat{\gamma}_{\mathrm{SRS},n}$ uses a two-pass approach. In the first pass we use DETERMINISTICSELECT from Algorithm 1 to get $\widehat{\xi}_{\mathrm{SRS},n}$, and the second pass computes the sum in the last representation in (11) to obtain $\widehat{\gamma}_{\mathrm{SRS},n}$. Because each pass takes $\Theta(n)$ time, the overall algorithm has worst-case $\Theta(n)$ complexity. We call this approach DETERMINISTICCVAR.

We can also apply QUICKSELECT in Algorithm 2 or HEAPSELECT in Algorithm 3 to compute the CVaR estimator $\widehat{\gamma}_{\mathrm{SRS},n}$ in (11). To simplify the discussion, we will describe these methods when $Y_1, Y_2, \ldots, Y_n$ are distinct, as is the case when they are i.i.d. from a continuous distribution F, as we have assumed.

Computing $\widehat{\gamma}_{\mathrm{SRS},n}$ can be accomplished by calling QUICKSELECT$(\mathscr{S}, k)$ in Algorithm 2 with $\mathscr{S} = \{Y_1, Y_2, \ldots, Y_n\}$ and $k = \lceil np \rceil$ with the following modifications. Whenever the condition in line 6 is true, the pivot x is too large to be $\widehat{\xi}_{\mathrm{SRS},n}$, so line 7 recurses on $\mathscr{L}$, which contains the items less than x, and the elements in $\mathscr{E} \cup \mathscr{L}$ are discarded. But those pruned elements are all greater than $\widehat{\xi}_{\mathrm{SRS},n}$, so they contribute to $\sum_{i=1}^{n} Y_i I(Y_i > \widehat{\xi}_{\mathrm{SRS},n})$ in (11). Thus, we introduce a global variable T, initialized as $T = 0$ before first calling QUICKSELECT, and immediately before line 7, update T by adding in all values from $\mathscr{E} \cup \mathscr{L}$ before they are pruned. Finally, just before line 8, also compute $\widehat{\gamma}_{\mathrm{SRS},n} = T / [n(1-p)]$. Since these modifications do not involve any additional comparisons, the expected number of comparisons remains as in (6). We call this approach QUICKCVAR.

We can compute $\widehat{\gamma}_{\mathrm{SRS},n}$ in (11) by calling a slightly modified HEAPSELECT(n,k) in Algorithm 3 with $k = \lceil np \rceil$. Recall that upon completion, PQ contains all of the s largest elements and the smallest of those is $\widehat{\xi}_{\mathrm{SRS},n}$, which is stored at root of the min-heap. Thus, after completing the loop in lines 6–11, we can compute $\sum_{i=1}^{n} Y_i I(Y_i > \widehat{\xi}_{\mathrm{SRS},n})$ as just the sum of the items in the min-heap except for the root. We call this approach HEAPCVAR, which uses the same number of comparisons as HEAPSELECT, and (8) gives an approximate upper bound for the expected number of comparisons.

5 NUMERICAL RESULTS

We ran numerical experiments using outputs from a *stochastic activity network* (SAN), which is a model satisfying (1). A project manager will often build a SAN to analyze the time to complete a project consisting of d activities with precedence relations (some activities must be completed before others can begin). The SAN is modeled as a directed acyclic graph having d edges, with the length X_j of edge j being the (random) time to complete activity j. All activities with edges into a particular node must be completed before starting any of the activities with edges leaving the node. A single source node denotes the start of the project, and a single sink corresponds to the project completing. We are interested in the (random) time Y to complete the project, which is the longest path from source to sink. Our study employed a small SAN from Hsu and Nelson (1990) with $d = 5$ activities, where the activity durations are i.i.d. exponential with mean 1, and the output is $Y = \max(X_1 + X_2, X_1 + X_3 + X_5, X_4 + X_5) = v(\mathbf{X})$ for $\mathbf{X} = (X_1, \ldots, X_5)$.

We implemented various methods from Sections 3 and 4 using C++ and python to compare their CPU times. We first pre-generated 10^9 i.i.d. observations of Y, and stored them in a file. Our codes then read this file to carry out timing experiments of the methods to compute estimators of the p-quantile and CVaR, given the data, for different values of $p \in (0,1)$ and the sample size n. We repeated this for $R = 10^3$ independent replications, and recorded the average and maximum CPU times across the R replications for each (p,n) pair. We report here only the average CPU times as the maximum times are similar except for a bit worse performance for methods whose worst-case complexities differ from their average complexities; e.g., QUICKSELECT has $\Theta(n)$ expected running time but $\Theta(n^2)$ worst-case time. Even though linear and quadratic behaviors can differ substantially when n is large, our experiments computed the worst case from only R i.i.d. replications, which is unlikely to capture the true worst case over all possible $n!$ permutations of the input of n elements, so the differences are not very large as $R \ll n!$. Moreover, when applying QUICKSELECT with n elements, the probability that the number of comparisons exceeds zn decreases exponentially fast in z (Grübel and Rösler 1996; Devroye 2001), so its behavior is usually linear in n.

For computing the p-quantile, we compared the methods DETERMINISTICSELECT (denoted as DSe) using a slight modification of Algorithm 1 described below; QUICKSELECT (QSe; Algorithm 2); HEAPSELECT (HSe; Algorithm 3); MERGESORT (MSo); HEAPSORT (Hso) and QUICKSORT (QSo). We altered Algorithm 1 so that it stops recursing when the number of elements is no more than a threshold $t_0 = 5$, at which point it switches to sorting. We also considered one more procedure, INTROSELECT (ISe), which starts off using QUICKSELECT before switching to another method. For INTROSELECT, our codes used library routines: `nth_element` for C++, and `numpy.partition` for python, which seem to be standard methods in these languages. For python, the library function `numpy.quantile` employs `numpy.partition` when the number of requested quantile levels p is small; although there is no current existing function named "quantile" in the C++ library, `nth_element` seems to be commonly used in practice. Alexandrescu (2017) studies the theoretical and empirical performance of several similar variants.

Figure 1 gives the average CPU times with C++ to compute the p-quantile, given the data. Except for DETERMINISTICSELECT, the performance of selection algorithms is at least an order of magnitude better than the sorting methods. Compared to all other methods, the behavior of HEAPSELECT is more sensitive to changes in the quantile level p, which determines the heap's size $s = n - k - 1$ since $k = \lceil np \rceil$. When $p \approx 1$, the small heap size $s \ll n$ makes HEAPSELECT an attractive choice against other methods. HEAPSELECT outperforms QUICKSELECT on extreme (large) values of p with a smaller sample size n, but the former's behavior degrades as n grows for fixed p. For example, at $p = 0.99$, HEAPSELECT beats QUICKSELECT for sample size $n = 100$, but QUICKSELECT does better for $n \geq 1000$. At $p = 0.999$, HEAPSELECT always performs better. These findings basically agree with the trends in Table 1 (based on approximate analyses), although they differ in the specific values of n_p where the crossovers occur.

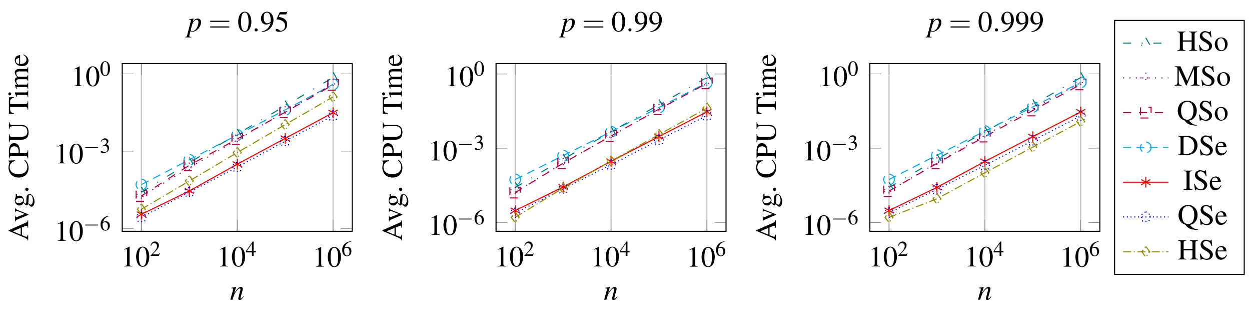

Figure 1: Average CPU times using C++ to compute the p-quantile estimator from $R = 10^3$ replications.

DETERMINISTICSELECT beats the sorting methods MERGESORT and HEAPSORT with a large enough sample size n, but never outpeforms QUICKSORT for the sample sizes we tested. (For the worst-case CPU times, which are not shown in the paper, DETERMINISTICSELECT outperforms the two sorting algorithms earlier as n increases.)

INTROSELECT (C++ library function `nth_element`), a version of QUICKSELECT that eventually switches to SELECTIONSORT (Goodrich and Tamassia 2015 Section 5.2.1), does slightly better than pure QUICKSELECT for small n for the worst-case CPU times (not shown), but the effect fades quickly as n grows, and eventually underperforms the latter.

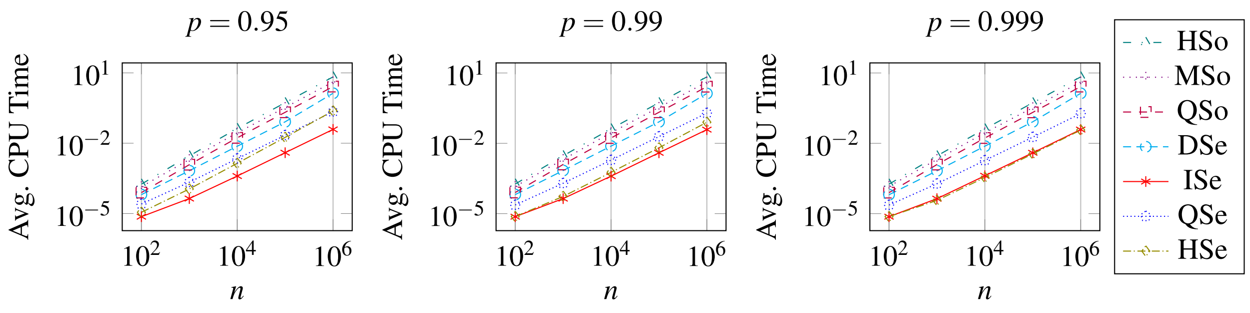

Figure 2: Average CPU times using python to compute the p-quantile estimator from $R = 10^3$ replications.

Figure 2 presents the average CPU times for python. The experimental results demonstrate clear trends in the performance of quantile algorithms across different sample sizes and quantile levels. While possessing a theoretical $\Theta(n)$ worst-case behavior, DETERMINISTICSELECT has a large leading constant coefficient to its linear term (see Section 3.1), making it slower than INTROSELECT and QUICKSELECT. However, DETERMINISTICSELECT always outperforms all sorting algorithms. This last behavior for python differs from that with C++, which may be due to differences in implementation choices, languages, etc.

INTROSELECT for python (`numpy.partition`) consistently exhibits the best performance (or very close to it) among all methods, benefiting from its hybrid approach (QUICKSELECT falling back to median-of-medians), with runtimes an order of magnitude lower than DETERMINISTICSELECT. A possible reason for why INTROSELECT performs so well is it is implemented using NumPy, which is compiled in C and optimized for parallel processing. In contrast, regular python code is usually interpreted at runtime (typically slower than executing compiled code) and does not exploit parallelism automatically. Highlighting the overhead of deterministic pivoting, QUICKSELECT is faster than DETERMINISTICSELECT, but slower than INTROSELECT, As with our C++ implement, the performance of the python code for HEAPSELECT is sensitive to the value of p.

Across all selection algorithms in python, runtime scales roughly linearly with n, but sorting-based approaches exhibit slightly super-linear growth, in agreement with their theoretical worse-than-linear complexity, with QUICKSORT being the fastest among them. The results suggest that in python, INTROSELECT (implemented with `numpy.partition`) is the most efficient for general use, but HEAPSELECT is sometimes slightly better for extreme quantiles ($p \approx 1$).

We also ran similar numerical experiments with C++ to compute the CVaR estimator (Section 4), but omit those plots. We used DETERMINISTICCVAR (DCV); QUICKCVAR (QCV); HEAPCVAR (HCV); and QUICKSORT (QSo), where we sum the largest $n-k$ items of the sorted list to compute the CVaR estimator in (11). The results that simultaneously compute quantile and CVaR values are consistent with the results only computing the quantile value.

6 CONCLUDING REMARKS

We studied the efficiencies of computing SRS estimators of a p-quantile and CVaR given n SRS-generated outputs. While some papers have suggested computing these estimators by first sorting the data, a selection method can accomplish this more quickly. While our paper considers the transient simulation problem, in which a simulation model is run n independent times to obtain n i.i.d. outputs, the methods considered can also apply to dependent data, as can arise in a steady-state simulation (Alexopoulos et al. 2019) or when employing Latin hypercube sampling (Dong and Nakayama 2017). We are currently examining algorithms for computing estimators of risk measures when applying variance-reduction techniques, such as importance sampling. In addition to a quantile and CVaR, we are also investigating fast methods for computing other quantile-based measures, such as distortion risk measures (Dhaene et al. 2012). Moreover, we are interested in efficient techniques for constructing confidence intervals for the risk measures.

While our paper has focused primarily on time efficiency of computing a risk measure, space (including memory and storage) usage can also be of concern when n is enormous (Munro and Paterson 1980). Such situations arise, e.g., in steady-state simulations (Alexopoulos et al. 2019), where dependence necessitates taking very large n, and for online/streaming data (Ghosh and Pasupathy 2013). In such cases, rather than trying to compute the exact p-quantile estimator, the user may be satisfied with a q-quantile estimator for $q \in (p-\varepsilon, p+\varepsilon)$ or $q \in (p(1-\varepsilon), p(1+\varepsilon))$ for a small given $\varepsilon > 0$. This is known as a *quantile sketch* (Chen and Guestrin 2016; Masson et al. 2019; Guptal et al. 2024). We plan to also investigate such approaches in a simulation context. Another topic for future studies is applying parallel processing for selection (Valiant 1983) and computing risk measures in a simulation setting.

ACKNOWLEDGMENTS

This work has been supported in part by the National Science Foundation under Grant No. CMMI-2345330. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Alexandrescu, A. 2017. "Fast Deterministic Selection". In *16th International Symposium on Experimental Algorithms (SEA 2017) Proceedings*, edited by C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, and R. Raman, Volume 75, 24:1–24:19: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Alexopoulos, C., D. Goldsman, A. C. Mokashi, K.-W. Tien, and J. R. Wilson. 2019. "Sequest: A Sequential Procedure for Estimating Quantiles in Steady-State Simulations". *Operations Research* 67(4):1162–1183.

Asmussen, S., and P. Glynn. 2007. *Stochastic Simulation: Algorithms and Analysis*. New York: Springer.

Blum, M., R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. 1973. "Time Bounds for Selection". *Journal of Computer and System Sciences* 7(4):448–461.

Carlsson, S., and J. Chen. 1995. "Heap Construction: Optimal in Both Worst and Average Cases?". In *Algorithms and Computation, 6th International Symposium, ISAAC '95*, edited by J. Staples, P. Eades, N. Katoh, and A. Moffat, Volume 1004 of *Lecture Notes in Computer Science*, 254–263. Berlin: Springer.

Chen, T., and C. Guestrin. 2016. "XGBoost: A Scalable Tree Boosting System". In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, 785–794: ACM.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein. 2022. *Introduction to Algorithms*. 4th ed. MIT Press.

Devroye, L. 2001. "On the Probabilistic Worst-Case Time of 'Find'". *Algorithmica* 31(3):291–303.

Dhaene, J., A. Kukush, D. Linders, and Q. Tang. 2012. "Remarks on Quantiles and Distortion Risk Measures". *European Actuarial Journal* 2(2):319–328.

Doberkat, E. E. 1984. "An Average Case Analysis of Floyd's Algorithm to Construct Heaps". *Information and Control* 61(2):114–131.

Dong, H., and M. K. Nakayama. 2017. "Quantile Estimation With Latin Hypercube Sampling". *Operations Research* 65(6):1678–1695.

Dong, H., and M. K. Nakayama. 2019. "A Tutorial on Quantile Estimation via Monte Carlo". In *Proceedings of the 13th International Conference in Monte Carlo & Quasi-Monte Carlo Methods in Scientific Computing*, edited by P. L'Ecuyer and B. Tuffin. Accepted.

Ghosh, S., and R. Pasupathy. 2013. "Low-Storage Online Estimators for Quantiles and Densities". In *2013 Winter Simulations Conference (WSC)* https://doi.org/10.1109/wsc.2013.6721470.

Goodrich, M. T., and R. Tamassia. 2015. *Algorithm Design and Applications*. Hoboken, N.J: Wiley.

Grübel, R., and U. Rösler. 1996. "Asymptotic Distribution Theory for Hoare's Selection Algorithm". *Advances in Applied Probability* 28(1):252–269.

Guptal, M., M. Singhal, and H. Wu. 2024. "Optimal Quantile Estimation: Beyond the Comparison Model". In *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*, 1137–1158. Piscataway, NJ: IEEE.

Hettinger, R. 2011. "Compare Algorithms for `heapq.smallest` (Python recipe)". https://code.activestate.com/recipes/577573-compare-algorithms-for-heapqsmallest/. accessed 27th March 2025.

Hoare, C. A. R. 1961. "Algorithm 65: Find". *Communications of the ACM* 4(7):321–322.

Hsu, J. C., and B. L. Nelson. 1990. "Control Variates for Quantile Estimation". *Management Science* 36(7):835–851.

Keslin, G., B. L. Nelson, B. Pagnoncelli, M. Plumlee, and H. Rahimian. 2024. "Ranking and Contextual Selection". *Operations Research*.

Knuth, D. E. 1972. "Mathematical Analysis of Algorithms". In *Information Processing 71: Proceedings of IFIP Congress 1971*, edited by C. V. Freiman, J. E. Griffith, and J. L. Rosenfeld, 19–27. Amsterdam: International Federation for Information Processing: North-Holland.

Masson, C., J. E. Rim., and H. K. Lee. 2019. "DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees". *Proceedings of the VLDB Endowment* 12(12):2195–2205.

McNeil, A. J., R. Frey, and P. Embrechts. 2015. *Quantitative Risk Management: Concepts, Techniques, Tools*. Revised ed. Princeton, New Jersey: Princeton University Press.

Motwani, R., and P. Raghavan. 1995. *Randomized Algorithms*. Cambridge: Cambridge University Press.

Munro, J. I., and M. S. Paterson. 1980. "Selection and Sorting with Limited Storage". *Theoretical Computer Science* 12(3):315–323.

National Academies of Sciences, Engineering, and Medicine 2024. *Foundational Research Gaps and Future Directions for Digital Twins*. 1st ed. Washington, D.C.: National Academies Press.

Schönhage, A., M. Paterson, and N. Pippenger. 1976. "Finding the Median". *Journal of Computer and System Sciences* 13(2):184–199.

Valiant, L. G. 1983. "Parallelism in Comparison Problems". *SIAM Journal on Computing* 4(2):348–355.

AUTHOR BIOGRAPHIES

SHA (HOLLY) CAO is a Ph.D. student of Computer Science as the New Jersey Institute of Technology. She received an undergraduate degree in Computer Science at New York University and master's degree in Financial Mathematics at University of Chicago. Her email address is sc2772@njit.edu.

TRUONG DANG is an undergraduate student of Computer Science at the New Jersey Institute of Technology. His email address is tdd4@njit.edu.

JAMES M. CALVIN is a professor in the Department of Computer Science at the New Jersey Institute of Technology. He received a Ph.D. in operations research from Stanford University. Besides simulation output analysis, his research interests include global optimization and probabilistic analysis of algorithms. His email address is calvin@njit.edu.

MARVIN K. NAKAYAMA is a professor in the Department of Computer Science at the New Jersey Institute of Technology. He received a Ph.D. in operations research from Stanford University, and is an associate editor for *ACM Transactions on Modeling and Computer Simulation*. His email: marvin@njit.edu.