Streaming Algorithms for Selection and Approximate Sorting

Richard M. Karp

International Computer Science Institute, Berkeley, USA
and University of California at Berkeley
karp@icsi.berkeley.edu

1 Introduction

Companies such as Google, Yahoo and Microsoft maintain extremely large data repositories within which searches are frequently conducted. In an article entitled "Data-Intensive Supercomputing: The case for DISC" Randal Bryant describes such data repositories and suggests an agenda for appying them more broadly to massive data set problems of importance to the scientific community and society in general.

Large-scale data repositories have become feasible because of the low cost of disc storage. For \$10,000 one can buy a processor with 10^{12} bytes of disc storage, divided into blocks of capacity $64,000$ bytes. A typical repository (far from the largest) might contain 1000 processors, each with 10^{12} bytes of storage.

It is of interest to develop streaming algorithms for basic information processing tasks within such data repositories. In this paper we present such algorithms for selecting the elements of given ranks in a totally ordered set of n elements and for a related problem of approximate sorting. We derive bounds on the storage and time requirements of our algorithms.

Such data repositories support random access to the disc blocks. Therefore, it is reasonable to assume that the stream of input data to our sorting and selection algorithms is a random permutation of the disc blocks.

We also consider parallel algorithms in which the data arrives in several independent streams, each arriving at a single processor. Since all the processors of such a repository are co-located, we assume that interprocessor communication is not a bottleneck.

2 Streaming Algorithms

The input to a streaming algorithm is a sequence of items that arrive over time. The output of the streaming algorithm on a given sequence is specified by a function from sequences into some range. The algorithm processes each item in turn and produces an output after the last arrival. The streaming algorithm may be of three types:

1. In a basic streaming algorithm the length of the input is specified in advance.
2. In an *anytime streaming algorithm* the input may end at any time, but an upper bound on the length of the input is given.
3. In an *everytime streaming algorithm* an upper bound on the length of the input is given, and the algorithm is required to produce a correct output for every prefix of the input.

The working storage of a streaming algorithm is a buffer of limited capacity. We are interested in the following measures of complexity: the capacity of the buffer and the time, or amortized time, to process an item.

In our case the items are keys drawn from a totally ordered set. We assume that the keys arrive in a random order, and the algorithm is required to be correct with high probability. If, more realistically, we assumed that the input consists of blocks of N keys, where the allocation of keys to blocks is arbitrary but the blocks arrive in a random order, then our results would still hold, except that the storage requirement would be multiplied by N.

We restrict attention to deterministic or randomized algorithms that gain information about the arriving keys solely by performing comparisons, and we measure time complexity by the number of comparisons.

We often make statements of the form "The algorithm is correct with high probability when provided with $O(f(n))$ units of the computational resource. (such as storage or time)." The precise meaning of such a statement is: "For every $\delta > 0$ there exist constants c and n_0 such that, for all $n > n_0$, the algorithm is correct with probability $\geq 1 - \delta$ when provided with $cf(n)$ units of the computational resource.." An algorithm is *optimal within a factor c* if, for n sufficiently large, its resource requirement is within a factor c of a lower bound that holds for every algorithm for the problem.

3 Results

The *α-quantile* of a totally ordered set of n keys is the $\lfloor \alpha n \rfloor$th smallest element. We present optimal algorithms (simultaneously for time and storage), under the random arrivals assumption, for the following problems:

1. **Selection:** Compute an α-quantile for a given α.
2. **Multiple selection:** Compute α-quantiles for many given values of α.
3. **Parallel selection:** In which the input is divided into streams, each with its own buffer, and the different streams communicate by message passing.
4. **Approximate selection:** Given α and ϵ, find a key whose rank differs from αn by at most ϵn.
5. **Approximate sorting:** Given a small positive constant ϵ, compute an ordering of the keys in which the rank assigned to each key agrees with its rank in the true ordering, within a relative error of ϵ.

The algorithm for selection is an everytime algorithm. The algorithms for multiple selection and parallel selection are anytime algorithms. The algorithm for approximate sorting requires two passes over the data.

Finally, as a byproduct of our analysis of approximate sorting, we give an elegant method for computing the expected number of comparisons for Quicksort, Quickselect and Multiple Quickselect (see [6]).

There is a large literature on streaming algorithms for sorting and selection. Our work differs from most of this literature because of the random arrivals assumption, and because we simultaneously optimize both storage and time, whereas most of the work on streaming algorithms considers only storage.

4 Selection

4.1 Previous Work on Randomized Algorithms for Selection

Among its many interesting results, the seminal paper of Munro and Paterson [5] presents a streaming algorithm with optimal storage $O(\sqrt{n})$ for the computation of the median assuming random arrival order. Their key observation, and one that we build upon, is that it is possible to maintain a buffer of $O(\sqrt{n})$ keys, such that, with high probability, at any stage in the sequence of arrivals, the median of every subsequent prefix of the entire arrival sequence of length n either lies in the buffer or has not arrived yet.

The paper [4] by Floyd and Rivest gives an algorithm for computing an α-quantile with high probability using $(1+\min(\alpha, 1-\alpha))n+o(n)$ comparisons.This result matches a simple lower bound derived as follows: let q be the α-quantile. Every key x except q must be compared with some key that is either q or lies strictly between x and q, and the first comparison involving x has probability at least $\min(\alpha, 1-\alpha)$ of failing to fulfill this condition.

The Floyd-Rivest algorithm is not presented as a streaming algorithm but can be adapted under the random arrivals assumption to a basic streaming algorithm with the original number of comparisons that requires storage $n^{2/3}\log n$.

We present an everytime streaming algorithm for computing an α-quantile under the random arrivals assumption with optimal storage $O(\sqrt{n})$ and optimal execution time $O(m) + O(\sqrt{n}\log^2 n)$ to process the first m arrivals.

Let $q(t)$ denote the α-quantile of the prefix of length t. By straightforward random walk arguments we establish the following claims:

1. With high probability the following holds for all t and t' with $t < t'$: if key $q(t')$ lies within the prefix of length t, its rank within that prefix differs from αt by at most $O(\sqrt{n})$.
2. With high probability the cardinality of the set $\{q(t), t = 1, 2, \cdots, n\}$ is at most $\sqrt{n}\log n$; i.e., the number of distinct medians of prefixes is small.

We assume that $\frac{1-\alpha}{\alpha} = a/b$ where a and b are small integers. This assumption is not essential, but simplifies exposition.

The algorithm makes deductions based on the assumption that the input stream satisfies the above two assertions. It is divided into stages. In the first stage $(a + b)\sqrt{n} + 1$ keys arrive, and in each subsequent stage $a + b$ keys arrive. At the start of any stage, after t keys have arrived, the algorithm maintains the following information.

1. The current α-quantile $q(t)$;
2. An interval (L, U) within which every future α-quantile must lie;
3. A set HIGH of $bc\sqrt{n}$ keys greater than $q(t)$ and a set LOW of $ac\sqrt{n}$ keys smaller than $q(t)$ such that every future α-quantile that has already arrived is contained in $HIGH \cup LOW \cup \{q(t)\}$.

In the first stage $(a+b)c\sqrt{n}+1$ keys arrive. The $ac\sqrt{n}$ smallest keys are placed in LOW, the $bc\sqrt{n}$ largest keys are placed in HIGH, and the remaining key is designated $q((a+b)\sqrt{n})$. U is set to $+\infty$ and L is set to $-\infty$. Each subsequent stage has the following phases:

1. $a+b$ keys arrive. Each arriving key greater than U is reassigned the value $+\infty$ and placed in HIGH, and each arriving key less than L is reassigned the value $-\infty$. Of the remaining arriving keys, those greater than $q(t)$ are placed in HIGH and those less than $q(t)$ are placed in LOW.
2. A rebalancing process is carried out in which, depending on the number of newly arriving keys that entered $HIGH$, a new α-quantile is determined, and at most $\max(a, b)$ keys are transferred between HIGH and LOW to achieve the properties that HIGH is of cardinality $bc\sqrt{n} + b$, LOW is of cardinality $ac\sqrt{n} + a$, every key in HIGH is greater than the current α-quantile and every key in LOW is less than the current α-quantile.
3. The b largest elements of HIGH and the a smallest elements of LOW are discarded.
4. L is set to the largest value that has ever been discarded from LOW, and U is set to the smallest value that has ever been discarded from HIGH.

The algorithm uses three mechanisms to achieve efficiency:

1. It keeps a count of the number of keys greater than U and the number of keys less than L that have not yet been discarded, but does not explicitly store those elements. The computational cost of identifying and discarding each such key is $O(1)$.
2. It stores the remaining elements of the sets HIGH and LOW in min-max priority queues, implemented as lazy binomial queues, which perform the insertkey, findmin and findmax operations in amortized time $O(1)$ and the extractmin and extractmax operations in time $O(\log n)$.
3. It maintains a doubly-linked linear list containing those keys that have ever becom the α-quantile or been transferred between HIGH and LOW. Once a key has entered this list, the computation time for each subsequent transfer of the key is $O(1)$. The computation time for the first transfer of any key is $O(\log n)$, the time for an extractmin or extractmax operation.
4. The computation time to discard an element that has not been determined to lie outside $[L, U]$ is $O(\log n)$, the time for an extractmin or extractmax operation.

For all k, the conditional probability that the kth arriving key is not immediately assigned the value $+\infty$ or $-\infty$, given the sequence of previous arrivals, is at most $\frac{(a+b)c\sqrt{n}+2}{k}$. It follows that, with high probability, the total number of such arriving keys is $O(\sqrt{n}\log n)$. Hence, for all m, the time required to process the first m arrivals is $O(m) + O(\sqrt{n}\log^2 n)$.

5 Multiple Selection

In this section we present an anytime streaming algorithm for the following problem. Let $\alpha_1, \alpha_2, \cdots, \alpha_k$ be an increasing sequence of numbers in $(0, 1)$. Given a stream of n keys arriving in a random order, find the $\alpha_1, \alpha_2, \cdots, \alpha_k$-quantiles of every prefix of the stream.

Let $\alpha_0 = 0, \alpha_{k+1} = 1$ and $p_i = \alpha_{i+1} - \alpha_i$, for $i = 1, 2, ..., , , k + 1$. We observe that any comparison-based algorithm to determine the given quantiles must determine the relation of each of the n keys to each of the quantiles. The number of such joint relations is slightly greater than $\frac{n!}{\pi_{i=1}^{k+1}(np_i)!}$. It follows that the expected number of comparisons for any deterministic or randomized algorithm is at least the logarithm base-2 of this quantity, which, by Stirling's approximation, is $nH(p_1, p_2, \cdots, p_{k+1}) + o(n)$ where $H(p_1, p_2, \cdots, p_{k+1})$ is the entropy function $-\sum_{i=1}^{k+1} -p_i \log_2 p_i$.

Our streaming algorithm is based on a binary search tree: a rooted ordered binary tree with k internal nodes labeled in one-to-one correspondence with the α_i, such that the label of the left child of a node is less than the label of the node, and the label of the right child of the node is greater than the label of the node. If the root of the tree is labeled α then the process starts by computing the α-quantile of the set of n keys. The keys less than the α-quantile flow to the left child of the root and the keys greater than the α-quantile flow to the right child of the root. Recursively, the left subtree of the root processes the keys it receives to compute the α_i quantiles of the set of n keys for all $\alpha_i < \alpha$, and the right subtree of the root processes the keys it receives to compute the α_i-quantiles of the set of n keys. for all $\alpha_i > \alpha$. A standard construction from information theory (the Shannon-Fano code) constructs a binary search tree such that, as the keys flow down the tree, the sum of the cardinalities of the sets of keys arriving at the k internal nodes is at most $(H(p_1, p_2, \cdots, p_{k+1}) + 1)n$. A slight variant of that construction ensures that the height of the tree is $O(\log k)$ while increasing the sum of the cardinalities by an arbitrarily small factor $1 + \epsilon$. If each of the k selection problems is solved using the randomized algorithm of Floyd and Rivest the total number of comparisons will be within a factor of $1.5(1 + \epsilon)$ of the information-theoretic lower bound (with high probability).

We will convert this binary search algorithm to an anytime streaming algorithm with storage requirement $O(\sqrt{nk})$ and amortized time $O(1)$ per key (whp), on the assumption that the keys arrive in a random order. To do so, we must reconcile two conflicting requirements:

1. To ensure that the keys arrive at each node in a random order, we require that the keys flowing into each node arrive in their original order;

2. To ensure that the process terminates within time $O(n)$, we require that, as a key flows down the tree, it must dwell at each node only for $O(1)$ time steps.

At first sight, this is an unsolvable dilemma. At each node, a key must be immediately routed to the left child or right child according to whether it is less than or greater than the quantile being computed at that node; but the quantile cannot be known until all the keys have arrived at the node. To resolve the dilemma, we run our everytime streaming algorithm for selection at each node, and route each arriving key immediately to the left child if it is less than the current α-quantile (rather than the unknown eventual α-quantile of the entire input stream), and to the right child if it is greater than or equal to the current α-quantile. Since the everytime selection algorithm processes the first m arriving keys in time $O(m+\sqrt{n}\log n)$ there will be an excess delay of at most $O(\sqrt{n}\log n)$ at each node and, since our binary search tree has height at most $O(\log k)$, a total excess delay of at most $O(\sqrt{n}\log n \log k$. However, a key will be misdirected if its relation to the current α-quantile is different from its relation to the final α-quantile. Fortunately, the keys that could potentially be misdirected are the ones that get transferred out of HIGH or out of LOW during the computation of the quantile at the node. These are precisely the keys that get placed in the doubly-linked list maintained by the algorithm, and the number of such keys is $O(\sqrt{n}\log n)$ (whp). Thus, after the computation of the final α-quantile, the selection algorithm can scan this list and send each of its children a list of all the misdirected keys. Each child can make appropriate corrections in time $O(\log n)$ per misdirected key. The correction computed at each child can affect its list of misdirected keys, and so on down the tree. The total delay incurred by the ripple effect of these misdirections is $O(\sqrt{n}\log^2 n \log^2 k)$. Thus the time required to compute all k α-quantiles is $O(n)$. The storage required at each node is proportional to the square root of the number of arriving keys; thus the total storage requirement is $O(\sqrt{nk})$.

6 Parallel Selection

In this section we consider the problem of selecting the α-quantile of a sequence of n keys, assuming that the keys arrive in k streams of length n/k to be processed in parallel by k processors. We assume that the keys arrive in a random order; *i.e.*, that all $n!$ assignments of the set of arriving keys to positions in the streams are equally likely. We give a parallel anytime algorithm based on the serial selection algorithm of Section 4. As before, we assume for convenience that $\alpha = \frac{a}{a+b}$ where a and b are small integers.

The algorithm starts by filling the buffers with arriving keys. It then goes through a series of stages, each of which (except the last) starts with all the buffers full. In each stage it is determined that the final α-quantile lies in an interval (L, U) (whp). As many keys less than L or greater than U as possible are then discarded from the buffers, subject to the requirement that the ratio between the numbers of discarded keys greater than U and less than L must be exactly b/a.The buffers are then replenished with keys from the streams. The processes of determining L and U and discarding high and low keys require communication and transfer of keys among the processors.

These processes are based on a parallel algorithm to compute an approximate β-quantile of the set of sk keys in the union of the k buffers. We begin by presenting such an algorithm for the case $\beta = 1/2$.Let 3^t be the largest power of 3 less than or equal to sk. The computation goes through t rounds of *thinning*, starting with 3^t keys from the union of the buffers. in each round the surviving keys are grouped randomly into sets of 3, and the median of each set of 3 keys survives to the next round.Analysis of this process shows that, with probability at least .96, the final surviving key is a γ-quantile, where $|\gamma - 1/2| < 2/3(11/8)^{-t}$.

During the thinning process some groups must be composed of nodes from different processors. For this purpose the processors configure themselves into a virtual linked list. Initially, each node performs the thinning process on the groups formed within its own buffer. Then, in subsequent rounds of thinning, the surviving keys are transferred to nodes whose addresses in the list are multiples of 3, then $3^2, 3^3$ etc.

For any β, the determination of an approximate β-quantile can be reduced to the determination of an approximate median by executing a special initial round of thinning. We present the details for the case $\beta < 1/2$. Let m be the greatest integer such that $(1-\beta)^m > 1/2$. Let $p \in (0,1)$ be such that $p(1-\beta)^m) + (1-p)(1-\beta)^{m+1}) = 1/2$. Then, in the special round, the keys are grouped randomly, where the size of each group is m with probability p and $m+1$ with probability $1-p$, and the smallest key in each group survives. Throughout the special round and the subsequent thinning rounds, any rule for grouping the surviving keys can be used, as long as it depends on the positions of keys within the buffers, but not on their values. since the assignment of the keys to input streams, and hence the assignment of keys to positions in the buffers, is random.

The processors use the thinning algorithm to find keys L and U such that all future $\frac{a}{a+b}$-quantiles lie in the interval (L, U) (whp). This claim holds provided that L is of of rank A and U is of rank $sk - B$ in the set of sk keys contained in the buffers of the k processors, such that $A \leq a((\frac{sk}{a+b} - c\sqrt{n})$ and $sk - B \leq b((\frac{sk}{a+b} - c\sqrt{n})$ To achieve this, the thinning algorithm is used to find approximate β and γ-quantiles, where $\beta = (1-\epsilon)a(\frac{sk}{a+b} - c\sqrt{n})$ and $\gamma = (1+\epsilon)b((\frac{sk}{a+b} - c\sqrt{n})$. L is set to the approximate β-quantile and U, to the approximate γ-quantile. Here ϵ is a small positive constant, and the factors $1-\epsilon$ and $1+\epsilon$ are safety factors to ensure that A and B are likely to satisfy the required inequalities even though the thinning algorithm only produces approximate β and γ-quantiles.

After L and U have been determined each processor counts the number of keys less than L and the number of keys greater than U in its buffer. The processors organize themselves into a virtual rooted binary tree and, aggregate their counts by passing messages toward the root. After $O(\log k)$ parallel message-passing steps the root contains the aggregate counts A and B of the numbers of keys less than L and greater than U. In the unlikely event that A and B fail to satisfy the required inequalities the randomized thinning algorithm is invoked to recompute L and U. If A and B do satisfy the inequalities then using message passing along edges directed away from the root, the processors are directed to discard ra of the packets less than L and rb of the packets greater than U, where $r = \min(\lfloor A/a \rfloor, \lfloor B/b \rfloor$. Each processor then receives keys from its input stream until its buffer is full.

The running time of the parallel algorithm is dominated by O(n/k), the time required by each processor to read its input stream. In addition, each of the $O(n/sk)$ stages requires time $O(\log(sk)$ time for the parallel communication required in computing L, U, A and B.

7 Approximate Selection

We begin with the following problem of computing an approximate median: given an array of n keys, choose a key x such that, with probability at least $1 - \delta$, the rank of x differs from $n/2$ by at most ϵn. Vitter [7] has given the following solution: set x equal to the median of a random sample of $O(\frac{1}{\epsilon^2} \log(\frac{1}{\delta}))$ keys. If the stream of keys arrives in a random order then we can use a prefix of the stream as the sample. By applying our streaming algorithm to this prefix, we obtain an approximate median using $O(\frac{1}{\epsilon^2} \log(\frac{1}{\delta}))$ comparisons and storage $O(\frac{1}{\epsilon}\sqrt{\log(1/\delta)})$.

Here we note that an approximate median can be computed by a streaming algorithm using a slightly larger number of comparisons but only two storage locations. The algorithm considers a series of arriving keys as candidates for the ϵ-approximate median.Each candidate in turn is compared to a sequence of arriving keys, and the algorithm keeps track of the *lead* of the candidate, defined as the number of times the candidate is larger than the arriving key, minus the number of times it is smaller. If the lead remains in the interval $(-a, a)$ for s steps then the candidate is declared to be an ϵ-approximate median. Otherwise it is dismissed and the next arriving key becomes the new candidate. Here $s = O(\frac{1}{\epsilon^2} \ln(\frac{1}{\delta}))$ and $a = 0.4s\epsilon$. Using Chernoff bounds we establish the following:

1. If the rank of the candidate differs from $n/2$ by at most $\frac{\epsilon}{8}$ then, with probability at least $1 - \delta$, the candidate will be accepted.
2. If the rank of the candidate is np, where $\frac{\epsilon}{8} < |p - 1/2| < \epsilon$ then the candidate may or may not be accepted, but the number of comparisons performed on it will not exceed s;
3. If the rank of the candidate is np, where $|p - 1/2| > \epsilon$, then the probability of incorrectly accepting the candidate is $O(e^{-\frac{s(|2p-1|-4\epsilon)^2}{6p}})$ and the expected number of comparisons until it is rejected is at most $\frac{.4n\epsilon}{|2p-1|}$. Since $|2p - 1|$ is uniformly distributed over the interval $(2\epsilon, 1)$,we find by integrating over this interval that the expected number of comparisons performed on a candidate with $|p - 1/2| > \epsilon$ is $O(\frac{1}{\epsilon} \ln 1/\epsilon)$.
4. The number of candidates considered will be a geometric random variable with expectation $O(\frac{1}{\epsilon})$ and the number of candidates considered with $|p - 1/2| < \epsilon$ will be a geometric random variable with expectation $O(1)$.

5. The probability that the accepted candidate is not an ϵ-approximate median is bounded above by a constant times δ;
6. The number of comparisons performed by the algorithm is $O(\frac{n}{\epsilon^2}\max(\ln 1/\delta, \ln(\frac{1}{\epsilon}))$ (whp).

The computation of an approximate α-quantile can be reduced to the computation of an approximate median using the reduction based on thinning given in Section 5.

8 Approximate Sorting

In certain applications it suffices to sort a set of elements approximately rather than exactly. For example, in ranking candidates for adnmission to an academic department it may be important to rank the best candidates exactly, but an increasingly rough ranking may be adequate as we go down the list. We formulate the problem of approximate sorting in terms of a parameter $\epsilon > 0$. Our requirement is that, for all r, a candidate of rank r is assigned a rank that differs from r by at most ϵr.

Let ϵ be a positive constant. Let $x_1, x_2, \cdots, x_n$ be a linearly ordered set of keys and let π be the unique permutation of $\{1, 2, \cdots, n\}$ such that $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$. Let σ be a permutation of $\{1, 2, \cdots, n\}$. Then σ is said to ϵ-sort the keys if, whenever $\pi(i) = \sigma(j)$, $(1-\epsilon)i \leq j \leq (1+\epsilon)i$. In other words, σ ϵ-sorts the keys if, for all r, the key of rank r in the true ordering has rank between $(1-\epsilon)r$ and $(1+\epsilon)r$ in the ordering σ.

We shall derive a lower bound on the number of comparisons required to ϵ-sort a set of n keys. Call a permutation θ of $\{1, 2, \cdots, n\}$ an ϵ-permutation if, for all i, $(1-\epsilon)i \leq \theta(i) \leq (1+\theta)i$. If π is the true ordering of the keys, then permutation σ ϵ-sorts the keys if and only if $\sigma \circ \pi^{-1}$ is an ϵ-permutation. Let $V(n, \epsilon)$ be the number of ϵ-permutations of $\{1, 2, \cdots, n\}$ Then, if an ϵ-sorting algorithm returns the permutation σ, then there are only $V(n, \epsilon)$ possibilities for the true permutation. Since *a priori* there are $n!$ possible true permutations, the program must be able to output at least $n!/V(n, \epsilon)$ permutations and,by a standard argument, the worst-case number of comparisons performed by any comparison algorithm for ϵ-sorting is at least the base-2 logarithm of this number of permutations. This lower bound also holds for the expected number of comparisons in a randomized algorithm when the true permutation is drawn uniformly at random from the set of all permutations.

$V(n, \epsilon)$ is the permanent of the $n \times n$ $0-1$-matrix A whose $i-j$ element is 1 if and only if $(1-\epsilon)i \leq j \leq (1+\epsilon)i$.Bregman's Theorem [1] states that if a_i is the number of 1's in the ith row of a $n \times n$ $0-1$-matrix then the permanent of the matrix is bounded above by $\pi_{i=1}^{n}(a_i!)^{\frac{1}{a_i}}$. For the matrix A , $a_i \leq \lceil 2\epsilon i \rceil$. A short calculation based on Stirling's Inequality yields : $\log_2 \frac{n!}{V(n,\epsilon)} \geq n \lg(\frac{e}{2\epsilon})$.

We shall give a two-pass streaming algorithm for ϵ-sorting. The first pass computes elements of all ranks of the form $\lceil \frac{n\epsilon}{(1+\epsilon)^i} \rceil$ for all positive integers i using the multiple selection algorithm of Section 5. In this case the entropy term

$H(p_1, p_2, \cdots, p_{k+1}))$ is $\lg(\frac{1}{\epsilon} + \frac{(1+\epsilon)\lg(1+\epsilon)}{\epsilon})$, which is less than $\lg(\frac{1}{\epsilon} + (1+\epsilon)\lg e$. Thus the execution time of phase 1 is at most $1.5(1 + \lg(\frac{1}{\epsilon} + (1+\epsilon)\lg e)n$.

In the second pass a binary search is executed on each key x to determine an i such that $r_i \leq x < r_{i+1}$, and an approximate rank is assigned to x accordingly. The number of comparisons performed in the second pass is at most $(1 + \lg(\frac{1}{\epsilon} + (1+\epsilon)\lg e)n$.

We present an alternative algorithm for the first pass in the spirit of the well-known algorithms Quicksort and Multiple Quickselect [6]. We first describe the algorithm in a setting where the keys to be approximately sorted are presented in random order in an array. We then modify the algorithm to obtain an anytime streaming algorithm.

The array extends from address 0 to address $n + 1$. The actual keys are in locations 1 to n; location 0 contains a sentinel key equal to $-\infty$ and location $n+1$ contains a sentinel key equal to $+\infty$. At a general step the array contains a set S of *occupied locations*. Initially, locations 0 and $n+1$ are considered occupied and the other locations are considered unoccupied.The following invariant properties hold at every step:

1. The n original keys occur in locations $1, 2, \cdots, n$ in some order;
2. If location i is occupied then the key it contains has rank i in the original set of keys, locations $1, 2, ..., \cdots, i-1$ contain the keys of rank less than i, and locations $i+1, \cdots, n$ contain the keys of rank greater than i.

If locations i and j are occupied, and all intervening locations are unoccupied, then the interval $[i, j]$ is considered *splittable* if $j - 1 > (1+\epsilon)(i+1)$. The computation terminates when no splittable intervals remain. At that point the array is ϵ-sorted.

Initially $[0, n+1]$ is a splittable interval. At each step, a random location within a splittable interval is chosen and each of the other keys in the interval is compared with the key $x*$ in that location. Based on those comparisons, the keys within the interval are rearranged such that $x*$ is preceded by the keys less than $x*$ and precedes the keys greater than $x*$.

Next we calculate the expected number of comparisons for this algorithm. Define the length of the interval $[i, j]$ to be $j - i + 1$. Interval $[i, j]$ is *potentially splittable* if $(j-1) > (1+\epsilon)(i+1)$. A potentially splittable interval becomes splittable if and only if the two end positions of the interval become occupied before any of the internal positions become occupied. If a potentially splittable interval of length t becomes splittable in the course of the algorithm then it will be split at the cost of $t - 3$ comparisons.

For each t we characterize the potentially splittable intervals of length t and the probability that they will be split. The conditions for an interval $[i, j]$ of length t to be potentially splittable are as follows:

- $t \geq 4$;
- $i \leq n + 2 - t$
- $i < \frac{t-1+\epsilon}{\epsilon}$

The probability of a potentially splittable interval $i, j]$ of length t becoming splittable is 1 if $i = 0$ and $j = n + 1$; $\frac{1}{t-1}$ if $i = 0$ and $j \leq n$ or $i \geq 1$ and $j = n + 1$; and $\frac{1}{\binom{t}{2}}$ if $i \geq 1$ and $j \leq n$.

Using these results we can compute the expected number of comparisons performed to split intervals of length t and, summing over t, we find that the expected number of comparisons performed by the algorithm is asymptotic to $n(\frac{2+3\epsilon}{1+\epsilon} + \ln(\frac{1+\epsilon}{\epsilon}))$.

Incidentally, by varying the definition of a potentially splittable interval, this approach also gives remarkably simple expected-time analyses of some classical randomized interval-splitting comparison algorithms such as Quicksort, Quickselect and Multiple Quickselect.

We now modify this algorithm to obtain an anytime streaming algorithm for the first phase. As the keys arrive we designate certain keys as *landmarks*; these play the same role as the keys occurring in occupied positions in the foregoing array-based algorithm. The landmarks are maintained in a self-balancing binary search tree such as a splay tree. Each arriving key is routed to a leaf of the tree (corresponding to an interval between consecutive landmarks) by comparing it with landmarks according to the usual insertion algorithm for a self-balancing binary search tree. The main difference from the array-based algorithm is that, because of storage limitations, we cannot retain all the keys that have arrived at a leaf. Instead, the algorithm counts the arriving keys, and also applies the thinning algorithm of Section 6 to compute an approximate median to be used in splitting the interval.The thinning algorithm can be implemented to run in working storage logarithmic in the number of arriving keys.

We also associate with each node (including both landmarks and leaves) an estimate of the number of keys that have arrived in the subtree rooted at that node. When a key arrives the estimate for each node along its insertion path is incremented by 1.

Let x and y be two consecutive landmarks. The interval between x and y is split when the estimate of the number of keys in that interval exceeds ϵ times the estimate of the number of keys less than or equal to x (the latter estimate is obtained from the estimates for nodes along the insertion path to x). In that case z, the approximate median computed by the thinning algorithm for the interval $[x, y]$, becomes a landmark; the leaf associated with that interval is replaced by a 2-leaf subtree rooted at z, and the estimate ascribed to each of the newly created intervals is set to half the estimate for the interval between x and y.To compensate for the inaccuracy of the approximate median provided by the thinning algorithm, the entire algorithm is run for a value of ϵ slightly smaller than the required tolerance.

With high probability,the following hold for any fixed ϵ: the number of landmarks created is $O(\log n)$, the storage requirement of the algorithm is $O(\log^2 n)$, and no interval between consecutive landmarks is splittable (*i.e.*, the actual number of keys in that interval does not exceed ϵ times the actual number of keys preceding that interval). The number of comparisons performed in the first phase is $O(n \log \frac{1}{\epsilon})$ (whp).

In the second pass each arriving key is inserted into the binary search tree created in the first pass, and a count of the exact number of keys in each interval is maintained. Then in a third pass, each key is reinserted and assigned its approximate rank according to the interval into which it falls.

References

1. Bregman, L.M.: Some properties of nonnegative matrices and their permanents. Soviet Math. Dokl 14, 945–949 (1973)
2. Bryant, R.E.: Data-intensive supercomputing: the case for DISC.Technical Report CMU-CS-07-128, Carnegie-Mellon University School of Computer Science (2007)
3. Chazelle, B.: The soft heap: an approximate priority queue with optimal error rate. Journal of the ACM 47 (2000)
4. Floyd, R.W., Rivest, R.L.: Expected time bounds for selection. Communications of the ACM 18(30), 165–172 (1975)
5. Munro, J.I., Paterson, M.S.: Selection and sorting with limited storage. Theoretical Computer Science 12, 315–323 (1980)
6. Prodinger, H.: Multiple quickselect: Hoare's find algorithm for several elements. Information Processing Letters 56, 123–129 (1995)
7. Vitter, J.S.: Random sampling with a reservoir. ACM Trans. on Math Software 11(1), 37–57 (1985)