SORTING REAL NUMBERS ON RANDOM ACCESS MACHINES

By

Daniel S. Blumenthal

B. S. (Computer Science) Syracuse University

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in
THE FACULTY OF GRADUATE STUDIES
COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

September 1995

© Daniel S. Blumenthal, 1995
In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Computer Science
The University of British Columbia
2366 Main Mall
Vancouver, Canada
V6T 1Z4

Date: September 7, 1995
Abstract

This work describes a family, Segment-Sort, of algorithms for rapid sequential sorting of real numbers. Two computational models are discussed which correspond to the two main types of Segment-Sort algorithm: deterministic and random. With the deterministic model, the Basic RAM, it is possible to sort input populations randomly chosen from a broad class of common probability distributions in "space" (number of memory words, each able to hold a real number) and average time both linear in the number of real numbers given as input. Included among these distributions are a variety of types containing singularities, unbounded oscillations and points of actual nonzero probability (atoms). With the second model, the Random RAM, one may sort \( n \) arbitrarily chosen distinct real numbers in \( O(n) \) operations using only \( O(n) \) memory words on average. Except for random integer selection on the Random RAM, both models are confined to simple binary arithmetic. The power of both models appears to stem largely from the combination of left and right shifting operations.
# Table of Contents

Abstract ii

Table of Contents iii

Acknowledgement v

1 Introduction 1

1.1 Earlier Work on the Sorting Problem 1

1.2 Terminology and Notational Conventions 2

1.3 Computational Models 3

1.4 The Power of Bidirectional Shifting 4

1.5 Derived Constant-Time Operations 6

1.6 Random Variables Bounded by Input Size and the “Logarithm Limit Rule” 6

2 Fast Deterministic Sorting 8

2.1 Linear-Time Deterministic Sorting 8

2.1.1 Double-Bucketing and Linear-Time Performance 8

2.1.2 The Basic Algorithm 9

2.1.3 Performance of Segment-Sort1 13

2.1.4 Resolving Singularities in Linear Expected Time 18

2.2 Additional Improvements on the Comparison-Based Time Complexity for Sorting 29

2.2.1 Infinite Collections of Atoms 29
2.2.2 Arbitrary Isolated P.D.F. Discontinuities .......................... 33

3 Random Sorting of Distinct Real Numbers in Linear Expected Time 35
   3.1 The Random-Segment-Sort Algorithm ............................... 36
   3.2 Analysis of Random-Segment-Sort ................................. 39

Bibliography 45
Acknowledgement

Firstly, I thank my supervisor, Dr. Nicholas Pippenger for helping me at every stage in the conceptual development and writing of this thesis. I thank NSERC for its financial support of this project. I also wish to thank Dr. Alan Wagner, whose guidance and recommendations have been of great assistance in making this a presentable document. In addition, I thank Dr. David Kirkpatrick for personally aiding me in the development of important ideas relating to the latter parts of this work.
Chapter 1

Introduction

1.1 Earlier Work on the Sorting Problem

The problem of sorting items into order by rank or numerical value is one that has been given alot of attention, particularly by computer scientists. A wide variety of different methods and models for sorting have been developed. The most widely used and heavily studied of these models are the comparison-based models, in which it is assumed that the only primitive operation available for getting information about the relative rankings of elements in a collection of ranked elements is the comparison operation, which takes two elements as operands and tells you whether the first element is less than, equal to, or greater than the second. It has been shown that under this assumption, given $n$ real numbers to sort, any possible sorting algorithm will have to do $\Omega(n \log_2 n)$ such comparison operations, on average, to put the numbers in order [Knu73].

There are, however, other models for sorting which do not depend solely on comparisons, in which it is possible to sort $n$ real numbers with fewer than $\Omega(n \log_2 n)$ primitive instructions all within the realm of simple arithmetic and random choice. Of particular interest among the algorithms supported by such models are the distributive sorting algorithms (e.g., bucket-sort as described in [CLR90]). The principal algorithms analyzed here fall within this class.
1.2 Terminology and Notational Conventions

In keeping with standard terminology, the number, \( t \), of primitive operations executed by a procedure, \( P \), for any given input \( S \) is indicated by saying that \( P \) requires or runs in “time” \( t \) on input \( S \). Note that when I refer to a quantity, \( x(n) \), as being “\( O(n) \),” I mean that there is a fixed real number \( k \) that is independent of all variable quantities, such that for all sufficiently large \( n \):

\[
|x(n)| < kn.
\]

The precise meaning of “sufficiently large” in this context may, however, be dependent on variable quantities (such as an underlying probability distribution from which inputs are randomly drawn, where \( n \) is the number of inputs). Furthermore, in referring to a quantity, \( x(n) \), as being “\( o(n) \),” I allow the bounding function implicit in the “little oh” notation to depend, similarly, on variable quantities such as underlying distributions.

Also, when I refer to a sort-routine as “standard,” I mean that it is a comparison-based sort-routine (e.g. heapsort) which, given \( n \) ranked elements as input, will sort them in \( O(n \log_2 n) \) time in the worst possible case, using \( O(n) \) extra memory words.

Present in the text as well are some more mathematical references whose precise meaning may not be apparent. Among these are references to “resolving” singularities and other probability distributions. Generally, by saying that a particular sorting algorithm \( (A) \) “resolves” a particular distribution \( (D) \) in \( O(f(n)) \) time, I mean that the expected time for the sorting by \( A \) of \( n \) elements randomly drawn from \( D \) is \( O(f(n)) \). In connection with probability distributions, I also refer to a probability density function as a p.d.f., and to the cumulative distribution function corresponding to any probability distribution as a c.d.f. I also refer to real values taken on by a random variable that have nonzero probabilities of occurrence as “atoms.” In a corresponding manner, I refer to buckets (i.e., subsets of the input generated in a bucket-sort [CLR90] or other bucketing
algorithm) whose corresponding bucket-intervals contain zero, one, or multiple atoms as "nonatomic," "monatomic," and "multi-atom" buckets, respectively.

Finally, to allow the analysis to proceed at a higher level, I also observe the following conventions. Typically, the primary quantities under discussion will be mathematical (i.e., numbers rather than memory locations or the binary string representations of numbers therein). Objects to which primitive instructions and other procedural steps are applied will commonly be treated as numbers where the intended meaning is clear (e.g., addition will be discussed in a mathematical context although the actual Add instruction described above, by nature, must operate on memory words). To denote the number of elements in a set or object \( S \) made up of multiple elements, I use the symbol \( | S | \), consisting of the symbol for the object itself surrounded by vertical bars. I also use surrounding vertical bars to represent the absolute value of a number, allowing the meaning of the bars in each case to be indicated by context.

### 1.3 Computational Models

Both the Basic RAM and the Random RAM have memory locations that can hold any real numbers. Indirect addressing is assumed on both models (i.e., memory locations passed as arguments may be computed values). The Basic RAM's primitive instruction set is included in that of the Random RAM. eight operations are regarded as primitive on the Basic RAM. These are:

1. **Copy\( (a,b) \):** Copy the contents of memory location \( a \) into memory location \( b \). Both \( a \) and \( b \) are permitted to be any of the active memory words or any contiguous external file segment.

2. **Add\( (a,b) \):** Add real numbers \( a \) and \( b \), and return their sum.
3. Negate(a): Return the additive inverse of real number a.

4. L-Shift(a,j): Return $2^j a$ (where $a$ is a real number and $j$ is an integer).

5. R-Shift(a,j): Return $2^{-j} a$ (where $a$ is a real number and $j$ is an integer).

6. Floor(a): Return the greatest integer less than or equal to the real number a.

7. Compare(a,b,$\{L_1, L_2, L_3\}$): Return the comparative relationship of $a$ to $b$ (where $a$ and $b$ are real numbers). That is, if $a < b$ then return the result “less than,” if $a = b$ then return the result “equal” and if $a > b$ then return the result “greater than.” If “less than” is returned, jump (i.e., transfer the flow of program control) to the instruction immediately following location $L_1$ in the program’s code. If “equal” is returned, jump to the instruction immediately following $L_2$, and if “greater than” is returned, jump to that immediately following $L_3$.

8. Point-Normalize(a): Return the greatest integer less than or equal to $\log_2(a)$.

The Random RAM is the same as the Basic RAM except for having in its primitive instruction set the following additional instruction:

Random(i,j): Return an integer randomly chosen from the among those between integers $i$ and $j$ inclusive, with each integer in this range having an equal chance of being chosen.

1.4 The Power of Bidirectional Shifting

The majority of “realistic” computational models which are commonly regarded as sequential, strictly speaking, do things in parallel, since, any time one performs a binary operation on two data that are longer than one bit each — like comparing two floating-point numbers — and counts it as a unit operation, all the bits of the two data are
implicitly accessed in parallel. The degree of implicit parallelism required reaches an extreme in the case of real number sorting on a digital machine, as even to perform basic actions like comparing two real numbers in finitely many unit steps, one may have to access any number of bits (in the case of two equal irrationals, a literally infinite number of bits) in parallel. Consequently, it is generally not a simple matter to classify a machine that handles real numbers as being strictly sequential, and typically, such classifications must take individual problems' contexts into account.

It is, perhaps, more instructive to analyze the power of a machine model relative to other models in the context of a specific problem. The primitive operation sets of the two machine models presented here, particularly that of the Random RAM (as we shall see in the third chapter), though limited to simple arithmetic (and random integer selection), confer a good deal of power in the context of real number sorting. In particular the combination of the two shift instructions (L-Shift and R-Shift) appears to be quite powerful. In fact, it has been proven that on a machine whose primitive instructions are Add, Negate, Compare, L-Shift, R-Shift [and Floor] [KR84], it is possible to sort an arbitrary collection of integers in worst-case linear-time, but on a machine with only the Add, Negate and Compare instructions, the worst-case complexity for sorting $n$ arbitrarily chosen integers is $\Omega(n \log_2 n)$ [PS]. It is interesting to note that even when the L-Shift instruction (without R-Shift) is added to these three instructions, the resulting machine model still requires $\Omega(n \log_2 n)$ time to sort $n$ integers in the worst case [Rei]. Since the results presented here for the Random RAM depend on its ability to sort arbitrary integer-valued input sets in worst-case linear-time, it is important when considering these results to keep in mind the power of the two shift instructions combined.
1.5 Derived Constant-Time Operations

There are two particularly convenient operations, Binary-Floor and Binary-Ceiling, which, though not defined as primitive themselves for either the Basic or the Random RAM, may be implemented in constant time using the Basic RAM's primitive operations. Binary-Floor($x$) takes as input a positive real number $x$ and returns $2^{\lfloor \log_2 x \rfloor}$, while Binary-Ceiling($x$) takes as input a positive real number, $x$, and returns $2^{\lceil \log_2 x \rceil}$. It is clear that the expression $\lfloor \log_2 x \rfloor$ can be computed from a single Point-Normalize operation, and raising 2 to the power of the resultant value amounts to a single shift. A slight modification yields the Binary-Ceiling. These operations are helpful in proving a of number basic results.

1.6 Random Variables Bounded by Input Size and the "Logarithm Limit Rule"

This section introduces a useful theorem and corollary thereof which is applied later on in the text. The corollary, in particular, allows easy establishment, within certain algorithms, of $O(n)$ expected time complexity for sorting the contents of a bucket when its corresponding bucket-interval lies adjacent to a singularity under certain constraints.

**Theorem 1** Let $X$ be any random variable whose value is at most $n$, and let $g$ be any nondecreasing function that assigns a nonnegative value to every nonnegative integer. Then

$$E[Xg(X)] \leq g(n)E[X].$$

**Proof:**

$X$ is always bounded above by $n$. Since $g$ is nondecreasing as well, it therefore follows that $g(X) \leq g(n)$, so that $Xg(X) \leq Xg(n)$ (always). Therefore, $E[Xg(X)] \leq E[Xg(n)]$
Since the sorting algorithms used as subroutines at the lowest level by the procedures described herein are standard sort-routines, the application of the preceding theorem to the case where \( g(x) = \log_2(x) \) has particular relevance. Hence the following corollary:

**Corollary 2 (The “Logarithm Limit Rule”)** Let \( n \) and \( X \) be as defined in Theorem 1. Then

\[
E[X \log_2 X] \leq \log_2 (n) E[X].
\]

**Proof:**

Just take \( g(x) = \log_2 (x) \) in Theorem 1.
Chapter 2

Fast Deterministic Sorting

2.1 Linear-Time Deterministic Sorting

2.1.1 Double-Bucketing and Linear-Time Performance

The analysis for deterministic linear-time results applies an adaptation of a theorem (Theorem 1.5 in [Dev86]) according to which the following holds:

Let \( f \) be any probability density function with nonzero values confined to \([0,1]\) for which
\[
\int f^2(x) \, dx < \infty.
\]
Then there is a distribution independent positive constant \( k \) such that there exists a positive integer \( N \) dependent upon \( f \) such that for every integer \( n > N \) for every random selection, \( S \), of \( n \) elements from \( f \), \( S \) may be bucket-sorted in expected time less than \( kn \).

This result is proven by reference to the "double-bucketing" algorithm, which plays a fundamental role in some of the deterministic Segment-Sort algorithms. Theorem 1.5 in [Dev86] entails an assumption of bottom-level quadratic-time sorting within buckets. Here, this theorem is adapted to apply to the case where a standard sort-routine is used at the bottom level, rather than a quadratic-time sort-routine.

The distribution-independent nature of double-bucketing's utility was observed independently by Tamminen in [Tam85]. The idea of double-bucketing itself is much older (see [Mac66]).
2.1.2 The Basic Algorithm

Below, a naive Segment-Sort algorithm (Segment-Sort1) is described in a pseudocode-like format. For Segment-Sort1, it is assumed only that the input numbers are randomly chosen from some probability distribution with support confined to the interval [0,1] and made up of two subdistributions or parts, (A) and (B), where part (A) is a probability density function, $f$, multiplied by some positive real constant $c \leq 1$ with $\int_{0}^{1} f(t) \log_2 f(t) \, dt < \infty$; and part (B) is a discrete distribution whose domain is a finite set of real numbers to each of which a nonzero probability corresponds. A probability distribution of this type may be defined precisely with respect to its cumulative distribution function (c.d.f.) by saying that its c.d.f. is the sum of two parts, $(A_c)$ and $(B_c)$ where:

1. there is a function $f$ — zero-valued outside [0,1] — such that $\int_{0}^{1} f(t) \log_2 f(t) \, dt < \infty$, and for any real $x$, $A_c(x) = \int_{0}^{x} f(t) \, dt$; and

2. $B_c$ is a step function of finitely many steps which rises from zero to its highest step within the interval [0,1]

Either part (A) or part (B) may be empty (i.e., either $A_c(x)$ or $B_c(x)$ may be equal to zero for all $x$). I refer to this type of probability distribution as an AB-distribution.

Let $S_n$ be a population of $n$ numbers randomly chosen from an arbitrary AB-distribution, and let $t(S_n)$ be the time that Segment-Sort1 requires to sort $S_n$. I prove here that for all sufficiently large $n$, $E[t(S_n)] \leq kn$ where $k$ is a constant independent of the distribution from which the numbers are chosen.

\[
\text{Segment-Sort1(Input\_File, Output\_File)}
\]
(1) Read the input from Input_File into an input array \( (\text{In}) \), and count the number \( (n) \) of input elements. Step (1) requires \( O(n) \) time.

(2) Load the input elements \( (\text{In}[1..n]) \) into buckets according to their values as in the first stage of an ordinary bucket sort [CLR90], except for using \( \text{Binary-Ceiling}(n) \) instead of \( n \) buckets. For each input element \( q \), the bucket that receives element \( \text{In}[q] \) will be bucket number \( \lceil (\text{Binary-Ceiling}(n)\text{In}[q]) + 1 \rceil \). Also count the elements that fall into each bucket and record the location of the bucket. Step (2) requires \( O(n) \) time.

(3) Apply steps (1) and (2) as in a recursive descent (but without exceeding a recursion depth of two) successively to each of the buckets formed in the preceding step that contains more than a certain fixed threshold number, \( E_1 \), of elements, applying a “bottom-level” sort-routine to the other primary (i.e., initially formed) buckets as in a standard bucket sort. This “double-bucketing,” [Dev86] will yield a (possibly-empty) collection of secondary buckets within any of the primary buckets that contains more than \( E_1 \) elements. Step (3) requires \( O(n) \) time.

(4) To each of these secondary buckets in turn, apply the Atom-Handler algorithm described below — passing as parameters the bucket and the number of its elements. When this procedure has been applied to each of the secondary buckets in each of the primary buckets, the end result will be a complete ordering of the input. (See the description of the Atom-Handler routine below for a time-complexity analysis of this step of Segment-Sort1.)

(5) Let a terminal bucket be any bucket that has thus far (i.e., by the end of step (4)) been formed but not further subdivided. Read the sorted elements in order (from the first element in the first terminal bucket to the last element in the last terminal bucket)
Chapter 2. Fast Deterministic Sorting

into Output_File. This final step of Segment-Sort1 requires $O(n)$ time.

Atom-Handler (< size of bucket (B) >, < (location of) bucket B >)

1. If $|B|$ (i.e., the size, or number of elements in B) exceeds a certain threshold number, $E_2$, of elements, then load the subpopulation, $S$, consisting of B's first $\sqrt{|B|}$ elements into an array; otherwise, sort the contents of B using a standard sort-routine and terminate this algorithm. Clearly, step (1) requires $O(\sqrt{|B|})$ time (per execution) in the worst case.

2. Use a standard sort-routine to sort $S$. This requires $O(\sqrt{|B| \cdot \log_2 |B|})$ time.

3. Apply the Find-Atom-Value subroutine (described below) to the (now sorted) sample $S$. This takes $O(|S|)$ time. Since $|S| = \sqrt{|B|}$, this step takes $O(\sqrt{|B|})$ time.

4. If an atom-value (rather than the value "nonatomic") has been returned in the preceding step, create three sub-buckets for the elements of B: the first for elements whose values are less than the atom-value ("low" elements), the third for elements whose values exceed the atom-value ("high" elements) and the middle bucket for elements that have precisely the atom-value ("atomic" elements). Establish a pointer to the location of each of these sub-buckets. Only three buckets (if any) are created in this step, so it takes $O(1)$ time.

5. If three buckets were created in the preceding step (step (4)), read the elements of B, comparing the value of each to the atom-value returned in step (3), and placing the low elements, atomic elements and high elements in their respective sub-buckets created.
in step (4). This requires $O(|B|)$ time.

(6) If $B$ has been partitioned into sub-buckets, independently sort the elements in the buckets for high and low elements using a standard sort-routine. The atomic elements are all identical in value and in a common bucket, so they’re already sorted in this case. If $B$ has not been subpartitioned, simply use a standard sort-routine to sort all of its elements. The amount of time required by this step is analyzed in the following section of this paper.

(7) Return the pointers created in step (4). This obviously takes constant time.

---

**Find-Atom-Value ($<$ finite nondecreasing sequence of ranked elements $(S)$ $>$)**

(1) Letting the current element value begin as that of the first element of $S$, set a counter (CT) equal to one, a “current mode count” register (CMC) equal to zero, and a “current mode value” register (CMV) equal to the current element value. This, of course, takes constant time.

(2) Read $S$ from beginning to end. While doing so, increment CT by one at each reading of an element that equals the previous element. At each reading of an element unequal to the previous element, if CT > CMC, set CMC equal to the value of CT, reset CT to 1, and then set CM equal to the most recently encountered element value.

(3) If CMC < $|S|/2$, then return the value "nonatomic" (i.e., having no elements derived from an atom). Otherwise, return the value in CMV as the atom-value.

By this process, the CMC register will end up containing the size of the largest subset of
elements of $S$ that have a common value, and the CMV register will contain that common
g value. Step (2) clearly takes $O(|S|)$ time. Thus this whole subroutine takes $O(|S|)$ time
per execution.

\subsection{Performance of Segment-Sort1}

Let $f$ be an arbitrary AB-Distribution, and let $S_n$ be a population of $n$ elements randomly
chosen from the distribution $f$. As indicated above, part (A) of any AB-distribution is
a probability density function, $g$, scaled by a proper fraction such that $g(x) \log_2 g(x)$
integrates to a finite value, and part (B) of any AB-distribution is a collection of atoms.
Let $T_n = t(S_n)$ be the expected time taken by Segment-Sort1 to sort $S_n$. I prove here
that there exists a positive real constant $k$ — not dependent upon $f$ — such that for all
sufficiently large $n$, $T_n \leq kn$; i.e., that:

$$T_n = O(n).$$

(1)

Let $U_n$ be the expected total time taken by step (4) of Segment-Sort1 in processing
$S_n$. Each of the first three steps and the fifth step of Segment-Sort1 requires $O(n)$ time
(where $n$ is the number of input elements) so $T_n = U_n + O(n)$. Thus it will be sufficient
to show that there exists a positive real constant $k$ — not dependent upon $f$ — such
that if $n$ is sufficiently large, $U_n \leq kn$; i.e., that:

$$U_n = O(n).$$

(2)

The first three steps of Segment-Sort1 effect a double-bucketing [Dev86] of the input.
Step (4) consists in applying the Atom-Handler routine to each of the buckets formed
in this double-bucketing process, and therefore, the total fraction of Segment-Sort1's
running time for which step (4) is responsible is essentially that for which the Atom-
Handler routine is responsible (ignoring the negligible time contribution of delays between
the calling and the actual beginning of the Atom-Handler routine, etc.) In other words,
we assume for the purposes of this analysis that the following condition holds:

**Condition 0:** for any positive integer \( n \): \( U_n \) equals the expected total time devoted by
Segment-Sort1 to the Atom-Handler routine.

The Atom-Handler routine — when called from Segment-Sort1 — subpartitions each
of these secondary buckets large enough to require subpartitioning and containing “atomic”
elements into three “tertiary” buckets, one of which is reserved for the elements derived
from part (B) of \( f \) (i.e., the atomic elements). If a given secondary bucket which the
Atom-Handler routine subpartitions contains elements derived from more than one atom,
it will not — as described above — succeed in separating the elements of that bucket
which part (B) of \( f \) is responsible for from those which part (A) is responsible for. How­
ever, by **Lemma 3** (below), if \( n \) is sufficiently large, no such “multi-atom” secondary
buckets will be passed to the Atom-Handler routine.

Thus the Atom-Handler routine effectively separates those input elements for which
part (A) is responsible from those for which part (B) is responsible. Once the atomic
elements of a “monatomic” secondary bucket have been deposited in their own tertiary
bucket, they — being all equal in value — require no further sorting, and all that remains
for Atom-Handler to do with the secondary bucket’s elements is the final sorting of those
elements derived from part (A) of \( f \).

**Lemma 3** If \( n \) is sufficiently large, no multi-atom buckets will be passed to the Atom-
Handler routine during the sorting of \( S_n \) by Segment-Sort1.

**Proof:**
Since the number of distinct atoms in the probability distribution $f$ is a finite number, there exists a minimal distance, $d$, between any two consecutive atoms of $f$. The distribution $f$ is an AB-distribution, so it is effectively confined to the interval $[0,1]$. Clearly, if we take $N = \lceil 1/d \rceil$, when $n > N$, no primary bucket will be wide enough to encompass two atoms.

The Expected Running Time of the Atom-Handler Routine

By Lemma 3, if $n$ is sufficiently large, not a single multi-atom bucket gets passed to the Atom-Handler routine. Consequently, I assume, for the purposes of this analysis, that every bucket passed to the Atom-Handler routine is either monatomic or nonatomic.

Let $B_0$, then, be a (monatomic or nonatomic) bucket created as an end product of the double-bucketing which occurs in steps (2) and (3) of Segment-Sort 1 and passed as input to the Atom-Handler routine in step (4) of Segment-Sort 1; and let $|B_0|$ be the number of elements in $B_0$. The Atom-Handler routine itself consists of seven steps. With $B_0$ as the input bucket under consideration, the following condition holds:

**Condition 1:** the first four steps and the seventh step of Atom-Handler individually, and thus, collectively, require $O(|B_0|)$ time per execution.

The two remaining steps, (5) and (6), do the bulk of Atom-Handler’s work. In step (5) (which is done iff $B_0$ is monatomic) the elements of $B_0$, as described above, are placed in one of three buckets accordingly as their values are either less than, equal to, or greater than that of $B_0$’s atom. This obviously requires only a single reading, comparison and writing for each element of $B_0$; which implies the following condition:

**Condition 2:** step (5) of Atom-Handler requires $O(|B_0|)$ time per execution.
In step (6), the elements of $B_0$, having already been loaded into tertiary buckets if necessary, undergo Segment-Sort1's final phase of sorting. If $B_0$ is monatomic, the atomic elements in the tertiary bucket designated for these need no further sorting, and if $B_0$ is nonatomic, there are no atomic elements to sort; so Atom-Handler devotes the remainder of its processing time (discounting the negligible time for step (7)) to the final sorting of the nonatomic elements of $B_0$, i.e., the elements of $B_0$ derived from part (A) of $f$. This remaining processing time is precisely that required by step (6).

According to the analysis in Section 1.5 of [Dev86], an elementary double-bucketing sort-routine using a quadratic-time sort-routine at the lowest level can sort $n$ elements — randomly selected from an arbitrary p.d.f. whose square integrates to a finite number — in $O(n)$ expected time. If, in the analysis given there, for each terminal bucket, $B$, the assumed bottom-level quadratic-time sort-routine is replaced by a standard sort-routine, the time-complexity of $|B|^2$ for the final sorting of $B$ is correspondingly replaced by a time complexity of $O(|B| \log_2 |B|)$. If a corresponding analysis is then carried through under this new assumption about bottom-level sorting, the result is that an elementary double-bucketing sort-routine using a standard sort-routine at the lowest level can sort $n$ elements — randomly selected from an arbitrary p.d.f. $g$ for which $g(x) \log_2 g(x)$ integrates to a finite number — in $O(n)$ expected time. The secondary buckets passed to the Atom-Handler routine in step (4) of Segment-Sort1 are the end products of an elementary double-bucketing, and part (A) of $f$ is a p.d.f. $(g)$ for which $g(x) \log_2 g(x)$ integrates to a finite number. Therefore, if Atom-Handler were to receive as inputs only those net portions of its actual input buckets deriving from part (A) of $f$, and proceed to sort these net portions without further partitioning — using a standard sort-routine — it would completely sort all these contributions from part (A) of $f$ in an expected time no greater than $kn$ where $k$ is a distribution-independent constant.

The buckets that get sorted in step (6) of Atom-Handler are formed by a further
partitioning of the net portions derived from part (A) of the secondary buckets received by Atom-Handler as input — and are thus smaller than these net portions. Their collective sorting (by a standard sort-routine) will therefore require at most twice as much time as that of these net portions derived from part (A). Let $V_n$ be the expected total time taken by step (6) of the Atom-Handler routine over the full course of Segment-Sort1’s execution in processing the population $S_n$ of $n$ elements randomly chosen from the AB-distribution $f$. Then by Theorem 1.5 in [Dev86]:

\[ V_n = O(n). \] (3)

Let $W_n$ be the expected total time taken by the first four steps and the seventh step of Atom-Handler over the full course of Segment-Sort1’s execution in processing $S_n$. Then it follows immediately from Condition 1 that:

\[ W_n = O(n). \] (4)

Let $X_n$ be the expected total time taken by the fifth step of Atom-Handler over the full course of Segment-Sort1’s execution in processing $S_n$. Then by Condition 2:

\[ X_n = O(n). \] (5)

It follows from Condition 0 that $U_n = V_n + W_n + X_n$. This together with equations (3), (4) and (5) gives us equation (2). Finally, equation (1) follows since we also have $T_n = U_n + O(n)$.

This completes the proof that Segment-Sort1 will sort random selections from an arbitrary AB-distribution in distribution-independent linear expected time.
2.1.4 Resolving Singularities in Linear Expected Time

Ordinary Double-Bucketing

It is known [DK81] that for any probability density function, $f$, if ordinary single-bucketing with a bottom-level standard sort-routine is used to sort a collection, $S_n$, of $n$ elements randomly drawn from $f$, then the expected time taken by such an algorithm to sort $S_n$ will be asymptotically bounded above by some function of the form $M n (\log_2 n)$ where $M$ is real, if and only if

$$\int_{\mathbb{R}} f(x) \log_2 f(x) \, dx < \infty.$$ 

In this section, it is shown that, unlike single bucketing, double-bucketing will yield such expected time performance in certain cases where the corresponding integral diverges, i.e. there is a probability density function, $f$, such that

$$\int_{\mathbb{R}} f(x) \log_2 f(x) \, dx = \infty.$$ 

but (nonetheless) if ordinary double-bucketing with a bottom-level standard sort-routine is applied to a collection of $n$ elements randomly drawn from $f$, then the expected time taken to sort the $n$ elements will be less than $M n (\log_2 n)$ for some positive real constant $M$.

Let $\mathbb{N}$ be the set of all nonnegative integers (i.e., $\mathbb{N} \equiv \{0, 1, 2, \ldots \}$). Let $a_f \equiv 1/((1/2) + \ln 2)$. Now, let $f : \mathbb{R} \rightarrow \mathbb{R}$ be defined by the equations:

$$f(x) = \frac{a_f}{x(\log_2 x)^2} \text{ if } 0 < x \leq 1/2,$$

$$f(x) = a_f \text{ if } 1/2 < x \leq 1, \text{ and}$$

$$f(x) = 0 \text{ otherwise.}$$
Then \( f \) is nonnegative real-valued everywhere, and (as shown below) \( f \) integrates to unity, so \( f \) is a probability density function. Finally, for any Riemann-integrable nonnegative real-valued function \( g \) on \( \mathbb{R} \) such that \( \int_{\mathbb{R}} g(x) \, dx \) is finite, I define the probability distribution of \( g \)'s shape, \( g_p \), as the unique probability density function for which there is a constant \( c \) such that for every real number \( x \), \( g_p(x) = cg(x) \).

The following theorem is a formal statement of the fact that double-bucketing, unlike single-bucketing, yields linear expected time sorting of data randomly drawn from \( f \).

**Theorem 4** A population of \( n \) elements randomly drawn from the probability density function \( f \):

(1) can be sorted in \( O(n) \) expected time by ordinary double-bucketing with a standard sort-routine applied to the final buckets thereby generated; and

(2) cannot be sorted in \( O(n) \) expected time by ordinary single-bucketing with a standard sort-routine applied to the final buckets thereby generated.

**Proof:**

The second assertion of this theorem follows immediately from Devroye and Klincsek's results [DK81] one consequence of which is that: For any probability density function \( f : \mathbb{R} \to \mathbb{R} \) with bounded support, if \( \int_{\mathbb{R}} f(x) \log_2 f(x) \, dx \) diverges then, given a random selection of \( n \) elements from \( f \) as input, the expected running time of a single-bucketing algorithm using a standard sort-routine at the lowest level is \( \omega(n) \).

Let \( n \) be any positive integer and let \( S_n \) be a population of \( n \) input elements randomly drawn from \( f \). A standard sort-routine runs in \( \Theta(k \log_2 k) \) expected time on an input set of size \( k \), so let \( h(x) \) be any real-valued function on \( \mathbb{R}^+ \) which is \( \Theta(x \log_2 x) \). If single
bucketing with a "bottom-level" sort-routine that runs in $h(k)$ expected time on an input set of size $k$ is applied to $S_n$, the expected time required for this single-bucketing sort will be $\omega(n)$ if $\int_R f(x) \log_2 f(x) \, dx$ diverges. Since $h(x)$ is $\Theta(x \log_2 x)$, this happens iff $\int_R h(f(x)) \, dx$ diverges. $\int_R h(f(x)) \, dx$ does indeed diverge, hence this theorem's second assertion.

Now (to prove the theorem's first assertion), suppose that ordinary double-bucketing with a standard sort-routine applied to the final buckets is used to sort $S_n$. To begin with, let $I \equiv \int_0^{1/2} f(x) \, dx$ and for every ordered pair $(b, c)$ such that $0 < b < c \leq 1/2$, let $I_{b,c} \equiv \int_b^c f(x) \, dx$. For any such $b$ and $c$:

$$I_{b,c} = \int_b^c \frac{af}{x(\log_2 x)^2} \, dx =$$

$$(\ln 2)^2 a_f \int_b^c \frac{1}{x(\ln x)^2} \, dx = (\ln 2)^2 a_f \int_{x=b}^{x=c} \frac{1}{[(\ln x)^2]} \, d(\ln x) =$$

$$(\ln 2)^2 a_f \int_{u=\ln b}^{u=\ln c} u^{-2} \, du = -(\ln 2)^2 a_f u^{-1}|_{\ln b}^{\ln c} = (\ln 2)^2 a_f \left(\frac{1}{\ln b} - \frac{1}{\ln c}\right).$$

Holding $c$ fixed while decreasing $b$, we find that $(1/\ln b) - (1/\ln c)$ approaches $-1/\ln c$ as $b$ approaches zero; therefore,

$$\int_0^c \frac{af}{x(\log_2 x)^2} \, dx \equiv \lim_{b \to 0} (I_{b,c}) = -\frac{(\ln 2)^2 a_f}{\ln c}.$$ 

Therefore
Chapter 2. Fast Deterministic Sorting

\[ I \equiv \lim_{x \to 0} (I_{x,1/2}) = \]

\[ -\frac{(\ln 2)^2 a_f}{\ln (1/2)} = a_f \ln 2, \]

so

\[ \int_{\mathbb{R}} f(x) \, dx = I + \int_{1/2}^{1} f(x) \, dx = \]

\[ (a_f \ln 2) + \frac{a_f}{2} = (\ln(2) + 1/2)a_f = \frac{\ln(2) + 1/2}{\ln(2) + 1/2} = 1, \]

which shows that \( f \) is, indeed, a p.d.f.

Now, consider the number of elements of \( S_n \) _expected_ to fall into the primary bucket-interval nearest to zero when ordinary single-bucketing (with \((0, 1]\) regarded as the full domain of possible input values) is applied to \( S_n \). This bucket-interval \((I_1)\) is \((0, 1/n]\), the first subinterval formed by partitioning \( f \)'s effective domain \(((0, 1])\) into \( n \) segments of equal width. The _proportion _of the elements of \( S_n \) expected to fall into \( I_1 \) will therefore equal

\[ \int_{0}^{1/n} f(x) \, dx \equiv \lim_{x \to 0} (I_{x,1/n}) = \]

\[ [1/(\frac{1}{2} + \ln 2)][ - 1/\ln (1/n)] = \]

\[ [1/(\frac{1}{2} + \ln 2)](1/\ln n) \text{ provided that } n > 2. \]
Therefore, the actual number of elements of \( S_n \) expected to fall into \( I_1 \) approaches \( \frac{1}{(\frac{1}{2} + \ln 2)}(n/\ln n) = O(n/\ln n) \) as \( n \) approaches infinity. Consequently, by the Logarithm Limit Rule, the expected time taken to sort the contents of the bucket corresponding to \( I_1 \) using a standard sort-routine will be \( O((n/\ln n) \log_2(n/\ln n)) = O(n) \).

Now, for each integer \( j \) such that \( 1 \leq j \leq [n/2] \), let \( I_j \) be the \( j \)th primary bucket-interval in the partition of \( (0,1] \) induced when single-bucketing is applied to \( S_n \) (i.e., the subinterval \((j - 1)/n, j/n]\)). The function \( f \) is monotonically decreasing and positive-valued on \((0, 1/2]\), so for every \( j \) such that \( 1 < j < \lfloor n/2 \rfloor \), \( 0 < f(j/n) < f((j - 1)/n) \).

In particular,
\[
f(j/n) = \frac{1}{(\frac{1}{2} + \ln 2)}(n/j)[ - 1/ \log_2 (j/n)]
\]
and
\[
f((j - 1)/n) = \frac{1}{(\frac{1}{2} + \ln 2)}(n/(j - 1))[ - 1/ \log_2 ((j - 1)/n)]
\]
so,
\[
1 < \frac{f((j - 1)/n)}{f(j/n)} = \frac{[j/(j - 1)]]}{\log_2 (j/n)/ \log_2 ((j - 1)/n)} < [j/(j - 1)] < 2 \text{ (for } j \geq 2).\]

That is, the value of \( f \) at the left end of any given primary bucket-interval beyond the first (up to primary interval number \( \lfloor n/2 \rfloor \)) is between one and two times as great as the value of \( f \) at the right end of the same interval. Since \( f \) is monotonic on \((0, 1/2]\), it follows that the value of \( f \) at any point in a given primary bucket-interval from the second to the \( (\lfloor n/2 \rfloor) \)th is between one and two times as great as the value of \( f \) at the
right end of the same interval. Therefore the following condition holds:

*Condition 3*: Given ordinary double-bucketing of a random selection of \( n \) elements from \( f \), each primary bucket-interval from the second to the \( ([n/2])^{th} \) inclusive will be the domain of some restriction, \( r \), of \( f \) having so moderate a peakedness [Dev86] that a random selection of \( k \) elements from the probability distribution of \( r \)'s shape can be sorted in \( O(k) \) expected time by a single (additional) phase of bucketing.

Since the proportion of elements expected to fall into primary bucket-interval number \( \lfloor n/2 \rfloor + 1 \) approaches zero as \( n \) approaches infinity we also have the following simple fact:

*Condition 4*: Primary bucket-interval number \( \lfloor n/2 \rfloor + 1 \) may be neglected in this time-complexity analysis.

Finally, since \( f \) is constant on each primary bucket-interval beyond number \( \lfloor n/2 \rfloor + 1 \), the following condition holds:

*Condition 5*: Each primary bucket-interval beyond number \( \lfloor n/2 \rfloor + 1 \) is the domain of a uniform subdistribution (i.e., constant-valued restriction), \( r_c \), of \( f \) which is such that a random selection of \( k \) elements from the probability distribution of \( r_c \)'s shape can be sorted in \( O(k) \) expected time by a single additional phase of bucketing.

If the single-bucketing responsible for generating the primary buckets corresponding to \( I_1, I_2, \cdots \) is just the first phase of an ordinary double-bucketing procedure, every primary bucket containing sufficiently many elements will be partitioned into secondary buckets. As indicated above, the contents of \( I_1 \) may be sorted in \( O(n) \) time by a standard sort-routine alone, so in the case of an ordinary double-bucketing procedure where a standard sort-routine is used to sort the final buckets, the total time devoted to the sorting of
Chapter 2. Fast Deterministic Sorting

$I_1$'s contents is $O(n)$. Moreover, it follows from Conditions 3, 4 and 5 that in this case, the total expected time required to sort all those elements that fall into primary buckets whose corresponding bucket-intervals lie beyond $I_1$ will be $O(n)$. Therefore, the total expected time required to sort all elements of $S_n$ via ordinary double-bucketing is $O(n) + O(n) = O(n)$.

Double-Bucketing with Shrinking Bucket-Intervals

Some probability densities have singularities too severe to be resolved by double-bucketing in linear-time when the bucket intervals designated in each application of the inherent single-bucketing algorithm are of equal width. In many such cases it is still possible to sort a random selection of $n$ elements from the given probability density function in $O(n)$ time by choosing bucket-intervals whose widths contract in the direction of each singularity (within a suitable neighborhood of it) at an appropriate rate. To sort in linear-time by applying such a technique, it is, of course, necessary either to know in advance the location of every singularity that warrants a sequence of contracting buckets or to locate all such singularities (with adequate precision) in $O(n)$ time. This section focuses on resolving a target singularity once its location is already known and assumes, without loss of generality, that this target singularity lies at zero.

Since it is possible to have a singularity whose c.d.f. $(F(x))$ steepens arbitrarily quickly as $x$ approaches zero, it may be necessary and sufficient (for linear-time resolution of the singularity by the kind of technique mentioned in the preceding paragraph) to assign a sequence of bucket-intervals that narrow as an arbitrarily fast-growing function of the sequence index. For example, a sequence of bucket-interval widths such as $1/2$, $1/(2^2)$, $1/(2^{2^2})$, ..., may be appropriate; or even a sequence for which the $j^{th}$ term is less than $1/A(j,j)$, where "A" stands for Ackermann's function. This section illustrates the use of bucket-interval contraction with a simple example, briefly outlining
a linear-time technique for partitioning an input set into buckets whose corresponding intervals shrink in accordance with the geometric series \( \{2^{-j}\} \); defining two probability density functions with singularities that can be resolved in linear-time by a form ("EDB-Sort") of double-bucketing with such geometric contraction in the first bucketing phase, but are not amenable to linear-time resolution by ordinary double bucketing; and then proving that EDB-Sort resolves any member of \( S_E \) in linear expected time but that sorting by ordinary double-bucketing does not.

Given an input population of \( n \) real numbers, \( S_n \), belonging to the interval \((0, 1]\), EDB-Sort first partitions these numbers into primary buckets by placing all input numbers belonging to \((2^{-j}, 2^{1-j}]\) into the bucket whose index is \( j \), for \( 1 \leq j \leq n - 1 \) (and placing all numbers belonging to \((0, 2^{-j}]\) in the bucket indexed by \( n \)). Except for this first phase of bucketing, EDB-Sort is identical to a sort by ordinary double-bucketing. Beyond its first bucketing phase, it has a second bucketing phase in which each primary bucket, \( B \), of size \(|B|\), where \(|B|\) is sufficiently large, is partitioned into a set of \(|B|\) secondary buckets whose intervals are of equal width. Then, a bottom-level phase of sorting by a standard sort-routine is applied in which the contents of each nonempty terminal bucket are sorted, and, finally, the elements are output, fully ordered, to a storage device. Determining which primary bucket a given input element, \( x \), belongs to requires only the (constant-time) computation of \( \text{Negate}(\text{Point-Normalize}(\text{Binary-Floor}(x))) \). The value of this expression is the index of the primary bucket into which \( x \) should be placed. Thus, EDB-Sort takes \( O(n) \) time provided that the actual net time devoted to the bottom-level brute-force sorting is \( O(n) \).

The two probability densities defined below (as stated formally in the next theorem) are resolvable in linear expected time by EDB-Sort but not by ordinary double-bucketing:

1. Let \( g : \mathbb{R} \to \mathbb{R} \) be defined by the equations:
Chapter 2. Fast Deterministic Sorting

\[ g(x) = \begin{cases} 
1 & \text{if } 0 < x \leq 1/2, \\
0 & \text{otherwise.}
\end{cases} \]

Let \( g_p \) be the probability distribution of \( g \)'s shape.

\[
\int_{\mathbb{R}} g(x) dx = 1/2 + \int_0^{1/2} \frac{1}{x(-\log_2 x)^{3/2}} dx = \\
(1/2) + (\ln 2)^{3/2} \int_0^{1/2} \frac{1}{x(-\ln x)^{3/2}} dx = \\
(1/2) - (\ln 2)^{3/2} \left[ \lim_{a \to 0} \left( \int_{-\ln a}^{\ln 2} u^{-3/2} du \right) \right] = \\
(1/2) + 2(\ln 2)^{3/2}((\ln 2)^{-1/2} - 0) = (1/2) + 2 \ln 2.
\]

Therefore \( g_p = \left[ \frac{1}{(1/2) + 2 \ln 2} \right] g \).

(2) Let \( h : \mathbb{R} \to \mathbb{R} \) be defined by the equations:

\[
h(x) = \begin{cases} 
1 & \text{if } 0 < x \leq 1/4, \\
0 & \text{otherwise.}
\end{cases}
\]

\[
h(x) = \frac{1}{x(-\log_2 x)[\log_2(-\log_2 x)]^{5/2}} \]

if \( 0 < x \leq 1/4, \)
Chapter 2. Fast Deterministic Sorting

\[ h(x) = 1 \text{ if } 1/4 < x \leq 1, \text{ and} \]

\[ h(x) = 0 \text{ otherwise.} \]

Let \( h_p \) be the probability distribution of \( h \)'s shape.

\[
\int_{\mathbb{R}} h(x) dx = 3/4 + \int_{0}^{1/4} \frac{1}{x(-\log_2 x)[\log_2(-\log_2 x)]^{5/2}} dx =
\]

\[
(3/4) + (\ln 2)^{7/2} \int_{0}^{1/4} \frac{1}{x(-\ln x)[\ln(-\ln x) - \ln(\ln 2)]^{5/2}} dx =
\]

\[
(3/4) - (\ln 2)^{7/2}[\lim_{a \to 0} \int_{\ln(-\ln a) - \ln(\ln 2)}^{\ln(\ln 4) - \ln(\ln 2)} u^{-5/2} du] =
\]

\[(3/4) + [2(\ln 2)^{7/2}/3][\ln(\ln 4) - \ln(\ln 2)]^{-3/2} = (3/4) + (2/3)(\ln 2)^2.\]

Therefore \( h_p = [1/(3/4 + (2/3)(\ln 2)^2)]h. \)

**Theorem 5** For any input size \( n \), a population of \( n \) elements randomly drawn from either \( g_p \) or \( h_p \):

1. can be sorted in \( O(n) \) expected time by double-bucketing with geometrically shrinking primary bucket-intervals and a standard sort-routine applied to the final buckets thereby generated (i.e., by EDB-Sort); and
(2) cannot be sorted in $O(n)$ expected time by either ordinary single-bucketing or ordinary double-bucketing with a standard sort-subroutine applied to the final buckets thereby generated.

Proof:

Consider first the probability density function $g_p$. Let $a_g = 1/((1/2) + 2 \ln 2)$. Then, on the interval $(0, 1/2]$, $g_p x = a_g / ((-\log_2 x)^{3/2})$, which is clearly monotonically decreasing on this interval. Therefore, for any $a$ and $b$ such that $0 < a < b < 1/2$, the ratio of $1/a$ to $1/b$ will be at least as great as the ratio of $g_p(a)$ to $g_p(b)$. When geometrically contracting bucket-intervals are used as indicated in the description of EDB-Sort above, the ratio of the reciprocal of the lower endpoint of the smaller to that of the lower endpoint of the larger of any two consecutive bucket-intervals will be two. It therefore follows that the ratio of maximum to minimum probability density for $g_p$ attained within any of the primary bucket-intervals lying between zero and $1/2$ is at most two. From this, it follows (by an argument analogous to that which applied Conditions 3 through 5 of the preceding section) that in the case of an input population of $n$ elements randomly drawn from $g_p$, the contents of all EDB-Sort-generated primary buckets beyond the first may be sorted in a total of $O(n)$ expected time by a second phase of bucketing followed by bottom-level standard sorting. The first primary bucket interval is $(0, 2^{-n}]$, and if you integrate $g_p$ from zero to $2^{-n}$, multiply by $n$, and then multiply the resulting product by its logarithm to get an expression for the expected time for standard sorting of its contents, you will find that EDB-Sort will process the contents of the first bucket in $O(n)$ expected time as well. Therefore, the total expected time for an EDB-Sort to process a random selection of $n$ elements from $g_p$ is $O(n)$. The proof that ordinary double-bucketing will not yield $O(n)$ expected time sorting of a random selection of $n$ elements from $g_p$ is equally straightforward. The first bucket's interval will extend at least from zero to $1/n^2$ when ordinary double-bucketing is used. Integrating $g_p$ from zero to $1/n^2$, multiplying
by \( n \), and then multiplying by the resulting product's logarithm yields an expression for the expected time for standard sorting of the first bucket's contents which is clearly \( \omega(n) \).

An argument essentially identical to that given here for \( g_p \) shows that this theorem's two conditions hold for \( h_p \) as well.

It is, in fact, possible, by the similar reasoning, to demonstrate that the two conditions of this theorem hold for all functions in an (uncountably) infinite class of functions having the form:

\[
\frac{1}{x(- \log_2 x)^{(1+\epsilon)} \prod_{j=1}^{k} \left[ \log_2^{(j)}(- \log_2 x) \right]^{b_j}}
\]

with \( k \geq 0 \) and \( \epsilon, b_1, \ldots, b_k \) in \((0, 1)\).

### 2.2 Additional Improvements on the Comparison-Based Time Complexity for Sorting

#### 2.2.1 Infinite Collections of Atoms

Another useful property of the Atom-Handler routine described above is that it may be used to sort size-\( n \) random selections from domain-bounded but otherwise arbitrary atomic distributions in \( o(n \log_2 n) \) expected time. A simple algorithm, Bucket-Sort/Test, that does just this is outlined below.
Bucket-Sort/Test(Input_File, Output_File)

(1) Apply the first two steps of Segment-Sort1.

(2) To each bucket formed in the preceding step, apply the Atom-Handler algorithm — passing as parameters the location of the bucket and the number of its elements. This step is the same as Step (4) of Segment-Sort1, but is applied here to primary buckets generated by just one level of ordinary bucketing.

(3) Read the sorted elements in order, from the first element in the first bucket to which Atom-Handler returns a pointer to the last element in the last such bucket, into Output_File.

Thus, let $f$ be an arbitrary probability distribution made up entirely of atoms with values in $[0,1]$. The atoms of $f$ may, in this case, be infinite in number and may even be dense on the interval $[0,1]$. The following nonetheless holds:

**Theorem 6** For every integer $n > 0$, let $P_n$ be a population of $n$ elements randomly chosen from $f$. Then any $P_n$ may be sorted by Bucket-Sort/Test in $o(n \log_2 n)$ expected time.

**Proof:** Let \( \{a_m\} \) be the sequence formed by taking the atoms of $f$ in order of decreasing weight (i.e., probability of occurrence). For any positive integer $m$, Let $a_m$ be the $m^{th}$ term of $\{a_m\}$, let $w(a_m)$ be the weight of the atom $a_m$, and let $A_m$ be the sum of the weights of terms $a_1, \ldots, a_m$. Since $f$ is a probability distribution made up of atoms, the total weight of all its atoms is equal to one. Therefore, the sequence of sums $\{A_m\} \equiv A_1, A_2, \ldots$ converges to one. This is equivalent to the condition:
For every $\epsilon > 0$ there is an integer $N$ such that for all integers $n > N$, $|A_n - 1| < \epsilon$.

Now, for every positive integer $n$, for every positive integer $j \leq n$, let $I_{j,n} = [(j - 1)/n, j/n)$. ($I_{j,n}$ is the subinterval of $[0,1]$ corresponding to the $j^{th}$ bucket formed in the single-bucketing phase of any execution of Bucket-Sort/Test on an input set of size $n$.) Also, let $M$ be an arbitrary fixed positive integer, and for every positive integer $n$, let $J(n)$ be the integer such that the value of the atom $a_M$ belongs to the subinterval $I_{J(n),n}$. The widths of the intervals in the sequence $I_{J(1),1}, I_{J(2),2}, \ldots$ approach zero as $n$ approaches infinity. For each positive integer $n$, let $H(n)$ be the index of the largest (lowest indexed) term of $\{a_m\}$ other than $a_M$ within the subinterval $I_{J(n),n}$.

Since every other atom in the sequence $\{a_m\}$ is a finite nonzero distance from $a_M$, $H(n)$ approaches infinity as $n$ approaches infinity. For $H(n) > M$, the total weight of all atoms other than $a_M$ within the subinterval $I_{J(n),n}$ is less than $1 - A_{H(n) - 1}$ because these atoms form a subsequence of the sequence formed by discarding the first $H(n) - 1$ terms of $\{a_m\}$. It therefore follows that the total weight of all atoms other than $a_M$ within the subinterval $I_{J(n),n}$ approaches zero as $n$ approaches infinity. Consequently, the expected proportion of elements having the value of $a_M$ to fall into a primary bucket generated by Bucket-Sort/Test that corresponds to $I_{J(n),n}$ will approach 1 as $n$ approaches infinity, so the value of $a_M$ will ultimately emerge as the majority value for such a bucket in Bucket-Sort/Test's testing phase.

For each integer $n > 0$, let $B_n$ be a primary bucket generated by Bucket-Sort/Test that corresponds to $I_{J(n),n}$, and let $\epsilon(n)$ be the expected proportion of elements to fall into $B_n$ that do not have the value of $a_M$. Clearly $\epsilon(n)$ approaches zero as $n$ approaches infinity. Since $\epsilon(n)$ is the expected proportion of elements to fall into $B_n$ that Bucket-Sort/Test will ultimately sort using its bottom-level standard sort-routine (the rest being found to have the value of $a_M$ and counted in linear time), it follows that Bucket-Sort/Test will
sort the contents of $B_n$ in $o(|B_n| \log_2 |B_n|)$ expected time.

Since $a_M$ is an arbitrary term of $\{a_m\}$, what holds for $a_M$ will also hold for each term of $\{a_m\}$ up to $a_j$, given an arbitrarily large integer $j$. Therefore, by the additivity of expected values (i.e., $\sum_{j=1}^{k} \{E[x_k]\} = E[\sum_{j=1}^{k} x_k]$), it follows that for any fixed positive integer $j$, Bucket-Sort/Test will collectively process all input elements having a value equal to that of a member of the set $S_0 \equiv \{a_1, \ldots, a_j\}$ in an arbitrarily small fraction of $|S_0| \log_2 |S_0|$ expected time, given a sufficiently large input size $n$. Thus, let $\epsilon_0$ be any number such that $0 < \epsilon_0 < 1$. Let $C$ be the constant coefficient implicit in the asymptotic upper bound of $O(n \log_2 n)$ for the running time of the bottom-level standard sort-routine used in Bucket-Sort/Test on an input set of $n$ elements. Take $j$ sufficiently large that $1 - A_j < (\epsilon_0/(C+1))$. Then, for any integer $m > 1$, $m(1 - A_j) \log_2 (m(1 - A_j)) < m(1 - A_j) \log_2 m < (\epsilon_0/(C+1))m \log_2 m$. When the input size $n$ is sufficiently large, the expected time needed by Bucket-Sort/Test to collectively process those elements derived from atoms $a_1, \ldots, a_j$ will be less than $(\epsilon_0/(C+1))n \log_2 n$, and, of course, as the standard sort-routine used requires less than $Ck \log_2 k$ time to sort any $k$ ranked elements (given any integer $k > 1$), the expected time needed to process those elements derived from the remaining atoms ($a_{j+1}, \ldots$) cannot exceed $Cn(1 - A_j) \log_2 (n(1 - A_j))$. Therefore, the expected time needed by Bucket-Sort/Test to process the input elements derived from atoms $a_1, \ldots a_j, a_{j+1}, \ldots$ (i.e., all the input elements) is less than

$$(\epsilon_0/(C+1))n \log_2 n + Cn(1 - A_j) \log_2 (n(1 - A_j)) <$$

$$(\epsilon_0/(C+1))n \log_2 n + (C\epsilon_0/(C+1))n \log_2 ((C\epsilon_0/(C+1))n) <$$

$$(\epsilon_0/(C+1))n \log_2 n + (C\epsilon_0/(C+1))n \log_2 n = \epsilon_0 n \log_2 n.$$
2.2.2 Arbitrary Isolated P.D.F. Discontinuities

Let \( f(x) \) be a probability density function that is bounded on every closed interval which is a subset of \((0, 1] \). (As usual, it is assumed that all positive density is confined to the interval \([0,1] \).) There may be a discontinuity at \( x = 0 \) which results from singularity, divergence by oscillation, or any other possible cause. Let \( S_n \) be a population of \( n \) elements randomly drawn from \( f \). Then the following is true.

**Theorem 7** Let \( S_n \) be given as input to a sorting algorithm consisting of a single phase of ordinary bucketing and the subsequent application of an standard sort-routine to the buckets thus formed (i.e., an ordinary bucket-sort). Then the expected time for the sorting \( S_n \) is \( o(n \log_2 n) \).

**Proof:** After the initial phase of bucketing, for \( 1 \leq j \leq n \), the \( j^{th} \) bucket (in order of increasing element value) will correspond to the subinterval \( ((j - 1)/n, j/n] \). Since \( f \) is a probability density function with nonzero values confined to \((0, 1] \), for every \( \epsilon \in (0, 1] \) there is a (unique) \( \delta \in (0, 1] \) such that \( \int_{\delta_0}^{\delta} f(x) \, dx = \epsilon \). Let \( \epsilon_0 \) be any number in \((0, 1] \) and let \( \delta_0 \) be the number for which \( \int_{\delta_0}^{\delta} f(x) \, dx = \epsilon_0 \). Since \( \delta_0 > 0 \), the range of \( f \)'s restriction to \([\delta_0, 1] \) has an upper bound \( (U) \). Let \( B_0 \) be the bucket formed in the bucketing of \( S_n \) which corresponds to the bucket-interval containing \( \delta_0 \) \( ([n\delta_0] - 1)/n, [n\delta_0]/n) \). Let \( B_1 \) be any bucket thus formed whose corresponding bucket-interval lies to the right of \( B_0 \)'s within \((0, 1] \). Since \( f \) is upper bounded by \( U \) over \( B_1 \)'s bucket-interval, the expected number of elements from \( S_n \) to fall into \( B_1 \) is upper bounded by \( U \). Therefore, since a standard sort-routine is used for the final sorting of \( B_1 \)'s contents, the expected time for this final sorting of \( B_1 \)'s contents is \( O(U \log_2 U) \). Since \( B_1 \) is an arbitrary bucket whose corresponding interval lies beyond that of \( B_0 \), and there are at most \( n - 1 \) such buckets, it therefore follows that the expected time required for the final sorting of all the buckets
beyond $B_0$ is $O(nU \log_2 U) = o(n \log_2 n)$. Since $\delta_0 > 0$, it is clear that for sufficiently large $n$, the lower endpoint of bucket-interval corresponding to $B_0$ is strictly positive, from which it follows that the restriction of the range of $f$ to this interval will have an upper bound. Let $U'$ be such an upper bound. Then the expected number of elements from $S_n$ to fall into $B_0$ (for sufficiently large $n$) will be upper bounded by $U'$. The integral of the restriction of $f$ to any bucket-interval to the left of $B_0$'s bucket-interval must be at most $\epsilon_0$ because the interval must lie entirely between zero and $\delta_0$. It follows immediately that the expected number of elements of $S_n$ to fall into any bucket whose corresponding interval lies to the left of $B_0$'s is upper bounded by $\epsilon_0 n$. Therefore, the ratio of the expected time taken (by the standard sort-routine) for the final sorting of any such bucket's contents to the expected number of elements of $S_n$ to fall into the bucket is $O(\log_2 (\epsilon_0 n)) = O(\log_2 n)$. Since the expected number of elements to fall into the union of all the buckets with corresponding intervals to the left of $B_0$'s is at most $\epsilon_0 n$, it follows that the total expected time for the final sorting of the buckets whose corresponding intervals lie to the left of $B_0$'s is $O(\epsilon_0 n \log_2 n)$. Therefore, the total expected time for the final sorting of all the buckets is $O(\epsilon_0 n \log_2 n) + O(U') + o(n \log_2 n) = O(\epsilon_0 n \log_2 n)$. Since $\epsilon_0$ may be arbitrarily small, the total expected time for this final phase of sorting is $o(n \log_2 n)$. Since the initial bucketing of the elements itself takes $O(n)$ time, the entire sorting of $S_n$ as described here takes $o(n \log_2 n)$ expected time.
In this final chapter, I present a surprisingly general and yet simple result concerning the Random RAM. It turns out that there are algorithms for the Random RAM by which one may sort $n$ distinct real numbers in $O(n)$ expected time regardless of how they are chosen, i.e., even if they are chosen by an enemy. A great deal of credit for this result must be given to Kirkpatrick and Reisch [KR84] for their algorithm by which, for any $m \in \mathbb{N}$, an arbitrary population of $m$ distinct integer values may be sorted in $O(m)$ absolute worst-case time. A slightly modified version of Kirkpatrick and Reisch’s linear-time integer-sorting algorithm, that handles duplicate integers by attaching to the end of each input element a unique index string, and achieves sorting of the real-valued inputs to Random-Segment-Sort by attaching them to integer-valued sort-keys, is used here as a subroutine for this chapter’s main algorithm, Random-Segment-Sort. The algorithm of Kirkpatrick and Reisch may be used for linear-time sorting of an arbitrary finite set of rational numbers, as these can be scaled up by their least common denominator, making them integers, and then scaled back down after sorting. By itself, it is, however, fundamentally incapable of sorting arbitrary real numbers in linear-time (as is immediately apparent from an inspection).

Random-Segment-Sort is presented in the usual pseudocode-like format. It is assumed, without loss of generality, that the input numbers lie within the interval $[0, 1]$, as their true domain can be computed and effectively reduced to this interval in $O(n)$ time by finding the extreme input values, and applying to all the input numbers a common
contraction (by shifting) and translation (by addition) that places these extreme values within [0, 1].

### 3.1 The Random-Segment-Sort Algorithm

Random-Segment-Sort, as mentioned above, is analyzed under the assumption that the input numbers are distinct. It will clearly have the same average-time performance in the natural context of input randomly drawn from an arbitrary p.d.f. (where the probability of two equal values is zero):

\[
\text{Random-Segment-Sort(Input\_File, Output\_File)}
\]

(1) Read the input from Input\_File into an input array (In), counting the number (n) of input elements.

(2) Letting N be the number of elements of the current input array (In\_c), set \( j \) equal to \( \text{Floor}(\text{Random}(0,N)) + 1 \). The memory variable \( j \) now contains a random index (randomly chosen from among all possible indices) for In\_c. At the “top,” or zeroth recursion level (where the \( j^{th} \) recursion level is the total sequence of all primitive instructions executed at a recursion depth of \( j \)), the current input array, In\_c, equals In and the number of elements in it is \( n \).

(3) Scan In\_c to find the value of the least of its elements strictly exceeding In\_c[\( j \)] in value, and set \( V \) equal to this value provided that this value exists. If such a value fails to exist (i.e., In\_c[\( j \)] is maximal within In\_c), return to step (2).

(4) At this point, the value of the expression \( | V - \text{In\_c}[j] | \) will be equal to the size of a randomly chosen “gap” between two consecutive values in the ascending linear order that
the input numbers comprise. Let \( \text{gap-size} \) be equal to the value of \( \text{Binary-Floor}(| V - \ln c_j |) \) and let \( \text{input-resolution} \) equal \( \frac{1}{\text{gap-size}} \). The values of \( \text{gap-size} \) and \( \text{input-resolution} \) may both be computed (in constant time) by \( \text{Point-Normalize} \), shifting, negation and addition operations. They will both be positive integral powers of two.

(5) Allocate the arrays \( \text{Integer-Parts}[1..N] \) and \( \text{Initial-Integer-Parts-Indices}[1..N] \). (This step need be nothing more than setting up pointers to the starting and ending points of the two arrays.)

(6) For every \( k \in \{1, \cdots, N\} \), set \( \text{Integer-Parts}[k] \) equal to the value of \( \ln c_k \), left-shift \( \text{Integer-Parts}[k] \) by a number of binary places equal to the value of \( \text{Point-Normalize}(\text{input-resolution}) \) (thereby effecting a multiplication by \( \text{input-resolution} \)), and then reduce the scaled-up value of \( \text{Integer-Parts}[k] \) to its integer part (with a floor operation).

(7) For every \( k \in \{1, \cdots, N\} \), set \( \text{Initial-Integer-Parts-Indices}[k] \) equal to \( k \). The value of \( \text{Initial-Integer-Parts-Indices}[k] \) now equals the initial \( \text{Integer-Parts} \) array-index \( (k) \) of the \( k^{th} \) element of the sequence comprised by the elements of \( \text{Integer-Parts} \) prior to their sorting.

(8) If \( N \leq 4 \), sort the elements of \( \text{Integer-Parts} \) using an insertion-sort [CLR90]. Otherwise, sort the elements of the \( \text{Integer-Parts} \) array in \( O(N) \) time using a version of Kirkpatrick and Reisch's "Algorithm IV" [KR84] modified to handle duplicate integers and carry satellite data as described here. Algorithm IV will always sort \( m \) arbitrary input integers in less than \( 30m \) operations (for any integer \( m \) exceeding 4). In the modified version used here, the input integers are initially left-shifted by \( \text{Binary-Ceiling}(N) \), and a unique index in the range \( 1, \cdots, N \) (padded on the left with zeroes to a length of \( \log_2(\text{Binary-Ceiling}(N)) \)) is attached to the end of each entry in the \( \text{Integer-Parts} \) array,
to distinguish duplicate integers. Also, the initial value of Initial-Integer-Parts-Indices\([k]\) for every \(k \in \{1, \cdots, N\}\) is treated as a satellite datum bound to the initial value of Integer-Parts\([k]\), so that the \(j^{th}\) term \((x_j)\) of the final sequence of elements in Initial-Integer-Parts-Indices will equal the index of the initial position corresponding to \(x_j\) in the Integer-Parts array.

(9) At this point, for every \(k \in \{1, \cdots, N\}\), the value of Integer-Parts\([k]\) equals the integer part of the value of In\(_c\)[Initial-Integer-Parts-Indices\([k]\)] scaled up by a factor of input – resolution. Arrange the elements of In\(_c\) so that their scaled-up integer parts are in ascending order as follows: allocate an auxiliary array, Aux-In\(_c\)[1..N]; for each \(k \in \{1, \cdots, N\}\), set Aux-In\(_c\)[\(k\)] equal to In\(_c\)[Initial-Integer-Parts-Indices\([k]\)]; and finally, for each \(k \in \{1, \cdots, N\}\), write the value of Aux-In\(_c\)[\(k\)] back into In\(_c\)[\(k\)].

(10) Allocate an array (Bucket\(_c\)[1..\(N\)]) to hold pointers to the beginnings of the “buckets” in In\(_c\), i.e., the maximal contiguous segments of In\(_c\) whose elements’ values all correspond to equal integers in Integer-Parts. Also, allocate an array (Bucket-Size\(_c\)[1..\(N\)]) to hold the sizes of these buckets.

(11) Let the number of “buckets” in In\(_c\) as described in step (10) be referred to as \(N'\). Read through Integer-Parts, and for each \(k \in \{1, \cdots, N'\}\), set Bucket\(_c\)[\(k\)] equal to the In\(_c\) array-index of the beginning of the \(k^{th}\) bucket (equal to the Integer-Parts array-index at which the \(k^{th}\) new value in Integer-Parts begins). Also, record the number of elements in each bucket (the “bucket size”) in Bucket-Size\(_c\) as you proceed.

(12) Deallocate the three auxiliary arrays Integer-Parts, Initial-Integer-Parts-Indices and Aux-In\(_c\).

(13) To each segment of the In array now corresponding to a bucket [delimited by pointers
created in step (11)] that contains more than one element, recursively apply steps (2) through (12) until the elements of In are fully ordered. The recursion should be applied to the buckets from left to right at each level.

(14) Write the $N$ elements of In, in order, into Output_File.

3.2 Analysis of Random-Segment-Sort

It can be verified by inspection that if Random-Segment-Sort terminates, it does so with the correct output, a linear ordering of the real numbers it receives as input. We will show that it runs in expected time $O(n)$, which implies that it terminates with probability one. The body of this analysis is thus devoted to the algorithm's outstanding feature, i.e., its linear expected time performance for distinct input numbers.

Steps (1) and (14) of the algorithm clearly take $O(n)$ time. I consider now the application of the component procedure ($P_s$) consisting of steps (2) through (12) to a “current segment” ($In_c$) of size $m > 1$ of the principal input array, In, which the algorithm uses for storage of the input at various stages of the sorting process.

Steps (2) and (3): Due to the storage of “bucket sizes” in step (11), the “size,” or number of elements in $In_c$ will always be immediately accessible. Consequently, step (2) takes constant time per execution. Step (3) also takes constant time per execution. The probability of returning to step (2) at the end of step (3) is only $1/m$, so steps (2) and (3) are both executed $O(1)$ times on average per execution of step (4) (which is only executed once at the top-level in each recursive call). Therefore, steps (2) and (3) collectively take $O(1)$ expected time per execution of step (4).
Steps (4) and (5): Step (4), as indicated, takes constant time per execution. Step (5) also takes constant time per execution and is performed only once per execution of step (4). Consequently these steps collectively require $O(1)$ time per execution of step (4).

Steps (6) through (9): These four steps each take $O(m)$ time per execution, and each is executed just once per execution of step (4). Therefore the time devoted to these four steps per execution of step (4) is $O(m)$.

Steps (10) through (12): Steps (10) and (12) clearly take constant time per execution. Step (11), which involves reading a size-$m$ array (Integer-Parts) and setting up pointers to a subset of its indices, takes $O(m)$ time per execution. Each of these three steps is executed exactly once per execution of step (4). Therefore the total time devoted to these three steps per execution of step (4) is $O(1) + O(1) + O(m) = O(m)$.

Thus the procedure $(P_s)$ consisting of steps (2) through (12) takes $O(m)$ time in a single application to a size-$m$ segment of the In array, and consequently takes $O(n)$ time in its application to the full In array at the top level.

In step (13), $P_s$ is recursively applied to each segment of In corresponding to a bucket of size at least two generated in the initial application of $P_s$ to In. At the top level in each recursive call, the gap size, $g$, ("$| V - \text{In}_c[j] |$" above) is, in effect, randomly chosen from among the sizes of the gaps between consecutive elements (currently held in $\text{In}_c$) in the final ordering of the input, so its expected rank among these gap sizes is that of median. The effective bucket-interval width, $w$, (referred to as "gap-size" above) is even smaller than $g$, and the relative ranking of $w$ among these gap sizes (expressed as a proper fraction) is directly proportional to the expected fraction of elements of $\text{In}_c$ that end up isolated in buckets of size 1 indexed by the $\text{Bucket}_c$ array. Therefore, at the top level in
Each recursive call, the expected proportion of isolated elements is at least $1/2$. That is, at least half the elements of $I_{n_c}$ are expected to be isolated in buckets of size 1 by a single application of $P_s$ to $I_{n_c}$, while at most half the elements of $I_{n_c}$ are expected to end up in multi-element buckets. Thus Random-Segment-Sort performs a linear expected number (i.e., $O(n)$) of primitive instructions excluding recursions, and calls itself recursively on subproblems of expected total size at most half the size of the input (within each recursive call). This implies that the expected running time of Random-Segment-Sort is $O(n)$.

It is, in fact, possible to guarantee the termination of Random-Segment-Sort without sacrificing its $O(n)$ average-time performance. Let Random-Segment-Sort$_0$ be an algorithm identical to the original Random-Segment-Sort, except that where the depth of the recursion within any recursive call of Random-Segment-Sort$_0$ is $n$, the subset of the original input passed to that call is submitted to a standard sort-routine for bottom level sorting. For any nonnegative integer $j$, let $r_j$ be the ratio of the fraction of original input elements that get isolated at a recursion depth of $j + 1$ or higher (in an application of Random-Segment-Sort$_0$ to a set of $n$ input elements) to the fraction of original input elements that get isolated at a recursion depth of $j$ or higher. Let $\epsilon$ be any number such that $0 < \epsilon < 1$. By the nature of the random choice of effective, relative bucket-interval widths (input scale-up factors) in Random-Segment-Sort$_0$, it is clear that, given values for $r_1, \ldots, r_j$, the probability that $r_{j+1}$ will exceed $1/2$ is less than $1/2$. That is:

$$\Pr\left( r_{j+1} > \frac{1}{2} \mid r_1, \ldots, r_j \right) < \frac{1}{2}. $$

Therefore, if $n$ is sufficiently large, the probability that with $n$ levels of recursion, the $r_i$'s for at least $(1 - \epsilon)n$ of these levels will exceed $1/2$ is less than
Chapter 3. Random Sorting of Distinct Real Numbers in Linear Expected Time

\[
2^{-(1-\epsilon)n}n \max\left(\left\lfloor \frac{n}{(1-\epsilon)n} \right\rfloor, \left\lfloor \frac{n}{(1-\epsilon)n} \right\rfloor \right) <
\]

\[
2^{-(1-\epsilon)n}n^{\epsilon n} = 2^{-(1-\epsilon)n}n^{\epsilon n+1}.
\]

Let \( h(n) \equiv 2^{-n/2} \) and let \( \epsilon_n \equiv 1/(\log_2 n)^2 \). Then the probability that with \( n \) levels of recursion, the \( r_i \)'s for at least \((1-\epsilon_n)n\) of these levels will exceed \( 1/2 \) is less than

\[
2^{-(1-\epsilon_n)n}n^{\epsilon_n n+1} = 2^{-(1-\frac{1}{\log_2 n})n} n^{\frac{n+\log_2 n}{\log_2 n}} <
\]

\[
2^{-(1-\frac{1}{\log_2 n})n} 2^{n\left(\frac{1}{\log_2 n}\right)} <
\]

\[
2^{n\left(\frac{1}{\log_2 n} + \frac{2}{\log_2 n} - 1\right)} < 2^{-n/2} \equiv h(n), \text{ for sufficiently large } n.
\]

If it is not the case that the \( r_i \)'s for at least \((1-\epsilon_n)n\) of levels exceed \( 1/2 \), then fewer than \((1-\epsilon_n)n\) of the \( r_i \)'s exceed \( 1/2 \), which implies that, given \( n \) recursive levels, at least \( \epsilon_n n = n/(\log_2^2 n) \) of the \( r_i \)'s are bounded above by \( 1/2 \). This condition implies that the total number of clustered input elements remaining has been divided by a factor of \( 2^{n/(\log_2^2 n)} \), which is much greater than \( n \) for large \( n \). It is impossible for the number of clustered elements remaining to be divided by more than \( n \). Therefore, when \( n \) is large, Random-Segment-Sort will terminate, before ever invoking its standard sort-routine,
Chapter 3. Random Sorting of Distinct Real Numbers in Linear Expected Time

with a probability of at least $1 - h(n)$. The probability that the standard sort-routine will be invoked is consequently bounded above by $h(n) \equiv 2^{-n/2}$, so even if it were to start from scratch on the entire original input, taking $\Theta(n \log_2 n)$ time, it would increase the expected running time of Random-Segment-Sort by at most $\Theta[(n \log_2 n)/(2^{n/2})] = O(1)$.

A Word on Space Requirements

Although time requirements are the principal focus of this analysis, it is worth noting that the expected number of memory words used by Random-Segment-Sort is also $O(n)$. This is evident from the following two facts:

(1) A single execution of any one of the Random RAM's primitive instructions affects only $O(1)$ memory words.

(2) Random-Segment-Sort performs a linear expected number of primitive instructions excluding recursions, and calls itself recursively on subproblems of expected total size at most half the size of the input, within each recursive call.

Since individual memory words can hold real numbers, this $O(n)$ "space" complexity does not have the same significance as it would if, for example, every memory word's contents could be represented with a fixed number of bits. For that reason, I prefer to call it "word" complexity.

The word and time complexities of Random-Segment-Sort are qualified by the fact that Kirkpatrick and Reisch's integer-sorting algorithm, used in step (8), requires a number of bits for intermediate results which is at least exponential in the total number of bits submitted to it as input (it is not a "conservative" algorithm). One may avoid this by adding
a bitwise AND and OR to the Random RAM's primitive instruction set, and substituting the conservative integer-sorting algorithm (signature sort) of Andersson, Hagerup, Nilsson and Raman [AHNR95] that runs in $O(n \log_2 \log_2 n)$ time, given $n$ input elements, for Kirkpatrick and Reisch’s algorithm in step (8) of Random-Segment-Sort. The result would be a modified version of Random-Segment-Sort having only slightly poorer expected time performance. By effecting an expected 50% or greater reduction, in going from each recursion depth to the next, in the number of input elements to be isolated at higher recursion depths, and having the asymptotic performance characteristic at any one recursion level indicated by the use of signature sort in step (8), this modified version of Random-Segment-Sort would achieve complete sorting of $n$ distinct real numbers in $O(n \log_2 \log_2 n)$ expected time. The constants associated with signature sort are also reasonably small.
Bibliography


