Fringe Analysis for Extquick: An \textit{in situ} Distributive External Sorting Algorithm

WALTER CUNTO

Centro Cientifico IBM, A.P. 64778, Caracas-1060, Venezuela

GASTÓN H. GONNET AND J. IAN MUNRO

Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

AND

PATRICIO V. POBLETE

Departamento de Ciencia de la Computación, Universidad de Chile, Blanco Encalada 2120, Santiago, Chile

A new \textit{in situ} external sorting algorithm, to be called Extquick, is developed and its time and space performance are analysed. It is shown that Extquick performs more efficiently than similar \textit{in situ} sorting algorithms based on Quicksort that appear in the literature. Since the computational tree of Quicksort-like sorting algorithms is equivalent to a search tree, techniques that model the time complexity of such a structure are then used for the analysis of Extquick. © 1991 Academic Press, Inc.

1. Introduction

Since its appearance in 1961, Quicksort has proven to be a time efficient algorithm which sorts \textit{in situ} arrays of elements in main memory (Hoare, 1961). This means that only logarithmic order extra space is required because of the stack handling for recursion. Different variations have been proposed, some of them improving the performance of the basic algorithm (Sedgwick, 1975; Singleton, 1969), and some others not being so successful in achieving the initially assumed improvement (Van Emde, 1970).

Quicksort algorithms are a special case of distributive sorting. The input set of elements is divided into two subsets such that the elements in one of the subsets are smaller than any element in the other subset. This partition process is recursively applied until the subsets are of a suitable small size and then sorted in one pass. In a more general fashion, distributive sorting algorithms partition the initial set into \( m \geq 2 \) relatively ordered subsets.
The partition is driven by a set of $m$ elements taken from the file. Samplesort (Frazer, 1970; Knuth, 1973b, 5.2.2) was one of the first algorithms using this approach. In this case, the parameter $m$ is defined to be nearly $N/\ln N$ such that the subfiles obtained by the partition process are of a size sortable in one pass.

Variations of Quicksort found in the literature differ in the partition scheme used. The standard algorithm splits the initial set by comparing the elements against a pivot element randomly chosen from the set (Hoare, 1961). An efficient variation uses the median of a small sample as the pivot element (Sedgwick, 1975; Singleton, 1969). Another scheme replaces the pivot element by a small sample of sorted elements which are updated during the scanning of the elements and whenever an element happens to fall between two of those sorted elements (Van Endem, 1970). Also, the elements guiding the partition process need not be in the set. In such cases, arithmetic averages can be used as pivots (Motzking, 1983). The handbook by Gonnet (1984) gives a comprehensive list of references on this specific subject.

The basic ideas of Quicksort have been adapted to the design of external sorting algorithms mainly because of its \textit{in situ} property (Cunto, 1984; Monard, 1980; Six and Wegner, 1982). All of them are external counterparts of internal quicksort algorithms (Van Endem, 1970; Motzking, 1983; Sedgwick, 1975).

In this paper, a simple external distributive sorting algorithm called Extquick is proposed and its time and space performance are analysed. It is found that this variation performs faster than previously reported external \textit{in situ} sorting algorithms (Cunto, 1984; Monard, 1980). The partition scheme in Extquick splits the file into $m$ subfiles taking the $m-1$ equidistant percentiles from a sample of size $km-1$. This variation generalises the idea of the median of a sample (Cunto, 1984; Singleton, 1969, Sedgwick, 1975). When $k-1$, Extquick becomes the external counterpart of Samplesort. For $m=2$, Extquick is the external counterpart of the Quicksort variation that chooses the median of a sample as the pivot element responsible for the partition process (Cunto, 1984). The description of the algorithm and details of its implementation are presented in the following section. Based on simulation results, guidelines on the choice of suitable values for the parameters $k$ and $m$ are provided in Section 3 and Section 4 discusses performance differences of our algorithm and External Quicksort (Gonnet, 1984; Monard, 1980).

The analysis of the proposed algorithm depends on the relation between Quicksort-like sorting algorithms and search trees (Knuth, 1973b; Poblete, 1985). Although most sorting algorithms of this kind do not build an explicit data structure, it is easy to observe that the recursive process of partitioning sets into subsets, defining the so-called \textit{computational tree}, is
actually a randomly generated search tree (order in the input data is assumed equiprobable). Therefore, Samplesort is equivalent to Unbalanced Multiway Trees (Knuth, 1973b, 6.4.2–10); External Quicksort (Gonnet, 1984; Monard, 1980) is similar to GBS-trees (Cunto, 1987), and the Quicksort variation whose pivot element is the median of a sample (Cunto, 1984) resembles the binary search trees with local reorganizations (Poblete, 1985).

The analogy is useful in the reverse direction. A related approach to Extquick has been adapted to Unbalanced Multiway trees resulting in a new data structure that in time and space performance competes with B-trees (Cunto, 1988).

We will prove in Section 5 that the time complexity of Extquick is asymptotically proportional to the element path of its corresponding computational tree. The element path of a tree (internal path in (Knuth, 1973a)) is the sum of element levels in it. This relation is also applicable to other Quicksort-like sorting algorithms. The unsuccessful search in search trees is an average over their inter-element-gap path (external path definition in (Knuth, 1973a)). However, both paths are linearly related; thus any technique used to model the inter-element-gap path in search trees can be adapted to model the element path of computational trees.

The fringe technique, based on Markovian processes, is a successful method for keeping track of changes in substructures at the deeper levels of dynamic trees, thus modelling the changes in the inter-element-gap path of search trees when inserting a new element in the structure (Cunto, 1987; Cunto, 1988; Poblete, 1985). The first reference suggesting the use of fringe techniques in the analysis of Quicksort-like sorting algorithms can be found in (Knuth, 1973b, 6.4.2–10). We formalise this usage and apply it to our sorting algorithm.

2. DESCRIBING EXTQUICK

Extquick recursively distributes a file into \( m \) subfiles. The partition process is driven by \( m - 1 \) pivots selected from a random sample of size \( km - 1 \). Subfiles are then processed by following the increasing sequence of subfile sizes. This approach bounds the size of the stack used for handling the recursion calls. If such care is not taken, it can be shown that the recursion stack may grow up to some proportion of the file size, a situation to be certainly avoided (see Section 5.1 for details). A brief description of Extquick is given in Fig. 1.

The computational model used to run our algorithm is based on a central processor, a controller driven disk, and a main memory. The most time consuming and asymptotically complex operation is that involving
procedure sort(X: file);
    if size(X) <= memory-size
        then
            read X into memory;
            sort the records in memory;
            write the records back to X
        else
            read from X a random sample S of \( km - 1 \) records;
            select from S the elements of rank \( ik \), \( 1 < i < m - 1 \)
            and denote the result \( x_1, x_2, \ldots, x_{km-1} \) such that \( x_i < x_{i+1} \);
            scan X and build the subfiles \( X_i = \{ x | x_{i-1} < x < x_i \}, i = 1, \ldots, m - 1 \);
            (*\( x_0 = -\infty \) and \( x_m = +\infty *\)*)
            let \( i_j \) be the index of the sequence such that size(\( X_{i_j} \)) \( \leq \) size(\( X_{i_{j+1}} \)), \( j = 1, \ldots, m - 1 \):
            for \( j := 1 \) to \( m \) do
                sort(\( X_{i_j} \))
            end_for
        end_if
    end_procedure.

FIG. 1. Extquick’s brief description.

data transfer between memory and disk. Data in memory are organised in buffers and data in disk are stored in tracks. This model describes computers ranging from personal workstations to large main-frames.

The disk can be viewed as a set of tracks of size \( t \). The track size \( t \) is the maximum number of fixed length records a disk track can store. Each read-from- or write-to-disk operation is time-equivalent to the latency of the disk device (time to scan a track). Thus, if improvement in the disk-memory data flow is sought, the block factor and the buffer size have to be set equal to \( t \).

The memory stores the program, the stack to handle the recursion, the array of \( m - 1 \) of pivots used for the partition process, and the buffers needed for I/O. Since the algorithm performs on each subproblem independently and logical subfile boundaries may not coincide with track boundaries (fragmentation phenomenon), \( m + 1 \) buffers suffice. Additionally, the buffer area can be used to process the sample, thus saving memory space. Except for buffer area, all others are of bounded size; hence for analysis purposes, \( (m + 1)t \), the buffer area size, is referred to as the memory size.

Since the fan-out parameter \( m \) must be specified in advance, \( k \) may range from 1 to the track size \( t \). When a random sample is loaded into memory, \( \lceil (km - 1)/t \rceil \) equidistant tracks located within the boundaries of the subfile to be sorted are read. This approach avoids the degenerative behaviour showed by Quicksort-like algorithms with almost sorted files.

To make Extquick \textit{in situ}, the size of each subfile has to be known before partitioning the file. This requires reading the entire file at each step and counting the number of records belonging to every subfile. Consequently, the boundaries of all subfiles are defined and the reading pass can be
used to test whether or not the file is already sorted. Clearly when \( m = 2 \) (Quicksort case) the extra reading pass can be avoided. However, the overhead due to this extra reading pass dissolves within the better refined divide and conquer obtained by a larger fan-out \( (m > 2) \).

Partitioning a file into \( m \) subfiles starts by reading the first track of each subfile into its corresponding buffer. Afterwards, the \( m \) buffers are scanned. During this scanning, each element is searched in the pivot array. An element belongs to the \( i \)th subfile if the search ends in the \( i \)th gap of the pivot array. Elements have to be exchanged whenever they belong to another subfile. These exchanges among subfiles are controlled by a linear list of subfiles. A subfile links to another subfile if there is a record in the buffer of the first subfile that goes to the buffer of the second subfile. Thus, record exchanges are performed any time a loop arises in the list. This procedure generalises the exchange scheme used by Quicksort where only two subproblems are present. When all elements in a buffer have been scanned and at least one of them exchanged, the buffer is written back to disk and the following track of the subfile is automatically read into the same buffer. Note that, if no elements in the scanned buffer has been exchanged, there is no need to write the buffer back to disk and only the following track has to be read in. After a subfile has been completely read, its corresponding end-of-file indicator is set.

*In situ* Extquick must also handle the fragmented tracks: tracks shared by consecutive logical subfiles. Extreme fragmented tracks have to be read an extra time when they are updated. Because the algorithm is sequential, no inconsistency arises when such tracks, are updated.

3. **Choice of \( m \) and \( k \)**

In practice the track size \( t \) may range from 4 to 1024. Also tracks may range from 1 Kbyte in personal workstation disk devices to 64 Kbytes in main-frame disk devices. Usually practical applications define record sizes ranging from 64 bytes to 2 Kbytes.

Overhead due to fragmented tracks is a function of the fan-out and the depth of the computational tree. A simple calculation shows that the number of fragmented tracks handled by the algorithm could be \( O(N) \) at worst. Therefore, assigning the maximum memory available is not necessarily the best decision to take and, even though larger values of \( k \) induce better partitions, the latter are obtained at the expense of reading larger samples.

Table 1 illustrates how the total number of track I/Os (read + written tracks) performed by Extquick oscillates. Only local minima and maxima with their corresponding \( m \) are displayed. Results were obtained from sorting random files of size \( N = 32,768 \) with \( k = t \) and different values of \( t \) and \( m \).
Table 2 presents the average and the variance of the total number of track I/Os and fragmented tracks handled. In this case, simulations from sorting random files of size $N = 4098$ with a track size $t = 8$ and different values of $m$ and $k$ provide the results.

Note that in Table 1 the total number of track I/Os is a damped oscillating function of the fan-out $m$. The global minimum is the second local minimum. The last row corresponds to the case when memory is large enough to fit the whole file in.

The first and third columns in Table 2 display large numbers of fragmented tracks that are responsible for the large number of track I/Os. In the first column, the high depth of the computational tree causes the large track fragmentation while a large fan-out causes such a behaviour in the third column. The second column corresponds to the best simulated results obtained for $N = 4096$ and $t = 8$.

Study of similar tables for a wide range of file sizes, track sizes, and sample sizes lets us conclude the following:

(i) Instead of $N$, $N/t$ seems to be a more suitable measure of the problem size.

(ii) By $N/t$ being fixed, no great differences in sorting time are observed for different values of $N$. 

### Table 1

<table>
<thead>
<tr>
<th>$t$</th>
<th>1024</th>
<th>512</th>
<th>256</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>486</td>
<td>1001</td>
<td>2724</td>
<td>5212</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>410</td>
<td>734</td>
<td>1369</td>
<td>2526</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>498</td>
<td>980</td>
<td>1967</td>
<td>3867</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>10</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>393</td>
<td>747</td>
<td>1435</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>13</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>422</td>
<td>868</td>
<td>1764</td>
<td>3543</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>62</td>
<td>26</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>63</td>
<td>127</td>
<td>255</td>
<td></td>
</tr>
</tbody>
</table>
TABLE 2
Simulated Average Number of Track I/Os (First Row)
and Fragmented Tracks (Second Row) for N = 4096 and t = 8

<table>
<thead>
<tr>
<th>k</th>
<th>m</th>
<th>20</th>
<th>41</th>
<th>400</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5094 ± 218</td>
<td>3398 ± 369</td>
<td>5101 ± 51</td>
<td></td>
</tr>
<tr>
<td></td>
<td>949 ± 120</td>
<td>477 ± 179</td>
<td>1989 ± 32</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5362 ± 303</td>
<td>3009 ± 239</td>
<td>5329 ± 46</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1182 ± 172</td>
<td>289 ± 119</td>
<td>2069 ± 26</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5469 ± 254</td>
<td>2894 ± 133</td>
<td>5444 ± 48</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1240 ± 141</td>
<td>229 ± 66</td>
<td>2091 ± 27</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5655 ± 291</td>
<td>2868 ± 58</td>
<td>5518 ± 54</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1345 ± 154</td>
<td>213 ± 28</td>
<td>2092 ± 28</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5741 ± 287</td>
<td>2868 ± 39</td>
<td>5592 ± 58</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1378 ± 148</td>
<td>211 ± 19</td>
<td>2097 ± 30</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5917 ± 270</td>
<td>2873 ± 29</td>
<td>5647 ± 57</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1453 ± 139</td>
<td>211 ± 15</td>
<td>2095 ± 30</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6006 ± 287</td>
<td>2878 ± 30</td>
<td>5707 ± 54</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1490 ± 143</td>
<td>211 ± 15</td>
<td>2098 ± 27</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>6132 ± 302</td>
<td>2881 ± 17</td>
<td>5757 ± 48</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1535 ± 150</td>
<td>210 ± 10</td>
<td>2098 ± 24</td>
<td></td>
</tr>
</tbody>
</table>

(iii) For a given N, if t is doubled, the sorting time decreases by more than a factor of 2.

(iv) If the file does not fit in memory and for a given t, the total number of track I/Os is a damped oscillating function of m with two local minima.

(v) The second local minimum is the global one.

(vi) For given N and t, the local minima are of the form $c(N)(N/t)^{x(N)}$.

(vii) The coefficients $c(N)$ and $x(N)$ are concave and convex functions of $\ln(N)$, respectively.

(viii) At m optimal ($m^*$), $k = t$ is the best choice since it provides the smallest variance of the sorting time.

(xi) For $m \neq m^*$, $k = t/2$ is a reasonable compromise.
Table 3 presents the optimal fan-outs for different file and track sizes. A simple regression indicates that

\[ m^* = (0.36900 + 0.02999 \log_2(N))(N/t)^{0.90780 - 0.02208 \log_2(N)} \]

closely approximates the optimal simulated \( m \). Similarly,

\[ m_1 = (1.41402 - 0.03656 \log_2(N))(N/t)^{0.35705 + 0.00736 \log_2(N)} \]

**TABLE 3**

Simulated Optimal Fan-out \( m \) (First Row) with Corresponding Track I/Os (Second Row) and Fragmented Tracks Handled (Third Row)

<table>
<thead>
<tr>
<th>( \log_2(N/t) )</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16384</th>
<th>32768</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>213 ± 3</td>
<td>214 ± 3</td>
<td>215 ± 2</td>
<td>215 ± 1</td>
<td>215 ± 1</td>
<td>206 ± 1</td>
</tr>
<tr>
<td></td>
<td>35 ± 2</td>
<td>36 ± 1</td>
<td>36 ± 1</td>
<td>36 ± 1</td>
<td>36 ± 1</td>
<td>30 ± 1</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>406 ± 6</td>
<td>408 ± 4</td>
<td>410 ± 3</td>
<td>410 ± 2</td>
<td>402 ± 1</td>
<td>398 ± 1</td>
</tr>
<tr>
<td></td>
<td>56 ± 3</td>
<td>59 ± 2</td>
<td>60 ± 1</td>
<td>60 ± 1</td>
<td>54 ± 1</td>
<td>48 ± 1</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>16</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>773 ± 10</td>
<td>777 ± 8</td>
<td>772 ± 5</td>
<td>774 ± 4</td>
<td>757 ± 3</td>
<td>749 ± 2</td>
</tr>
<tr>
<td></td>
<td>89 ± 5</td>
<td>90 ± 4</td>
<td>87 ± 3</td>
<td>89 ± 2</td>
<td>78 ± 1</td>
<td>78 ± 1</td>
</tr>
<tr>
<td>8</td>
<td>30</td>
<td>27</td>
<td>25</td>
<td>22</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>1485 ± 18</td>
<td>1448 ± 13</td>
<td>1485 ± 9</td>
<td>1465 ± 6</td>
<td>1450 ± 4</td>
<td>1433 ± 2</td>
</tr>
<tr>
<td></td>
<td>131 ± 10</td>
<td>136 ± 1</td>
<td>135 ± 5</td>
<td>126 ± 4</td>
<td>112 ± 2</td>
<td>101 ± 2</td>
</tr>
<tr>
<td>9</td>
<td>45</td>
<td>41</td>
<td>34</td>
<td>30</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2869 ± 22</td>
<td>2881 ± 17</td>
<td>2842 ± 10</td>
<td>2822 ± 7</td>
<td>2808 ± 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>197 ± 13</td>
<td>210 ± 9</td>
<td>186 ± 6</td>
<td>173 ± 9</td>
<td>165 ± 3</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>64</td>
<td>55</td>
<td>49</td>
<td>43</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5561 ± 26</td>
<td>5553 ± 18</td>
<td>5529 ± 13</td>
<td>5487 ± 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>284 ± 15</td>
<td>283 ± 10</td>
<td>270 ± 7</td>
<td>244 ± 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>88</td>
<td>76</td>
<td>67</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10812 ± 56</td>
<td>10844 ± 56</td>
<td>10802 ± 13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>392 ± 27</td>
<td>395 ± 28</td>
<td>371 ± 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>123</td>
<td>105</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>21343 ± 98</td>
<td>21262 ± 25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>553 ± 45</td>
<td>550 ± 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>173</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>42235 ± 337</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>808 ± 120</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
if $\text{size}(X) \leq \text{memory\_size}$
then sort in one pass
else
    compute $m_1$ and $m^*$;
    if $(m^* + 1)t \leq \text{memory\_size}$
    then
        $m := m^*$;
        $k := t$
    else
        $m := \min \left( m_1, \left[ \frac{\text{memory\_size}}{t} - 1 \right] \right)$;
        $k := \left\lceil \frac{t}{2} \right\rceil$
    endif
end_if

FIG. 2. Procedure for selecting $m$ and $k.$

estimates the first local minimum. Figure 2 describes an $m$ and $k$ selection procedure for practical purposes.

4. EXTQUICK vs PREVIOUS QUICKSORT-LIKE ALGORITHMS

Basically Extquick ought to be compared with the variants appearing in (Cunto, 1984; Gonnet, 1984). Although these variants have the same asymptotic analytical performance, in practice the second seems to be faster than the first. Therefore the comparison will be exclusively restricted to Extquick and External Quicksort.

A performance comparison between both algorithms is not easy because the simulation data of External Quicksort performance available in the literature (Gonnet, 1984) corresponds to a simplified model which does not take track size into consideration and consequently the effect of fragmented tracks in the total performance. This means that the performance reported in (Gonnet, 1984) is a lower bound if a more realistic computational model is taken into consideration.

The reported average number of passes over a file performed by External Quicksort is $\log_2(N/M) - 0.924$, where $M$ is the size of the sample kept in memory to drive the partition process. A pass includes reading-from- and writing-to-disk the $N$ elements of a file. Since at least two buffers are needed by External Quicksort to handle the partition process and the whole record has to be kept in the sample, $M = (m + 1)t - 2$ seems to relate to the memory needed by both sorting algorithms. Finally, the number of passes obtained in the case of External Quicksort has to be multiplied by $2N/t$ to get a reasonable correspondance with the homologous Extquick results presented in Table 4.
### Table 4
Performance Comparison for Extquick (First Row) vs External Quicksort (Second Row)

<table>
<thead>
<tr>
<th>log$_2$(N/t)</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16384</th>
<th>32768</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>213</td>
<td>214</td>
<td>215</td>
<td>215</td>
<td>215</td>
<td>206</td>
</tr>
<tr>
<td></td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>133</td>
</tr>
<tr>
<td>6</td>
<td>406</td>
<td>408</td>
<td>410</td>
<td>410</td>
<td>402</td>
<td>393</td>
</tr>
<tr>
<td></td>
<td>225</td>
<td>225</td>
<td>225</td>
<td>225</td>
<td>243</td>
<td>265</td>
</tr>
<tr>
<td>7</td>
<td>773</td>
<td>777</td>
<td>772</td>
<td>774</td>
<td>757</td>
<td>749</td>
</tr>
<tr>
<td></td>
<td>509</td>
<td>531</td>
<td>555</td>
<td>555</td>
<td>608</td>
<td>874</td>
</tr>
<tr>
<td>8</td>
<td>1485</td>
<td>1448</td>
<td>1485</td>
<td>1465</td>
<td>1450</td>
<td>1433</td>
</tr>
<tr>
<td></td>
<td>1137</td>
<td>1216</td>
<td>1275</td>
<td>1374</td>
<td>1448</td>
<td>2003</td>
</tr>
<tr>
<td>9</td>
<td>2869</td>
<td>2881</td>
<td>2842</td>
<td>2822</td>
<td>2808</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2679</td>
<td>2820</td>
<td>3104</td>
<td>3295</td>
<td>3347</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>5561</td>
<td>5553</td>
<td>5529</td>
<td>5487</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6346</td>
<td>6802</td>
<td>7150</td>
<td>7544</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10812</td>
<td>10844</td>
<td>10802</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14881</td>
<td>15750</td>
<td>16513</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>21343</td>
<td>21262</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>33958</td>
<td>35845</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>42235</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>76181</td>
</tr>
</tbody>
</table>

Note that in the previous table Extquick is superior to External Quicksort for cases when N/t is not small and for files of medium size or bigger. Finally, if Extquick and External Quicksort asymptotic performances are compared in terms of passes, we found that 1.5 log$_m$(N) ≤ log$_2$(N) for m > 2 (see Section 5.2.2 and (Gonnet, 1984)), thus providing further support in favor of our algorithm.

### 5. The Analysis Technique

Usually the main concern in external sorting is the performance of I/O operations, not only because they are the most time consuming of all the operations performed, but also because they generally determine the order of complexity of these algorithms. Since algorithms read files into memory
and write them back into subfiles, the analysis technique considers the logical reading and writing of a record as the elementary operation. Physical I/O complexity, which is usually determined by the size of the physical blocks that must be handled, is not an exact proportion of the logical I/O complexity. The increment, proportional to the size of the file, is due to the handling of the physical track fragmentation shared by consecutive logical subfiles.

The method assumes that all permutations in the input data are equally likely. Moreover, the partition scheme must ensure that this assumption holds for all the subfiles generated during the computation: that is the case of Extquick. To prove this we have to observe that the number of permutations of $N - m + 1$ integers that contain a fixed subpermutation of size $t$ is equal to the number of ordered configurations which results from distributing $N - m + 1 - t$ records into $t + 1$ subfiles. Consequently any permutation in a subfile containing $t$ records may occur with probability $1/t!$.

Intuitively the time complexity of the algorithm as a function of the logical I/O operations is influenced by the way a file is distributed into subfiles. It is obvious that the algorithm may perform poorly especially when the partition process continuously generates empty subfiles. Certainly the probability of this occurring is very small. As mentioned in the introduction, if the processing order of the subfiles is not controlled, the unnecessary space requirement for handling the recursion may turn out to be linear with respect to the size of the initial file.

The present section is divided into two subsections. The first shows that the algorithm space requirement is only a logarithmic function of the size of the file to sort. The second subsection is devoted to the average run-time analysis of the algorithm which essentially computes the coefficient of the highest term in the complexity expression of logical I/O operations performed.

5.1. Extquick Space Requirement

The space in main memory is directly related to the maximum size of the stack which handles the recursion at any moment during the ordering of a file. Thus, let $S(N)$ denote the maximum size of the stack; if the strategy of processing smaller subfiles first is taking into consideration, $S(N)$ can be recursively defined as

$$S(N) = \max_{1 \leq i \leq m} \left \{ S(N_i) + m - i \left| \sum_{1 \leq i \leq m} N_i = N - m + 1, N_i \leq N_{i+1} \right. \right \}$$

if $N \geq m$

$$S(N) = 1 \quad \text{otherwise.}$$

(1)
Lemma 1. The maximum size of the stack to handle the recursion in Extquick is a logarithmic function of the size of the file to be sorted; that is,

\[ S(N) \leq (m - 1) \log_m(N) \quad \text{if} \quad N \geq m. \]

Proof. The proof of the lemma is by induction on the size of the file. Clearly, when \( N = m \) the lemma is true. After the partition, all the subfiles are strictly less in size than the original file size. Substituting the inductive hypothesis into (1), we get

\[ S(N) = \max_{1 < i < m} \left\{ (m - 1) \log_m(N_i) + m - i \left| \sum_{1 \leq i \leq m} N_i = N - m + 1, N_i \leq N_{i+1} \right. \right\} \]

if \( N \geq m. \)

Because smaller subfiles are processed first, the size of the \( i \)th subfile to be processed is upper-bounded by the case when all the smaller files are empty and the non-empty subfiles are of equal size; that is,

\[ N_i \leq (N - m + 1)/(m - i + 1). \]

The formula can then be expressed as

\[ S(N) \leq \max_{1 \leq i \leq m} \left\{ (m - 1) \log_m((N - m + 1)/(m - i + 1)) + m - i \right\} \]

\[ \times \sum_{1 \leq i \leq m} N_i = N - m + 1, N_i \leq N_{i+1} \right\} \quad \text{if} \quad N \geq m. \]

Because the function to maximise is convex with only one internal minimum, its maximum value is reached at the boundary values of \( i \). Thus, for \( i = 1 \) or \( i = m \) the lemma follows. \( \square \)

Observe that, when the files are evenly split and \( N \) is one less than a power of \( m \),

\[ S(N) \geq (m - 1) \log_m \left( \frac{N + 1}{2} \right). \]

5.2. Extquick Time Complexity

This subsection will be also divided into two parts. The first relates the time complexity of Extquick to the associated computational trees while
the second formally adapts the fringe technique for the analysis of our algorithm.

5.2.1. Sorting Effort in Terms of the Fringe of the Computational Tree

The main purpose of this section is to relate the sorting effort (measured in terms of the average number of logical I/O operations performed by the algorithm when sorting a random file $X$ of $N$ records) as a function of the distribution of subtrees in the fringe of the computational tree associated to Extquick.

Two types of blocks can be distinguished in a computational tree: internal blocks, which have $m-1$ elements and $m$ descendants, and fringe blocks, which hold a variable number of elements and no descendant. The fringe of a tree with respect to a collection of configurations $C = \{T'; q < r < s\}$ is defined as the set of subtrees in the tree that are isomorphic to configurations in $C$; the set of subtrees contains all the fringe blocks and no adjacent subtrees can be merged in order to produce a bigger configuration in $C$. The fringe collection associated with Extquick includes all the configurations that, because of their size, are sorted in one pass. If memory size $= km - 1$, $C$ includes $k(m-1)+1$ types of fringe configurations denoted by $T'$, $k \leq r \leq km$, where $r$ is the number of inter-element-gaps in the configuration $T'$ with $r - 1$ elements. This collection $C$ is depicted in Fig. 3.

Let $A_{N,k}'$ denote the average number of fringe subtrees of type $T'$ at level $k \geq 1$, $k \leq r \leq km$, in a tree with $N$ elements. $A_{N,k}'(z)$ is the generating function associated to $A_{N,k}'$ and $A_N(z)$ the corresponding vector. The vector $A_N(z)$ is a good description of the tree fringe since $A_N'(1)$ provides the number of fringe subtrees isomorphic to $T'$, while $A_N'(1)$ provides the contribution to the element path by the same type of fringe subtree. The number of internal blocks is denoted by $n_i$, $N_f = \sum_{r=k}^{km} (r-1) A_N'(1)$ and $n_f = \sum_{r=k}^{km} A_N'(1)$ are the number of elements and the number of fringe blocks in the tree fringe. It is important to note that if $N < km$ the computational tree consists of only one fringe block; therefore $n_f = 1$, $N_f = N$, and $n_i = 0$. Conversely if $N \geq km$, the computational tree has a root (internal block) with $m$ descendants which are denoted by $S_j$, $1 \leq j \leq m$. This observation is helpful since the following lemmata are proved by induction on $N$ distinguishing between the two cases mentioned above.

\[
\begin{array}{cccccc}
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
T^2 & T^3 & T^4 & T^5 & T^6 \\
\end{array}
\]

Fig. 3. Fringe configurations for Extquick $k = 2$, $m = 3$. 

When $N \geq km$, any descendant $S_j$, $1 \leq j \leq m$, has $N^j$ elements, $n^j_i$ internal blocks, $n^j_f$ fringe blocks, and $N^j_f$ fringe elements. Thus,

$$N = \sum_{j=1}^{m} N^j + m - 1,$$

$$n_i = \sum_{j=1}^{m} n^j_i + 1,$$

$$n_f = \sum_{j=1}^{m} n^j_f,$$

and

$$N_f = \sum_{j=1}^{m} N^j_f.$$

**Lemma 2.** The number of fringe blocks and the number of internal blocks in a computational tree associated to Extquick's execution are related as follows:

$$n_f = (m - 1) n_i + 1.$$

**Lemma 3.** The number of fringe blocks plus the number of elements in such blocks present in the computational tree derived from an Extquick's execution when sorting $N$ elements is one more than the number of elements sorted. That is,

$$n_f + N_f = N + 1.$$

**Corollary 4.** The sum of inter-element gaps in the fringe blocks is one more than the number of elements to be sorted. Thus,

$$\sum_{r=k}^{km} r A^r_N(1) = N + 1.$$

As we will prove later on in this section, the sorting effort is proportional to the element path $I$ of the computational tree. More precisely, the level $l(x)$ of an element $x \in X$ is the number of blocks in the path from the root block of the block holding $x$. A one block length path is defined by the root. Thus, $I = \sum_{x \in X} l(x)$.

Let us also define $F = \sum_{r=k}^{km} A^r_N(1)$ and $E = \sum_{r=k}^{km} (r-1) A^r_N(1)$ to be the fringe block path and the fringe element path, respectively. If $N < km$ then $F = 1$ and $I = E = N$. When $N \geq km$, and the corresponding element path,
fringe block path and fringe element path of each descendant $S_j$, $1 \leq j \leq m$, are denoted by $I^j$, $F^j$, and $E^j$ respectively, the following identities hold:

\[
I = \sum_{j=1}^{m} I^j + N,
\]

\[
F = \sum_{j=1}^{m} F^j + n_f,
\]

and

\[
E = \sum_{j=1}^{m} E^j + N_f.
\]

**Lemma 5.** The element path, the fringe block path, and the fringe element path of the computational tree satisfy the following identity:

\[
I = F + E - mn_f - 1.
\]

**Corollary 6.** The element path relates to the fringe subtree distribution as follows:

\[
I = \sum_{r=k}^{km} rA_N^{r1}(1) - \frac{m}{m-1} \sum_{r=k}^{km} A_N^r(1) + \frac{1}{m}.
\]

Without consideration of the fragmentation effect, elements in internal blocks are read twice and written once, while elements in the fringe blocks are only read and written once. The extra reading pass happens when the number of elements to sort during the pass is greater than $km - 1$ elements and thus the algorithm has to know in advance the size of the subfiles to be generated by the partitioning step. Since the sorting effort is measured in terms of the read/write operations performed by the algorithm, elements in internal blocks have to be charged with one and one half units, elements in fringe blocks with only one unit.

**Lemma 7.** The number of logical I/O performed by Extquick is proportional to the element path and the number of fringe blocks. That is,

\[
S_{m,N} = \frac{3}{2} I - \frac{1}{2} N_f - A_N^2(1).
\]

We have reached the main point of the section which is to relate Extquick's sorting effort to the fringe subtree distribution of its associated computational tree.
THEOREM 8. The number of logical I/O performed by Extquick can be computed in terms of the fringe subtree distribution as follows:  

\[ S_{m,N} = \frac{3}{2} \sum_{r=k}^{km} r A_{N}^{r}(1) - \frac{1}{2} \frac{4m-1}{m-1} \sum_{r=k}^{km} A_{N}^{r}(1) - \frac{1}{2} (N+1) - A_{N}^{2}(1) + \frac{1}{m} \]

\[ = \frac{3}{2} \sum_{r=k}^{km} r A_{N}^{r}(1) - O(N). \]

5.2.2. Fringe Technique Applied to Extquick

Since the input file is assumed to be random and the recursive process builds random subfiles, the distribution of fringe subtrees in the computational trees derived from the sorting of \( N \) elements can be viewed as the distribution resulting from sorting \( N-1 \) elements and inserting, with equal probability, the \( N \)th element into any of the \( N \) inter-element gaps.

Figure 4 shows the Markovian chain of configuration changes when \( N \) elements are sorted, in terms of sorting \( N-1 \) elements. The arrows also show the probability of changing from a given configuration to another.

LEMMA 9. The dynamic changes of the fringe of computational trees when sorting \( N \) elements is described in terms of the recurrence

\[ A_{N}(z) = \left( 1 + \frac{H(z)}{N} \right) A_{N-1}(z), \quad N < km, \]

\[ A_{km-1}^{T}(1) = (0, 0, ..., 1), \]

where

\[ H(z) = \begin{pmatrix}
-k & 0 & \ldots & 0 & (m-1)kmz \\
0 & (k+1) & \ldots & 0 & kmz \\
k & 0 & \ldots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \ldots & -(km-1) & 0 \\
0 & 0 & \ldots & km-1 & -km
\end{pmatrix}. \]

Proof. With probability \( r/N \) a given configuration \( T' \) changes into \( T'^{r+1} \) except for \( T'^{km} \) which changes, with probability \( km/N \), into \( m-1 \) configurations of type \( T^k \) and one of type \( T^{k+1} \) at one deeper level. \[ \]

LEMMA 10. The matrix \( H(1) \) has all its eigenvalues pairwise distinct, its biggest eigenvalues is 1, and the real part of all the remaining eigenvalues is strictly less than 1.
Proof. This proof is a slight variation of similar lemmata in (Knuth, 1973, 6.2.4-10; Poblete, 1985).

Let
\[ p(l, z) = \det(H(z) - lI) \]
\[ = (-1)^{km} (km + l)((k + 1)\frac{km}{m-1})z - (k + 1)^{\frac{km}{m-1}} \]
and
\[ p(\lambda) = p(\lambda, 1). \]

We can easily check that $-km$ and $1$ are unique roots of $p(\lambda)$.

Since $|p(\lambda)| > 0$ for $\lambda > 1$, none of the roots are greater than 1. Also, the root cannot be of multiplicity bigger than 1, otherwise $p(\lambda) = 0$ and $p'(\lambda) = 0$ would disagree.

The matrix $H(z)$ can be decomposed into
\[ H(z) = E(z) D(\lambda(z)) E(z)^{-1}, \]
where $D(\lambda(z))$ is the diagonal matrix of eigenvalues $\lambda(z)^T = (\lambda_1(z), ..., \lambda_{km-1} + 1(z))$ of $H(z)$ and $E(z)$ the non-singular matrix of eigenvectors related to the eigenvalues. Therefore, the result of iteration on the recurrence given above is
\[ A_N(z) = \prod_{j = km}^N (I + D(\lambda(z))/N) A_{km-1}(z). \]

The previous recurrence allows us to express the sorting effort of Theorem 8 in terms of a linear combination of the eigenvalues of $H(z)$. To follow such a direction let us define
\[ P_{m,N}(z) = \sum_{r = k}^{km} r A_N^r(z). \]
Thus, from Corollary 4, \( P_{m,N}(1) = N + 1 \), and Theorem 8 can be rewritten as \( S_{m,n} = P'_{m,N}(1) + O(N) \).

Moreover,

\[
P_{m,N}(z) = \sum_{r=1}^{k(m-1)+1} \alpha_r(z) \prod_{j=km}^{N} (1 + \lambda_r(z)/j),
\]

since the eigenvector matrix \( E(z) \) and the number of inter-element gaps in the fringe configurations \( T' \) do not depend on \( \lambda \).

**Lemma 11.** The linear coefficients \( \alpha_r(z) \), \( 1 \leq r \leq k(m-1) + 1 \), as defined before, satisfy \( \alpha_1(1) = km(N+1) \) and \( \alpha_r(1) = 0, 2 \leq r \leq k(m-1) + 1 \).

**Proof.** We refer our proof to the Lemma 3.4 in (Poblete, 1985). The only difference resides in the fact that in our case \( P_{m,N}(1) = N + 1 \) is not a probability function, which will not affect the sequence of the proof. 

Using Lemma 11 in the computation of \( P'_{m,N}(1) \) we then conclude that the sorting effort performed by Extquick when sorting \( N \) elements is directly related to the largest eigenvalue of the transition matrix \( H(z) \). That is,

\[
S_{m,N} = \lambda'_1(1)(N+1)H_N + O(N)
\]

\( (H_N = \sum_{i=1}^{N} 1/i) \).

The eigenvalues \( \lambda_r(z) \) are the roots of the polynomial

\[
p(\lambda(z), z) = \det(H(z) - \lambda(z)I);
\]

therefore

\[
\lambda'(z) = -\frac{\partial p}{\partial z}\bigg|_{\lambda(z)} \frac{\partial p}{\partial \lambda(z)}
\]

evaluated at \( z = 1 = \lambda_1(1) \) (Lemma 10).

**Theorem 12.** The effort performed by Extquick when sorting \( N \) elements, measured in terms of logical I/O operations, is

\[
S_{m,N} = \frac{3}{2} \frac{1}{H_{km} - H_k} (N + 1) H_N + O(N).
\]

**Proof.** From Lemma 10,

\[
p(\lambda, z) = (-1)^{k(m-1)}(km + \lambda)((k + 1)^{k(m-1)}z - (k + \lambda)^{k(m-1)}).
\]
The theorem follows by differentiating and evaluating $p(\lambda, z)$ at $\lambda = z = 1$.

It is easy to see that the time complexity can be bound in practice by $S_{m,N} = (3/2) N \log_m N + O(N)$, since harmonic numbers are basically logarithmic functions. Table 5 compares performance results presented in Tables 3 and 4 with Extquick asymptotic performance expressed as $1.5 \log(N/N) N/\lambda$. Looking at this table, we may conclude that Extquick performance should be of the form $S_{m,N} = (3/2) N \log_m N - O(N)$ for large values of $N$.

### 6. Further Research

Extquick and External Quicksort performances were compared in Section 4. The main conclusion from that analysis favored Extquick in the case of non-small file sizes and file size to track size ratios. In this context a natural approach arises: to combine Extquick and External Quicksort and therefore to produce a hybrid algorithm that would improve the performance of both basic algorithms.

Skipping details, the External Quicksort sorted sample of size $M$ will be divided into $m - 1$ parts. The partition process will produce $m$ subfiles alternating with $m - 1$ sorted segments. Similarly to Extquick, if $m > 2$ is assumed, a preliminary reading pass over the file must be performed. This reading pass will determine the number of elements in every subfile. Further balancing in the subfile size distribution can be done by
reassigning the elements falling between the extremes of each of the $m - 1$ sorted segments.

Once again, a more detailed implementation with emphasis on the \textit{in situ} aspect of the hybrid algorithm as the method for selecting the parameters has to be proposed. Furthermore, simulation results and a formal asymptotic analysis of the algorithm would judge the practicality of such an approach.

\section*{Acknowledgments}

The authors thank Dr. C. Mendoza, Dr. J. Rivero, and an anonymous referee for pointing out improvements in the presentation of this paper.

\section*{References}


