Lutz M. Wegner

Sorting

The Turku Lectures

Turku Centre *for* Computer Science

TUCS Lecture Notes
No 22, February 2014

Lutz M. Wegner

Sorting – The Turku Lectures

Revised Edition of the Sorting Algorithms Presented in Turku, Finland

May 11 – 15, 1987

TUCS Lecture Notes 22

ISBN 978-952-12-3020-2

February 2014

For Jukka

Preface

In April 1987, Professor Timo Järvi from the University of Turku wrote me a letter, informing me that the Finnish Academy of Sciences had provided funds to invite me for a series of lectures on "Sorting". The idea for these lectures came from my good friend and co-author of several papers Jukka Teuhola, lecturer at the Department of Mathematical Sciences and Computer Science. The talks then took place at the University of Turku from May 11th till May 15th 1987.

Since about 1980 I had worked on various algorithms for internal, external, and network sorting and wrote my "Habilitationsschrift" on variants of *Quicksort* in 1982. My intention in 1987 was to present to the audience in Finland my personal classification on sorting which would hopefully reflect – at least in parts – the state of the art of this classical topic in algorithms and data structures.

However, that claim was hopelessly over-ambitious as I discovered in writing down the foils. Too many publications had appeared in the Mid-Eighties for my overview to be complete and up-to-date. Thus I had to concentrate on those variants where I and my co-authors had contributed. Indeed, no single book could in those days – or by now – present all the bright ideas, clever implementations and deep theoretical results in "Sorting" in a uniform way. In fact, even Donald Knuth's Second Edition of his Volume 3 "Sorting and Searching", which appeared in 1997 (First Edition 1973), takes a biased view, although unmatched in terms of scrutiny and scientific depth.

Why then publish a lecture held in 1987 some 25 years later? The answer is two-fold. The lecture was held in the Pre-Internet-Age with old-fashioned transparencies. Unless transcribed by hand now and made available in digital form, the material will be lost. Most likely parts would be reinvented later under different names and by new authors.

Secondly, even if the old material is not directly usable today, where all algorithmic knowledge must appear as linkable service or otherwise be labeled out-dated and unusable, the notes have an historical and aesthetic value. A multitude of small, hand-crafted, tricky solutions all address one problem

which is so simple that it can be explained to a layman: Rearrange a sequence of items as efficiently as possible into some desired order. This gives the program fragments and analytical insights a minimalistic and miniature-like appearance which reflects the golden age of algorithmics of the Seventies and Eighties or, in a way, a lost art.

Acknowledgment
Marion Moser managed to decipher my hand-written foils and entered the text and formulas for almost all chapters in LaTeX. Her help is gratefully acknowledged. Kai Schweinsberg and Stefan Achler helped with proofreading for which I am very thankful. However, all mistakes are mine and I appreciate comments for corrections and improvements.

Finally, I would like to thank the Turku Center *for* Computer Science for accepting my submission as part of their Lecture Notes Series and – in particular – Tomi Mäntylä for the expert handling of the material.

Kassel Lutz Wegner
February 2014

Contents

Chapter 1
What to Look for in Sorting

1.1 Introduction

A study from the Eighties claims that mainframes spend 25 % of their CPU-time on sorting.

Example: DB "join" and "projection"

```
select STUDENTS.NAME, COURSE, DEPT
from SIGN_UP, STUDENTS
where SIGN_UP.NAME = STUDENTS.NAME
```

Table 1.1 SIGN_UP

NAME	COURSE
Jones	ALG+DS
Smith	DB1
Miller	ALG+DS
Smith	PROG1
Clark	DB1
Jones	PROG1

Table 1.2 STUDENTS

NAME	SEM	DEPT
Smith	1	CSC
Miller	5	MATH
Jones	1	MATH
Clark	3	CSC

Table 1.3 SIGN_UP $\bowtie$ STUDENTS

NAME	COURSE	DEPT
Jones	ALG+DS	MATH
Smith	DB1	CSC
Miller	ALG+DS	MATH
Smith	PROG1	CSC
Clark	DB1	CSC
Jones	PROG1	MATH

This database example also poses the question of how to detect and eliminate duplicates, say after a projection on NAME in the result above, if requested by the SQL `DISTINCT`-clause.

> Indeed, I believe that virtually every important aspect of programming arises somewhere in the context of sorting and searching! [Knu98, p. v]

Sorting is considered an ideal subject for

- algorithm design and software engineering
- studies of efficiency and performance evaluation
- concrete complexity theory

and was at its height in the Fifties and Sixties, as can be seen from e. g.

- ACM Symp. on Sorting, Nov. 1962 and CACM May 1963
- Knuth's bibliography in Computing Reviews 13 (1972)

but was also reconsidered in later years, e.g. in

- IEEE TC Special issue on sorting, April 1985

1.2 A realistic model

Knuth [Knu98, p. 4] proposes the following model. We are given N items (*records*)

$$R_1, R_2, ..., R_N$$

which form together a *file*. Each record R_j has a key, K_j, which governs the sorting. Additionally the record may contain "satellite information".

An ordering relation "$<$" is specified on the keys so that, for any three key values a, b, c, the following conditions hold:

i) exactly one of the possibilities $a < b$, $a = b$, $b < a$ is true (law of trichotomy)
ii) if $a < b$ and $b < c$, then $a < c$ (law of transitivity)

Thus a linear (total) ordering is assumed.

The goal of sorting is to determine a permutation $p(1)p(2)\ldots p(N)$ of the records, which puts the keys in non-decreasing order:

$$K_{p(1)} \leq K_{p(2)} \leq \cdots \leq K_{p(N)}.$$

The following issues are left unspecified:

- How are the records given (tape, disc, on-line, ...)?
- How many records are given and is the number known in advance?
- Is the key range known?

- Which operations are permitted (comparisons only, arithmetic operations)?
- How is the reordering going to take place?
- Is there an initial ordering?
- How much extra space is there available?

1.3 Internal vs External Sorting

There are two useful sub-models

- sorting an array in memory
- sorting a disc file

1.3.1 Sorting an Array (Internal Sorting)

This model assumes a data structure similar to the one given below (in PASCAL-notation).

```
TYPE
  item = RECORD
    key : integer;
    info : infotype
  END;
  sequence = ARRAY[1..N] OF item;
VAR
  a : sequence;
```

We then want a procedure

```
XYZ-Sort(VAR s : sequence);
```

which, when called with `XYZ-Sort(a)` rearranges array `a` s.t. afterwards

$$\texttt{a[i].key} \leq \texttt{a[i+1].key}\ \forall\, i\ (1 \leq i < N).$$

The assumptions in this model are the following:

- records can be exchanged, which possibly requires fixed length.
- access to each record incurs the same cost.
- the order is determined by key comparisons only.
- all records fit into core (main memory).

1.3.2 Sorting a Disc File (External Sorting)

The term "disc file" applies to all files located on external media with random access to individual records, as opposed to files with sequential access like tapes or streams. The corresponding data structures are now as follows (again in PASCAL-notation).

```
TYPE
  page = ARRAY[1..s] OF item;
  file = ARRAY[1..M] OF page;
  buffer = ARRAY[1..k] OF page;
VAR
  f : file;
```

The assumptions for a useful discussion reflect the classical storage hierarchies:

- $k \ll M$ and k is fixed during execution.
- the page size is fixed, e. g. 4 KB.
- the file does not fit into core.
- the access is *not* equally expensive for all pages on the disc.
- input/output-times dominate the total execution time (with exceptions).
- records are sequentially ordered within the pages.
- pages occupied by the file are *not necessarily* consecutive and sequentially ordered within cylinders.
- cylinders occupied by the file are *not necessarily* consecutive and ordered.
- the page replacement can be controlled by the user.

Other models worth investigating include:

- linear lists (chained nodes, internal sorting)
- parallel machine sorting (VLSI, SIMD)
- network sorting (LAN, WAN)
- tape sorting (bubble memory)

1.4 Assessing the Quality of a Sort

1.4.1 Asymptotic Running Time

Apart from measuring the actual running times of a program for various inputs, where the results depend on the programming language and computer used, it is customary to measure (analyze) the performance of a sorting algorithm in a more abstract way. This can be the number of e. g.

- key comparisons

- disc accesses
- MIX-instructions
- elementary PASCAL-, C-, Java-, ... instructions
- loop iterations
- record exchanges or moves

needed in the best, average, worst case, for certain key ranges, different numbers of records, initial orderings, presorted or multiset inputs. The result is always given as a function of the size n of the input, mostly using the so-called "Big-Oh-Notation".

$$f(n) = O(g(n))$$

iff there exist constants k and n_0 s. t.

$$|f(n)| \leq k \cdot |g(n)| \ \forall n > n_0.$$

This gives an upper bound on the growth rate of the function ignoring multiplicative and additive constants and justifies the term asymptotic running time.

A lower bound is noted in the "Big-Omega-Notation"

$$f(n) = \Omega(g(n)) \text{ iff } g(n) = O(f(n))$$

or more precisely: there is a constant k s. t.

$$f(n) \geq k \cdot g(n) \text{ infinitely often.}$$

Less frequently the "Little-oh-Notation"

$$f(n) = o(g(n)) \text{iff } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

is used to indicate that $f(n)$ grows asymptotically slower than $g(n)$. In particular, we always assume that in specifying $f(n) = O(g(n))$, there is no known $h(n) = o(g(n))$ s. t. $f(n) = O(h(n))$, i. e. if we give an upper bound, then it should be sharp.

Upper and lower bound together form the "Big-Theta-Notation"

$$f(n) = \Theta(g(n)) \text{ iff } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

1.4.2 Actual Running Times

Even when the asymptotic running time has been analyzed (which can be difficult), it is often useful to determine and to compare constants. What is the "set-up time" of the program? Can inner loops be optimized? Are so-

called "improvements", e. g. removal of tail recursion, effective? Is there an efficient *and* short implementation of a particular algorithm?

The problem, as mentioned before, is that all measurements are relative to a machine, language, and implementation of a comparator. If running in a multi-tasking environment, there can be "cycle stealing" and execution times may depend on the work load of the system. Still, pairwise comparisons of sorting functions under equal testing conditions should be run and the out-performance of one implementation over the other (the ranking) is an important information which remains valid even as the absolute measuring values become obsolete.

1.4.3 Extra Space

By convention, adding links (pointers) to records given as arrays – and not explicitly as a linked list – counts as $O(n)$ extra space. The unit of measurement is *words* which is the same as *bytes* in the Big-Oh-Notation or in *bits*, if more precision is needed. In particular, *in situ sorting* is defined as having at most

$$O((\log n)^2) \text{ extra bits}$$

which is the same as allowing a stack of at most logarithmic (in the size of n) height holding binary representations of integers in the order of n. This contrasts to *minimal storage sorting* which allows only $O(1)$ extra space.

Unfortunately, it seems that the use of the terms *in-place* (or Latin *in situ*) is inconsistent in the literature with a tendency to imply constant extra space, i.e. only $O(1)$ additional words are allowed [Wik13b]. On the other hand, almost all textbooks call *Quicksort*, which – when sorting n records – needs a stack of $O(\log n)$ words for recursion, an in situ sort, which would be wrong by the $O(1)$ definition. Conversely, the term *minimum storage* stems from Knuth's Exercise 3 in Section 5.5 [Knu98, p. 390] and is defined there as $O((\log n)^2)$ extra bits. This is somewhat unfortunate, as $O(1)$ is asymptotically less than $O((\log n)^2)$ and therefore should be called minimum storage (or *minimal storage*, as we do). However, now that the confusion is in the world, no amount of words will be able to restore order, we are afraid to say.

In discussing external sorting, extra space means space on the disc and space in core. Today, internal space is not as important as it used to be. When contemplating an external in situ sort, the problem of a system crash should be kept in mind, as it can lead to permanent loss of data.

1.4.4 Stable Sorting

A sorting algorithm is termed *stable*[1] if equal keys retain their relative order [Knu98, p. 4]. As an example, consider a file with passenger names and flight numbers which has been sorted according to name. If that file is then resorted according to flight number, a stable sort will keep all passengers per flight in ascending order of their names. Another example is a file of blind bids in an auction where the bids are serially recorded. If that file is then sorted according to the offered price, bids with equal value retain the order in which they were given. This is handy if the earliest bid wins and the entry time had not recorded. The problem is also somewhat related to the question of a speed-up if the input keys form a multiset (bag).

1.4.5 Smoothness

The term *smoothness* in sorting was introduced by Edsger W. Dijkstra in 1981, when he invented a *Heapsort*-variant, which he published in 1982 as Smoothsort [Dij82]. A sort is called smooth if its performance improves from $O(N \log N)$ to $O(N)$ for pre-sorted or multiset inputs of size N. In particular, a sort for multisets with a running time

$$T(N) = O(N \log n)$$

where n is the number of *distinct* keys, fulfills this definition.

1.4.6 Practical Considerations

A sort should be as uncomplicated and "good natured" as possible. Preference should be given to algorithms which

- require no stoppers $+\infty$, $-\infty$
- handle numeric and text keys
- handle variable length records
- show little variation in running time
- are easy to program, good to maintain, have a short text and are verified
- do not depend on a particular architecture
- restart easily
- work well in a paging environment
- can be applied to networks (distributed files)

[1] We use the term *nonstable* rather than *unstable* for the opposite to avoid its connotation with *unreliable*.

- can be run on a SIMD, MIMD machine
- permit interleaved I/O-traffic
- allow ascending and descending ordering
- can detect and eliminate duplicates on-the-fly
- are suitable for split-key files

 A good example is the *Quicksort* version for the UNIX software library as discussed by J. L. Bentley and M. D. McIlroy [BM93]. We return to this article in Chapter 12.

1.4.7 Taxonomy

Knuth's Vol. 3 Book on Sorting and Searching contains a table [Knu98, p. 382] "A Comparison of Internal Sorting Methods Using the `MIX` Computer" which lists the asymptotic running times (average and maximum), number of executed `MIX` instructions for input length $N = 16$ and $N = 1000$, space requirements, length of code and whether the method is stable for 14 algorithms given in the book. The table is a marvel of accuracy and epitomizes the way Donald Knuth combines computer science and mathematics in his scientific work. It should be consulted before discussing the relative merits of one sort over the other.

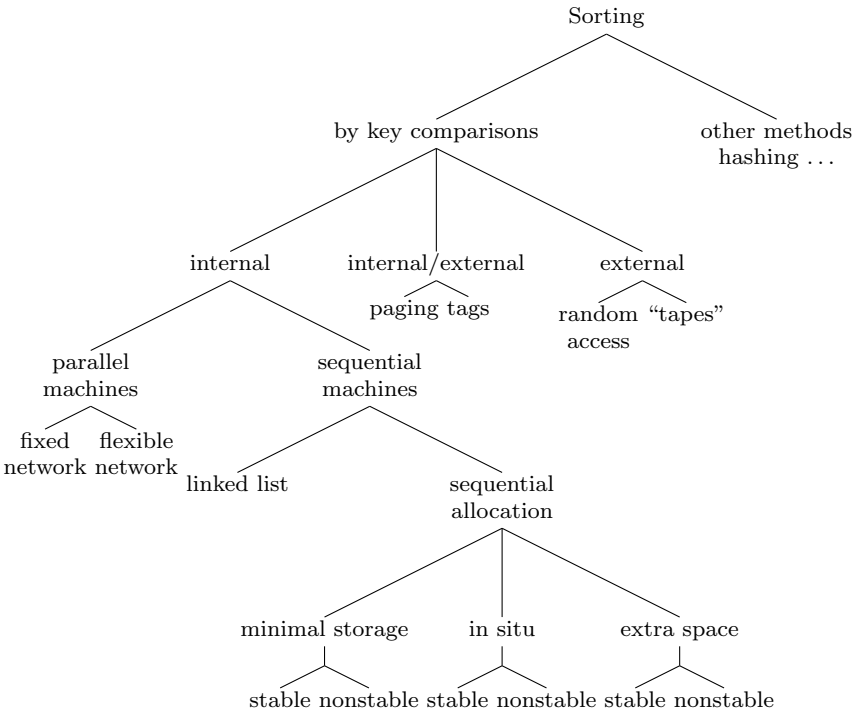


Fig. 1.1 A Taxonomy of Sorting

Chapter 2
Hoare's Invention and Sedgewick's Analysis

2.1 Classical Quicksort

2.1.1 Hoare's Original Publication

In 1962, Sir Charles Antony Richard Hoare, better known as Tony Hoare, invented a recursive, in situ, partition-exchange sorting algorithm which he, on the ground of its perceived efficiency, called *Quicksort* [Hoa62].

Listing 2.1 *Quicksort* general scheme.

```
function Quicksort (F: file): file;
var H: keytype; {pivot}
  F1, F2 : file;
begin
  if |F| ≤ 1
  then Quicksort := F {nothing to do}
  else
  begin {Phase 1 - partition}
    H := r1.key;
    F1 := set of all ri ∈ F with ri.key ≤ H (i > 1);
    F2 := set of all ri ∈ F with ri.key ≥ H (i > 1);
    {Phase 2 - sort recursively}
    Quicksort := Quicksort(F1), r1, Quicksort(F2)
  end {else}
end {Quicksort}
```

Susan M. Merrit, who tried to come up with a new taxonomy of sorting algorithms, calls *Quicksort* a *hardsplit/easyjoin* sort [Mer85]. This refers to the fact, that the partition phase is where the key comparisons and record moves happen, while the recursion phase simply keeps track of sorted and yet unsorted sub-sequences and joins them without comparisons or moves.

Actually, the split is not so hard and has an efficient implementation using pairwise exchanges of records whose keys are "out of order" when scanning the file from left and right.

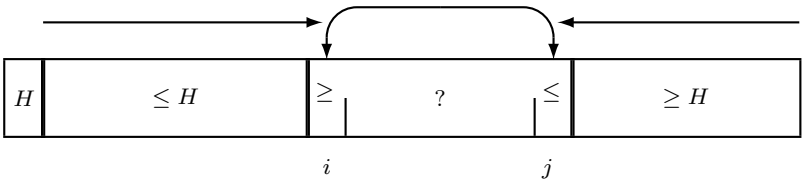


Fig. 2.1 Two-way scan with partition-exchange

Listing 2.2 The inner partition loop.

```
while i ≤ j do
begin
  repeat i := i+1 until a[i].key ≥ H;
  repeat j := j-1 until a[j].key ≤ H;
  switch(a[i], a[j])
end {while}
```

The value H, according to which the file is split in each recursion step, is called the *pivot*, the french word for *turning point*. The function `switch(x, y)` in the nested-loop implementation exchanges the two values of the two variables x and y; often the function is named `swap(x, y)` or is denoted by `x :=: y`.

Knuth [Knu98, p. 114] mentions that all comparisons are made against the same key H, which can be kept in a register. Similarly, variables i and j can be kept in registers with fast pre-increment and pre-decrement operations for the index fetch in most processor architectures. This speeds up key comparisons. We will show below that only about $N/6$ exchanges happen for sequences of N records in random order. In other words, 2/3rd of all records remain unmoved in each stage. We will also explain below why it is better to stop on keys equal to H.

2.1.2 A First Implementation

However, as simple as the algorithm might look, its implementation details are tricky. General wisdom has it that one out of three published *Quicksort* algorithms contains at least one subtle error, in many cases even several blunders. Even in our own suggestion for the partitioning loop above (cf. listing

2.2), one might expect an additional guard prior to the switch of records; should it not read "`if i < j then switch() else skip`" to prevent the erroneous switch at the end, when i and j have crossed? Yes and no. Yes, the test is a simple fix. No, not needed, if we undo this last switch afterwards. The logic behind this strategy is that one extra call of switch() is more efficient than permanently testing for crossed pointers prior to all exchanges. Details will be shown below.

In general, the program should handle arbitrary input, including sequences of length zero or one and pre-sorted or multiset data without crashing or a mistake in the output. A rather common error is an array access with an index out of bounds.

Even if the program does not crash, it may degenerate to a $O(n^2)$ performance by splitting off one key only in each round. This happens for input already sorted when the first or last key is chosen as pivot. In that case the recursion stack may grow to a size of $O(n)$ extra space which violates our in situ claim. Avoiding degeneration is not easy and depends on the pivot selection in combination with an adaptive partitioning strategy. Limiting the stack to logarithmic size, on the other hand, is easy and can be achieved by sorting the shorter of the two subfiles, which result from partitioning, first. Again, more on these points can be found in the analysis of *Quicksort* below.

We now give the original implementation, as presented in Turku, in *Pseudo-Pascal*. It assumes that we sort an array $a[0..N]$, where $a[0].key$ is an artificial key smaller than or at most equal to any key in the actual input sequence in $a[1..N]$, i.e.

$$a[0].key \leq a[i].key \qquad \forall i\ (1 \leq i \leq N).$$

Record $a[0]$ will act as a stopper (sentinel) to avoid special boundary checks. This is a common programming trick mentioned e.g. in [Knu98, p. 4]. The outermost call is then Quicksort$(s, 1, N)$ for a given sequence s with N records, i.e. parameters l and r index the first and last (true) record in the array representing s. Sequence s must not be empty, i.e. we assume $N \geq 1$, which could be assured by a simple wrapper around the actual recursive procedure shown below.

Listing 2.3 The original Turku *Quicksort*.

```
procedure Quicksort(var a: sequence; l, r: integer);
{call for a[1..N] and a[0].key ≤ a[i].key ∀i (1 ≤ i ≤ N)}
var H: keytype;
    rt, lt: integer;
begin
  if l+1<r {sequence size larger than 2}
  then
  begin
    lt := l − 1; rt := r;
    H := a[r].key {last key as pivot}
```

```
    while lt ≤ rt do
    begin
      repeat lt := lt + 1 until a[lt].key ≥ H;
      repeat rt := rt − 1 until a[rt].key ≤ H;
      switch(a[lt], a[rt])
    end {while}
    switch(a[lt], a[rt]);    switch(a[lt], a[r]);
    if l < rt then Quicksort(a, l, rt);
    if lt + 1 < r then Quicksort(a, lt + 1, r)
  end
  else
    if a[l].key > a[r].key then switch(a[l], a[r])
end {Quicksort}
```

Exercise 2.1. This implementation contains two switches $a[lt] :=: a[rt]$ and $a[lt] :=: a[r]$. The first switch corrects the erroneous last exchange inside the while-loop. The second switch moves the pivot element into the middle. Given that an exchange $x :=: y$ must be programmed as three assignments $aux := x; x := y; y := aux$ in most programming languages, try to implement the two connected switches with as little assignments as possible.

Unfortunately, the design is not very elegant. The body of the procedure starts with a test $l + 1 < r$ which catches sequences of size less or equal to 2. These short sequences are handled by one direct comparison and records are switched, if and only if in wrong order. The code works also for $l = r$ (size 1) which is only needed for the extreme case that *Quicksort* is originally used on a file with one element only. It does not work for size 0 (an empty file) as mentioned before.

Recursive calls for sub-sequences of size 0 or 1 are avoided inside the body by placing a guard before the recursive call: $l < rt$, resp. $lt + 1 < r$. A test for short subfiles prior to the recursive call and then again right at the start inside the procedure seems odd at first glance. However, the point is that we should avoid recursive calls for subfiles of size 0 or 1 alltogether, rather than first call the procedure and then test for size 0 or 1 inside. The reason is that a sub-routine call with parameter passing is a fairly expensive operation as compared to in-line code. Moreover, handling short subfiles really matters. Unless we decide to stop the recursion early[1], every record in the file will eventually be treated as a subfile of size 1.

The additional test at the start of the procedure body is then effectively only a special treatment for files of size $n = 2$ which are easy to "sort" with

[1] Hoare had already suggested in his original publication to switch to another sorting method for short subfiles of length 3 or 4 [Hoa62, p. 11]. Sedgewick later showed that the cut-off point is closer to about 9 or 10 [Sed78, p. 850] and suggested to leave the subfiles untreated and to perform one scan with *Insertion Sort* on the complete file afterwards.

exactly one key comparison. As a side effect, it also catches files of size 1. In a more optimized implementation, we would leave larger subfiles (say less than $n = 11$) untreated as mentioned in the footnote.

Publications on *Quicksort* from the late Sixties often mention a *Hillmore improvement*. It turns out this refers to the actual certification [Hil62] of Hoare's algorithms 63, 64, and 65: *Partition, Quicksort, Find* in [Hoa61]. Hillmore suggested in his comment to stop the recursion on files of length 2 or less, just as we do, and reports improvements of 4–5 percent in the number of comparisons and 25–30 percent in the number of procedure calls. In the same Issue of *Communications of the ACM* which carried Hillmore's certification, Floyd introduced *Treesort* as Algorithm 113 [Flo64].

The idea to sequentially number the algorithmic knowledge in the new computing science seems odd to us in the age of the Web. The attempt stems from 1960, when a new editorial department called "Algorithms" was added to the *Communications*

> .. to publish algorithms consisting of "procedures" and programs in the ALGOL language.[Her60, p. 73]

The honor of having contributed Algorithm 1, called *QUADI*, goes to R. J. Herbold and the citation is from the preface to his article. Contributions could take on three forms, *Algorithms*, *Certifications* of previously published algorithms, and *Remarks* on previously published algorithms. The idea seems to have stem from the Computation Laboratory of the National Bureau of Standards in Washington D.C. and contributions could be send to J. H. Wegstein from the NBS. The idea was bold in several ways, not the least in picking ALGOL as the preferred programming language for the algorithms. As older readers might remember, IBM was the dominating supplier of computing equipment in those days and IBM had decided that the European-born ALGOL was a non-language.

Certifications by other authors were published in the same *Communications* afterwards and meant running an implementation of the algorithm. Often they included performance measurements from running implementations in FORTRAN or ALGOL. Hillmore's certification thus contains the following sentence:

> The body of the procedure find was corrected to read [...] and the trio of procedures was then successfully run using the Elliott ALGOL translator on the National-Elliott 803. [Hil62, p. 439]

The ALGOL compiler for the 803 was actually written by nobody else but Tony Hoare, employed by the British company *Elliott Brothers* in August 1960 as a programmer [Wik13a].


Thomas Ottmann and Peter Widmayer present a cleaner version of Wegner's Turku *Quicksort*[2] (listing 2.3) in their classical *Algorithms and Data Structures* textbook [TO12].

Listing 2.4 Textbook *Quicksort* from [TO12].

```
procedure qsort(var a: sequence; l, r: integer);
{call for a[1..N] and a[0].key ≤ a[i].key ∀i (1 ≤ i ≤ N)}
var v, i, j: integer;
begin
  if r > l {size greater 1}
  then
  begin
    i := l − 1; j := r; v := a[r].key {right key as pivot}
    while true do
    begin
      repeat i := i + 1 until a[i].key ≥ v;
      repeat j := j − 1 until a[j].key ≤ v;
      if i ≥ j then break; {i is pivot position}
      switch(a[i], a[j])
    end;
    switch(a[i], a[r]);
    qsort(a, l, i − 1); qsort(a, i + 1, r)
  end
end
```

In listings 2.2, 2.3 and 2.4 we note that in the partitioning part the repeat-loop stops on keys greater or equal to the pivot, resp. less or equal. Why stop on equality in both loops? For one thing it prevents the left pointer from running past the pivot if the pivot is the largest key in the file, which is the case for sorted input. The second argument concerns multiset input, in particular unary or binary data. Although superfluous exchanges are performed, the file is split in the middle and degeneration is avoided. This was first observed and suggested by R. C. Singleton in 1969 [Sin69].

2.1.3 Lomuto's One-Way Quicksort

So far the programs simply take the rightmost key as pivot – or the leftmost, for that matter. This is a poor choice as it leads to degeneration, i.e. a running time of $O(n^2)$, when applied to already sorted input. Several authors have

[2] T. Ottmann could rely on Wegner's work as he served as supervisor for Wegner's Habilitationsschrift at Karlsruhe, much like D. E. Knuth relied on Sedgewick's work, whose PhD supervisor he was at Stanford.

suggested better pivot selection strategies and we will consider some of them below.

We will start with a third basic variant of *Quicksort*, called Lomuto's *Quicksort*, as cited by Jon Bentley in his programming pearls series in CACM in 1984 [Ben84]. It possesses an advanced pivot selection and has several other interesting properties. The listing uses the choice of identifiers from the original publication.

Listing 2.5 Lomuto's *Quicksort* from [Ben84].

```
{sorting array X[1..N], no sentinel element required}
procedure QSort(L, U: integer);
var T: keytype; I, LastLow: integer;
begin
  if L<U {size greater 1}
  then
  begin
    swap(X[L], X[RandInt(L,U)]);
    T := X[L];
    LastLow := L;
    for I := L+1 to U do
      {Invariant: X[L+1 .. LastLow] < T and
                  X[LastLow+1 .. I-1] ≥ T}
      if X[I] < T
      then
      begin
        LastLow := LastLow + 1; swap(X[LastLow], X[I])
      end;
    swap(X[L], X[LastLow]);
    QSort(L, LastLow - 1); QSort(LastLow + 1, U)
  end
end
```

We note that the sort takes a random element from the sequence as pivot[3]. This was already suggested by Hoare in his original publication [Hoa62, p.11]. As in other randomized algorithms, e.g. randomized search, the average split position will be nearly optimal and the probability for degeneration is negligible. However, *RandInt* is an "expensive" function, therefore stopping recursion for short subfiles, not included here, is even more important. Hoare also suggested taking the pivot from a small sample of the file, but he couldn't analyse the benefits. Sedgewick showed in [Sed77b, Sed78] that the effort for taking larger samples doesn't pay off and that a median-of-three pivot selection is preferable, where the three keys are taken from the first, middle and last position. This idea goes back to Richard C. Singleton in 1969 [Sin69].

[3] $RandInt(L, U)$ returns a pseudo-random integer in the range $[L..U]$.

Secondly, the sort is a *one-way Quicksort*, i.e. the partition phase makes a single scan from left to right (cf. Figure 2.2). All records r with $r.key \geq T$ move like a wheel[4] to the right. The left half (collected records r with

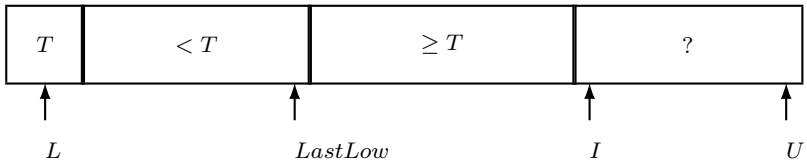


Fig. 2.2 One-way scan with partition exchange

$r.key < T$) keeps the records in their original relative order, i.e. it is *stable* in the sense defined in Subsection 1.4.4 on page 7. Records with equal keys which are added to the "wheel", however, loose their relative input order.

Moreover it is easy to see that for random input, half of the keys will be less and half will be greater than the pivot element, on the average. This implies that $N/2$ swaps take place, on the average. Is this more than in a *two-way* partition *Quicksort*? Yes, a lot more! On page 12 we mentioned $N/6$ exchanges for a file in random permutation[5].


Our claim, that Lomuto's *Quicksort* makes $N/2$ swaps on the average, was based on Bentley's argument, that

> [...] after a typical partitioning of N elements, there are about $N/2$ elements above the partition value and $N/2$ elements below it. [Ben84, p. 288]

Actually, things are a bit more complicated. In the classical two-way *Quicksort*, records "out-of-order" are searched and pairwise exchanged. In Lomuto's one-way *Quicksort*, the swap is biased. The number of swaps is always equal to the number of keys *below* the partition value, i.e. the size of the left subfile. Moreover, a swap moves only one key into the proper subfile, the other key just changes its position within the subfile it already belonged to. Now, if the left subfile becomes short, there will be less than $N/2$ swaps, should it be large, there will be more than $N/2$ swaps. As the size of the left subfile fluctuates evenly between shorter and longer, this then averages out to $N/2$ swaps.

---

[4] The term *wheel* for a sequence of items, which is moved by removing the first (last) item and placing it at the end (start), was introduced by Teuhola and Wegner in an article on minimal space duplicate deletion [TW91].

[5] Remember that we are discussing in-memory sorting at the moment. If the data reside on a rotating disk, a one-way scan with many exchanges might offset a two-way scan with fewer exchanges when blocks are fetched in reverse storage order.

It seems almost paradoxical that in the classical two-way *Quicksort*, a refined pivot selection strategy increases the number of swaps per partition phase. If we could split the file always perfectly in the middle, it would be $N/4$ swaps. For the almost perfect median-of-three selection, Sedgewick [Sed77b] shows that it is about $N/5$. For a single, randomly picked pivot element, we will show below with some help from combinatorics that $N/6$ is the asymptotic value as mentioned.

Why then search for a good pivot selection strategy, if it increases the number of swaps? Fact is that an even split greatly reduces the number of recursion steps which outweighs the larger data movement per split by far and thus yields better overall performance.

2.2 Sedgewick's Analysis

2.2.1 The Number of Exchanges in the Partitioning Stage

The following analysis regarding the number of exchanges in the partitioning stage is based on [Sed77b, p. 333f]. Assuming random permutation of input, selecting the first record as pivot element is as good a choice as any other.

If $a[1]$ has the kth smallest key in the file, what is the number of keys $> a[1].key$ in $a[2], a[3], \ldots, a[k]$? The probability, that there are t of them, is

$$p(t) = \frac{\binom{N-k}{t}\binom{k-1}{k-1-t}}{\binom{N-1}{k-1}} = \frac{A \cdot B}{C}.$$

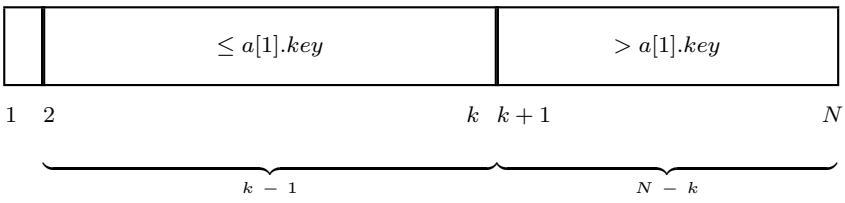


Fig. 2.3 Final arrangement with kth largest key as pivot

In case your knowledge of combinatorics is as rusty as mine, here is some help.

C is the number of combinations (orderings ignored) without repetition of $N-1$ elements to the $k-1$st class, which is the same as the number of ways to pick $k-1$ elements out of a set of $N-1$ elements. A is the number of ways to pick t elements greater than the pivot key out of $N-k$ elements. Finally, B is the number of ways to pick the others $k-1-t$ elements less than or equal to the pivot out of $k-1$ elements.

Now form the average, where the pivot key is the kth largest ($1 \le k \le N$) with probability $1/N$, and sum over all t to get the expected value

$$E = \frac{1}{N} \sum_{k=1}^{N} \sum_{t=0}^{k-1} t \cdot \frac{\binom{N-k}{t}\binom{k-1}{k-1-t}}{\binom{N-1}{k-1}} . \tag{2.1}$$

This might look terrifying, but is actually harmless thanks to *Vandermonde's convolution*, published in 1772, which – according to Knuth [Knu68, p. 58] – was already known in Chu Shih-chieh's 1303 treatise:

$$\sum_{k} \binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n} \qquad \text{integer } n.$$

We start by moving the denominator out of the last sum, as it doesn't depend on the summing variable t.

$$\begin{aligned} E &= \frac{1}{N} \sum_{k=1}^{N} \frac{1}{\binom{N-1}{k-1}} \sum_{t=0}^{k-1} t \binom{N-k}{t}\binom{k-1}{k-1-t} \\ &= \frac{1}{N} \sum_{k=1}^{N} \frac{1}{\binom{N-1}{k-1}} \sum_{t=0}^{k-1} \frac{t\ (N-k)!\ (k-1)!}{t!\ (N-k-t)!\ (k-1-t)!\ t!} \\ &= \frac{1}{N} \sum_{k=1}^{N} \frac{1}{\binom{N-1}{k-1}} \sum_{t=0}^{k-1} \frac{(N-k)!}{(N-k-t)!\ t!} \frac{(k-1)\ (k-2)!}{(k-1-t)!\ (t-1)!}. \end{aligned} \tag{2.2}$$

Now remove $(k-1)$ from the last sum, reformulate into binomial form

$$= \frac{1}{N} \sum_{k=1}^{N} \frac{k-1}{\binom{N-1}{k-1}} \sum_{t=0}^{k-1} \binom{N-k}{t}\binom{k-2}{k-1-t} \tag{2.3}$$

and apply Vandermonde's convolution

$$= \frac{1}{N} \sum_{k=1}^{N} \frac{k-1}{\binom{N-1}{k-1}} \binom{N-2}{k-1} = \frac{1}{N} \sum_{k=1}^{N} \frac{(k-1)(N-k)}{N-1} \tag{2.4}$$

$$= \frac{1}{N(N-1)} \sum_{k=1}^{N} (kN - k^2 - N + k)$$

$$= \frac{1}{N(N-1)} \left[\frac{N^2(N+1)}{2} - \frac{N(N+1)(2N+1)}{6} - N^2 + \frac{N(N+1)}{2} \right]$$

$$= \frac{1}{N-1} \cdot \frac{3N(N+1) - (N+1)(2N+1) - 6N + 3(N+1)}{6}$$

$$= \frac{N^2 - 3N + 2}{6(N-1)} = \frac{N-2}{6} . \tag{2.5}$$

2.2.2 The Average Number of Key Comparisons

The common way to judge a comparison-based sort is to analyse the number of key comparisons which is needed to sort N records on the average and possibly in the worst case. With *Quicksort* being a recursive algorithm, this gives rise to an analysis with a recurrence relation. In fact, for most *Computer Science* students, *Quicksort* is the first (and often only) topic in their studies, where they get into contact with this useful technique.

Assume the number of key comparisons in partitioning a file of size N is $N+1$ (two extra comparisons for the crossing of the indices). Next, each key is equally likely as pivot and therefore all subfile lengths between 0 and $N-1$ occur with equal probability. At least one record is split off, subfiles of length 0 and 1 are not treated any further. This yields the following recurrence relations:

$$\bar{c}(N) = N + 1 + \frac{1}{N} \sum_{s=1}^{N} \left(\bar{c}(s-1) + \bar{c}(N-s) \right) , \tag{2.6}$$

for $N \geq 2$, $\bar{c}(0) = \bar{c}(1) = 0$, and for $N+1$ we get

$$\bar{c}(N+1) = N + 2 + \frac{1}{N+1} \sum_{s=1}^{N+1} \left(\bar{c}(s-1) + \bar{c}(N+1-s) \right) . \tag{2.7}$$

The "trick" is now to multiply equation (2.6) by N and (2.7) by $N+1$ to get rid of the fractions and then to subtract (2.7) from (2.6).

$$N\bar{c}(N) - (N+1)\bar{c}(N+1) = N(N+1) - (N+1)(N+2) + 2\sum_{s=1}^{N-1}\bar{c}(s) - 2\sum_{s=1}^{N}\bar{c}(s) \tag{2.8}$$

$$\bar{c}(N+1) = (N+2) - N + \frac{N+2}{N+1}\,\bar{c}(N) \tag{2.9}$$

Since we are after a closed formula for $\bar{c}(N)$, we substitute N for $N+1$ with the caveat that they are now only valid for $N \geq 3$. We then need to unravel the recurrence for a while to see what kind of series we get.

$$\begin{aligned}
\bar{c}(N) &= 2 + \frac{N+1}{N}\,\bar{c}(N-1) \\
&= 2 + \frac{N+1}{N}\,\left(2 + \frac{N}{N-1}\,\bar{c}(N-2)\right) \\
&= 2 + \frac{2(N+1)}{N} + \frac{N+1}{N-1}\,\left(2 + \frac{N-1}{N-2}\,\bar{c}(N-3)\right) \\
&= 2 + \frac{2(N+1)}{N} + \frac{2(N+1)}{N-1} + \cdots + \frac{N+1}{4}\,\left(2 + \frac{4}{3}\,\bar{c}(2)\right)
\end{aligned} \tag{2.10}$$

and with $\bar{c}(2) = 3$ we arrive at

$$= 2 + 2(N+1)\,\left[\frac{1}{N} + \frac{1}{N-1} + \cdots + \frac{1}{4}\right] + N + 1. \tag{2.11}$$

In equation (2.11) we detect (with three missing elements) the series

$$\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} + 1 = \sum_{i=1}^{n}\frac{1}{i}$$

known as the nth *harmonic number* H_n (see p. 138 in the Appendix). So we end up with

$$\begin{aligned}
\bar{c}(N) &= 2 + 2(N+1)\,\left(H_N - \frac{1}{3} - \frac{1}{2} - 1\right) + N + 1 \\
&= 2(N+1)H_N - 8N/3 - 2/3 = 2(N+1)(H_{N+1} - 4/3).
\end{aligned} \tag{2.12}$$

The nth harmonic number H_n has a handy estimate, because always

$$\ln n \leq H_n \leq \ln n + 1.$$

The analysis thus shows that *Quicksort* without optimizations sorts a file of size N with N distinct values in a random permutation with about $2N \ln N$ key comparisons which is asymptotically the same as $O(N \log N)$. Note, how-

ever, that the analysis above assumes that the partition phase returns two subfiles (or more subfiles in refined variants) which each are again in random order.

2.3 Quicksort in the Sixties and Seventies

2.3.1 Summing Up the Classical Results

Tony Hoare invented *Quicksort* in 1961. Throughout the Sixties variants with suggested improvements appeared. The precise analysis, including the multiset case (cf. Chapter 3), came from Sedgewick in the late Seventies as the result of doing his PhD on *Quicksort* under the supervision of Donald Knuth.

The following statements summarize what was generally known about and recommended for *Quicksort* up to, say, the Mid-Seventies.

- Under unlucky circumstances, the performance of *Quicksort* can degenerate. The worst case complexity is thus $c_{max}(N) = O(N^2)$.
- The average complexity of *Quicksort* (as measured by the number of key comparisons) is asymptotically optimal and can be analyzed by means of the recurrence relation

$$\bar{c}(N) = \{N-1, N, N+1\} + \frac{1}{N} \sum_{s=1}^{N} (\bar{c}(s-1) + \bar{c}(N-s))$$

  for $N \geq 2$ and $\bar{c}(0) = \bar{c}(1) = 0$. In variants where the partitioning phase takes $N+1$ comparisons, this gives

$$2(N+1)(H_{N+1} - \frac{4}{3}) = O(N \log\ N).$$

- For single pivot selection and input files in a random permutation, the number of exchanges per partition phase is $N/6$.
- Roger S. Scowen [Sco65] originally proposed to take the middle element as pivot instead of a random element as suggested by Hoare. However, a noticeable improvement in average performance and a better guard against degeneration is achieved with a median-of-three pivot selection, where the three elements are picked from first, middle, and last element. Also the scan for records, which are to be exchanged, should stop on $\leq$ and $\geq$ instead of $<$ and $>$. These last two points go back to [Sin69].
- The recursion should stop for short subfiles. The idea seems to also go back to Scowen [Sco65], who termed his algorithm QUICKERSORT. Sedgewick [Sed77b, Sed78] later confirmed $n = 11$ as a good cut-off point

and proposed to use a single insertion sort run on the total file at the very end of the sort[6].

- Sorting the shorter subfile first reduces the recursion stack to at most $\log_2 N$ depth [Sco65].
- Since *Quicksort* is a classical case of tail recursion[7], the second recursive call can be eliminated with a jump (and some variable assignments) to the start of the routine.
- Explicit recursive calls can be eliminated all together and be replaced by a self-administered stack. Improvements, if any, will be minimal at the expense of a greatly complicated algorithm. This idea also goes back to Scowen's QUICKERSORT.
- No speed-up can be expected on pre-sorted or multiset input for the variants presented so far.

In 1974, Rudolf Loeser did a comprehensive performance comparison of several *Quicksort* variants [Loe74], including QUICKERSORT from Scowen [Sco65], QSORT from van Emden [vE70b, vE70a], Hoare's original algorithm, Floyd's *Heapsort* (TREESORT3), *Shellsort* and a few others.

QSORT is a *Quicksort* optimization proposed by Marteen H. van Emden, who had further analyzed the average performance of *Quicksort* [vE70b]. He suggested using an interval $[X, Z]$ as "pivot element", s.t. all keys $\leq X$ are in F_1, keys $\geq Z$ are in F_2. The interval is refined as the partioning proceeds and partitioning ends with a middle file (pair) $\{X, Z\}$. The actual algorithm *qsort* for this improvement was given in [vE70a].

Van Emden went on to show that this interval technique reduced the average number of key comparisons $\bar{c}$ to about $1.140N \log_2 N$ as compared to Hoare's $2N \ln N \approx 1.386N \log_2 N$. Loeser, however, observed an increase of the actual measured running time of *qsort* over the original *Quicksort* by about 50 percent.

It is certainly wise to be sceptical when authors claim substantial improvements for their *Quicksort* variants as compared to versions which were already on the "market" in the early Seventies. In that respect, Robert Sedgewick's very refined version [Sed78] (Corrigendum: CACM 22(6): 368 (1979)) with median-of-three pivot selection, insertion sort for subfiles of size $M \leq 9$, own stack management, sets the mark against which to measure yourself. From own experimental data with random and with certain biased inputs, the author of this monograph observed that the fairly straight-forward program in listing 2.3 runs about 10 percent slower than Sedgewick's ultimate recommendation. A table giving some recorded running times follows on page 55 below.

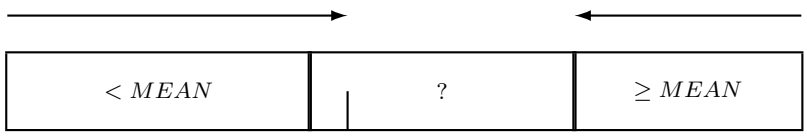
---

[6] Knuth [Knu98, p. 389] cites LaMarca and Ladner [LL97] who found out that in modern caching systems it is actually better to sort short subfiles immediately and not to wait till the end.

[7] *Tail recursion* is given when the body of a recursive function or procedure F has a recursive call of F as its last statement.

2.3.2 The Case Against Meansort

In 1983, Dalia Motzkin published a *Quicksort*-variant, which she called MEANSORT [Mot83]. As the name suggests, the algorithm uses the computed mean of a subfile as pivot value to partition that subfile. The mean is computed from the sum of the keys in each respective subfile, as illustrated in Figure 2.4.



$if\ K_i < MEAN\ then\ \{LEFT\_SUM \leftarrow LEFT\_SUM + K_i; i \leftarrow i + 1\}\ else \ldots$

Fig. 2.4 Summing up key values as the partitioning progresses

The idea of a "fictitious pivot value", thought to possibly lie between two keys, is not entirely new. Knuth [Knu98, p. 128] reports in Vol. 3 of TAoCP that John McCarthy had suggested very early using (min + max)/2, where the minimal and maximal key value of a subfile is determined "on the fly" much as in Meansort. Similarly, a fictitious pivot value, e.g. (first + last)/2, is useful in external or linked list *Quicksort* versions, either because placing an actual pivot element into its final position at the end of the split would require an extra read/write operation, or because the middle key for a median-of-three strategy is not accessable. However, there is no reason why the selection of this fictitious pivot value should be based on more than a few (say 2 to 5) probes, as shown in Sedgewick's analysis [Sed77b].

Nevertheless, Motzkin concludes her contribution with strong claims regarding the qualities of her modifications.

> Meansort is a considerable improvement over standard Quicksort. It has a better average behavior, the worst case occurs less often, and it is efficient in situations where there are repeating keys. Space requirements are minimal (as in most other versions of Quicksort) since the file is sorted in place, and the stack requirements are $\log_2 N$, provided that smaller subfiles are sorted first. This sort is not stable, but the Stable Quicksort [8] ([Mot81] note added in quote) technique may be applied here, if stability is needed. [Mot83, p. 251]

The actual algorithm published in the *Communications* is shown – in the original style – in Listing 2.6. There K_I, K_J denote the keys of records R_I, R_J.

Listing 2.6 Meansort from [Mot83] with corrections included.

PROCEDURE MEANSORT (M, N, MEAN)

```
IF M < N THEN
  LEFT_SUM ← 0
  RIGHT_SUM ← 0
  I ← M
  J ← N
  LOOP
    WHILE K_I < MEAN AND I ≠ J DO
      LEFT_SUM ← LEFT_SUM + K_I
      I ← I + 1
    END WHILE
    WHILE K_J ≥ MEAN AND I ≠ J DO
      RIGHT_SUM ← RIGHT_SUM + K_J
      I ← J − 1
    END WHILE
    IF I ≠ J THEN INTERCHANGE (R_I, R_J) ELSE
    EXIT LOOP ENDIF
  FOREVER
  IF I = M THEN RETURN ENDIF
  RIGHT_SUM ← RIGHT_SUM + K_J
  SUB_FILE_MEAN ← LEFT_SUM / (I − M)
  CALL MEANSORT(M, I − 1, SUB_FILE_MEAN)
  SUB_FILE_MEAN ← RIGHT_SUM/(N − J + 1)
  CALL MEANSORT(J, N, SUB_FILE_MEAN)
ENDIF
END OF PROCEDURE
```

The rather bold statement cited above, together with a few typographical errors, elicted a large number of responses. The editor of *Computing Pratices*, at that time Marie Jensen, decided to not publish the individual correspondences but to ask Dalia Motzkin for a comprehensive reply, which appeared as *Technical Correspondence* in July 1984 [MK84].

The reply addressed some of the obvious weaknesses of the approach.

- The first pass has no *MEAN* available; Motzkin suggests to start with

  $$\text{if } K_{first} \leq K_{last} \text{ then } MEAN := K_{first} + 1 \text{ else } MEAN := K_{last} \ .$$

- Summing up large integers can lead to an overflow.
- The published paper does not work for a file $F\langle 2, 1\rangle$ unless *MEAN* is of type **real**.
- The reported number of key comparisons were too low as they omitted the comparisons in the first pass.

The reply also contained a revised version MSort with modified loops and new measurement tables, including a comparison to a median-of-three *Quicksort* called SSort. The reply still claimed a 10 percent superiority of MSort over SSort. It did not contain running times.

In an unpublished reply to Motzkin's original contribution in Communications, Wegner implemented a revised Pascal version of MEANSORT. Listing 2.7 shows the program but without stopping for short subfiles. For the first pass, the value for *MEAN* is computed from the sum of all keys which does not add any key comparisons. After all, if having a precise pivot key is so important, why not apply the mean computation to the first split as well.

Listing 2.7 Wegner's revised Pascal version of MEANSORT [Mot83].

```
procedure Meansort(left, right: index);
var k: index; firstsum: keytype;
  procedure MSort(el, r: index; mean: keytype);
  label 99;
  var ls, rs: keytype {sum left, resp. right subfile};
    i, j: index {pointer into file};
  begin
    if el < r then
    begin {filesize > 1}
      i := el; j := r; ls := 0; rs := 0;
    99:
      while a[i].key ≤ mean do
      begin ls := ls + a[i].key; i := i + 1
      end;
      while a[j].key > mean do
      begin rs := rs + a[j].key; j := j − 1
      end;
      if i < j then
      begin {pointers have not crossed}
        ls := ls + a[j].key; rs := rs + a[i].key;
        switch(a[i], a[j]);
        i := i + 1; j := j − 1; goto 99
      end;
      if i ≤ r then
      begin
        MSort(el, i − 1, ls div (i − el));
        MSort(j + 1, r, rs div (r − j))
      end
    end {filesize > 1 − else skip}
  end {MSort};
begin {nonrecursive part of Meansort frame}
  firstsum := 0;
  for k := left to right do
    firstsum := firstsum + a[k].key;
  a[right + 1] := firstsum {stopper in a[N+1]};
  MSort(left, right, firstsum div (right − left + 1))
end;
```

When running this version of MEANSORT with *Insertion Sort* for short files of length $M < 11$, the running times were about 30 percent higher than with a comparable *Quicksort*. Other published versions, e.g. an optimized MEANSORT by Klaus Lagally and Bernhard Ziegler from Stuttgart University [LZ85], reported similar results. It is therefore safe to conclude that the elaborate pivot computation suggested in MEANSORT does not pay off.

Chapter 3
A Smooth and Stable Quicksort for Linked Lists

3.1 Teaching an Old Dog Some New Tricks

3.1.1 The Classical Sorting Problem versus Multisets

As mentioned in Chapter 1, the *Sorting Problem* is to bring N records (keys) into ascending (descending) order using pairwise comparisons. Quite often the key values form a *multiset*, as in the table below, resembling a flight reservation system.

Name	Flight#
Adams	265
Arnold	107
Atkins	911
Baldwin	107
Bell	699
Burns	265
Byron	107
Carter	480
Chou	265
Davis	699
⋮	⋮

sort
$\Longrightarrow$
Key = Flight#

Flight#	Name
107	Arnold
107	Baldwin
107	Byron
⋮	⋮
265	Adams
265	Burns
265	Chou
⋮	⋮

Sorting multisets brings up two issues: *Stability* and *Smoothness* (cf. the Sections 1.4.4 and 1.4.5 for definitions).

3.1.2 Stability

Stability is not an issue if extra space is provided as in the classical *Mergesort*. If no or little extra space is provided, as in *Quicksort* or *Heapsort*, things become more complicated. As a matter of fact, both *Quicksort*, as array version, and *Heapsort* are nonstable, i.e. equal keys do not retain their relative order. Smoothness is considered further below.

 Stable, in situ sorting has always been a challenge. Knuth formulated it as an exercise in his first edition of vol. 3 of TAoCP [Knu73b, p. 388].

> *(Stable sorting in minimum storage.)* A sorting algorithm is said to require *minimum storage* if it uses only $O((\log N)^2)$ bits of memory space for its variables besides the space needed to store the N records. [...]
>
> Design a stable minimum-storage sorting algorithm which requires only $O(N(\log N)^2)$ units of time in its worst case. [*Hint:* It is possible to do stable minimum-storage merging in $O(N \log N)$ units of time.]

In the answers to the exercises in his first edition [Knu73b, p. 665] Knuth gives a solution for stable minimum-storage merging, which is due to V. Pratt, and quotes the L. Trabb Pardo paper [Par77] as

> ... the best possible answer to this problem: It is possible to do stable merging in $O(n)$ time and stable sorting in $O(n \log n)$ time, using only $O(\log n)$ bits of auxiliary memory for a fixed number of index variables.

In his second edition [Knu98, p. 701f], Knuth adds to the L. Trabb Pardo result [Par77] versions with improved constants by B.-C. Huang and M. A. Langston [HL92] and mentions another stable merging algorithm for M items with N items when M is much smaller than N [Sym95].

We will return to V. Pratt's solution and the Trabb Pardo algorithm in Chapter 12 which is devoted to the aspects of in situ merging. In the *Turku Lecture*, three other stable, minimum storage sorting methods by Rivest, Preparata and Dudzinski&Dydek were mentioned, although this selection was somewhat arbitrary.

- Ronald L. Rivest [Riv73] presented a fast, stable, minimum storage algorithm in 1973. Its average running time is $O(N(\log N)^2)$, but the worst case is still $O(N^2)$.
- Franco P. Preparata [Pre75] gave a fast, stable sorting algorithm with absolutely minimum storage, defined as $O(\log N)$ extra bits. Its average running time is again $O(N(\log N)^2)$.
- Dudzinski and Dydek came up in 1980 with another stable merging algorithm [DD81]. Their merging of vectors of length M and N ($M \leq N$) needs $O(M \log(N/M + 1))$ comparisons and $O((M + N) \log M)$ assignments in O(log M) space.

Their discussion has also been delegated to Chapter 12 because they are more related to stable, in situ merging than to *Quicksort*.

In 1981, Dalia Motzkin was the first to note that *Quicksort*, when applied to a linked list, yields a stable, $O(N \log N)$ average time sort [Mot81]. Wegner [Weg82] independently proposed a *Quicksort* variant for linked lists and considered a *three-way split* for eliminating records with keys equal to the pivot element from recursion. This leads to an *Quicksort* with average running time $O(N \log n)$, where N is the total number of records to sort and n is the number of distinct key values among them ($1 \leq n \leq N$).

However, algorithms sorting linked lists play in a different league and should not be compared directly with random access internal sorts.

3.1.3 Smoothness

With respect to smoothness for data stored in an array, we observe that *Quicksort* so far fails to pick up sequences of equal keys or – even worse – degenerates to $O(N^2)$ running time. The same holds for presortedness, which does not help *Quicksort*, but rather poses a risk for degeneration. Indeed, Sedgewick summarizes in his fundamental analysis of *Quicksort* with equal keys the situation as follows [Sed77a, p. 244]:

> Now, if we wish to use *Quicksort* to sort a file containing equal keys, we must decide how to treat keys equal to the partitioning element during the partitioning process. Ideally, we would like to get all of them into position in the file, with all the keys with a smaller value to their left, and all the keys with a larger value to their right. Unfortunately, no efficient method for doing so has yet been devised, so we shall have some keys equal to the partitioning element in the left subtile and some in the right.

Much of the following will be devoted to the argument, that *there are efficient methods* for doing so, not only in linked lists [Weg82, Weg83], but in random-access storage as well [Weg85].

3.1.4 A Global View of Sorting with Respect to Smoothness and Stability

A very rough classification of sorting as of the early Eighties is given in the following table, where *Mergesort*, *Heapsort* and *Quicksort* refer to the classical textbook variants. A check mark (tick) in brackets, as seen in the *presorted* column, denotes partial fulfillment, e.g. because the algorithms need additional "twists" to profit from presorted input.

The table also includes other notable solutions for smoothness with respect to presortedness, among them an algorithm by C. R. Cook and Do Jiu Kim

for nearly sorted lists [CK80][1] and Edsger W. Dijkstra's *Smoothsort* [Dij82]. Trabb Pardo stands for the class of sorts based on stable, in situ merging.

The last two rows represent a class of *Quicksort* variants aimed at smoothness as described in [Weg83, Weg85].

	Smooth on			
	Multisets	Presorted	Stable	In Situ
Mergesort	–	(√)	√	–
Quicksort	–	–	–	√
Heapsort	–	–	–	√
Trabb Pardo [Par77]	–	–	√	√
Smoothsort [Dij82]	√	√	–	√
Cook + Kim [CK80]	√	√	–	–
Linked List Quicksort	√	(√)	√	√
Smooth Array Quicksort	√	(√)	–	√

3.2 A Linked List Solution

3.2.1 TRISORT – A Three-Way Linked List Quicksort

As mentioned above, Dalia Motzkin [Mot81] was the first to observe stability in a one-way linked list *Quicksort*. However, she did not provide any analysis nor did she suggest a three-way ($<$, $=$, $>$) split. Independently, Wegner [Weg82] suggested sorting a linked list with a *Quicksort* variant incorporating such a three-way split, which guarantees stability and promises smoothness, and analyzed the algorithm, which he called TRISORT.

Obviously, TRISORT is stable and eliminates all records with key value equal to the pivot from further recursion. Wegner suggests taking the last key as pivot, because the test on "last key" needs to be done only when $k_i = H$ is true, i.e. we have a kind of sentinel technique. The actual implementation in PASCAL is not so pretty, mostly because the handling of linked lists with explicit pointers is awkward.

[1] In the article, Cook and Kim gave no name to their hybrid sort, calling it simply *New Sorting Algorithm*. They describe it as a combination of Quickersort [Sco65] and Straight Insertion Sort with merging [CK80, p. 622].

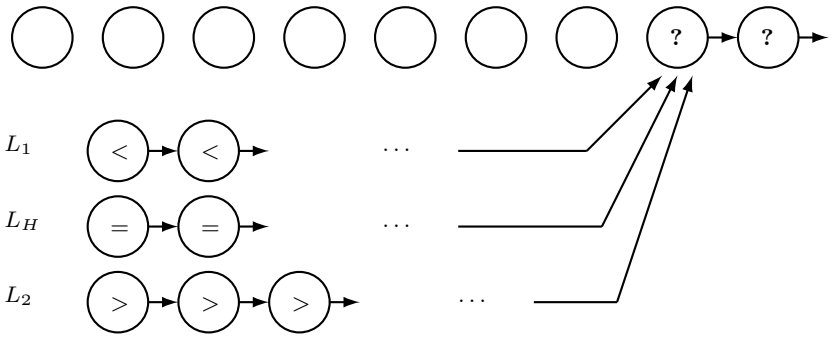


Fig. 3.1 TRISORT one-way scan with three-way split [Weg82].

Listing 3.1 Basic loop to merge sequences in place.

```
procedure TRISORT (var first, last : ref);
{linked list from first to last, 3 global dummy nodes sl, sh, sr}
var sfl, sfr, sfh, lt, rt, ht, p : ref;
  H : keytype;
begin
  if first <> last {|L| > 1}
  then
  begin
    lt := sl; rt := sr; ht := sh; sh↑.next := first;
    p := sh; H := last↑.key {Pivot = last};
    repeat {terminates with p = last}
      p := p↑.next;
      if p↑.key < H
      then
        repeat {node goes to L1 }
          lt↑.next := p; lt := p; p := p↑.next
        until p↑.key >= H; {node ↛ L1}
      while p↑.key > H do
      begin {node goes to L2}
        rt↑.next := p; rt := p; p := p↑.next;
        if p↑.key < H
        then
          repeat {node goes to L1 }
            lt↑.next := p; lt := p; p := p↑.next
          until p↑.key >= H; {node ↛ L1}
      end {while; post−condition: p↑.key = H};
      ht↑.next := p; ht := p
    until p = last {list partitioned};
```

```
    sfl := sl ↑ .next; sfr := sr ↑ .next; sfh := sh ↑ .next
    {pointers to start of lists}
    if lt <> sl
    then
    begin {L1 not empty}
      TRISORT (sfl, lt);
      first := sfl; lt ↑ .next := sfh
    end {of sorting L1 and relinking to LH}
    else first := sfh; {if L1 is empty, L starts with LH}
    if rt <> sr
    then
    begin {sub−list L2 not empty}
      TRISORT (sfr, rt);
      ht ↑ .next := sfr; last := rt; {link LH to L2}
      last ↑ .next := nil {last node has nil pointer}
    end {of sorting L2}
  end {|L| > 1}
end {TRISORT};
```

3.2.2 A Multiset Analysis of TRISORT

For the analysis of a three-way split *Quicksort* we can rely on the fundamental research of Sedgewick on *Quicksort with Equal Keys* [Sed77a].

Without restriction of generality we are sorting integers $1, \ldots, n$. In case of a *multiset* file $S = \{x_1 \cdot 1, x_2 \cdot 2, \ldots, x_n \cdot n\}$ integer i occurs x_i-times with $x_1 + x_2 + \ldots + x_n = N$.

Special cases.

$n = N$	no duplicates
$n = 2$	a binary file
$n = 1$	a unary file
$x_i = M = \dfrac{N}{n}\ \forall\ 1 \leq i \leq n$	a constant repetition factor M

By definition, a file F is random iff each of the

$$\binom{N}{x_1, x_2, \ldots, x_n} = \frac{N!}{x_1! x_2! \ldots x_n!}$$

permutations is equally likely.

If we now decide to count binary PASCAL-comparisons, rather than the ternary MIX-comparisons, we observe for TRISORT that we have

exactly 1 key comparison	$\forall\ k_i < H$,
exactly 2 key comparisons	$\forall\ k_i \geq H$ and
exactly 1 pointer comparison	$\forall\ k_i = H$.

Therefore the average number of key comparisons is

$$\begin{aligned}\bar{c}\,(x_1,\ldots,\ x_n) =& \frac{1}{N}\sum_{1\le i\le n} x_i\,(1\cdot(x_1+\ldots+x_{i-1})\ +\ 2\cdot(x_i+\ldots+x_n))\\ &+\frac{1}{N}\sum_{1\le i\le n} x_i\,(\bar{c}\,(x_1,\ldots,\ x_{i-1})+\bar{c}\,(x_{i+1},\ldots,\ x_n)) \quad (3.1)\end{aligned}$$

with $\bar{c}(x) = 2x$. This comes out to

$$\bar{c}(x_1,\ldots,\ x_n) = \sum_{1\le i\le n} 2x_i + 3\sum_{1\le k<i\le n}\frac{x_k x_i}{x_k+\ldots+x_i} \quad (3.2)$$

The cases with a constant repetition factor are the most intuitive to compare.

TRISORT with PASCAL comparisons	$3(N+M)H_n - 4N$
lower bound with `MIX` comparisons	$2(N+M)H_n - 3N - n$

In other words, TRISORT reaches the lower bound for multiset *Quicksort* programs as we will show below. To do so, we need a bit of mathematics for recurrence relations with multinomials.

3.2.3 The Sedgewick and Burge Tricks

The following techniques for solving multinomial recurrence relations are taken from [Sed77a, Bur76]. Start with multiplying the basic recurrence relation 3.1 by N.

$$\begin{aligned}N\cdot\bar{c}\ (x_1,\ldots,\ x_n) =& \sum_{1\le i\le n} x_i\Big(1\cdot(x_1+\ldots+x_{i-1})+2\cdot(x_i+\ldots+x_n)\Big)\\ &+\sum_{1\le i\le n} x_i\Big(\bar{c}(x_1,\ldots,\ x_{i-1})+\bar{c}(x_{i+1},\ldots,\ x_n)\Big) \quad (3.3)\end{aligned}$$

Take the same formula for $x_2,\ldots,x_n$.

$$\begin{aligned}(N-x_1)\bar{c}\ (x_2,\ldots,\ x_n) =& \sum_{2\le i\le n} x_i\Big(1\cdot(x_2+\ldots+x_{i-1})+2\cdot(\ldots)\Big)\\ &+\sum_{2\le i\le n} x_i\Big(\bar{c}(x_2,\ldots x_{i-1})+\bar{c}(\ldots)\Big) \quad (3.4)\end{aligned}$$

Subtract 3.4 from 3.3.

$$N\bar{c}(x_1,\ldots,\ x_n) - (N - x_1)\ \bar{c}\ (x_2,\ldots,\ x_n) = \\ 3x_1 \sum_{1\le i\le n} x_i - x_1^2 + x_1\ \bar{c}(x_2,\ldots,\ x_n) \\ + \sum_{2\le i\le n} x_i \left(\bar{c}(x_1,\ldots,\ x_{i-1}) - \bar{c}(x_2,\ldots,\ x_{i-1})\right) \tag{3.5}$$

Substitute $G(x_1,\ldots,\ x_n)$ for $\bar{c}(x_1,\ldots,\ x_n) - \bar{c}(x_2,\ldots,\ x_n)$ and difference again, this time with $G(x_1,\ldots,\ x_{n-1})$.

$$N \cdot G(x_1,\ldots,\ x_n) = 3x_1 \sum_{1\le i\le n} x_i - x_1^2 \\ + \sum_{2\le i\le n} x_i\ G(x_1,\ldots,\ x_{i-1}) \tag{3.6}$$

$$(N - x_n)G(x_1,\ldots,\ x_{n-1}) = 3x_1 \sum_{1\le i\le n-1} x_i - x_1^2 \\ + \sum_{2\le i\le n-1} x_i\ G(x_1,\ldots,\ x_{i-1}) \tag{3.7}$$

Subtract 3.7 from 3.6 and telescope the right-hand sides.

$$\begin{aligned} G(x_1,\ldots,\ x_n) &= G(x_1,\ldots,\ x_{n-1}) + \frac{3x_1 \cdot x_n}{x_1 + \ldots + x_n} \\ &= G(x_1,\ldots,\ x_{n-2}) + \frac{3x_1x_{n-1}}{x_1 + \ldots + x_{n-1}} + \frac{3x_1x_n}{x_1 + \ldots + x_n} \\ &\vdots \\ &= G(x_1) + \sum_{2\le i\le n} \frac{3x_1x_i}{x_1 + \ldots + x_i} \end{aligned} \tag{3.8}$$

Resubstitute now:

$$\begin{aligned} \bar{c}(x_1,\ldots,\ x_n) &= \bar{c}(x_2,\ldots,\ x_n) + \bar{c}(x_1) + 3 \sum_{2\le i\le n} \frac{x_1x_i}{x_1 + \ldots + x_i} \\ &= \bar{c}(x_3,\ldots,\ x_n)\ + \bar{c}(x_2)\ + \\ &\qquad + 3 \sum_{3\le i\le n} \frac{x_2x_i}{x_2 + \ldots + x_i} + \bar{c}(x_1) + 3 \sum_{2\le i\le n} \ldots \\ &= \sum_{1\le i\le n} \bar{c}(x_i) + 3 \sum_{1\le k<i\le n} \frac{x_kx_i}{x_k + \ldots + x_i}. \end{aligned} \tag{3.9}$$

This is the result 3.2 from above. Although it is precise, it is not very useful in terms of insight. However, we can compare it to the lower bound which Sedgewick gives for multiset sorting with *Quicksort* programs (see discussion in Section 3.2.4 below).

For the moment, take again the special case $N/n = M = x_i \quad \forall i$.

$$\begin{aligned}\bar{c}(n \cdot M) &= 2n \cdot M + 3 \sum \frac{M^2}{(i-k+1)M} \\ &= 2n \cdot M + 3M \sum_{1 \le k \le n-1} \; \sum_{k+1 \le i \le n} \frac{1}{i-k+1} \\ &= 2nM + 3M \sum_{1 \le k \le n-1} \; \sum_{1 \le i \le n-k} \underbrace{\frac{1}{(i+k)-k+1}}_{i+1}. \end{aligned} \tag{3.10}$$

The "trick" here is to shift the index from $i = k+1, \ldots, n$ to $i = 1, \ldots, n-k$. Now, with a bit of help from the Appendix A concerning *harmonic numbers* H_n, we get the TRISORT result with PASCAL-comparisons from p. 35 above.

$$\begin{aligned}\bar{c}(n \cdot M) &= 2nM + 3M \sum_{1 \le k \le n-1} (H_{n-k+1} - 1) \\ &= \ldots = 2nM + 3M\Big(\sum_{1 \le i \le n} H_i - n \Big) \qquad (3.11) \\ &= 3(N+M)H_n - 4N. \qquad (3.12)\end{aligned}$$

As an aside, note that the original publication [Weg82] suffered from a printing error in equation 3.11.

3.2.4 Sedgewick's Lower Bound

The art in algorithmic complexity theory is not only to analyze the performance of a given solution to a general problem, here sorting, but to come up with a *lower bound* for the computational effort, which no algorithm in a reasonable model can underscore. To this end, Sedgewick defined the class of *Quicksort programs* [Sed77a, p. 244]:

> A program is called a *Quicksort program* if it
>
> - splits a file into 3 subfiles, s.t. [...],
> - considers one element to select the pivot,
> - generates two random sequences from a given random sequence.

Observation: The linked list *Quicksort* programs, e.g. TRISORT, are *Quicksort* programs according to the definition above.
Q: How good is TRISORT?
A: It is optimal!

The answer follows from the lower bound which Sedgewick gives [Sed77a]. He proves that any *Quicksort* program will take at least

$$\bar{c}(x_1, \ldots, x_n) = N - 1 + \frac{1}{N} \sum_{1 \le j \le n} x_j \Big(\bar{c}(x_1, \ldots, x_{j-1}) + \bar{c}(x_{j+1}, \ldots, x_n) \Big) \tag{3.13}$$

comparisons, on the average, to sort a *multiset* file $S = \{x_1 \cdot 1, x_2 \cdot 2, \ldots, x_n \cdot n\}$, which telescopes to

$$G(x_1, \ldots, x_n) = G(x_1) + \sum_{2 \le i \le n} \frac{2 x_1 x_i}{x_1 + \ldots + x_i} \tag{3.14}$$

Indeed, the only difference to 3.8 is a 2 in the result as against a 3 in our analysis, because Sedgewick counts three-way `MIX`-comparisons versus we work with two-way PASCAL-Comparisons as mentioned already on p. 35.

Why are we so picky about counting PASCAL-comparisons and the constants that come with this assumption? It turns out, we can come up with other strategies to speed up the sort in the presence of multisets or sorted runs in the input. This is the topic of the next chapter.

Chapter 4
Two- and Three-Way Quicksorts for Linked Lists and Arrays

4.1 MIX- vs PASCAL-Comparisons

Own experimental results suggest that the effort to split a file into *three* subfiles, using – on the average – $3N/2$ binary comparisons, does not pay off for few repetitions of keys. In other words: *"To divide less is to conquer more!"*

As we will show, there is another, new strategy. The idea is to hope for a unary subfile. If the subfile is indeed unary, there is nothing more to do. Otherwise, on the first key $\neq$ pivot element H, continue with what has been scanned so far as left subfile and split the remainder into sequences $\leq H$ and $> H$.

The analysis for this *Quicksort* variant, shown below, is very tricky and essentially follows Sedgewick's *upper bound* computation [Sed77a]. The result for the new strategy gives $\bar{c} \leq 2(N+1)H_N - 7N/3 - 1/3$ for $x_i = 1$ (all keys distinct) and $2NH_{n-1}+N$ for $x_i = M$ $(1 \leq i \leq n)$, i.e. the case with constant repetition factor. In comparison: for TRISORT we obtained $3(N+1)H_N - 4N$ and $3(N+M)H_n - 4N$.

4.2 Introducing LINKSORT

For simplicity we start with a stable version for linked lists, but (unstable) array implementations work just the same. Compared to any linked list *Quicksort*, the modifications which are required (cf. Listing 4.1) are minimal. The basic idea is shown in Figure 4.1. Clearly we need one additional key comparison when we first detect a key $\neq H$. Thereafter we test on $\leq H$. Thus, in total we have N key comparisons for a unary file and $N+1$ key comparisons otherwise in the partitioning phase for a subfile of length N.

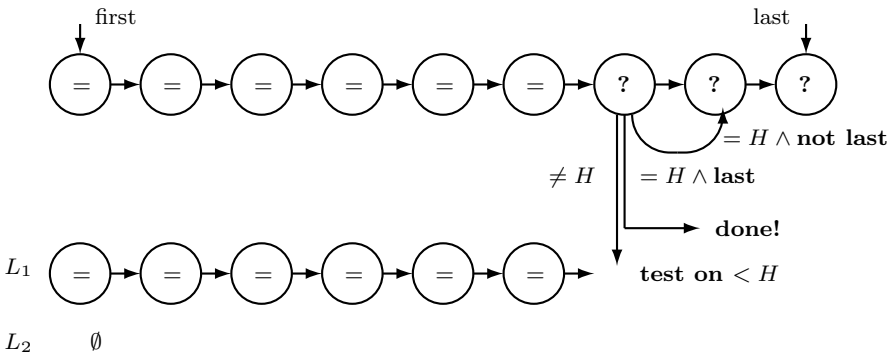


Fig. 4.1 LINKSORT with initial run of keys equal to pivot [Weg83].

Listing 4.1 LINKSORT3 – detect unary runs.

```
procedure LINKSORT3(var first, last : ref);
{linked list from first to last, 2 global dummy nodes sl and sr}
label 99;
var sf1, sf2, lt, rt : ref;
  H : keytype;
begin
  if first = last then goto 99; {exit for singleton list}
  lt := sl; lt↑.next := first; {initialize empty sublist L1}
  H := last↑.key {Pivot = last};
  while lt↑.next↑.key = H do
  begin {try for a unary file}
    if lt↑.next = last then goto 99; {yes, a unary subfile}
    lt := lt↑.next
  end;
  { list contains a key ≠ H, do ordinary two−way split}
  rt := sr; {initialize empty sublist L2}
  if lt↑.next↑.key > H
  then
  begin
    rt↑.next := lt↑.next;
    repeat
      rt := rt↑.next
    until rt↑.next↑.key ≤ H;
    lt↑.next := rt↑.next
  end
  {precondition for main partition loop: lt↑.next↑.key ≤ H}
  while lt↑.next <> last do
  begin
```

```
    repeat {replaces lt := lt↑.next;}
      lt := lt↑.next
    until lt↑.next↑.key ≥ H;
    if lt↑.next↑.key > H
    then
    begin
      rt↑.next := lt↑.next;
      repeat rt := rt↑.next until rt↑.next↑.key ≤ H;
      lt↑.next := rt↑.next
    end {then}
  end {while};
  { list now partitioned − sort subfiles recursively and relink results }
  sf2 := sr↑.next; {start of L2 saved}
  if lt <> sl then LINKSORT3(sl↑.next, lt); {sort left sublist L1}
  lt↑.next := last;
  sf1 := sl↑.next;
  {start of L1 saved; if L1 empty, then sf1 points to pivot node}
  if rt <> sr
  then
  begin {sublist L2 not empty}
    LINKSORT3(sf2, rt);
      last↑.next := sf2 {connect pivot node to L2}
      rt↑.next := nil
      last := rt
  end {of sorting L2}
  first := sf1 {return start of sorted list}
99:
end {LINKSORT3};
```

Unfortunately, LINKSORT3 is not very pretty, with two `goto`-Statements and unraveled `repeat`- and `while`-loops. The excuse is that we present a somewhat optimized version which avoids some pointer, resp. index, comparisons. Note that we choose the last key as pivot and therefore cannot overshoot the list with a key value $> H$.

4.3 Analysis of LINKSORT

We now enter the analysis of LINKSORT3 for multiset input. LINKSORT3 starts the partition phase by testing on a unary file. If the subfile $F = H, H, \ldots, H$, we are done, otherwise we do an ordinary two-way split, eliminating one key (H) from the recursion. The transition from testing on $=, \neq$ to $\leq, >$ requires one additional key comparison.

For the behavior on a multiset $\{x_1 \cdot 1,\ x_2 \cdot 2,\ \ldots\ ,\ x_n \cdot n\}$ with n distinct values, we can rely on two fundamental papers by Sedgewick [Sed77a] and Burge [Bur76].

Theorem 4.1 (Sedgewick). *If a Quicksort program sorts a random binary file with less than $2N \cdot H_N$ comparisons, on the average, then it needs no more than*

$$2N(1-1/n)H_{n-1} - 4N + \ldots$$

comparisons for a random n-ary file with N elements.

LINKSORT3 is a *Quicksort* program according to the definition in 3.2.4 and its behavior on binary files can be easily computed.

Lemma 4.2. *To sort a random, binary file with N elements,* LINKSORT3 *requires, on the average, exactly*

$$3N - \frac{2^N - 1}{2^{N-1} - 1} - \frac{N(N-3)}{2^N - 2}$$

comparisons.

Proof. We need to solve

$$\begin{aligned}\bar{c}(x_1,\ x_2) = \ & x_1 + x_2 + 1 + \frac{x_1}{x_1 + x_2}\,[\bar{c}(x_1 - 1) + \bar{c}(x_2)] + \\ & \frac{x_2}{x_1 + x_2}\,\bar{c}(x_1,\ x_2 - 1)\end{aligned} \tag{4.1}$$

and form the average over all multiplicities of x_1 and x_2.

$$\bar{c}_N = \frac{1}{2^N - 2} \sum_{x=1}^{N-1} \binom{N}{x}\ \bar{c}(x, N-x)$$

Theorem 4.3 (Sedgewick). *A Quicksort program, which partitions N elements with $N+1$ comparisons, requires – on the average – no more than*

$$\begin{aligned}2 \sum_{4 \le k+3 \le j \le n} & \frac{x_k x_j}{1 + x_{k+1} + \ \ldots\ + x_{j-1}} + \\ & \sum_{k=1}^{n-2} \bar{c}(x_k,\ x_{k+1},\ x_{k+2}) - \sum_{k=1}^{n-3} \bar{c}(x_{k+1},\ x_{k+2})\end{aligned} \tag{4.2}$$

comparisons to sort the multiset $\{x_1,\ x_2,\ \ldots\ ,\ x_n\}$.

This upper bound for *Quicksort* programs appears with minor changes in the analysis for LINKSORT3. Note that in the recurrence relation below, we have $N+1$ comparisons for the partition phase around pivot key $H = j$ and that we may subtract 1 from x_j from the recursion, as we eliminate the pivot key $(1 \le j \le n)$.

$$\bar{c}(x_1,\ \ldots\ ,\ x_n) = N + 1 + \frac{1}{N}\sum_{j=1}^{n} x_j\left(\bar{c}(x_1,\ \ldots\ ,\ x_j - 1) + \bar{c}(x_{j+1},\ \ldots\ ,\ x_n)\right) \quad \text{for } n > 1, \bar{c}(x) = x, \tag{4.3}$$

which evaluates to

$$\begin{aligned} &= 2\sum_{3\le k+2\le j\le n} \frac{x_k x_j}{1 + x_k + \ldots + x_{j-1}} + \\ &\sum_{k=1}^{n-1}\Big(2\left(x_k + x_{k+1}\right) \ + \ \frac{x_{k+1}\left(x_{k+1} - 1\right)}{x_k + 2} + \\ &\qquad \frac{x_k\ x_{k+1}}{x_k + \ 1} - \frac{x_k}{\binom{x_k\ +\ x_{k+1}}{x_k}}\ \Big) + \\ &\sum_{k=1}^{n-2} \frac{x_{k+1}^2}{x_k + x_{k+1}} \left(\frac{1}{\binom{x_k\ +\ x_{k+1}\ +\ x_{k+2}}{x_{k+2}}} - 1 \right) - \\ &\sum_{k=1}^{n-2} x_{k+1} \end{aligned} \tag{4.4}$$

This is a very unyielding formula. To get a feeling for the relative merits of a two-way versus a three-way split, it is useful to argue with fixed multiplicities, i.e. a *repetition factor* M, s.t. $N = M \cdot n$. As mentioned above in Section 4.1, LINKSORT3 requires less than $2NH_{n-1} + N$ PASCAL-comparisons, TRISORT $3(N + M)H_n - 4N$ and the lower bound with ternary `MIX`-comparisons becomes $2(N + M)H_n - 3N - n$. Plotting these functions for $\bar{c}_M$ for a file of size $N = 1000$, we get Figure 4.2. Note that the y-axis has a logarithmic scale. M is set to 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000 and therefore runs from all-distinct keys to unary, where the latter requires N comparisons for LINKSORT3 and lower bound, $2N$ comparisons for TRISORT.

As it turns out, the upper bound $2NH_{n-1} + N$ for LINKSORT3 is not very sharp for small values of M. Therefore we added experimental measurements of key comparisons from 20 permutations. This curve, in turn, seems unusually uneven for unknown reasons, possibly caused by a biased permutation generator. A repetition of these experiments by independent researchers would be welcome.

The important insight is, that a partition strategy which only catches unary subfiles at the end is not much slower in the case of high repetition factors but considerably faster when no or only few duplicates are present. In particular

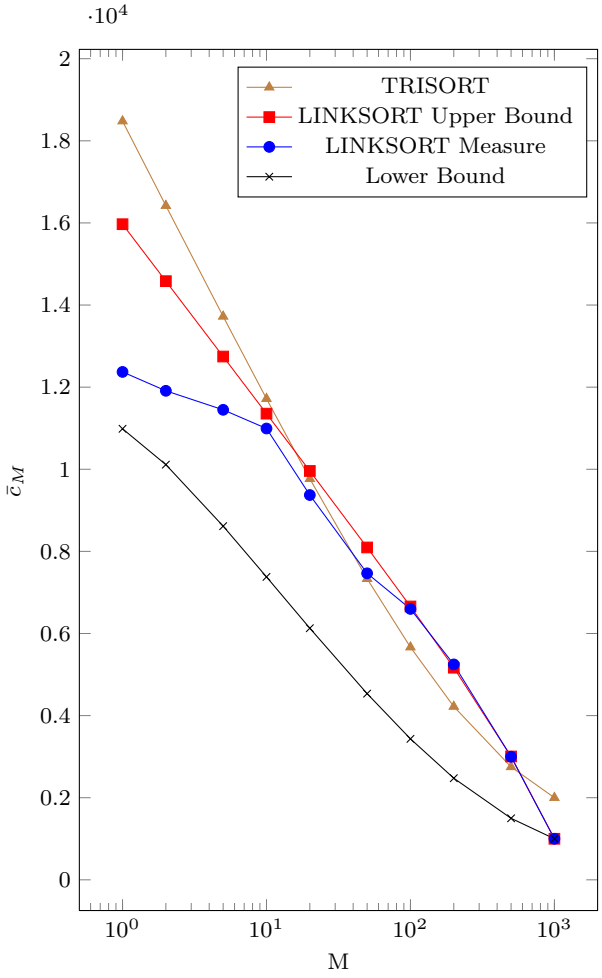


Fig. 4.2 Average Key Comparisons

the case $M = 2$ occurs frequently in practice[1], when two almost identical lists are thrown together and then sorted (instead of first sorting each list and then merging them) to find missing duplicates.

We illustrate the point again with another plot of execution times (cf. Figure 4.3) measured at the time of the TURKU LECTURE with the same setting ($N = 1000$, 20 permutations, varying repetition factor M) as above. We added the observed execution times for Sedgewick's highly refined Median-of-three *Quicksort* with elimination of short subfiles to bring the figures into

[1] Say a list of online-reservations which each need a written confirmation which are recorded in a second list.

perspective. A table presentation of all measurements, including other array-variants follows in Section 4.2.

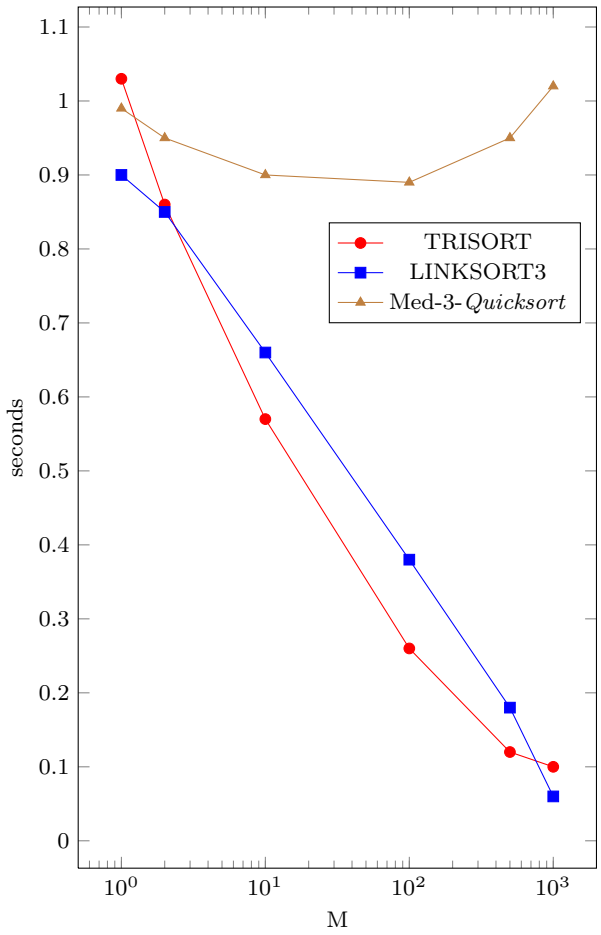


Fig. 4.3 Average Execution Times

The second plot illustrates again the argument: late detection of duplicates (LINKSORT3) is slightly slower with high values of M, but does well in the important cases $M = 1$ and $M = 2$. Median-of-three *Quicksort* cannot profit from multisets, but it does not degenerate either as a *Quicksort* with single key pivot selection would.

4.4 Five Ways to Achieve Smoothness

We have seen so far that there are ways to speed up *Quicksort* in the presence of duplicates without penalty for the case of files with all distinct keys. We applied the ideas of early and late duplicate detection to linked lists with the additional benefit of stability (records with equal keys maintain their relative order).

Is there any way of carrying these ideas with two-way or three-way split back over to input data stored as an array? Yes there is, Wegner in [Weg85] actually gives five solutions, all in situ, none of them stable, however.

4.4.1 SLIDESORT

The idea is to scan the file from both ends doing a three-way split (early duplicate detection). Keys which are recognized as equal to the pivot form a middle subfile which needs to "slide" to the right. An efficient way to do this is by means of a "wheel", a data structure we mentioned already in connection with Lumoto's *Quicksort* on page 18. As we explained there, interchangeably adding records to the wheel and rolling it destroys stability, but the exchange of the other keys during the partition phase is no better, as can be seen from Figure 4.4.

Essentially, the new key inspected in position i determines what happens. If it is less than the pivot H, the "wheel" of keys equal to H is rolled. If $A[i] = H$, the wheel is extended (not shown in Figure 4.4). If $A[i] > H$, we scan from the right until we find in position $A[j]$ a key $\leq H$ which we can swap for $A[i]$. If in this case $A[j]$ is strictly less than H, we do a triangular swap as shown in the middle of Figure 4.4, otherwise an ordinary swap of $A[i]$ and $A[j]$.

4.4.2 HEAD&TAIL-SORT

Once this "wheel technique" is understood, other strategies are easy to devise. One strategy is again a three-way split with early detection of keys equal to the pivot, but with two pivot subfiles on the left and right end. Accordingly we name this *Quicksort* variant HEAD&TAIL-SORT. The pivot subfiles are swapped into the middle when partitioning is complete. Details can be taken from Figure 4.5.

Again stability cannot be achieved. Apart from this defect, which afflicts all these smooth variants, the extra exchange of the pivot subfiles at the end of the partition phase slows this variant in the case of many duplicates. Running times for the variants are provided further below on page 55.

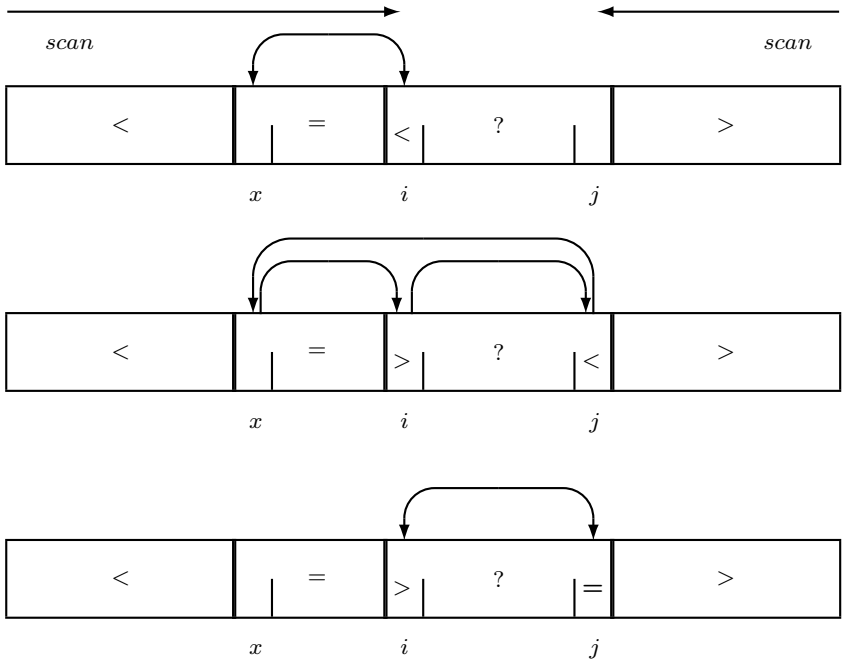


Fig. 4.4 SLIDESORT – a three-way split with sliding pivot subfile

4.4.3 UNISORT

UNISORT is a one-way scan, three-way split *Quicksort* variant (early detection of duplicates). Only one subfile for pivot elements is formed, here on the left. UNISORT is thus similar to HEAD&TAIL-SORT in that it forms a subfile of records with key equal to pivot H, which needs to be swapped into the middle afterwards. Like Lumoto's Sort with a one-way scan it has a higher number of swaps. As can be seen in Figure 4.6, a record r_i with key-value $> H$ is swapped with r_j regardless of what the key is in r_j. Of course that could be changed to include a scan from the right. All in all, this variant is not very competitive either but its code is pleasantly short.

4.4.4 DOUBLESORT

Early detection of duplicates in a three-way split implemented with (binary) PASCAL-comparisons requires a second key comparison for keys $\leq H$, resp.

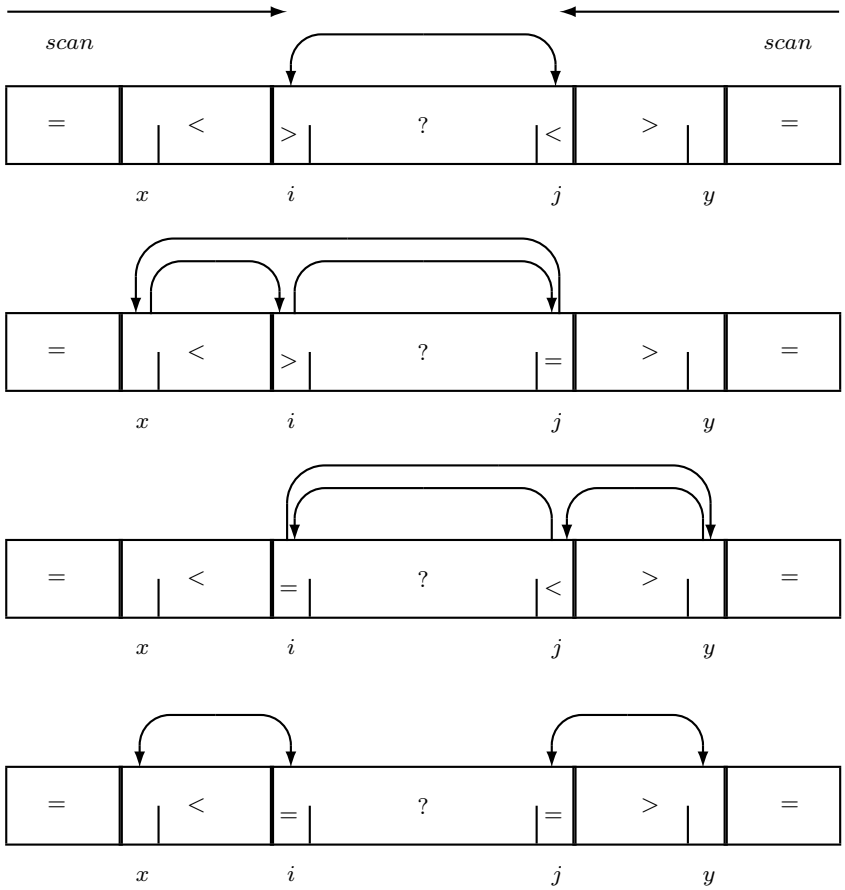


Fig. 4.5 HEAD&TAIL-SORT – a three-way split with two pivot subfiles

$\geq H$. Up to now, this test was done right-away, but we might as well postpone it for a second scan which gives rise to DOUBLESORT.
The advantage of DOUBLESORT is that we don't need the extra movement of records into the middle as this is part of the second scan. On the other hand, there are extra index comparisons to control the loops. All versions, including the two still to come, have tricky details concerning empty subfiles (if H happens to be the largest or smallest key) or when called with an empty or singleton subfile.

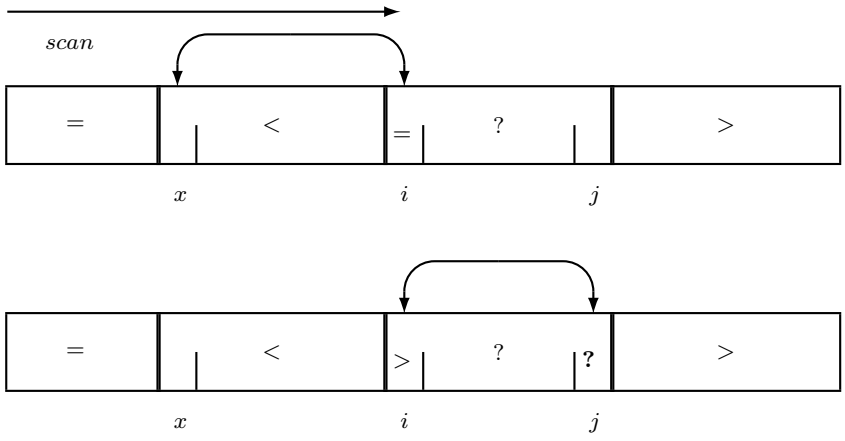


Fig. 4.6 UNISORT – a three-way split with one pivot subfile on the left

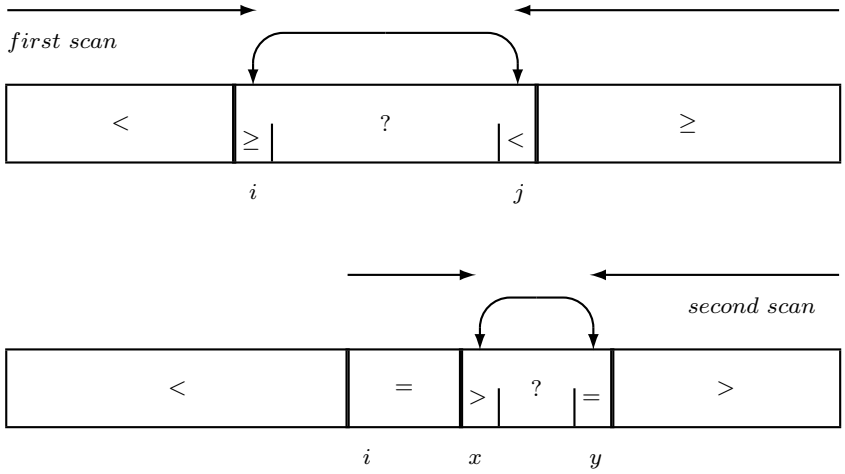


Fig. 4.7 DOUBLESORT – a three-way split with two scans

4.4.5 General Approach for a Three-way Analysis

Before we look at the array-variant of Linksort, which is a two-way split with late detection of duplicates, let us indicate how to analyze these three-way variants using SLIDESORT as an example. We begin with counting the *extra key comparisons* which SLIDESORT needs on top of the N comparisons per

partition phase. Assume the initial test on each key for the left and right scans is as depicted in Figure 4.8.

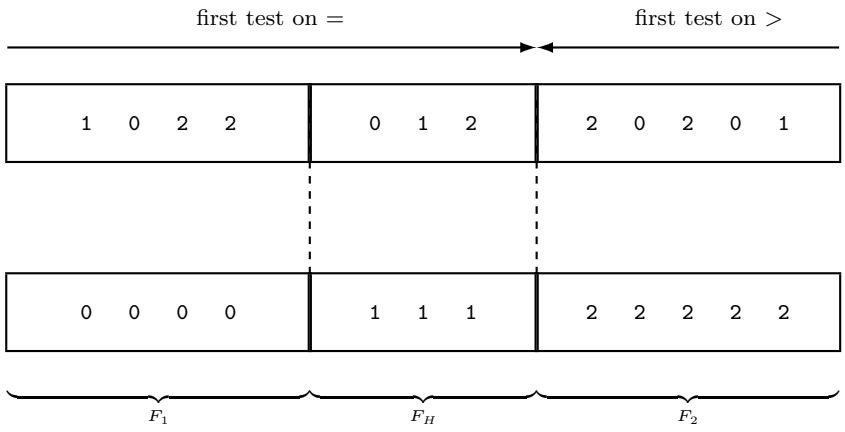


Fig. 4.8 SLIDESORT example with $H = 1$

The first – fairly trivial – observation is that the number of keys $> H$ in F_1 and in F_H equals the number of keys $\leq H$ in F_2. Thus the extra key comparisons in the split phase are

$$\begin{aligned} &= \#\text{keys} \neq H \text{ in left scan } (F_1 \text{ and } F_H) + \\ &\quad \#\text{keys} \leq H \text{ in right scan } (F_2) \\ &= 2 \cdot \#\text{keys} > H \text{ in left scan } (F_1 \text{ and } F_H) + \\ &\quad \#\text{ keys} < H \text{ in left scan } (F_1 \text{ and } F_H). \end{aligned}$$

Figure 4.9 shows the situation again with multinomials added.

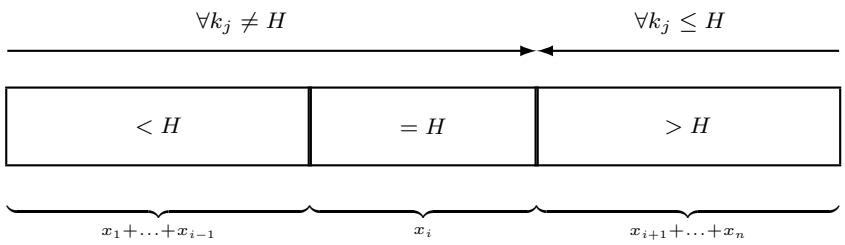


Fig. 4.9 SLIDESORT extra comparisons

The total number of key comparisons is then

$$N + \frac{1}{N} \sum_{1 \le i \le n} x_i(x_1 + \ldots + x_{i-1}) + \frac{1}{N} \sum_{1 \le i \le n} \sum_{0 \le t \le x_{i+1} + \ldots + x_n} x_i \cdot t \cdot p(t)$$

with

$$p(t) = \frac{\binom{x_1 + \ldots + x_i}{t} \binom{x_{i+1} + \ldots + x_n}{x_{i+1} + \ldots + x_n - t}}{\binom{N}{x_{i+1} + \ldots + x_n}}$$

where t is the number of keys $\le H$ which is detected in the right scan. $p(t)$ is then formed from the number of combinations without repetition.

A reformulation applying Vandermonde's convolution twice and with the observations from above gives the average number of key comparisons in the split phase as

$$N \; + \; \frac{1}{N} \sum_{1 \le i \le n} x_i \Big(\sum_{0 \le s \le x_1 + \ldots + x_{i-1}} s \cdot p_i(s) + \sum_{0 \le t \le x_{i+1} + \ldots + x_n} 2t \cdot p_i(t) \Big)$$

with $x_{i+1} + \ldots + x_n = |F_2|$,

$$p_i(s) = \binom{x_1 + \ldots + x_i}{s} \; \binom{x_{i+1} + \ldots + x_n}{x_1 + \ldots + x_i - s} \Big/ \binom{N}{x_1 + \ldots + x_i}$$

and $p_i(t)$ accordingly.

Table 4.1 below shows comparisons and exchanges for each of the variants for the special case $N = n$ (all distinct keys).

4.4.6 UNARYSORT

What is missing so far is a variant which follows the LINKSORT3 line, i.e. does a two-way split with late detection of duplicates. Remember, this required one extra key comparison in the split phase. The array-variant is called UNARYSORT as it "hopes" for a unary subfile. The sort assumes a stopper key in $A[N+1]$ which is greater than $A[1].key$.

Listing 4.2 UNARYSORT – detect unary runs.

```
procedure UNARYSORT(l, r : integer);
{sort recursively an array a[l..r] eliminating unary subfiles
```

```
{but examining each key exactly once except for one key; assumes
a[N+1] > a[1] for the original file in a[1..N]}
var i, j: integer;
  H, aux: keytype { elements to be sorted consist of keys only };
begin {UNARYSORT};
  H := a[l]; i := l + 1;
  {a unary subfile?}
  while a[i] = H do i := i + 1;
  {a[i] ≠ H}
  if i ≤ r {not unary}
  then
  begin {partition}
    i := i − 1; j := r + 1;
    while i < j do
      repeat i := i + 1 until a[i] > H;
      repeat j := j − 1 until a[j] ≤ H;
      swap (a[i], a[j])
    end {while}
    {switch back a[i], a[l], a[j]}
    circ (a[i], a[l], a[j]); {a circular swap,
      i.e. circ(a,b,c): temp := a; a := b; b:= c; c := temp}
    if l ≤ j − 1 then UNARYSORT(l, j − 1);
    if i < r then UNARYSORT(i, r)
  end {then}
end {UNARYSORT};
```

The program for UNARYSORT is very similar to a classical *Quicksort* which eliminates one single pivot record from recursion. Indeed, we just added a little prelude which tests for unary subfiles. As we saw, this is just as effective as early duplicate detection at a much smaller price for extra comparisons. The following Table 4.1 shows this. The rightmost column gives execution times for this particular case $N = n = 1000$. These measured execution times will reappear in Table 4.2 at the end of this chapter.

4.5 Quicksort for presorted files

So far *Smoothness* referred to the behavior (speed-up) of *Quicksort* on multisets. Often, a similar behavior on presorted files is assumed as well. Indeed, none of the algorithms above will profit from presortedness. Even worse, without extra precautions, like median-of-three pivot selection, the algorithms degenerate to execution times which are quadratic with respect to the input length.

Presortedness had been mentioned on page 32. The algorithm by C. R. Cook and Do Jiu Kim for nearly sorted lists [CK80] as well as what Edsger

Table 4.1 Effort in partition phase

special case $N = n$	MIX comparisons	Pascal comparisons	exchanges	time
TRISORT	N	$9N/6$	–	1.03
HEAD & TAIL	N	$\leq 8N/6$	$N/6 + \ldots$	1.38
UNI	N	$9N/6$	$N/2 + \ldots$	2.25
SLIDE	N	$10N/6$	$N/2$	1.96
DOUBLE	$3N/2$	$9N/6$	$N/6$	1.44
LINKSORT	N	$N + 1$	–	0.90
QUICKSORT	N	N	$N/6$	1.08
UNARYSORT	N	$N + 1$	$N/6$	1.23

W. Dijkstra published as *Smoothsort* [Dij82] both take advantage of biased input. Presortedness can be defined formally, as is done by Heikki Mannila [Man85], e.g. by counting the number R of records which must be removed, s.t. the remaining records form an ascending file, and setting this number in relation to the input length N. Often, the term "run" (or upward run) appears in connection with presortedness, meaning sequences of ascending keys within a permutation of keys. The frequency of runs and their length can also be measures of presortedness. Knuth defines this more precisely in [Knu98, p. 35ff.].

Here we suggest a *Quicksort*-variant, called RUNSORT, which – similar to LINKSORT3 and UNARYSORT – "hopes" for a run and, if detected, stops recursion for this already sorted subfile. If the scan fails to find a complete run as subfile, the remainder is split as usual. Figure 4.10 illustrates the general strategy.

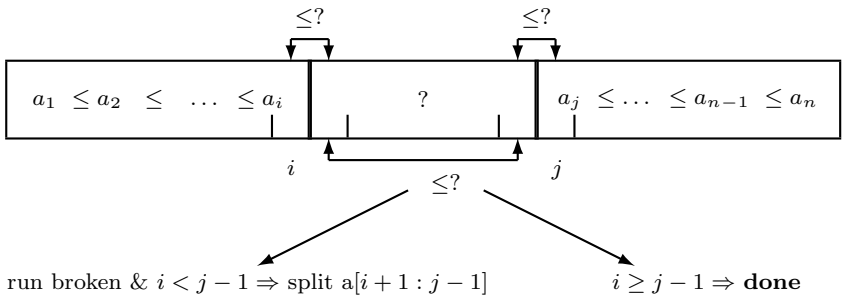


Fig. 4.10 RUNSORT – searching for runs

Listing 4.3 RUNSORT – detecting presortedness.

```
procedure RUNSORT (l, r : integer);
  var i, j: integer; H : keytype;
  procedure swap(var a, b: keytype);
      ...
  begin {RUNSORT};
    if a[l] > a[r] then swap (a[l], a[r]);
    i := l; j := r;
    while (i + 1 < j) and (a[i] ≤ a[i + 1]) and
        (a[j − 1] ≤ a[j]) and (a[i + 1] ≤ a[j − 1]) do
    begin
      i := i + 1; j := j − 1
    end;
    if i < j − 1 {not a run}
    then
    begin {partition}
      H := (a[i] + a[j]) div 2;
      repeat
        repeat i := i + 1 until a[i] ≥ H;
        repeat j := j − 1 until a[j] ≤ H;
        if i < j then swap (a[i], a[j])
      until i ≥ j;
      if l < i − 1 then RUNSORT(l, i − 1);
      if j + 1 < r then RUNSORT(j + 1, r);
    end {i < j − 1}
  end {RUNSORT};
```

One tricky aspect of RUNSORT is the pivot selection when the search for a complete run fails. As Listing 4.3 shows, we go for a fictitious pivot element computed from $(a[i] + a[j])$ div 2. This is safe, as we know that $a[i]$ is the maximum for the left subfile so far, $a[j]$ is the minimum for the right subfile, and $a[i] \leq a[j]$.

All in all, RUNSORT is competitive with e.g. the sort of Cook and Kim [CK80] and beats Dijkstra's *Smoothsort* hands down. On the other side, a partition-exchange sort cannot really preserve runs unless they form runs-of-runs. In Table 4.2 we included input consisting of two adjacent runs of 900 and 100 keys for input length $N = 1000$. The execution time shows that RUNSORT cannot profit from this arrangement, but does not degenerate either.

Table 4.2 Execution times for various sorts and inputs

$N = 1000$ keys		multisets					presorted files					2 runs
algorithm	random	M = 2	M = 10	M = 100	M = 500	M = 1000	p = 0%	p = 5%	p = 10%	p = 20%	p = 50%	900/100
Quicksort (textbook)	1.08	1.01	1.00	1.03	1.08	1.05	–	–	–	–	–	–
Quicksort (Sedgewick)	0.99	0.95	0.90	0.89	0.95	1.02	0.50	0.51	0.54	0.62	0.82	0.91
TRISORT	1.03	0.86	0.57	0.26	0.12	0.10	–	–	–	–	–	–
LINKSORT	0.90	0.85	0.66	0.38	0.18	0.06	–	–	–	–	–	–
HEAD&TAIL	1.38	1.21	0.87	0.51	0.31	0.26	–	–	–	–	–	–
UNISORT	2.25	2.05	1.50	0.69	0.38	0.25	–	–	–	–	–	–
SLIDESORT	1.96	1.68	1.39	0.45	0.15	0.05	–	–	–	–	–	–
DOUBLESORT	1.44	1.28	0.85	0.40	0.17	0.10	–	–	–	–	–	–
UNARYSORT	1.23	1.14	0.82	0.42	0.18	0.05	–	–	–	–	–	–
Heapsort (textbook)	2.41	2.40	2.40	2.25	1.43	0.64	2.52	2.50	2.50	2.48	2.45	2.52
Smoothsort (Dijkstra)	6.75	6.75	6.64	5.62	2.80	0.91	0.96	1.42	1.80	2.62	4.65	5.43
Cook-and-Kim-Sort	1.41	1.39	1.31	1.32	1.16	0.64	0.26	0.44	0.58	0.79	1.21	0.54
RUNSORT	1.31	1.25	1.09	0.74	0.42	0.12	0.11	0.54	0.66	0.79	0.94	1.31

4.6 Conclusion

The intention of this chapter was to show that Sedgewick [Sed77a, p. 244] was too pessimistic in assuming that no known efficient method can come close to the lower bound for multiset sorting with *Quicksort*. Both a three-way split with early removal of duplicates can be implemented, as well as a two-way split with late detection of unary subfiles. Both linked list versions (TRISORT, LINKSORT) are particularly fast.

Among the array versions, UNARYSORT would be a good candidate for further refinement with median-of-three pivot selection and *Insertionsort* for short subfiles. The extra key comparison per split phase is a good investment. It helps in the presence of a large number of duplicates just as much as a three-partitioning strategy does. On the other hand, when the input is a proper set, at most $2N$ extra key comparisons are incurred (cf. Formula 12.3 in the Appendix), which can be reduced even more, when short subfiles are handled differently.

A reliable pivot selection method, e.g. median-of-three (not possible for linked list data) would also eliminate the innocent looking little dashes in Table 4.2. Each of them stands for the embarrassing fact, that the sort degenerates heavily in the case of (pre-)sorted input which is not unlikely to occur in practice. The good news, however, is that we are still twice as fast as the only other competitor for in situ $O(N \log N)$ average and worst case sorting – *Heapsort*.

Chapter 5
An External Quicksort

5.1 The Locality-of-Reference Myth

A natural question is to ask whether *Quicksort* is also a good candidate for external sorting? The following answers speak against *Quicksort*.

> What sorting algorithm should be used? The method of merging is a fairly natural choice; other methods of internal sorting do not lend themselves so well to a disk implementation, ... [Knu98, p. 264]

> The severity of interpass paging depends on the size of real storage, because the locality shifts dramatically after a pass. [Lor75, p. 342]

Brawn, Gustavson and Mankin [BGM70] report that FIFO-*Quicksort* is worse than FIFO-merging. Other noteworthy comments on aspects of external sorting – and virtual memory in particular – can be found in [TAH84], [CK80], [Den68], [Den80], [Pet74], [Sed78], [Sha85], [Ver86], [Wai85], and [Wal77].

5.1.1 EXQUISIT - An External Quicksort

Against all odds, Hans-Werner Six and Lutz Wegner decided to have a closer look at *Quicksort*'s behavior in external sorting [SW84] resulting in an external sort called EXQUISIT. Assuming the sort reads and writes data blockwise, we might as well look at a file F of 4 blocks which are read into a (FIFO-)buffer S which can simultaneously hold two blocks. Assume also a "traditional" two-way *Quicksort*, i.e. there are scans from the left and right end to partition the file. Figure 5.1 shows a possible scenario.

The following items are worth noting before we attempt to find out how good (or bad) the idea really is.

- Shackelton [Sha85] reports that an external *Quicksort* had been in use in England in the Sixties on an ATLAS-Computer, but was never analyzed.

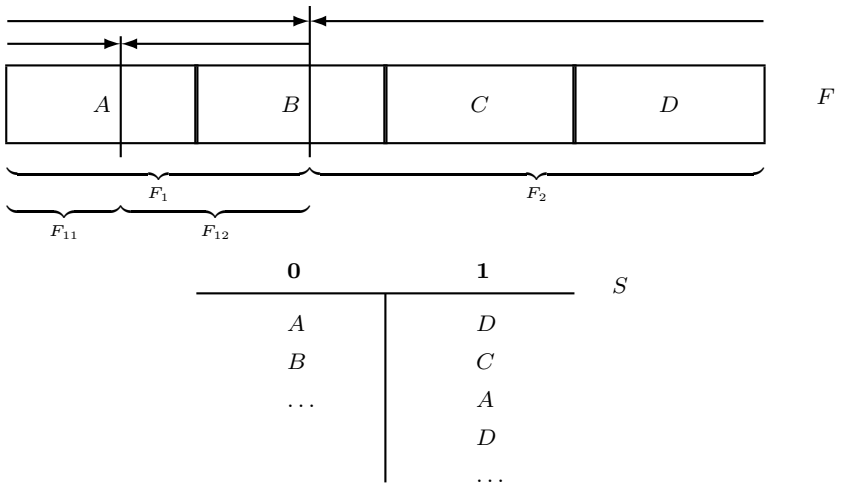


Fig. 5.1 Reading and Writing of blocks in an external *Quicksort*

- A fictitious pivot element $H = (first + lastkey)/2$ or similarly $(min + max)/2$ would be advantageous.
- Blocks are fetched on-demand with delayed rewriting.
- We would sort the left subfile first and use indirect "buffer addressing".

Quicksort variants with a fictitious pivot element are easily designed (cf. Listing 5.1).

Listing 5.1 QUICKS – *Quicksort* with a fictitious pivot element.

```
procedure QUICKS(l, r : integer);
var lt, rt, H, aux:integer;
begin lt := l − 1; rt := r + 1;
  H := (s[l] + s[r]) div 2; {pivot selection}
  while lt < rt do
  begin
    repeat lt := lt + 1 until s[lt] ≥ H;
    repeat rt := rt − 1 until s[rt] ≤ H;
    if lt < rt
    then
    begin {pointers have not crossed}
      aux := s[lt]; s[lt] := s[rt]; s[rt] := aux;
    end
  end;
  if l < lt − 1 then QUICKS (l, lt − 1);
  if rt + 1 < r then QUICKS (rt + 1, r)
end {of QUICKS};
```

5.2 Analysis of EXQUISIT

EXQUISIT is trivially an $O(N \log N)$ key comparisons algorithm with the same constants as any other two-way *Quicksort*. However, the important measure for us is the number of block accesses!

Definition 5.1. Let $L(F)$ denote the position of the first record of F in the first block (page) of file F, $L(F) \in \{1, \ldots, s\}$, s the page size.

To come up with any sensible analysis we need to make a few assumptions concerning randomness.

1) File F is a random permutation of records and keys are pairwise disjoint; without restricting generality, let the keys of F be the set of natural integers, i.e. $keys(F) = \{1, 2, \ldots, N\}$.
2) $L(F)$ has equal distribution over $1, \ldots, s$, $p(L(F) = k) = \frac{1}{s}$ (F starts at a random position in the first block).
3) All $N-1$ pivot splits are equally likely, i.e. if F is split into F_1 and F_2, then $\exists k$, $k \in \{1, \ldots, N-1\}$ with $|F_1| = k$, $|F_2| = N - k$, $p(|F_1| = k) = \frac{1}{(N-1)}$ (fictitious pivot element).

Lemma 5.2. *Let F have size $k \geq 5$ blocks. If F has already been split into F_1 and F_2, each with size > 2 blocks, then for any split position $t \in 1, \ldots, N-1$ we need $k-1$ block reads to split F_1 and F_2.*

Proof. The argument can be derived from the following Figure 5.2. With the

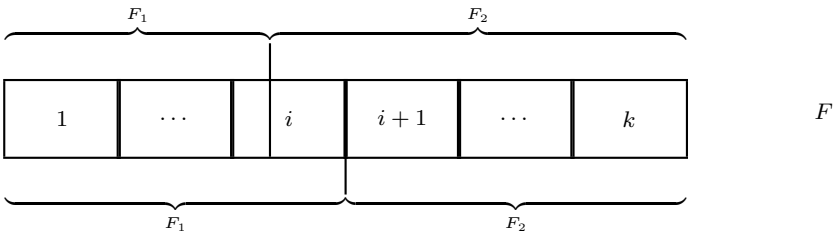


Fig. 5.2 Possible split positions in an external *Quicksort*

exception of the very first scan, the leftmost block of a file is already in core, unless the split happens to fall exactly on the border. Thus, for F_1 we have $i-1$ reads and for F_2 we have $k-i$ reads, in total then $k-1$.

□

Note that $k-1$ is an upper bound for $|F_i| < 3$ blocks.

Definition 5.3. Let $K(L(F))$ denote the number of blocks which F occupies with starting position $L(F)$.

The following results are given without explicit proof because they are mostly combinatorial facts.

Lemma 5.4. *The upper bound for the number of block reads – when F is initially already split – is*

$$g(N) = \frac{1}{s}\sum_{L(F)=1}^{s}(K(L(F))-1) + \frac{1}{N-1}\sum_{t=1}^{N-1}(g(t)-g(N-t)) \qquad \text{for } N > s+1$$

$$g(N) = 0 \qquad \text{for } N \leq s+1.$$

Lemma 5.5. *The average number of blocks occupied by a file of size N and block size s is*

$$\frac{1}{s}\sum_{L(F)=1}^{s}K(L(F)) = \frac{N-1}{s}+1.$$

Theorem 5.6. *The expected number of block reads $EBR(N)$ to sort a file of size N is*

$$\leq \frac{2N}{s}\ln\left(\frac{N}{s+1}\right)+\frac{3N}{s}+1-\frac{N}{s(s+1)}$$

Conclusion: For large N, we have $M = \frac{N}{s}$ as the number of blocks of F. Let EBR' be the number of block reads as a function of the number of blocks M, then $EBR'(M) = 2M\ \ln\ M + 3M$.

This coincides with our actual measurements which yield $\approx 1.5M\ \ln\ M$. However, what about the worst case? Does an average behavior of $O(M \log M)$ block reads imply an $O(M^2)$ degeneration? Unfortunately, not. In the worst case, each partitioning removes only one of the N records and requires loading all K/s blocks for a subfile of size $K(1 \leq K \leq N)$. Asymptotically this gives $O(N \cdot M)$.

Example 5.7. Set the file size to $|F| = N = 50000$ records with $s = 50$ records per block. Each record has a 9-digit key. Then $M = 1000, 2M \ln M + 3M \approx 16815$. Given a disk with an 18 ms block read/write, we sort the file in 300 s = 5 min on the average, assuming I/O dominates the sort.

In the worst case, we have $NM = 50000 \cdot 1000 = 50000000$ (50 millions) block reads and writes which totals up to a sort time of about 250 hours or more than 10 days! To prevent this, a $(min + max)/2$-strategy for pivot selection would be advisable, resulting in a worst case of $O(M \log max)$. With 9-digit keys, we would end up with ≤ 30000 block fetches in ≤ 9 minutes.

5.3 Head Travel and Experimental Findings

In Section 1.3.2 we discussed assumptions regarding the placement of the file on a disc. Just about the only safe statement for an access cost model is to say that neighboring blocks are most likely faster to fetch than those far apart, as that covers blocks within one cylinder and – to a lesser degree – neighboring cylinders.

To get a handle on sorting with non-uniform block access times, we consider a linear block model and measure the distance (in number of blocks) between successive reads and writes[1]. Now consider the case when EXQUISIT splits a file F of M blocks (cf. Figure 5.3).

Fig. 5.3 Head travel in successive block fetches from both ends of the file

The precise amount of head travel to partition M blocks in a two-way scan depends on several assumptions, some of them rather arbitrary. For one, assume that the M blocks of the file are fetched intermittently from the left and right end of F, i.e. after each read/write, the head reverses and travels to the other end (see note below).

More precisely, consider the start of a partition phase for subfile F. The head has read block 1 and is thought to stay at the end of block 1. Then it must travel over $M - 1$ blocks to reach the start of block $M + 1$ (past subfile F) to write back the last block that was in the buffer before splitting F, presumably block $M + 1$. After writing that block, the head stands at the end of $M + 1$. To read block M, the head moves back two blocks to the start of M. In total this adds up to a distance of $M + 1$ blocks. After having read block M, the head of the disk returns to the start of block 1 which is re-written, followed by reading block 2 (zero distance). This return then has a cost of M. When the pointers meet at the end in one block, we save two fetches. With this logic we arrive at a total cost of $(M + 1)(M + 2)/2 - 2$ blocks.

[1] An obvious deficit of this model is that is assumes sequential forward reading/writing in a file is as expensive as reading/writing blocks in reverse order, which is not true on a revolving disk.

The total head travel $f(M)$ for the sort is now calculated with the common recurrence relation under the assumption that each block is equally likely as split position. Let $g(M)$ be the effort for splitting subfile F with $|F| = M$, then EXQUISIT requires, on the average, a head travel of

$$f(M) = g(M) + \frac{1}{M} \sum_{t=1}^{M-1} (f(t) + f(M-t))$$

blocks. Inserting $g(M) = (M+1)(M+2)/2 - 2$, this gives

$$\begin{aligned} f(M) &= \frac{3M^2}{2} + 6(M-1)H_{M-2} + \frac{M}{2} - 3 \\ &= O(M^2 + M \ln M). \end{aligned}$$

Inserting other estimates for $g(M)$ gives very similar results, but the leading term $\frac{3M^2}{2}$ remains the same! Thus, within the linear block model, an external *Quicksort* with a simple two-way scan generates $O(M^2)$ head travel. Whether this justifies the claim that *Quicksort* has a poor locality of reference can only be decided by looking at other methods. The obvious contender is *Mergesort*.

One questionable assumption is the complete head reversal after each read/write of blocks. For one thing, if the scan up, resp. scan down, in the left, resp. right, block start at the same time, then each block has a 50:50-chance to require replacement first. Therefore, in half the cases, the head still is at the end where the next read/write is demanded.

Secondly, it is fairly easy to devise a strategy for pre-fetching. All we need is two extra buffers (one extra for each side) for a total of, say, four blocks[2]. Now the sort simply remembers the index of the last read/write block. Say the head was last on the left side and we exhaust block b_i from the left side (the scan reaches the end of the buffer for this block b_i). Then, unless the extra buffer has a fresh block, we fetch block b_{i+1} and pre-fetch b_{i+2}.

The other case is when we exhaust a block b_j on the right end and block b_{j-1} is not yet in the buffer, all this while we are still on the left side. The head must now travel to the right, rewrite b_j and b_{j+1}, then read b_{j-1} and, while being at it, pre-fetches b_{j-2}. The same story repeats itself for the situation where the last read/write was on the right side. Of course, the number of read/writes cannot be reduced this way, but head travel would be halved.

Overall the head travel considerations stand on shaky ground. If the disk is shared in a multi-programming environment, there will be intervening fetches. Furthermore disks employ batched read/writes, which are arranged in a way in which an elevator works: requests are re-ordered s.t. the arm with all heads travels (sweeps) from the outermost cylinder (the basement) to the

[2] This is in contradiction to what Verkamo observed in [Ver87]; see also the discussion in Section 5.5.

innermost cylinder (top floor) and then back servicing cylinders in-between on both passages.

5.4 Mergesort as Contender

In [SW84] a p-way *Mergesort* was implemented to run against EXQUISIT. The merge order p is chosen as follows: Let x be the file size and k be the core size (in records) and assume that the initial phase of the merge sort produces $r \leq \lceil \frac{x}{k} \rceil$ presorted sequences. Using a p-way merge, the sort is completed in $\lceil \log_p(r) \rceil$ passes where each pass requires $\lceil \frac{x}{\lfloor \frac{k}{p+1} \rfloor} \rceil$ external reads (and writes), because the core is partitioned into p input buffers and one output buffer which are of equal size. Hence the optimal merge order p_{opt} is computed as the value which minimizes the number of external reads given by

$$\lceil \log_p(r) \rceil \cdot \lceil \frac{x}{\lfloor \frac{k}{p+1} \rfloor} \rceil.$$

For the simulation and measurements, the internal sort area is declared as an `integer array S[1:2s]`[3]. In the tests, s is always chosen in such a way that $2s$ is a multiple of $p_{opt}+1$, i.e. $2s = (p_{opt}+1) \cdot s'$ for some s'. Hence the file to be sorted consists of M' blocks of size s' s.t. $M' \cdot s' = M \cdot s$, where M and s are the corresponding parameters in EXQUISIT. The file and the external working space are represented by an `integer array E'[1:2M', 1:s']` and data are transferred in terms of logical blocks of size s'.

The *Mergesort* starts as usual with forming strings of length $2s$ using *Quicksort*. If the keys of a string form an ascending sequence with the previous string, then the two strings are concatenated to an over-length string. In our tests, an over-length string never occurred using pseudo-random input data. Hence the number of presorted strings is always $\frac{M' \cdot s'}{(p_{opt}+1) \cdot s'} = \frac{M'}{p_{opt}+1}$.

To estimate the head travel distance for *Mergesort*, we assume that p_{opt} successive writes are followed by p_{opt} successive reads and vice versa, which is an optimistic guess. Next, within our linear block model, we assume that we merge from file F into file F' and F' is located right after (or immediately before) F, each of size M blocks. In other words, there is no interleaving (see Figure 5.4).

In each pass the head travels $2M/p_{opt}$ times a distance of M. When multiplied by the number of passes, which is $\log_{p_{opt}}(M)$ we arrive at a total of $O(M^2 \log_{p_{opt}}(M))$ blocks for the head travel in *Mergesort* in the linear block model without interleaving. This is by a factor of $\log_p(M)$ higher than the value for the external *Quicksort*! The analytical result is confirmed by our experimental findings which are given in the following tables.

[3] In the original paper, N denoted the block size. We changed this to s as we used N for total file size in records in the multiset analysis in this monograph.

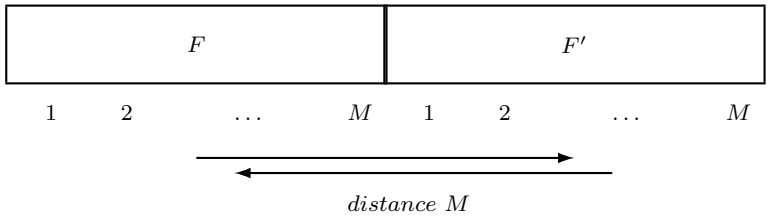


Fig. 5.4 Head travel in *Mergesort* without interleaving

The simulations were run for a file of 2400 records. The number was chosen because it divides easily for different core sizes and is large enough to show the efficiency of the programs.

The core size varies from 12 records to 1200 records. Thus between 0.5 % and 50 % of the file fits into the internal memory. Accordingly, M and s for EXQUISIT, respectively M' and s' for *Mergesort* are varied s.t. $2s$, resp. $(p_{opt}+1)\cdot s'$ equals the core size. As a consequence, the NBR-column gives the number of block reads of size s, resp. of size s'. Thus EXQUISIT transfers data in larger portions, except during the presorting phase of the merge sort, where filling the buffer was counted as one block read. In practice, the slightly smaller number of seek operations might be balanced by larger transfer times. For the core sizes 48 and 240 in the *Mergesort*-table, the counts NBR/NBW and DHT include copying the file back, since after the last pass, the file did not come to rest in its original place.

Note also, that the DHT-values (*distance of head travel*) are multiplied by s resp. by s', i.e. the distance is measured in records passed over, which is more precise if $M \neq M'$.

Table 5.1 EXQUISIT with '$(\min(F)+\max(F))/2$ strategy'

M	s	core	percent	NBR	NBW	DHT
400	6	12	0.5 %	3643	3540	779 490
200	12	24	1 %	1657	1642	444 336
100	24	48	2 %	728	722	247 344
40	60	120	5 %	215	210	107 280
20	120	240	10 %	86	84	59 760
10	240	480	20 %	35	34	35 520
4	600	1200	50 %	8	8	13 200

Table 5.2 EXQUISIT with '$(first + lastkey)/2$ strategy'

M	s	core	percent	NBR	NBW	DHT
400	6	12	0.5 %	4262	3726	786 960
200	12	24	1 %	1953	1832	440 352
100	24	48	2 %	868	845	257 760
40	60	120	5 %	300	298	132 120
20	120	240	10 %	123	123	78 600
10	240	480	20 %	48	48	46 560
4	600	1200	50 %	12	12	19 800

Table 5.3 Mergesort

M'	s'	core	percent	NBR/NBW	DHT	p_{opt}
800	3	12	0.5 %	4200	15 911 952	3
600	4	24	1 %	1900	6 707 080	5
200	12	48	2 %	900	3 310 872	3
80	30	120	5 %	260	903 960	3
50	48	240	10 %	120	422 496	4
30	80	480	20 %	35	113 600	5
6	400	1200	50 %	8	34 400	2

As a final word we like to add that the measured execution times indicated that EXQUISIT ran 25 % faster than *Mergesort* in our setting.

5.5 Verkamo's Observation

Verkamo examined the behavior of internal sorting algorithms applied to external files in connection with virtual memory (cf. [Ver86]) and variants of *Quicksort* adapted to external sorting [Ver87] [Ver88] [Ver89], including our algorithm EXQUISIT. Similar results can be found in an article by Alanko, Erkioe and Haikala on sorting in virtual memory [TAH84].

In [Ver87], Verkamo observes that *Quicksort* has two phases (with no sharp separation):

(1) a scanning – partitioning phase (long subfiles),
(2) a completion phase (short subfiles).

Keeping more than *two pages* in core during phase (1) does no good, i.e. a small working set is advantageous. However, phase (2) profits from a larger working set, which means that pages of completed subfiles are kept in core for a while. The usual LRU page replacement policy should help here.

Verkamo supplements his arguments with very illustrative array reference patterns which we cannot reproduce here. However, we copied the table with results for four *Quicksort* variants for external sorting [Ver87]. The comparison includes

- *Qsort*: the original *Quicksort*,
- *Psort*: a variable size *Quicksort*,
- *Esort*: EXQUISIT,
- *Wsort*: *Quicksort* combined with *Mergesort*.

Table 5.4 Parameter values.

file size (words)	1000, 2000, 5000, 10000, 20000
page size (words)	32, 64, 128, 256, 512, 1024
window size (refs)	1K, 2K, 4K, 8K, ..., 1024K

Table 5.5 Optimal window sizes for different file and page sizes.

file size	1000	2000	5000	10000	20000	selected window
Qsort:						
32	1K	1K	1K	1K	1K	1K
64	2K	1K	1K	1K	1K	1K
128	2K	2K	2K	2K	2K	2K
256	16K	4K	4K	4K	4K	4K
512	16K	8K	8K	4K	4K	8K
1024	–	8K	8K	8K	4K	8K
Wsort:						
32	32K	16K	16K	16K	16K	16K
64	64K	32K	32K	32K	32K	32K
128	128K	64K	64K	64K	64K	64K
256	64K	32K	32K	32K	32K	32K
512	4K	2K	32K	32K	32K	32K
1024	–	2K	1K	1K	1K	1K

Table 5.6 Processing times and number of faults.

	Qsort	Psort	Esort	Wsort
N = 1000				
proc. time:	.345	.351	.398	.372–1.238
number of faults:				
p=32	258	80	209	141
p=64	129	39	84	57
p=128	54	19	32	26
p=256	23	9	9	21
p=512	5	4	2	11
N=10000				
proc. time:	4.77	4.85	5.20	8.95–23.35
number of faults:				
p=32	4404	2279	3470	2398
p=64	2277	1138	1558	1061
p=128	1051	565	677	469
p=256	479	284	288	661
p=512	210	135	126	324
p=1024	105	65	49	243

Esort comes out with lower number of page faults than e.g. *Qsort*, but has slightly higher running times. *Psort* has extremely low page faults but only slightly lower execution times. *Wsort* varies very strongly in terms of processing time.

5.6 The Monard-Algorithm

Gonnet's Handbook of Algorithms and Data Structures contains a section on *External Quicksort* [Gon84, p. 160–162] which is based on an article by M.C. Monard [Mon80].

The basic idea is to keep a sorted buffer of size $k \geq 3$ and to try to balance the split, similar to van Emden's "improvement" [vE70b, vE70a]. The result of calling $partition(l, r, i, j)$ is a three-way split with subfile F_2 between $i+1$ and $j-1$ sorted (see Figure 5.5).
In-between the situation is as depicted in Figure 5.6. The algorithm adjusts i, iff procedure *extract_lower* writes a record r_x with $r_x.key < r_{lower\_bound}.key$

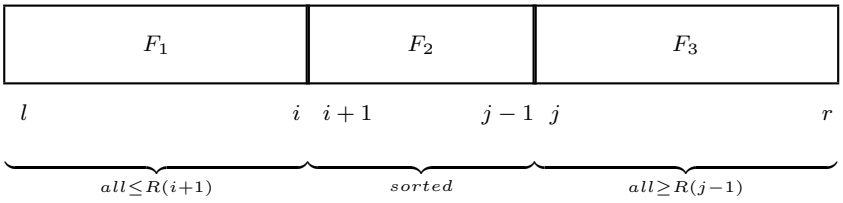


Fig. 5.5 Final situation after splitting $F(l..r)$.

back into the file, resp. adjusts j, iff *extract_upper* writes a record r_x with $r_x.key > r_{upper\_bound}.key$.

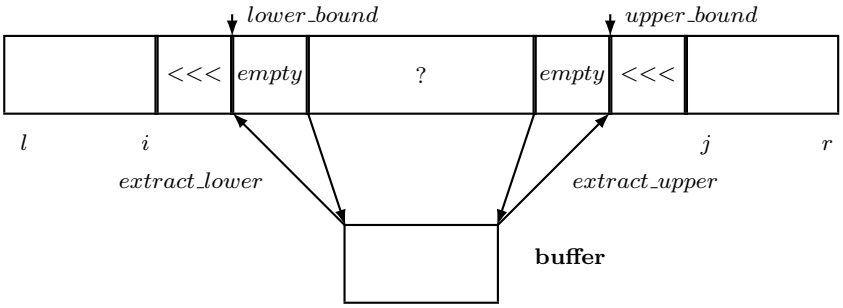


Fig. 5.6 Situation in-between with transfer from and to buffer.

Monard's analysis yields the familiar $O(N^2)$ worst case behavior for a file of N records. As for the average performance, both the paper by Monard and Gonnet's book report a running time proportional to

$$\bar{f}(N) \;=\; a \cdot N \cdot \ln N + O(N)$$

with

$$a \;=\; \frac{1}{H_{2K} - H_K}$$

which is approximately $\ln 2 \approx 0.693147$.

For a ten-page analysis, this is a weak statement and seems to indicate that block-I/O has not been given sufficient attention.

However, Gonnet gives an empirical formula for the expected number of passes:

$$E[P_N] = \log_2\left(\frac{N}{k}\right) - 0.924$$

with k the buffer size (in records). This implies the familiar

$O(N \log M)$	for the I/O-traffic (in records) and
$O(M \log M)$	for the page traffic (in pages).

Gonnet then concludes, because the algorithm works in place, that

> *External Quicksort* seems to be an ideal sorting routine for direct access files.[Gon84, p. 162]

This is a different view from what the discussion started out with. Even if *Quicksort* is not an *ideal* candidate, it seems that *Quicksort* is at least a serious contender for external random access sorting. Clearly, the two most prominent features of *Quicksort*, fast inner loops and only about $N/6$ swaps, do not have such a big influence in external sorting as they do not affect the I/O-traffic. Large main memories, which are common today, are of little good in the initial partition phases but should come into play for shorter subfiles. Overlapped I/O, smoothness both for multiset and presorted input, and a fail-safe behavior are issues which need further investigation.

Chapter 6
Quicksort in a Network

In this chapter we are looking at two network sorting solutions proposed by L. Wegner [Weg84] and D. Rotem, N. Santoro and J.B. Sidney [DRS85].

6.1 Applications and Alternatives

Network sorting applies to large files, distributed over M stations (hosts), connected by some type of network (LAN, WAN), e.g. a nationwide library information system, some office workload distribution, or sorting M disks with M processors.

Let N be the size of the file in records, M the number of stations, where we have n_i records in station i. Often we consider the special case $n_i = n = N/M \ \forall (1 \leq i \leq M)$.

Example 6.1. Consider a car rental agency. Assume 13 500 records, 1 000 Bytes each. The total file size is then 13.5 MB. Let there be 350 stations, connected by low bandwidth lines 64 KBit/s and about 300 ms packet delivery time. For the two network sorts under discussion (W $\equiv$ Wegner, RSS $\equiv$ Rotem/Santoro/Sidney), we would expect the following execution times[1].

6.2 Distributed Sorting

The sorting method of Rotem, Santoro and Sidney is similar to an interpolation distribution, but works in two steps:

(1) for $i = 1,\ 2,\ \ldots,\ M-1$ find the k_i-smallest key (all records belonging to station i), but without actually moving data;

[1] $\log M$ appears in both estimates; here $\log_2 350 \approx 8.45$.

Table 6.1 Sorting time influenced by data flow and message delivery.

Author(s)	asymptotic behavior	volume data/messages	time
Data flow			
W	$N \log M$	114 MB	4 h
RSS	N	13.5 MB	30 min
Messages			
W	$M \log M$	2957	14,7 min
RSS	$M^2 \log M$	1035125	86.5 h

(2) distribute all records according to the computed k_i values (implies giving away data to each other station and receiving data from each other station, on the average).

Step (1) requires

$M^2 \log(N)$ message pairs at most

$M^2 \log(M)$ message pairs on the average

Step (2) requires

M^2 messages

N data flow (number of records sent)

Rotem, Santoro and Sidney claim that their method is worst case optimal.

6.3 Quicksort as Network Sort

Wegner's suggestion is based on *Quicksort*'s divide and conquer method, but adopted s.t.

- there are no inter-station key comparisons
- data movement is in bulk
- all stations are kept busy (speed-up through parallel operations)
- there is little overhead for a controller node
- the routing requirement is simple
- no extra space is needed (in situ sorting)

Initially, all stations "belong" to file F (file F is stretched over all M stations). Each station i does a median-of-three sampling and returns the found key value H_i to the coordinator (e.g. one of the stations). The coordinator computes a pivot value H from the received H_i, e.g. as median or mean. This value H is then broadcast to all stations.

Table 6.2 Network sorting

Method	#messages	data flow	comparisons	extra space
central sort	$2M$	$2N$	$N \log N$	N
interpolation	M^2	N	$2N \log n$	n (1 Station)
iterated merging	M^2	$M \cdot N$	$N \log n + 2MN$	$2n \cdot M = 2N$
partition-exchange [Weg84]	$M \log M$	$N \log M$	$N \log N$	M (at 1 Station)
2-step distribution [DRS85]	$M^2 \log M$	$\max\{N, M^2 \log M\}$	?	M^2

Each station now splits its records according to H in the normal partition-exchange method of *Quicksort*. Each station then reports to the coordinator the number of keys less than or equal to H and the number of keys greater than H. The coordinator sums up these values and can then compute the global split position for the subfiles F_1 and F_2, say some position p in station k (cf. Example in Figure 6.1).

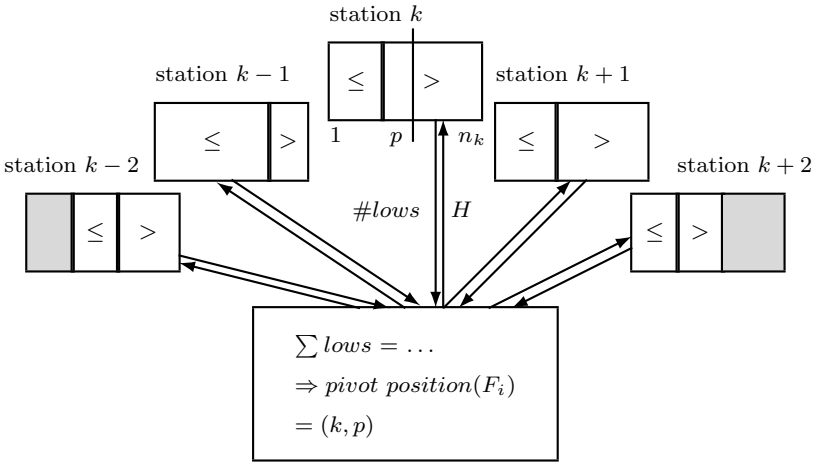


Fig. 6.1 Five stations splitting a subfile.

We now have $k-1$ stations which need to get rid of their records $> H$ and $M-k$ stations which must loose their records $\leq H$. This is done by pair-wise exchange between stations below k and above k. It would seem that one should try to match best fitting stations, but that is not true. In fact, the problem of finding a message minimal pairing of stations is (weakly) NP-complete. The proof uses a reduction to the SUBSET SUM problem.

Rather, the pairwise exchange between two stations i and j works in a way s.t. the block which is swapped has size $s = \min$(what i wants to send, what j wants to send). That needs 2 messages of size s. One station (the one that wanted to get rid of s records) is now exhausted, say i, the other station j has a remainder and now matches up with the successor $i+1$. This continues until F_1 and F_2 are formed with the final swap into and from station k.This leads to the following Lemma 6.2.

Lemma 6.2. *To split a M-stations subfile takes at most* $2(M-1)$ *messages.*

Proof. Each pair of messages exhausts at least one station. Thus

$$f(x,y) = \begin{cases} f(x-1,y)+2 \text{ for } x > 1 \\ f(x,y-1)+2 \text{ for } y > 1 \end{cases}$$

with $f(1,1) = 2$ and $M = x + y$.

Lemma 6.3. *The expected number of stations which a random subfile of size N occupies is*

$$\frac{N-1}{n} + 1,$$

with n the (uniform) station size.

Proof. By summing over all starting positions in the first station, which are equally likely, it follows that the file will stretch into one more station than it would need size-wise.

The number of messages needed to sort the distributed file of size N follows again from the general recurrence relation for $N > n$ with the factor 2 from the observation in Lemma 6.2.

$$S(N) = \ 2(N-1)/n + \sum_{1 \le k \le N-1} (p_k S(k) + p_{N-k} S(N-k))$$

where p_k is the probability, that a subfile of length k results. With naive sampling we get

$$S(N) = \ 2(N-1)/n + \frac{2}{N} \sum S(k) \qquad \text{with } S(n) = 0,$$

which leads to

$$S(N) = \ \frac{4(N+1)}{n} (H_N - H_{n+1}) + \frac{8}{n} - \frac{4(N+1)}{n(n+1)}.$$

With the usual estimate for the harmonic series we have

$$S(N) = S(n \cdot M) \approx 4M \left(\ln M + 0.57721 + \frac{1}{2M} \right) + \frac{8}{n} - \frac{4M}{n+1}.$$

Similarly the number of swaps is

$$\overline{S}(N) = \frac{N+1}{n+2} \cdot \frac{n-1}{n}$$

which is essentially the number of stations M. Alternatively, assuming perfect sampling, the recurrence becomes

$$S(N) = \ 2(N-1)/n + 2S(N/2)$$

which collapses for $N > n$ to

$$S(N) = \ 2M \log_2 M - 2M - 2M/n + 2/n.$$

The associated data flow results from shipping half of the N records in each of the $\log_2 M$ partitioning rounds, which gives

$$\frac{N}{2} \log_2 M.$$

To supplement the analysis, we present experimental findings in Table 6.3.

Table 6.3 Partition-exchange vs two-step distribution.

M	n	time	rounds	swaps	messages	flow
[Weg84]						
2	512	12.23	3	1	10	1666
8	128	12.48	5	6	82	2258
32	32	13.21	7	31	462	3900
128	8	19.16	9	112	2044	4806
512	2	37.54	11	232	5234	5462
[DRS85]						
2	512	23.70			2	524
8	128	17.18			56	886
32	32	12.24			609	995
128	8	8.46			983	1018
512	2	2.85			1019	1024

In summary, Wegner's partition-exchange moves records in multiple hops to their final destination. This leads to an increasingly larger number of messages and a high data flow as the number of stations grows. On the positive side, the book-keeping is simple and the partitioning in the later rounds can be done in parallel.

Rotem, Santoro and Sidney move the records only once. Thus the data flow is essentially N. The number of messages is $O(M^2)$ only if $M \approx n$ and each station gives records away to almost every other node. For the case of many stations with very few records, the method behaves like an internal pointer sort. Conversely, if there are very few stations, the book-keeping seems expensive and leads to surprisingly high running times.

Chapter 7
A Stackless Quicksort

7.1 Small wonders and a big problem

This chapter deals with a curious little affair. For years the standard phrase concerning extra space for *Quicksort* was " ... the stack can be limited to logarithmic size by sorting the shorter subfile first."

Suddenly, in 1986, three stackless *Quicksort*-variants appeared:

- Bing-Chao Huang and Donald E. Knuth [BCK86],
- Lutz M. Wegner [Weg87],
- Branislav Ďurian [Dur86].

The new phrase should therefore be " ... no extra space is needed if a stackless version is used".

7.2 The Ďurian-Algorithm

The basic idea of this stackless *Quicksort* is to search for the right end of next subfile to be split. The end is "marked" by the rule that the first key *not* belonging to the next subfile must have a value which is $>$ the first key in the subfile. Seen the other way around, we must move a key to the front of the subfile, say P' s.t. all other keys in the subfile are $\leq P'$. The pivot element from the previous split which created this (left) subfile fulfills this property (see Figures 7.1 and 7.2). Subfile S is partitioned around P s.t. $k_i \leq P$ in S_1, $k_i > P$ in S_2. At the end, switch (first key in S_2, P'), then sort S_1, afterwards search for end of S_2.

The sort, in the style given by Ďurian in [Dur86] is shown in Listing 7.1.

Listing 7.1 Stackless *Quicksort* as given in [Dur86].

```
procedure IQS(var n : integer);
{elements to sort are in A[1..n],
```

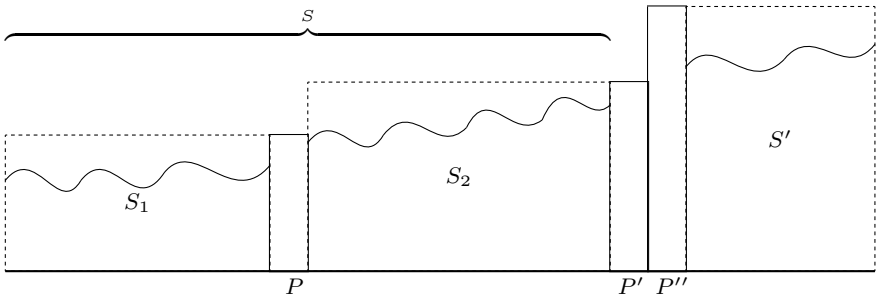


Fig. 7.1 Ďurian stackless *Quicksort*

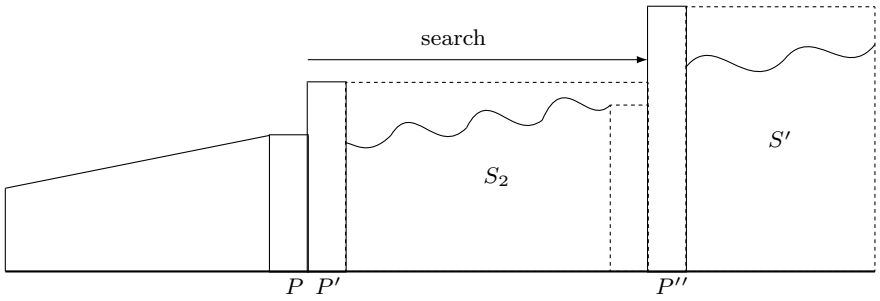


Fig. 7.2 Swapping the first key in S_2 with P'

```
   sentinels are A[n+1]:=maxval−1, A[n+2]:=maxval, maxval=∞}
begin
  l := 1; r := n + 1; m := 9;
  repeat
    while r − l > m do
    begin
      i := l; j := r; p := A[l]; {or choose pivot as median−of−three}
      repeat
        repeat i := i + 1 until A[i] ≥ p;
        repeat j := j − 1 until A[j] ≤ p;
        if i < j then swap(A[i], A[j])
      until i ≥ j;
      A[l] :=: A[j]; A[j] := p;
      swap(A[i], A[r]); {instead of pushing onto the stack}
      r := j
```

```
    end;
    l := r; repeat l := l + 1 until A[l] ≠ p;
    if l ≤ n then {instead of popping from the stack
                   do a sequential search for bounds}
    begin
      p := A[l]; r := l;
      repeat r := r + 1 until A[r] > p;
      r := r − 1; A[l] := A[r]; A[r] := p
    end
  until l > n;
  insertsort (n)
end {IQS}
```

Ďurian suggests as an improvement to use binary search in order to find the right border in less steps. He also claims that this stackless version is about 20 % slower than a good *Quicksort*.

7.3 The Algorithm of Bing-Chao Huang and Donald E. Knuth

This stackless *Quicksort*, called *OSort* in [BCK86], is a one-way sort which negates keys to indicate the end of a subfile yet to be sorted. The negated key is the pivot; the sort terminates with all keys negative. The algorithm needs a stopper, say key value -1, in position $A[N+1]$. Otherwise the technique is similar to other one-way partition strategies which "roll" records $> H$ to the right. Listing 7.2 shows the program as given by Huang[1] and Knuth.

Listing 7.2 B.-C. Huang and Knuth Stackless *Quicksort*.

```
procedure OSORT(N : integer);
label 3;
var L : 1..M; {progress indicator}
    J : 1..M; {end of current pivoting region}
    I : 1..M; {gap in current pivoting region}
    P : entry; {the pivot record}
begin R[N + 1].key := −1;
for L := 1 to N do
  begin while R[L].key > 0 do
    begin J := L; I := L; P := R[L];
      3: repeat J := J + 1 until P.key > R[J].key;
      if R[J].key > 0 then
```

[1] Bing-Chao Huang did his MS with Donald E. Knuth and his PhD with Michael A. Langston [Lan13]. For the *OSort*-paper he had switched first and last name creating some confusion in bibliographies.

```
      begin R[I] := R[J]; I := I + 1;
        R[J] := R[I]; goto 3;
      end;
      P.key := −P.key;  R[I] := P;
    end;
    R[L].key := −R[L].key;
  end;
end;
```

Obviously the sort is no general method as it works for positive integers only. Without better pivot selection and Insertionsort for short subfiles it is about 60 % slower than a good *Quicksort*.

7.4 The Wegner-Algorithm

GOSort below is taken from [Weg87]. *GOSort* stands for *Generalized OSort*. As the name suggests, it generalizes the idea from above using the maximum key from the left subfile as a stopper. The algorithm is rather complicated, possibly not optimal and quite slow. See Figure 7.3 for the general idea. What

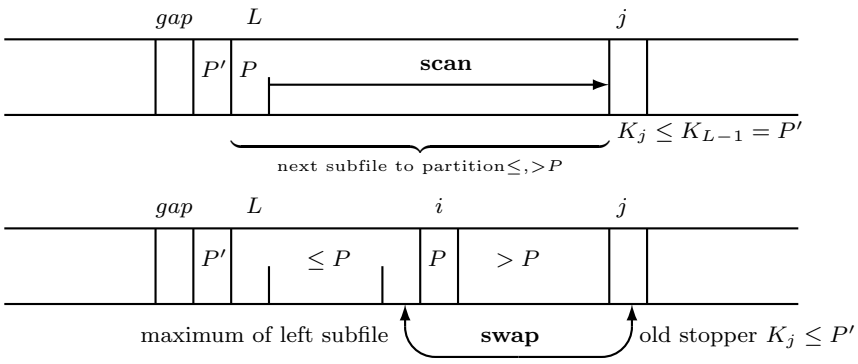


Fig. 7.3 Stackless *Quicksort* with maximum from left subfile as stopper

cannot be seen from Figure 7.3 is the terminating situation when a subfile has length ≤ 2. In that case we move record R_j to position $L-2$ (the gap) and advance L to $i+1$. Details can be taken from the implementation in Listing 7.3.

Listing 7.3 Wegner's Stackless *Quicksort*.

```
procedure GOSORT(N: integer; MINVALUE : keytype);
label 3;
var I : 1..NMAX;   {current pivot location}
    J : 1..NMAX + 1;  {end of current subfile}
    L : 1..NMAX + 2;  {left end of subfile}
    P : ENTRY;     {pivot record}

begin {GOSORT}
  L := 1; R[N + 1].key := MINVALUE; R[0].key := MINVALUE;
  while L <= N + 1 do
    if R[L].key <= R[L − 1].key
    then {R[L] is stopper; an empty subfile}
    begin R[L − 2] := R[L]; L := L + 2 end
    else
    begin J := L; I := L; P := R[L];
      3: repeat J := J + 1 until P.key >= R[J].key;
      if R[J].key > R[L − 1].key
      then {J not on stopper}
      begin
        if R[J].key >= R[I − 1].key
        then R[I] := R[J] {R[I] must be maximum of left subfile}
        else begin R[I] := R[I − 1]; R[I − 1] := R[J] end;
        I := I + 1; R[J] := R[I]; goto 3
      end
      else {J on stopper}
      begin
        R[I] := P;
        if L + 2 >= I
        then begin R[L − 2] := R[J]; R[J] := R[I − 1]; L := I + 1 end
        else begin P := R[I − 1]; R[I − 1] := R[J]; R[J] := P end
      end
    end {partition a non−empty subfile, end of while}
end {GOSORT}
```

7.5 Is There a Simple, Linear Time, Stable, In-place Partition Algorithm?

A stackless *Quicksort* is certainly an academic exercise. Nobody would accept a slower sort just to save $O((\log N)^2)$ extra bits. The true challenge is stable, in situ partitioning similar to stable, in situ merging. At the time of the Turku Lecture, no method was known. John A. Ellis from the Uni-

versity of Victoria in Victoria B.C., Canada had published an internal report (and confirmed it in personal communication) which claimed to provide a stable, linear average time, in situ partition algorithm based on the "perfect shuffle" [Ell85]. However, there seems to be no follow-up publication and the only other citation found concerns a paper on stable, in situ *merging*, jointly authored by John Ellis and Minko Markov [EM00].

Chapter 8
The Never Ending Wonders of the Heap

8.1 The Basic Heapsort

A *heap* in data structure terms is a complete binary tree stored in an array s.t. each node is $\geq$ its sons (cf. Figure 8.1). In particular, sons of a node

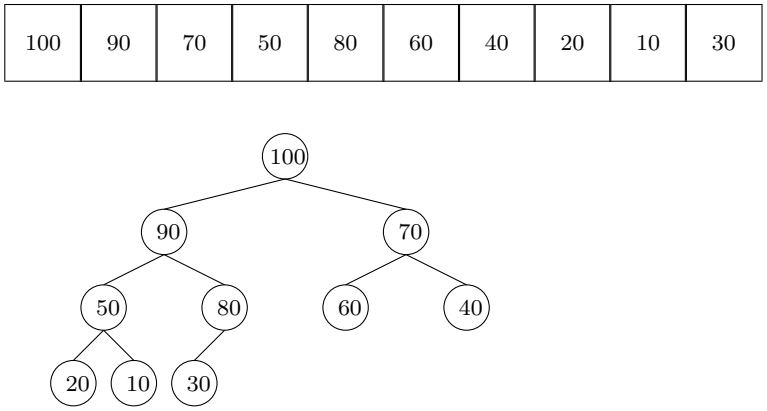


Fig. 8.1 Ten random nodes stored as heap.

at position i are at positions $2i$ and $2i + 1$. Essentially, this implements a priority queue where you can insert a new element (with a given priority) and delete the maximal element (the one having highest priority). It is easy to turn this structure with its sift-down operation into a sort. The original publications are due to J.W.J. Williams [Wil64] (*Heapsort*) and R.W. Floyd [Flo64] (*Treesort*). A textbook-version of *Heapsort* is given in Listing 8.1.

Listing 8.1 Internal HEAPSORT as a PASCAL-program.

```
{A textbook Heapsort as an internal sorting method.}
{It sorts the array a from 1 to r.}
procedure heapsort(var a: sequence; r: index);
var i: index;
    t: entry;

  procedure sift (var a: sequence; i, r: index);
  {Let a[i] sink into its proper position in heap ending at a[r].}
  var j: index;
    t: entry;
  begin
    while 2 * i <= r do {a[i] has a left son}
    begin
      j := 2 * i;
      if j < r
      then {a[i] has a right son}
        if a[j].key < a[j+1].key
        then j := j + 1; {right son greater}
      if a[i].key < a[j].key
      then {swap}
      begin
        t := a[i]; a[i] := a[j]; a[j] := t;
        i := j {keep on sifting}
      end
      else i := r {heap has been reconstructed}
    end {while}
  end {sift}

begin {heapsort}
  for i := r div 2 downto 1 do
    sift (a, i, r);
  {sort heap now by extracting continuously the maximum}
  for i := r downto 2 do
  begin
    t := a[i]; a[i] := a[1]; a[1] := t;
    sift (a, 1, i-1)
  end
end {heapsort}
```

The principle of *Heapsort*, as proposed by Williams [Wil64], is then to

1) build a heap bottom up (actually suggested by Floyd [Flo64]) and
2) remove the root and insert the rightmost leaf at the root position which sinks down.

For the first part, a simple worst case analysis could work like this:

- $N/2$ keys require no sinking in,
- $N/4$ keys sink at most 1 level,
- $N/8$ keys sink at most 2 levels,
- $N/16$ keys sink at most 3 levels,
- ...
- $N/N = 1$ key sinks at most $\lfloor \log_2 N \rfloor$ levels.

Formally, the number of sinking steps (levels) is (see also page 136 in the Appendix)

$$\begin{aligned}\sum_{i=1}^{\lfloor \log_2 N \rfloor} (i-1)\frac{N}{2^i} &= N\left[\sum_{i=1}^{x} \frac{i}{2^i} - \sum_{i=1}^{x} \frac{1}{2^i}\right] \qquad \text{with } x = \lfloor \log_2 N \rfloor \\ &= 2N - \lfloor \log_2 N \rfloor - 2 - (N-1) = N - \lfloor \log_2 N \rfloor - 1 \leq N.\end{aligned}$$

As for the sinking down phase, the worst case involves inserting N times into a heap of size $2, \ldots, N-1$ which requires at most 2 * maximal-path-length key comparisons [Knu73b, p. 155]:

$$\begin{aligned}2 \cdot \sum_{i=2}^{N} \lfloor \log_2 i \rfloor &= 2 \cdot \Big((N+1)\lfloor \log_2 N \rfloor - 2^{\lfloor \log_2 N \rfloor + 1} + 2 \Big) \\ &\approx 2N\lfloor \log_2 N \rfloor + 2\lfloor \log_2 N \rfloor - 4N + 4.\end{aligned}$$

The *average analysis* of the selection phase

> ... is another story, which remains to be written! [Knu73b, p. 156][Knu98, p. 155]

Actually, Knuth gave some pretty good estimates already in his first edition and adds more in his second edition citing R. Schaffer and R. Sedgewick as sources. Prior to that, Gonnet's Handbook [Gon84, p. 128] listed the average number of key comparisons as

$$E[C_N] = 2N \log_2 N - 3.0250N.$$

The average values from Knuth [Knu73b, pp. 148-157] are given in Table 8.1. For the values A, B and C he gives the following upper bounds:

- $A \leq 1.5N$,
- $B \leq N\lfloor \log N \rfloor$,
- $C \leq N\lfloor \log N \rfloor$.

8.2 Floyd‘s Improvement

In Exercise 18, Knuth asks (and answers) for an improvement of *Heapsort* which is attributed to R. W. Floyd [Knu73b, p. 158]. Because we insert a

	Quantity	Build-up	Selection	Total
P,	no. of sift-up passes	$\lfloor N/2 \rfloor - 1$	$N - 1$	$N + \lfloor N/2 \rfloor - 2$
A,	no. of passes where key ends up in an interior node	$0.1967N \pm 0.3$	$0.152N$	$0.349N$
B,	total no. of keys promoted up during sift ups	$0.74403N - 1.3 \ \ln N$	$N \log N - 2.61N$	$N \log N - 1.87N$
C,	total no. of cases where right son > left son detected	$0.47034N - 0.8 \ \ln N$	$\frac{1}{2} N \log N - 1.4N$	$\frac{1}{2} N \log N - 0.9N$
D,	there is no right son	1.8 ± 0.2 for N even 0 for$N$ odd	$\ln N \pm 2$	$\ln N$

leaf, it will most likely sink all the way to the bottom. The improvement is to sink a hole instead!

In the following example (Figure 8.2), we remove root 310 and insert the rightmost leaf 100. The keyphrase is a *path of maximal sons* along which the

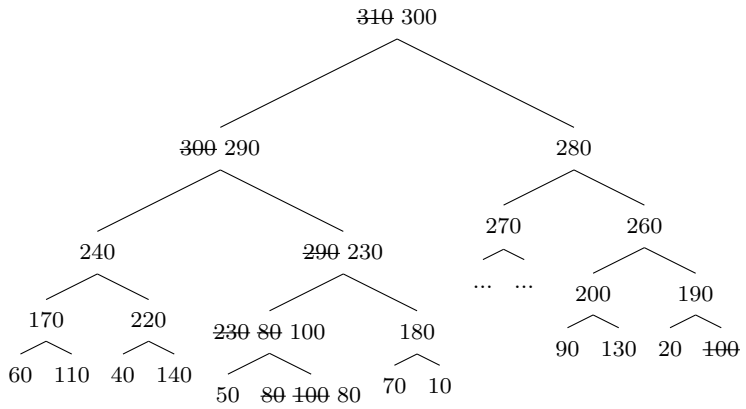


Fig. 8.2 Sinking a hole to insert node value 100.

hole sinks to the leaf-level. The probabilities of an inserted leaf to clime up i levels are [Knu73b, p. 619, answer to ex. 18]:

$i = 0:$ 0.848

$i = 1:$ 0.135

$i = 2:$ 0.016

This method of letting a hole sink down halfs the number of key comparisons and improves the asymptotic speed from $16N \log N + 0.2N$ to $13N \log N + O(N)$. We will return to this improvement in Chapter 10 in connection with external sorting.

8.3 Variations and Open Points

8.3.1 Minimum Heap

We look at some variations of the *Heapsort*-theme without claiming completeness. The first one is a heap with the minimum at the root – essentially it is a "trivial" modification which turns out to be nasty to program. Our version of the resulting sort, named MEAPSORT, is given in Listing 8.2.

Listing 8.2 Internal MEAPSORT – a *Heapsort*-variant.

```
{MEAPSORT}
{A modified version of heapsort is shown. It has the minimum at its}
{root and the root is to the right. This sort is used for sorting}
{pages in algorithm hillsort below. It's input is an array a from 1}
{to r. The secret formula for converting an index i in heapsort}
{into j in meapsort is j = r - i + 1.}
procedure meapsort(var a: sequence; r: index);
var i: index;
  t: entry;

  procedure sift(var a: sequence; i, k: index);
  {let a[k] sink into its proper position in the heap ending at a[i],
  i.e. k is the right end, i the leftmost leaf. The array stretches
  from 1 to r.}
  var j: index;
    t: entry;
  begin
    while 2 * k - r >= i do {a[k] has a son}
    begin
      j := 2 * k - r; {j on son with larger index}
      if j > i
      then {a[i] has a right son}
        if a[j].key > a[j-1].key
        then j := j - 1; {smaller index son has smaller key}
      if a[k].key > a[j].key
      then {swap because father is bigger}
      begin
        t := a[k]; a[k] := a[j]; a[j] := t;
        k := j  {keep on sifting}
      end
      else k := 0 {heap has been reconstructed}
    end {while}
  end {sift};

begin {meapsort}
  for i := r - (r div 2) + 1 to r do
    sift (a, 1, i);
  {sort heap now by extracting continuously the minimum}
  for i := 1 to r-1 do
  begin
    t := a[i]; a[i] := a[r]; a[r] := t;
    sift (a, i+1, r)
  end
end {meapsort}
```

8.3.2 Finding the k'th Largest Key

Rather than doing a complete sort, find the largest (smallest) k keys:

- build a heap $\Longrightarrow O(N)$ steps
- delete k times the root $\Longrightarrow O(k \cdot \log N)$ steps

8.3.3 Min-Max-Heaps

M.D. Atkinson, J.R. Sack, N. Santoro and T. Strothotte [ASSS86] consider *Min-Max-Heaps* as generalized priority queues. Such a structure is useful when both the maximum and minimum value are needed, say in a buffer with limited space which holds messages. We need both the most urgent message as well as the least important one to possibly make space for newly arriving posts. A heap which supports this scenario is depicted in Figure 8.3, taken from the aforementioned article. Note the alternating min-max levels.

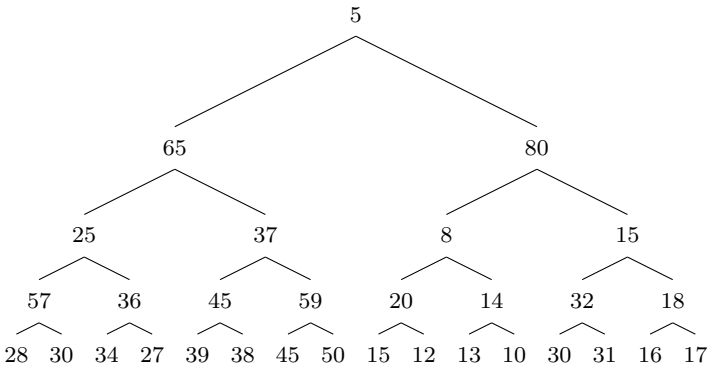


Fig. 8.3 A min-max-heap with alternating min-max-levels.

8.3.4 Median and k'th Largest Key

If the task is to insert, find or delete the median, resp. k'th largest key, a heap-like structure comes in handy (cf. Figure 8.4). The idea is to first build a *MIN-HEAP*, s.t. the root is in position k. This is done going bottom-up. Secondly, we build a *MAX-HEAP*, s.t. the root is in position $k-1$. Now for all $a[i].key$ in the *MAX-HEAP* check whether $a[i].key >$ the root of the

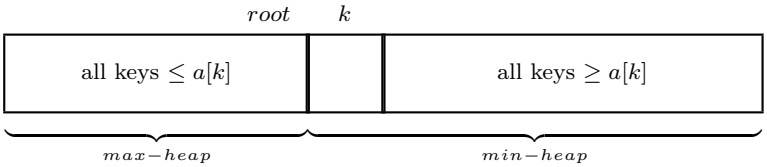


Fig. 8.4 The k'th largest key in array a.

MIN-HEAP. If yes, do a swap(i, k). Determining the running time for this scheme is non-trivial and skipped here.

8.3.5 Stability

Heapsort is an in-place sorting algorithm with an average and maximum running time of $O(N \log N)$. If it were stable, i.e. if records with equal keys retained their relative input order, all our desires were fulfilled. Consider Figure 8.5 with a unary file of six records. The subscripts denote the (invisible) ordering of the input. The input order was not retained, but this looks sus-

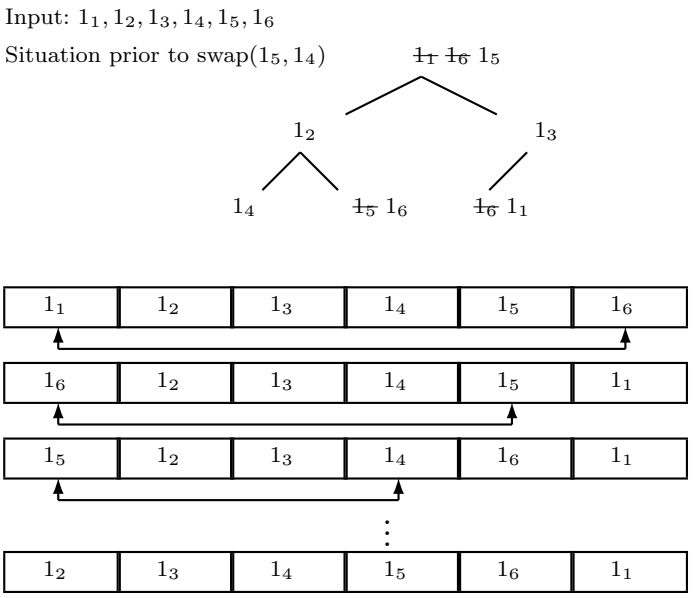


Fig. 8.5 *Heapsort* sorting a unary file.

piciously like a cyclic shift. However, that is not true in general as seen from a second example involving a binary file of Zeroes and Ones (cf. Figure 8.6).

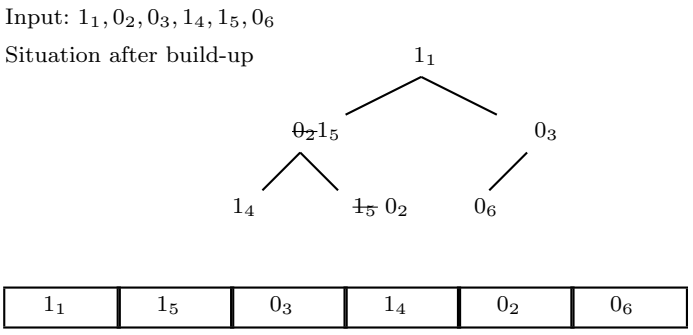


Fig. 8.6 *Heapsort* sorting a binary file.

The *Turku Lecture* ends here investigating stability, commenting that the situation is obscure, but not hopeless. As of this writing, no stable *Heapsort*-variant was found.

8.3.6 Deapsort – Duplicate Elimination with Heaps

This section describes a fictitious algorithm, termed DEAPSORT, which was suggested during the *Turku Lecture*. The idea is to combine sorting with on-the-fly, in situ duplicate elimination. In building the heap and in the selection phase, we could delete a key which is identical to its brother or son and replace it by the rightmost leaf (or leftmost leaf in a min-heap with root to the right). Note that this leaf might continue up or down.

As an example, consider a worst case heap s.t. no two brothers are equal, respectively no father is equal to one of the sons. Call such a heap a *duplicate-free heap* (df-heap)[1]. Figure 8.7 is a min-heap. The maximal values of N for $n = 1, 2, \ldots, 8$ in this example are 1, 2, 4, 6, 10, 14, 22, 30.

Observation: A df-heap with n distinct values has at most $N = f(n) = 2^{n/2+1} - 2$ nodes, for n even, and $N = (f(n-1) + f(n+1))/2$ nodes for n odd.

As another example consider an input of $N = 10$ records with $n = 5$ distinct values (Figure 8.8). If we choose to build a min-heap with the root to the right, we collect duplicates in the heap building phase on the left as

[1] Worst case in the sense that we want the largest df-heap with N nodes that can be build with n distinct keys.

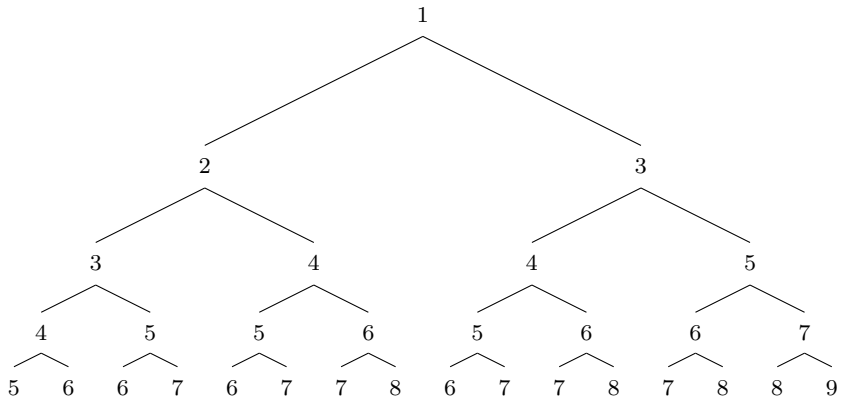
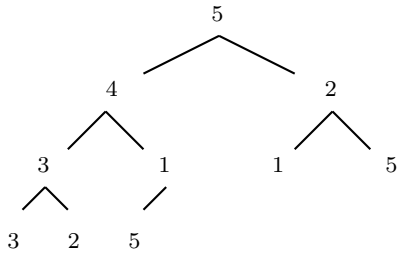


Fig. 8.7 A df-heap: no father-son or brother-nodes have equal keys.

Input: 5 2 3 5 1 1 3 2 4 5

Input as tree structure but without heap property yet



Possible output

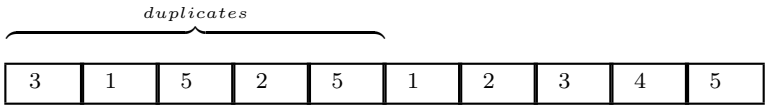


Fig. 8.8 Duplicate elimination in a min-heap.

shown in Figure 8.8. At the end of phase 1, however, the df-heap might still contain duplicates.

Thus we need to complete the sort by continuously extracting the root as in normal *Heapsort*. As for collecting the additionally detected duplicates in this phase, some new strategy is needed of where to keep the duplicates, possibly collecting them in a wheel. Another strategy could be to switch for phase 2 to a normal sort of the remaining records (including remaining duplicates),

say using *Heapsort*, and to eliminate the duplicates from the sorted sequence afterwards, say with the wheel technique.

In the example here, all duplicates were detected in phase 1. Note that the order of duplicates shown in Figure 8.8 depends very much on implementation details and might not be reproducible by any algorithm.

As the hand-waving here indicates, the idea was never followed-up with a concise algorithm. Instead Teuhola and Wegner later presented a minimal space, average linear time duplicate deletion algorithm, termed *DDT*, which is based on hashing [TW91].

Chapter 9
A Smooth Heapsort

This chapter discusses variations of *Heapsort* adjusted to presorted input. They were proposed by E.W. Dijkstra and A.J.M van Gasteren [DvG82, Dij82]. The algorithm in the second paper was named SMOOTHSORT by Dijkstra. Stefan Hertel rewrote this sort and analyzed it [Her83].

The basic idea in the variations is the same as in *Heapsort*, i. e. we have a tree of some type in the left side of the file from which we can extract continuously the maximum which we attach to the already sorted sequence on the right side of the file (cf. Figure 9.1). Now, however, the unsorted prefix

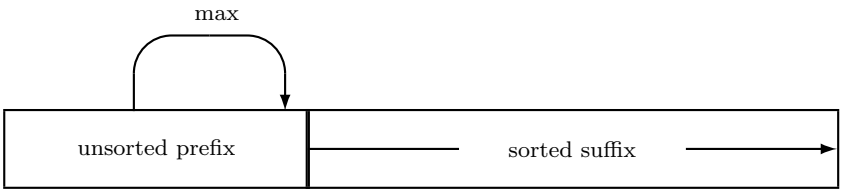


Fig. 9.1 Basic construction for *Heapsort* and variants

is a "leftward tree" with its root (maximum) to the *right*. Thus, the tree is pruned at the *root*! The sort, like in *Heapsort*, works in two phases: building the "tree" and depleting the "tree".

9.1 First Phase

We build a forest of trees whose size is determined by *Leonardo-numbers*.

$$L_n = L_{n-1} + L_{n-2} + 1$$

$$\ldots 41 \;\; 25 \;\; 15 \;\; 9 \;\; 5 \;\; 3 \;\; 1 \;\; 1 \;\; (-1)$$

The roots of the trees will be in *non-decreasing* order. The trees are stored in *post-order*, i.e. each son is situated to the left of its father. In contrast, heaps are usually stored level-wise.

Also note that in a Leonardo-tree, a node has either two sons or none. The ordering of trees is s.t. the bigger tree is to the left (cf. Figure 9.2). Next we

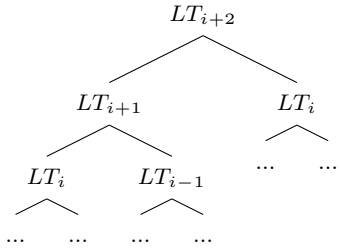


Fig. 9.2 Ordering of Leonardo-trees

need to look at a suitable representation of the Leonardo-numbers to keep track of which trees are present. Consider a

$$\text{triple } (p,\ b,\ c)$$

with

- p a bitstring which tells which trees are present; a bit-value '1' represents an occurring size; all rightmost 0's are deleted.
- b size of the smallest existing Leonardo-tree (tells how many zeros have been deleted)
- c next smaller Leonardo-number to b, i.e. if $b = L_i$ then $c = L_{i-1}$.

Example 9.1. The data are given in an array $A[0 \ldots 12]$, i.e. A has size 13. Its Leonardo-factorization is 9, 3, 1.

$$p' = 10110 \;\; \overrightarrow{\text{delete rightmost '0's}} \;\; p = 1011_2 = 11_{10}$$

$$(p,\ b,\ c) = (11,\ 1,\ 1)$$

We observe that – in this representation – there are never any two adjacent 1's, except for the rightmost two bits.

Next the authors provide procedures *Up, Up1, Down, Down1* to compute the next bigger (smaller) Leonardo-number, assuming global variables *b, c, b1, c1* (Listing 9.1).

Listing 9.1 Computing Leonardo-numbers.

```
procedure Up;
var aux: integer;
begin
  aux := b + c + 1;      {next bigger L−N}
  c := b;
  b := aux
end;                     {Up1 the same with b1, c1}

procedure Down;
var aux: integer;
begin
  aux := b − c − 1;      {next smaller L−N}
  b := c;
  c := aux
end;                     {Down1 the same with b1, c1}
```

The "magic" part is the new *sift*-procedure shown in Listing 9.2. Dijkstra explains it as follows [Dij82, p. 228].

- $sift$ applied to an element without larger son is a skip,
- $sift$ applied to an element $M[r1]$ that is exceeded by its largest son $M[r2]$ consists of a $swap(M[r1],\ M[r2])$ followed by a $sift$ applied to $M[r2]$.

An example is given in Figure 9.3, here with $sift$ for $key = 8$ at the root. Figure 9.4 shows the array storage afterwards.

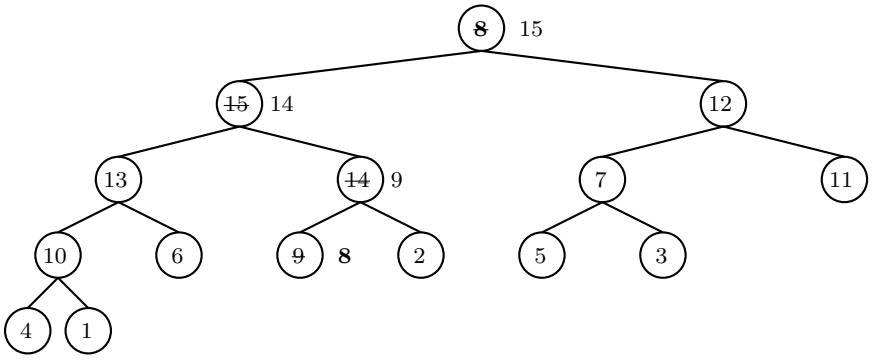


Fig. 9.3 Sift $key = 8$ into a heap ordered according to *Leonardo-numbers*.

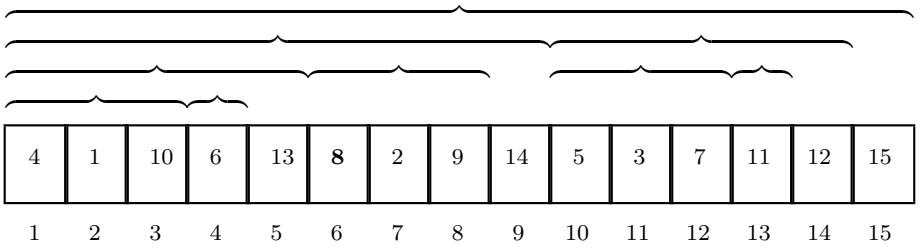


Fig. 9.4 Array storage with $key = 8$ ending up in a leaf position.

Listing 9.2 Procedure *sift* for heaps.

```
procedure sift;                                   {r1 is root}
  var r2: integer;
  begin
    while b1 >= 3 do
    begin
      r2 := r1 − b1 + c1;
      if M[r2] < M[r1 − 1]
      then begin r2 := r1 − 1; Down1 end; {largest son in r2}
      if M[r1] ≥ M[r2]
      then b1 := 1
      else begin swap(M[r1], M[r2]); r1 := r2; Down1 end;
    end {while}
  end {sift};
```

The meaning of *r1, r2, c1, b1* is depicted in Figure 9.5. As an example,

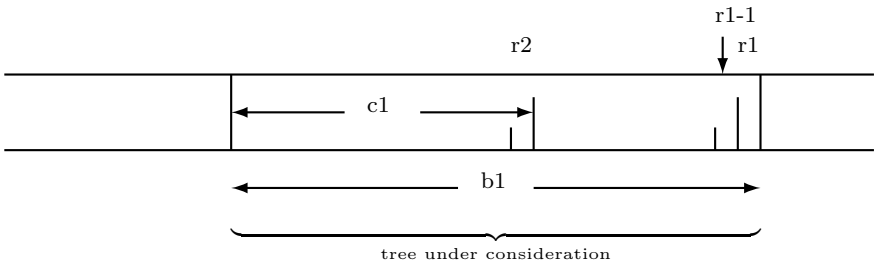


Fig. 9.5 Meaning of variables in procedure *sift*.

consider the input sequence in Figure 9.6. For $N = 13$ keys, the Leonardo-factorization is (9, 3, 1). The trees generated in the first nine steps are shown in Figure 9.7. Now, the next bigger Leonardo-tree won't fit into an $N = 13$ array. Therefore there is a procedure *trinkle* which has to restore a sorted-root forest, as shown in Figure 9.8. The rule for the exchange of roots is as

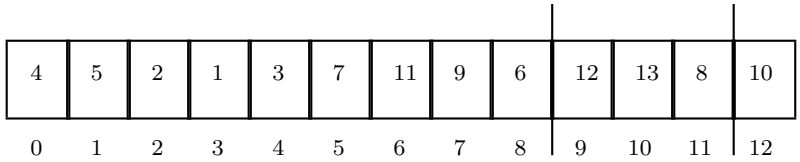


Fig. 9.6 Input for phase 1 – original order in array.

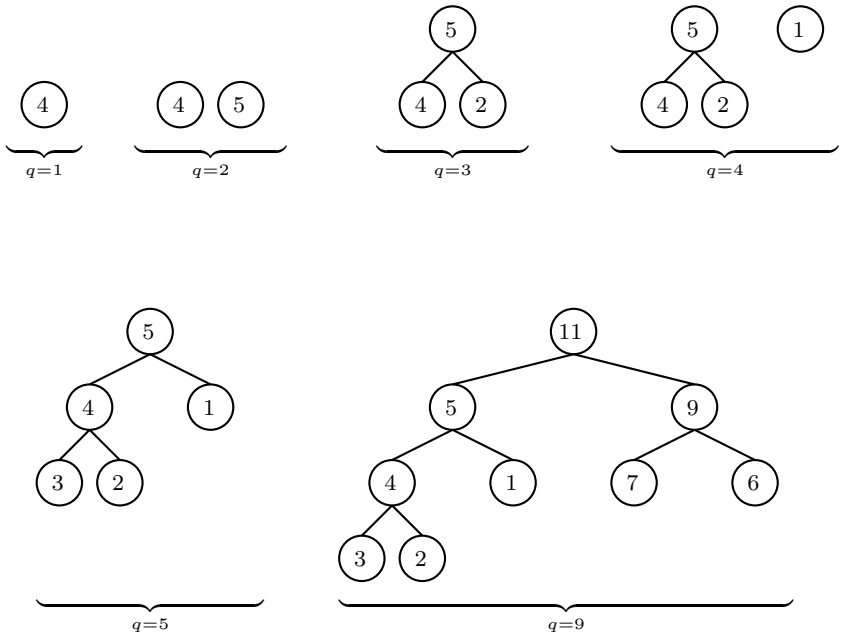


Fig. 9.7 Leonardo-tree building up to $q = 9$.

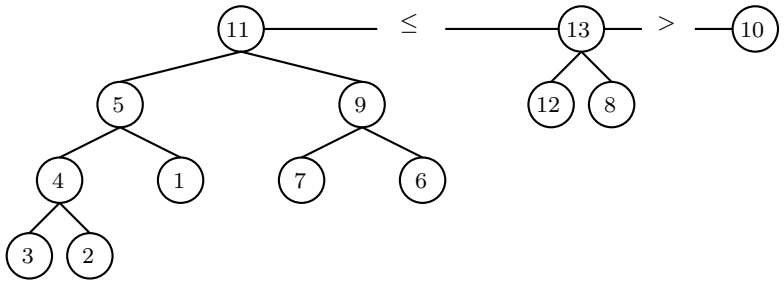


Fig. 9.8 Situation when calling *trinkle* for the given forest.

follows. Exchange the root of the current tree with the root of the tree to the left iff the root to the left is *larger* than the current root and larger than both sons of current root (if there are any). Otherwise sink the new root into the current tree. If an exchange happened, continue exchanging roots to the left.

In Example above (cf. Figure 9.8) we had $13 > 10$ and there were no sons for the single node 10. Therefore we exchange 13 and 10. This gives the situation shown in Figure 9.9. Now we have $11 > 10$, but 11 is *not* larger than

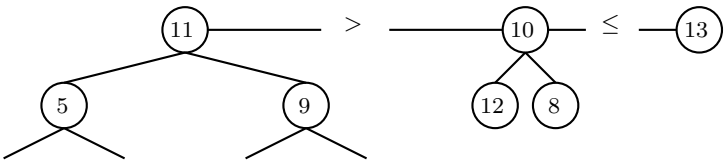


Fig. 9.9 After exchanging the roots of the rightmost "trees".

both sons of 10 which are 12 and 8. Therefore we sink 10 into the Leonardo-tree of size 3 giving the final situation shown in Figure 9.10. This terminates the explanation of the tree building phase.

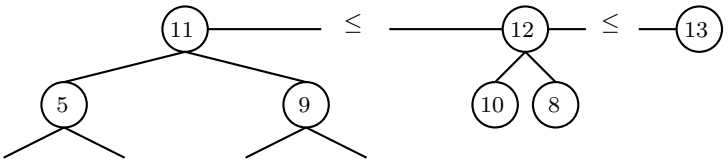


Fig. 9.10 Situation at the end of phase 1.

9.2 Second Phase

In the second phase we continuously delete the rightmost root (maximum). A variable q running from N to 1 controls the progress. If the root had two sons, use a procedure *semitrinkle* to take care of the two subtrees, exchange their roots if necessary and use *trinkle* (cf. Listing 9.3) to restore the *root-sorted forest*.

Listing 9.3 Procedure *trinkle* for heaps.

while q <> 0 **do**

```
begin {look at next node}
  q := q − 1;
  if b = 1 {a tree of size 1}
  then
  begin
    r := r − 1; {new root}
    p := p − 1; {delete tree of size 1}
    while (p mod 2 = 0) and (p > 0) do
    begin p := p div 2; Up end
  end {get rightmost LT}
  else
  begin
    p := p − 1;
    r := r − b + c; {on left subtree}
    if p > 0 then semitrinkle;
    Down;
    p := p * 2 + 1; {insert new tree}
    r := r + c;
    semitrinkle;
    Down;
    p := p * 2 + 1 {insert new tree}
  end
end;
```

Let us look at the rules for pruning the root of the first heap in a root-sorted forest. Figure 9.11 illustrates the situation in general. An arrow from node x to y indicates that $x \leq y$. Currently, node A is to be pruned. The

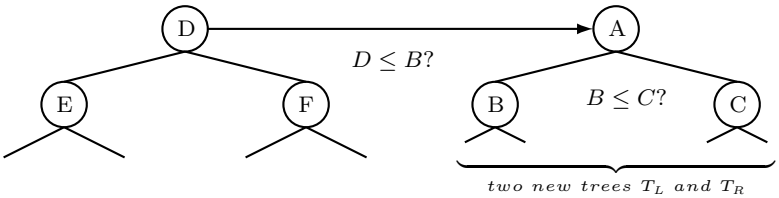


Fig. 9.11 Pruning node A.

algorithm first compares D with B using semitrinkle. If $D > B$, then swap $(D,\ B)$, which keeps heap property in T_L, and trinkle B into the root-sorted forest. Next, if root(T_L) $> C$ then swap(C,root(T_L)) and trinkle C into the root-sorted forest.

Consider our example again with node $A = 12$ being pruned (Figure 9.12). Since D is *not* $\leq B$, we need to call swap(11, 10) and must trinkle the value 10, which does nothing. Now 11 is not ≤ 8, we need another swap and trinkle 8 which again does nothing as 8 is a leaf node. This leads to a situation

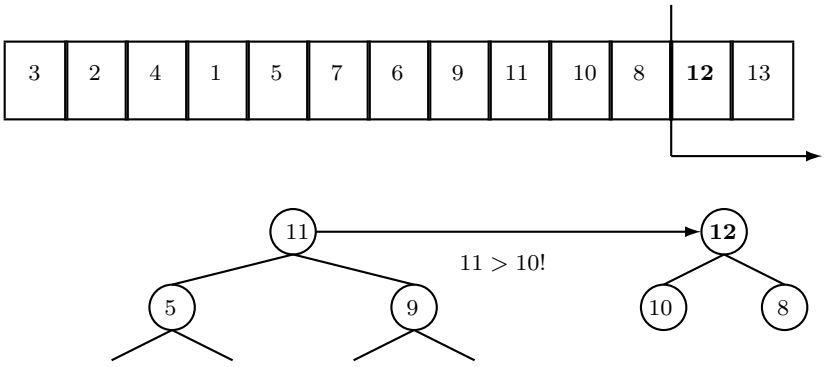


Fig. 9.12 Example: Pruning node $A = 12$.

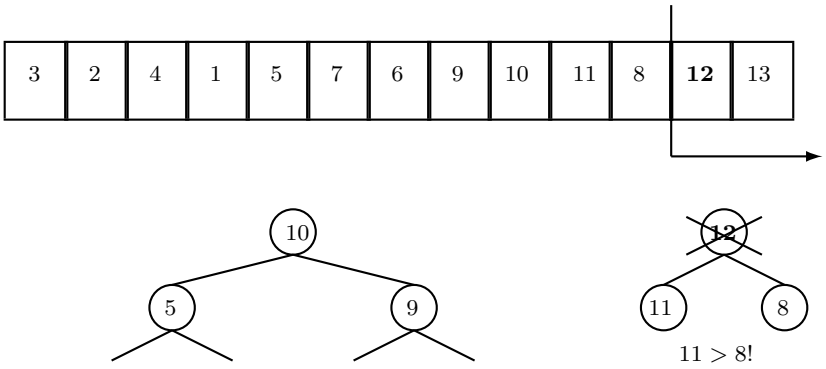


Fig. 9.13 After pruning node $A = 12$ and swap(11,10).

where the root of the left tree is 10, which is > 8. As a result, there is another swap(10,8) and 8 sinks into the heap bringing 9 to the root (Figure 9.14). Deleting afterwards 11, 10, 9 and 8 is straightforward and requires no swaps or calls of trinkle. After deleting 8 we come up with three "trees", 5, 7 and 6. Here, $5 \leq 7$, but $7 > 6$ which leads to swap(7,6) and trinkle for 6. The reader is invited to picture the situation and the following steps which include swap(4,1) and trinkle 1 which is a sift-operation. This concludes the description of phase 2.

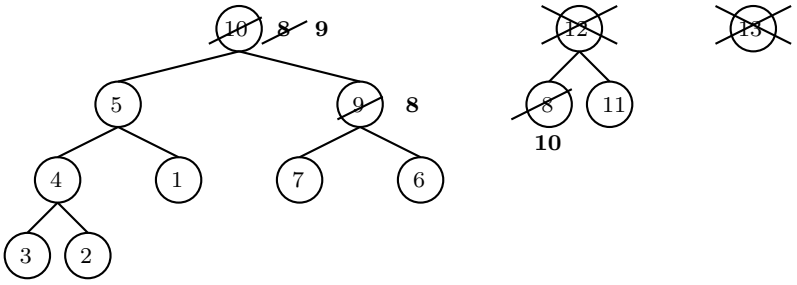


Fig. 9.14 After swap(10,8) and trinkle for value 8.

9.3 Hertel's Revision and Mehlhorn's Measure of Presortedness

Hertel [Her83] rewrote SMOOTHSORT using complete binary trees. He can show that the first phase has complexity $O(n)$, but SMOOTHSORT does not achieve Mehlhorn's [Meh79, Meh84] upper bound

$$O(n(1 + \log(F/n)))$$

for presorted input, where F is the no. of inversions. More precisely, $F = \sum_{i=1}^{n} f_i$ with $f_i = |\{x_j; j > i \text{ and } x_j < x_i\}|$[1]. Rather, in sorting a permutation with $O(n \log n)$ inversions, SMOOTHSORT may require $\Theta(n \log n)$ swaps.

[1] F is a measure of presortedness. A completely sorted file has $F = 0$, a completely unsorted file has $F \approx n^2$.

Chapter 10
Heapsort for External Sorting

10.1 Sorting Complex Objects

Consider a so-called NF^2 (Non-First-Normal-Form) relational database with relation-valued (nested) attributes, lists, multisets etc. (cf. Table 10.1). How would duplicate detection work for such a data model? If it were to be based on sorting, we need to define an ordering relation "<".

Let "<" be defined for atomic objects (integers, Booleans, characters, texts, reals, ...) as usual, i. e. we use a numeric, respectively lexicographic, ordering. For complex objects, there are several alternatives. We choose one based on a recursively defined minimum.

Definition 10.1. For objects A and B of the same type, define

$$A < B \iff$$

1) $FIRST(A) < FIRST(B)$ or
2) $FIRST(A) = FIRST(B)$ and $A \setminus FIRST(A) < B \setminus FIRST(B)$ or
3) A empty and B not empty,

where

a) $FIRST(\{\ \})$, $FIRST(<>)$, $FIRST([\ ])$ are undefined
b) for sets $FIRST(\{l_1, \ldots, l_n\}) = l_i$ with $l_i < l_j\ \forall\ i\ \neq\ j$
c) for lists $FIRST(< l_1,\ l_2, \ldots, l_n >) = l_1$
d) for tuples $FIRST([m_1 : l_1, \ldots, m_n : l_n]) = l_i$ with $m_i < m_j\ \forall\ i\ \neq\ j$, where $m_i < m_j$ refers to some pre-defined ordering of the columns, say by attribute names,

and $x \setminus FIRST(x)$ denotes object x with $FIRST(x)$ removed.

Example Consider a set of tuples (a relation) with atomic values.

{REPORTS}			
<AUTHOR>	TITLE	{DESCRIPTION}	
NAME		KEYWORD	WEIGHT
Dadam	Time Versions in NF^2-Database Systems visited	Databases	30
		Time Versions	50
Teuhola		NF^2	20
Dadam	Time Versions in NF^2-Database Systems Revisited	Time Versions	30
		Databases	50
Teuhola		NF^2	20
Wegner	Stackless Quicksort ...	Sorting	50
		Analysis	50
Teuhola	The External Heapsort	Sorting	25
		Heapsort	25
		NF^2	20
Wegner		Analysis	30

{PERSON}	
NAME	AGE
SMITH	40
JONES	40
SMITH	30
JONES	40

Example Consider a relation with a relation-valued attribute.

{PERSON}			
NAME	AGE	{CHILD}	
		NAME	AGE
SMITH	40	ANNE	18
		MIKE	16
JONES	40	ANNE	19
		JUDIE	23
SMITH	30	{ }	
JONES	40	ANNE	19
		JUDIE	23
		ANNE	19

To effectively support our ordering relation, we want a method which

1.) delivers the minimum in $O(N)$ steps,
2.) continues to deliver the next key in ascending order every $O(\log N)$ steps,
3.) can be "frozen" in-between,
4.) must work for external files.

The first two properties could be called "sorting in real time" or "on-line sorting". The following Table 10.1 reviews the competitors. Clearly, *Heapsort* is the most fitting candidate. *Heapsort* was invented by J. W. J. Williams in 1964 [Wil64] and is related to a previous scheme, called *Treesort*, invented by Robert W. Floyd [Flo62]. However, does it work for external files?

Table 10.1 Candidates for real-time sorting

Method	min key in $O(N)$	next every $\log N$ steps
Selection/Insert	$\surd$	–
Bubble	$\surd$	–
Merge	–	(–)
Radix	–	(–)
Shell	–	(–)
Quicksort	($\surd$)	($\surd$)
Heapsort	$\surd$	$\surd$

10.2 Heapsort in a Paging Environment

Timo O. Alanka, Hannu H. A. Erkiö, and Ilkha J. Haikala [TAH84] examined the behavior of sorting algorithms in virtual memory. Figure 10.1 shows a heap in a paged memory, where the pages have been filled level-wise from the top.

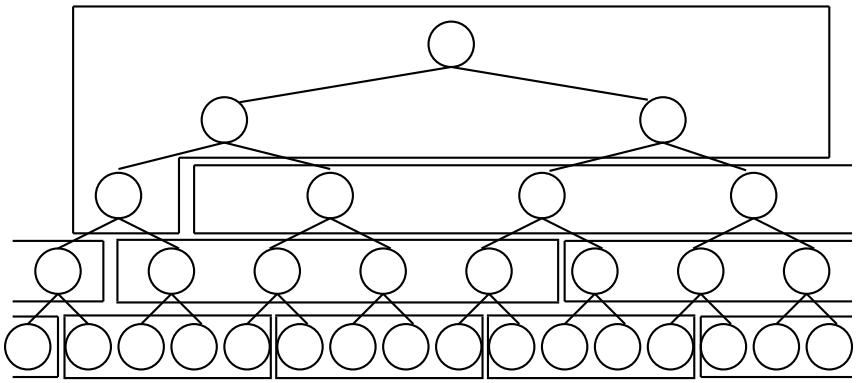


Fig. 10.1 A heap with four nodes per page.

For N records stored in m pages, each heapify step requires $\log_2 m$ page accesses. This implies $O(N \log_2 m)$ page accesses in total. A little back-of-the-envelope calculation gives the following estimate.

Example 10.2. Assume a 4 MB relation, 40 Bytes per tuple, the classical 4 KB page, a slow 18 ms disk. The total running time is then $100,000 \log_2(1000) \cdot 18/1000[s] = 5[h]$.

However, our aim is a $m \log_2 m$ method, which would sort the relation in 3 minutes!

10.3 Hillsort - An External Heapsort

A solution for an external *Heapsort* was devised by Teuhola and Wegner and ultimately published as HILLSORT in 1989 [WT89]. The basic idea is as follows.

- Consider a heap of pages.
- Keep the records sorted in ascending order within the pages.
- Use a new invariant for the ordering of pages (cf. Figure 10.2).
- Replace key comparisons by merge operations of pages.

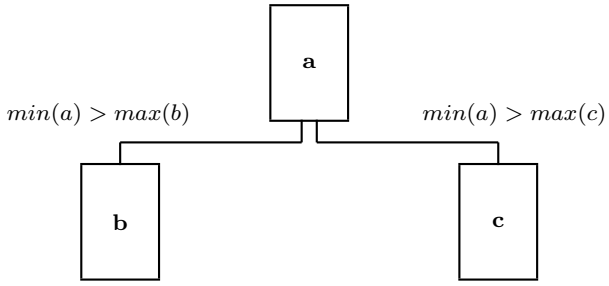


Fig. 10.2 Invariant for heap of pages.

This ordering of a, b and c can be achieved with two merge operations of pages (see Listing 10.1).

Listing 10.1 Maintaining the heap invariant in HILLSORT.

```
if min(b) < min(c)
then
begin
  merge_pages(b, c); highson := c {the upper half is in c}
end
else
begin
  merge_pages(c, b); highson := b {the upper half is in b}
end;
merge_pages(highson, a) {the upper half is in a}
```

In the actual PASCAL-program, this routine is called as `merge_pages(f, i, j)`, where f is a file (array) of pages and parameters i and j are indices

(integers) into the array of pages. Procedure `swap_pages(f, i, j)` does the obvious, i. e. exchanges the two pages i and j. Finally, `heapify(f, i, r)` restores the heap property s.t. page i contains the largest records in the file. It uses internally `merge_pages()` from above.

A first sketch of HILLSORT along the lines of a classical, basic *Heapsort* is given in Listing 10.2.

Listing 10.2 First sketch of HILLSORT.

```
procedure Hillsort(var f: paged_file; m: page_no);
var i: page_no;
  procedure sort_page(var f: paged_file; i: page_no);
    ...
  procedure swap_pages(var f: paged_file; i, j: page_no);
    ...
  procedure heapify(var f: paged_file; i, r: page_no);
    ...
begin {body of Hillsort}
  for i := 1 to m do sort_page (f, i);
  for i := m div 2 downto 1 do heapify (f, i, m);
  for i := m downto 2 do
  begin
    swap_pages(f, 1, i);
    heapify(f, 1, i−1)
  end
end {of Hillsort};
```

We now look at details of procedure `heapify(f, i, r)`. In case a reader remembers that there is an improvement of *Heapsort* due to Robert W. Floyd, which lets a "hole" sink into the heap [Flo64], we are sorry to say that this does not work for a heap of pages. However, there is a similarly smart idea as part of the cited article [WT89], described below in Section 10.5.

Listing 10.3 The heapify algorithm adapted to a heap of pages.

```
procedure heapify (var f: paged_file; i, r: page_no);
var
  k: page_no;
  stop: boolean;

  function min(i: page_no): keytype;
  begin min := f[i, 1].key end;

  function max(i: page_no): keytype;
  begin max := f[i, pagesize].key end;

  procedure merge_pages(var f: paged_file; i, j: page_no);
  var aux: page_type;
```

```
    procedure merge(var i, j, k: sequence; length: integer);
      ... {an ordinary mergesort with auxiliary space} ...

  begin {merge_pages}
    merge(f[i], f[j], aux, pagesize)
  end; {merge_pages}

begin {body of heapify}
  while 2*i <= r do
  begin
    j := 2 * i;
    k := j;
    stop := false;
    if j < r {i has right son}
    then
    begin
      if min(j) < min(j+1) {lower half to j}
      then k := j + 1
      else j := j + 1;
      if min(i) >= min(k) {will heapifying end?}
      then stop := true;
      if max(j) > min(k) {are sons disjoint?}
      then merge_pages(f, j, k) {larger half to k}
    end;
    if min(i) < max(k) {are father and son disjoint?}
    then
    begin
      merge_pages(f, k, i);
      if stop then i := r else i := k
    end
    else i := r {heapifying ends}
  end {while}
end; {heapify}
```

The final HILLSORT-algorithm has a few subtle refinements, e. g. there is an additional mapping from real pages to virtual pages to avoid physical swapping. Initial sorting of pages is done using an internal *Heapsort*, merging of pages uses an ordinary internal merge. They are omitted in Listing 10.4.

Listing 10.4 The complete HILLSORT with indirect page addressing.

```
procedure Hillsort(var f: paged_file; var p: page_table;
                   m: page_no; pagesize: index_in_page);
var i, aux: page_no;

  procedure sort_page(var f: paged_file; i: page_no);
```

```
    procedure heapsort(var a: sequence; r: index);
      ... {use any suitable internal sort} ...

  begin {sort_page}
    heapsort(f[p[i]], pagesize)
  end; {sort_page}

  procedure swap_pages(i, j: page_no);
  var aux: page_no;
  begin aux := p[i]; p[i] := p[j]; p[j] := aux end;

  procedure merge_pages(var f: paged_file; i, j: page_no);
  var aux: page_type;

    procedure merge(i, j, k: sequence; length: integer);
      ... {use any suitable merge algorithm} ...

  begin {merge_pages}
    merge(f[p[i]], f[p[j]], aux, pagesize);
  end; {merge_pages}

  function min(i: page_no): keytype;
  begin min := f[p[i],1].key end;

  function max(i: page_no): keytype;
  begin max := f[p[i], pagesize].key end;

  procedure heapify(var f: paged_file; i, r: page_no);
    ... {see Listing 10.3 } ...

begin {body of Hillsort}
  for i := 1 to m do sort_page(f, i);
  aux := p[m];
  for i := m downto 2 do p[i] := p[i−1];
  p[1] := aux; {root = physically the last page}
  for i := m div 2 downto 1 do
    heapify(f, i, m); {build the heap}
  for i := m downto 2 do
  begin {swap the root and i'th page logically}
    swap_pages(1, i);
    heapify(f, 1, i−1)
  end
end; {Hillsort}
```

10.4 A Little Analysis Never Hurts

The number of page accesses per heapify step is

$$2 \cdot (pathlength + 1) = \#merges + \#swaps + 2.$$

The total number of page accesses is then

$$\begin{aligned} &\leq 2 \sum_{i=2}^{m} (\lfloor \log\ i \rfloor + 1) \qquad \text{[Knu98, p. 154f]} \\ &\approx 2m \lfloor \log m \rfloor + m \\ &+ 3.5\ m \qquad \text{for building the heap bottom-up.} \end{aligned}$$

In total, the number of ordering operations is $\leq 2m\lfloor \log m \rfloor + m$. This matches the observed value for a page size > 2 which is reported in [WT89] as $2m \log_2 m - 2.925m$.

As for the buffer size, anything larger than four pages improves the performance. The number four derives from the fact that we need to keep a "father" and two "son" pages in core and need one extra page for merging. If we increase the buffer above the four pages, say we buy k extra pages, we can hold $\log(k+2) - 1$ top-levels of the heap and therefor save $O(m \log k)$ accesses. This yields a speed-up from $O(m \log m)$ to $O(m \log(m/k))$.

Figure 10.3 shows the composition of ordering operations between pages for various number of records per page. The special case of a page size 1 turns HILLSORT into *Heapsort* with no merging. As page sizes increase, the effect of randomness decreases.

> In other words, once the heap has been built it has a very regular structure where brothers are not disjoint and keys on level i tend to be larger than keys on level $i+1$. As a result, a page sinking down practically never stops before it reaches the bottom (Knuth's value $A = 0.349N$ [Knu73b, p. 149], here $0.344m$ for $s = 1$, becomes 0 for $s > 10$). Similarly, the number of father-son merges is surprisingly low. Since we heapify $1.5m$ times and since the largest number of father-son merges we measured was $1.425m$, each page comes to a halt with less than one merge on the average [WT89, p. 924].

10.5 A Heap with Holes

As we mentioned above, Floyd suggested to let a "hole" sink down the heap which saves comparisons between father and son [Flo64] and requires only one son-son comparison. Once the hole hits the bottom (becomes a leaf), insert the new key and let is sift up to its final position. This will not be very high, Knuth gives for the sift-up levels (0, 1, 2 from the bottom) the respective probabilities (0.848, 0.135, 0.016).

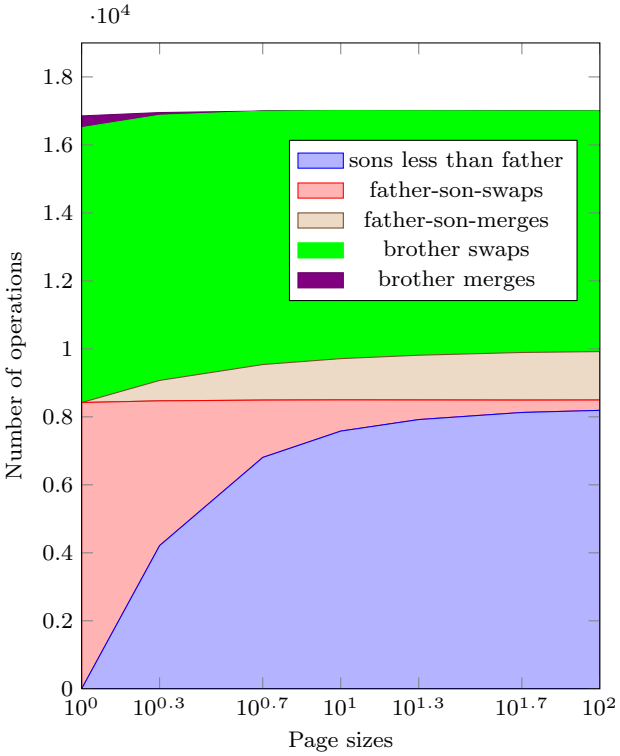
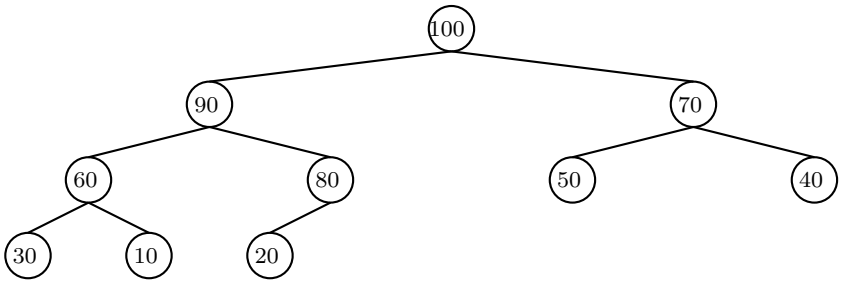


Fig. 10.3 Composition of operations for HILLSORT with $N = 1000$ and varying page size.

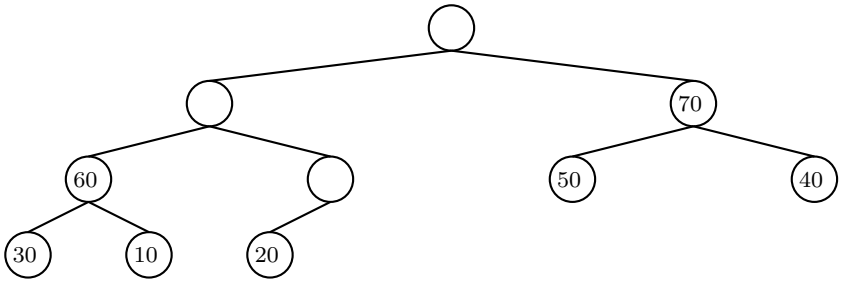
In HILLSORT we observe the same behavior, almost all father-son operations become a father-son swap rather than a merge. However, the swap comes for free with virtual addressing, the real cost is the loading of the two brother pages per level. This cost remains the same regardless of whether we sink a hole or a real page [WT89, p. 921]. Figure 10.5 shows the composition of the heap during the extraction phase of the modified HILLSORT, starting out with all real pages and ending with black holes only.

Thus, a totally new idea is needed and it was Teuhola's idea to suggest a heap of constant size with an increasing number of holes. This is similar to a tournament sort which continuously extracts the minimum and inserts a new hole until there are only holes left. The improved HILLSORT provides an additional table for that with two bits/page. They are sufficient to distinguish between

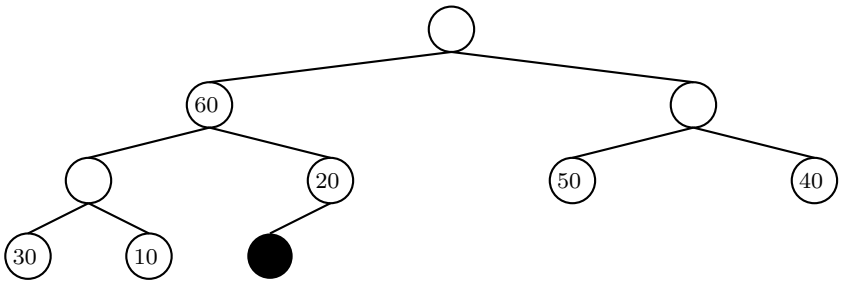
- real pages,
- white holes (never have a white hole as brother),
- black holes (are the root of an empty heap).



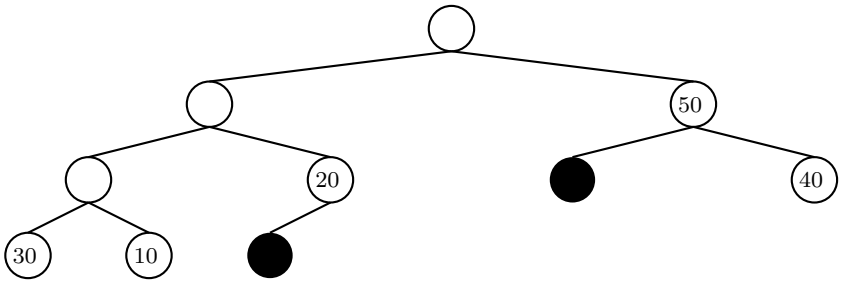
(a) Original heap



(b) After deleting 100, 90, 80



(c) After deleting 70



(d) After deleting 60

Fig. 10.4 Removing the first five records from a heap-with-holes.

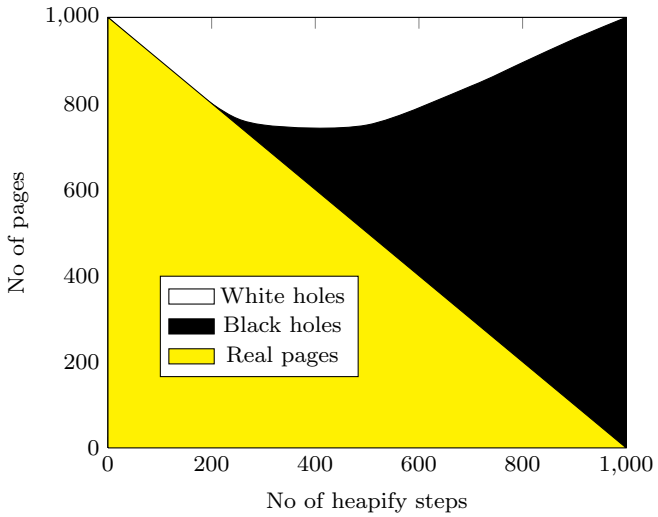


Fig. 10.5 Proportions of holes in modified HILLSORT.

The extraction process is very similar to heapify() in *Heapsort* and described in detail in [WT89]. Figure 10.4 hints at the principle by removing the first five records from a heap-with-holes holding originally ten records.

Although the path length in this heap-with-holes remains fixed at $\log m$, as compared to a shrinking path length otherwise, the savings are large: we can cut the number of page I/O-operations during this phase in half. The reason is that in a full binary tree, the average path length differs from the maximum path length by only 1. On the other hand, in sifting a page up, we load at most one page per level (the sifted up page remains in core). Indeed, empirical evidence in [WT89] from a simulation suggests that the modified HILLSORT with holes reduces the number of accesses by 40 percent and the CPU time by about 15 percent as compared to the unmodified HILLSORT. Having brother pages in successive locations on the disc reduces seek time and is another advantage apart from being truly in situ.

Chapter 11
Sorting Based on in situ Merging

11.1 State of the Art in Minimum Space Merging

We consider here methods other than *Quicksort*- and *Heapsort*-derivatives for stable, in situ sorting. Clearly, this must include the in situ merging solutions already mentioned in Chapter 3. At the time of the TURKU LECTURES, Gonnet [Gon84, p. 146] had reported the following results about stable in situ merging (Table 11.1). During the lecture, three stable, minimum space

Table 11.1 Characteristics of in-place merging algorithms (from [Gon84])

comparisons	add. space (words)	stable?	source
$O(N)$	$O(1)$	no	Kronrod [Kro69]
$O(N)$	$O(\log N)$	yes	Horvath$^{(*)}$ [Hor74]
$O(N)$	$O(1)$	yes	Luis Trabb Pardo [Par77]
$O(kN)$	$O(N^{1/k})$	yes	Wong [Won81]
$O(m\log(n/m+1))$	$O(\log m)$	yes	Dudzinski&Dydek$^{(\dagger)}$ [DD81]

$^{(*)}$ involves manipulation of keys.
$^{(\dagger)}$ $n+m=N,\ n\geq m$ without restricting generality; conjecture is that it could be modified to work in $O(1)$.

sorts were briefly discussed, as mentioned in Chapter 3. They were the contributions of Rivest [Riv73][1], Preparata [Pre75] and Dudzinski&Dydek [DD81]. We will sketch the basic ideas below.

[1] Actually, Rivest's solution is not a stable merge algorithm but rather provides in situ partitioning. We include it here among the competitors for the *Quicksort*- and *Heapsort*-variants.

As can be seen from Table 11.1, the Dudzinski&Dydek-algorithm, which involves a recursive merge with in situ swapping of strings, is included, but the Rivest and Preparata solutions are not. Nor is the solution by V. Pratt included, which we also mentioned in Chapter 3 following Knuth's hint [Knu73b, p. 665]. Another solution not in the table, but mentioned in the TURKU LECTURES, is the stable, minimum storage sorting algorithm due to Robert B. K. Dewar [Dew74]. It sorts a file of size N in time $O(N^{3/2})$ both in the worst case and on the average. The space requirement is $O((\log_2 N)^2)$ extra bits.

Not mentioned in the TURKU LECTURES, but from that time, is a solution by Svante Carlsson [Car86]. He also provides another overview (see Table 11.2) including his own algorithm. The second Trabb Pardo entry there refers to Trabb Pardo's BLOCKMERGE solution with a slightly higher number of assignments, the entry by Donald E. Knuth is termed LINEAR MERGE.

Method	extra space (bits)	assignments	comparisons
[Car86]	$O(m \log n)$	$O(m+n)$	$O(m \log(\frac{n}{m}+1))$
[Par77]	$O(\log n)$	$O(m+n)$	$O(m+n)$
[DD81]	$O((\log m)(\log n))$	$O((m+n)\log m)$	$O(m \log(\frac{n}{m}+1))$
[Par77]	$O(\log n)$	$O(m^2+n)$	$O(m+n)$
[HL72]	$O(\log n)$	$O(m+n^2)$	$O(m \log(\frac{n}{m}+1))$
[Knu73a]	$O(m+n)$	$O(m+n)$	$O(m+n)$

Table 11.2 Worst case behavior for stable merging

Finally, there is yet another interesting contribution by Mannila and Ukkonen, titled "A Simple Linear-Time Algorithm for in Situ Merging" [MU84]. It is not included in the table above, because it is a *non-stable* merge. However, it claims to be an $O(n)$-time, $O(1)$-space, simple solution to this classical problem. We will give some details below in Section 11.6.

11.2 Pratt's Solution.

This review refers to the remark by Donald Knuth [Knu73b, p. 665] concerning stable merging.

> Given two non-decreasing runs α, β to be merged, determine in a straightforward way the subruns $\alpha_1\alpha_2\alpha_3\beta_1\beta_2\beta_3$ such that α_2 and β_2 contain precisely the keys of α and β having the median value of the entire file. By successive "reversals", first forming $\alpha_1\alpha_2\beta_1^R\alpha_3^R\beta_2\beta_3$, then $\alpha_1\beta_1\alpha_2^R\beta_2^R\alpha_3\beta_3$, then $\alpha_1\beta_1\alpha_2\beta_2\alpha_3\beta_3$, we are in a position to complete the merge recursively on subfiles $\alpha_1\beta_1$ and $\alpha_3\beta_3$ which are of length $\leq N/2$.

We illustrate the idea with an example (cf. Figures 11.1, 11.2 and 11.3). The

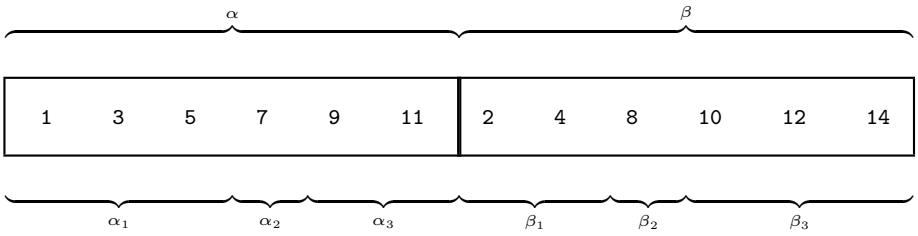


Fig. 11.1 Stable merging of runs α and β around medians α_2 and β_2.

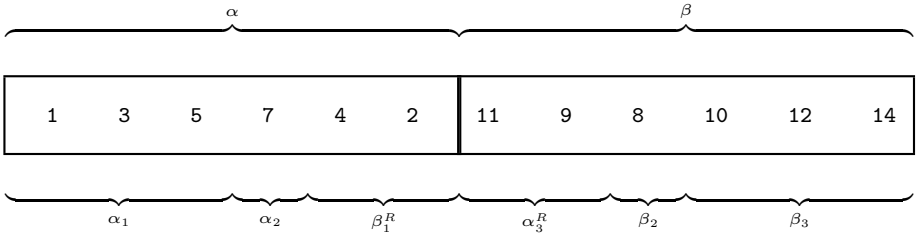


Fig. 11.2 Situation after first reversal of α_3 and β_1.

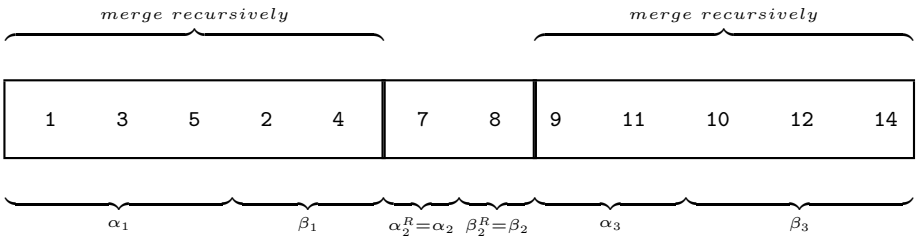


Fig. 11.3 Situation after second and third reversal.

reader will find a very similar set of figures in the context of the proposal by Dudzinski and Dydek [DD81], discussed in Section 11.5 below.

11.3 The Rivest-Algorithm

Rivest's stable, in situ partitioning [Riv73] is a combination of three clever sub-algorithms, denoted A, B and C here.

Algorithm A { calls B within} We are to partition sub-array $M[b..c]$. If there are at least two records to split, use the last key as pivot.

$$\text{if } b \geq c \text{ then stop else } m := M[c] \ \{pivot\ m\}$$

Now divide $M[b],\ \ldots,\ M[c]$ in a stable way using *Algorithm B*, s.t. afterwards

$$\underbrace{M[b], \ldots,}_{\leq m}\underbrace{M[k],}_{=m}\ \underbrace{M[k+1], \ldots,\ M[c]}_{>m}$$

Next, use a variant of *Algorithm B* to collect all keys $= m$

$$\underbrace{M[b], \ldots,\ M[k'],}_{<}\ \underbrace{\ldots,\ M[k],}_{=\,m}\ \underbrace{M[k+1], \ldots,\ M[c]}_{>}$$

and sort the remaining subfiles $M[b..k']$ and $M[k+1..c]$ recursively.

Algorithm C {called within B} We are to exchange two adjacent sequences of unequal length in array M in a stable way.

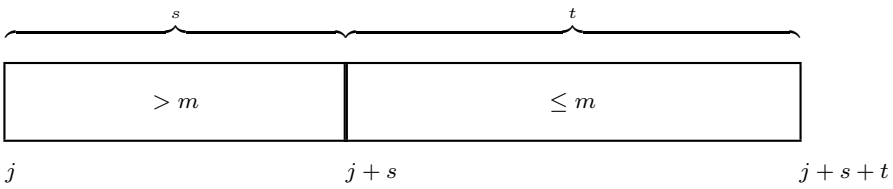


Fig. 11.4 Stable exchange of two sequences of unequal length.

Listing 11.1 Rivest's suggestion for the in place exchange of two unequal length strings [Riv73].

```
d := min (s, t);
while d > 0 do
begin
  for i := 0 to d − 1 do
    M[j + s + i] :=: M[j + s + i − d];
  if s < t
  then begin j := j + d; t := t − d end
  else s := s − d;
```

$d := \min(s, t)$
end {*while*}

The situation after the first exchange is shown below in Figure 11.5.

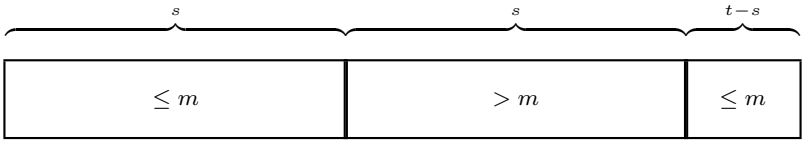


Fig. 11.5 After first exchange of sub-sequences of equal length.

Observation: The stable exchange in Listing 11.1 requires $s + t$ exchanges, when both s and t are even, $s+t-1$ otherwise, because each exchange places one record into the final position[2].

Algorithms B {called within A, calls C inside} We are to partition array $M[l..r]$ in several passes over the file, swapping adjacent "blocks" of records as shown in Figure 11.6.

Listing 11.2 Partitioning the file in several passes.

{*we are given array* $M[l..r]$ *and pivot* $m := M[r]$}
B1: {*skip over elements in place*}
  while $(l < r)$ **and** $(M[l] \leq m)$ **do** $l := l + 1$;
  while $(l < r)$ **and** $(M[r] > m)$ **do** $r := r - 1$;
  if $l \geq r$ **then** stop;
B2: $j := l$;
B3: $s := 0;\ t := 0$;
  while $M[j + s] > m$ **do** $s := s + 1$;
  while $(j + s + t < r)$ **and** $(M[j + s + t] \leq m$ **do** $t := t + 1$;
B4: Exchange s keys $> m$ with t keys $\leq m$
  $M[j], \ldots, M[j + s - 1]$ with $M[j + s], \ldots, M[j + s + t - 1]$
  $j := j + s + t$ {*now* $j > r$ *or* $M[j] > m$}
B5: **if** $j > r$ **goto** B1; {*next pass*}
B6: {*continue pass, skip next two runs*}
  while $M[j] > m$ **do** $j := j + 1$;
  while $(j < r)$ **and** $(M[j] \leq m)$ **do** $j := j + 1$;
  goto B1

[2] Let w^R denote the mirror image of string w. The oldest known method for exchanging two strings u and v of unequal length in place is by forming $(u^R v^R)^R = vu$. The origin of this algorithm is lost in the sands of history. This mirror exchange also requires $s + t$, resp. $s + t - 1$ swaps in total.

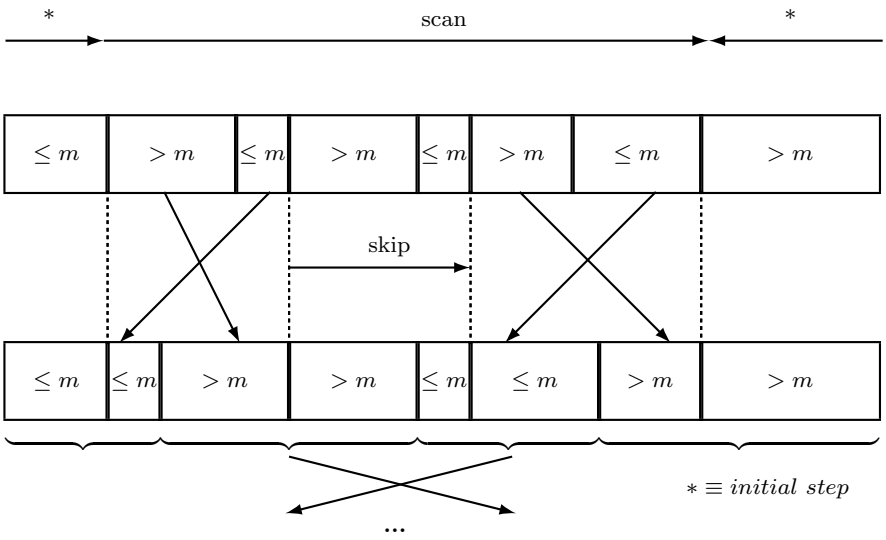


Fig. 11.6 Partitioning the file in several passes.

Analysis At each partition pass, the number of "runs" is halved, i.e. there are $\leq \log_2 n$ passes, each requiring n comparisons and $\leq n$ exchanges. Thus one complete partitioning stop requires $O(n \log_2 n)$ time instead of n. There are $O(\log_2 n)$ partitioning levels – on the average – and therefore $O(n(\log_2 n)^2)$ steps on the average.

11.4 The Preparata-Algorithm

While the Rivest-Algorithm is a partition exchange sort like *Quicksort*, the Preparata-Algorithm presented now is a stable, in situ merge sort. Sorted sequences of increasing length $(4,\ 8, \ldots,\ N/2)$ are merged using *recursive merging* (to be later changed into iterative merging in order to make it a minimum space sort). The method to exchange strings of unequal length, say r and s, is different from the one in Rivest's stable, in situ sorting algorithm and uses only $r + s + \gcd(r, s)$ assignments, as compared to $3(r + s)$ in the solution by Rivest[3].

Consider two sorted sequences as shown in Figure 11.7.

Listing 11.3 Basic loop to merge sequences in place.

$i := j := 0$;
while $i + j <$ desired size **do**

[3] As we mentioned, $x :=: y$ is implemented as $temp := x; x := y; y := temp$.

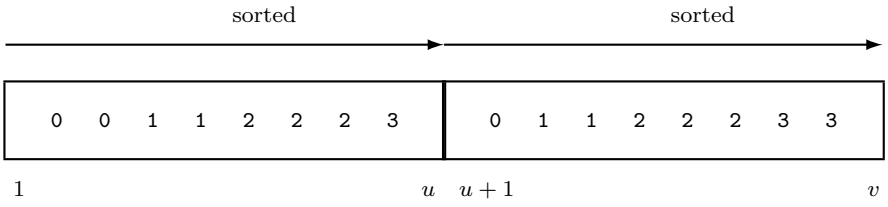


Fig. 11.7 Starting out with two sorted sequences.

if $(V[i+1] \leq V[u+j+1])$ **and** $(i < u)$ **or** $(j = v-u)$
then $i := i+1$
else $j := j+1$;

Assume the desired size is 8. Then the basic loop stops with the situation given in Figure 11.8.

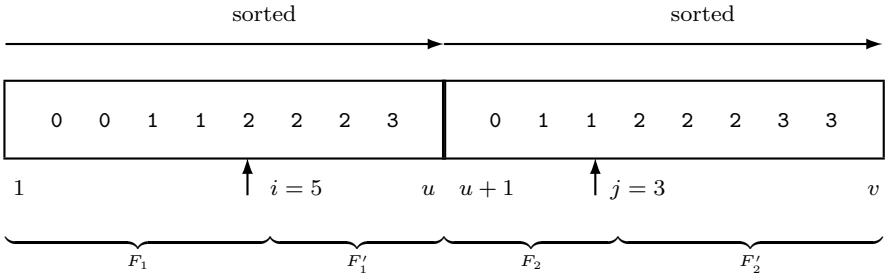


Fig. 11.8 The loop stops with $i + j \geq$ desired size 8.

Now stably exchange F_1' and F_2 and merge recursively F_1 with F_2 and F_1' with F_2' as shown in Figure 11.9.

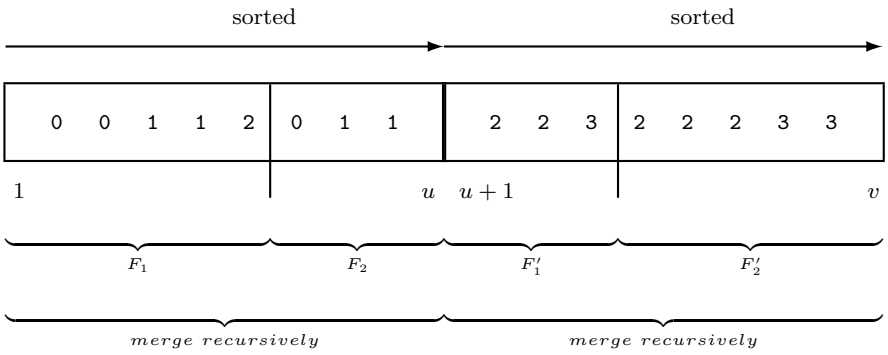


Fig. 11.9 After swapping F_1' and F_2.

The stability of the method follows from the invariant which we can derive.
Invariant: $\max(F_1, F_2) \leq \min(F_1', F_2') \wedge$ if $\max(F_1,\ F_2) = \min(F_1',\ F_2') = x$, then all occurrences of x in F_1 and F_2 are to the left of all occurrences of x in F_1' and F_2'!

11.5 The Dudzinski-Dydek-Algorithm

After Luis Trabb Pardo had solved the stable minimum storage sorting problem in 1977 with a merging algorithm which required $O(m+n)$ assignments and comparisons for sorted vectors of length n and m, Dudzinski and Dydek came up in 1980 with another stable merging algorithm, called RECMERGE. Their merging of vectors of length m and n ($m \leq n$) needs $O(m \log(n/m+1))$ comparisons and $O((m+n)\log m)$ assignments in $O(\log m)$ space. The basic idea is best explained by an example (cf. Figure 11.10).

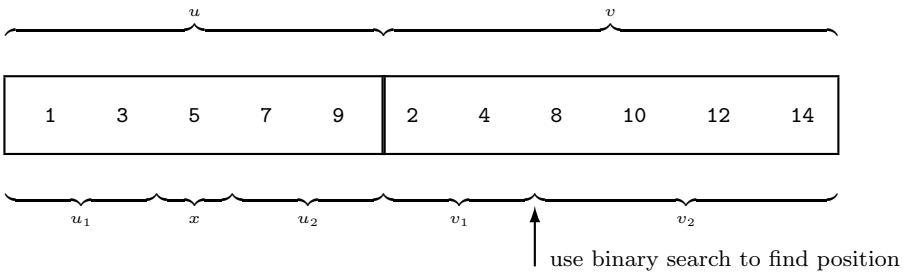


Fig. 11.10 Stable merging of sorted vectors u and v around pivot x.

After finding the split position, stably exchange xu_2 with v_1. This leads to the situation shown in Figure 11.11.

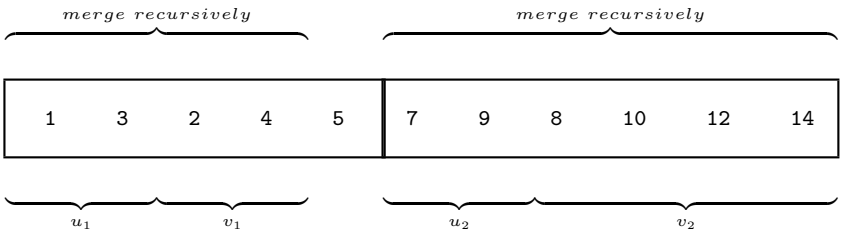


Fig. 11.11 Situation after exchange and prior to recursive merging u_1 with v_1 and u_2 with v_2.

Obviously this requires a high number of swaps, namely $O((n+m)\log m)$ with $n \geq m$. S. Dvorák and Branislav Durian [DD88] later claim to have improved their method, called merging by decomposition, and add a version with $O(1)$-space requirement.

11.6 The Mannila-Ukkonen-Algorithm

The table above does not include a contribution by Mannila and Ukkonen, titled "A Simple Linear-Time Algorithm for in Situ Merging" [MU84], because it is a *nonstable* merge. However, it claims to be an $O(n)$-time, $O(1)$-space, simple solution to this classical problem.

The basis idea is to divide the given two lists $C[1 \ldots n],\ C[n+1 \ldots n+m]$ into sublists, where the lists in the first sequence $C[1 \ldots n]$ are of equal length $\lceil \sqrt{n} \rceil$ and the lists of the second sequence are of *variable length*.

Thus $C_1 = C[1 \ldots n] = A_1 A_2 \ldots A_k$ (k sublists), $C_2 = C[n+1 \ldots n+m] = B_1 B_2 \ldots B_k$ (k sublists), where the B_i are chosen s.t. merge(C_1, C_2) = merge(A_1, B_1), merge(A_2, B_2), ..., merge(A_k, B_k). Figure 11.12 shows an example.

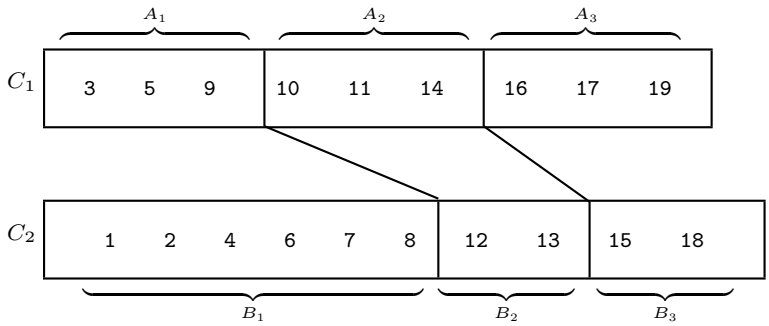


Fig. 11.12 Sequences C_1 and C_2 and their corresponding subsequences.

Finding all B_i's requires $\sqrt{n} \cdot \log(m)$ time using binary search. Next rearrange $C_1 C_2 = A_1 \ldots A_k B_1 \ldots B_k$ s.t. $A_1 B_1 A_2 B_2 \ldots A_k B_k$ and merge(A_i, B_i) $\forall i$. This merging of A_i with B_i is done using Kronrod's idea of a $\sqrt{n}$ "scratch pad" inside the file (Figure 11.13).

When merge(X, Y) is called using Z as work space, Z is disturbed, but can be resorted in linear time in place, say by *Insertion Sort* or *Heapsort*.

Next look at the rearrangement of blocks. The basic idea there is to use a "wheel" of blocks $A_i,\ A_{i+1}, \ldots$, but not necessarily in original order (Figure 11.14).

More precisely, the "wheel" is a permutation of $A_j, \ldots,\ A_k$ with at most one A_i possibly split into A' and A'' (Figure 11.15).

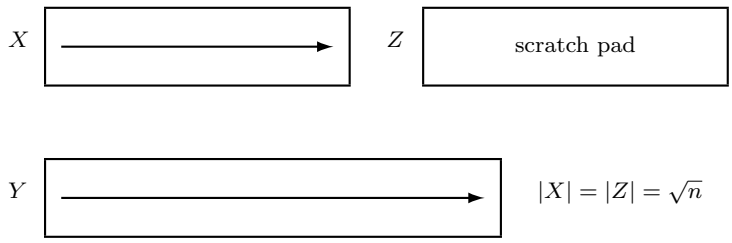


Fig. 11.13 Sequences X, Y and *scratch pad* Z.

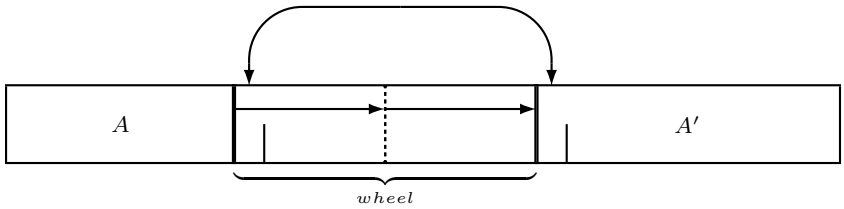


Fig. 11.14 A wheel between A and A'.

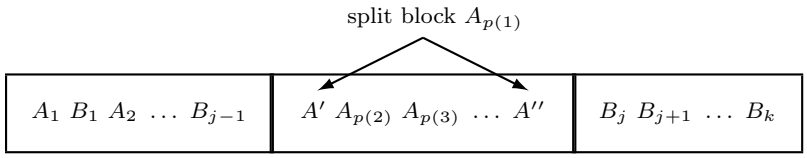


Fig. 11.15 A wheel of A-blocks with one block split.

To find A_j, search for the block (fixed sizes!) with smallest first (or last) key. This takes $O(\sqrt{n})$ time. Move the block to the front, insert some other block into the free space and roll the wheel to the right to bring B_j to the front. All this requires constant work space. Note also that the indizes to the blocks B_j are found "on-line" and therefore require no extra space.

The article does not say where stability is lost, but it seems that it happens both in the "wheel" and in the $\sqrt{n}$ "scratch-pad".

11.7 The Trapp-Pardo-Algorithm

As Knuth already mentioned in his first edition of TAoCP Vol. 3 [Knu68, p. 665], Luis Trabb Pardo, who happens to be his own Ph.D. student, came up with a stable merging algorithm which works in $O(N)$ time and gives rise to a stable sort with asymptotic time $O(N \log N)$ and $O(\log N)$ extra bits of auxiliary space. The solution involves no key transformations.

In his second edition of Vol. 3 [Knu98, p. 702], Knuth mentions additionally the contribution by B.-C. Huang and M. A. Langston [HL92] with the same asymptotic time and space requirements, but much better constants. Indeed, their algorithm seems to be the champion according to Katajainen, Pasanen, and Teuhola [KPT96, p. 2], who proposed their own in-place *Mergesort* based on the original Kronrod method [Kro69]. Here we look at the basic features of the solution by Trapp Pardo.

There are two auxiliary transformations: *stable insertion* of two adjacent blocks and *direct merging* of blocks.

(a) Stable insertion of two subsequent blocks: insert(U, V)
The idea is depicted in Figure 11.16.

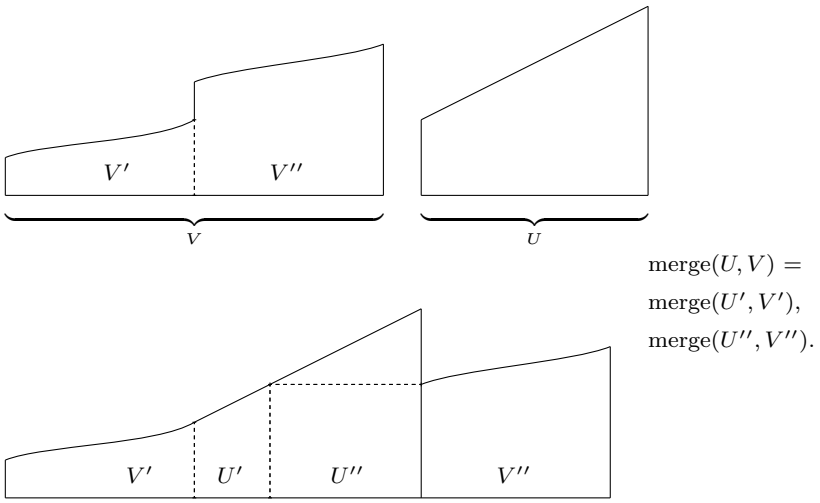


Fig. 11.16 Stable insertion of U into V.

(b) Direct Merging
Direct merging comes in two variants:

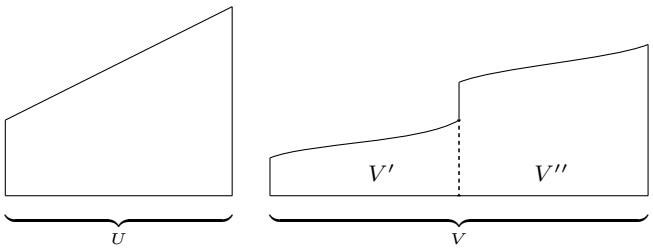
- BLOCK_MERGE_FORWARD(U, V),
- BLOCK_MERGE_BACKWARD(U, V).

Consider here forward merging of blocks (Figure 11.17).

The basic idea for the Trabb-Pardo-Algorithm is a *partition-merge* strategy, where *partition* means *stable segment insertion* and *merge* means *local stable merging of blocks*.

Segment-Insertion of two sequences of blocks

$$U = U_1 \ldots U_i \ldots U_k$$
$$V = V_1 \ldots V_j \ldots V_l$$



BMF$(U, V) \rightarrow$
insert$(U, V) \rightarrow$
BMB/BMF(U'', V'').

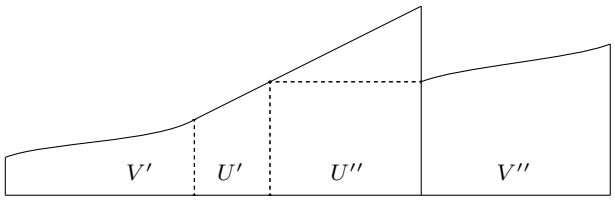


Fig. 11.17 Forward merging of U and V.

with $|U_i| = |V_j| \quad \forall 1 \leq i \leq k,\ 1 \leq j \leq l$ implies a permutation of U and V. The side conditions for the permutation (cf. Figure 11.18) are

$$U_i \text{ is between } V_j \text{ and } V_{j+1} \text{ only if}$$
$$last(V_j) < first(U_i) \leq last(V_{j+1}).$$

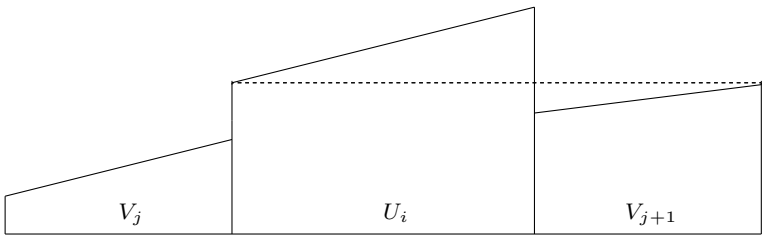


Fig. 11.18 Condition for U_i to rest between V_j and V_{j+1}.

In the example below (Table 11.3), the first row represents the keys from the sequences U and V which are to be merged. The row below gives the (invisible) ordering inside U (lower case letters) and V (upper case letters) for the reader to check stability.
The permutation is s.t. the first record in the last U-block is $\leq$ the last record in the first V-block *and* the last record in the last V-block is strictly $<$ the first record in the first U block. Thus, in the example, we have:

Table 11.3 Example permutation of U and V

1 2	1 1	2 2	2 3	2 3	3 3	4 5	5 5	6 8
a b	A B	c d	e f	C D	E F	g h	G H	i j
U_1	V_1	U_2	U_3	V_2	V_3	U_4	V_4	U_5

$$first(U_1) = 1 \leq last(V_1) = 1 \text{ and } last(V_1) = 1 < first(U_2) = 2,$$
$$first(U_3) = 2 \leq last(V_2) = 3 \text{ and } last(V_3) = 3 < first(U_4) = 4,$$
$$first(U_4) = 4 \leq last(V_4) = 5 \text{ and } last(V_4) = 5 < first(U_5) = 6.$$

11.8 Later Developments

We have already mentioned some later results for the stable in situ merging problem, e.g. Katajainen, Pasanen, and Teuhola [KPT96] from 1996. Another result from around that time for stable, in place merging is by Antonios Symvonis [Sym95] with $O(m \log(\frac{n}{m} + 1))$ comparisons and $O(m + n)$ assignments and a constant amount of extra space. The algorithm evolved from the unstable solution by Mannila and Ukkonen [MU84].

In 2000 this was followed by an asymptotically efficient in-place algorithm by Geffert, Katajainen, and Pasanen [GKP00] and an optimized, stable, in-place merging algorithm by J. Chen [Che03] which claims to improve and simplify the Geffert et al. result.

A more recent contribution is by Pok-Son Kim and Arne Kutzner [KK06] from 2006. Their merging algorithm requires also $O(m \log(\frac{n}{m} + 1))$ comparisons and $O(m + n)$ assignments and claims to be faster in practice than the algorithm by Dudzinski and Dydek [DD81].

Finally, Katajainen and Träff [KT97] did a fairly complete analysis (asymptotic results and actual running time measurements) of merging programs including some competitors like *Quicksort* and *Heapsort* We mention it here because it includes an in-place *Mergesort*.

Chapter 12
Summary and Outlook

12.1 Sorting from a Practical Perspective

From a practical point of view, the Bentley and McIlroy paper *Engineering a sort function* [BM93] touches everything that is relevant in designing a fast, yet robust sorting program. In their case, the result was to be used as a system function for the UNIX operating system replacing the trusted Version 7 `qsort`. They chose again *Quicksort* as the base sorting algorithm for its well-known average speed. Some optimizations and safe-guards against degeneration were known from the literature, in particular from Sedgewicks seminal paper *Implementing Quicksort Programs* [Sed78], others were newly developed, some suggestions omitted.

Interesting enough, a three-way split was chosen which is identical to our Head&Tail-Sort (see Section 4.4.2) and [Weg85] is cited. A new pivot selection strategy was employed, called *pseudomedians* which changes with the size of the array to be partitioned. Insertion sort is used for short subfiles, an efficient `swap` macro helps with larger records, a test $n > 1$ guards recursive calls. A hand-coded stack and elimination of tail recursion are not used, nor is the shorter subfile sorted first and there is no use of a sentinel.

Stability for equal keys was not a design issue and a speed-up for multisets or presorted input was less important than preventing degeneration in these cases. However, in the end a speed-up for multiple occurrences of keys was nevertheless achieved through the three-way split mentioned above.

One returning issue in the previous chapters is that a two-way split with late detection of duplicates is just as fast on multisets as a three-way split with early duplicate detection (or faster if the programming language has no three-way comparisons). This is true for the linked list versions (Linksort vs Trisort) as well as the array versions (Unarysort vs Slidesort et al.). Runsort is another two-way split variation with detection of sorted runs. As mentioned there (Section 4.5), however, partition-exchange sorts are not good in handling presorted input.

On the other hand, the variations in performance of these *Quicksort*-variants are minimal as compared to the observed execution times for *Heapsort* and the Dijkstra invention SMOOTHSORT (cf. Table 4.2). A *Heapsort*-variant, called DEAPSORT, with on-the-fly duplicate deletion was sketched in Section 8.3.6 and it would be a worthy exercise to present a running version.

Heapsort was also identified as a good base sorting method for ordering and duplicate deletion for complex objects like nested relational tables. This is because so-called deep comparisons can be "frozen" and continued later, requiring linear set-up time and delivering a new minimum every $O(\log N)$ steps (cf. Table 10.1). This led to the development of an external *Heapsort* which uses a *heap of pages* and *merging of pages* where the internal sort would use key comparisons.

Heapsort has the most pleasant property of running in asymptotic time $O(N \log N)$ in the average and the worst case. It also uses only constant extra space. The same restriction on extra space is possible for *Quicksort*-variants which do away with the stack. They are discussed in Chapter 7. However, these algorithms are academic exercises and there is no practical reason for not allowing a stack of logarithmic size holding values in binary notation, i.e. $O((\log N)^2)$ bits extra space, called minumum space sorting according to Knuth's Exercise 3 in Section 5.5 [Knu98, p. 390].

Neither *Heapsort* nor *Quicksort* can be turned into a stable sort unless data are given as linked lists. Too many people have looked at *Heapsort* and its stability problem to seriously expect someone to invent a stable variant. With *Quicksort* things aren't as clear. In Section 2.1.3 on page 18 we mentioned one-way scans for partitioning a file (e.g. the Lomuto-algorithm). These one-way partitioning methods often leave the left subfile in original order but mess-up the order of keys when adding them to the "wheel" which is rolled to the right. So in a way, they are semi-stable and it seems that it needs just one more clever idea to make the other half stable as well.

Indeed, there are stable, minimum space, linear time partitioning algorithms, e.g. by Katajainen and Pasanen [KP92], respectively by Munro, Raman, and Salowe [MRS90]. However, they are much like the in place merging algorithms, i.e. rely on tricky blockwise exchanges and encodings. We don't expect any practical use out of them as the constants are too high to make them acceptable in real use. We return to this topic below.

However, *Quicksort* might still be good for one or the other practical surprise. As an example, David R. Martin maintains a web-site on sorting where he discusses three-way quicksorts [Mar07], which in turn contains a link to a Java core library with a new Quicksort variant by Vladimir Yaroslavskiy that uses *two* pivot elements [Yar09]. Let the pivot elements be P_1 and P_2, then we get a three-way split s.t. the first subfile contains all records with key value $< P_1$, the middle file has all keys k s.t. $P_1 \leq k \leq P_2$. Finally, the right subfile has keys $> P_2$. In a personal mail included in that page, Jon Bentley ranks this invention on par with Tony Hoare's original idea for *Quicksort* and Sedgewick's analysis.

Currently, Florian Günther is writing a master's thesis [Gün13] which includes re-implementing in Java some of the algorithms which were presented in this monograph. Furthermore he is measuring their performance under modern work loads. If possible, we will add this variant to the contestants to see whether the expectations are justified.

12.2 Sorting from a Theoretical Perspective

In 1996, Munro and Raman published a paper on fast, stable in-place sorting with $O(n)$ data moves [MR96] as a further refinement on stable in-place sorting. We quote from the *Introduction and Motivation* with our references substituted [MR96, p. 151f].

> The difficulty of designing stable in-place algorithms can be seen from the complexity, increase in the constant factor, and the history of:
>
> 1. The first stable in-place $O(n)$ merging algorithm [Par77] (1977) over the first unstable in-place $O(n)$ merging algorithm [Kro69] (1969), [Knu73b, Exercise 5.2.4. Problem 18].
> 2. The first stable in-place $O(n \lg n)$ sorting algorithm [Par77] (1977) over the first unstable in-place $O(n \lg n)$ sort, Heapsort [Wil64] (1964).
> 3. The first stable in-place algorithm to sort a multiset optimally [KP92] (1992) over the first unstable in-place algorithm that sorts a multiset optimally [MR91] (1991).
> 4. The first stable in-place sorting algorithm performing $O(n)$ data moves [MRS90] (1990) over the first unstable in-place $O(n)$ moves sort, Selection Sort [Knu73b], [Fri56] (1956).
>
> Furthermore, for the first three problems, the asymptotic complexity has been established to be the same for both the unstable and stable versions. In fact, once these problems were well understood, the complexity and the constant factor have been improved significantly [HL88], [HL92][1] for these (at least for the first two) problems.

Further below, Munro and Raman state as still unknown problem whether there is an in-place sort (stable or unstable) that performs $O(n)$ data moves and $O(n \lg n)$ comparisons in the worst case. However, these $O(n)$ data move sorts are only of theoretical interest as they use extreme coding schemes to achieve their constraints. As an example, a sequence of $2m$ distinct keys can be used to store an m-bit number by ordering adjacent keys s.t. $k_i < k_{i+1}$ represents a zero and $k_i > k_{i+1}$ a one. If the sequence was already sorted, the order can be restored afterwards with m comparisons and up to m swaps. Clearly, such techniques are far from any use in practical algorithms, but yet their study might give clues on that missing simple, stable, smooth, in situ *Quicksort*.

[1] Munro and Raman cite the preliminary version with the same title, in: Proceedings of the International Conference on Computing and Information, Toronto, Ontario, Canada, May 1989, pp. 71-80

Appendix A

Common Series Encountered in the Analysis of Algorithms

Geometric series

$$\begin{aligned} S_n &= a + aq + aq^2 + \cdots + aq^{n-1} \\ &= a\frac{1-q^n}{1-q} \end{aligned} \tag{12.1}$$

for $q \neq 1$ and $S_n = na$ for $q = 1$.

A frequent application is the case with $a = 1$ and $q = 1/2$:

$$\sum_{i=1}^{x} \frac{1}{2^i} = \frac{1-\left(\frac{1}{2}\right)^{x+1}}{\frac{1}{2}} - 1 = 1 - \frac{2}{2^{x+1}} \ . \tag{12.2}$$

In particular for $x = \lfloor \log_2 N \rfloor$ we get

$$\sum_{i=1}^{\lfloor \log_2 N \rfloor} \frac{1}{2^i} = 1 - \frac{1}{N} \ . \tag{12.3}$$

When multiplied by N and starting with $i = 0$ we have the common case of $N + N/2 + N/4 + \cdots + 1$ which adds up to $2N - 1$. So if your algorithm – say a partition phase in a variant of *Quicksort* – looks at all N records once and then at half of them again and then at a forth of the N records again

etc. until it reaches the end with only one record, that whole part still has linear complexity.

Interestingly enough we still remain linear if the algorithm looks at all N records once and then at half of them *twice* again and then at a forth of the N records *three* times again etc. until it reaches the end with only one record, which is inspected additionally $x = \lfloor \log_2 N \rfloor$ times. In other words, even with the summation index i also appearing in the nominator, the sum is less than $2N$, resp. less than $3N$, if the sum starts with $i = 0$.

$$
\begin{aligned}
& N \textstyle\sum_{i=1}^{x} \frac{i}{2^i} = \\
& \\
& = N[\left(\tfrac{1}{2}\right)^1 \quad + \left(\tfrac{1}{2}\right)^2 + \ldots + \left(\tfrac{1}{2}\right)^x + \\
& \qquad\qquad \left(\tfrac{1}{2}\right)^2 + \ldots + \left(\tfrac{1}{2}\right)^x + \\
& \qquad\qquad\qquad\qquad \ldots \\
& \qquad\qquad\qquad\qquad + \ \left(\tfrac{1}{2}\right)^x \ ] \\
& \\
& = N \cdot x \cdot \left(\frac{1 - (\frac{1}{2})^{x+1}}{1 - \frac{1}{2}} - 1 \right) - N \sum_{j=1}^{x-1} \left(\left(\frac{1}{2}\right)^1 + \left(\frac{1}{2}\right)^2 + \cdots + \left(\frac{1}{2}\right)^j \right) \\
& \\
& = -Nx + 2Nx - \frac{2Nx}{2^{x+1}} - \left[N \sum_{j=1}^{x-1} \left(\frac{1 - \left(\frac{1}{2}\right)^{j+1}}{1 - \frac{1}{2}} - 1 \right) \right] \\
& \\
& = Nx - \frac{2Nx}{2^{x+1}} + N(x-1) - 2N(x-1) + 2N \left(\frac{1 - (\frac{1}{2})^{x+1}}{1 - \frac{1}{2}} - \frac{1}{2} - 1 \right) \\
& \\
& = Nx + Nx - N - 2Nx + 2N - \frac{2Nx}{2^{x+1}} + 4N - N - 2N - \frac{4N}{2^{x+1}} \\
& = 2N - \frac{2Nx}{2^{x+1}} - \frac{4N}{2^{x+1}} \ . \qquad (12.4)
\end{aligned}
$$

Now set $x = \lfloor \log_2 N \rfloor$

$$N \sum_{i=1}^{\lfloor \log_2 N \rfloor} \frac{i}{2^i} =$$

$$= 2N - \frac{2N\lfloor \log_2 N \rfloor}{2^{\lfloor \log_2 N \rfloor + 1}} - \frac{4N}{2^{\lfloor \log_2 N \rfloor + 1}}$$

$$= 2N - \lfloor \log_2 N \rfloor - 2 \tag{12.5}$$

The Heap-Creation Formula

This useful formula appears in Kurt Mehlhorn's original text book, Vol. 1 *Sorting and Searching* [Meh84, p. 47]. It concerns the creation phase of Heapsort. Think of the heap as a balanced tree of size $N = 2^k - 1$ for $k \geq 1$. On level i there are 2^i nodes ($0 \leq i < k$). Building up the heap, nodes on level i can sink to level $k - 1$ with 2 comparisons per level. The total cost is then limited by

$$\sum_{i=0}^{k-2} 2(k-1-i)2^i = 2^{k+1} - 2(k+1)$$
$$= 2N - 2\lfloor \log N \rfloor \ .$$

The sum is a modification of the general formula for the series

$$\sum_{i=1}^{k} i \cdot 2^i = (k-1)2^{k+1} + 2 \qquad \text{for } k \geq 1 \ . \tag{12.6}$$

The formula appears again in the estimate for the selection phase, where we remove the root of the heap and insert the rightmost element, which then may sink again down to the lowest level i, requiring up to 2 comparisons on each level. This bounds the effort to

$$2\sum_{i=0}^{k-1} i \cdot 2^i = 2(k-2)2^k + 4 \ .$$

Mehlhorn actually proves (12.6) in a fully general form in the appendix of his text book [Meh84, p. 303].

$$
\begin{aligned}
f(x) &= \sum_{i=1}^{k} i \cdot x^i = x \sum_{i=1}^{k} i \cdot x^{i-1} = \\
&= x \sum_{i=1}^{k} \frac{d}{dx} x^i = x \frac{d}{dx} \left(\sum_{i=1}^{k} x^i \right) = x \frac{d}{dx} \left(\sum_{i=0}^{k} x^i - 1 \right) \\
&= x \frac{d}{dx} \left(\frac{x^{k+1} - 1}{x - 1} - 1 \right)
\end{aligned}
$$

In case your calculus is as rusty as mine, only to be topped by an ignorance of combinatorics, remember how to compute the first derivative of a function with a quotient?

$$\left(\frac{u}{v}\right)' = \frac{u'v - uv'}{v^2}$$

So with this little refresher we get

$$\frac{x + x^{k+1} \cdot (k \cdot x - k - 1)}{(x-1)^2}$$

Our bound is then $f(x)$ at $x = 2$, i.e.

$$f(2) = 2 + (k-1)2^{k+1}.$$

Harmonic numbers

This summary is taken from Section 1.2.7. in Knuth's Vol. 1 *The Art of Computer programming* [Knu68], it's Appendix B, 3., and again from the Appendix in Kurt Mehlhorn's text book [Meh84].

The *n'th* harmonic number H_n is defined as

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \qquad \text{for } n \geq 0$$

and increases unbounded, but just barely.

$$H_{2^m} \geq 1 + \frac{m}{2}$$

Fortunately, H_n has a handy estimate because always

$$\ln\ n \leq H_n \leq \ln\ n + 1$$

or, more precisely, using *Euler's Constant* γ

$$H_n = \ln\ n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \epsilon$$

where

$$0 < \epsilon < \frac{1}{252n^6} \text{ and } \gamma = 0.5772156649\ldots\ ,$$

so – among friends – H_n can be sold as $\ln\ n$.

Sometimes you need the sum of harmonic numbers. Very useful is then

$$\sum_{1\le k\le n} H_k = (n+1)H_n - n\ .$$

A table with the first 25 exact values is given in Knuth Vol. 1 [Knu68, Appendix B].

References

[ASSS86] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte, *Min-max heaps and generalized priority queues*, CACM 29 **No. 10** (1986), 996–1000.

[BCK86] Huang Bing-Chao and Donald E. Knuth, *A oneway, stackless quicksort algorithm*, BIT **26** (1986), 127–130.

[Ben84] Jon Louis Bentley, *How to sort*, Commun. ACM **27** (1984), no. 4, 287–291.

[BGM70] Barbara S. Brawn, Frances G. Gustavson, and Efrem S. Mankin, *Sorting in a paging environment*, Commun. ACM **13** (1970), no. 8, 483–494.

[BM93] Jon Luther Bentley and Malcolm Douglas McIlroy, *Engineering a sort function*, Software: Practice and Experience **23 (11)** (1993), 1249–1265.

[Bur76] William H. Burge, *An analysis of binary search trees formed from sequences of nondistinct keys*, J. ACM **23** (1976), no. 3, 451–454.

[Car86] Svante Carlsson, *Splitmerge - a fast stable merging algorithm*, Inf. Process. Lett. **22** (1986), no. 4, 189–192.

[Che03] Jingchao Chen, *Optimizing stable in-place merging*, Theor. Comput. Sci. **302** (2003), no. 1-3, 191–210.

[CK80] C.R. Cook and D.J. Kim, *Best sorting algorithms for nearly sorted lists*, Communications of the ACM 23:11 **23** (1980), 620–624.

[DD81] Krzysztof Dudzinski and Andrzej Dydek, *On a stable minimum storage merging algorithm*, Inf. Process. Lett. **12** (1981), no. 1, 5–8.

[DD88] S. Dvorák and Branislav Durian, *Merging by decomposition revisited*, Comput. J. **31** (1988), no. 6, 553–556.

[Den68] P.J. Denning, *The working set model program behavior*, Communications of the ACM **3:5** (1968), 323–333.

[Den80] ———, *Working sets past and present*, IEEE Transactions on Software Engineering **SE-6:1** (1980), 64–84.

[Dew74] Robert B.K. Dewar, *A stable minimum storage sorting algorithm*, IPL 2 (1974), 162–164.

[Dij82] Edsger W. Dijkstra, *Smoothsort, an alternative for sorting in situ*, Sci. Comput. Program. **1** (1982), no. 3, 223–233.

[DRS85] N. Santoro D. Rotem and J.B. Sidney, *Distributed sorting*, IEEE TSE **C-34, 4** (1985), 372–376.

[Dur86] Branislav Durian, *Quicksort without a stack*, MFCS (Jozef Gruska, Branislav Rovan, and Juraj Wiedermann, eds.), Lecture Notes in Computer Science, vol. 233, Springer, 1986, pp. 283–289.

[DvG82] E.W. Dijkstra and A.J.M. van Gasteren, *An introduction to three algorithms for sorting in situ*, IPL 15 **3** (1982), 129–134.

[Ell85] John A. Ellis, *An in place, linear time, stable partitioning algorithm*, Tech. report, Dept. of Computer Science, University of Victoria, Victoria B.C., Canada V8W 2Y2, 1985.

[EM00] John A. Ellis and Minko Markov, *In situ, stable merging by way of the perfect shuffle*, Comput. J. **43** (2000), no. 1, 40–53.

[Flo62] Robert W. Floyd, *Algorithm 113: Treesort*, Commun. ACM **5** (1962), no. 8, 434.

[Flo64] ———, *Algorithm 245: Treesort3*, Commun. ACM **7** (1964), no. 12, 701.

[Fri56] Edward H. Friend, *Sorting on electronic computer systems*, J. ACM **3** (1956), no. 3, 134–168.

[GKP00] Viliam Geffert, Jyrki Katajainen, and Tomi Pasanen, *Asymptotically efficient in-place merging*, Theor. Comput. Sci. **237** (2000), no. 1-2, 159–181.

[Gon84] Gaston H. Gonnet, *Handbook of algorithms and data structures*, Addison-Wesley, 1984.

[Gün13] Florian Günther, *Sortierverfahren (working title)*, Tech. report, Universität Kassel, Fachbereich Elektrotechnik \Informatik, 2013.

[Her60] R. J. Herbold, *Algorithms: Quadi*, Commun. ACM **3** (1960), no. 2, 74–.

[Her83] Stefan Hertel, *Smoothsort's behaviou an presorted sequences*, IPL 16 (1983), 165–170.

[Hil62] J. S. Hillmore, *Certification of algorithms 63, 64, 65: Partition, quicksort, find*, Commun. ACM **5** (1962), no. 8, 439–.

[HL72] Frank K. Hwang and Shen Lin, *A simple algorithm for merging two disjoint linearly-ordered sets*, SIAM J. Comput. **1** (1972), no. 1, 31–39.

[HL88] Bing-Chao Huang and Michael A. Langston, *Practical in-place merging*, Commun. ACM **31** (1988), no. 3, 348–352.

[HL92] ———, *Fast stable merging and sorting in constant extra space*, Comput. J. **35** (1992), no. 6, 643–650.

[Hoa61] C. A. R. Hoare, *Algorithms 63, 64, 65: partition, quicksort, find*, Commun. ACM **4** (1961), no. 7, 321–322.

[Hoa62] ———, *Quicksort*, The Computer Journal **5:1** (1962), 10–15.

[Hor74] Edward C. Horvath, *Efficient stable sorting with minimal extra space*, STOC (Robert L. Constable, Robert W. Ritchie, Jack W. Carlyle, and Michael A. Harrison, eds.), ACM, 1974, pp. 194–215.

[KK06] Pok-Son Kim and Arne Kutzner, *On optimal and efficient in place merging*, SOFSEM (Jirí Wiedermann, Gerard Tel, Jaroslav Pokorný, Mária Bieliková, and Julius Stuller, eds.), Lecture Notes in Computer Science, vol. 3831, Springer, 2006, pp. 350–359.

[Knu68] Donald E. Knuth, *The art of computer programming, volume i: Fundamental algorithms*, Addison-Wesley, 1968.

[Knu73a] ———, *The art of computer programming, volume i: Fundamental algorithms, 2nd edition*, Addison-Wesley, 1973.

[Knu73b] ———, *The art of computer programming, volume iii: Sorting and searching*, Addison-Wesley, 1973.

[Knu98] ———, *The art of computer programming. vol. 3: Sorting and searching, 2nd ed.*, vol. 3, Addison-Wesley, 1998.

[KP92] Jyrki Katajainen and Tomi Pasanen, *Stable minimum space partitioning in linear time*, BIT **32** (1992), no. 4, 580–585.

[KPT96] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola, *Practical in-place mergesort*, Nord. J. Comput. **3** (1996), no. 1, 27–40.

[Kro69] M. A. Kronrod, *Optimal ordering algorithm without operational field*, Soviet Mathematics - Doklady 10 (1969), 744–.

[KT97] Jyrki Katajainen and Jesper Larsson Träff, *A meticulous analysis of mergesort programs*, CIAC (Gian Carlo Bongiovanni, Daniel P. Bovet, and Giuseppe Di Battista, eds.), Lecture Notes in Computer Science, vol. 1203, Springer, 1997, pp. 217–228.

[Lan13] Michael A. Langston, personal communication, 2013.

[LL97] Anthony LaMarca and Richard E. Ladner, *The influence of caches on the performance of sorting*, Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms (Philadelphia, PA, USA), SODA '97, Society for Industrial and Applied Mathematics, 1997, pp. 370–379.

[Loe74] Rudolf Loeser, *Some performance tests of "quicksort" and descendants*, Commun. ACM **17** (1974), no. 3, 143–152.

[Lor75] Harold Lorin, *Sorting and sort systems*, Addison Wesley, 1975.

[LZ85] Klaus Lagally and Bernhard Ziegler, *Optimized meansort: Omsort*, Tech. Report 3, Institut f"ur Informatik der Universit"at Stuttgart, 1985.

[Man85] H. Mannila, *Measures of presortedness and optimal sorting algorithms*, IEEE TC **C-34** (1985), 318–325.

[Mar07] David R. Martin, *Quick sort (3 way partition)*, 2007, [Online; accessed 28-May-2013].

[Meh79] Kurt Mehlhorn, *Sorting presorted files*, Theoretical Computer Science (Klaus Weihrauch, ed.), Lecture Notes in Computer Science, vol. 67, Springer, 1979, pp. 199–212.

[Meh84] ———, *Data structures and algorithms 1: Sorting and searching*, Monographs in Theoretical Computer Science. An EATCS Series, vol. 1, Springer, 1984.

[Mer85] Susan M. Merrit, *An inverted taxonomy of sorting algorithms*, CACM **28, 1** (1985), 96–99.

[MK84] Dalia Motzkin and John Kapenga, *Technical correspondence: More about meansort*, Commun. ACM **27** (1984), no. 7, 719–722.

[Mon80] M.C. Monard, *Design and analysis of external quicksort algorithms*, Ph.D. thesis, PUC University, Rio de Janeiro - Brazil, 1980.

[Mot81] D. Motzkin, *A stable quicksort*, Softw. Practice and Experience **11** (1981), 607–611.

[Mot83] Dalia Motzkin, *Meansort*, Commun. ACM **26** (1983), no. 4, 250–251.

[MR91] J. Ian Munro and Venkatesh Raman, *Sorting multisets and vectors in-place*, WADS (Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, eds.), Lecture Notes in Computer Science, vol. 519, Springer, 1991, pp. 473–480.

[MR96] ———, *Fast stable in-place sorting with o (n) data moves*, Algorithmica **16** (1996), no. 2, 151–160.

[MRS90] J. Ian Munro, Venkatesh Raman, and Jeffrey S. Salowe, *Stable in situ sorting and minimum data movement*, BIT **30** (1990), no. 2, 220–234.

[MU84] Heikki Mannila and Esko Ukkonen, *A simple linear-time algorithm for in situ merging*, Inf. Process. Lett. **18** (1984), no. 4, 203–208.

[Par77] Luis Trabb Pardo, *Stable sorting and merging with optimal space and time bounds*, SIAM J. Comput. **6** (1977), no. 2, 351–372.

[Pet74] J.G. Peters, *The effects of paging on sorting algorithms*, Tech. report, Dept. of Applied Analysis and Computer Science, University of Waterloo, Ontario, Canada, August 1974.

[Pre75] Franco P. Preparata, *A fast stable sorting algorithm with absolutely minimum storage*, Theor. Comput. Sci. **1** (1975), no. 2, 185–190.

[Riv73] R.L. Rivest, *A fast stable minimum-storage sorting algorithm*, Tech. report, IRIA, 1973.

[Sco65] Roger S. Scowen, *Algorithm 271: quickersort*, Commun. ACM **8** (1965), no. 11, 669–670.

[Sed77a] R. Sedgewick, *Quicksort with equal keys*, SIAM J Comput **6, No. 2** (1977), 240–267.

[Sed77b] Robert Sedgewick, *The analysis of quicksort programs*, Acta Inf. **7** (1977), 327–355.

[Sed78] ———, *Implementing quicksort programs*, Commun. ACM **21** (1978), no. 10, 847–857.

[Sha85] P. Shackelton, *Correspondence on – sorting a random access file in situ*, The Computer Journal **27:3** (1985), 270–275.

[Sin69] Richard C. Singleton, *Algorithm 347: an efficient algorithm for sorting with minimal storage*, Commun. ACM **12** (1969), no. 3, 185–187.

[SW84] Hans-Werner Six and Lutz M. Wegner, *Sorting a random access file in situ*, The Computer Journal **27** (1984), no. 3, 270–275.

[Sym95] Antonios Symvonis, *Optimal stable merging*, Comput. J. **38** (1995), no. 8, 681–690.

[TAH84] H. H. A. Erkioe T.O. Alanka and I. J. Haikala, *Virtual memory behavior of some sorting algorithms*, IEEE TSE **SE 10 No. 4** (1984), 422–431.

[TO12] Peter Widmayer Thomas Ottmann, *Algorithmen und datenstrukturen*, Spektrum Akademischer Verlag, 5. Aufl. 2012.

[TW91] Jukka Teuhola and Lutz Michael Wegner, *Minimal space, average linear time duplicate deletion*, Commun. ACM **34** (1991), no. 3, 62–73.

[vE70a] Maarten H. van Emden, *Algorithms 402: Increasing the efficiency of quicksort*, Commun. ACM **13** (1970), no. 11, 693–694.

[vE70b] ———, *Increasing the efficiency of quicksort*, Commun. ACM **13** (1970), no. 9, 563–567.

[Ver86] A. Inkeri Verkamo, *Comparison of external sorting and internal sorting in virtual memory*, Perform. Eval. **6** (1986), no. 2, 135–145.

[Ver87] ———, *Performance of quicksort adapted for virtual memory use*, Comput. J. **30** (1987), no. 4, 362–371.

[Ver88] ———, *External quicksort*, Perform. Eval. **8** (1988), no. 4, 271–288.

[Ver89] ———, *Performance comparison of distributive and mergesort as external sorting algorithms*, Journal of Systems and Software **10** (1989), no. 3, 187–200.

[Wai85] R.L. Wainwright, *A class of sorting algorithms based on quicksort*, Communications of the ACM **28:4** (1985), 396–402.

[Wal77] H. Waldschmidt, *Sortieren bei virtuellem speicher*, Computing **19** (1977), 1–14.

[Weg82] Lutz Wegner, *Sorting a linkes list with equal keys*, IPL **15,5** (1982), 205–209.

[Weg83] ———, *The linksort family-design and analysis of fast, stable quicksort derivatives*, Tech. Report Report 123, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, 1983.

[Weg84] ———, *Sorting a distributed file in a network*, Computer Networks **8** (1984), 451–461.

[Weg85] ———, *Quicksort for equal keys*, IEEE TC **C-34 No. 4** (1985), 362–367.

[Weg87] Lutz Michael Wegner, *A generalized, one-way, stackless quicksort*, BIT **27** (1987), no. 1, 44–48.

[Wik13a] Wikipedia, *Elliott 803 — wikipedia, the free encyclopedia*, 2013, [Online; accessed 17-January-2013].

[Wik13b] ———, *In-place algorithm — wikipedia, the free encyclopedia*, 2013, [Online; accessed 24-January-2013].

[Wil64] J.W.J. Williams, *Algorithm 232 (heapsort)*, CACM **7** (1964), 347–348.

[Won81] Jin Kiu Wong, *Some simple in-place merging algorithms*, BIT **21** (1981), no. 2, 157–166.

[WT89] Lutz Michael Wegner and Jukka Teuhola, *The external heapsort*, IEEE Trans. Software Eng. **15** (1989), no. 7, 917–925.

[Yar09] Vladimir Yaroslavskiy, *Replacement of quicksort in java.util.arrays with new dual-pivot quicksort*, 2009, [Online; accessed 28-May-2013].

Index

Turku Centre for Computer Science
TUCS Lecture Notes

1. **Ralph-Johan Back och Joakim von Wright**, Matematik med lite logik: Strukturerade härledningar I gymnasiematematiken
2. **Ralph-Johan Back och Joakim von Wright**, Matematik med lite logik: En kort kurs i talteori
3. **Ralph-Johan Back och Joakim von Wright**, Matematik med lite logik: Studentexamen i lång matematik, våren 2003
4. **Ralph-Johan Back ja Joakim von Wright**, Matematiikkaa logiikan avulla: Rakenteiset päättelyketjut lukiomatematiikassa
5. **Ralph-Johan Back ja Joakim von Wright**, Matematiikkaa logiikan avulla: Lyhyt lukuteorian kurssi
6. **Ralph-Johan Back ja Joakim von Wright**, Matematiikkaa logiikan avulla: Pitkän matematiikan ylioppilaskoe, kevät 2003
7. **Ralph-Johan Back och Joakim von Wright**, Matematik med lite logik: Introduktion till strukturerade härledningar
8. **Ralph-Johan Back och Joakim von Wright**, Matematik med lite logik: Logik för strukturerade härledningar
9. **Ralph-Johan Back och Joakim von Wright**, Matematik med lite logik: Strukturerade härledningar som allmänt bevisformat
10. **Ralph-Johan Back ja Joakim von Wright**, Matematiikkaa logiikan avulla: Johdatus rakenteisiin päättelyketjuihin
11. **Ralph-Johan Back ja Joakim von Wright**, Matematiikkaa logiikan avulla: Logiikka ja rakenteiset päättelyketjut
12. **Ralph-Johan Back ja Joakim von Wright**, Matematiikkaa logiikan avulla: Rakenteiset päättelyketjut yleisenä todistusmuotona
13. **Jarkko Kari (Editor)**, Proceedings of JAC 2010 – journées Automates Cellulaires
14. **Mike Stannet, Danuta Makowiec, Anna T. Lawniczak and Bruno N. Di Stefano**, Proceedings of the Satellite Workshops of UC 2011
15. **Timo Leino (Editor)**, Proceedings of the IRIS 2011 Conference
16. **Hongxiu Li (Editor)**, Studies on Inequalities in Information Society – Proceedings of the Conference, Well-Being in the Information Society. WIS 2012
17. **Vesa Halava, Juhani Karhumäki and Yuri Matiyasevich (Editors)**, RuFiDiM II, Proceedings of the Second Russian Finnish Symposium on Discrete Mathematics 2012
18. **Michael Butler, Stefan Hallerstede and Marina Waldén (Editors)**, Proceedings of the 4th Rodin User and Development Workshop
19. **Hongxiu Li and Jonna Järveläinen (Editors)**, Effective, Agile and Trusted eServices Co-Creation – Proceedings of the 15th International Conference on Electronic Commerce ICEC 2013
20. **Juhani Karhumäki, Markus Whiteland and Luca Zamboni (Editors)**, Local Proceedings of WORDS 2013
21. **Juha-Pekka Soininen, Sergey Balandin, Johan Lilius, Petri Liuha and Tullio Salmon Cinotti (Editors)**, Proceedings of the Open International M3 Semantic Interoperability Workshop
22. **Lutz M. Wegner**, Sorting – The Turku Lectures

Turku Centre *for* Computer Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www. tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Division for Natural Sciences and Technology
- Department of Information Technologies

ISBN 978-952-12-3020-2
ISSN 1797-8831

Lutz M. Wegner

Sorting – The Turku Lectures

Lutz M. Wegner

Sorting – The Turku Lectures

Lutz M. Wegner

Sorting – The Turku Lectures