Introduction

Sorting is the computational process of rearranging a given sequence of items from some total order into ascending or descending order. Because sorting is a task in the very core of Computer Science, efficient algorithms were developed early. The first practical and industrial applications of computers had many uses for sorting. It is still a very frequent occurring problem, often appearing as a preliminary step to some other computational task. A related application to sorting is computing *order statistics*, for example, finding the median, the smallest or the largest of a set of items. Although finding order statistics is immediate once the items are sorted, sorting can be avoided and faster algorithms have been designed for the k-th largest element, the most practical of which is derived from the structure of a sorting method. Repeated queries for order statistics may be better served by sorting and using an efficient data structure that implements the abstract data type dictionary, with the ranking as the key.

Sorting usually involves data consisting of records in one or several files. One or several fields of the records are used as the criteria for sorting (often a small part of the record) and are called the *key*. Usually, the objective of the sorting method is to rearrange the records so that the keys are arranged in numerical or alphabetical order. However, many times, the actual rearrangement of all the data records is not necessary, but just the logical reorder by manipulating the keys is sufficient.

In many applications of sorting, elementary sorting algorithms are the best alternative. One has to admit that in the era of the Internet and the World Wide Web, network lag is far more noticeable that almost any sorting on small data sets. Moreover, sorting programs are often used once (or only a few times) rather than repeated many times, one after another. Simple methods are always suitable for small files, say less than 100 elements. The increasing speeds in every time less expensive computers are enlarging the size for which basic methods are adequate. More advanced algorithms require more careful programming, and their correctness or efficiency is more fragile to thorough understanding of their mechanisms. Also, sophisticated methods may not take advantage of existing order in the input, which may already be sorted, while elementary methods usually do. Finally, elementary sorting algorithms usually have a very desirable property, named *stability*; that is, they preserve the relative order of items with equal keys. This is usually expected in applications generating reports from already sorted files, but with a different key. For example, long distance phone calls are usually recorded in a log in chronological order by date and time of the call. When reporting bills to customers, the carrier sorts by customer name, but the result should preserve the chronological order of the calls made bye each particular customer.

Advanced methods are the choice in applications involving a large number of items. Also, they can be used to build robust, general-purpose sorting routines [5, 7, 6, 11]. Elementary methods should not be used for large, randomly permuted files. An illustrative trade-off between a sophisticated technique that results in general better performance over the simplicity of the programming is the sorting of keys and pointers to records rather than the entire records that we mentioned earlier. Once the keys are sorted, the pointers to the complete records can be used in a pass over the file to rearrange the data in the desired order. It is usually more laborious to apply this technique because we must construct the auxiliary sequence of keys and pointers (or in the case of long keys, the keys can be kept with the records as well). However, many exchanges and data moves are saved until the final destination of each record is known. In addition, less space is required if, as in common in practice, keys are only a small part of the record. When the keys are a quarter or more of alphanumeric records, this technique is not worth the effort. However, as records contain multi-media data or other large blobs of information, the sorting of the keys and pointers is becoming best practice.

In recent years, the study of sorting has been receiving attention [25, 26, 31, 32, 33] because computers offer now a sophisticated memory hierarchy, with CPUs having large space in a cache faster than random access memory. When the file to be sorted is small enough that all the data fits in random access memory (main memory) as an array of records or keys, the sorting process is called **internal sorting**. Today, memory sizes are very large, and the situation when the file to be sorted is so large that the data does not entirely fit in main memory seems a matter for the past. However, the fact of the matter is that computers have more stratified storage capacity that trades cost for speed and size. Even at the level of micro instruction, there are caches to anticipate code, and as we mentioned, CPUs today manage caches as well. Operating systems are notorious for using virtual memory and storage devices have increased capacity in many formats for magnetic disks and DVDs. While sequential access in tapes does seem a thing of the past, there is now the issue of the data residing on servers over a computer network. Perhaps the most important point is that algorithms that take advantage of locality of reference, both, for the instructions as well as for the data are better positioned to perform well.

Historically, **external sorting** remains the common terms for sorting files that are to large to be held in main memory and the rearrangement of records is to happen on disk drives. The availability of large main memories has modified how buffers are allocated to optimize the input/output costs between the records in

main memory add the architectures of disks drives [22, 34].

Sorting algorithms can also be classified into two large groups according to what they require about the data to perform the sorting. The first group is called *comparison-based*. Methods of this class only use the fact that the universe of keys is linearly ordered. Because of this property, the implementation of *comparison-based* algorithms can be generic with respect to the data type of the keys, and a comparison routine can be supplied as a parameter to the sorting procedure. The second group of algorithms assumes further that keys are restricted to a certain domain or data representation, and use knowledge of this information to dissect subparts, bytes or bits of the keys.

Sorting is also ideal for introducing issues regarding algorithmic complexity. For comparison-based algorithms, it is possible to precisely define an abstract model of computation (namely, *decision trees*) and show lower bounds on the number of comparisons any sorting method in this family would require to sort a sequence with n items (in the worst case and in the average case). A comparison-based sorting algorithm that requires $O(n \log n)$ comparisons is said to be optimal, because it matches the $\Omega(n \log n)$ lower bound. Thus, in theory, no other algorithm could be faster. The fact that algorithms that are not comparison-based can result in faster implementations illustrates the relevance of the model of computation with respect to theoretical claims of optimality and the effect that stronger assumptions on the data have for designing a faster algorithm [1, 24].

Sorting illustrates *randomization* (the fact that the algorithm can make a random choice). From the theoretical angle, randomization provides a more powerful machine that has available random bits. In practice, a pseudo-random generator is sufficient and randomization delivers practical value for sorting algorithms. In particular, it provides an easy protection for sophisticated algorithms from special input that may be simple (like almost sorted) but harmful to the efficiency of the method. The most notable example is the use of randomization for the selection of the pivot in *Quicksort*.

In what follows, we will assume that the goal is to sort into ascending order, since sorting in descending order is symmetrical, or can be achieved by sorting in a ascending order and reversing the result (in linear time, which is usually affordable). Also, we make no further distinction between the records to be sorted and their keys, assuming that some provision has been made for handling this as suggested before. Through our presentation the keys may not all be different, since one of the main applications of sorting is to bring together records with matching keys. We will present algorithms in pseudo-code in the style introduced by Cormen et al [9], or when more specific detail is convenient, we will use PASCAL code. When appropriate, we will indicate possible trade-offs between clarity and efficiency of the code. We believe that efficiency should not be pursued at the extreme, and certainly not above clarity. The costs of programming and code maintenance are usually larger than the slight efficiency gains of tricky coding. For example, there is a conceptually simple remedy to make every sorting routine **stable**. The idea is to precede it with the construction of new keys and sort according to lexicographical order the new keys. The new key for the i-th item is the pair (k_i, i), where k_i is the original sorting key. This requires the extra management of the composed keys and adds to the programming effort the risk of a faulty implementation. Today, this could probably be solved by simply choosing a competitive stable sort, perhaps at the expense of slightly more main memory or CPU time.

Underlying Principles

Divide-and-conquer is a natural, top-down approach for the design of an algorithm for the abstract problem of sorting a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ of n items. It consists of dividing the problem into smaller subproblems hoping that the solution of the subproblems are easier to find and then composing the partial solutions into the solution of the original problem. A prototype of this idea is *Mergesort*; where the input sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ is split into two sequences $X_L = \langle x_1, x_2, \ldots, x_{\lfloor n/2 \rfloor} \rangle$ (the left subsequence) and $X_R = \langle x_{\lfloor n/2 \rfloor + 1}, \ldots, x_n \rangle$ (the right subsequence). Finding solutions recursively for sequences with more than one item and terminating the recursion with sequences of only one item (since these are always sorted) provides a solution for the two subproblems. The overall solution is found by describing a method to merge two sorted sequences. In fact, internal sorting algorithms are variations of two forms of using divide-and-conquer:

Conquer form: Divide is simple (usually requiring constant time), conquer is sophisticated.

Divide form: Divide is sophisticated, conquer is simple (usually requiring constant time).

Again, *Mergesort* is an illustration of the conquer from. The core of the method is the merging of the two sorted sequences, hence its name.

The prototype of the **divide form** is *Quicksort* [19]. Here, one item x_p (called the *pivot*) is selected from the input sequence X and its key is used to create two subproblems $X_{\leq}$ and $X_{\geq}$, where $X_{\leq}$ contains items in X with keys less or equal than the pivot's, while items in $X_{\geq}$ have items larger or equal than the pivot's. Now,

	Internal Sorting Algorithms	
	Conquer form	Divide form
comparison based	*Mergesort* *Insertion Sort*	*Quicksort* *Selection Sort* *Heapsort* *Shellsort*
restricted universe		*Bucket Sort* *Radix Sort*

Figure 1: The landscape of internal sorting algorithms.

recursively applying the algorithm for $X_{\leq}$ and $X_{\geq}$ results in a global solution (with a trivial conquer step that places $X_{\leq}$ before $X_{\geq}$).

Sometimes we may want to conceptually simplify the *conquer form* by not dividing into two subproblems of roughly the same size, but rather divide X into $X' = \langle x_1, \ldots, x_{n-1} \rangle$ and $X'' = \langle x_n \rangle$. This is conceptually simpler in two ways. First, X'' is a trivial subproblem, because it has only one item and thus it is already sorted. Therefore, in a sense we have one subproblem less. Second, the merging of the solutions of X' and X'' is simpler than the merging of two sequences of almost equal length, because we just need to place x_n in its proper position amongst the sorted items from X'. Because the method is based upon inserting into an already sorted sequence, sorting algorithms with this idea are called **insertion sorts**.

Insertion sorts vary according to how the sorted list for the solution of X' is represented by a data structure that supports insertions. The simplest alternative is to have the sorted list stored in an array, and this method is named *Insertion Sort* or *Straight Insertion Sort* [21]. However, if the sorted list is represented by a *level-linked-tree* with a *finger*, the methods has been named *A-sort* [24] or *Local Insertion Sort* [23].

The complementary simplification in the divide form makes one subproblem trivial by selecting a pivot so $X_{\leq}$ or $X_{\geq}$ consist of just one item. This is achieved if we select the pivot as the item with the smallest or with the largest key in the input. The algorithms under this scheme are called *selection sorts*, and they vary according to the data structure used to represent X so that the repeated extraction of the maximum (or minimum) key is efficient. This is typically the requirement of the *priority queue* abstract data type with the keys as the priorities. When the priority queue is implemented as an array and the smallest key is found by scanning this array, the method is *Selection Sort*. However, if the priority queue is an array organized into a heap, the method is called *Heapsort*.

Using divide-and-conquer does not necessarily mean that the division must be into two subproblems. It may divide into several subproblems. For example, if the keys can be manipulated with other operations besides comparisons, *Bucket Sort* uses an interpolation formula on the keys to partition the items between m buckets. The buckets are sets of items, which are usually implemented as *queues*. The queues are usually implemented as linked lists that allow insertion and removal in constant time in first-in first-out order, as the abstract data type queue requires. The buckets represent subproblems to be sorted recursively. Finally, all the buckets are concatenated together. *Shellsort* divides the problem of sorting X into several subproblems consisting of interlaced subsequences of X that consist of items d positions apart. Thus, the first subproblem is $\langle x_1, x_{1+d}, x_{1+2d}, \ldots \rangle$ while the second subproblem is $\langle x_2, x_{2+d}, x_{2+2d}, \ldots \rangle$. *Shellsort* solves the subproblems by applying *Insertion Sort*; however, rather than using a **multiway merge** to combine these solutions, it reapplies itself to the whole input with a smaller value d. Careful selection of the sequence of values of d results in a practical sorting method.

We have used divide-and-conquer to conceptually depict the landscape of sorting algorithms (refer to Fig 1). Nevertheless, in practice, sorting algorithms are usually not implemented as recursive programs. Instead, a non-recursive equivalent analog is implemented (although computations may be performed in different order). In applications, the non-recursive version is more efficient since the administration of the recursion is avoided. For example, a straightforward non-recursive version of *Mergesort* proceeds as follows. First, the pairs of lists $\langle x_{2i-1} \rangle$ and $\langle x_{2i} \rangle$ (for $i = 1, \ldots, \lfloor n/2 \rfloor$) are merged to form sorted lists of length two. Next, the pairs of lists $\langle x_{4i-1}, x_{4i-2} \rangle$ and $\langle x_{4i-3}, x_{4i} \rangle$ (for $i = 1, \ldots, \lfloor n/4 \rfloor$) are merged to form lists of length four. The process builds sorted lists twice as long in each round until the input is sorted. Similarly, *Insertion Sort* has a practical iterative version illustrated in Fig 2.

Placing a sorting method in the landscape provided by divide-and-conquer allows easy computation of its time requirements, at least under the O notation. For example, algorithms of the conquer form have a divide part that takes $O(1)$ time to derive two subproblems of roughly equal size. The solutions of the subproblems

INSERTION SORT (X, n);

```
1: X[0] ← ∞;
2: for j ← 2 to n do
3:    i ← j − 1;
4:    t ← X[j];
5:    while t < X[i] do
6:       X[i + 1] ← X[i];
7:       i ← i − 1;
8:    end while
9:    X[i + 1] ← t;
10: end for
```

Figure 2: The sentinel version of *Insertion Sort*.

may be combined in $O(n)$ time. This results in the following recurrence for the time $T(n)$ to solve a problem of size n:

$$T(n) = \begin{cases} 2T(n)(\lceil n/2 \rceil) + O(n) + O(1) & \text{if } n > 1, \\ O(1) & n = 1. \end{cases}$$

It is not hard to see that each level of recursion takes linear time and that there are at most $O(\log n)$ levels (since n is roughly divided by 2 at each level). This results in $T(n) = O(n \log n)$ time overall. If the divide form splits into one problem of size $n - 1$ and one trivial problem, the recurrence is as follows:

$$T(n) = \begin{cases} T(n-1) + O(1) + O(\texttt{conquer}(n-1)) & \text{if } n > 1, \\ O(1) & n = 1, \end{cases}$$

where $\texttt{conquer}(n-1)$ is the time required for the conquer step. In is not hard to see that there are $O(n)$ levels of recursion (since n is decremented by one at each level). Thus, the solution to the recurrence is $T(n) = O(1) + \sum_{i=1}^{n} O(\texttt{conquer}(i))$. In the case of *Insertion Sort*, the worst case for $\texttt{conquer}(i)$ is $O(i)$, for $i = 1, \ldots, n$. Thus, we have that *Insertion Sort* is $O(n^2)$. However, *Local Insertion Sort* assures $\texttt{conquer}(i) = O(\log i)$ time and the result is an algorithm that requires $O(n \log n)$ time. We will not pursue this analysis any further, confident that the reader will be able to find enough information here or in the references to identify the time and space complexity of the algorithms presented, at least up to the O notation.

Naturally, one may ask why there are so many sorting algorithms if they all solve the same problem and fit a general framework. It turns out that, when implemented, each has different properties that makes them more suitable for different objectives.

First, comparison-based sorting algorithms are ranked by their theoretical performance in the comparison-based model of computation. Thus, an $O(n \log n)$ algorithm should always be preferred over an $O(n^2)$ algorithm if th files are large. However, theoretical bounds may be for the worst case, or the expected case (where the analysis assumes that the keys are pairwise different and all possible permutations of items are equally likely). Thus, an $O(n^2)$ algorithm should be preferred over an $O(n \log n)$ algorithm if the file is small, or if we know that the file is already almost sorted. Particularly, if in such a case, the $O(n^2)$ algorithm turn into an $O(n)$ algorithm. For example, *Insertion Sort* requires exactly $Inv(x) + n - 1$ comparisons and $Inv(X) + 2n - 1$ data moves, where $Inv(X)$ is the number of *inversions* in a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$; that is, the number of pairs (i, j) where $i < j$ and $x_i > x_j$. On the other hand, if *Quicksort* is not carefully implemented, it may degenerate to $\Omega(n^2)$ performance on nearly sorted inputs.

If the theoretical complexities are equivalent, other aspects come into play. Naturally, the next criteria is the size of the constant hidden under the O notation (as well as the size of hidden minor terms when the file is small). These constants are affected by implementation aspects. The most significant are now listed.

- The relative costs of swaps, comparisons, and all other operations in the computer at hand (and for the data types of the keys and records). Usually, swaps are more costly than comparisons, which in turn are more costly than arithmetic operations; however, comparisons may be just as costly as swaps or an order of magnitude less costly, depending on the length of the keys, records and strategies to rearrange the records.

- The length of the machine code, so code remains in memory, under an operating system that administers paging or in the cache of the microprocessor.

- Similarly, the locality of references to data or the capacity to place frequently compared keys in a CPU register.

Finally, there may be restrictions that force the choice of one sorting method over another. These include limitations like data structures holding the data may be a linked list instead of an array, or the space available may be seriously restricted. There may be a need for stability, or the programming tool may lack recursion (now, this is unusual). In practice, a hybrid sort is usually the best answer.

State of the Art and Best Practices

Comparison-Based Internal Sorting

Insertion Sort

Fig. 2 presented *Insertion Sort*. This algorithm uses sequential search to find the location, one item at a time, in a portion already sorted of the input array. It is mainly used to sort small arrays.

Besides being one of the simplest sorting algorithms, which results in simple code, it has many desirable properties. From the programming point of view, its loop is very short (usually taking advantage of memory management in the CPU cache or main memory), the key of the inserted element may be placed in a CPU register and access to data exhibits as much locality as it is perhaps possible. Also, if a minimum possible key value is known, a *sentinel* can be placed at the beginning of the array to simplify the inner loop, resulting in faster execution. Another alternative is to place the item being inserted, itself as a sentinel each time. From the applicability point of view, it is stable; recall this means records with equal keys remain in the same relative order after the sort. Its $\Theta(n^2)$ expected-case complexity and worst-case behavior makes it only suitable for small files, and thus, it is usually applied in *Shellsort* to sort the interleaved sequences. However, the fact that it is **adaptive** with respect to the measure of disorder Inv makes it suitable for almost sorted files with respect to this measure. Thus, it is commonly used to sorted roughly sorted data produced by implementations of *Quicksort* that do not follow recursion calls once the sub-array is small. This idea helps *Quicksort* implementations achieve better performance, since the administration of recursive calls for small files is more time consuming that one call that uses *Insertion Sort* on the entire file to complete the sorting.

Insertion Sort also requires only constant space; that is, space for a few local variables (refer to Fig. 2). From the point of view if quadratic sorting algorithms, it is a clear winner. Investigations of theoretical interest have looked at comparison-based algorithms where space requirements are constant, and data moves are linear. Only in this case, *Insertion Sort* is not the answer, since *Selection Sort* achieves this with equivalent number of comparisons. Thus, when records are very large and no provision is taken for avoiding expensive data moves (by sorting a set of indices rather than the data directly), *Selection Sort* should be used.

Shellsort

One idea for improving the performance of *Insertion Sort* is to observe that each element, when inserted into the sorted portion of the array, travels a distance equal to the number of elements to its left which are greater than itself (the number of elements inverted with it). However, this traveling is done in steps of just adjacent elements and not by exchanges between elements far apart. The idea behind *Shellsort* [21] is to use *Insertion Sort* to sort interleaved sequences formed by items d positions apart, thus allowing for exchanges as far as d positions apart. After this, elements far apart are closer to their final destinations, so d is reduced to allow exchanges of closer positions. To ensure the output is sorted, the final value d is one.

There are many proposals for the *increment sequence* [16]; the sequence of values d. Some proposals are $\langle 2^k - 1, 2^{k-1} - 1, \ldots, 7, 3, 1\rangle$, $\langle 2^p 3^q, \ldots, 9, 8, 6, 4, 3, 2, 1\rangle$, and $S_i = \langle \ldots, 40, 13, 4, 1\rangle$ where $S_i = 3S_{i-1} + 1$. It is possible that a better sequence exists; however, the improvement that they may produce in practice is almost not visible. *Shellsort* is guaranteed to be clearly below the quadratic behavior of *Insertion Sort*. The exact theoretical complexity remains elusive but large experiments conjecture $O(n(\log n)^2)$, $O(n^{1.5})$, and $O(n \log n \log\log n)$ comparisons are required for various increment sequences. Thus, it will certainly be much faster than quadratic algorithms, and for medium-size files it would remain competitive with $O(n \log n)$ algorithms.

From the programming point of view, *Shellsort* is simple to program. It is *Insertion Sort* inside the loop for the increment sequence. It is important not to use a version of *Insertion Sort* with sentinels, since for the rounds larger than d many sentinels would be required. Not using a sentinel demands two exit points for the most inner loop. This can be handled with a clean use of a **goto**, but in languages which short-cut connectives (typically C, C++ and so on) this feature can be used to avoid goto-s. Fig 3 shows pseudocode for *Shellsort*.

SHELLSORT (X, n);

```
 1: d ← n;
 2: repeat
 3:    if d < 5 then
 4:       d ← 1
 5:    else
 6:       d ← (5 * d − 1) div 11;
 7:    end if
 8:    for j ← d + 1 to n  do
 9:       i ← j − d;
10:       t ← X[j];
11:       while t < X[i] do
12:          X[i + d] ← X[i];
13:          i ← i − d;
14:          if  i < d then
15:             goto 18
16:          end if
17:       end while
18:       X[i + d] ← t;
19:    end for
20: until  d ≤ 1;
```

Figure 3: *Shellsort* with increment sequence $\langle \lfloor n\alpha \rfloor, \lfloor \lfloor n\alpha \rfloor \alpha \rfloor, \ldots, \rangle$ with $\alpha = 0.4545 < 5/11$.

Unfortunately, *Shellsort* loses some of the virtues of *Insertion Sort*. It is no longer stable, and its behavior in nearly sorted files is adaptive but not as marked as for *Insertion Sort*. However, the space requirements remain constant, and its coding is straight forward and usually results in a short program loop. It does not have a bad case and it is a good candidate for a library sorting routine. The usual recommendation when facing a sorting problem is to first try *Shellsort* because a correct implementation is easy to achieve. Only if it proves to be insufficient for the application at hand should a more sophisticated method be attempted.

Heapsort

The *priority queue* abstract data type is an object that allows the storage of items with a key indicating their priority as well as retrieval of the item with the largest key. Given a data structure for this abstract data type, a sorting method can be constructed as follows. Insert each data item in the priority queue with the sorting key as the priority key. Repeatedly extract the item with the largest key from the priority queue to obtain the items sorted in reverse order.

One immediate implementation of the priority queue is an unsorted list (either as an array or as a linked list). Insertion in the priority queue is trivial; the item is just appended to the list. Extraction of the item with the largest key is achieved by scanning the list to find the largest item. If this implementation of the priority queue is used for sorting, the algorithms is called *Selection Sort*. Its time complexity is $\Theta(n^2)$, and as was already mentioned, its main virtue is that data moves are minimal.

A second implementation of a priority queue is to keep a list of th items sorted in descending order by their priorities. Now, extraction of the item with the largest priority requires constant time, since it is known that the largest item is at the front of the list, However, inserting a new item into the priority queue implies scanning the sorted list for the position of the new item. Using this implementation of a priority queue for sorting we observe that we obtain *Insertion Sort* once more.

The above implementations of a queue offer constant time either for insertion or extraction of the item with maximum key, but in exchange for linear time for the other operation. Thus, priority queues implemented like this may result in efficient methods in applications of priority queues where the balance of operations is uneven. However, for sorting, n items are inserted and also n items are extracted; thus, a balance is required between the insert and extract operations. This is achieved by implementing a priority queue as a *heap* [1, 21] that shares the space with the array of data to be sorted. An array $A[1 \ldots n]$ satisfies the heap property if $A[\lfloor k/2 \rfloor] \geq A[k]$, for $2 \leq k \leq n$. In this case, $A[\lfloor k/2 \rfloor]$ is called the *parent* of $A[k]$ while $A[2k]$ and $A[2k+1]$ are called the *children* of $A[k]$. However, an item may have no children or only one child, in which case its called a

HEAPSORT (X, n);

```
1: for i ← (n div 2) down-to 2 do
2:    SINK(i,n)
3: end for
4: for i ← n down-to 2 do
5:    SINK(1,i)
6:    t ← X [1];
7:    X[1] ← X[i];
8:    X[i] ← t;
9: end for
```

Figure 4: The pseudocode for *Heapsort*.

SINK $(k, limit)$;

```
1: while 2 * k ≤ limit do
2:    j ← 2k;
3:    if j < limit then {two children}
4:       if X[j] < X[j + 1] then
5:          j ← j + 1;
6:       end if
7:    end if
8:    if X[k] < X[j] then
9:       t ← X[j];
10:      X[j] ← X[k];
11:      X[k] ← t;
12:      k ← j
13:   else
14:      k ← limit + 1 {force loop exit}
15:   end if
16: end while
```

Figure 5: The sinking of one item into a heap.

leaf. The heap is constructed using all the elements in the array and thereafter is located in the lower part of the array. The sorted array is incrementally constructed from the item with the largest key until the element with the smallest key in the high part of the array. The first phase builds the heap using all the elements, and careful programming guarantees this requires $O(n)$ time. The second phase repeatedly extracts the item with largest key from the heap. Since the heap shrinks by one element, the space created is used to place the element just extracted. Each of the n updates in the heap takes $O(\log i)$ comparisons, where i is the number of items currently in the heap. In fact, the second phase of *Heapsort* exchanges the first item of the array (the item with largest key) with the item in the last position of the heap, and sinks the new item at the top of the heap to reestablish the heap property.

Heapsort can be efficiently implemented around a procedure SINK for repairing the heap property (refer to Fig. 4). Procedure SINK(k,$limit$) moves down the heap, if necessary, exchanging the item at position k with the largest of its two children, and stopping when the item at position k is no longer smaller that one of its children (or when $k > limit$); refer to Fig. 5. Observe that the loop in SINK has two distinct exits, when item k has no children and when the heap property is reestablished. For our pseudocode, we have decided to avoid the use of goto-s. However, the reader can refer to our use of goto in Fig. 3 for an idea to construct an implementation that actually saves some data moves. Using the procedure SINK, the code for *Heapsort* is simple. From the applicability point of view, *Heapsort* has the disadvantage that is not stable. However, it is guaranteed to execute in $O(n \log n)$ time in the worst case and requires no extra space.

It is worth revising the analysis of *Heapsort*. First, let us look at SINK. In each pass around its loop, SINK at least doubles the value of k, and SINK terminates when k reaches $limit$ (or before, if the heap property is reestablished earlier). Thus, in the worst case SINK requires $O(h)$ where h is the height of the heap. The loop

in line 1 and line 2 for *Heapsort* constitute the core of the first phase (in our code, the heap construction is completed after the first execution of line 4). A call to SINK is made for each node. The first $\lfloor n/2 \rfloor$ calls to SINK are for heaps of height 1, the next $\lfloor n/4 \rfloor$ are for heaps of height 2, and so on. Summing over the heights, we have that the phase for building the heap requires

$$O\left(\sum_{i=1}^{\log n} i\frac{n}{2^i}\right) = O(n)$$

time. Now, the core of the second phase is the loop from line 3. These are n calls to SINK plus a constant for the three assignments. The i-th of the SINK calls is in a heap of height $O(\log i)$. Thus, this is $O(n \log n)$ time. We conclude that the first phase of *Heapsort* (building the priority queue) requires $O(n)$ time. This is useful when building a priority queue. The second phase, and thus, the algorithm, requires $O(n \log n)$ time.

Heapsort does not use any extra storage, nor does it require a language supplying recursion (recursion is now common but few programmers are aware of the memory requirements of a stack for function calls). For some, it may be surprising that *Heapsort* destroys the order in an already sorted array to re-sort it. Thus, *Heapsort* does not take advantage of existing order in the input, but it compensates this with the fact that its running time has very little variance across the universe of permutations. Intuitively, items at the leaves of the heap have small keys, which makes the sinking usually travel down to a leaf. Thus, almost all the n updates in the heap take at least $\Omega(i)$ comparisons, making the number of comparisons vary very little from one input permutation to another. Although its average case performance may not be as good as *Quicksort* (a constant larger than two is usually the difference), it is rather simple to obtain an implementation that is robust. It is a very good choice for an internal sorting algorithm. Sorting by selection with an array having a heap property is also used for external sorting.

Quicksort

For many applications a more realistic measure of the time complexity of an algorithm is its expected time. In sorting, a classical example is *Quicksort* [19, 29], which has an optimal expected time complexity of $O(n \log n)$ under the decision tree model, while there are sequences that force it to perform $\Omega(n^2)$ operations (in other words, its worst-case time complexity is quadratic). If the worst-case sequences are very rare, or the algorithm exhibits small variance around its expected case, then this type of algorithm is suitable in practice.

Several factors have made *Quicksort* a very popular choice for implementing a sorting routine. *Quicksort* is a simple divide-and -conquer concept, the partitioning can be done in a very short loop and is also conceptually simple, its memory requirements can be guaranteed to be only logarithmic on the size of the input, the pivot can be placed in a register and, most importantly, the expected number of comparisons is almost half of the worst-case optimal competitors, most notably *Heapsort*. In their presentation, many introductory courses on algorithms favor *Quicksort*. However, it is very easy to implement *Quicksort* in a way that it seems correct and extremely efficient for many sorting situations. But, it may be hiding $O(n^2)$ behavior for a simple case (for example, sorting n equal keys). Users of such library routine will be satisfied initially, only to find out later that on something that seems a simple sorting task, the implementation is consuming too much time to finish the sort.

Fine tuning of *Quicksort* is a delicate issue [5]. Many of the improvements proposed may be compensated by reduced applicability of the method or more fragile and less clear code. Although partitioning is conceptually simple, much care is required to avoid common pitfalls. Among these, we must assure that the selection and placement of the pivot maintains the assumption about the distribution of input sequences. That is, the partitioning must guarantee that all permutations of smaller sequences are equally likely when the permutation of n items are equally likely. The partitioning must also handle extreme cases, which occur far more frequently in practice than the uniformity assumption for the theoretical analysis. These extreme cases include the case where the file is already sorted (either in ascending or descending order) and its subcase, the case in which the keys are all equal, as well as the case in which many keys are replicated. One common application of sorting is bringing together items with equal keys [5].

Recall that *Quicksort* is a prototype of divide-and-conquer with the core of the work performed during the divide phase. The standard *Quicksort* algorithm selects from a fixed location in the array a splitting element or *pivot* to partition a sequence in two parts. After partitioning, the items of the subparts are in correct order with respect to each other. Most partition schemes result in *Quicksort* not being stable. Fig. 6 presents a version for partitioning that is correct and assures $O(n \log n)$ performance even if the keys are all equal; it does not require sentinels and the indices i and j never go out of bounds from the subarray. The drawback of using fixed location pivots for the partitioning is when the input is sorted (in descending or ascending order). In this cases, the choice of th pivot drives *Quicksort* to $\Omega(n^2)$ performance. This is still the case for the routine presented

```
int PARTITION (X, l, r);

1: pivot ← X[l];
2: i ← l − 1;
3: j ← r + 1;
4: loop
5:    repeat
6:       j ← j − 1;
7:    until  X[j] ≤ pivot
8:    repeat
9:       i ← i + 1;
10:   until  X[i] ≥ pivot
11:   if  i < j then
12:      exchange X[i] ↔ X[j];
13:   else
14:      return j;
15:   end if
16: end loop
```

Figure 6: A simple and robust partitioning.

```
QUICKSORT (X, l,r);

1: if l < r then
2:    split ← PARTITION(X,l,r);
3:    QUICKSORT(X,l,split);
4:    QUICKSORT(X,split+1,r);
5: end if
```

Figure 7: Pseudocode for *Quicksort*. An array is sorted with the call QUICKSORT(X,1,n).

here. However, we have accounted for repeated key values, so if there are e key values equal to the pivot's, then $\lceil e/2 \rceil$ end up in the right subfile. If all keys are always different, the partition can be redesigned so that it leaves the pivot in its correct position and out of further consideration. Fig. 7 presents the global view of *Quicksort*. The second subfile is never empty (i.e., $p < r$), and thus, this *Quicksort* always terminates.

The most popular variants to protect *Quicksort* from worst-case behavior are the following. The splitting item is selected as the median of a small sample, typically three item (the first, middle, and last element of the subarray). Many results show that this can deteriorate the expected average time by about 5% to 10% (depending on the cost of comparisons and how many keys are different). This approach assures that a worst-case happens with negligible low probability. For this variant, the partitioning can accommodate the elements used in the sample so then no sentinels are required, but there is still the danger of many or equal keys.

Another proposal delays selection of the splitting element; instead, a pair of elements that determine the range for the median is used. As the array is scanned, every time an element falls between the pair, one of the values is updated to maintain the range as close to the median as possible. At the end of the partitioning, two elements are in their final partitions, dividing the interval. This methods is fairly robust, but it enlarges the inner loop deteriorating performance, there is a subtle loss of randomness, and it also complicates the code significantly. Correctness for many equal keys remains a delicate issue.

Other methods are not truly comparison-based; for example, they use pivots that are the arithmetic averages of the keys. These methods reduce the applicability of the routine and may loop forever on equal keys.

Randomness can be a useful tool in algorithm design, especially if some bias in the input is suspected. A randomization version of *Quicksort* is practical because there are many ways in which the algorithm can proceed with good performance and only a few worst cases. Some authors find displeasing the need to use a pseudo-random generator for a problem as well studied as sorting. However, we find that the simple partitioning routine presented in Fig 7 is robust to many of the aspects that make partitioning difficult to code and can remain simple and robust while also handling worst-case performance with the use of randomization. The randomized version

ROUGHLY QUICKSORT (X, l, r);

1: **while** $r - l > 10$ **do**
2: $i \leftarrow$ RANDOM(l,r);
3: exchange $X[i] \leftrightarrow X[l]$;
4: split $\leftarrow$ PARTITION)$X,l,r)$;
5: ROUGHLY QUICKSORT$(X,l,$split$)$;
6: $l \leftarrow$ split $+1$;
7: **end while**

Figure 8: Randomized and tuned version of *Quicksort*.

of *Quicksort* is extremely solid and easy to code; refer to Fig. 8. Moreover, the inner loop of *Quicksort* remains extremely short; it is inside partition and consists of modifying and integer by 1 (increment or decrement, a very efficient operation in current hardware) and comparing a key with the key of the pivot (this value, along with the indexes i and j, can be placed in a register of the CPU). Also, access to the array exhibits a lot of locality of reference. By using the randomized version, the space requirements become $O(\log n)$ without the need to sort recursively the smallest of the two subfiles produced by the partitioning. If ever in practice the algorithm is taking to long (something with negligible probability), just halting it and running it again will provide a new seed with extremely high probability of reasonable performance.

Further improvements can now be made to tune up the code (of course, sacrificing some simplicity). One of the recursive calls can be eliminated by **tail recursion removal**, and thus the time for half of the procedure call is saved. Finally, it is not necessary to use a technique such as *Quicksort* itself to sort small files of less than 10 items by a final call to *Insertion Sort* to complete the sorting. Fig. 8 illustrates the tuned hybrid version of *Quicksort* that incorporates these improvements. To sort a file, first a call is made to ROUGHLY QUICKSORT$(X,1,n)$ immediately followed by the call INSERTION SORT(X,n). Obviously both calls should be packed under a call for *Quicksort* to avoid accidentally forgetting to make both calls. However, for testing purposes, it is good practice to call them separately. Otherwise, we may receive the impression the implementation of the *Quicksort* part is correct, while *Insertion Sort* is actually doing the sorting.

It is worth revising the analysis of *Quicksort*(alhtough itneresting new analyzes have emerged [10, 14]. We will do this for the randomized version. This version has no bad inputs. For the same input, each run has different behavior. The analysis computes the expected number of comparisons performed by *Randomized Quicksort* on an input X. Because the algorithm is randomized, $T_{RQ}(X)$ is a random variable, and we will be interested in its expected value. For the analysis, we evaluate the largest expected value $T_{RQ}(X)$ over all inputs with n different key values. Thus, we estimate $E[T_{RQ}(n)] = max\{E[T_{RQ}(X)] \mid \|X\| = n\}$. The largest subproblem for a recursive call is $n-1$. Thus, the recursive form of the algorithm allows the following derivation, where c is a constant.

$$\begin{aligned} E[T)RQ(n)] &= \sum_{i=1}^{n-1} Prob[i=k] E\left[\begin{array}{l}\sharp \text{ of comparisons when subcases} \\ \text{are of sizes } i \text{ and } n-i\end{array}\right] + cn \\ &\leq cn + \frac{1}{n}\sum_{i=1}^{n-1}(E[T_{RQ}(i)] + E[T_{RQ}(n-i)]) \\ &= cn + \frac{2}{n}\sum_{i=1}^{n-1} E[T_{RQ}(i)]. \end{aligned} \quad (1)$$

Since $E[T_{RQ}(0)] \leq b$ and $E[T_{RQ}(1)] \leq b$ for some constant b, then it is not hard to verify by induction that there is a constant k such that $E[T_{RQ}(n)] \leq kn\log n = O(n\log n)$, for all $n \geq 2$, which is the required result. Moreover, the recurrence (1) can be solved exactly to obtain an expression that confirms the constant hidden under the O notation is small.

Mergesort

Mergesort is not only a prototype of the conquer from in divide-and-conquer, as we saw earlier. *Mergesort* has two properties that can make it a better choice over *Heapsort* and *Quicksort* in many applications. The first of these properties is that *Mergesort* is naturally a stable algorithm, while additional efforts are required to

```
type
      list ↑ item;
      item = record k: keytype;
                    next : list;
             end;

function merge (X1,X2 :list) : list;
var head, tail, t: list;
begin
      head := nil;
      while X2 <> nil
             do
                    if X1 = nil (* reverse roles of X2 and X1 *)
                    then begin X1:=X2; X2:= nil; end
                    else  begin
                                 if X2↑.k > X1↑.k
                                        then begin t:= X1; X1:=X1↑.next end;
                                        else  begin t:=X2; X2:=X2↑.next end;
                                 t↑.next :=nil;
                                 if head = nil  then head:=t;
                                                else tail↑.next:=t;
                                 tail:=t;
                           end;
      if head = nil   then head:=X1;
                      else tail↑.next :=X1;
      merge:=head
end
```

Figure 9: PASCAL code for merging two linked lists.

obtain stable versions of *Heapsort* and *Quicksort*. The second property is that access to the data is sequential; thus, data does not have to be in an array. This makes *Mergesort* and ideal method to sort linked lists. It is possible to use a divide form for obtaining a *Quicksort* version for lists that is also stable. However, the methods for protection against quadratic worst cases still make *Mergesort* a more fortunate choice. The advantages of *Mergesort* are not without cost. *Mergesort* requires $O(n)$ extra space (for another array or for the pointers in the linked list implementation). It is possible to implement *Mergesort* with constant space, but the gain hardly justifies the added programming effort.

Mergesort is based upon merging two sorted sequences of roughly the same length. Actually, *merging* is a very common special case of sorting, and it is interesting in its own right. Merging two ordered list (or roughly the same size) is achieved by repeatedly comparing the head elements and moving the one one with the smaller key to the output list. Fig. 9 shows PASCAL code for merging two linked lists. The PASCAL code for *Mergesort* is shown in Fig. 10. It uses the function for merging of the previous figure. This implementation of *Mergesort* is more general that a sorting procedure for all the items in a linked list. It is a PASCAL function with two parameters, the head of the lists to be sorted and an integer n indicating how many elements from the head should be included in the sort. The implementation returns as a result the head of the sorted portion and the head of the original list is a VAR parameter adjusted to point to the $(n+1)$-th element of the original sequence. If n is larger or equal to the length of the list, the pointer returned includes the whole list and the VAR parameter is set to nil.

Restricted Universe Sorts

In this section we present algorithms that use other aspects about the keys to carry out the sorting. These algorithms were very popular at some point, and were the standard to sort punched cards. With the emergence of comparison-based sorting algorithms, which provided generality as well as elegant analyzes and matching bounds, these algorithms lost popularity. However, their implementation can be much faster than comparison-based sorting algorithms. The choice between comparison-based methods and these types of algorithms may

```
function mergesort( VAR: head; n:integer): list;
var t: list;
begin
      if head = nil
      then mergesort = nil
      else if n > 1
           then mergesort :=merge ( mergesort(head, n div 2), mergesort(head, (n+1) div 2) )
           else begin
                      t:= head;
                      head := head↑.next ;
                      t ↑.next := nil ;
                      mergesort := t;
                end
end
```

Figure 10: PASCAL code for a merge function that sorts n items of a linked list.

depend on the particular application. For a general sorting routine, many factors must be considered, and the criteria to determine which approach is best suited should not be limited to just running time. If the keys meet the conditions for using these methods, they are certainly a very good alternative. In fact, today's technology has word lengths and memory sizes that make competitive many of the algorithms presented here. These algorithms were considered useful for only small restricted universes. Large restricted universes can be implemented with the current memory sizes and current word sizes for many practical cases. Recent research has shown theoretical [3] and practical [4] improvements on older versions of these methods.

Distribution Counting

A special situation of the sorting problems $X = \langle x_1, \ldots, x_n \rangle$ is the sorting of n distinct integers in the range [1,m]. If the value of $m = O(n)$, then the fact that the x_i are distinct integers allows a very simple sorting method that runs in $O(n)$ time. Use a temporary array T:[1,m] and place each x_i in $T[x_i]$. Scan T to collect th items in sorted order (where T was initialized to hold only 0). Note there that we are using the power of the Random Access Machine to locate an entry in an array in constant time.

This idea can be extended in many ways; the first is to handle the case when the integers are no longer distinct, and the resulting method is called *Distribution Counting*. The fundamental idea is to determine, for each x_i, its *rank*. The rank is the number of elements less or equal (but before x_i in X) than x_i. The rank can be used to place x_i directly in its final position in the output array OUT. To compute the rank, we use the fact that the set of possible key values is a subset of the integers in [1,m]. We count the number E_k of values in X that equal k, for $k = 1, \ldots, m$. Arithmetic sums $\sum_{k=1}^{i} E_k$ can be used to find how many x_j are less or equal to x_i. Scanning though X, we can now find the destination of x_i when x_i is reached. Fig. 11 presents the pseudocode for the algorithm. Observe that two loops are till n and two till m. From this observation, the $O(n + m) = O(n)$ time complexity follows directly. The method has the disadvantage that extra space is required; however, in practice, we will use this method when m fits our available main memory, and in such cases, this extra space is not a problem. A very appealing property of *Distribution Counting* is that it is a stable method. Observe that not a single comparison is required to sort. However, we need an array of size m, and the key values must fit the addressing space.

Bucket Sort

Bucket Sort is an extension to the idea of finding out where in the output array each x_i should be placed. However, the keys are not necessarily integers. We assume that we can apply an interpolation formula to the keys to obtain a new key in the real interval [0,1) which proportionally indicates where the x_i should be relative to the smallest and largest possible keys. The interval [0,1) is partitioned into m equal-sized consecutive interval, each with an associated queue. The item x_i is placed in the queue q_j when the interpolation address from its key lies in the j-th interval. The queues are sorted recursively, and then concatenated starting from the lower interval. The first-in last-out properties of the queues assures that, if the recursive method is stable, the overall sorting is stable. In particular, if *Bucket Sort* is called recursively, the method is stable. However, in practice, it

DISTRIBUTION COUNTING(X,m,OUT);

```
1: for k ← 1 to m do
2:    count[k] ← 0;
3: end for
4: for i ← 1 to n do
5:    count[X_i] ← count[ X_i] + 1;
6: end for
7: for k ← 2 to m do
8:    count[k] ← count[k]+count[k − 1];
9: end for
10: for i ← n down-to 1 do
11:    OUT[count[X_i]] ← X_i;
12:    count[X_i] ← count[X_i]-1;
13: end for
```

Figure 11: Pseudocode for *Distribution Counting*.

BUCKET SORT (X,n,m);

```
1: for i ← 0 to m − 1 do
2:    Queue[i] ← empty;
3: end for
4: for i ← 1 to n do
5:    insert x_i into Queue[⌊ interpol(x_i)m⌋];
6: end for
7: for i ← 0 to m − 1 do
8:    SORT(Queue[i]);
9: end for
10: for i ← 1 to m − 1 do
11:    Concatenate Queue[i] at the back of Queue[0];
12: end for
13: return Queue[0];
```

Figure 12: The code for *Bucket Sort*.

is expected that the queues will have very few items after one or two partitions. Thus, it is convenient to switch to an alternative stable sorting method to sort the items in each bucket, most preferable *Insertion Sort*. For the insertion into the queues, an implementation that allows insertion in constant time should be used. Usually, linked lists with a pointer to their last item is the best alternative. This also a assures that the catenation of the queues is efficient.

The method has an excellent average-case time complexity, namely, it is linear (when $m = \Theta(n)$). However, the assumption is a uniform distribution of the interpolated keys in [0,1). In the worst scenario, the method may send every item to one bucket only, resulting in quadratic performance. The difficulty lies in finding the interpolation function. These functions work with large integers (like the maximum key) and must be carefully programmed to avoid integer overflow.

However, the method has been specialized so that k rounds of it sort k-tuples of integers in [1,m] in $O(k(n+m))$ time, and also to sort strings of characters with excellent results [1]. Namely, strings of characters are sorted in $O(n + L)$ time where L is the total length of the strings. In this cases, the alphabet of characters defines a restricted universe and the interpolation formula is just a displacement from the smallest value. Moreover, the number of buckets can be made equal to the different values of the universe. These specializations are very similar to radix sorting, which we discuss next.

Radix Sort

Radix Sort refers to a family of methods where the keys are interpreted as representation in some base (usually

a power of 2) or a string over a given small but ordered alphabet. The radix sorts examines the digits of this representation in as many rounds as the length of the key to achieve the sorting. Thus, radix sorts perform several passes over the input, in each pass, performing decisions by one digit only.

The sorting can be done from the most significant digit toward the least significant digit or the other way around. The radix sort version that goes from the most significant towards least significant digit is called *Top-Down Radix Sort*, *MSD Radix Sort*, or *Radix Exchange Sort* [16, 21, 30]. It resembles *Bucket Sort*, and from the perspective of divide-and-conquer is a method of the divide form. The most significant digit is used to split the items into groups. Next, the algorithms is applied recursively to the groups separately, with the first digit out of consideration. The sorted groups are collected by the order of increasing values of the splitting digit. Recursion is terminated by groups of size one. If we consider the level of recursion as rounds over the strings of digits of the keys, the algorithm keeps the invariant that after the i-th pass, the input is sorted according to the first i digits of the keys.

The radix sort version that proceeds from the least significant digit toward the most significant digit is usually called *Bottom-Up Radix Sort*, *Straight Radix Sort*, *LSD Radix Sort*, or just *Radix Sort* [16, 21, 30]. It could be considered as doing the activities of each round in different order, Split the items into groups according to the digit under consideration, group the items in order of increasing values of the splitting digit. Apply the algorithm recursively to all the items, but considering the next more significant digit. At first, it may not seem clear why this method is even correct. It has a dual invariant to the *Top-Down Sort*; however, after the i-th pass, the input is sorted according to the last i digits of the keys. Thus, for this *Bottom-Up* version to work, it is crucial that the insertion of items in their groups is made first-in first-out order of a queue. For the *Top-Down* version, this is only required to ensure stability. Both methods are stable though.

The *Top-Down* version has several advantages and disadvantages with respect to the *Bottom-Up* version. In the *Top-Down* version, the algorithm only examines the distinguishing prefixes, while the entire set of digits of all keys are examined by the *Bottom-Up* version. However, the *Top-Down* version needs space to keep track of the recursive calls generated, while the *Bottom-Up* version does not. If the input digits have a random distribution, then both versions of radix sort are very effective. However, in practice this assumption regarding the distribution is not the case. For example, if the digits are the bits of characters, the first leading bits of all lower case letters are the same in most character encoding schemes. Thus, *Top-Down Radix Sort* deteriorates with files with many equal keys (similar to *Bucket Sort*).

The *Bottom-Up* version is like a *Distribution Counting* on the digit that is being used. If fact, this is the easiest way to implement it. Thus, the digits can be processed more naturally as groups of digits (and allowing a large array for the distribution counting). This is an advantage of the *Bottom-Up* version over the *Top-Down* version.

It should be pointed out that radix sorts can be considered linear in the size of the input, since each digit of the keys is examined only once. However, other variants of the analysis are possible; these include modifying the assumptions regarding the distribution of the keys or according to considerations of the word size of the machine. Some authors think that the n keys require $\log n$ bits to be represented and stored in memory. From this perspective, radix sorts require $\log n$ passes with $\Omega(n)$ operations on them, still amounting to $O(n \log n)$ time. In any case, radix sorts are a reasonable method for a sorting routine, or a hybrid one. One hybrid method proposed by Sedgewick [30] consists of using the *Bottom-Up* version of radix sort, but for the most significant half of the digits of the keys. This makes the file almost sorted, so the sort can be finished by *Insertion Sort*. The result is linear sorting methods for most current word sizes on randomly distributed keys.

Order Statistics

The k-th *order statistic* of a sequence of n items is the k-th larger item. In particular, the smallest element is the first order statistic while the largest element is the n-th order statistic. Finding the smallest or the largest item in a sequence can easily be achieved in linear time. For the smallest item, we just have to scan the sequence, remembering the smallest item seen so far. Obviously, we can find the first and second statistic in linear time by the same procedure, just remembering the two smallest items seen so far. However, as soon as $\log n$ statistics are required, it is best to sort the sequence and retrieve any order statistic required directly.

A common request is to find jointly the smallest and largest items of a sequence of n items. Scanning though the sequence remembering the smallest and largest items seen so far requires that each new item be compared with what is being remembered; thus $2n + O(1)$ comparisons are required. A better alternative in this case is to form $\lfloor n/2 \rfloor$ pairs of items, and perform the comparisons in pairs. We find the smallest among the smaller items in the pairs, while the largest is found among the larger items of the pairs (a final comparison may be required for an element left out when n is odd). This results in $\lfloor 3n/2 \rfloor + O(1)$ comparisons, which in some applications is worth the programming effort.

```
SELECT (X,l,r,k0;

1: if  r = l then
2:    return X[l];
3: end if
4: i ← RANDOM(l,r);
5: exchange X[i] ↔ X[l];
6: split ← PARTITION(X,l,r);
7: if k ≤ split then
8:    return SELECT(X,l,split,k);
9: else
10:   return SELECT(X,split+1,r,k-split);
11: end if
```

Figure 13: Randomized version for selection of the k-th largest.

The fact that the smallest and largest items can be retrieved in $O(n)$ time without the need for sorting made the quest for linear algorithms for the k-th order statistic a very interesting one for some time. Still, today there are several theoreticians researching the possibility of linear selection of the median (the $\lfloor n/2 \rfloor$-th item) with a smaller constant factor. As a matter of fact, selection of the k-th largest item is another illustration of the use of average case complexity to reflect a practical situation more accurately than worst-case analysis. The theoretical worst-case linear algorithms are so complex that few authors dare to present pseudocode for them. This is perfectly justified, because nobody should implement worst-case algorithms in the light of very efficient algorithms in the expected case, which are far easier conceptually, they are simpler in programming effort terms and can be protected from worst-case performance (by making such worst-case extremely unlikely).

Lets consider divide-and-conquer approaches to finding the k-th largest element. If we take the conquer from as in *Mergesort*, it seems difficult to imagine how the k-th largest item of the left subsequence and the k-largest item of the right subsequence relate to the k-th largest item of the overall subsequence. However, if we take the divide form, as in *Quicksort*, we see that partitioning divides the input and conceptually splits by the correct rank of the pivot. If the position of the pivot is $i \geq k$, we only need to search for the $X_{\leq}$ subsequence. Otherwise, we have found i items that we can remove from further consideration, since they re smaller than the k-th largest. We just need to find the $k-i$ largest in the subsequence $X_{\geq}$. This approach to divide-and-conquer results in only one subproblem to be pursued recursively. The analysis results in an algorithm that requires $O(n)$ time in the expected case. Such a method requires, again, careful protection against the worst case. Moreover, it is more likely that a file that is being analyzed for its order statistics has been inadvertently sorted before, setting up a potential worst case for a selection methods whose choice of the pivot is not adequate. In the algorithm presented in Fig. 13 we use the same partitioning algorithm as the section for *Quicksort*; refer to Fig. 7.

External Sorting

There are many situations where external sorting is required for the maintenance of a well-organized database. Files are often maintained in sorted order with respect to some attribute to facilitate searching and processing. External sorting is not only used to produce organized output, but also to efficiently implement complex operations such as a relational join. Although main memory sizes have consistently been enlarged, they have not been able to keep up with the ability to collect and store data. Fields like *Knowledge Discovery and Data Mining* have demonstrated the need to cleverly manipulate very large volumes of information, and the challenges for managing external storage. External sorting manages the trade-offs of rapid random access of internal memory with the relatively fast sequential access of secondary disks by sorting in two phases; a *run creation phase* and a *merge phase*. During the first phase, the file to be sorted is divided into smaller sorted sequences called *initial runs* or *strings* [21]. These runs are created by bringing into main memory a fragment of the file. During the second phase, one or more activations of *multi-way merge* are used to combine the initial runs into a single run [27].

Currently, the sorting of files that are too large to be held in main memory is performed on disk drives [27]; see Fig. 14. Situations where only one disk drive is available are now uncommon, since this usually results into very slow sorting processes and complex algorithms while the problem can be easily solved with another disk drive (which is affordable today) or a disk array with independent read-writing heads. In each pass (one for

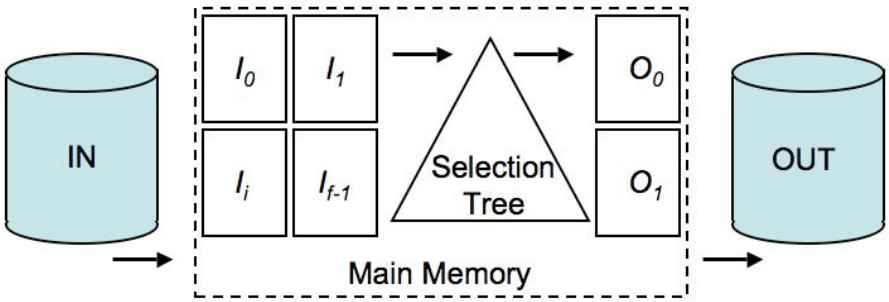

Figure 14: The model for a pass of external sorting.

run-creation and one of more for merging) the input file is read from the IN disk drive. The output of one pass in the input for the next, until a single run is formed; thus the IN and OUT disks swap roles after each pass. While one of the input buffers, say I_i, $i \in \{0, f-1\}$, is being filled, the sorting process reads records from some of the other input buffers $I_0, \ldots, I_{i-1}, I_{i+1}, \ldots, I_{f-1}$. The output file of each pass is written using *double buffering*. While one of the buffers, say O_i, $i \in \{0, 1\}$, is being filled, the other buffer O_{1-i} is being written to disk. The roles of O_i and O_{1-i} are interchanged when one buffer is full and the other is empty. In practice, the goals is to produce as much sequential reading and writing as possible, although these days, the operating system may take over the request for reading and writing to disks from other processes in the same machine. Hopefully, several data records are read in a Input/Output operation forming a *physical blocks*, while the capacity of the buffers defines the size of a *logical block*. For the description of external sorting methods, the use of logical blocks is usually sufficient and we will just name them *blocks*.

During the run-creation phase, the number f of input buffers is 2 and reading is sequential using double buffering. During the merge pass, the next block to be read is normally from a different run and the disk arm must be repositioned. Thus, reading is normally not sequential. Writing during the merge is, however, faster than reading, since normally it is performed sequentially and no seeks are involved (except for the occasional seek for the next cylinder when the current cylinder is full). In each pass, the output is written sequentially to disk. Scattered writing during a pass in anticipation of saving seeks because of some sequential reading of the next pass has been shown to be counterproductive [34]. Thus, in the two-disk model (Fig. 14) the writing during the merge will completely overlap with the reading and its time requirements are a minor concern. In contrast to the run-creation phase in which reading is sequential, merging may require a seek each time a data block is read.

Replacement selection usually produces runs that are larger than the available main memory; the larger the initial runs, the faster the overall sorting. Replacement selection allows full overlapping of I/O with sequential reading and writing of the data, and it is standard for the run creation phase. The classic result on the performance of replacement selection establishes that, when the permutations in the input files are assumed to be equally likely, the asymptotic expected length of the resulting runs is twice the size of available main memory [21]. Other researchers have modified replacement selection, such that, asymptotically, the expected length of an initial run is more than twice the size of available main memory. These methods have received limited acceptance because they require more sophisticated I/O operations and prevent full overlapping; hence, the possible benefits hardly justify the added complexity of the methods. Similarly, any attempt to design a new run-creation methods that profits from the existing order in the input file almost certainly have inefficient overlapping of I/O operations. More recently, it has been mathematically confirmed that the lengths of th runs created by replacement selection increase as the order of the input files increases [12].

During the run-creation phase *Replacement Selection* consists of a *selection tree*. This structure is a binary tree where nodes hold the smaller of their two children. It is called selection tree because the item at the root of the tree holds the smallest key. By tracing the path up the smallest key from its place at a leaf to the root, we have selected the smallest item among those in the leaves. If we replace the smallest item with another value at the corresponding leaf, we only are required to update the path to the root. Performing the comparisons along this path updates the root as the new smallest item. Selection trees are different from heaps (ordered to extract the item with the smallest keys) in that selection trees have fixed size. During the selection phase, the selection tree is initialized with the first P elements of the input file (where P is the available internal memory). Repeatedly, the smallest item is removed from the selection tree and placed in the output stream, the next item from the input file is inserted in its place as a leaf in the selection tree. The name *Replacement Selection* comes from the fact that the new item from the input file replaces the item just selected to the output stream. To

make certain that items enter and leave the selection tree in the proper order, the comparisons are not only with respect to the sorting keys, but also with respect to the current run being output. Thus, the selection tree uses lexicographically the composite keys (r,*key*), where r is the *run-number* of the item, and *key*, is the sorting key. The run number of an item entering the selection tree is know by comparing it to the item which it is replacing. If it is smaller than the item just sent to the output stream, the run number is one more than the current run number; otherwise, it is the same run number as the current run.

The Merge

In the merging phase of external sorting, blocks from each run are read into main memory, and the records from each block are extracted and merged into a single run. *Replacement selection* (implemented with a *selection tree*) is also used as the process to merge the runs into one [21]. Here, however, each leaf is associated with each of the runs being merged. The *order* of the merge is the number of leaves in the selection tree. Because main memory is a critical resource here, items in the selection tree are not replicated, but rather a tree of losers is used [21].

There are many factors involved in the performance of disk drives. For example, larger main memories implies larger data blocks and the *block-transfer rate* is now significant with respect to *seek time* and *rotational latency*. Using a larger block size reduces the total number of reads (and seeks) and reduces the overhead of the merging phase. Now, a merge pass requires at least as many buffers as the order ω of the merge. On the other hand, using only one buffer for each run, maximizes block size, and if we perform a seek for each block, it reduces the total number of seeks. However, we cannot overlap I/O. On the other hand, using more buffers, say, two buffers for each run, increases the overlap of I/O, but reduces the block size and increases the total number of seeks. Note that because the amount of main memory is fixed during the merge phase, the number of buffers is inversely proportional to their size.

Salzberg [28] found that, for almost all situations, the use of $f = 2\omega$ buffers assigned as pairs to each merging stream outperforms the use of ω buffers. It has been shown that double buffering cannot take advantage of nearly-sorted data [13]. Double buffering does not guarantee full overlap of I/O during merging. When buffers are not fixed to a particular run, but can be reassigned to another run during the merge, the are called *floating buffers* [20]. Using twice as many floating buffers as the order of the merge provides maximum overlap of I/O [20]. Zheng and Larson [34] combined Knuth's [21] *forecasting* and floating-buffers techniques and proposed six to ten times as many floating input buffers as the order of the merge, but also require less main memory. Moreover, it was also demonstrated that techniques based on floating buffers profit significantly from nearly-sorted files [13].

Floating Buffer

The *consumption sequence* is the particular order in which the merge consumes blocks from the runs being merged. This sequence can be computed by extracting the highest key (the last key) from each data block (during the previous pass) and sorting them. The time taken to compute the consumption sequence can be overlapped with the output of the last run and the necessary space for the subsidiary internal sort is also available then; thus, the entire consumption sequence can be computed during the previous pass with negligible overhead. The floating-buffers technique exploits the knowledge of the consumption sequence to speed up reading.

We illustrate double buffering and floating buffers with a merging example of four runs that are placed sequentially as shown in Fig. 15. Let $C = \langle C_1, C_2, \ldots, C_T \rangle$ be the consumption sequence, where C_i identifies a data block with respect to its location on the disk. For example, consider

$$C = \langle 1, 8, 13, 18, 2, 9, 14, 19, 3, 10, 4, 5, 15, 11, 12, 16, 17, 20, 6, 21, 7, 22, 23 \rangle^{1}.$$

Double buffering uses twice as many buffers as runs, and a seek is required each time a block is needed from a different run. Moreover, even when a block is needed from the same run, this may not be known at exactly the right time; therefore, the disk will continue to rotate and every read has rotational latency. In the example, double buffering reads one block from each run (with a seek in each case) and then it reads a second block from each run (again with a seek in each case). Next, the disk arm travels to block 3 to read a new bock from the first run (one more seek). Afterward, the arm moves to block 10 to get a new block from the second run. Then, it moves to block 4 and reads block 4 and block 5, but a seek is not required for reading block 5 since the run-creation phase places blocks from the same run sequentially on the disk. In total, double buffering performs 19 seeks.

[1] Alternatively, the consumption sequence can be specified as $C = \langle c_1, c_2, \ldots, c_T \rangle$ where c_i is the run from which the i-block should be read. For this example, $C = \langle 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 1, 1, 3, 2, 2, 3, 3, 4, 1, 4, 1, 4, 4 \rangle$.

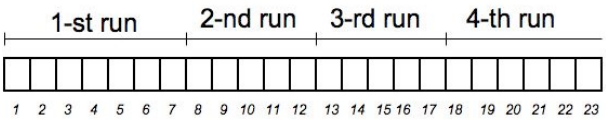

Figure 15: An example of four runs (written sequentially) on a disk.

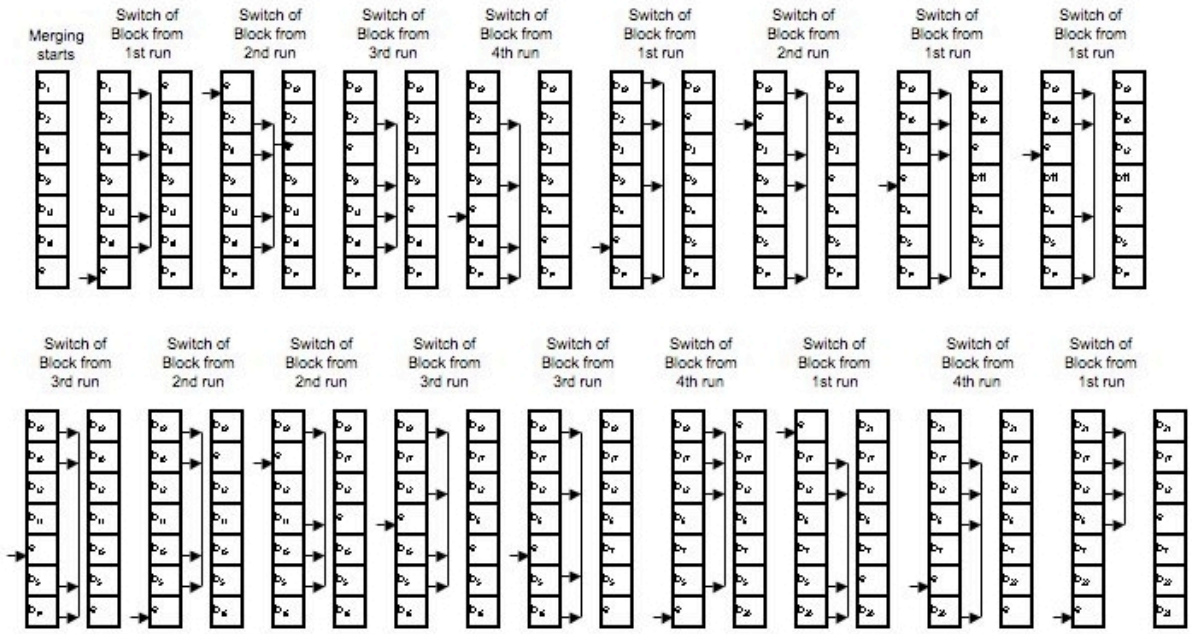

Figure 16: Merging with seven floating buffers.

We now consider the effect of using only seven buffers (of the same size as before) managed as floating buffers [20]. In this case, we use only three more buffers than the number of runs but we use knowledge of the consumption sequence. The seven buffers are used as follows: four buffers contain a block from each run that are currently being consumed by the merge, two buffers contain look-ahead data, and one buffers is used for reading new data. In the previous example, after the merging of the two data blocks 1 and 7, the buffers are as follows.

block 24	block 2	empty	block 8	block 15	block 23	block 16
buffer 1	buffer 2	buffer 3	buffer 4	buffer 5	buffer 6	buffer 7

Data from buffers 2, 4, 5, and 6 are consumed by the merge and are placed in an output buffer. At the same time, one data block is read into buffer 3. As soon as the merge needs a new block from run 3, it is already in buffer 7 and the merge releases buffer 5. Thus, the system enters a new state in which we merge buffers 2, 4, 6, and 7, and we read a new block into buffer 5. Fig. 16 shows the merge when the blocks are read in the following order:

$$\langle \underline{1,2}, \underline{8,9}, \underline{13}, \underline{18}, \underline{14}, \underline{19}, 3, 4, 5, \underline{10,11,12}, \underline{15,16,17}, \underline{6,7}, \underline{20,21,22,23} \rangle. \quad (2)$$

The letter e denotes an empty buffer and b_i denotes a buffer that holds data block i. Reading activity is indicated by an arrow into a buffer and merging activity is indicated by an arrow out of a buffer. The ordered sequence in which the blocks are read from the disk into main memory is called *reading sequence*. A reading sequence is *feasible* if every time the merge needs a block, it is already in main memory and reading never has to wait because there is no buffer space available in main memory. Note that the consumption sequence (with two or more floating buffers for each run) is always a feasible sequence [20]. In the example of Fig. 16, not only is the new reading sequence feasible and provides just in-time blocks for the merge, but also it requires only 11 seeks and uses even less memory than double buffering!

Computing a Feasible Reading Sequence

In the previous section, we have illustrated that floating buffers can save main memory and the overhead due to seeks. There are, however, two important aspects of using floating buffers. First, floating buffers are effective when knowledge of the consumption sequence is used to compute reading sequences [34]. The consumption sequence is computed by the previous pass (and for the first pass, during the run-creation phase). The consumption sequence is the consumption order of data-blocks in the next pass. Thus, the buffer size for the next pass must be known by the previous pass. Overall, the buffer size and the number f of input floating buffers for each pass must be chosen before starting the sorting.

Before sorting we usually know the length $|X_I|$ of the file, and assuming it is in random order, we expect, after run-creation, initial runs of twice the size P of available main memory. That is, $E[Runs(X_0] = |X_I|/(2P)$. Now, we can decide the number of merge passes (most commonly only one) and the order ω of these merge passes. Zheng and Larson [34] follow this approach and recommend the number f of floating buffers to be between 6ω and 10ω. Once the value of f is chosen, the buffer size is determined, and where to partition the input into data blocks is defined. The justification for this strategy is that current memory sizes allow it and an inaccurate estimate of the number of initial runs or their sizes seems not to affect performance [34]. It has been shown that if the input is nearly sorted, the fact that the technique just described may chose f much larger than 10ω does not affect floating buffers. Moreover, for nearly sorted files, reading during the merge becomes almost sequential, and over 80% of seeks can be avoided [13].

The second difficulty consist of computing feasible reading sequences that minimize the number of seeks, Zheng and Larson [34] have related the problem of finding the optimal feasible reading sequence to the travelling salesman problem thus, research has concentrated on approximation algorithms.

To describe precisely the problem of finding a feasible reading sequence with fewer seeks we will partition into what we call *groups*, the blocks in a read sequence that re adjacent and belong to the same run. The groups are indicated by underlining in the sequence shown as Equation (2). A seek in needed at the beginning of each group, because a disk head has to move to a different run. Inside a group, we read blocks from a single run sequentially, as placed by the run-creation phase or a previous merge pass. Note that there is no improvement in reading data blocks from the same run in different order than in the consumption sequence. For long groups, a seek may be required from one cylinder to the next, but such a seek takes minimum time because reading is sequential. Moreover, in this case, the need to read a block from the next cylinder is known early enough to avoid rotational latency. Thus, we want to minimize the number of groups while maintaining feasibility.

We now describe *group shifting*, an algorithm that computes a feasible reading sequence with fewer seeks. Group shifting starts with a consumption sequence $C = \langle C_1, \ldots, C_T \rangle$ as the initial feasible reading sequence. It scans the groups in the sequence twice. The first scan produces a feasible reading sequence that is the input for the next scan. A scan builds a new feasible reading sequence incrementally. The first ω groups of the new reading sequence are the first ω groups of the previous reading sequence because, for $i = 1, \ldots, \omega$, the optimal feasible reading sequence for the first i groups consist of the first i groups of the consumption sequence. In each scan, the groups of the previous sequence are analyzed in the order they appear. During the first scan an attempt is made to move each group in turn forward and catenate it with the previous group from the same run while preserving feasibility. A single group that results from the catenation of groups B_j and B_k is denoted by $(B_j B_k)$. During the second scan an attempt is made to move back the previous group from the same run under analysis, while preserving feasibility. We summarize the algorithm in Fig. 17.

For an example of a forward move during the first scan, consider $b = 4$, $M^b = \langle \underline{1}, \underline{8}, \underline{13}, \underline{18} \rangle$, and group $b+1$ is $\underline{2}$. Then, $M^{b+1} = \langle \underline{1, 2}, \underline{8}, \underline{13}, \underline{18} \rangle$. As an example of a backward move during the second scan consider $b = 18$, $M^b = \langle \underline{1, 2}, \underline{8, 9}, \underline{13}, \underline{18}, \underline{14}, \underline{19}, \underline{3, 4, 5}, \underline{10, 11, 12}, \underline{15, 16, 17}, \underline{20, 21}, \underline{6, 7} \rangle$, and group $b+1$ is $\underline{22, 23}$. Moving $\underline{20, 21}$ over $\underline{6, 7}$ gives the optimal sequence of Fig 16.

The algorithm uses the following fact to test that feasibility is preserved. Let $C = \langle C_1, \ldots, C^T \rangle$ be the consumption sequence for ω runs with T data blocks. A reading sequence $R = \langle R_1, \ldots, R_T \rangle$ is feasible for $f > \omega + 1$ floating buffers if and only if, for all k such that $f \leq k \leq T$, we have $\{C_1, C_2, \ldots, C_{k-f+\omega}\} \subset \{R_1, R_2, \ldots, R_{k-1}\}$.

Research Issues and Summary

We now look at some of the research issues on sorting from the practical point of view. In the area of internal sorting, advances in data structures for the abstract data type *dictionary* or for the abstract data type *priority queue* may result in newer or alternative sorting algorithms. The implementation of dictionaries by variants of binary search tress where the items can easily (in linear time) be recovered in sorted order with an *in-order* traversal, results in an immediate sorting algorithm. We just insert the items to be sorted into the tree

GROUP-SHIFTING ($C = \langle C_1, \ldots, C_T \rangle$ original consumptions sequence for a merge pass of order ω)

```
1: N ← ⟨, C_1, C_2, ..., C_ω⟩;
2: while b < T do
3:    let B be the (b+1)-th group in C and let the sequence N be N = ⟨B_1, B_2, ..., B_p⟩, where B_l is
      the last group in N from the same run as B.
4:    if group B can be moved forward and joined to B_l preserving feasibility then
5:       N = ⟨B_1, ..., B_{l-1}, (B_l B), B_{l+1}, ..., B_p⟩,
6:    else
7:       append B to N
8:    end if
9: end while
10: M ← first ω groups in N;
11: b ← ω;
12: while b < |N| do
13:    Let B be the (b+1)-th group in N and let the sequence M be M = ⟨B_1, ..., B_p⟩, where B_l is the
       last group in M from the same run as B.
14:    if group B_l can be shifted back to the end of N and joined to group B preserving feasibility then
15:       M ← ⟨B_1, ..., B_{l-1}, B_{l+1}, ..., B_P, (B_l B)⟩
16:    else
17:       append B to M
18:    end if
19: end while
20: return M
```

Figure 17: The algorithm for reducing seeks during the merging phase with floating buffers.

implementing the dictioanry using sorting keys as the dictionary keys. Later, we extract the sorted order from the tree. An insertion sort is obtained for each representation of the dictionary. Some interesting advances, at the theoretical level, but perhaps at the practical level, have been obtained using data structures like *Fusion Trees* [2, 15]. Although these algorithms are currently somewhat complicated and they make use of dissecting keys and careful placing of information in memory words, the increase in the word size of computers is making them practically feasible.

Another are of research is the more careful study of alternatives offered by radix sorts. Careful analyzes have emerged for these methods and they take into consideration the effect of non-uniform distributions. Moreover, simple combinations of *Top-Down* and *Bottom-up* have resulted in hybrid radix sorting algorithms with very good performance [3]. In practical experiments, Andersson and Nilsson observer that their proposed *Forward Radix Sort* defeats some of the best alternatives offered by the comparison-based approach. When sorting integers in a restricted universe, the theoretical trade-offs of time and space have been pushed extensively lately [17, 18].

A third area of research with possible practical implications is the area of adaptive sorting. When the sorting algorithms take advantage of existing order in the input, the time taken by the algorithm to sort is a smooth growing function of the size of the sequence and the disorder in the sequence. In this case,we say that the algorithm is *adaptive* [24]. Adaptive sorting algorithms are attractive because nearly sorted sequences are common in practice [21, 24, 30]; thus, we have the possibility of improving on algorithms that are oblivious to the existing order in the input.

So far we presented *Insertion Sort* as an example of this type of algorithm. Adaptive algorithms have received attention for comparison-based sorting. Many theoretical algorithms have been found for many *measures of disorder*. However, from the practical point of view, these algorithms usually involve more machinery. This additional overhead is unappealing because of its programming effort. Thus, room remains for providing adaptive sorting algorithms that are simple for the practitioner. Although some of these algorithms [8] have been shown to be far more efficient in nearly sorted sequences for just a small overhead on randomly permuted files, they have nor received wide acceptance.

Finally, let us summarize the alternatives when facing a sorting problem. First we must decide if our situation is in the area of external sorting. A model with two disk drives is recommended in this case. Use *Replacement*

Selection for run creation and for merging, using floating buffers during the second phase. It is possible to tune the sorting for just one pass during the second phase.

If the situation is internal sorting of small files, then *Insertion Sort* does the job. If we are sorting integers, or character strings, or some restricted universe, then *Distribution Counting*, *Bucket Sort*, and *Radix Sort* are very good choices. If we are after a stable method, *restricted universe sorts* are also good options. If we want something more general, the next level up is *Shellsort*, and finally the tuned versions of $O(n \log n)$ comparison-based sorting algorithms. If we have serious grounds to suspect the inputs are nearly sorted, we should consider adaptive algorithms. Whenever the sorting key is a small portion of the data records, we should try to avoid expensive data moves by sorting a file of keys and indexes. Always preserve a clear and simple code.

Defining Terms

Adaptive: A sorting algorithm that can take advantage of existing order in the input, reducing its requirements for computational resources as a function of the amount of disorder in the input.

Comparison-based algorithm: A sorting method that uses comparisons, and nothing else about the sorting keys, to rearrange the input into ascending or descending order.

Conquer form: An instantiation of the divide-and-conquer paradigm for the structure of an algorithm where the bulk of the work is combining the solutions of subproblems into a solution for the original problem.

Divide form: An instantiation of the divide-and-conquer paradigm for the structure of an algorithm where the bulk of the work is dividing the problem into subproblems.

External sorting: The situation where the file to be sorted is to large to fit in main memory. The need to consider that Random Access to data items is limited and sequential access is inexpensive.

Insertion sort: The family of sorting algorithms where one item is analyzed at a time and inserted into a data structure holding a representation of a sorted list of previously analyzed items.

Internal sorting: The situation when the file to be sorted is small enough to fit in main memory and using uniform cost for random access is suitable.

Multiway merge: The mechanism by which ω sorted runs are merged into a single run. The input runs are usually organized in pairs and merged using the standard method for merging two sorted sequences. The results are paired again, and merged, until just one run is produced. The parameter ω is called the order of the merge.

Restricted universe sorts: Algorithms that operate on the basis that the keys are members of a restricted set of values. They may not require comparisons of keys to perform the sorting.

Selection sorts: The family of sorting algorithms where the data items are retrieved from a data structure, one item at a time, in sorted order.

Sorting arrays: The data to be sorted is placed in an array and access to individual items can be done randomly. The goal of the sorting is that the ascending order matches the order of indices in the array.

Sorting linked lists: The data to be sorted is a sequence represented as a linked list. The goal is to rearrange the pointers of the linked list so that the linked list exhibits the data in sorted order.

Stable: A sorting algorithm where the relative order of items with equal keys in the input sequence is always preserved in the sorted output.

Bibliography

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Co., Reading, MA, 1974.

[2] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *J. Comput. Syst. Sci.*, 57(1):74–93, 1998.

[3] A. Andersson and S. Nilsson. A new efficient radix sort. In *Proceedings of the 35th Annual Symposium of Foundations of Computer Science*, pages 714–721. IEEE Computer Society, 1994.

[4] A. Andersson and S. Nilsson. Implementing radixsort. *ACM Journal of Experimental Algorithms*, 3:7, 1998.

[5] J. Bentley and M.D. McIlroy. Engineering a sort function. *Software – Practice and Experience*, 23(11):1249–1265, November 1993.

[6] G.S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithms. *ACM Journal of Experimental Algorithmics*, 12(Article 2.2), 2007.

[7] J.-C. Chen. Building a new sort function for a c library. *Softw. Pract. Exper.*, 34(8):777–795, 2004.

[8] C.R. Cook and D.J. Kim. Best sorting algorithms for nearly sorted lists. *Communications of the ACM*, 23:620–624, 1980.

[9] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge Massachusetts, 1990.

[10] M. Durand. Asymptotic analysis of an optimized quicksort algorithm. *Inf. Process. Lett.*, 85(2):73–77, 2003.

[11] S. Edelkamp and P. Stiegeler. Implementing heapsort with $(n \log n - 0.9n)$ and quicksort with $(n \log n + 0.2n)$ comparisons. *ACM Journal of Experimental Algorithms*, 7:5, 2002.

[12] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *Computing Surveys*, 24:441–476, 1992.

[13] V. Estivill-Castro and D. Wood. Foundations for faster external sorting. In *Fourteenth Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 414–425, Madras, India, 1994. Springer Verlag Lecture Notes in Computer Science 880.

[14] J.A. Fill and S. Janson. The number of bit comparisons used by quicksort: an average-case analysis. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 300–307, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[15] M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[16] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures, 2nd edition.* Addison-Wesley Publishing Co., Don Mills, Ontario, 1991.

[17] Y. Han. Improved fast integer sorting in linear space. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 793–796, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[18] Y. Han. Deterministic sorting in o(nlog logn) time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004.

[19] C.A.R. Hoare. Algorithm 64, Quicksort. *Communications of the ACM*, 4(7):321, July 1961.

[20] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Science Press, Inc., Woodland Hill, CA., 1976.

[21] D.E. Knuth. *The Art of Computer Programming, Vol.3: Sorting and Searching*. Addison-Wesley Publishing Co., Reading, MA, 1973.

[22] P.-A. Larson. External sorting: Run formation revisited. *IEEE Transaction on Knowledge and Data Engineering*, 15(4):961–972, July/August 2003.

[23] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34:318–325, 1985.

[24] K. Mehlhorn. *Data Structures and Algorithms, Vol 1: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin/Heidelberg, 1984.

[25] N. Rahman and R. Raman. Analysing cache effects in distribution sorting. *ACM Journal of Experimental Algorithms*, 5:14, 2000.

[26] N. Rahman and R. Raman. Adapting radix sort to the memory hierarchy. *ACM Journal of Experimental Algorithms*, 6:7, 2001.

[27] B. Salzberg. *File Structures: An Analytic Approach*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.

[28] B. Salzberg. Merging sorted runs using large main memory. *Acta Informatica*, 27:195–215, 1989.

[29] R. Sedgewick. *Quicksort*. Garland Publishing Inc., New York and London, 1980.

[30] R. Sedgewick. *Algorithms*. Addison-Wesley Publishing Co., Reading, MA, 1983.

[31] R. Sinha and J. Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *ACM Journal of Experimental Algorithms*, 9, 2004.

[32] R. Sinha, J. Zobel, and D. Ring. Cache-efficient string sorting using copying. *ACM Journal of Experimental Algorithms*, 11, 2006.

[33] R. Wickremesinghe, L. Arge, J. S. Chase, and J. S. Vitter. Efficient sorting using registers and caches. *ACM Journal of Experimental Algorithms*, 7:9, 2002.

[34] L.Q. Zheng and P. A. Larson. Speeding up external mergesort. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):322–332, 1996.

Further Information

For the detailed arguments that provide theoretical lower bounds for comparison-based sorting algorithms, the reader may consult early works on algorithms [1, 24]. These books also include a description of sorting strings by *Bucket Sort* in time proportional to the total length of the strings.

Sedgewick's book on algorithms [30] provides illustrative descriptions of *Radix Sorts*. Other interesting algorithms, for example, *Linear Probing Sort*, usually have very good performance in practice although they are more complicated to program. They can be reviewed in Gonnet and Baeza-Yates' handbook [16].

We have omitted here algorithms for external sorting with tapes, since they are now very rare. However, the reader may consult classical sources [16, 21].

For more information on the advances in fusion trees and radix sort, as well as data structures and algorithms, the reader may wish to review the *ACM Journal of Experimental Algorithms*, the *Proceedings of the IEEE Symposium on Foundations of Computer Science* (FOCS), the *Proceedings of the annual ACM-SIAM Symposium on Discrete algorithms* (SODA), or the *Proceedings of the ACM Symposium on the Theory of Computing*.