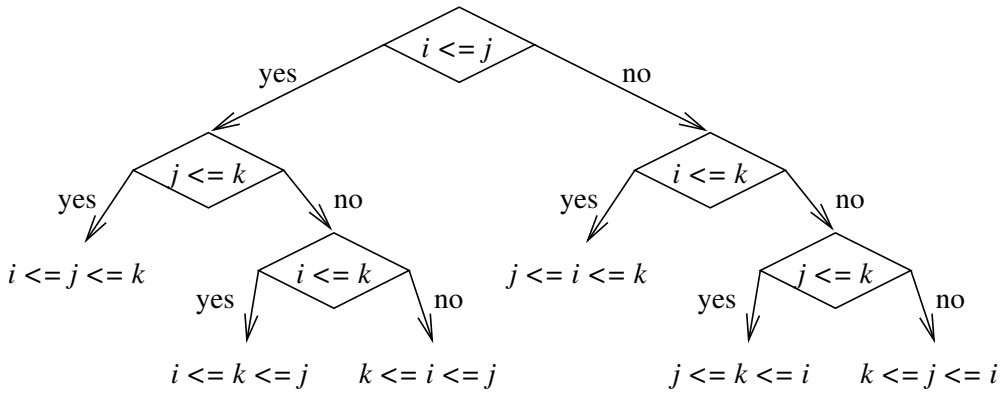
to have all the information needed to sort an arbitrary collection of items. Then we can see how well particular sorting algorithms compare against that theoretical lower bound.

In general, questions of this kind are rather hard, because of the need to consider *all* possible algorithms. In fact, for some problems, optimal lower bounds are not yet known. One important example is the so-called *Travelling Salesman Problem* (TSP), for which all algorithms, which are known to give the correct shortest route solution, are extremely inefficient in the worst case (many to the extent of being useless in practice). In these cases, one generally has to relax the problem to find solutions which are *probably approximately correct*. For the TSP, it is still an open problem whether there exists a feasible algorithm that is guaranteed to give the exact shortest route.

For sorting algorithms based on comparisons, however, it turns out that a tight lower bound does exist. Clearly, even if the given collection of items is already sorted, we must still check all the items one at a time to see whether they are in the correct order. Thus, the lower bound must be at least n, the number of items to be sorted, since we need at least n steps to examine every element. If we already knew a sorting algorithm that works in n steps, then we could stop looking for a better algorithm: n would be both a lower bound and an upper bound to the minimum number of steps, and hence an *exact bound*. However, as we shall shortly see, no algorithm can actually take fewer than $O(n\log_2 n)$ comparisons in the worst case. If, in addition, we can design an algorithm that works in $O(n\log_2 n)$ steps, then we will have obtained an exact bound. We shall start by demonstrating that every algorithm needs at least $O(n\log_2 n)$ comparisons.

To begin with, let us assume that we only have three items, i, j, and k. If we have found that $i \leq j$ and $j \leq k$, then we know that the sorted order is: i, j, k. So it took us two comparisons to find this out. In some cases, however, it is clear that we will need as many as three comparisons. For example, if the first two comparisons tell us that $i > j$ and $j \leq k$, then we know that j is the smallest of the three items, but we cannot say from this information how i and k relate. A third comparison is needed. So what is the *average* and *worst* number of comparisons that are needed? This can best be determined from the so-called *decision tree*, where we keep track of the information gathered so far and count the number of comparisons needed. The decision tree for the three item example we were discussing is:



So what can we deduce from this about the general case? The decision tree will obviously always be a binary tree. It is also clear that its *height* will tell us how many comparisons will be needed in the worst case, and that the average length of a path from the root to a leaf will give us the average number of comparisons required. The leaves of the decision tree are

all the possible *outcomes*. These are given by the different possible orders we can have on n items, so we are asking how many ways there are of arranging n items. The first item can be any of the n items, the second can be any of the remaining $n-1$ items, and so forth, so their total number is $n(n-1)(n-2)\cdots 3\cdot 2\cdot 1 = n!$. Thus we want to know the height h of a binary tree that can accommodate as many as $n!$ leaves. The number of leaves of a tree of height h is at most 2^h, so we want to find h such that

$$2^h \geq n! \quad \text{or} \quad h \geq \log_2(n!)$$

There are numerous approximate expressions that have been derived for $\log_2(n!)$ for large n, but they all have the same dominant term, namely $n\log_2 n$. (Remember that, when talking about time complexity, we ignore any sub-dominant terns and constant factors.) Hence, no sorting algorithm based on comparing items can have a better average or worst case performance than using a number of comparisons that is approximately $n\log_2 n$ for large n. It remains to be seen whether this $O(n\log_2 n)$ complexity can actually be achieved in practice. To do this, we would have to exhibit at least one algorithm with this performance behaviour (and convince ourselves that it really does have this behaviour). In fact, we shall shortly see that there are several algorithms with this behaviour.

We shall proceed now by looking in turn at a number of sorting algorithms of increasing sophistication, that involve the various strategies listed above. The way they work depends on what kind of data structure contains the items we wish to sort. We start with approaches that work with simple arrays, and then move on to using more complex data structures that lead to more efficient algorithms.

9.4 Bubble Sort

Bubble Sort follows the *exchange sort* approach. It is very easy to implement, but tends to be particularly slow to run. Assume we have array `a` of size `n` that we wish to sort. Bubble Sort starts by comparing `a[n-1]` with `a[n-2]` and swaps them if they are in the wrong order. It then compares `a[n-2]` and `a[n-3]` and swaps those if need be, and so on. This means that once it reaches `a[0]`, the smallest entry will be in the correct place. It then starts from the back again, comparing pairs of 'neighbours', but leaving the zeroth entry alone (which is known to be correct). After it has reached the front again, the second-smallest entry will be in place. It keeps making 'passes' over the array until it is sorted. More generally, at the ith stage Bubble Sort compares neighbouring entries 'from the back', swapping them as needed. The item with the lowest index that is compared to its right neighbour is `a[i-1]`. After the ith stage, the entries `a[0],...,a[i-1]` are in their final position.

At this point it is worth introducing a simple 'test-case' of size `n = 4` to demonstrate how the various sorting algorithms work:

4	1	3	2

Bubble Sort starts by comparing `a[3]=2` with `a[2]=3`. Since they are not in order, it swaps them, giving | 4 | 1 | 2 | 3 |. It then compares `a[2]=2` with `a[1]=1`. Since those are in order, it leaves them where they are. Then it compares `a[1]=1` with `a[0]=4`, and those are not in order once again, so they have to be swapped. We get | 1 | 4 | 2 | 3 |. Note that the smallest entry has reached its final place. This will *always* happen after Bubble Sort has done its first 'pass' over the array.

Now that the algorithm has reached the zeroth entry, it starts at the back again, comparing `a[3]=3` with `a[2]=2`. These entries are in order, so nothing happens. (Note that these numbers have been compared before – there is nothing in Bubble Sort that prevents it from repeating comparisons, which is why it tends to be pretty slow!) Then it compares `a[2]=2` and `a[1]=4`. These are not in order, so they have to be swapped, giving | 1 | 2 | 4 | 3 |. Since we already know that `a[0]` contains the smallest item, we leave it alone, and the second pass is finished. Note that now the second-smallest entry is in place, too.

The algorithm now starts the third and final pass, comparing `a[3]=3` and `a[2]=4`. Again these are out of order and have to be swapped, giving | 1 | 2 | 3 | 4 |. Since it is known that `a[0]` and `a[1]` contain the correct items already, they are not touched. Furthermore, the third-smallest item is in place now, which means that the fourth-smallest *has to be* correct, too. Thus the whole array is sorted.

It is now clear that Bubble Sort can be implemented as follows:

```
for ( i = 1 ; i < n ; i++ )
   for ( j = n-1 ; j >= i ; j-- )
      if ( a[j] < a[j-1] )
         swap a[j] and a[j-1]
```

The outer loop goes over all `n` $-$ `1` positions that may still need to be swapped to the left, and the inner loop goes from the end of the array back to that position.

As is usual for comparison-based sorting algorithms, the time complexity will be measured by counting the number of comparisons that are being made. The outer loop is carried out $n-1$ times. The inner loop is carried out $(n-1)-(i-1) = n-i$ times. So the number of comparisons is the same in each case, namely

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=i}^{n-1} 1 &= \sum_{i=1}^{n-1}(n-i) \\ &= (n-1)+(n-2)+\cdots+1 \\ &= \frac{n(n-1)}{2}.\end{aligned}$$

Thus the worst case and average case number of comparisons are both proportional to n^2, and hence the average and worst case time complexities are $O(n^2)$.

9.5 Insertion Sort

Insertion Sort is (not surprisingly) a form of *insertion sorting*. It starts by treating the first entry `a[0]` as an already sorted array, then checks the second entry `a[1]` and compares it with the first. If they are in the wrong order, it swaps the two. That leaves `a[0],a[1]` sorted. Then it takes the third entry and positions it in the right place, leaving `a[0],a[1],a[2]` sorted, and so on. More generally, at the beginning of the ith stage, Insertion Sort has the entries `a[0],..., a[i-1]` sorted and inserts `a[i]`, giving sorted entries `a[0],...,a[i]`.

For the example starting array | 4 | 1 | 3 | 2 |, Insertion Sort starts by considering `a[0]=4` as sorted, then picks up `a[1]` and 'inserts it' into the already sorted array, increasing the size of it by 1. Since `a[1]=1` is smaller than `a[0]=4`, it has to be inserted in the zeroth slot,

but that slot is holding a value already. So we first move `a[0]` 'up' one slot into `a[1]` (care being taken to remember `a[1]` first!), and then we can move the old `a[1]` to `a[0]`, giving

1	4	3	2

.

At the next step, the algorithm treats `a[0],a[1]` as an already sorted array and tries to insert `a[2]=3`. This value obviously has to fit between `a[0]=1` and `a[1]=4`. This is achieved by moving `a[1]` 'up' one slot to `a[2]` (the value of which we assume we have remembered), allowing us to move the current value into `a[1]`, giving

1	3	4	2

.

Finally, `a[3]=2` has to be inserted into the sorted array `a[0],...,a[2]`. Since `a[2]=4` is bigger than 2, it is moved 'up' one slot, and the same happens for `a[1]=3`. Comparison with `a[0]=1` shows that `a[1]` was the slot we were looking for, giving

1	2	3	4

.

The general algorithm for Insertion Sort can therefore be written:

```
for ( i = 1 ; i < n ; i++ ) {
   for( j = i ; j > 0 ; j-- )
      if ( a[j] < a[j-1] )
         swap a[j] and a[j-1]
      else break
}
```

The outer loop goes over the `n - 1` items to be inserted, and the inner loop takes each next item and swaps it back through the currently sorted portion till it reaches its correct position. However, this typically involves swapping each next item many times to get it into its right position, so it is more efficient to store each next item in a temporary variable `t` and only insert it into its correct position when that has been found and its content moved:

```
for ( i = 1 ; i < n ; i++ ) {
   j = i
   t = a[j]
   while ( j > 0 && t < a[j-1] ) {
      a[j] = a[j-1]
      j--
   }
   a[j] = t
}
```

The outer loop again goes over `n-1` items, and the inner loop goes back through the currently sorted portion till it finds the correct position for the next item to be inserted.

The time complexity is again taken to be the number of comparisons performed. The outer loop is always carried out $n-1$ times. How many times the inner loop is carried out depends on the items being sorted. In the worst case, it will be carried out i times; on average, it will be half that often. Hence the number of comparison in the worst case is:

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=1}^{i} 1 &= \sum_{i=1}^{n-1} i \\ &= 1+2+\cdots+(n-1) \\ &= \frac{n(n-1)}{2};\end{aligned}$$

and in the average case it is half that, namely $n(n-1)/4$. Thus average and worst case number of steps for of Insertion Sort are both proportional to n^2, and hence the average and worst case time complexities are both $O(n^2)$.

9.6 Selection Sort

Selection Sort is (not surprisingly) a form of *selection sorting*. It first finds the smallest item and puts it into `a[0]` by exchanging it with whichever item is in that position already. Then it finds the second-smallest item and exchanges it with the item in `a[1]`. It continues this way until the whole array is sorted. More generally, at the ith stage, Selection Sort finds the ith-smallest item and swaps it with the item in `a[i-1]`. Obviously there is no need to check for the ith-smallest item in the first $i-1$ elements of the array.

For the example starting array | 4 | 1 | 3 | 2 |, Selection Sort first finds the smallest item in the whole array, which is `a[1]=1`, and swaps this value with that in `a[0]` giving | 1 | 4 | 3 | 2 |. Then, for the second step, it finds the smallest item in the reduced array `a[1],a[2],a[3]`, that is `a[3]=2`, and swaps that into `a[1]`, giving | 1 | 2 | 3 | 4 |. Finally, it finds the smallest of the reduced array `a[2],a[3]`, that is `a[2]=3`, and swaps that into `a[2]`, or recognizes that a swap is not needed, giving | 1 | 2 | 3 | 4 |.

The general algorithm for Selection Sort can be written:

```
for ( i = 0 ; i < n-1 ; i++ ) {
   k = i
   for ( j = i+1 ; j < n ; j++ )
      if ( a[j] < a[k] )
         k = j
   swap a[i] and a[k]
}
```

The outer loop goes over the first `n-1` positions to be filled, and the inner loop goes through the currently unsorted portion to find the next smallest item to fill the next position. Note that, unlike with Bubble Sort and Insertion Sort, there is exactly one swap for each iteration of the outer loop,

The time complexity is again the number of comparisons carried out. The outer loop is carried out $n-1$ times. In the inner loop, which is carried out $(n-1)-i = n-1-i$ times, one comparison occurs. Hence the total number of comparisons is:

$$\begin{aligned}\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 &= \sum_{i=0}^{n-2}(n-1-i)\\ &= (n-1)+\cdots+2+1\\ &= \frac{n(n-1)}{2}.\end{aligned}$$

Therefore the number of comparisons for Selection Sort is proportional to n^2, in the worst case as well as in the average case, and hence the average and worst case time complexities are both $O(n^2)$.

Note that Bubblesort, Insertion Sort and Selection Sort all involve two nested for loops over $O(n)$ items, so it is easy to see that their overall complexities will be $O(n^2)$ without having to compute the exact number of comparisons.

9.7 Comparison of $O(n^2)$ sorting algorithms

We have now seen three different array based sorting algorithms, all based on different sorting strategies, and all with $O(n^2)$ time complexity. So one might imagine that it does not make much difference which of these algorithms is used. However, in practice, it can actually make a big difference which algorithm is chosen. The following table shows the measured running times of the three algorithms applied to arrays of integers of the size given in the top row:

Algorithm	128	256	512	1024	O1024	R1024	2048
Bubble Sort	54	221	881	3621	1285	5627	14497
Insertion Sort	15	69	276	1137	6	2200	4536
Selection Sort	12	45	164	634	643	833	2497

Here O1024 denotes an array with 1024 entries which are already sorted, and R1024 is an array which is sorted in the *reverse* order, that is, from biggest to smallest. All the other arrays were filled randomly. Warning: tables of measurements like this are *always* dependent on the random ordering used, the implementation of the programming language involved, and on the machine it was run on, and so will never be exactly the same.

So where exactly do these differences come from? For a start, Selection Sort always makes $n(n-1)/2$ comparisons, but carries out *at most* $n-1$ swaps. Each swap requires three assignments and takes, in fact, more time than a comparison. Bubble Sort, on the other hand, does a lot of swaps. Insertion Sort does particularly well on data which is sorted already – in such a case, it only makes $n-1$ comparisons. It is worth bearing this in mind for some applications, because if only a few entries are out of place, Insertion Sort can be very quick. These comparisons serve to show that complexity considerations can be rather delicate, and require good judgement concerning what operations to count. It is often a good idea to run some experiments to test the theoretical considerations and see whether any simplifications made are realistic in practice. For instance, we have assumed here that all comparisons cost the same, but that may not be true for big numbers or strings of characters.

What exactly to count when considering the complexity of a particular algorithm is always a judgement call. You will have to gain experience before you feel comfortable with making such decisions yourself. Furthermore, when you want to improve the performance of an algorithm, you may want to determine the biggest user of computing *resources* and focus on improving that. Something else to be aware of when making these calculations is that it is not a bad idea to keep track of any constant factors, in particular those that go with the dominating sub-term. In the above examples, the factor applied to the dominating sub-term, namely n^2, varies. It is $1/2$ for the average case of Bubble Sort and Selection Sort, but only $1/4$ for Insertion Sort. It is certainly useful to know that an algorithm that is linear will perform better than a quadratic one *provided the size of the problem is large enough*, but if you know that your problem has a size of, say, at most 100, then a complexity of $(1/20)n^2$ will be preferable to one of $20n$. Or if you know that your program is only ever used on fairly small samples, then using the simplest algorithm you can find might be beneficial overall – it is easier to program, and there is not a lot of compute time to be saved.

Finally, the above numbers give you some idea why, for program designers, the general rule is to *never* use Bubble Sort. It is certainly easy to program, but that is about all it has going for it. You are better off avoiding it altogether.

9.8 Sorting algorithm stability

One often wants to sort items which might have identical keys (e.g., ages in years) in such a way that items with identical keys are kept in their original order, particularly if the items have already been sorted according to a different criteria (e.g., alphabetical). So, if we denote the original order of an array of items by subscripts, we want the subscripts to end up in order for each set of items with identical keys. For example, if we start out with the array $[5_1, 4_2, 6_3, 5_4, 6_5, 7_6, 5_7, 2_8, 9_9]$, it should be sorted to $[2_8, 4_2, 5_1, 5_4, 5_7, 6_3, 6_5, 7_6, 9_9]$ and not to $[2_8, 4_2, 5_4, 5_1, 5_7, 6_3, 6_5, 7_6, 9_9]$. Sorting algorithms which satisfy this useful property are said to be *stable*.

The easiest way to determine whether a given algorithm is stable is to consider whether the algorithm can ever swap identical items past each other. In this way, the stability of the sorting algorithms studied so far can easily be established:

Bubble Sort This is stable because no item is swapped past another unless they are in the wrong order. So items with identical keys will have their original order preserved.

Insertion Sort This is stable because no item is swapped past another unless it has a smaller key. So items with identical keys will have their original order preserved.

Selection Sort This is not stable, because there is nothing to stop an item being swapped past another item that has an identical key. For example, the array $[2_1, 2_2, 1_3]$ would be sorted to $[1_3, 2_2, 2_1]$ which has items 2_2 and 2_1 in the wrong order.

The issue of sorting stability needs to be considered when developing more complex sorting algorithms. Often there are stable and non-stable versions of the algorithms, and one has to consider whether the extra cost of maintaining stability is worth the effort.

9.9 Treesort

Let us now consider a way of implementing an *insertion sorting* algorithm using a data structure better suited to the problem. The idea here, which we have already seen before, involves inserting the items to be sorted into an initially empty binary search tree. Then, when all items have been inserted, we know that we can traverse the binary search tree to visit all the items in the right order. This sorting algorithm is called *Treesort*, and for the basic version, we require that all the search keys be different.

Obviously, the tree must be kept balanced in order to minimize the number of comparisons, since that depends on the height of the tree. For a balanced tree that is $O(\log_2 n)$. If the tree is not kept balanced, it will be more than that, and potentially $O(n)$.

Treesort can be difficult to compare with other sorting algorithms, since it returns a tree, rather than an array, as the sorted data structure. It should be chosen if it is desirable to have the items stored in a binary search tree anyway. This is usually the case if items are frequently deleted or inserted, since a binary search tree allows these operations to be implemented efficiently, with time complexity $O(\log_2 n)$ per item. Moreover, as we have seen before, searching for items is also efficient, again with time complexity $O(\log_2 n)$.

Even if we have an array of items to start with, and want to finish with a sorted array, we can still use Treesort. However, to output the sorted items into the original array, we will need another procedure `fillArray(tree t, array a, int j)` to traverse the tree `t` and fill the array `a`. That is easiest done by passing and returning an index `j` that keeps track of the next array position to be filled. This results in the complete Treesort algorithm:

```
treeSort(array a) {
   t = EmptyTree
   for ( i = 0 ; i < size(a) ; i++ )
      t = insert(a[i],t)
   fillArray(t,a,0)
}

fillArray(tree t, array a, int j) {
   if ( not isEmpty(t) ) {
      j = fillArray(left(t),a,j)
      a[j++] = root(t)
      j = fillArray(right(t),a,j)
   }
   return j
}
```

which assumes that `a` is a pointer to the array location and that its elements can be accessed and updated given that and the relevant array index.

Since there are n items to insert into the tree, and each insertion has time complexity $O(\log_2 n)$, Treesort has an overall average time complexity of $O(n\log_2 n)$. So, we already have one algorithm that achieves the theoretical best average case time complexity of $O(n\log_2 n)$. Note, however, that if the tree is not kept balanced while the items are being inserted, and the items are already sorted, the height of the tree and number of comparisons per insertion will be $O(n)$, leading to a worst case time complexity of $O(n^2)$, which is no better than the simpler array-based algorithms we have already considered.

Exercise: We have assumed so far that the items stored in a Binary Search Tree must not contain any duplicates. Find the simplest ways to relax that restriction and determine how the choice of approach affects the *stability* of the associated Treesort algorithm.

9.10 Heapsort

We now consider another way of implementing a *selection sorting* algorithm using a more efficient data structure we have already studied. The underlying idea here is that it would help if we could pre-arrange the data so that selecting the smallest/biggest entry becomes easier. For that, remember the idea of a *priority queue* discussed earlier. We can take the value of each item to be its priority and then queue the items accordingly. Then, if we remove the item with the highest priority at each step we can fill an array in order 'from the rear', starting with the biggest item.

Priority queues can be implemented in a number of different ways, and we have already studied a straightforward implementation using *binary heap trees* in Chapter 8. However, there may be a better way, so it is worth considering the other possibilities.

An obvious way of implementing them would be using a sorted array, so that the entry with the highest priority appears in `a[n]`. Removing this item would be very simple, but inserting a new item would always involve finding the right position and shifting a number of items to the right to make room for it. For example, inserting a `3` into the queue $[1, 2, 4]$:

n	0	1	2	3	4	5
`a[n]`	1	2	4			

n	0	1	2	3	4	5
`a[n]`	1	2		4		

n	0	1	2	3	4	5
`a[n]`	1	2	3	4		

That kind of item insertion is effectively insertion sort and clearly inefficient in general, of $O(n)$ complexity rather than $O(\log_2 n)$ with a binary heap tree.

Another approach would be to use an unsorted array. In this case, a new item would be inserted by just putting it into `a[n+1]`, but to delete the entry with the highest priority would involve having to find it first. Then, after that, the last item would have to be swapped into the gap, or all items with a higher index 'shifted down'. Again, that kind of item deletion is clearly inefficient in general, of $O(n)$ complexity rather than $O(\log_2 n)$ with a heap tree.

Thus, of those three representations, only one is of use in carrying out the above idea efficiently. An unsorted array is what we started from, so that is not any help, and ordering the array is what we are trying to achieve, so heaps are the way forward.

To make use of binary heap trees, we first have to take the unsorted array and re-arrange it so that it satisfies the heap tree priority ordering. We have already studied the *heapify* algorithm which can do that with $O(n)$ time complexity. Then we need to extract the sorted array from it. In the heap tree, the item with the highest priority, that is the largest item, is always in `a[1]`. In the sorted array, it should be in the last position `a[n]`. If we simply swap the two, we will have that item at the right position of the array, and also have begun the standard procedure of removing the root of the heap-tree, since `a[n]` is precisely the item that would be moved into the root position at the next step. Since `a[n]` now contains the correct item, we will never have to look at it again. Instead, we just take the items `a[1],...,a[n-1]` and bring them back into a heap-tree form using the *bubble down* procedure on the new root, which we know to have complexity $O(\log_2 n)$.

Now the second largest item is in position `a[1]`, and its final position should be `a[n-1]`, so we now swap these two items. Then we rearrange `a[1],...,a[n-2]` back into a heap tree using the bubble down procedure on the new root. And so on.

When the ith step has been completed, the items `a[n-i+1],...,a[n]` will have the correct entries, and there will be a heap tree for the items `a[1],...,a[n-i]`. Note that the size, and therefore the height, of the heap tree decreases at each step. As a part of the ith step, we have to bubble down the new root. This will take at most twice as many comparisons as the height of the original heap tree, which is $\log_2 n$. So overall there are $n - 1$ steps, with at most $2\log_2 n$ comparisons, totalling $2(n-1)\log_2 n$. The number of comparisons will actually be less than that, because the number of bubble down steps will usually be less than the full height of the tree, but usually not much less, so the time complexity is still $O(n\log_2 n)$.

The full Heapsort algorithm can thus be written in a very simple form, using the bubble down and heapify procedures we already have from Chapter 8. First `heapify` converts the

array into a binary heap tree, and then the for loop moves each successive root one item at a time into the correct position in the sorted array:

```
heapSort(array a, int n) {
   heapify(a,n)
   for( j = n ; j > 1 ; j-- ) {
      swap a[1] and a[j]
      bubbleDown(1,a,j-1)
   }
}
```

It is clear from the swap step that the order of identical items can easily be reversed, so there is no way to render the Heapsort algorithm *stable*.

The average and worst-case time complexities of the entire Heapsort algorithm are given by the *sum* of two complexity functions, first that of `heapify` rearranging the original unsorted array into a heap tree which is $O(n)$, and then that of making the sorted array out of the heap tree which is $O(n\log_2 n)$ coming from the $O(n)$ bubble-downs each of which has $O(\log_2 n)$ complexity. Thus the overall average and worst-case complexities are both $O(n\log_2 n)$, and we now have a sorting algorithm that achieves the theoretical best worst-case time complexity. Using more sophisticated priority queues, such as Binomial or Fibonacci heaps, cannot improve on this because they have the same delete time complexity.

A useful feature of Heapsort is that if only the largest $m \ll n$ items need to be found and sorted, rather than all n, the complexity of the second stage is only $O(m\log_2 n)$, which can easily be less than $O(n)$ and thus render the whole algorithm only $O(n)$.

9.11 Divide and conquer algorithms

All the sorting algorithms considered so far work on the whole set of items together. Instead, *divide and conquer* algorithms recursively split the sorting problem into more manageable sub-problems. The idea is that it will usually be easier to sort many smaller collections of items than one big one, and sorting single items is trivial. So we repeatedly split the given collection into two smaller parts until we reach the 'base case' of one-item collections, which require no effort to sort, and then merge them back together again. There are two main approaches for doing this:

Assuming we are working on an array `a` of size n with entries `a[0],...,a[n-1]`, then the obvious approach is to simply split the set of indices. That is, we split the array at item $n/2$ and consider the two sub-arrays `a[0],...,a[(n-1)/2]` and `a[(n+1)/2],...,a[n-1]`. This method has the advantage that the splitting of the collection into two collections of equal (or nearly equal) size at each stage is easy. However, the two sorted arrays that result from each split have to be *merged* together carefully to maintain the ordering. This is the underlying idea for a sorting algorithm called *mergesort*.

Another approach would be to split the array in such a way that, at each stage, all the items in the first collection are no bigger than all the items in the second collection. The splitting here is obviously more complex, but all we have to do to put the pieces back together again at each stage is to take the first sorted array followed by the second sorted array. This is the underlying idea for a sorting algorithm called *Quicksort*.

We shall now look in detail at how these two approaches work in practice.

9.12 Quicksort

The general idea here is to repeatedly split (or partition) the given array in such a way that all the items in the first sub-array are smaller than all the items in the second sub-array, and then concatenate all the sub-arrays to give the sorted full array.

How to partition. The important question is how to perform this kind of splitting most efficiently. If the array is very simple, for example [4,3,7,8,1,6], then a good split would be to put all the items smaller than 5 into one part, giving [4,3,1], and all the items bigger than or equal to 5 into the other, that is [7,8,6]. Indeed, moving all items with a smaller key than some given value into one sub-array, and all entries with a bigger or equal key into the other sub-array is the standard *Quicksort* strategy. The value that defines the split is called the *pivot*. However, it is not obvious what is the best way to choose the pivot value.

One situation that we absolutely have to avoid is splitting the array into an *empty* sub-array and the whole array again. If we do this, the algorithm will not just perform badly, it will not even terminate. However, if the pivot is chosen to be an item in the array, and the pivot is kept in between and separate from both sub-arrays, then the sub-arrays being sorted at each recursion will always be at least one item shorter than the previous array, and the algorithm is guaranteed to terminate.

Thus, it proves convenient to split the array at each stage into the sub-array of values smaller than or equal to some chosen pivot item, followed by that chosen pivot item, followed by the sub-array of values greater than or equal to the chosen pivot item. Moreover, to save space, we do not actually split the array into smaller arrays. Instead, we simply *rearrange* the whole array to reflect the splitting. We say that we *partition* the array, and the *Quicksort* algorithm is then applied to the sub-arrays of this partitioned array.

In order for the algorithm to be called recursively, to sort ever smaller parts of the original array, we need to tell it *which part* of the array is currently under consideration. Therefore, Quicksort is called giving the lowest index (`left`) and highest index (`right`) of the sub-array it must work on. Thus the algorithm takes the form:

```
quicksort(array a, int left, int right) {
   if ( left < right ) {
      pivotindex = partition(a,left,right)
      quicksort(a,left,pivotindex-1)
      quicksort(a,pivotindex+1,right)
   }
}
```

for which the initial call would be `quicksort(a,0,n-1)` and the array `a` at the end is sorted. The crucial part of this is clearly the `partition`(`a`, `left`, `right`) procedure that rearranges the array so that it can be split around an appropriate pivot `a`[`pivotindex`].

If we were to split off only one item at a time, Quicksort would have n recursive calls, where n is the number of items in the array. If, on the other hand, we halve the array at each stage, it would only need $\log_2 n$ recursive calls. This can be made clear by drawing a binary tree whose nodes are labelled by the sub-arrays that have been split off at each stage, and measuring its height. Ideally then, we would like to get two sub-arrays of roughly equal size (namely half of the given array) since that is the most efficient way of doing this. Of course, that depends on choosing a good pivot.

Choosing the pivot. If we get the pivot 'just right' (e.g., choosing 5 in the above example), then the split will be as even as possible. Unfortunately, there is no quick guaranteed way of finding the optimal pivot. If the keys are integers, one could take the average value of all the keys, but that requires visiting *all* the entries to sample their key, adding considerable overhead to the algorithm, and if the keys are more complicated, such as strings, you cannot do this at all. More importantly, it would not necessarily give a pivot that is a value in the array. Some sensible *heuristic* pivot choice strategies are:

- Use a random number generator to produce an index `k` and then use `a[k]`.
- Take a key from 'the middle' of the array, that is `a[(n-1)/2]`.
- Take a small sample (e.g., 3 or 5 items) and take the 'middle' key of those.

Note that one should *never* simply choose the first or last key in the array as the pivot, because if the array is almost sorted already, that will lead to the particularly bad choice mentioned above, and this situation is actually quite common in practice.

Since there are so many reasonable possibilities, and they are all fairly straightforward, we will not give a specific implementation for any of these pivot choosing strategies, but just assume that we have a `choosePivot(a,left,right)` procedure that returns the index of the pivot for a particular sub-array (rather than the pivot value itself).

The partitioning. In order to carry out the partitioning within the given array, some thought is required as to how this may be best achieved. This is more easily demonstrated by an example than put into words. For a change, we will consider an array of strings, namely the programming languages: [c, fortran, java, ada, pascal, basic, haskell, ocaml]. The ordering we choose is the standard lexicographic one, and let the chosen pivot be "fortran".

We will use markers | to denote a partition of the array. To the left of the left marker, there will be items we know to have a key smaller than or equal to the pivot. To the right of the right marker, there will be items we know to have a key bigger than or equal to the pivot. In the middle, there will be the items we have not yet considered. Note that this algorithm proceeds to investigate the items in the array *from two sides*.

We begin by swapping the pivot value to the end of the array where it can easily be kept separate from the sub-array creation process, so we have the array: [|c, ocaml, java, ada, pascal, basic, haskell | fortran]. Starting from the left, we find "c" is less than "fortran", so we move the left marker one step to the right to give [c | ocaml, java, ada, pascal, basic, haskell | fortran]. Now "ocaml" is greater than "fortran", so we stop on the left and proceed from the right instead, without moving the left marker. We then find "haskell" is bigger than "fortran", so we move the right marker to the left by one, giving [c | ocaml, java, ada, pascal, basic, | haskell, fortran]. Now "basic" is smaller than "fortran", so we have two keys, "ocaml" and "basic", which are 'on the wrong side'. We therefore swap them, which allows us to move both the left and the right marker one step further towards the middle. This brings us to [c, basic | java, ada, pascal | ocaml, haskell, fortran]. Now we proceed from the left once again, but "java" is bigger than "fortran", so we stop there and switch to the right. Then "pascal" is bigger than "fortran", so we move the right marker again. We then find "ada", which is smaller than the pivot, so we stop. We have now got [c, basic | java, ada, | pascal, ocaml, haskell, fortran]. As before, we want to swap "java" and "ada", which leaves the left and the right markers in the same place: [c, basic, ada, java | | pascal, ocaml, haskell, fortran], so we

stop. Finally, we swap the pivot back from the last position into the position immediately after the markers to give [c, basic, ada, java | | fortran, ocaml, haskell, pascal].

Since we obviously cannot have the marker indices 'between' array entries, we will assume the left marker is on the left of `a[leftmark]` and the right marker is to the right of `a[rightmark]`. The markers are therefore 'in the same place' once `rightmark` becomes smaller than `leftmark`, which is when we stop. If we assume that the keys are integers, we can write the partitioning procedure, that needs to return the final pivot position, as:

```
partition(array a, int left, int right) {
   pivotindex = choosePivot(a, left, right)
   pivot = a[pivotindex]
   swap a[pivotindex] and a[right]
   leftmark = left
   rightmark = right - 1
   while (leftmark <= rightmark) {
      while (leftmark <= rightmark  and  a[leftmark] <= pivot)
          leftmark++
      while (leftmark <= rightmark  and  a[rightmark] >= pivot)
          rightmark--
      if (leftmark < rightmark)
         swap a[leftmark++] and a[rightmark--]
   }
   swap a[leftmark] and a[right]
   return leftmark
}
```

This achieves a partitioning that ends with the same items in the array, but in a different order, with all items to the left of the returned pivot position smaller or equal to the pivot value, and all items to the right greater or equal to the pivot value.

Note that this algorithm doesn't require any extra memory – it just swaps the items in the original array. However, the swapping of items means the algorithm is not *stable*. To render quicksort stable, the partitioning must be done in such a way that the order of identical items can never be reversed. A conceptually simple approach that does this, but requires more memory and copying, is to simply go systematically through the whole array, re-filling the array `a` with items less than or equal to the pivot, and filling a second array `b` with items greater or equal to the pivot, and finally copying the array `b` into the end of `a`:

```
partition2(array a, int left, int right) {
   create new array b of size right-left+1
   pivotindex = choosePivot(a, left, right)
   pivot = a[pivotindex]
   acount = left
   bcount = 1
   for ( i = left ; i <= right ; i++ ) {
      if ( i == pivotindex )
         b[0] = a[i]
      else if ( a[i] < pivot || (a[i] == pivot && i < pivotindex) )
```

```
            a[acount++] = a[i]
         else
            b[bcount++] = a[i]
      }
      for ( i = 0 ; i < bcount ; i++ )
         a[acount++] = b[i]
      return right-bcount+1
   }
```

Like the first partition procedure, this also achieves a partitioning with the same items in the array, but in a different order, with all items to the left of the returned pivot position smaller or equal to the pivot value, and all items to the right greater or equal to the pivot value.

Complexity of Quicksort. Once again we shall determine complexity based on the number of comparisons performed. The partitioning step compares each of n items against the pivot, and therefore has complexity $O(n)$. Clearly, some partition and pivot choice algorithms are less efficient than others, like `partition2` involving more copying of items than `partition`, but that does not generally affect the overall complexity class.

In the *worst case*, when an array is partitioned, we have one empty sub-array. If this happens at each step, we apply the partitioning method to arrays of size n, then $n-1$, then $n-2$, until we reach 1. Those complexity functions then add up to

$$n + (n-1) + (n-2) + \cdots 2 + 1 = n(n+1)/2$$

Ignoring the constant factor and the non-dominant term $n/2$, this shows that, in the worst case, the number of comparisons performed by Quicksort is $O(n^2)$.

In the *best case*, whenever we partition the array, the resulting sub-arrays will differ in size by at most one. Then we have n comparisons in the first case, two lots of $\lfloor n/2 \rfloor$ comparisons for the two sub-arrays, four times $\lfloor n/4 \rfloor$, eight times $\lfloor n/8 \rfloor$, and so on, down to $2^{\log_2 n - 1}$ times $\lfloor n/2^{\log_2 n - 1} \rfloor = \lfloor 2 \rfloor$. That gives the total number of comparisons as

$$n + 2^1\lfloor n/2^1 \rfloor + 2^2\lfloor n/2^2 \rfloor + 2^3\lfloor n/2^3 \rfloor + \cdots + 2^{\log_2 n - 1}\lfloor n/2^{\log_2 n - 1} \rfloor \approx n\log_2 n$$

which matches the theoretical best possible time complexity of $O(n\log_2 n)$.

More interesting and important is how well Quicksort does in the *average case*. However, that is much harder to analyze exactly. The strategy for choosing a pivot at each stage affects that, though as long as it avoids the problems outlined above, that does not change the complexity class. It also makes a difference whether there can be duplicate values, but again that doesn't change the complexity class. In the end, *all* reasonable variations involve comparing $O(n)$ items against a pivot, for each of $O(\log_2 n)$ recursions, so the total number of comparisons, and hence the overall time complexity, in the average case is $O(n\log_2 n)$.

Like Heapsort, when only the largest $m \ll n$ items need to be found and sorted, rather than all n, Quicksort can be modified to result in reduced time complexity. In this case, only the first sub-array needs to be processed at each stage, until the sub-array sizes exceed m. In that situation, for the best case, the total number of comparisons is reduced to

$$n + 1\lfloor n/2^1 \rfloor + 1\lfloor n/2^2 \rfloor + 1\lfloor n/2^3 \rfloor + \cdots + m\log_2 m \approx 2n.$$

rendering the time complexity of the whole modified algorithm only $O(n)$. For the average case, the computation is again more difficult, but as long as the key problems outlined above are avoided, the average-case complexity of this special case is also $O(n)$.

Improving Quicksort. It is always worthwhile spending some time optimizing the strategy for defining the pivot, since the particular problem in question might well allow for a more refined approach. Generally, the pivot will be better if more items are sampled before it is being chosen. For example, one could check several randomly chosen items and take the 'middle' one of those, the so called *median*. Note that in order to find the median of all the items, without sorting them first, we would end up having to make n^2 comparisons, so we cannot do that without making Quicksort unattractively slow.

Quicksort is rarely the most suitable algorithm if the problem size is small. The reason for this is all the overheads from the recursion (e.g., storing all the return addresses and formal parameters). Hence once the sub-problem become 'small' (a size of 16 is often suggested in the literature), Quicksort should stop calling itself and instead sort the remaining sub-arrays using a simpler algorithm such as Selection Sort.

9.13 Mergesort

The other divide and conquer sorting strategy based on repeatedly splitting the array of items into two sub-arrays, mentioned in Section 9.11, is called *mergesort*. This simply splits the array at each stage into its first and last half, without any reordering of the items in it. However, that will obviously not result in a set of sorted sub-arrays that we can just append to each other at the end. So mergesort needs another procedure `merge` that merges two sorted sub-arrays into another sorted array. As with binary search in Section 4.4, integer variables `left` and `right` can be used to refer to the lower and upper index of the relevant array, and `mid` refers to the end of its left sub-array. Thus a suitable mergesort algorithm is:

```
mergesort(array a, int left, int right) {
   if ( left < right ) {
      mid = (left + right) / 2
      mergesort(a, left, mid)
      mergesort(a, mid+1, right)
      merge(a, left, mid, right)
   }
}
```

Note that it would be relatively simple to modify this mergesort algorithm to operate on linked lists (of known length) rather than arrays. To 'split' such a list into two, all one has to do is set the pointer of the $\lfloor n/2 \rfloor$th list entry to null, and use the previously-pointed-to next entry as the head of the new second list. Of course, care needs to be taken to keep the list size information intact, and effort is required to find the crucial pointer for each split.

The `merge` algorithm. The principle of merging two sorted collections (whether they be lists, arrays, or something else) is quite simple: Since they are sorted, it is clear that the smallest item overall must be either the smallest item in the first collection or the smallest item in the second collection. Let us assume it is the smallest key in the first collection. Now the second smallest item overall must be either the second-smallest item in the first collection, or the smallest item in the second collection, and so on. In other words, we just work through both collections and at each stage, the 'next' item is the current item in either the first or the second collection.

The implementation will be quite different, however, depending on which data structure we are using. When arrays are used, it is actually necessary for the `merge` algorithm to create a new array to hold the result of the operation at least temporarily. In contrast, when using linked lists, it would be possible for `merge` to work by just changing the reference to the next node. This does make for somewhat more confusing code, however.

For arrays, a suitable *merge* algorithm would start by creating a new array `b` to store the results, then repeatedly add the next smallest item into it until one sub-array is finished, then copy the remainder of the unfinished sub-array, and finally copy `b` back into `a`:

```
merge(array a, int left, int mid, int right) {
   create new array b of size right-left+1
   bcount = 0
   lcount = left
   rcount = mid+1
   while ( (lcount <= mid) and (rcount <= right) ) {
      if ( a[lcount] <= a[rcount] )
         b[bcount++] = a[lcount++]
      else
         b[bcount++] = a[rcount++]
   }
   if ( lcount > mid )
      while ( rcount <= right )
         b[bcount++] = a[rcount++]
   else
      while ( lcount <= mid )
         b[bcount++] = a[lcount++]
   for ( bcount = 0 ; bcount < right-left+1 ; bcount++ )
      a[left+bcount] = b[bcount]
}
```

It is instructive to compare this with the `partition2` algorithm for Quicksort to see exactly where the two sort algorithms differ. As with `partition2`, the merge algorithm never swaps identical items past each other, and the splitting does not change the ordering at all, so the whole Mergesort algorithm is *stable*.

Complexity of Mergesort. The total number of comparisons needed at each recursion level of mergesort is the number of items needing merging which is $O(n)$, and the number of recursions needed to get to the single item level is $O(\log_2 n)$, so the total number of comparisons and its time complexity are $O(n\log_2 n)$. This holds for the worst case as well as the average case. Like Quicksort, it is possible to speed up mergesort by abandoning the recursive algorithm when the sizes of the sub-collections become small. For arrays, 16 would once again be a suitable size to switch to an algorithm like Selection Sort.

Note that, with Mergesort, for the special case when only the largest/smallest $m \ll n$ items need to be found and sorted, rather than all n, there is no way to reduce the time complexity in the way it was possible with Heapsort and Quicksort. This is because the ordering of the required items only emerges at the very last stage after the large majority of the comparisons have already been carried out.

9.14 Summary of comparison-based sorting algorithms

The following table summarizes the key properties of all the comparison-based sorting algorithms we have considered:

Sorting Algorithm	Strategy employed	Objects manipulated	Worst case complexity	Average case complexity	Stable
Bubble Sort	Exchange	arrays	$O(n^2)$	$O(n^2)$	Yes
Selection Sort	Selection	arrays	$O(n^2)$	$O(n^2)$	No
Insertion Sort	Insertion	arrays/lists	$O(n^2)$	$O(n^2)$	Yes
Treesort	Insertion	trees/lists	$O(n^2)$	$O(n\log_2 n)$	Yes
Heapsort	Selection	arrays	$O(n\log_2 n)$	$O(n\log_2 n)$	No
Quicksort	D & C	arrays	$O(n^2)$	$O(n\log_2 n)$	Maybe
Mergesort	D & C	arrays/lists	$O(n\log_2 n)$	$O(n\log_2 n)$	Yes

To see what the time complexities mean in practice, the following table compares the typical run times of those of the above algorithms that operate directly on arrays:

Algorithm	128	256	512	1024	O1024	R1024	2048
Bubble Sort	54	221	881	3621	1285	5627	14497
Selection Sort	12	45	164	634	643	833	2497
Insertion Sort	15	69	276	1137	6	2200	4536
Heapsort	21	45	103	236	215	249	527
Quicksort	12	27	55	112	1131	1200	230
Quicksort2	6	12	24	57	1115	1191	134
Mergesort	18	36	88	188	166	170	409
Mergesort2	6	22	48	112	94	93	254

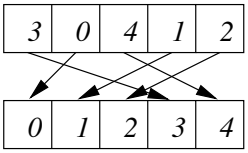
As before, arrays of the stated sizes are filled randomly, except O1024 that denotes an array with 1024 entries which are already sorted, and R1024 that denotes an array which is sorted in the *reverse* order. Quicksort2 and Mergesort2 are algorithms where the recursive procedure is abandoned in favour of Selection Sort once the size of the array falls to 16 or below. It should be emphasized again that these numbers are of limited accuracy, since they vary somewhat depending on machine and language implementation.

What has to be stressed here is that there is no 'best sorting algorithm' in general, but that there are usually good and bad choices of sorting algorithms *for particular circumstances*. It is up to the program designer to make sure that an appropriate one is picked, depending on the properties of the data to be sorted, how it is best stored, whether all the sorted items are required rather than some sub-set, and so on.

9.15 Non-comparison-based sorts

All the above sorting algorithms have been based on comparisons of the items to be sorted, and we have seen that we can't get time complexity better than $O(n\log_2 n)$ with comparison based algorithms. However, in some circumstances it is possible to do better than that with sorting algorithms that are not based on comparisons.

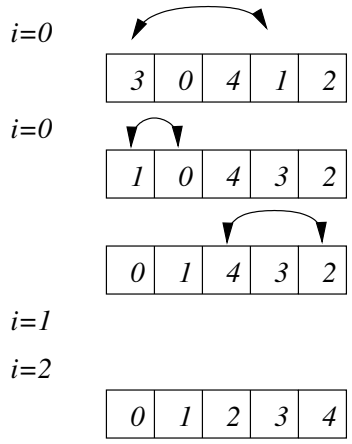
It is always worth thinking about the data that needs to be sorted, and whether comparisons really are required. For example, suppose you know the items to be sorted are the numbers from 0 to $n-1$. How would you sort those? The answer is surprisingly simple. We know that we have n entries in the array and we know *exactly which items should go there and in which order.* This is a very unusual situation as far as general sorting is concerned, yet this kind of thing often comes up in every-day life. For example, when a hotel needs to sort the room keys for its 100 rooms. Rather than employing one of the comparison-based sorting algorithms, in this situation we can do something much simpler. We can simply put the items directly in the appropriate places, using an algorithm such as that as shown in Figure 9.1:




```
create array b of size n
for ( i = 0 ; i < n ; i++ )
    b[a[i]] = a[i]
copy array b into array a
```

Figure 9.1: Simply put the items in the right order using their values.

This algorithm uses a second array `b` to hold the results, which is clearly not very memory efficient, but it is possible to do without that. One can use a series of swaps within array `a` to get the items in the right positions as shown in Figure 9.2:




```
for ( i = 0 ; i < n; i++ ) {
   while ( a[i] != i )
      swap a[a[i]] and a[i]
}
```

Figure 9.2: Swapping the items into the right order without using a new array.

As far as time complexity is concerned, it is obviously not appropriate here to count the number of comparisons. Instead, it is the number of swaps or copies that is important. The algorithm of Figure 9.1 performs n copies to fill array `b` and then another n to return the result to array a, so the overall time complexity is $O(n)$. The time complexity of the algorithm of Figure 9.2 looks worse than it really is. This algorithm performs at most $n-1$ swaps, since one item, namely `a[a[i]]` is always swapped into its final position. So at worst, this has time complexity $O(n)$ too.

This example should make it clear that in particular situations, sorting might be performed by much simpler (and quicker) means than the standard comparison sorts, though most realistic situations will not be quite as simple as the case here. Once again, it is the responsibility of the program designer to take this possibility into account.

9.16 Bin, Bucket, Radix Sorts

Bin, Bucket, and Radix Sorts are all names for essentially the same non-comparison-based sorting algorithm that works well when the items are labelled by small sets of values. For example, suppose you are given a number of dates, by day and month, and need to sort them into order. One way of doing this would be to create a queue for each day, and place the items (dates) one at a time into the right queue according to their day (without sorting them further). Then form one big queue out of these, by concatenating all the day queues starting with day 1 and continuing up to day 31. Then for the second phase, create a queue for each month, and place the dates into the right queues *in the order that they appear in the queue created by the first phase*. Again form a big queue by concatenating these month queues in order. This final queue is sorted in the intended order.

This may seem surprising at first sight, so let us consider a simple example:

[25/12, 28/08, 29/05, 01/05, 24/04, 03/01, 04/01, 25/04, 26/12, 26/04, 05/01, 20/04].

We first create and fill queues for the days as follows:

01: [01/05]
03: [03/01]
04: [04/01]
05: [05/01]
20: [20/04]
24: [24/04]
25: [25/12, 25/04]
26: [26/12, 26/04]
28: [28/08]
29: [29/05]

The empty queues are not shown – there is no need to create queues before we hit an item that belongs to them. Then concatenation of the queues gives:

[01/05, 03/01, 04/01, 05/01, 20/04, 24/04, 25/12, 25/04, 26/12, 26/04,28/08, 29/05].

Next we create and fill queues for the months that are present, giving:

01: [03/01, 04/01, 05/01]
04: [20/04, 24/04, 25/04, 26/04]
05: [01/05, 29/05]
08: [28/08]
12: [25/12, 26/12]

Finally, concatenating all these queues gives the items in the required order:

[03/01, 04/01, 05/01, 20/04, 24/04, 25/04, 26/04, 01/05, 29/05, 28/08, 25/12, 26/12].

This is called *Two-phase Radix Sorting*, since there are clearly two phases to it.

The extension of this idea to give a general sorting algorithm should be obvious: For each phase, create an ordered set of queues corresponding to the possible values, then add each item in the order they appear to the end of the relevant queue, and finally concatenate the the queues in order. Repeat this process for each sorting criterion. The crucial additional detail is that the queuing phases must be performed in the order of the significance of each criteria, with the *least significant* criteria first.

For example, if you know that your items to be sorted are all (at most) two-digit integers, you can use Radix Sort to sort them. First create and fill queues for the last digit, concatenate, then create and fill queues for the first digit, and concatenate to leave the items in sorted order. Similarly, if you know that your keys are all strings consisting of three characters, you can again apply Radix Sort. You would first queue according to the third character, then the second, and finally the first, giving a *Three phase* Radix Sort.

Note that *at no point*, does the the algorithm actually *compare* any items at all. This kind of algorithm makes use of the fact that for each phase the items are *from a strictly restricted set*, or, in other words, the items are of a particular form which is known *a priori*. The complexity class of this algorithm is $O(n)$, since at every phase, each item is dealt with precisely once, and the number of phases is assumed to be small and constant. If the *restricted sets* are small, the number of operations involved in finding the right queue for each item and placing it at the end of it will be small, but this could become significant if the sets are large. The concatenation of the queues will involve some overheads, of course, but these will be small if the sets are small and linked lists, rather than arrays, are used. One has to be careful, however, because if the total number of operations for each item exceeds $\log_2 n$, then the overall complexity is likely to be greater than the $O(n\log_2 n)$ complexity of the more efficient comparison-based algorithms. Also, if the *restricted sets* are not known in advance, and potentially large, the overheads of finding and sorting them could render Radix sort worse than using a comparison-based approach. Once again, it is the responsibility of the program designer to decide whether a given problem can be solved more efficiently with Radix Sort rather than a comparison-based sort.