29 Fast Sorting

1 2-way partitioning

Later we will make a very fast method for sorting an array, but first we tackle a problem that will turn out to be an important stepping stone. Given `int[] b = new int[1000]`, duly initialised, we want to rearrange `b` so that all the negative elements occur before all the naturals. For example, if `b` at the outset is, say

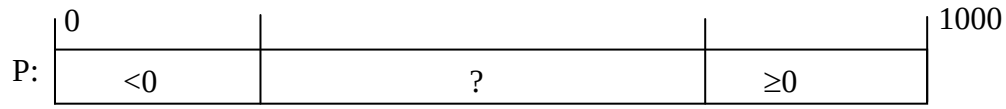
3	0	-1	-2	4	.....	–4	0

then finally we want something like:

-2	-1	-4	.....	4	0	0	3

The order of the negatives doesn't matter, just as long as they all occur in a block on the left, and similarly we only ask of the naturals that they all end up in a block on the right. This may not seem a terribly interesting problem, but put that aside for a short while.

To design the program we do a thought experiment. We imagine that we have written the program and it is executing. Then we freeze-frame it in mid-flight and ask ourselves: what does the array look like? The following seems likely:



In words, we imagine the array as consisting of three partitions: `b[0..j-1]` contains only negatives, `b[k..999]` contains only naturals, and `b[j..k-1]` contains a mixture. Other scenarios may be possible, but we'll pursue this one which we call P (for "picture"). P will be the *invariant* of the loop we're trying to write – "invariant" because we aim to keep it true throughout the execution of the loop. The range of the indices `j` and `k` is $0 \le j \le k \le 1000$.

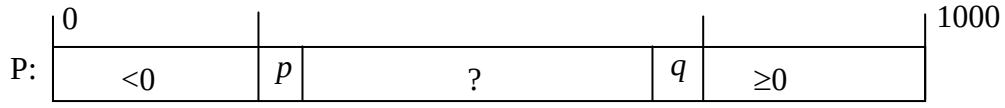
Initially, the central partition (marked ?) is co-extensive with the entire array, because at the outset no elements have been examined. Hence initially we will have `j` = 0 and `k` = 1000. In short, the following assignments establish P:

```
int j = 0; int k= 1000;
```

The program should finish when the central partition is empty, i.e. when `j` = `k`, because then every element must lie in the leftmost or the rightmost partition. So the program will have the shape

```
int j = 0; int k = 1000;
while (j!=k) {
    "make progress, staying faithful to P"
}
```

What is "progress"? Initially `j` and `k` are far apart, and at the end they are equal. Hence we make progress if each iteration of the loop either increments `j` or decrements `k` (or both). This means, in essence, that we have to move an element from the central partition to one of the others, according to whether it's negative or natural. The obvious element to examine is either *p* or *q* as indicated below (*p* and *q* denote the values of `b[j]` and `b[k-1]`, respectively):



(i) To increment `j`, we require $p < 0$ if we are to maintain P.
(ii) To decrement `k`, we need $q \ge 0$ if we are to maintain P.

The problem remains of how to proceed when $p \ge 0$ and $q < 0$, and here it takes a flash of insight. Observe that if we swap *p* and *q*, we create a situation in which `b[j]`<0 holds (because `b[j]` is now *q* which we are assuming is less than 0) and `b[k-1]`≥0 holds (because `b[k-1]` is now *p* which we are assuming is at least 0). Hence we remain faithful to P if we swap *p* and *q* and

both increment `j` and decrement `k` (progress only requires that we do one of these, but we may as well do both):

```
int t = b[j]; b[j] = b[k-1]; b[k-1] = t;
j++; k--;
```

The complete code is:

```
int j = 0; int k = 1000;
while (j!=k) {
    if (b[j]<0) j++;
    else if (b[k-1]>=0) k--;
    else {
        int t = b[j]; b[j] = b[k-1]; b[k-1] = t;
        j++;  k --;
    }
}
```

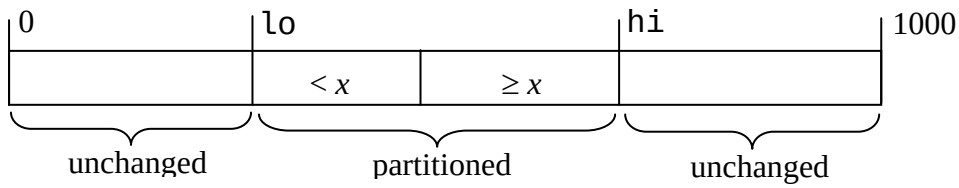
The problem we have solved is called *partitioning* an array. Elements are moved into the left or right partitions according to whether or not they are less than 0; we call 0 the *pivot* element of the partition.

We can do some trivial tidying up. The code accesses `b[k-1]` in three places, and subsequently decrements `k`. It is just a little cheaper and a little tidier if we decrement `k` first and access `b[k]` rather than `b[k-1]`:

```
int j = 0; int k = 1000;
while (j!=k) {
    if (b[j]<0) j++;
    else {
        k--;
        if (b[k]<0) {
            int t = b[j];  b[j] = b[k];  b[k] = t;
            j++;
        }
    }
}
```

Two generalisations

We make two simple generalisations. Firstly, we prefer to partition not the entire array but a segment delimited by indices `lo` and `hi`, say, i.e. we partition `b[lo..hi-1]` where $0 \le lo \le hi \le 1000$. If `lo`=0 and `hi`=1000 then in effect we partition the entire array. Secondly, instead of partitioning with 0 as pivot, we use an arbitrary integer *x*. In summary, we want to rearrange `b[lo..hi-1]` so that it finally looks like:



The changes to the code are trivial:

```
// Partition b[lo..hi-1] with pivot x
int j = lo; int k = hi;
while (j!=k) {
    if (b[j]<x) j++;
    else {
        k--;
        if (b[k]<x) {
            int t = b[j];  b[j] = b[k];  b[k] = t;
            j++;
        }
    }
}
```

2 Quicksort

Quicksort is a very fast algorithm for sorting an array. It relies on the following property of lists:

> Let *s1* and *s2* be lists (of integers, say) such that
> all elements in *s1* are no greater than all elements in *s2*.
> If *s1*and *s2* are sorted, so is the combined list comprised of *s1* followed by *s2*.

For example, in the following two lists observe that no value in the left hand list exceeds any value in the right hand list:

6 4 5 4 8 7 6 9 8

First sort the left hand list without examining or changing the right hand one:

4 4 5 6 8 7 6 9 8

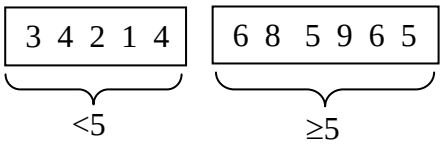
Now sort the right hand list without examining or changing the left hand one:

4 4 5 6 **6 7 8 8 9**

and observe that the entire line is sorted. Indeed, the same property holds for any number of lists: if each list contains only values that do not exceed the values in the list to its right, then the concatenation of all the lists is sorted if the constituent lists are sorted individually. We can exploit this property to make a recursive sorting algorithm based on partitioning. Suppose we want to sort the sequence:

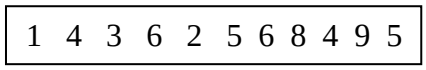
5 4 3 6 2 9 6 8 4 5 1

We first partition it using as pivot the first element, say (here 5). This results in a sequence consisting of two partitions:

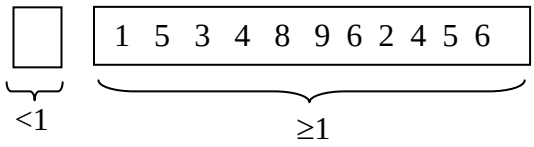


The order of the numbers in each partition is not significant. Now we need only sort each partition. How? Recursively, of course! For the recursion to work we have to deal with small sequences explicitly. But that's easy, because sequences of length 0 or 1 are sorted for free.

There is a snag, however. Suppose the smallest element happened to occur in the first position:

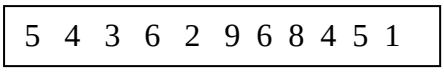


Partitioning with pivot 1 yields an empty left partition:

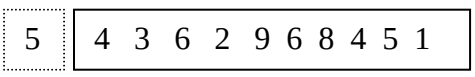


(Again, the order of the elements is not significant.) We cannot now profitably sort the second partition recursively because it is not smaller the original – the recursion is not guaranteed to terminate. In fact, if the array consisted of nothing but 1's, the program would be guaranteed never to terminate (although, paradoxically, the array is sorted!).

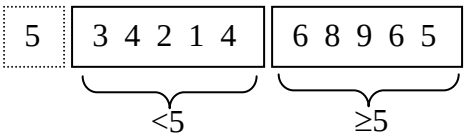
The way out is as follows. First, choose the first element as pivot as proposed, but exclude it from the partitioning. For example, suppose we are to sort the following sequence:



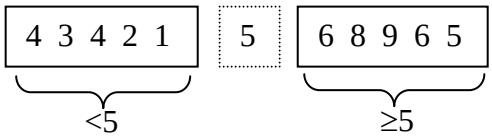
We choose the first element (i.e. 5) as pivot, with the intention of partitioning only the boxed elements to its right:



After partitioning, we arrive at, say:

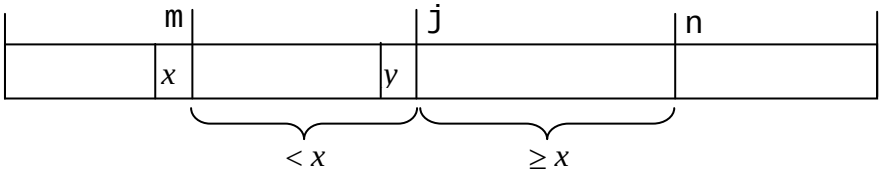


Again, the order of the numbers in each partition is not significant. Now, swap the initial

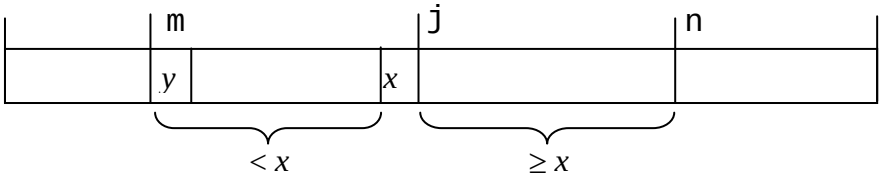


element (i.e. 5) with the final element in the first partition, resulting in:
(as a side-effect, the left partition appears to move one slot to the left). Observe that we have rearranged the original sequence into two partitions, separated by a lone value in the middle (the pivot). The pivot exceeds all the values in the left partition, and is less than or equal to the values in the right partition. Hence, to sort the sequence it only remains to sort the left and right partitions in turn – the pivot is already in its final position and need take no further part in proceedings. This time we can be sure that each partition is smaller than the original, because the original contained the pivot which is absent from both partitions.

In summary, we can sort a segment `b[m..n-1]` of array `b` as follows. If `b[m..n-1]` has 0 or 1 elements, i.e. if `n-m`<2, then it is sorted without further work. If `b[m..n-1]` has at least two elements, we first partition `b[m+1..n-1]` with pivot `b[m]` resulting in:



Above, *x* denotes the value of `b[m]`, and *y* the (final) value of `b[j-1]`. Now swap the contents of `b[m]` and `b[j-1]`:



It only remains to recursively sort `b[m..j-2]` and `b[j..n-1]`. The code follows.

```
static void sort(int[] b) {  // Sort b
    quickSort(b, 0, b.length);
}

static void quickSort(int[] b, int m, int n) {
// Sort b[m..n-1], 0<=m<=n<=b.length
    if (n-m >= 2) {
```

```
            int x = b[m];
            // Partition b[m+1..n-1] with pivot x
            int j = m+1;  int k = n;
            while (j!=k) {
                if (b[j]<x) j++;
                else {
                    k--;
                    if (b[k]<x) {
                        int t = b[j];  b[j] = b[k];  b[k] = t;
                        j++;
                    }
                }
            }
            // swap b[m] (= x) & b[j-1]
            b[m] = b[j-1]; b[j-1] = x;
            // sort b[m..j-2] and b[j..n-1]
            quickSort(b, m, j-1);  quickSort(b, j, n);
        }
    }
```

Quicksort is very efficient indeed, so much so it will sort large arrays in only a second or two in comparison to the many hours that selection sort takes. In some situations it may perform as poorly as selection sort, but these are few and far between (oddly enough, it is more likely to perform poorly when the array is nearly sorted to begin with). It is the preferred sorting methods in industrial practice. However, for sorting small arrays in non-critical software components, selection sort is acceptable and is often used because it is simpler to code.

3 Quicksort: reducing memory demands ADVANCED

`quickSort()`may fail when sorting large arrays because of excessive memory demands. Each recursive invocation leads to the creation of a clone, and clones occupy memory. `quickSort()` can create huge numbers of clones. You might like to confirm this by sorting increasingly larger arrays until the program fails. (If necessary you can encourage bad behaviour by sorting an array whose elements are already sorted except that the first element is larger than all others – see below.) With minor but non-trivial amendments, we can drastically reduce the maximum number of clones that can be in existence at any one time, to the extent that for all practical purposes we eliminate any danger of running out of memory.

The first recursive call

Let us observe how `quickSort()` behaves on an array containing ten integers as follows:

```
9  0  1  2  3  4  5  6  7  8
```

Observe that the array is sorted except for the first element which is larger than all others. When `quickSort(b,0,10)` begins execution, it chooses `b[0]` – the largest value – as

pivot. Partitioning will result in a right partition that is empty and a left partition as underlined:

```
9  0  1  2  3  4  5  6  7  8
```

(If you need to be convinced of this, examine the code and you will see that when the pivot is the largest element, the partitioning loop increments `j` on every iteration, and `j` is in effect the inter-partition border.) After the pivot is swapped with the final element in the left partition, the array looks like:

```
8  0  1  2  3  4  5  6  7  9
```

Now `quickSort()` is invoked recursively for the first time (`quickSort(b,m,j-1)`, or substituting actual parameter values `quickSort(b,0,9)`). This leads to the simultaneous existence of two copies of `quickSort()` – the original one and the clone. The newly-created clone has the job of sorting the underlined segment above. Observe that the segment has the same pattern as the original array – it is sorted except that the first element is larger than all others in the segment. Hence the pivot will again be the largest value, and partitioning will leave the right segment empty and the left segment as underlined below:

```
8  0  1  2  3  4  5  6  7  9
```

After the pivot is swapped with the final element in the left partition, the array looks like:

```
7  0  1  2  3  4  5  6  8  9
```

Now `quicksort()` is invoked recursively a second time to sort the underlined segment. The original and the first clone are still in existence, and so the new recursive call leads to three copies being in existence simultaneously. Observe that the underlined segment is just two elements shorter than the entire array. Observe also that again it has the largest value in the first position. The pattern of behaviour therefore repeats itself. It should now be evident that as many clones will be created as there are elements in the original array (less one). The clones will occupy more memory than the array of integers being sorted, and may well exhaust all available memory. We have to address this if `quickSort()` is to be capable of sorting large arrays.

The remedy is surprisingly simple. The body of `quickSort()` has two recursive invocations, one immediately after the other:

```
quickSort(b, m, j-1);  quickSort(b, j, n);
```

Clearly, the order of these invocations is insignificant: `quickSort()` still sorts if we recursively sort the right partition followed by the left partition, or vice versa. We can therefore choose to sort the smaller of the two partitions first, i.e. we can replace the preceding pair of statements with

```
if (j-1-m < n-j) { // left partition smallest -- sort first
    quickSort(b, m, j-1); quickSort(b, j, n);
}
else { // right partition smallest -- sort first
```

```
        quickSort(b, j, n); quickSort(b, m, j-1);
    }
```

This turns out to have a remarkable effect on the rate of cloning. Observe that each clone in the scenario above was created by the first of the two recursive calls. In the amended version, the first call creates a clone that sorts the smaller of the partitions, and this can be no larger than half the original segment. Similarly, the next clone is charged with sorting a partition that can be no larger than half the size again, i.e. at most a quarter the size of the entire array. Hence the number of clones that can exist simultaneously as a result of the first recursive call is no more than the number of times we can halve the size of the original array, i.e. $log_2 n$ where n denotes the size of the array. This is extremely satisfactory. For an array of size 1,000,000 the amended code reduces the maximum number of simultaneously live clones generated by the first recursive call from one million to about twenty.

The second recursive call

We tackle the clones arising from the second call. Actually, we propose the radical step of eliminating the second call. Observe in the original version of `quickSort()` that the second call is always the final act carried out by a clone. More precisely, the body of `quickSort()` has the form:

```
if (n-m>2) then {
    ...
    quickSort(b, j, n);
}
```

The clone making the second recursive call continues to exist until the spawned clone returns, even though its work is done. Only when the clone returns does it terminate and release the memory it occupies. This remains true when we introduce the modification described above – the second recursive call occurs in two places, but both are final actions:

```
if (n-m>2) then {
    ...
    if (j-1-m < n-j) {
        ... quickSort(b, j, n);
    }
    else {
        ... quickSort(b, m, j-1);
    }
}
```

Such recursive calls are said to be *tail-recursive*. Tail recursion is expensive because clones lie around occupying memory long after they have done their useful work.

There is a standard if slightly surprising trick for eliminating tail-recursion from procedures:

re-use the existing clone instead of spawning a new one. A clone re-uses itself by explicitly initialising the formal parameters with the arguments of the tail-recursive call, and then proceeding to the start of its body. For a simple example, consider the following method:

```
static void writePos(int[] w, int i) { // print postives in w[i..], 0<=i<=w.length
    if (i<w.length) {
        if (w[i]>0) System.out.print(w[i] + " ");
        writePos(w,i+1);
    }
}
```

The invocation `writePos(w,i+1)` is tail-recursive. To eliminate it we copy the value of `i+1` into `i` (there is no need to assign to w because we are making no change to it) and proceed to the start of the if-statement – just replace the keyword `if` by `while`! The amended program is:

```
static void writePos(int[] w, int i) {
    while (i<w.length) {
        if (w[i]>0) System.out.print(w[i] + " ");
        i = i+1;
    }
}
```

You should be able to convince yourself that the two methods behave identically. We apply the same trick to `quickSort()`. In place of

```
quickSort(b, j, n);
```

we copy the value of `j` into the corresponding formal parameter `m` (there is no need to assign to parameters `b` and `n` because we are making no change to them). Similarly, in place of

```
quickSort(b, m, j-1);
```

(where it is the final action) we copy the value of `j-1` into variable `n` (again, there is no need to assign to parameters `b` and `m`). Then we continue executing at the first statement of the method The improved `quickSort()` is shown below.

```
static void quickSort(int[] b, int m, int n) {
// Sort b[m..n-1], 0<=m<=n<=b.length
    while (n-m >= 2) {  // 'while' in place of 'if' to eliminate tail recursion
        int x = b[m];
        // Partition b[m+1..n-1] with pivot x
        int j = m+1;  int k = n;
        while (j!=k) {
            if (b[j]<x) j++;
            else {
```

```
                k--;
                if (b[k]<x) {
                    int t = b[j];  b[j] = b[k];  b[k] = t;
                    j++;
                }
            }
        }
        // swap b[m] (= x) & b[j-1]
        b[m] = b[j-1]; b[j-1] = x;
        // sort b[m..j-2] and b[j..n-1], shortest first
        if (j-1-m < n-j) { // left partition smallest -- sort first
            quickSort(b, m, j-1);
            m = j; // tail-recursive quickSort(b, j, n) eliminated
        }
        else { // right partition smallest -- sort first
            quickSort(b, j, n);
            n = j-1; // tail-recursive quickSort(b, m, j-1) eliminated
        }
    }
}
```