32 Fast Sorting

1 3-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 zeros, which in turn appear before all the positives. For example, if `b` at the outset is, say

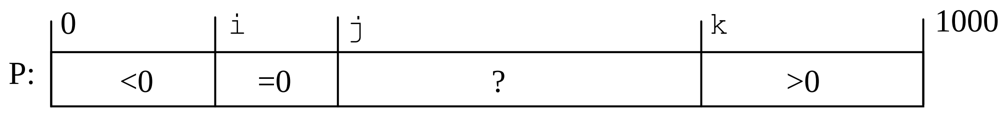
3	0	-1	-2	4	.....	–4	0

then finally we want something like:

-2	-1	-4	.....	0	0	.....	4	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 positives 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 four partitions: `b[0..i-1]` contains only negatives, `b[i..j-1]` contains only zeros, `b[k..999]` contains only positives, 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 `i`, `j` and `k` is 0≤`i`≤`j`≤`k`≤1000.

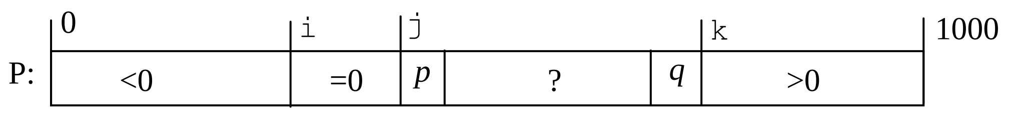
Initially, the partition of unknowns (marked ?) is co-extensive with the entire array (because at the outset no elements have been examined) and the other partitions are empty. Hence initially we will have `i` = 0, `j` = 0 and `k` = 1000. In short, the following assignments establish P:

```
int i = 0; 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 its designated partition. So the program will have the shape

```
int i = 0; 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 partition of unknowns to one of the others, according to whether it's negative, zero, or positive. 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) We can also contrive to increment `j` if *p* < 0, as follows. We first swap *p* with `b[i]`, and increment `i` – this ensures `b[0..i-1]` continues to contain negative only. As `b[j]` now contains 0 we are back at the preceding case and so increment `j`.

The problem remains of how to proceed when *p*>0 and here it takes a little insight. Observe that if now we swap *p* and *q*, we establish `b[k-1]`>0, because `b[k-1]` is now *p* which we are assuming is positive. Hence we have the final case:

(iii) If *p*>0 we maintain P if we swap *p* and *q* and decrement `k`.

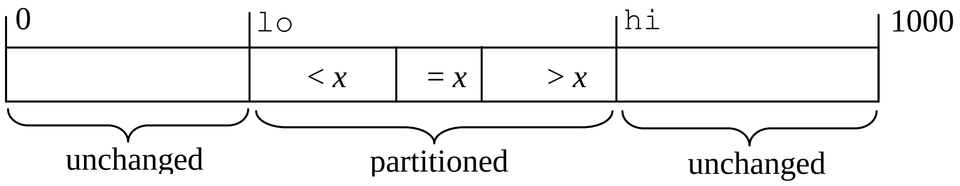
So the complete code is:

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

The problem we have solved is called *partitioning* an array. Elements are moved into some partition according to whether they are less than 0, equal to 0, or greater than 0; we call 0 the *pivot* element of the partition.

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≤`lo`≤`hi`≤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 i = lo; int j = lo; int k = hi;
while (j!=k) {
     if (b[j]==x) j++;
    else if (b[j]<x) {
        int t = b[i]; b[i] = b[j]; b[j] = t;
        i++; j++;
    }
```

```
        else {
            k--;
            int t= b[j]; b[j] = b[k]; b[k] = t;
        }
    }
```

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:

$$\boxed{5\ \ 4\ \ 3\ \ 6\ \ 2\ \ 9\ \ 6\ \ 8\ \ 4\ \ 5\ \ 1}$$

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

$$\underbrace{\boxed{3\ \ 4\ \ 2\ \ 1\ \ 4}}_{<5}\quad \underbrace{\boxed{5\ \ 5}}_{=5}\quad \underbrace{\boxed{6\ \ 8\ \ 9\ \ 6}}_{>5}$$

The order of the numbers in each partition is not significant. By the above property of lists it only remains to sort the left and right partitions. How? Recursively, of course! For the recursion to work the sequences we sort recursively must be smaller than the original sequence, which is clearly the case here. Additionally, we have to deal with small sequences explicitly.

But that's easy, because sequences of length 0 or 1 are sorted for free.

The code follows.

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

static void quickSort(int[] b, int lo, int hi) {
// Sort b[lo..hi-1], 0<=lo<=hi<=b.length
    if (hi-lo >= 2) { // b[lo..hi-1] has at least 2 elements
        int x = b[lo];
        // Partition b[lo..hi-1] with pivot x
        int i = lo; int j = lo; int k = hi;
        while (j!=k) {
            if (b[j]==x) j++;
            else if (b[j]<x) {
                int t = b[i]; b[i] = b[j]; b[j] = t;
                i++; j++;
            }
            else {
                k--;
                int t= b[j]; b[j] = b[k]; b[k] = t;
            }
        }
        // sort b[lo..i-1] and b[j..hi-1]
        quickSort(b, lo, i);  quickSort(b, j, hi);
    }
}
```

Quicksort is very efficient indeed, so much so that 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 Optimisations

Quicksort is a very fast algorithm, but the speed can drop off if partitioning leads to left and right partitions of widely differing sizes (e.g. where the left partition is very small and the right is very large). This happens if the chosen pivot (x in the algorithm above) turns out to be one of the smaller or larger values in the segment being partitioned. Empirically, we find that the chances of this are greatly reduced if we choose as pivot the "middle of three" values, e.g. the middle value from `b[lo]`, `b[hi]`, and `b[(lo+hi)/2)]`. So we might replace `int x = b[lo];` with, for example,

```
int x = b[lo]; int y = b[hi-1] ; int z = b[(lo+hi)/2];
if (y>z) { int t = y; y = z; z = t } // y<=z
if (x<y) x = y; else if (x>z) x = z;
```

Observe also that the order of the two recursive invocations is insignificant: `quickSort()` still works 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 (i-lo<hi-j) { // left partition smallest -- sort first
    quickSort(b, lo, i); quickSort(b, j, hi);
}
else { // right partition smallest -- sort first
    quickSort(b, j, hi); quickSort(b, lo, i);
}
```

This has the effect of ensuring that the number of recursive calls that are alive at any one time (and therefore the number of clones of *quicksort* occupying memory) is kept to a minimum.

Further optimisations are commonly done.