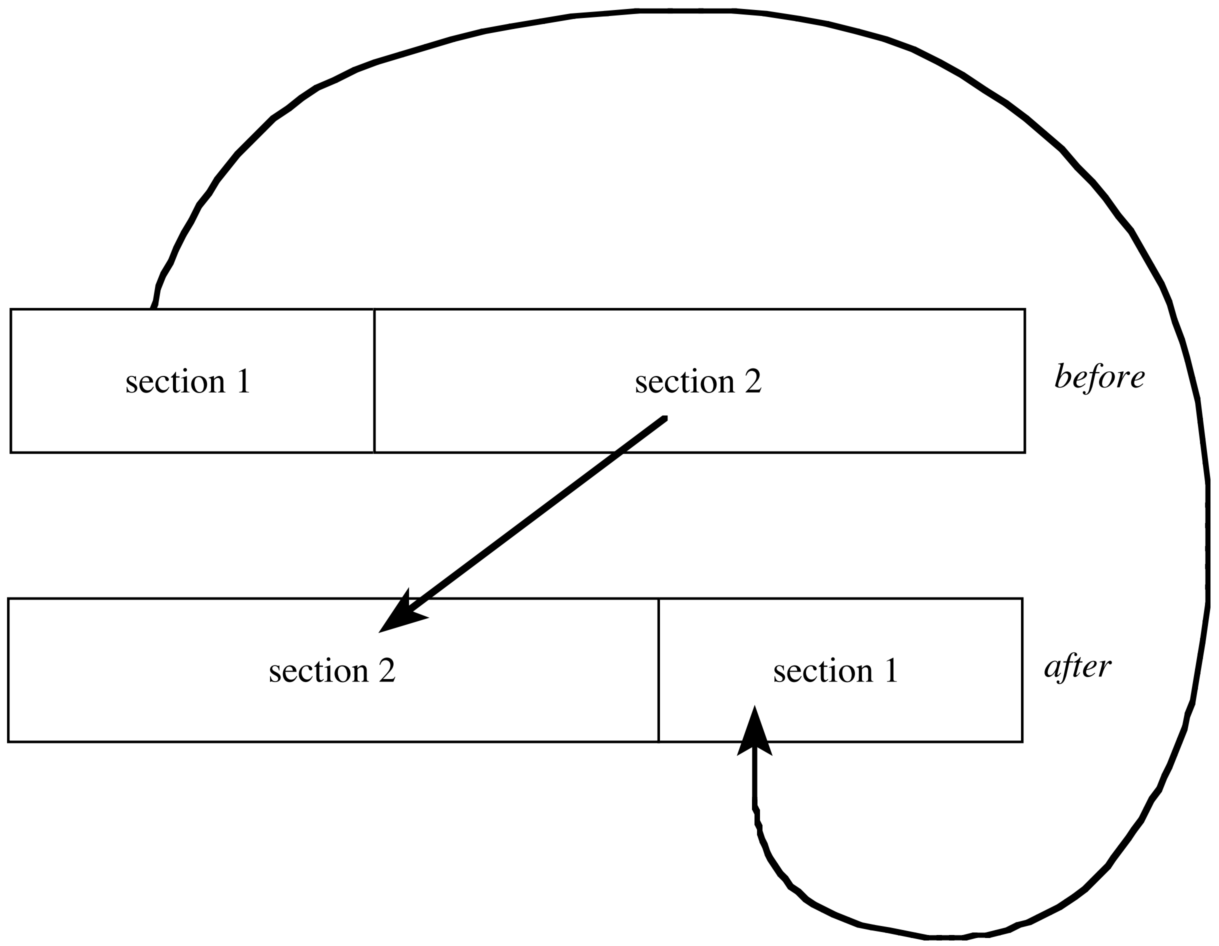
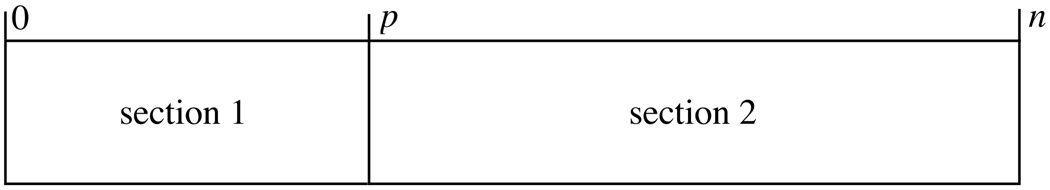
Saving space: a circular shift algorithm.

Consider the problem of shifting the elements of a large array ‘circularly’ by some significant distance.



I emphasise size and distance, because this is fundamentally a problem about *space*, and it becomes interesting only when we have a large problem.

I presume that we have some position p at which the array should be divided:



The predicate calculus specification is straightforward:

$$\forall i \begin{pmatrix} \left(0 \le i \le p \rightarrow A'\left[n - p + i\right] = A\left[i\right]\right) \wedge \\ \left(p \le i < n \rightarrow A'\left[i - p\right] = A\left[i\right]\right) \end{pmatrix}$$

This is not at all a mysterious program to write, if we have a spare array to hand:

```
type[] B = new type(A.length);
for (i=0; i<p; i++) B[A.length-p+i]=A[i];
for (i=p; i<n; i++) B[i-p]=A[i];
for (i=0; i<A.length; i++) A[i]=B[i];
```

This program is $O(N)$ in time and $O(N)$ in space.

But we may not always have that much space.

The space problems can be reduced a little:

```
type[] B = new type(p);
for (i=0; i<p; i++) B[i]=A[i];
for (i=p; i<n; i++) A[i-p]=A[i];
for (i=0; i<p; i++) A[A.length-p+i]=B[i];
```

Now it's $O(p)$ in space, and a little quicker in execution (less copying). We have a better bound on the time: it's $O(N + p)$.

But we still have a program which uses too much space: in the worst case p can be close to N.

We might reduce the worst case space usage to $N/2$, but this program will always have a space problem.

There is a better way.

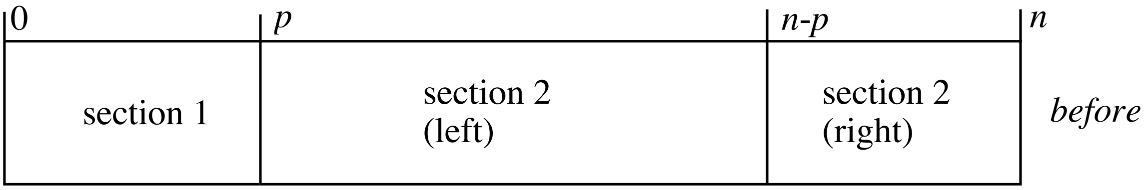
Trading speed for space.

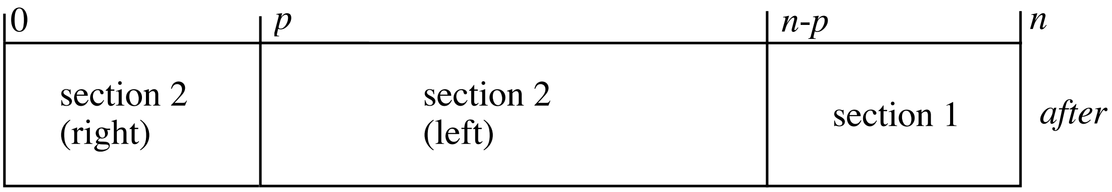
I shall abandon, for a while, the search for a faster solution.

We can save space by moving things around more often.

Suppose that $p \le n - p$: that is, the left section is the smaller.

Then we might begin by swapping $A_{0..p-1}$ with $A_{n-p..n-1}$:

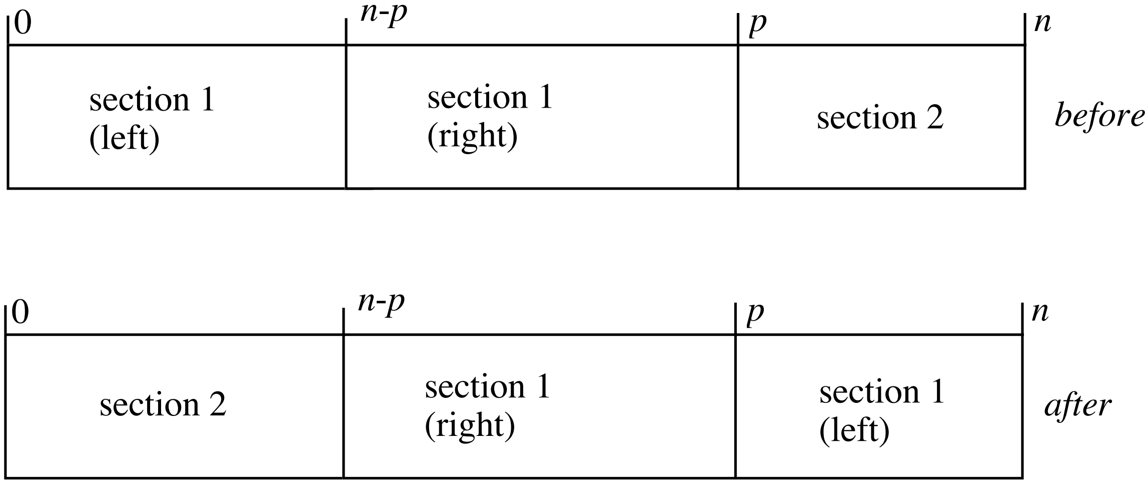




We can do that work using only one extra variable (to implement the swap operation):

```
for (i=0; i<p; i++) A[i]<->A[n-p+i];
```

Now of course if section 2 is the smaller, it won’t work because of overlap: but then we can do something very similar to swap section 2 into place:




```
for (i=0; i<n-p; i++) A[i]<->A[p+i];
```


Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

In either case we have reduced the problem to that of reordering the left and right parts of section 2 (first case) or section 1 (second case) – clearly a case for repetition.

Here's the whole program. Amazingly enough the end-limits *m* and *n* vary, but the boundary *p* always stays in the same place!

```
for (m=0, n=A.length; m!=p && n!=p; ) {
  if (p-m<=n-p) { // shift section 1
    for (i=0; i<p-m; i++) A[i+m]<->A[n-p+i];
    n=n-(p-m); // section 1 is in place
  }
  else { // shift section 2
    for (i=0; i<n-p; i++) A[i+m]<->A[p+i];
    m=m+(n-p); // section 2 is in place
  }
}
```

This program doesn't use much space – $O(1)$, because of the variables i, m, n and p, plus the variable needed for the swaps – but it does a lot too much work.

Each swap takes three assignments; each time we shift a section into place we put a similarly-sized section in the wrong place (unless p divides the interval $m..n$ exactly in half).

We can do better ...

Richard Bornat
Dept of Computer Science
QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

A perfect circular shift.

What should move into A_0? Why, A_p. And what should move into A_p? Why, A_{2p} ... and so on, till we fall off the end of the array because $j \times p > n$.

We don't have to stop there. A_i should be replaced by A_{i+p}, if that's within the array, or else by A_{i+p-n} – because it is a circular shift! And so on, till we get back to A_0 again.

In a complicated multi-way exchange you only need on temporary variable! Here's a bit of program which does the job:

```
type t=A[0];
for (i=0, j=p;
     j!=0;
     i=j, j = j+p<n ? j+p : j+p-n)
  A[i]=A[j];
A[i]=t;
```

This program moves quite a bit of the array around, and it only uses variables *i*, *j* and *t*.

But if p divides n exactly this program won't do the whole problem: if $p = n \div 2$ it only exchanges A_0 and A_p; if $p = n \div 3$ it only rotates A_0, A_p and A_{2p}; and so on.

And if p and n have factors in common this program won't solve the whole problem. In fact if the greatest common divisor of p and n is q then this program will move exactly $n \div q$ things. But then the nice thing is that we can use the same idea, starting again with A_1 ...

Here's the complete program:

```
for (m=0, count=0; count!=n; m++) {
  type t=A[m];
  for (i=m, j=m+p;
       j!=m;
       i=j, j = j+p<n ? j+p : j+p-n, count++)
    A[i]=A[j];
  A[i]=t; count++;
}
```

That program only uses variables i, j and t; it does $O(n)$ assignments; it does the 'extra' assignment `t=A[m]` only $\gcd(n, p)$ times.

If $p = n \div 2$ then it has no advantage over the earlier segment-swapping program, but in all other cases it does a lot less work.

A proof that it works is remarkably difficult ...