Quicksort: the best of sorts?

Weiss calls this ‘the fastest-known sorting algorithm’.

Quicksort takes $O(N^2)$ time in the worst case, but it is easy to make it use time proportional to $N \lg N$ in almost every case. It is claimed to be faster than *mergesort*.

Quicksort can be made to use $O(\lg N)$ space – much better than *mergesort*.

Quicksort is faster than Shellsort (do the tests!) but it uses more space.

Note that we are comparing ‘in-store’ sorting algorithms here. Quite different considerations apply if we have to sort huge ‘on-disc’ collections far too large to fit into memory.

Quicksort is a tricky algorithm, and it’s easy to produce a method that doesn’t work or which is much, much slower than it need be.

Despite this, I recommend that you understand how *quicksort* works. It is a glorious algorithm.

> *If you can’t understand quicksort, Sedgewick says that you should stick with Shellsort.*
>
> *Do you understand Shellsort? Really?*

The ***basic idea*** behind *quicksort* is:
<u>partition; sort one half; sort the other half</u>

Input is a sequence $A_{m..n-1}$:

a if $m + 1 \geq n$ then the sequence is sorted already – do nothing;

b1 if $m + 1 < n$, re-arrange the sequence so that it falls into two halves $A_{m..i-1}$ and $A_{i..n-1}$, swapping elements so that every number in $A_{m..i-1}$ is (≤) each number in $A_{i..n-1}$, but not bothering about the order of things *within* the half-sequences

loosely, the first half-sequence is (≤) the second;

b2 sort the half-sequences;

b3 ***and that's all***.

Step (b1) – called the **partition** *step – re-arranges A into two halves so that everything in the first half is correctly positioned relative to everything in the second half. Then we don't have to merge those halves, once we've sorted them.*

We have seen that *mergesort* is $O(N \lg N)$, because it is a "repeated halving" algorithm with $O(N)$ time spent on the *mergehalves* work, and $O(1)$ time spent on the trivial case.

Quicksort, by a similar argument, will be $O(N \lg N)$ if it satisfies some important provisos:

a the partitioning work, together with the re-combination, is $O(N)$;

b the trivial case is $O(1)$;

c the partitioning algorithm divides the problem into two more-or-less equal parts at each stage.

Proviso (b) is obviously satisfied (*quicksort* does nothing in the trivial case).

Proviso (a) is easy to satisfy, as we shall see.

Proviso (c) is the hard bit.

Suppose that we have picked a value *p* such that about half the values in the array are ($\leq p$) and the other half are ($p<$). Then the following loop *partitions* the sequence $A_{m..n-1}$ into two approximately equal halves:

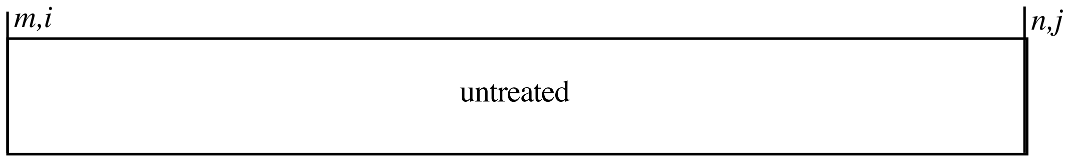
P1
```
for (int i=m,j=n; i!=j; ) {
  if (A[i]<=p) i++;
  else
  if (p<A[j-1]) j--;
  else {
    A[i]<->A[j-1]; i++; j--;
  }
}
```

I've written `A[i]<->A[j-1];` *in place of a tedious sequence of assignments. It clarifies the algorithm.*

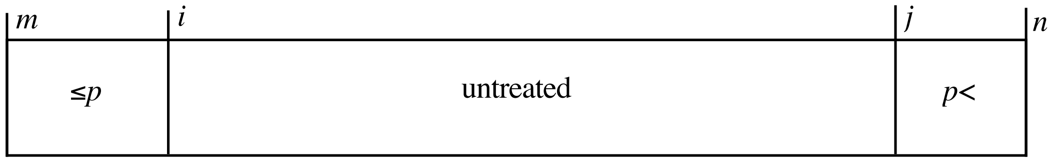
The empty `INC` *in the* for *isn't a mistake.*

Notice that all we ever do is exchange elements: it's obvious that this program makes a permutation of the original sequence.

To begin with we have this picture:



Each time we increase *i* because $A_i \le p$, or decrease *j* because $p < A_{j-1}$, we expand the areas within which we know something about the elements:



Don't be misled by the picture: either of the outer partitions might be empty – we might always increase i or always decrease j, and never the other!

Eventually this stops because either $i = j$, or $\neg\left(A_i \le p\right) \wedge \neg\left(p < A_{j-1}\right)$ - i.e. $p < A_i \wedge A_{j-1} \le p$. Then we exchange A_i and A_{j-1}, we increase *i* and reduce *j*.

It's still safe to terminate when i = j, because when we increase i and reduce j together, they don't go past each other! *We know that i < j, which is the same as* $i \le j-1$*; since* A_i *is (p<) and* A_{j-1} *is (≤p), we know that* $i < j-1$*; and therefore* $i+1 \le j-1$.

Now P1 is not the best partitioning loop we shall see, but it is $O(N)$ in time and $O(1)$ in space.

If we could pick a value p which 'partitions' the sequence neatly into two more-or-less equal-length half sequences, ***then*** P1 would be the basis of an $O(N \lg N)$-time algorithm.

Picking an approximately-median value p turns out to be the whole problem.

Averaging the elements of the sequence won't do it. Can you see why?

Picking the middle element of the sequence - element $A_{(i+j)\div 2}$ – *won't do it. Can you see why?*

We could find the median element if the sequence was already sorted ... but we're sorting it ...

Some choices of *p* are disastrous.

We build the P1 algorithm into a sorting method, and we have

Q1
```
void quicksort(type[]A, int m, int n) {
  if (m+1<n) { // two values at least
    type p = ... something ...;
    for (int i=m,j=n; i!=j; ) {
      if (A[i]<=p) i++;
      else
      if (p<A[j-1]) j--;
      else {
        A[i]<->A[j-1]; i++; j--;
    }
    quicksort(A, m, i);
    quicksort(A, i, n);
  }
}
```

In order to be sure that the recursion will terminate, we must be sure that the each of the sequences $A_{m..i-1}$ and $A_{i..n-1}$ is smaller than the input $A_{m..n-1}$: that is, we must be sure that $i \neq m$ and $i \neq n$.

But if we pick a value p which is smaller than any in the array, then the partition loop will never do an exchange and will finish with $i = j = n$ – the ‘large element’ partition will be empty.

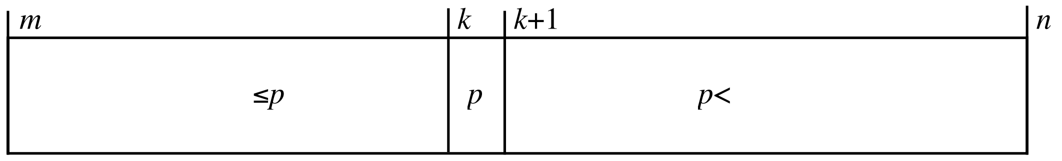
If we pick a value p which is larger than any in the array the partition loop will never do an exchange and will finish with $m = i = j$ – the ‘small element’ partition will be empty.

In either case one of the recursive calls will be just the same problem as the original. Almost certainly the method, given the same problem, will pick the same p, which will have the same effect as before, and the Q1 method will loop indefinitely.

Non-looping *quicksort*, with a different partition algorithm.

An algorithm isn't a program. quicksort is the idea 'partition; sort; sort'.

Pick a value p from the sequence $A_{m..n-1}$. Then re-arrange the array so that it consists of *three* sub-sequences: $A_{m..k-1}$, which contains values $(\leq p)$; $A_{k..k}$, which contains the value p; and $A_{k+1..n-1}$, which contains values $(p<)$:



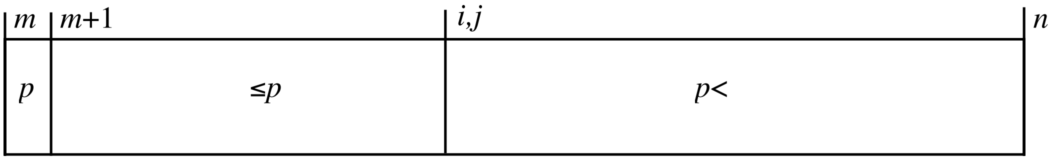
This algorithm has an important property: it puts one element – A_k – 'in place'.

Because the middle partition cannot be empty, neither of the outer partitions can be the whole array, and the algorithm can't loop.

One possible choice for p is A_m.

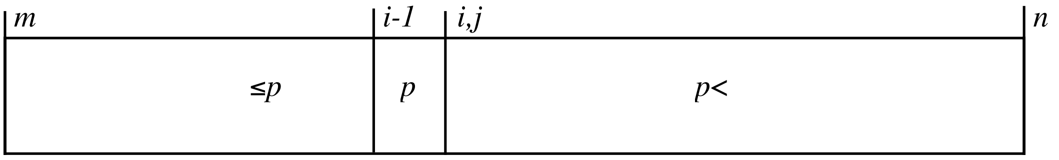
we shall see that this is not an **efficient** *choice, but it remains a possible* **correct** *choice.*

The partition technique used in P1 will partition $A_{m+1..n-1}$ using the value A_m, giving



Because of the properties of P1, either of the partitions $A_{m+1..i-1}$, $A_{j..i-1}$ might be empty

then we must swap A_m with A_{i-1}, giving



why is always safe to make that swap?

why would it sometimes be unsafe *to swap A_m with A_i?*

now $m \neq i$, $i-1 \neq j$; the algorithm can't loop!

Once we've put p in its place A_{i-1}, we call the *quicksort* algorithm on $A_{m..i-2}$, call it again on $A_{i..n-1}$, and we are finished.

Q2
```
void quicksort(type[] A, int m, int n) {
  if (m+1<n) { // two elements at least
    type p=A[m];
    for (int i=m+1,j=n; i!=j; ) {
      if (A[i]<=p) i++;
      else
      if (p<A[j-1]) j--;
      else {
        A[i]<->A[j-1]; i++; j--;
      }
    }
    A[m]=A[i-1]; A[i-1]=p;
    quicksort(A, m, i-1);
    quicksort(A, i, n);
  }
}
```

The work in quicksort is all in the partitioning, before the recursive calls.

In mergesort it was all in the merging, after the recursive calls.

How fast will Q2 run?

In the best case the two partitions will always be about the same size as each other, and Q2 will take $O(N \lg N)$ execution time.

In the worst case p will be always be an extreme value: one of the outer partitions will always be empty and the other size $N-1$; each method call will put one element in place; total execution time will be $O((N-1)+(N-2)+...+1)$ which is $O(N^2)$.

The worst case will occur just when the input sequence is either sorted or reverse-sorted.

We will be close to the worst case when we add a few elements to a sorted sequence and then sort the whole thing.

Luckily, we can do much, much, better.

Algorithms matter more than details.

The Q2 version of *quicksort* takes $O\left(N^2\right)$ time rather too often, because in practice it might rather often pick a poor value of p.

It isn't worth working on its details – using pointers instead of array indices, reducing the number of comparisons, speeding up the loop – until this major question is solved.

Breakpoint 1.

You have seen:

- *the basic idea of quicksort;*
- *a particular implementation Q2, which uses a particular selection of a 'pivot value' p;*
- *an argument that Q2 takes* $O(N \lg N)$ *time in the best case;*
- *an argument that in the worst case Q2 is* $O\left(N^2\right)$ *in time.*

A pleasant diversion: saving space in *quicksort*.

Q2 is $O(N)$ in space in the same worst case that gives $O(N^2)$ time: there will be N recursive calls, each allocating $O(1)$ space, none returning till the last has finished.

But the order in which the partitioned sequences are sorted doesn't matter.

This sequence:

```
    quicksort(A, m, i-1);
    quicksort(A, i, n);
```

has the same effect as

```
    quicksort(A, i, n);
    quicksort(A, m, i-1);
```

Whichever order we choose, the *second* recursive call can be eliminated in favour of repetition.

We change the `if` to a `while` to play this trick:

```
Q3 void quicksort(type[] A, int m, int n) {
     while (m+1<n) { // two elements at least
       type p=A[m];
       for (int i=m+1,j=n; i!=j; ) {
         if (A[i]<=p) i++;
         else
         if (p<A[j-1]) j--;
         else {
           A[i]<->A[j-1]; i++; j--;
         }
       }
       A[m]=A[i-1]; A[i-1]=p;
       quicksort(A, m, i-1);
       m=i; // and repeat ...
     }
   }
```

Now there *is* a reason to choose an order of sorting: one of the halves gets sorted by a recursive call, the other gets sorted by a repetition.

Each recursive call uses space: to use the minimum we give the recursive call the *smaller* of the two partitions to deal with!

Q4
```
void quicksort(type[] A, int m, int n) {
  while (m+1<n) { // two elements at least
    type p=A[m];
    for (int i=m+1,j=n; i!=j; ) {
      if (A[i]<=p) i++;
      else
      if (p<A[j-1]) j--;
      else {
        A[i]<->A[j-1]; i++; j--;
      }
    }
    A[m]=A[i-1]; A[i-1]=p;
    if (i-1-m<n-i) {
      quicksort(A, m, i-1); m=i;
    }
    else {
      quicksort(A, i, n); n=i-1;
    }
  }
}
```

The worst case for Q4, so far as space is concerned, is when the partition given to the recursive call is at its largest.

That happens when the partitions are equal sizes, and the problem has been divided exactly in half!

So in its *worst space case* Q4 uses space proportional to $\lg N$ – each method call uses constant space, and the maximum depth of the recursion is $\lg N$.

Therefore Q4 is $O(\lg N)$ in space.

<u>Breakpoint 2</u>.

> *Q4 does 'partition; sort; loop', and that makes it* $O(\lg N)$ *in space*
>
> *But it is still* $O(N^2)$ *in time in the worst case ..*

Another diversion: *quickfind*.

We often want to find the ‘median value’ of a sequence: a value m which occurs in $A_{0..n-1}$, such that there are about as many values ($\leq m$) in the sequence as there are values ($\geq m$).

Generalise that problem. Suppose we have an unsorted array of length n, we have an index k such that $0 \leq k < n$, and we want to find the element which will appear at position k of the sorted array.

The median is just the value which will appear at position $n \div 2$.

We might sort the array (taking $O(N \lg N)$ time if we’re lucky) and then pick the ith element. But we can do better.

The idea behind the *quickfind* algorithm, which does the job in $O(N)$ time, is that the partition algorithm (second version, above) rearranges the array so that just one element is in place. If that’s the place you want to look in then you’ve finished, otherwise repeat the process with one of the two partitions.

```
type quickfind(type[] A, int m, int n, int k) {
  while (m+1<n) {
    type p=A[m];
    for (int i=m+1, j=n; i!=j; )
        ... // partition loop
    A[m]=A[i-1]; A[i-1]=p;
    if (k<i-1) n=i-1;
    else
    if (i<=k) m=i;
    else break;
  }
  return A[k];
}
```

We shall see other versions of the partition loop; you can write a version of quickfind based on almost any of them.

Sedgewick says this is a linear algorithm. The maths is beyond us in this course, but I hope you are prepared to experiment – and to read up the references in Sedgewick and in Weiss if you are interested.

Back to reality: speeding up *quicksort*.

In principle we can't do it: there must always be a worst-case sequence that forces *quicksort* to be $O\left(N^2\right)$ in time.

$$\boxed{\text{But we } \textit{can} \text{ make it extremely unlikely that we will come across a worst-case sequence: just a few possibilities out of the } N! \text{ possible permutations we might encounter.}}$$

There is a sense in which it ***doesn't matter***: We can show that the average execution time of *quicksort* is $O(N \lg N)$.

All we have to do is to be sure that the worst case doesn't happen very often!

We don’t want the worst case to be something obvious like:

- a sorted sequence;
- a reverse-sorted sequence;
- a sequence in which all the values are equal;
- an almost-sorted sequence (one in which you have added a few elements to the beginning or end of a sorted or reverse-sorted sequence);
- a sequence in which the first (or the last, or the middle) number is the largest (or the smallest).

Those sorts of sequences crop up *all the time*. We want our worst case to be very rare.

Average cost calculation.

Taken from Weiss pp239-240.

The cost of an execution of *quicksort* consists of the $O(N)$ cost of partitioning plus the cost of two recursive calls.

The average cost will be the cost of partitioning plus the average cost of two recursive calls.

We assume that in selecting the pivot, we are equally likely to partition the array into 'small elements' and 'large elements' in any of the possible ways: small elements size 0, large elements size $N-1$; small elements size 1, large elements $N-2$; ... ; small elements size $N-1$, large elements size 0.

If the average cost of dealing with a sequence of size k is $\text{av}(k)$, then the average cost of each of the recursive calls will be

$$\frac{\text{av}(0)+\text{av}(1)+\ldots+\text{av}(N-2)+\text{av}(N-1)}{N}$$

The average cost of the whole execution will be approximately

$$av(N) = N + 2\left(\frac{av(0) + av(1) + \ldots + av(N-2) + av(N-1)}{N}\right)$$

This neglects the constant of proportionality in the cost of partitioning: but if you insert it, the analysis doesn't change.

It also neglects constant terms, but once again, it doesn't change the analysis.

There's a standard way to simplify problems like these, and with a bit of juggling we get

$$N\,\mathrm{av}(N) - (N-1)\mathrm{av}(N-1) = 2\,\mathrm{av}(N-1) + 2N - 1$$

Ignoring the constant, and shifting terms around, you get

$$N\,\mathrm{av}(N) = (N+1)\mathrm{av}(N-1) + 2N$$

That in turn gives you

$$\frac{\mathrm{av}(N)}{N+1} = \frac{\mathrm{av}(N-1)}{N} + \frac{2}{N+1}$$

But then you can use that equation (putting $N-1$ in place of N) to give you

$$\frac{\text{av}(N-1)}{N} = \frac{\text{av}(N-2)}{N-1} + \frac{2}{N}$$

Substituting back into the first equation gives

$$\frac{\text{av}(N)}{N+1} = \frac{\text{av}(N-2)}{N-1} + \frac{2}{N} + \frac{2}{N+1}$$

and so on and on, until eventually

$$\frac{\text{av}(N)}{N+1} = \frac{\text{av}(1)}{2} + \frac{2}{1} + \frac{2}{2} + \ldots + \frac{2}{N} + \frac{2}{N+1}$$

Now $\text{av}(1)/2$ is a constant, and $1/1 + 1/2 + \ldots + 1/N + 1/(N+1)$ is, so I'm told, $O(\lg N)$. So $\text{av}(N)/(N+1)$ is $O(\lg N)$, and therefore $\text{av}(N)$ is $O(N \lg N)$.

On average this is a good algorithm; it remains to make the worst case very, very unlikely to occur.

Step 1: equal treatment for values (=p)

When all the values in the array are equal, the methods so far will put them all in the left partition, since they will all be (≤*p*). Then Q4 is guaranteed to take $O(N^2)$ time.

> *This case arises rather often when searching large collections of small values: for example, if we sort a large array of bytes we are certain to produce an answer in which there are long sub-sequences of identical elements, because there are only 256 different byte values.*

The problem is that the Q4 partition is *unbalanced*: it puts all the values (=*p*) in one half.

To remedy the problem, we have to make sure that elements (=*p*) are equally likely to end up in either partition.

On a sequential, deterministic machine we can't write the algorithm so that both the non-exchange steps accept values (=*p*). We have to make neither of them accept those values:

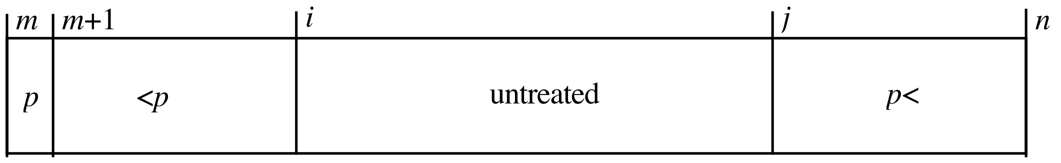
P2
```
while (i!=j) {
  if (A[i]<p) i1++;
  else
  if (p<A[j-1]) j--;
  else {
    A[i]<->A[j-1]; i++; j--;
  }
}
```

This partition algorithm performs more exchanges and is therefore slower than the algorithm in Q4, but it still takes $O(N)$ time to partition.

In an array of identical elements, all (=*p*), P2 will achieve equal-sized partitions. That makes it more likely that we will achieve $O(N \lg N)$ time.

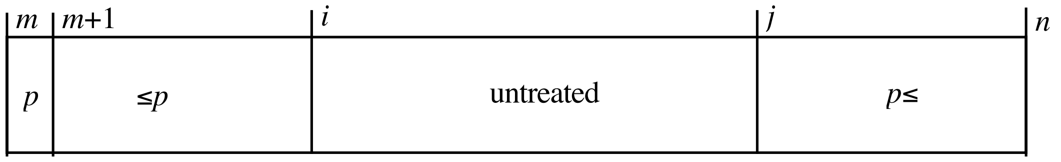
We can make the <u>details</u> *more expensive but reduce the* <u>overall cost</u>. *Naively concentrating on details doesn't always pay off.*

We begin, as before, by taking p as A_m. We advance i and j as far as they can go:



When they stop, either $i = j$ and we've finished, or $p \le A_i \wedge A_{j-1} \le p$; in the second case we exchange and then advance i and j towards each other.

Putting an element ($\le p$) in the lower segment makes that segment ($\le p$); similarly we have made the upper segment ($p\le$). That's OK: we wanted a balanced partition! Now we have



And so on, we hope, until $i = j$. But the P2 loop has a defect.

When P1 does an exchange we have both $i < j$ and $A_i \leq p < A_{j-1}$. From the second we deduce that $A_i \neq A_{j-1}$ and therefore $i \neq j-1$; from that and the first we deduce that $i < j-1$, which means that $i+1 \leq j-1$, and therefore increasing i and reducing j wasn't dangerous.

Now we have $A_i \leq p \leq A_{j-1}$ and we can't deduce that $A_i \neq A_{j-1}$; although $i < j$ and therefore $i \leq j-1$, we can't be sure that $i < j-1$.

> *If $i = j-1$ we swap that element with itself, which is a bit wasteful but never mind.*

In the case that $i = j-1$, the instructions `i++; j--;` will make $i = j+1$, and the `i!=j` test in the *for* instruction will never be satisfied.

The exchange must produce a state in which either $i < j$, $i = j$ or in the worst case $i = j + 1$. We can fix the program rather easily:

```
P3 while (i<j) {
      if (A[i]<p) i++;
      else
      if (p<A[j-1]) j--;
      else {
        A[i1]<->A[j-1]; i++; j--;
      }
    }
```

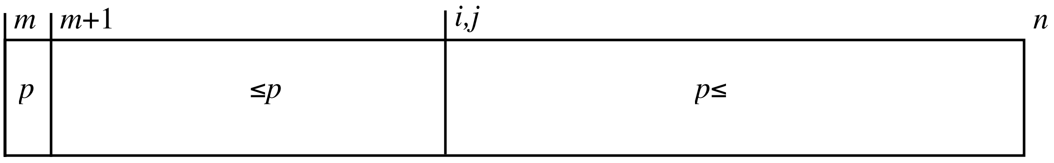
You might have expected me to write `i<j` in P1 and P2; I had a good reason for not doing so.

By writing `i!=j` I ensured that when the loop stopped, I knew *exactly* what the state of the variables would be.

When I write `i<j`, I have to be much more careful. The problem now is to find where to put the pivot value *p* after the array is partitioned.

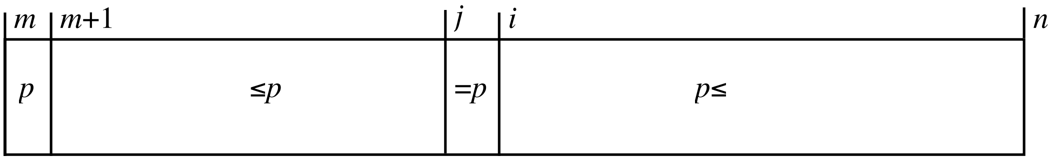
*We write our loops not only so that they finish, but also so that we know **how** they finish.*

What might the result of partitioning be? We might finish with a picture like this:



– in which case we should put the pivot in A_{i-1} (or we could call it A_{j-1}; it's the same element)

– or we might have a picture like this:



– in which case we should put the pivot in A_{j-1}.

So we must always put the pivot into A_{j-1}, and the 'sort;sort' phase then has to deal with the subsequences $A_{m..j-2}$ and $A_{i..n-1}$.

The method has changed quite a bit, first to save space and now to balance the partitions:

Q5
```
void quicksort(type[] A, int m, int n) {
  while (m+1<n) { // two elements at least
    type p=A[m];
    for (int i=m+1, j=n; i!=j; ) {
      if (A[i]<p) i++;
      else
      if (p<A[j-1]) j--;
      else {
        A[i]<->A[j-1]; i++; j--;
      }
    }
    A[m]=A[j-1]; A[j-1]=p;
    if (j-1-m<n-i) {
      quicksort(A, m, j-1);
      m=i;
    }
    else {
      quicksort(A, i, n);
      n=j-1;
    }
  }
}
```

Q5 is $O(\lg N)$ in space and gives $N \lg N$ performance with sequences of identical values.

Looking for a better pivot.

Attempt 1: pick the middle element.

If we think that sorted and reverse-sorted sequences and almost-sorted sequences will happen rather often, then we will be more likely to find a value which equally partitioned the segment if we look in the *middle*:

Q6
```
void quicksort(type[] A, int m, int n) {
  while (m+1<n) { // two elements at least
    int k=(m+n)/2; type p=A[k];
    A[k]=A[m];
    for (int i=m+1,j=n; i<j; )
      ... // partition as P3
    A[m]=A[j-1]; A[j-1]=p;
      ... // sort; loop as Q5
  }
}
```

Q6 would pick a good pivot for sorted, reverse-sorted and for almost-sorted sequences; it would also do well on sequences of identical elements.

Would it be hard to construct a sequence which fooled the ‘pick the middle element’ strategy? I don’t think so (put the largest element in the middle of the sequence, put the next largest two in the middle of the sub-sequences that surround it, the next largest four in the middle of the sub-sequences that ... and so on).

But you might think that was a very unlikely arrangement ... Perhaps so.

can you construct an argument which shows that this ‘unlikely’ arrangement is less likely than sorted, or reverse sorted? I don’t know such an argument.

Attempt 2: pick a random element.

We can protect ourselves against sorted sequences and any other rationally-ordered sequence by choosing an element at random, if we had a random-number generator ...

What we can do is choose *pseudo*-randomly. It takes constant time – a small number of multiplications and divisions – and it's in the Java library:

Q7
```
void quicksort(type[] A, int m, int n) {
  while (m+1<n) { // two elements at least
     int k = m+(random.nextInt()%(n-m));
     type p=A[k];
     A[k]=A[m];
     for (int i=m+1,j=n; i<j; )
       ... // partition as P3
     A[m]=A[j-1]; A[j-1]=p;
       ... // sort; loop as Q5
   }
}
```

This method uses a non-local variable `random`*, which contains an object of type* `Random`*: see the Java API documentation.*

We can do better.

Attempt 3: pick the best pivot out of three.

Sedgewick and Weiss suggest that we should look at the first, the middle and the last element and find the median of those three (which takes 2 or 3 comparisons):

Q8
```
void quicksort(type[] A, int m, int n) {
   while (m+1<n) { // two elements at least
      int k=(m+n)/2,
            q=A[m]<A[k] ?
                   (A[k]<A[n-1] ? k :
                         (A[m]<A[n-1] ? n-1 : m)
                   ) :
                   (A[m]<A[n-1] ? m :
                         (A[k]<A[n-1] ? n-1 : k));
      type p=A[q];
      A[q]=A[m];
      for (int i=m+1, j=n; i<j; )
         ... // partition as P3
      A[m]=A[j-1]; A[j-1]=p;
         ... // sort; loop as Q5
   }
}
```

Now ***there are very few permutations*** of an N-element sequence which would force $O(N^2)$ time on Q8.

You might like to try to find such a permutation.

Weiss says median-of-three is as good as you need, but what does he know? What does anyone know? Why not experiment?

Reducing comparisons and exchanges.

Each time round the P3 loop we do two or three comparisons: `i<j`, `A[i]<p`, `p<A[j-1]`. Sedgewick (and Weiss, in his earlier book) say you should use median-of-three to select a pivot and two others, and use the method of sentinels to reduce the number of times you check `i<j`.

I say pooh! I know better than that, I think. (You might like to experiment; it wouldn't be the first time I'd been wrong about a program ...)

Step 1: Using the method of sentinels.

Q8 looks at three values to choose a pivot: A_m, A_k and A_{n-1}. It chooses the median as *p*. It follows that of the other two values, one is ($\leq p$) and the other is ($p\leq$).

We might re-arrange the array so that the value which is ($\leq p$) is in A_{m+1}, and the value which is ($p\leq$) is in A_{n-1} (smaller value at $m+1$ end, larger at $n-1$ end).

Those values are 'sentinels' for the partition loop. We start with $i = m+2$ and $j = n-2$. We can increase *i* as long as $A_i < p$, because the sentinel in A_{n-1} will stop that search if nothing else does. We can reduce *j* as long as $p < A_{j-1}$, because the sentinel in A_{m+1} will stop that search if nothing else does.

In those circumstances, this loop won't check `i<j` quite as often as the ones we have seen so far:

P4
```
for (int i=m+2,j=n-1; i<j; ) {
  while (A[i]<p) i++;
  while (p<A[j-1]) j--;
  if (i!=j) {
    A[i]<->A[j-1]; i++; j--;
  }
}
```

The inner *while* loops can't overrun the bounds of the sub-sequence, because of the values in A_{m+1} and A_{n-1}:

- $A_{m+1} \leq p$ is true to begin with, and each time we increase i we keep it true;
- $p \leq A_{n-1}$ is true to begin with, and each time we reduce j we keep it true;

Then if $i = j$ we definitely *do not want* to swap the elements, so we need a check for this possibility.

We need to rearrange the array – put smallest of $\langle A[m], A[(m+n) \div 2], A[n-1] \rangle$ in $A[m+1]$, median in $A[m]$, largest in $A[n-1]$ – to get the sentinels in place.

It's a little insertion sort, which does 2 or 3 comparisons and 1, 2, 3 or 4 exchanges:

S1
```
A[m+1]<->A[(m+n)/2];
if (A[m+1]>A[m]) A[m+1]<->A[m];
if (A[m]>A[n-1]) {
  A[m]<->A[n-1];
  if (A[m+1]>A[m]) A[m+1]<->A[m];
};
```

I claim that it's possible to save the small amount of work done by the exchanges, using only the pivot as sentinel.

The algorithm came from Jon Rowson, but he can't remember where he found it. So some unsung author is going uncredited. Sorry, whoever you are.

Choose a pivot element in the array, but *leave the pivot element where it is*. Run the partition loop almost as P4, but starting with *i* = *m* and *j* = *n*:

P5
```
int k=(m+n)/2,
     q=A[m]<A[k] ?
            (A[k]<A[n-1] ? k :
                (A[m]<A[n-1] ? n-1 : m)
            ) :
            (A[m]<A[n-1] ? m :
                (A[k]<A[n-1] ? n-1 : k));
type p=A[q];
for (int i=m,j=n; i<j) {
  while (A[i]<p) i++;
  while (p<A[j-1]) j--;
  if (i!=j) {
    A[i]<->A[j-1]; i++; j--;
  }
}
```

If this partitioning mechanisms is to work, the inner *while*s mustn’t exceed the bounds of the segment being sorted; if it is to form the basis of a reliable *quicksort*, neither of the partitions which it produces must be the whole of the segment to be sorted.

On first execution of the *for* body, the pivot p is in element q. It doesn't matter where q is, provided that $m \le q < n$. Then we know:

- $\exists r : m \le r < j \wedge A_r \ge p$, because $r = q$ will do if nothing else will;
- $\exists s : i < s \le n \wedge A_{s-1} \le p$, because $s = q + 1$ will do if nothing else will.

From the first proposition we can be sure that the first *while* must stop; from the second we can be sure that the second *while* will stop.

From the two we can be sure that when those *while*s do stop we must have $i < j$; so we will be sure to carry out one exchange at least.

Following that first exchange we either have $i \ge j$ – and then the *for* terminates – or else $i < j$, and then we have:

- $A_{i-1} \le p$, so $\exists r' : m \le r' < j \wedge A_{r'} \le p$ and the left-hand sentinel is in place;
- $A_j \ge p$, so $\exists s' : i \le s' \le n \wedge A_{s'} \ge p$ and the right-hand sentinel is in place.

The rest of the argument follows exactly the argument that was used above to justify the operation of S1.

> *This version is* no better *than S1 in O(...), but it seems to avoid a tiny bit of work, and I think it's a neater program. That last is a thing worth having for itself.*

What this analysis shows is that *we don't need to place sentinels*: first time through the pivot does the job; later times there must be something in each of the end segments and we have the sentinels for free.

One oddity of this version is that at the end of the partition loop *we don't know where the pivot is*: it has almost certainly been moved by one of the exchanges. So we can't put the pivot into place between the segments.

But that doesn't matter: it is impossible for any of the three segments to be the entire sequence, because the loop always does one exchange, and so the algorithm won't loop.

This version can be adapted to any choice of pivot element, though I illustrate median-of-three:

Q9
```
void quicksort(type[] A, int m, int n) {
  while (m+1<n) {
    int k=(m+n)/2,
        q=A[m]<A[k] ?
             (A[k]<A[n-1] ? k :
                 (A[m]<A[n-1] ? n-1 : m)
             ) :
             (A[m]<A[n-1] ? m :
                 (A[k]<A[n-1] ? n-1 : k));
    type p=A[q];
    for (int i=m,j=n; i<j) {
      while (A[i]<p) i++;
      while (p<A[j-1]) j--;
      if (i!=j) {
        A[i]<->A[j-1]; i++; j--;
      }
    }
    if (j-m<=n-i) {
      quicksort(A, m, j); m=i;
    }
    else {
      quicksort(A, i, n); n=j;
    }
  }
}
```

Avoiding recursive calls when sorting short sequences.

Recursive calls take time. Usually several times as much as an assignment instruction.

Recall the recursive halving argument:

$$\left.\begin{array}{c}
N \\
N \div 2, \qquad\qquad\qquad N \div 2 \\
N \div 4, \qquad N \div 4, \qquad N \div 4, \qquad N \div 4 \\
N \div 8, N \div 8, N \div 8, N \div 8, N \div 8, N \div 8, N \div 8, N \div 8 \\
\text{... and so on, until ...} \\
1,1,1,1, \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots , 1,1,1,1
\end{array}\right\} \lg N \text{ rows} : O(N \lg N)$$

At the top level there is one call, at the next level two, then four, and so on. Each step down halves the size of the subproblems, at the cost of doubling the number of recursive calls.

Half the recursive calls are in the penultimate line, splitting problems of size 2 into problems of size 1. Three-quarters are in that line and the line above, splitting problems of size 4 into problems of size 2 and then 1. And so on.

Now it seems that insertion sort, although it is $O\left(N^2\right)$, is faster than *quicksort* on small problems – just because of the relative cost of insertion sort's exchanges and *quicksort*'s recursive calls.

Our authorities (Sedgewick, Weiss) claim that for values of N less than some limit N_0, which they variously put at '15' or 'anywhere between 5 and 20', it is faster to use insertion sort than *quicksort*.

> *The actual cutoff point will depend on the machine you are using, the language, and the implementation of the language.*

I leave it to you

- to alter the procedure to use insertion sort appropriately;
- to show that using insertion sort (which is $O\left(N^2\right)$) as part of *quicksort* still allows us to claim that *quicksort* is $O(N \lg N)$;
- to find the point at which it is worth switching.

> *we do not rule out the possibility that such problems might form the basis of an examination question.*

Engineering is about trade-offs

quicksort seems to be the best at large *N*, insertion sort the best at small *N*. We can combine them, and get the best of both worlds.

But we have to *trade*: space for speed or vice-versa, (insertion sort uses less space than *quicksort*).

Sometimes we might have to trade development cost for runtime cost, sometimes the other way round.

There are many tradeoffs of this kind.

For example: if we want to minimise the number of exchanges, but don't mind about the number of comparisons, selection sort is better than any other kind of sort.

There are no answers which are right everywhere at every time.

Key points

the basic idea of *quicksort* is “partition; sort; sort”.

quicksort can be made to be $O(\lg N)$ in space (worst case), and $O(N \lg N)$ in time (on average).

the worst-case analysis is that *quicksort* is $O(N^2)$ in execution time, but we can make that worst case very, very unlikely to occur.

quicksort is a subtle algorithm: it’s easy to define it so that it loops, and it’s easy to define it sub-optimally.

most of the difficulty in defining and understanding *quicksort* lies in the detail of the partition algorithm.

we can speed up our use of *quicksort* by using a fast $O(N^2)$ sort to deal with ‘small’ sorting problems.

programming is a subtle business.

quicksort is an important algorithm, which every self-respecting computer scientist should know and understand.

<u>Coda: the original *quicksort*</u>.

A work of genius.

I leave it to you to deduce how it works. I leave it to you to decide whether this version could compete with Q9 using its own partition algorithm but avoiding one of the recursive calls.

> *I have distilled this method from algorithm 63 (partition) and 64 (*quicksort*) in Communications of ACM vol 4, p 321, submitted by C.A.R. Hoare.*
>
> *Those algorithms were expressed in Algol 60, a wonderful language now no longer in practical use. I've translated the original into a single method to make it as understandable as possible to a Javaglot audience.*

```
void OriginalQuicksort(type[] A, int m, int n) {
   if (m+1<n) {
    int k = m+(Random.nextInt()%(n-m)),
        i=m, j=n;
    type p=A[k];
    while (true) {
      while (i<n && A[i]<=p) i++;
      while (j>m && A[j-1]>=p) j--;
      if (i<j) {
        A[i]<->A[j-1]; i++; j--;
      }
      else
        break;
    }
    if (k<j) {
      A[k]<->A[j-1]; j--;
    }
    else
    if (i<=k) {
      A[i]<->A[k]; i++;
    }
    OriginalQuicksort(A, m, j);
    OriginalQuicksort(A, i, n);
  }
}
```