Merging sorted sequences.

Suppose I have two sequences, not necessarily the same length, each sorted by (≤):

<13,23,24,24,35,80,85,86,87,88,90,92>

<9,14,25,29,32,44,66,81,82,90,91,94,98,99>

Then I can ***merge*** them into a single sorted sequence, treating each like a stack of cards and taking the smallest exposed card each time:

<13,...>, <9,...>: take 9 from right, exposing 14.

<13,...>, <14,...>: take 13 from left, exposing 23.

<23,...>, <14,...>: take 14 from right, exposing 25.

<23,...>, <25,...>: take 23 from left, exposing 24.

<24,...>, <25,...>: take 24 from left, exposing second 24.

... and so on.

When there are equal numbers I can take from either side:

<90,92>, <90,91,94,98,99>: ***either*** take 90 from left, exposing 92, ***or*** take 90 from right, exposing 91.

When one side is empty, I must take from the other:

<>, <94,98,99>: take 94 from right

... and so on

I keep this process going till both sides are empty.

The result will certainly be the sorted sequence <9, 13, 14, 23, 24, 24, 25, 29, 32, 35, 44, 66, 80, 81, 82, 85, 86, 87, 88, 90, 90, 91, 92, 94, 98, 99>

This program merges $A[am..an-1]$ with $B[bm..bn-1]$, putting the result into $C[cm..cn-1]$, where $cn = cm + (an - am) + (bn - bm)$:

```
int ia, ib, ic, cn;
for (ia=am, ib=bm, ic=cm, cn=cm+an-am+bn-bm;
     ic!=cn; ) {
  if (ia==an)             // A is exhausted
    C[ic++]=B[ib++];
  else
  if (ib==bn)             // B is exhausted
    C[ic++]=A[ia++];
  else
  if (A[ia]<=B[ib])       // A can come first
    C[ic++]=A[ia++];
  else                    // B must come first
    C[ic++]=B[ib++];
}
```

The for *has an empty* INC *part: this* **<u>isn't</u>** *a mistake.*

You may have to brush up your understanding of ***exactly*** *what formulæ like* `ia++` *mean.*

This program takes $O(NA + NB)$ execution time, where NA and NB are the lengths of the array segments merged; it takes $O(1)$ space.

It’s easy to package it as a method:

```
public void merge(type[] A, int am, int an,
                  type[] B, int bm, int bn,
                  type[] C, int cm) {
  int ia, ib, ic, cn;
  for (ia=am, ib=bm, ic=cm, cn=cm+an-am+bn-bm;
       ic!=cn; ) {
    if (ia==an) C[ic++]=B[ib++];
    else
    if (ib==bn) C[ic++]=A[ia++];
    else
    if (A[ia]<=B[ib]) C[ic++]=A[ia++];
    else C[ic++]=B[ib++];
  }
}
```

In this program, and throughout the discussion of mergesort, I’m assuming that we are dealing with arrays of some type which can be ordered. They needn’t necessarily be integers or strings.
I’ve used the (≤) operator to compare elements of the arrays: in reality you might have to use a method and write something like `A[ia].lesseq(B[ib])`.

Warning: the method is not as robust as it might seem!

- It will crash if $an > am$ but either an or am is outside the limits of the array A;
- likewise for bn, bm and B;
- it will crash if cm or cn is outside the limits of C;
- if $(an - am) + (bn - bm)$ is negative it will exceed the limits of C and crash;
- it may do very stupid things, including crashing, if $an < am$ or $bn < bm$.

So the specification of this method ought to include lots of precautionary conditions.

You may like to practise your specification skills by writing down some of those conditions.

Computer scientists believe that it is better that a program crashes than it does the wrong thing and carries on. Hence the tests in the code above are all either $==$ or $\neq$.

It would certainly be possible to write a version which worked even though $am > an$ or $bm > bn$. Is it necessary to do so? Would it be sensible to do so?

Using *merge* to speed up insertion sort.

Because insertion sort takes $O\left(N^2\right)$ execution time, it's tempting to halve the size of the problem we give it.

> *the same applies if we try to speed up selection sort or bubble sort.*

Sorting one half-size array with an $O\left(N^2\right)$ algorithm takes one quarter the time that it would take to sort the whole array; so sorting two half-size arrays would take one half the time that it would take to sort the whole array.

And then *merging the results*, using the program above, would be $O(N)$

– so if we do two half-sorts and a merge, we get an algorithm which should be about twice as fast as insertion sort.

Assume `insertionsort(type[] X, int m, int n)` sorts the array segment $X[m..n-1]$.

Then this method sorts $A[m..n-1]$ using the auxiliary array $B[m..n-1]$

```
void splitsort(type[] A, type[] B,
               int m, int n) {
  if (n-m>=2) { // sort two elements or more
    int k = (m+n)/2; // the midpoint
    insertionsort(A, m, k);
    insertionsort(A, k, n);
    merge(A, m, k, A, k, n, B, m);
    for (int i=m; i<n; i++) A[i]=B[i];
  }
}
```

merge puts the answer into B: line 8 copies it back again.

It's a pity that we have to include line 8, but nevertheless, at sufficiently large problem sizes $splitsort(A,B,m,n)$ will be faster than $insertionsort(A,m,n)$, ***despite the wasteful copying***.

You may like to try to construct the argument which supports that assertion.

I assume that the execution cost of the new *formula is at worst* $O(N)$.

We can speed it up a bit by making *two* steps of halving.

If each step is an advantage, why not use two or more?

This program uses *splitsort1* to achieve a sort in array $S[0..n-1]$:

```
type[] T = new type[S.length];
splitsort1(S,T,0,S.length);
```

Here's *splitsort1*:

```
void splitsort1(type[] A, type[] B,
                int m, int n) {
  if (n-m>=2) { // sort two elements or more
    int k = (m+n)/2; // the midpoint
    splitsort2(A, B, m, k);
    splitsort2(A, B, k, n);
    merge(A, m, k, A, k, n, B, m);
    for (int i=m; i<n; i++) A[i]=B[i];
  }
}
```

splitsort2 still works by using insertion sort:

```
void splitsort2(type[] A, type[] B,
                int m, int n) {
  if (n-m>=2) { // sort two elements or more
    int k = (m+n)/2; // the midpoint
    insertionsort(A, m, k);
    insertionsort(A, k, n);
    merge(A, m, k, A, k, n, B, m);
    for (int i=m; i<n; i++) A[i]=B[i];
  }
}
```

These two together would be faster than *splitsort*, for the same reason as *splitsort* is faster than insertion sort.

So I could do the same trick again, and again, and ...

But surely, *any* method which sorts one array, using another as auxiliary storage, would do in place of *splitsort2* – or indeed in place of *splitsort1*.

This is the principle of ***procedural abstraction****: we use methods according to their specification.*

```
void mergesort
  (type[] A, type[] B, int m, int n) {
  if (n-m>=2) { // sort two elements or more
    int k = (m+n)/2; // the midpoint
    mergesort(A, B, m, k);
    mergesort(A, B, k, n);
    mergehalves(A, B, m, k, n);
    for (int i=m; i<n; i++) A[i]=B[i];
  }
}
```

mergehalves takes $A[m..k-1]$ and $A[k..n-1]$ and merges them into $B[m..n-1]$:

```
public void mergehalves
  (type[] A, type[] B, int m, int k, int n) {
  int ia1, ia2, ib;
  for (ia1=m, ia2=k, ib=m; ib!=n; ) {
    if (ia1==k) B[ib++]=A[ia2++];
    else
    if (ia2==n) B[ib++]=A[ia1++];
    else
    if (A[ia1]<=A[ia2]) B[ib++]=A[ia1++];
    else B[ib++]=A[ia2++];
  }
}
```

Self-definition *may* be acceptable.

Experts don't try to imagine the order in which recursive methods like *mergesort* do their thing.
Become an expert.

"A rose is a rose is a rose" defines a thing in terms of itself, and is meaningless (as a definition).

Haven't I defined *mergesort* in terms of *mergesort*?

No: I have defined "*mergesort* on a sequence length $n - m$" in terms of "*mergesort* on a sequence length $(n - m) \div 2$" – *not at all* the same thing.

That's because

1 *mergesort* on a trivial sequence (zero or one elements, $n - m < 2$) does nothing at all, because such a sequence is already sorted;

2 we define "*mergesort* on a sequence length $(n - m) \div 2$" in terms of "*mergesort* on a sequence length $((n - m) \div 2) \div 2$", and so on down;

3 If an integer is (≥ 2), you can't keep dividing it by 2 indefinitely without reaching 1.

Conclusion: *mergesort* terminates because:

- each recursive call is given a *shorter* sequence than its parent;
- you can't go on indefinitely reducing the size of the sequences without reaching a sequence size 1 or 0, when the problem is trivial.

You *don't have to think* about how a recursive method is going to be executed.

All you have to do is guarantee:

- that ***if*** the recursive calls do their work, ***then*** the method achieves the desired result;
- that each recursive call is given a 'smaller' problem than its parent;
- that you can't go on reducing the size of the problem indefinitely without reaching a 'base case';
- that the base case really does the business.

Experts know that to try to imagine what happens, and in what order, is *not the way* to design or to understand a recursive procedure.

How fast does *mergesort* run?

So far you've seen a few different patterns of execution:

$$\left.\begin{array}{l} k \\ k \\ k \\ \ldots \\ k \end{array}\right\} N \text{ times} : O(N) \qquad \left.\begin{array}{l} M \\ M \\ M \\ \ldots \\ M \end{array}\right\} N \text{ times} : O(N \times M)$$

$$\left.\begin{array}{l} N \\ N \\ N \\ \ldots \\ N \end{array}\right\} N \text{ times} : O\left(N^2\right) \qquad \left.\begin{array}{l} 1 \\ 2 \\ 3 \\ \ldots \\ N \end{array}\right\} N \text{ rows} : O\left(N^2\right)$$

Now we have a new one: *repeated halving*.

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

The work done by *mergesort* on an array of size *N* is:

- ***either*** a test, a couple of recursive calls, and a merge;
- ***or*** a single assignment.

The first alternative is $O(N)$, *apart from the work done in the recursive calls*, because *mergehalves* is $O(N)$, and the copying back is $O(N)$.

The second alternative is $O(1)$.

So the work done is (work proportional to N) + (the work done by two recursive calls, each working on an array of size about $N \div 2$).

Each of those recursive calls does (work proportional to about $N \div 2$) + (the work done by two recursive calls, each working on an array of size about$N \div 4$).

But there are two of them, so the total is (work proportional to N) + (the work done by 4 recursive calls, each working on an array of size about $N \div 4$).

And so on: at each level you get (work proportional to N) + (the work done by 2^k recursive calls, each working on an array of size about $N \div 2^k$).

Eventually 'about $N \div 2^k$' becomes 1, and the work of the recursive call in that case is $O(1)$.

'about $N \div 2^k$' never becomes 0 in the program I have shown you. Can you see why this is?

$$\boxed{\text{The magic is perfect, and it is understandable. Run it in races against Shellsort.}}$$

Nothing comes for nothing.

Mergesort will be much faster than insertion sort on average, even on quite small examples, and will be faster than Shellsort on sufficiently large examples.

experiment will show where the boundaries are

But it has a flaw: ***it uses a lot of space***. It uses $O(N)$ space, in fact: the space used by the *to* array.

If you have the space, and you want the speed, you may be prepared to pay the price.

In general it is always possible to trade space for speed, and vice-versa.

Proof later ...

Making *mergesort* faster still!!

We can't make *mergesort* better than $O(N \lg N)$, but we can affect the constants of proportionality.

That is, we might be able to make it run two or three or four ... times faster, or we might make it start up more quickly on small examples.

Step 1: eliminating the copying.

The copying in *mergesort* is offensive. But it can be avoided!

We have two arrays: one holds the data we want to sort, the other is auxiliary storage. We need to distinguish which array we want the answer in, and which can be used, when we need it, for temporary storage.

Actually all we need is to switch the roles of ‘answer array’ and ‘auxiliary array’ on each recursive call!

The method heading is

```
void mergesort
  (type[] from, type[] to, type[] aux,
   int m, int n) {
```

and it is used like this:

```
type[] T = new type[S.length];
mergesort(S,S,T,0,S.length);
```

– sort *S*, putting the answer in *S* and using *T* as auxiliary storage.

Here's the whole method:

```
void mergesort
  (type[] from, type[] to, type[] aux,
   int m, int n) {
  if (n-m>=2) { // sort two elements or more
    int k = (m+n)/2; // the midpoint
    mergesort(from, aux, to, m, k);
    mergesort(from, aux, to, k, n);
    mergehalves(aux, to, m, k, n);
  }
  else
    to[m]=from[m]; // assuming m<n
}
```

Once again, it *doesn't pay* to try to trace the action of this method. Think inductively, like an expert.

Provided that this method takes its data from `from` *and puts its result in* `to` *– and it does, on lines 8 and 11 – it will work correctly.*

If `to` *and* `from` *are the same array, then line 11 does some unnecessary work. But it's a tiny amount, and it's a price worth paying.*

The method switches the roles of *to* and *aux* on each recursive call.

Transfer of data *from*→*to* happens on line 11. Transfer *aux*→*to* happens on line 8. Lines 6 and 7 ask for transfer *from*→*aux*.

One important property is that the two calls work in different areas of the array: line 6 works in the $m..k-1$ region of *from*, *to* and *aux*; line 7 works in the $k..n-1$ region. Neither touches the region of the other. They can't interfere.

This method should be measurably faster than the original. You should measure the difference, in the lab.

> *How much improvement might you expect? The original mergesort does about* $N \lg N$ *array assignments, plus the work connected with about N* for *loops, when copying back to→from. This one replaces all that with N array assignments, and it uses an extra argument in about N procedure calls* ($N/2 + N/4 + ... + 1 \approx N$)*. Each argument provision might cost about the same as an assignment, so it's replacing* $O(N \lg N)$ *work with* $O(N)$ *in just a part of the algorithm. There may not be much in it.*

Step 2: eliminating tests in mergehalves

mergehalves does too many comparisons. Each time round the loop it does two, three or four comparisons – `ia1==k`, `ia2==n`, `A[ia1]<=A[ia2]`, `ib!=n` – all in the inner loop of the algorithm.

There's a trick we can use to eliminate limit-comparisons like `ia1==k` and `ia2==n`. The trick is called ***the method of sentinels***.

Suppose we have to merge $A_{m..k-1}$ and $A_{p..n-1}$: suppose that A_n exists and is bigger than anything in $A_{m..k-1}$; similarly A_k exists and is bigger than anything in $A_{p..n-1}$.

If we exhaust $A_{m..k-1}$ first, we shall be looking at A_k, which we will never choose because `A[k]<=A[ia2]` is bound to fail. If we exhaust $A_{p..n-1}$ first, we shall be looking at A_n, which we will never choose because `A[ia1]<=A[n]` is bound to succeed.

So, if we can place sentinels, we *shan't need limit-comparisons*.

But we haven't room for such sentinels: we have to merge $A_{m..k-1}$ and $A_{k..n-1}$.

Ah! but suppose that $A_{m..k-1}$ was sorted by (≤) – big elements at the top – and that $A_{k..n-1}$ was sorted by (≥) – big elements at the bottom. We only ever look at one or other of the sentinels (only one of the halves can be exhausted first): either A_k would be a sentinel for $A_{m..k-1}$, or A_{k-1} would be a sentinel for $A_{k..n-1}$!

I leave the details of the implementation to you - you have to tell *mergesort* which way to sort – (≤) or (≥) order – and you have to make sure everything fits together. It's possible, and it just about doubles the speed!

This time the speedup should be a measurable fraction. We are affecting the constant of proportionality, by speeding up the $O(N)$ work of mergehalves significantly.

Step 3: eliminating recursion in favour of a loop

Mergesort is recursive: it doesn't need to be.

> *but it is **easier to understand** if it's recursive! The iterative implementation is really intricate; the algorithm is naturally recursive.*
>
> *The message of the Landin diagram: **first** get it right, **then** speed it up.*

Each procedure call in a program takes time: about the same time as 10 assignment instructions.

> *That's true in C or C++: what is it in Java?*

If we can replace procedure calls by assignments, we can speed *mergesort* up a lot.

We begin by merging pairs of consecutive single-element sub-sequences from A to B – $A_{1..1}$ with $A_{2..2}$, $A_{3..3}$ with $A_{4..4}$, ... – so that B consists of sorted two-element sub-sequences – $B_{1..2}$, $B_{3..4}$, ...

Then we merge consecutive pairs of two-element sub-sequences from B to A – $B_{1..2}$ with $B_{3..4}$, $B_{5..6}$ with $B_{7..8}$, ... – so that A consists of sorted four-element sub-sequences.

And so on, back and forth, until we produce a sorted array.

I leave the details to you (though we may return to this question, perhaps in an exam).

It won't be possible, in the worst case, to avoid an 'extra copy' from B back to A.

Be careful when $n - m$ is not exactly a power of 2!

Steps 4 and beyond ...

Instead of beginning by merging one-element sub-sequences, merge ‘runs’: sub-sequences of *A* which happen to be in order when you start. Then try to see if you can carry on by merging ‘runs’ from *B* ...

There is space for a competition here! The fastest *mergesort* could win a prize ...

Key points

recursion is not self-definition: we don't define *f* in terms of *f*, but $f(x)$ in terms of $f(y)$, where *y* is a smaller/simpler problem than *x*.

a recursion will terminate provided that the sequence of calls $f(x_0), f(x_1), \ldots$ which it generates always reaches $f(x_{triv})$, which can be solved without the use of *f* (there are other possible conditions, but in principle they are the same as this one).

sometimes algorithms are best presented recursively.

by the principle of *repeated halving*, any recursive method *f* which executes the sequence "*prepare*; *f*(one half); *f*(the other half); *tidyup*" will do its work in $O(N \lg N)$ time provided that (a) *prepare* and *tidyup* together are $O(N)$ and (b) $f(x_{triv})$ is $O(1)$.

merge (*mergehalves*) is $O(N)$ in execution time, $O(1)$ in space.

mergesort is $O(N \lg N)$ in execution time, $O(N)$ in space.

mergesort is faster but uses more space than insertion / selection / bubble sort, each of which is $O(1)$ in space but $O(N^2)$ in time.

mergesort is faster than *Shellsort* given a sufficiently large problem, but uses more space.

we can speed up *mergesort* quite a lot, but we will never beat the $O(N \lg N)$ time barrier, or the $O(N)$ space barrier, with the *mergesort* algorithm.