Come Together

This month we look at sorting large files

by Julian Bucknall

There's a newsgroup I frequent on a regular basis which is so secret that if you discovered its address from me I'd have to kill you. It's populated by Delphi programmers whose opinions seem perpendicular to each other. Flame wars burn bright then die. Every now and then, though, there's a message which stops me in my tracks and makes me realise I will continue to have a job writing about algorithms into the far future. There may be some truly amazing Delphi programmers out there, but sometimes their knowledge about the basics of computer science and our professional foundations is sadly lacking. So, stick with me and we'll learn together.

The latest message went a little like this (paraphrased to protect the innocent and extract the essential details): 'I needed to sort a file containing fixed length ASCII records. The file is 100Mb in size. I tried reading it into a sorted string list but after several hours I killed the program, as it didn't seem to be doing anything except grind my disk drive to dust. I then tried various data structures, linked lists and binary search trees in particular, and again the process was taking so long I killed it. Finally, in desperation, I dug out an old copy of Turbo Database Toolbox because I remembered it had a sort facility, converted it from DOS Pascal to 32-bit Delphi and used that.'

For those newbies amongst you who might never have heard of it, Turbo Database Toolbox was an implementation of a B-tree written for Turbo Pascal (one version appeared with Turbo Pascal 2, and it was updated for Turbo Pascal 3 and later). Essentially, it enabled you to write database applications by implementing a file system with index files. TurboPower's B-Tree Filer was a drop-in replacement with extra functionality, and it was because of my intimate knowledge of this latter product that I managed to get a job here over seven years ago.

Anyway, enough of the history. Database Toolbox had an implementation of a mergesort that enabled you to sort very large files very quickly and it was this code that my correspondent had ported and converted to Delphi 5. In this article, then, we'll talk about mergesort.

Yesterday

Way back in Issue 37 (September 1998), I discussed various sorting algorithms, ending up with quicksort. Quicksort is important because it is an O(*n*log*n*) algorithm: its speed is proportional to the number of items multiplied by the log of the number of items. There are a couple of other O(*n*log*n*) algorithms: heapsort (discussed in Issue 39) and mergesort, the subject of this article. (If you don't have the old issues, I recommend getting the *Collection 2000* CD-ROM.)

Quicksort has an infuriating problem: although in the general case the algorithm is extremely fast, there are other cases where the algorithm deteriorates into an O(n^2) process. Sometimes predicting this deterioration can be extremely hard. In Issue 37 I discussed various ways we could get around this problem: the median-of-three partitioning method and using an insertion sort if the partitions were small enough. Neither heapsort nor mergesort suffer from this problem: they are O(*n*log*n*) algorithms no matter what type of input data is fed to them. Mergesort has another important quality that makes it an important algorithm to master: it can be extended to efficiently sort data that cannot be fitted into memory. In other words, it is the algorithm of choice when you have to sort very large files. In this guise it is known as an external sort.

If you read old algorithms books, you'll find that mergesort is introduced in a peculiar fashion and, indeed, this is how I first met it. In these old books, it's always used to solve the problem of sorting data stored on tape when you don't have much main memory. Even when I wanted to use it the first time, tapes were only used for backup purposes and it took me a while to convert it to work with disk files.

Help!

If you think about it for a moment, you'll see that sorting a file that can't be read into memory is going to be a completely different animal to sorting a set of data that completely resides in memory. The reason is the time needed to read the data out of the file. This time is a complex function of the *seek time* (the time taken to move the disk's read/write head to the correct track on the disk), the *latency time* (the time taken in waiting for the correct sector to move under the read/write head as the disk platter spins), and the *transfer time* (the time taken to read the data from the disk into main memory). On a modern machine, this all seems to be instantaneous, but in reality it isn't: the CPU can do an *awful* lot of work whilst waiting for the data to arrive in memory from a read request. In practice, you can easily sort a block of data in the same time it takes to read the block off the disk. So an external sort has to worry about making sure that the use of the disk hardware is optimised; indeed, some of the more sophisticated sort variants make use of temporary files on different disks.

Of course, this process of reading a file also has an unknown component: the operating system. It may, in its wisdom, read blocks in advance of you requiring them. With Windows you can even give a hint to the system that this is how you want things to happen: you can state that you are going to read the file sequentially and the operating system will then stream the data in as you are using it.

So, mergesort, then. The idea behind this algorithm is to organize the file into larger and larger *runs* of records. A run of records is a set of records of some size that are sorted into order. We trivially assume that the original file is organized into runs of one record (this is trivial in the sense that a run of one record is automatically sorted). We then organize the file into runs of two records, then into runs of four records, then into runs of eight, and so on, moving up the powers of two. Eventually, we will reach a power of two that is greater than the number of records in the file, and at that point the file as a whole is then obviously sorted. This algorithm is usually known as the bottom-up mergesort.

An important point about bottom-up mergesort is this: we do not sort the file *in situ*. At the end of the mergesort algorithm, we will still have the original file extant, together with a file containing all the same records but in sorted order. During the sorting process, at the end of each stage of creating longer runs of sorted records, we will have three complete copies of the data, the original plus two others. So, if we had a 100Mb file as in my correspondent's example we would require a further 200Mb to efficiently sort the file.

In practice we make use of four ancillary files, each of which will contain roughly half the records during the mergesort process. At the end of the algorithm, one of these files will contain all the records, sorted in order; the others will be empty.

What Goes On

The first step is to read through the original file (which, if you remember, contains runs of 1 record), retrieving records in pairs. The first pair of records is written to temporary file F1 in order (forming a run of length 2), the second pair of records is written to temporary file F2 in order (again a run of length 2). The third pair of records is written to F1 again, the fourth pair to F2, and we continue like this, alternating between F1 and F2 until we have processed all the records.

After this first step we have two files, F1 and F2, containing runs of records of length 2.

The next step is to take these two files F1 and F2 and read records from them and create runs of length 4 in files G1 and G2. We read a record from F1 and one from F2, compare them and write the smaller to G1. If the record written was from F1, read another record from F1, otherwise read the next from F2. Compare the record we haven't written with the one we just read, and write the smaller one to G1. At this point we will have exhausted a run from one of F1 or F2 (in other words, we'll have read two records from one of these files) and therefore we can write the other two records to G1. We have created a run of 4 records in G1. Now we do the same with the next two runs from F1 and F2, but we write the run of 4 to G2. After this, we continue writing runs of 4 to G1 and G2 alternately until we've read through F1 and F2.

We now reset files F1 and F2 so that we start writing records from the beginning. We take the first runs of 4 from G1 and G2 and create a run of 8 in F1. The next two runs of 4 are merged into a run of 8 into F2, and then the next two runs of 4 are merged into a run of 8 in F1, and so on, so forth.

➤ *Listing 1: SplitFileFixed splits the input stream into two output streams.*

```
function ReadRecFixed(aStream : TStream; var aBuffer;
  aRecLen : integer) : boolean;
var
  BytesRead : longint;
begin
  BytesRead := aStream.Read(aBuffer, aRecLen);
  Result := BytesRead = aRecLen;
end;
procedure SplitFileFixed(aInFile : TStream; aF1 : TStream;
  aF2 : TStream; aRecLen : integer;
  aCompare: TaaMergeCompare);
const
  FirstFile = false;
  SecondFile = true;
var
  Rec1 : pointer;
  Rec2 : pointer;
  F    : array [boolean] of TStream;
  Have1st : boolean;
  Have2nd : boolean;
  Use1st1st : boolean;
  DestFile  : boolean;
begin
  F[FirstFile] := aF1;
  F[SecondFile] := aF2;
  Rec1 := nil;
  Rec2 := nil;
  try
    {allocate the record buffers}
    GetMem(Rec1, aRecLen);
    GetMem(Rec2, aRecLen);
    {we start out with the first output file}
    DestFile := FirstFile;
    {read the first two records}
    Have1st := ReadRecFixed(aInFile, Rec1^, aRecLen);
    Have2nd := ReadRecFixed(aInFile, Rec2^, aRecLen);
    {in a loop read the records in pairs and write them in
      sequence to the output file, alternating between
      output files; the loop stops when we can't read any
      more records}
    while Have1st do begin
      {order the two records}
      if Have2nd then
        Use1st1st := aCompare(Rec1, Rec2) <= 0
      else
        Use1st1st := true;
      {write them out in order to the current output file}
      if Use1st1st then begin
        F[DestFile].WriteBuffer(Rec1^, aRecLen);
        if Have2nd then
          F[DestFile].WriteBuffer(Rec2^, aRecLen);
      end else begin
        F[DestFile].WriteBuffer(Rec2^, aRecLen);
        F[DestFile].WriteBuffer(Rec1^, aRecLen);
      end;
      {switch output files}
      DestFile := not DestFile;
      {read the next two records}
      Have1st := ReadRecFixed(aInFile, Rec1^, aRecLen);
      Have2nd := ReadRecFixed(aInFile, Rec2^, aRecLen);
    end;
  finally
    if (Rec2 <> nil) then
      FreeMem(Rec2);
    if (Rec1 <> nil) then
      FreeMem(Rec1);
  end;
end;
```

This all seems a little wishy-washy, so let's be more practical. Like last time we'll use some cards to illustrate the algorithm. Extract out all the hearts from a deck of cards, shuffle and deal them into a line.

This is our original row, a typical shuffling of the hearts:

```
5 A 8 J 2 3 Q 6 K 4 9 10 7
```

Now we perform our first pass and separate the cards into two rows containing runs of 2. Reading from the left, we pick the cards up in pairs, order them and then put them down in two rows, alternating between the rows:

```
F1: A 5 | 2 3 | 4 K | 7
F2: 8 J | 6 Q | 9 10
```

(For ease of reading the runs, I have separated them with | characters). Now we move to the second process: continually merging these runs into longer and longer runs, alternating between rows.

Pick up the Ace and the 8. The Ace is smaller and hence gets placed into the G1 row. Pick up the 5 from F1 (we've now exhausted that particular run). It is less than 8, so it gets placed in the G1 row. We've no more cards in the current F1 run, so we merely place the 8 followed by the Jack in the G1 row. Now we switch to the G2 row, and do the same with both the next runs from F1 and F2. We continue like this until we reach:

```
G1: A 5 8 J | 4 9 10 K
G2: 2 3 6 Q | 7
```

We now have to merge the first two runs of 4 from G1 and G2 into a run of 8 in the F1 row. The next two runs are merged into G2 and so on. This penultimate pass produces runs of 8 like this:

```
F1: A 2 3 5 6 8 J Q
F2: 4 7 9 10 K
```

The final merge tries to produce runs of 16, but of course there are only 13 cards and so one of the rows becomes empty with all the cards in the other row:

```
G1: A 2 3 4 5 6 7 8 9 10 J Q K
G2:
```

Row G1 contains the sorted cards.

➤ *Listing 2: MergeRunsFixed merges two input streams into two output streams.*

```
function MergeRunsFixed(aF1 : TStream; aF2 : TStream;
  aG1 : TStream; aG2 : TStream; aRecLen : integer;
  aRunLen : integer; aCompare: TaaMergeCompare) : boolean;
const
  FirstFile = false;
  SecondFile = true;
type
  {record that describes processing of a single input file}
  TInputFile = packed record
    ifStrm      : TStream; {stream}
    ifRec       : pointer; {record buffer}
    ifRecsInRun : integer; {records to go in run}
    ifEOF       : boolean; {stream is exhausted}
  end;
var
  F : array[boolean] of TInputFile;
  G : array [boolean] of TStream;
  SrcFile   : boolean;
  DestFile  : boolean;
  FileId    : boolean;
begin
  {assume that this merge pass will finish completely}
  Result := true;
  {initialize the input file records}
  with F[FirstFile] do begin
    ifStrm := aF1;
    ifRec := nil;
    ifRecsInRun := 0;
    ifEOF := false;
  end;
  with F[SecondFile] do begin
    ifStrm := aF2;
    ifRec := nil;
    ifRecsInRun := 0;
    ifEOF := false;
  end;
  {set up the output files}
  G[FirstFile] := aG1;
  G[SecondFile] := aG2;
  try
    {clear the output streams}
    {NOTE: this only works for Delphi 3 and above, since
      only their TStreams have a SetSize accessor method}
    G[FirstFile].Size := 0;
    G[SecondFile].Size := 0;
    {reset the input streams, allocate the record buffers,
      and set the EOF flags}
    for FileId := FirstFile to SecondFile do
      with F[FileId] do begin
        ifStrm.Seek(0, soFromBeginning);
        GetMem(ifRec, aRecLen);
        ifEOF := ifStrm.Size = 0;
      end;
    {make sure the first output goes to G1}
    DestFile := FirstFile;
    {cycle until we manage to exhaust both input files}
    while (not F[FirstFile].ifEOF) or
          (not F[SecondFile].ifEOF) do begin
      {if we start writing to the second file, we won't
        finish the merge process this time}
      if (DestFile = SecondFile) then
        Result := false;
      {initialize ready for merging next runs}
      F[FirstFile].ifRecsInRun := aRunLen;
      F[SecondFile].ifRecsInRun := aRunLen;
      {read the first two records in the respective runs}
      with F[FirstFile] do
        if ReadRecFixed(ifStrm, ifRec^, aRecLen) then
          dec(ifRecsInRun)
        else begin
          ifRecsInRun := -1;
          ifEOF := true;
        end;
      with F[SecondFile] do
        if ReadRecFixed(ifStrm, ifRec^, aRecLen) then
          dec(ifRecsInRun)
        else begin
          ifRecsInRun := -1;
          ifEOF := true;
        end;
      {merge the two runs, one from F1 the other from F2}
      while ((F[FirstFile].ifRecsInRun >= 0) or
           (F[SecondFile].ifRecsInRun >= 0)) do begin
        {find the smaller record of the two current ones}
        {if the run from F1 is exhausted then the record
          from F2 is the 'smaller'}
        if (F[FirstFile].ifRecsInRun < 0) then
          SrcFile := SecondFile
        {if the run from F2 is exhausted then the record
          from F1 is the 'smaller'}
        else if (F[SecondFile].ifRecsInRun < 0) then
          SrcFile := FirstFile
        {otherwise we need to actually compare the records
          to find the smaller}
        else
          SrcFile := aCompare(F[FirstFile].ifRec,
                     F[SecondFile].ifRec) > 0;
        {write smaller record to current output file}
        G[DestFile].WriteBuffer(F[SrcFile].ifRec^, aRecLen);
        {read the next record from the file whose record
         we just used}
        with F[SrcFile] do
          if (ifRecsInRun <= 0) then
            ifRecsInRun := -1
          else if ReadRecFixed(ifStrm, ifRec^, aRecLen) then
            dec(ifRecsInRun)
          else begin
            ifRecsInRun := -1;
            ifEOF := true;
          end
      end;
      {having merged two runs, switch output files}
      DestFile := not DestFile;
    end;
  finally
    if (F[SecondFile].ifRec <> nil) then
      FreeMem(F[SecondFile].ifRec);
    if (F[FirstFile].ifRec <> nil) then
      FreeMem(F[FirstFile].ifRec);
  end;
end;
```

```
procedure aaMergesortFixed(const aInFile : string;
  const aOutFile : string; aRecLen : integer;
  aCompare : TaaMergeCompare);
var
  InFile : TFileStream;
  F      : array [1..2] of TaaTempFileStream;
  G      : array [1..2] of TaaTempFileStream;
  Merged : boolean;
  FIsSrc : boolean;
  RunLen : integer;
  Path   : string;
  MergedFileName : string;
begin
  InFile := nil;
  F[1] := nil;
  F[2] := nil;
  G[1] := nil;
  G[2] := nil;
  try
    {open the file to be sorted}
    InFile := TFileStream.Create(aInFile,
      fmOpenRead+fmShareDenyWrite);
    {split the file into the first two file components}
    Path := ExtractFilePath(aOutFile);
    {split the input file into two files containing runs of
      length 2}
    F[1] := TaaTempFileStream.Create(Path, fmOpenReadWrite);
    F[2] := TaaTempFileStream.Create(Path, fmOpenReadWrite);
    SplitFileFixed(InFile, F[1], F[2], aRecLen, aCompare);
    RunLen := 2;
    {perform the first merge pass}
    G[1] := TaaTempFileStream.Create(Path, fmOpenReadWrite);
    G[2] := TaaTempFileStream.Create(Path, fmOpenReadWrite);
    Merged := MergeRunsFixed(F[1], F[2], G[1], G[2],
      aRecLen, RunLen, aCompare);
    {now we continually merge the runs until we end up with
      a single file containing all the records}
    FIsSrc := true;
    while not Merged do begin
      RunLen := RunLen * 2;
      FIsSrc := not FIsSrc;
      if FIsSrc then
        Merged := MergeRunsFixed(F[1], F[2], G[1], G[2],
          aRecLen, RunLen, aCompare)
      else
        Merged := MergeRunsFixed(G[1], G[2], F[1], F[2],
          aRecLen, RunLen, aCompare);
    end;
    {we've now merged all records into either F1 or G1;
      rename file containing all records to output file
      name, and then delete the other three temporaries}
    if FIsSrc then begin
      MergedFileName := G[1].FileName;
      F[1].DeleteOnDestroy := true;
    end else begin
      MergedFileName := F[1].FileName;
      G[1].DeleteOnDestroy := true;
    end;
    F[2].DeleteOnDestroy := true;
    G[2].DeleteOnDestroy := true;
  finally
    G[2].Free;
    G[1].Free;
    F[2].Free;
    F[1].Free;
    InFile.Free;
  end;
  RenameFile(MergedFileName, aOutFile);
end;
```

➤ *Listing 3: The external mergesort.*

Tomorrow Never Knows

Looks pretty simple explained like this, but when writing the implementation the devil is in the details. The first process is to take the original file and to split it into two separate files, each containing runs of length 2. This is the purpose of the `SplitFileFixed` routine in Listing 1. As you can see we read the input file (here implemented as an instance of a stream) record by record into two buffers. We compare the two buffers by use of an external comparison routine and then write the smaller of the two to the output stream, followed by the larger. We then switch output streams. We continue this process until we have read all the records in the input file and have written them all to the two output files.

I make use of an ancillary routine called `ReadRecFixed` here to read a record, and to return true if the record was read, or false if the end of the stream was reached. This makes the code in `SplitFileFixed` a little easier to read.

The second process is to merge two files of run length x into two files of run length $2x$. We shall repeatedly call this routine, the first time with $x = 2$, until we have completely sorted the data. This process is shown by the `MergeRunsFixed` routine in Listing 2. It's more complex than `SplitFileFixed` mainly because we have to make sure we don't read beyond the end of a run when we're merging runs. In other words, should we be merging two runs of length 2, we don't want to read a third record from one of the files. Of course, we may also reach end-of-file when we're reading records as well, so we should cater for that as well (and of course reaching end-of-file also means reaching the end of a run).

`MergeRunsFixed` has another important function as well: it returns a `Boolean` value that states whether the records have all been merged into one output file, leaving the other empty. This is the signal to the caller that all the records have been sorted and that the mergesort is complete. This is simple enough to determine: if no records are written to the second output file during a call to `MergeRunsFixed`, the records have been completely merged and hence sorted.

Now we can bring the two routines together in a driver routine that performs the complete mergesort operation on a given input file to produce a sorted output file. Listing 3 shows this procedure. It takes the names of two files, the input file and the output file, and the length of the fixed length records contained in the input file. From this it creates the necessary streams, calls `SplitFileFixed` to start the whole process off, and then repeatedly calls `MergeRunsFixed` until the records are all sorted in one stream. It then renames this final output stream to the required file name.

I tested the efficiency of this routine on my home machine. I created a 25.6Mb input file containing 400,000 64-byte random records containing uppercase ASCII characters. I then sorted these records. It took 280 seconds (4.7 minutes). Not bad. Now, agreed, on my home machine I could probably have read this file into memory in one block, used quicksort on it and then written it out, but we're after experience with the algorithm.

I Should Have Known Better

This is pretty good, but is there anything we can do to improve matters? Well, the first point to make is much the same as one we made with the original quicksort: for small sequences of data, since there are an awful lot of them, it's better to use a different technique. For bottom-up mergesort this translates into two distinct improvements.

```
procedure SelectionSort(aBlock : pointer; aRecCount :
  integer; aRecLen : integer; aCompare : TaaMergeCompare);
var
  i, j    : integer;
  TempRec : pointer;
  iPtr    : PChar;
  jPtr    : PChar;
  MinPtr  : PChar;
begin
  GetMem(TempRec, aRecLen);
  try
    iPtr := aBlock;
    for i := 0 to (aRecCount - 2) do begin
      MinPtr := iPtr;
      jPtr := iPtr;
      for j := succ(i) to pred(aRecCount) do begin
        inc(jPtr, aRecLen);
        if (aCompare(jPtr, MinPtr) < 0) then
          MinPtr := jPtr;
      end;
      Move(iPtr^, TempRec^, aRecLen);
      Move(MinPtr^, iPtr^, aRecLen);
      Move(TempRec^, MinPtr^, aRecLen);
      inc(iPtr, aRecLen);
    end;
  finally
    FreeMem(TempRec);
  end;
end;
function SplitFileFixedBlock(aInFile : TStream;
  aF1 : TStream; aF2 : TStream; aRecLen : integer;
  aCompare : TaaMergeCompare) : integer;
const
  FirstFile = false;
  SecondFile = true;
var
  Block : pointer;
  F     : array [boolean] of TStream;
  DestFile  : boolean;
  BlockSize : integer;
  BytesRead   : longint;
  RecCount    : integer;
begin
  F[FirstFile] := aF1;
  F[SecondFile] := aF2;
  Block := nil;
  try
    Result := (128 * 1024) div aRecLen;
    BlockSize := Result * aRecLen;
    GetMem(Block, BlockSize);
    {we start out with the first output file}
    DestFile := FirstFile;
    {read the first block}
    BytesRead := aInFile.Read(Block^, BlockSize);
    RecCount := BytesRead div aRecLen;
    {in a loop sort the block and write it to the output
      file, alternating between output files; the loop stops
      when we can't read any more blocks}
    while (RecCount <> 0) do begin
      {sort the block}
      SelectionSort(Block, RecCount, aRecLen, aCompare);
      {write out sorted block to current output file}
      F[DestFile].WriteBuffer(Block^, BytesRead);
      {switch output files}
      DestFile := not DestFile;
      {read the next block}
      BytesRead := aInFile.Read(Block^, BlockSize);
      RecCount := BytesRead div aRecLen;
    end;
  finally
    if (Block <> nil) then
      FreeMem(Block);
  end;
end;
```

➤ *Listing 4: The improved block-oriented split routine.*

```
{split the input file into two files containing runs}
F[1] := TaaTempFileStream.Create(Path, fmOpenReadWrite);
F[2] := TaaTempFileStream.Create(Path, fmOpenReadWrite);
RunLen := SplitFileFixedBlock(InFile, F[1], F[2], aRecLen, aCompare);
```

➤ *Listing 5: The minor change required to call SplitFileFixedBlock.*

The first optimization is in the `SplitFilesFixed` routine: instead of creating runs of 2 records by reading records in pairs, read the records in blocks of several and quicksort them before writing them out into the two files F1 and F2. That means there will be many fewer calls to `MergeRunsFixed` since we'll start out with the original split files having larger run lengths.

For example, if we have 64-byte records, we could read them in blocks of 64, say (ie, 4,096 bytes), quicksort them (or selection or insertion sort them, there's not that many after all) and write them out to files F1 and F2 alternately. We would then start out calling `MergeRuns` with files containing run lengths of 64 instead of 2. We'd save 6 calls to `MergeRuns` (for run lengths 2, 4, 8, and so on), which translates to saving 6 complete reads through the set of records, a valuable saving.

The other optimisation we could do is in reading the records from our temporary files in blocks instead of one at a time. This would in essence cache the records. For Windows this probably wouldn't help too much: a better option would be to signal the operating system that we want to sequentially read the file and the system will be on our side, caching for us. That way we don't have to write a lot of file caching code.

The changes to `SplitFileFixed` are shown in Listing 4. As you can see, instead of reading the records singly, we read them in blocks. We size the blocks such that each is roughly 128Kb in size and contains an integral number of records. We pass this run length back to the caller so it knows where to start when calling `MergeRunsFixed`. Listing 5 shows the minor change required. I decided to use a selection sort this time, although a quicksort might be faster. Selection sort works very well when the cost of comparing two items is much smaller than the cost of exchanging them into their respective places.

Another test: this time sorting 400,000 random 64-byte records took 150 seconds or 2.5 minutes: a vast improvement, showing that removing a lot of the file access makes great sense.

I'll Be Back

By the way, I noticed that as I timed various tests that the tests tended to get longer and longer. The disk tended to thrash a little more. I'm guessing this is due to the increasing fragmentation of the contents of my disk as I experimented. I'll test this theory when I have more time; for now, Our Esteemed Editor is champing at the bit to get this article typeset and I can't extend my deadline any more! I hope you enjoyed this foray into external mergesort: it's an important algorithm that many don't know and yet should be a staple in our programming toolbox.

Julian Bucknall is looking forward to being more organized, having finished a stint in the play that can be briefly heard at the end of 'I Am The Walrus'. Email Julian at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.
© *Julian M Bucknall, 2000*