QUANTA HELPLINE — Dilwyn Jones

Members wishing to submit helpline requests via email can use the email address helpline@quanta.org.uk or if you prefer to use traditional post, please send the helpline request to me via the address printed inside the front cover of the newsletter.

Obviously, we cannot guarantee to answer every query we receive, but we will do our best! Where we have been unable to answer the queries, we may print the help request as an open request in the newsletter to ask if any of the readers can come up with a solution. And, of course, if readers feel that they have a better solution than we came up with, or would like to correct any errors we make, please write to us!

Merge Sorting, Duplicates and Binary Searching

For this issue, I'm going to discuss merge sorting files, searching lists and eliminating duplicates. I received what appeared to be a simple and straightforward query via the helpline a few weeks back and didn't realise at first how much work would be involved in researching it! As you'll see below, the reply turned into quite a full-blown article.

The original question was:

"I have a number of word list files containing thousands of words, each sorted into alphabetical order. What I'd like to do is to join these files together into one sorted list and eliminate any duplicates resulting from joining together the lists, then add my own words to the list. I've tried using various QL text editors and word processors and none really seems to manage such long lists of words."

Dealing with the last sentence of the query first – the usual reason is that since word lists are usually a text file list with an end of line character at the end of each word, word processors in particular see these lines as very short paragraphs. For example, here's a simple list:

and
but
either
or

In a plain text file on the QL, these words are all followed by a linefeed character, otherwise known as CHR$(10). Word processors usually work by paragraphs, not individual lines of text. So short lines quickly eat up the available number of paragraphs.

Some QL text editors and word processors use a 16 bit line counter value, limiting the program to an absolute maximum of 32,767 entries (or about 65535 if using unsigned arithmetic). Word lists of the type described, especially if to be used with a spell checker program, for example, can easily consist of 20,000 to 50,000 words and if you join these together, you will quickly see that word processors in particular will quickly struggle to control this number of paragraphs, however small they might be! Equally, if the lists are to be handled by a BASIC or compiled BASIC program, with the words held in arrays, you will just as quickly run into problems as the maximum number of array entries you can get into a string array is also about 32,767 entries and anyway, just imagine the time it would take to try to sort such a huge number of strings in BASIC. You could try using Steve Poole's arborescent sort routines previously published in Quanta, or Alan Turnbull's Quicksort on Library disk UG01, or even Ron Dwight's machine code sorting routines on the same disk, but any sort of this nature will only be able to cope with the maximum possible dimensions of string arrays in BASIC.

So we need to look at this from a different perspective.

Merge Sort

As the word lists are already in a sorted file (I'm presuming these are the kind of word lists you can download from the internet, e.g. from Geoff Wicks's website at http://members.multimania.co.uk/geoffwicks/dictionaries.htm or from my website at http://www.dilwyn.me.uk/diction/index.html) one simple way of joining the files and sorting while merging is to use something called a Merge Sort. Basically what this does is to take one entry from each file and write the appropriate entry (usually the lowest in alphabetical order) to a new output file, then read another entry from the same input file and keep doing this until the supply of words from one file is exhausted, then write out the remainder of the other file until that too is exhausted. And so am I after typing that – don't worry, it's actually easier than it sounds!

At this point I'd like to express my gratitude to Miguel Angel Rodriguez Jodar who works as an associate professor at the Architecture and Computer Technology Dept., University of Seville, Spain. Besides his academic duties, he runs a small computer museum, placed at the main hall of the Computing Science High College facilities. Miguel also has a website for people who tinker mostly with Spectrum hardware projects at http://www.zxprojects.com and he also takes an interest in QL systems. I asked for help with this query on the ql-users mailing list and true to the helpful nature of that list, Miguel popped up to offer me a "pseudo code" listing (not a QL BASIC listing, although fairly similar) which I could use as the basis for a QL BASIC routine.

A little bit of QL BASIC programming later and here's what I came up with. The listing should be fairly self-explanatory – I've included a lot of REMark statements to explain what each part does. Please note: both input files must already have been sorted into alphabetical order.

```
1000 DEFine PROCedure Merge_Sort_Files (inputfile1$,inputfile2$,outputfile$)
1010   LOCal x$,y$,loop1
1020   OPEN_IN  #3,inputfile1$
1030   OPEN_IN  #4,inputfile2$
1040   OPEN_NEW #5,outputfile$
1045   :
1050   IF EOF(#3) OR EOF(#4) THEN
1060     REMark oops, one file is empty before we've even started!
1070     IF EOF(#3) THEN
1080       Copy_Remainder #4 TO #5 : REMark file1 is empty
1090     ELSE
1100       Copy_Remainder #3 TO #5 : REMark file2 is empty
1110     END IF
1120   ELSE
1130     REMark neither file empty, so get first entry from both files
1140     INPUT #3,x$
1150     INPUT #4,y$
1155     :
1160     REPeat merging
1170       REMark compare x$ and y$
1180       IF x$ < y$ THEN
1190         REMark element x$ is smaller; write x$ to output file
1200         REMark and read new x$ provided there is any;
1210         REMark otherwise copy the rest of file2 to output file
1220         PRINT #5,x$
1230         IF NOT EOF(#3) THEN
1240           INPUT #3,x$
1250         ELSE
1260           PRINT #5,y$ : Copy_Remainder #4 TO #5 : EXIT merging
1270         END IF
1280       ELSE
1290         REMark element y$ is smaller; write y$ to output file
1300         REMark and read new y$ provided there is any
1310         REMark otherwise copy the rest of file1 to output file
1320         PRINT #5,y$
1330         IF NOT EOF(#4) THEN
1340           INPUT #4,y$
1350         ELSE
1360           PRINT #5,x$ : Copy_Remainder #3 TO #5 : EXIT merging
```

```
1370         END IF
1380       END IF
1390     END REPeat merging
1400   END IF
1410   CLOSE #3 : REMark input file1
1420   CLOSE #4 : REMark input file2
1430   CLOSE #5 : REMark output file
1440 END DEFine Merge_Sort_Files
1450 :
1460 DEFine PROCedure Copy_Remainder (ip_chan,op_chan)
1470   LOCal copying,y$
1480   REPeat copying
1490     IF EOF(#ip_chan) THEN EXIT copying
1500     INPUT #ip_chan,y$
1510     PRINT #op_chan,y$
1520   END REPeat copying
1530 END DEFine Copy_Remainder
```

LISTING 1 : mergesort2_bas

So, that solves the first part of the problem – to merge sort the two input files, just enter the command Merge_Sort_Files 'file1','file2','output_file' (enter the relevant filenames in place of 'file1', 'file2' and 'output_file' of course).

Eliminate Duplicates

The next step is to eliminate duplicates from the merged list. I'll treat this a separate programming issue just to simplify the matter. We need to remember that what we have done is to merge two text files into one, both of which were already sorted, and the newly created merged file is also in alphabetical order.

The easiest way of doing this is to copy all of the entries from the new file into yet another new file. We remember what the last entry was, and if the next entry is the same, we simply don't copy it.

```
100 DEFine PROCedure Eliminate_Duplicates (original_file$,new_file$)
110   LOCal copying,word$
120   OPEN_IN #5,original_file$
130   OPEN_NEW #6,new_file$
140   previous$ = '' : REMark remember what previous entry was
150   REPeat copying
160     IF EOF(#5) THEN EXIT copying : REMark all done
170     INPUT #5,word$
180     IF NOT(word$ == previous$) THEN
190       PRINT #6,word$
200       previous$ = word$
210     END IF
220   END REPeat copying
230   CLOSE #5 : CLOSE #6
240 END DEFine Eliminate_Duplicates
```

LISTING 2 : eliminateduplicates_bas

So if we wanted to eliminate duplicate entries from a file called "mergedwords_txt" we would use the above routine to copy the file to a new file called "noduplicates_txt" as follows, assuming that both files are/will be in ram drive 1:

Eliminate_Duplicates "ram1_mergedwords_txt" TO "ram1_noduplicates_txt"

Note how I use the keyword TO instead of a comma between the filenames. You can use either. I just find the keyword TO makes it easier to read – more meaningful. This is one of the great things about QL BASIC. Another is the use of the "approximately equal to" operator, which allows strings to be compared irrespective of case, that is, it allows the QL to treat DILWYN and DiLwYn as being equal. Very useful – it helps prevent you having to convert all

string text to the same case for comparison purposes, using functions like UPPER$ and LOWER$ available in some toolkits.

The mergesort routine above copes with having some words in the files in what is called Mixed Case, i.e. proper nouns with the first letter in upper case. It simply puts upper case words first.

Adding New Entries

The last part of this project is to allow new words to be added to the list. I'll look at a couple of ways of doing this, both of which have their limitations, but at least should offer me the chance to explain searching through sorted files to locate matching entries and how to determine where the new entry should go.

First, I'll look at the simplest brute force approach! Basically, it's as simple as copying the entire file until we find the point at which the new word should go, insert the new word at that point, then copying the remainder of the file to the new file. The trouble with this type of brute force approach is that (a) we have to search through the entire file for every word entered, which might take a long time if it's a word which comes late in the alphabet, such as the word 'zebra', and (b) we have to copy to a new file each time, so it needs twice as much space.

Here is one way of achieving this. We take the 'safe' approach of copying the amended file to a new name, before deleting the original, copying the new file to the original name and once all that's been successful, delete the newly created temporary file.

```
100 REMark add a new entry to a file
110 CLS : CLS #0
120 INPUT #0,'Word list filename > ';ip$
130 INPUT #0,'Name of temporary file > ';op$
140 REPeat program
150   CLS # 0
160   INPUT #0,'New word > ';word$
170   IF word$ = '' THEN EXIT program
180   OPEN_IN #3,ip$
190   OPEN_NEW #4,op$
200   found% = 0
210   REPeat write_out
220     IF EOF(#3) THEN EXIT write_out
230     INPUT #3,str$
240     IF str$ == word$ AND found% = 0 THEN found% = 1
250     IF found% = 0 THEN
260       IF str$ > word$ THEN
270         REMark found where to add the new word
280         PRINT #4,word$
290         found% = 1
300       END IF
310     END IF
320     PRINT #4,str$
330   END REPeat write_out
340   IF found% = 0 THEN PRINT #4,word$
350   CLOSE #3
360   CLOSE #4
370   REMark change filename back to original
380   DELETE ip$
390   COPY op$ TO ip$
400   DELETE op$
410 END REPeat program
```

LISTING 3 : addwordstolist_bas

The above approach works, but is slow and clumsy, although it can (slowly) handle very large word lists subject to enough space being available on the media to hold both the original and temporary new file.

An easier way, if the list is not too long to fit into a string array, is to load the file into a string array and use a method known as a “binary chop” to locate where the new word should be added, or indeed if the word already exists in the array. This involves starting to look halfway through the list and see which half the word is likely to belong in. Having worked that out, we then split that half into a further pair of halves and repeat the process until we find the required point.

```
100 REMark using binary search to add data to a pre-sorted array
110 REMark array is called array$()
120 max%    = 500 : REMark maximum number of entries allowed
130 widest% = 20  : REMark longest word 10 characters long
140 DIM array$(max%-1,widest%)
150 :
160 number% = 0   : REMark how many entries currently in the list
170 CLS : CLS #0
180 :
190 INPUT #0,'Load which file > ';ip$
200 IF ip$ <> '' THEN
210   OPEN_IN #3,ip$
220   REPeat loop
230     IF EOF(#3) THEN EXIT loop
240     INPUT #3,array$(number%)
250     number% = number% + 1
260     IF number% >= max% THEN EXIT loop
270   END REPeat loop
280   CLOSE #3
290 END IF
300 PRINT number%;' entries in list so far.'
310 :
320 REMark enter new words to add to list (unless already in list)
330 added% = 0 : REMark running track of number of words added so far
340 REPeat program
350   IF number% >= max% THEN PRINT'Array full.' : EXIT program : REMark no
room for more
360   INPUT #0,'Word to add > ';word$
370   IF word$ = '' THEN EXIT program
380   Add_Entry word$
390 END REPeat program
400 :
410 REMark save the updated list (use same name if a list was loaded)
420 PRINT \ 'Number of added entries : ';added%
430 IF added% > 0 THEN
440   op$ = ip$
450   IF op$ = '' THEN INPUT #0,'Save as > ';op$
460   IF op$ <> '' THEN
470     PRINT #0,'Saving ';op$;' ...'
480     OPEN_NEW #3,op$
490     PRINT #3,array$(0 TO number%-1)
500     CLOSE #3
510   END IF
520 END IF
530 :
540 PRINT #0,'Program finished.'
550 STOP
560 :
570 DEFine PROCedure Add_Entry (new_word$)
580   LOCal lo%,mid%,hi%,loop,a
590   REMark if list empty, just insert at start
600   IF number% = 0 THEN
610     array$(0) = new_word$ : number% = 1
620     PRINT '"';new_word$;'" added. Total entries = ';number%
630     added% = added% + 1 : RETurn
640   END IF
650   :
660   REMark binary search for insertion point
670   lo% = 0         : REMark lowest subscript
680   hi% = number%-1 : REMark highest subscript
```

```
690   REPeat loop
700     mid% = (lo%+hi%) DIV 2
710     IF Lower_Case$(new_word$) < Lower_Case$(array$(mid%)) THEN
720       hi% = mid% - 1 : IF lo% > hi% THEN EXIT loop
730     ELSE
740       lo% = mid% + 1 : IF lo% > hi% THEN mid% = lo% : EXIT loop
750     END IF
760   END REPeat loop
770   :
780   IF mid% > 0 THEN
790     REMark does the new word already exist in the list?
800     IF array$(mid%-1) == new_word$ THEN
810       PRINT '"';new_word$;'" is already in the list.'
820       RETurn
830     END IF
840   END IF
850   :
860   REMark shuffle up to make room for new word in correct place
870   FOR a = number% TO mid%+1 STEP -1 : array$(a) = array$(a-1)
880   array$(mid%) = new_word$
890   number% = number% + 1 : added% = added% + 1
900   PRINT '"';new_word$;'" added. Total entries = ';number%
910 END DEFine Add_Entry
920 :
930 DEFine FuNction Lower_Case$ (str$)
940   LOCal a,cde,t$
950   t$ = str$
960   FOR a = 1 TO LEN(str$)
970     cde = CODE(t$(a))
980     IF cde >= 65 AND cde <= 90 THEN cde = cde + 32 : t$(a) = CHR$(cde)
990   END FOR a
1000   RETurn t$
1010 END DEFine Lower_Case$
```

Listing 3 : binarysearch_bas

This listing is a bit longer than the others, because it's a complete program to generate a word list, although limited in the number of words it can handle. I've set the limit as 500 words in line 120, each of no more than 20 characters long, set in line 130. Alter these if you wish to adapt it for larger word lists.

What it does is to ask you if you wish to enter a 'base' file. If not and you wish to start a new word list, just press ENTER to make a blank filename in line 190.

Now keep entering words until you wish to finish – enter a blank word to finish. The program keeps a running count of the total number of words in the list using the variable 'number%'. It also keep a running total of the number of new words added in this session – the variable 'added%'.

Once you have entered a word in line 360, it then calls the procedure Add_Entry to see if the word should be added to the list or not. This procedure starts at line 570. It performs the following actions:

1. If the list is empty, it simply adds the word as the first entry and returns (lines 600 to 640).
2. If not empty, it defines a set of pointers, marking the lower bound of the section (lo%), upper bound of the section (hi%) and a median pointer (mid%) which it tries to set at about halfway between the two bounds. When comparing strings, it uses the Lower_Case$ function defined in lines 930 to 1010 to ensure that all comparisons are done in the same case – if one of your toolkits has an extension called LOWER$ or equivalent to do this, use that as it will be faster than a simple BASIC function like this. Depending on the result of the comparison made in line 710, the pointers to the bounds are adjusted accordingly until lo% becomes greater than hi%. When this

happens, it knows it has found the entry just above where your new word would go in the file.

3. Now that we have found where the word would go, we check the entry just below this (if there is one – line 780). If this is the same as our new word (line 800) we tell the user that the word already exists in the list and don't add the new word. Note the use of '==' to ensure case independent comparison.
4. If the word is not already in the file, we shuffle the part of the array above where the new word would go up by one position in the array (line 870) - note how we do this backward from the top of the array using the "STEP -1" to avoid accidentally overwriting everything! Finally, line 880 places the new word into the list and we increment the total number of words and the number of new words added (line 890) before returning to ask for the next word to be entered.

Whilst the listing is quite long, do bear in mind that it is a complete working program and the search routine we are interested in is mainly just the code in lines 670 to 760.

If the limit imposed by how large you can dimension QL string arrays is restrictive, what you would then have to is to consider storing the list as fixed length entries in an allocated block of memory, change the pointers to be floating point values rather than integers and try to manipulate the strings in memory as though they were part of an array by using memory string peeks and pokes and using move memory commands to do the shuffling of entries. SBASIC has PEEK$ and POKE$ functions to help you do this. Sadly, it's beyond the scope of this article and not a programming job for the faint-hearted!

Conclusion

I hope that this article has been useful to some readers – we have covered a lot of programming ground and it's probably a bit much to take on in just in one go. The programming techniques should prove useful to those wishing to handle textual data in large files like this.

If anyone has ideas on how to improve these routines, or to take them a stage further, I'd be pleased to hear from you and publish your ideas in a future issue.

In the meantime, I'll try to make sure that the listings are placed on the website and library disk ML01 (magazine listings) for those who'd rather save their typing fingers.