PROGRAMMER'S CHALLENGE

by Bob Boonstra, Westford, MA

BESORT

The theme of this issue of the magazine is the Be Operating System, so it seemed appropriate to focus the Challenge on the BeOS and give you a chance to use the BeOS for Macintosh CD-ROM bundled in this issue. Since the BeOS will be new to most of you, this Challenge will be a simple one. The problem is to write a window class that will display and sort a list of strings by one of three specified methods: a bubble sort, an exchange sort, and an algorithm of your choosing. The prototype for the class you should write is:

```
typedef enum SortType {
  kBubbleSort = 1,
  kExchangeSort = 2,
  kMySort = 3
} SortType;

class SortWindow : public BWindow {

public:
                SortWindow(BRect frame);
virtual void  DoSort(
    char *thingsToSort[],  /* list of strings to sort, also returns sorted list */
    int numberOfThings,      /* number of thingsToSort */
    SortType sortMethod);    /* sort method to use */
};
```

My test code will open three instances of your window class and ask each one to sort a copy of the same list of strings, one by the kBubbleSort method, one by kExchangeSort, and one by kMySort. Your SortWindow constructor should create a BListView, and attach it to your SortWindow. When the DoSort method is invoked, you should display the thingsToSort, and sort them into ascending ASCII order by the sortMethod algorithm. Each time two thingsToSort are exchanged, the BListView display should be updated. When the sort is complete, DoSort should post a B_QUIT_REQUESTED message to the application. The list should be sorted in place and returned in thingsToSort.

This will be a native PowerPC Challenge, using the latest Macintosh CodeWarrior environment, targeted for the BeOS. Solutions must be coded in C++. The code will be tested on my 8500 using the BeOS. (In the event I cannot get the BeOS to run on my Mac, I will run the tests on a 2x133MHz BeBox with one processor disabled.) The winner will be the solution that completes all three sorts correctly in the minimum time.

THREE MONTHS AGO WINNER

Congratulations to **Andy Antoniewicz** (Mountain View, CA) for narrowly beating second and third place finishers Greg Cooper and Ludovic Nicolle in the October DNA Match Challenge. Of the fifteen entries I received for this Challenge, ten worked completely correctly, two were partially correct, and the remaining three crashed my machine.

Recall that the DNA Match Challenge was essentially a string matching problem, where the strings were allowed to differ in a specified number of positions (or fewer). The object was to return the number of near matches of a fragment string found in a reference database string.

My intent had been to test the solutions submitted using very long database strings. The run times of the solutions imposed a practical limit of about 2 MB on the size of the database string in an individual test case. The fragments to be matched were all significantly shorter than the database string, as indicated in the problem statement. The tests ranged from requiring very accurate matches, with a small value for the number of differences allowed, to approximate matches that could differ in up to half of the characters in the reference string.

The problem statement allowed for a timed initialization routine that would be executed once prior to testing matches against multiple fragments for a given database string. None of the top-ranked solutions made any significant use of this option, although a number of people used it to initialize small translation tables. Several people commented that it was difficult to find a use for this initialization routine when the scratch storage provided was smaller than the maximum size database string.

Andy's winning entry parses the fragment string to create, for each character in the DNA "alphabet", a list of offsets where that character is located in the fragment. He then walks the database string and increments a match counter for each possible alignment of the fragment that matches the database at that character position. Andy uses a circular buffer twice the size of the fragment to store the match counts, which allows him to perform bounds checking on that buffer only once per database character rather than within the innermost character matching loop. I had to read the code several times and run through a few cases manually before the light went on and I understood the algorithm, after which I found it quite clever.

While A.C.C. Murphy's solution did not place in the top five, one of his algorithms used a refinement worthy of note. He kept a running total of the counts of characters in a fragment-sized segment of the database, only checking for a specific match if the frequency counts were close enough to those of the fragment. In test cases where the character frequencies of the fragment were significantly different than much of the database, this technique might have done very well.

The table below summarizes the results for each correct or partially correct entry, including total execution time for all of the test cases and code size. Numbers in parenthesis after a person's name indicate that person's cumulative point total for all previous Challenges, not including this one. An asterisk indicates a result that was partially correct and therefore not eligible to win.

Name	Language	Total Time	Code Size
Andy Antoniewicz (4)	C	123823	728
Greg Cooper (17)	C	126233	872
Ludovic Nicolle (14)	C	126646	528
Michael Panchenko (6)	C	148243	800
Bjorn Davidsson (4)	C	149171	424
A.C.C. Murphy (10)	C	151086	1800
Ernst Munter (224)	C++	176521	752
Peter Lewis (32)	C	195312	240
Mark Day	C	197385	848
Larry Landry (29)	C	267118	1376
Alan Hart (*)	C	210701	848
Xin Xu (*)	C	223015	1576

TOP 20 CONTESTANTS

Here are the Top 20 Contestants for the Programmer’s Challenge. The numbers below include points awarded over the 24 most recent contests, including points earned by this month's entrants.

Rank	Name	Points	Rank	Name	Points

1.	Munter, Ernst	193	11.	Kasparian, Raffi	22
2.	Gregg, Xan	114	12.	Cutts, Kevin	21
3.	Larsson, Gustav	87	13.	Nicolle, Ludovic	21
4.	Lengyel, Eric	40	14.	Picao, Miguel Cruz	21
5.	[Name deleted]	40	15.	Brown, Jorg	20
6.	Lewis, Peter	32	16.	Gundrum, Eric	20
7.	Boring, Randy	27	17.	Karsh, Bill	19
8.	Cooper, Greg	27	18.	Stenger, Allen	19
9.	Antoniewicz, Andy	24	19.	Mallett, Jeff	17
10.	Beith, Gary	24	20.	Nevard, John	17

There are three ways to earn points: (1) scoring in the top 5 of any Challenge, (2) being the first person to find a bug in a published winning solution or, (3) being the first person to suggest a Challenge that I use. The points you can win are:

1st place	20 points	5th place	2 points
2nd place	10 points	finding bug	2 points
3rd place	7 points	suggesting Challenge	2 points
4th place	4 points		

Here is Andy's winning solution:

DNA_MATCH.C


```
/*
   Problem:
      DNA string match with wildcard & constant distance

   Notes:
   No setup calculations on the database are performed.
   It seemed kind of pointless to attempt the equivalent
   of a 50 to 1 loss, less compression of the database
   ( it was 1000 to 1 for the first problem statement ).

   This is a simple and quick single byte per pass index
   and count type algorithm. It uses three tables:
      a byte to code index table "aTable",
      a fragment index list array "alphaList",
      and a hit count circular array "matchQueue".
   A precursor list is built and used once to build the
   alphaList array, and I do not clean up after (plenty
   of allocated storage).

   Once built, the aTable contains an index into the
   alphaList for each given alphabet letter in the
   fragment. The alphaList contains a sequential list of
   all occurrences of that character in the fragment
   string.

   The algorithm then increments the match count for all
   possible alignments of the fragment for each database
   character. Since the maximum misalignment is less
   than the fragment size, only the previous fragmentSize
   database characters need to be considered for matches
```

at any particular time. Hence the circular queue.

The database search execution time is order(pN) where
N = number of database characters
p = average fragment entries per alphabet char
(for example: A=3,C=3,G=2,T=2 --> p = 2.5)

The storage used is almost totally dependent on the
fragment size. Storage used in bytes
= 20 (storage struct)
+ 256 (16bit aTable)
+ 2 * fragment chars (16bit precursor)
+ 2 * fragment chars (16bit alphaList)
+ 2 * alphabet chars (more alphaList)
+ 4 * fragment chars (2x16bit matchQueue)

```
********* ********* ********* ********* ********* ********/

#define kAlphaTableSize  128
#define kEndOfAlphaList   -1

typedef struct {
  long storageSize;     // total storage available
  long usedStorage;     // total storage used
  long fragmentSize;    // size in chars of fragment
  short *alphaList;     // start of fragment index list
  short *matchQueue;     // start of match count queue

  short alphaTable[ kAlphaTableSize ];
  short precursor[];     // start of match count queue
} DNAStore;

/********* ********* ********* ********* ********* ********
  Function Prototypes
 ********* ********* ********* ********* ********* ********/

void InitMatch(
  char *alphabet,   // legal characters in database
  char *database,   // the reference database
  void *storage,    // pre-allocated storage
  long storageSize    // size of storage in bytes
);

long FindMatch(    // return number of matches
  char *alphabet,     // legal characters in database
  char *database,     // the reference database
  void *storage,    // pre-allocated storage
  char *fragment,     // the fragment to find
  long diffsAllowed,      // num of diffs allowed between
            //  a "match" and the database
  long matchPosition[]  // match return array
);

void BuildAlphaList(
  char    *alphabet, // legal characters in database
  char  *fragment,   // the fragment to find; 0 term
  DNAStore *storage      // my storage area
```

```
);

/********* ********* ********* ********* ********* ********
 * InitMatch
 *    This routine does nothing except to store the
 *    size of memory allocated by the calling program.
 *    All of the structures used are based on the fragment
 *    to be searched for.
 ********* ********* ********* ********* ********* ********/
```

InitMatch

```
void InitMatch(
  char *alphabet,   // legal characters in database
  char *database,   // the reference database
  void *storage,      // pre-allocated storage
  long storageSize    // size of storage in bytes
)
{
  ((DNAStore*)storage)->storageSize = storageSize;

} // end of InitMatch


/********* ********* ********* ********* ********* ********
 * BuildAlphaList
 *
 * This routine has three parts:
 *    Build the AlphaTable
 *       The AlphaTable is an index table used to find
 *       the start location inside the AlphaList
 *       that corresponds to the given alphabet
 *       character.
 *    Build the AlphaList precursor
 *       This is a linked list of the indexes to each
 *       character in the fragment. It is used once
 *       to build the AlphaList and never re-used.
 *    Build the AlphaList
 *       This is a sorted list of indexes for each
 *       alphabet character. The head of each list is
 *       stored in the AlphaTable, and each list is
 *       ended by a -1.
 *
 ********* ********* ********* ********* ********* ********/
```

BuildAlphaList

```
void BuildAlphaList(
  char    *alphabet,    // legal characters in database
  char  *fragment,      // the fragment to find; 0 term
  DNAStore *storage       // my storage area
)
{
  short *aTable;        // ptr in the AlphaTable
  short *precursor;     // the AlphaList precursor
  short *alphaList;     // the AlphaList

  char *aString;        // ptr to a character string
  long aChar;           // a character from a string
```

```
 long count;
 short index;

/*********
 * Initialize alphabet entries to kEndOfAlphaList
 * Proper function for characters not in the alphabet
 * requires that the other entries be preset to zero.
 * This is automatic if the storage is cleared by
 * the calling program and if the alphabet does not
 * change between calls to InitMatch.
 */
 aTable  = storage->alphaTable;
 aString = alphabet - 1;
 while( (aChar = (long)(*(++aString)) ) > 0x00 )
 {
   *(aTable+aChar) = kEndOfAlphaList;
 }

/*********
 * Build precursor linked index list
 * This list is only used to produce the
 * AlphaList below. The precursor and
 * the AlphaList are rebuilt for each
 * new fragment that will be searched for.
 */
 precursor  = storage->precursor;
 aString  = fragment - 1;
 index   = 0;
 count   = 0;
 while( (aChar = (long)(*(++aString))) > 0x00 )
 {
   index = *(aTable+aChar);  // get prev head
   *(aTable+aChar) = count;  // put new head
   *(precursor+count) = index;        // store prev head
   count++;
 }
 storage->fragmentSize = count;

/*********
 * Build AlphaList
 * by walking each alphabet character's precursor
 * list and writing the index list into the
 * AlphaList the algorithm gets a sorted list of
 * indexes into the fragment for each letter in the
 * alphabet. The aTable will point to the first entry
 * in the AlphaList and the last entry of each
 * character list is equal to the constant
 * kEndOfAlphaList ( = -1 ). Note that AlphaList
 * location 0 is used to ignore database characters
 * which are not in the given alphabet (it was free).
 */
 alphaList = precursor + count;
 *(alphaList) = kEndOfAlphaList;
 aString = alphabet - 1;
 count = 1;
 while( (aChar = (long)(*(++aString)) ) > 0x00 )
 {
```

```
    index = *(aTable+aChar);
    *(aTable+aChar) = count;
    while( index != kEndOfAlphaList )
    {
      *(alphaList+count) = index;
      index = *(precursor+index);
      count++;
    }
    *(alphaList+count) = kEndOfAlphaList;
    count++;
  }
  storage->alphaList  = alphaList;
  storage->matchQueue = alphaList + count + 1;

} // end of BuildAlphaList


/********* ********* ********* ********* ********* ********
 * FindMatch
 *
 * For each character in the database increment every
 * match count that has the same character ( a hit ).
 * If after all possible alignments have been tallied,
 * the match count is greater than or equal to the value
 * of the threshold, then a match has been found.
 * Add the matching entry to the return array and continue
 * until all database characters have been tested.
 *
 ********* ********* ********* ********* ********* ********/
```

FindMatch

```
long FindMatch(          // return number of matches
  char *alphabet,      // legal characters in database
  char *database,      // the reference database
  void *storage,       // pre-allocated storage
  char *fragment,      // the fragment to find
  long diffsAllowed,   // num of diffs allowed between
                       //  a "match" and the fragment
  long matchPosition[] // match return array
)
{
  DNAStore *theStore; // typed storage
  short   *matchTop;   // top of match array
  short   *matchCur;   // current match array entry
  short   *aTable;       // alpha to aList index table
  short   *aList;      // array of hit offsets
  short   *hitOffset;    // current hit offset entry
  char  *dbString;   // current database char entry
  long  dbChar;      // current database character
  long  count;         // current database location
  long  numMatch;      // number of matches found
  long  fragSize;      // fragment size in bytes
  long  loop;          // loop counter
  long  hitPos;      // current hit offset position
  long  threshold;   // the threshold for matching

  theStore = (DNAStore*)storage;
```

```
BuildAlphaList( alphabet,fragment,theStore);
fragSize = theStore->fragmentSize;
matchTop = theStore->matchQueue;
matchCur = matchTop + fragSize;

theStore->usedStorage
   = (long)(matchCur+fragSize)-(long)theStore;
/*********
 * Clear the Match Count Array
 *  clear out the match counts for the entire array
 */
for( count=0; count<fragSize; count++ )
{
  *(matchCur+fragSize) = 0;
  *matchCur-- = 0;
}

/*********
 * Walk the Database
 *
 * The match count array is a double size circular
 * queue of match counts. It is double size so I do
 * not need to check the array bounds in the inner
 * match count increment loop.
 */

dbString = database - 1;
aTable = theStore->alphaTable;
aList = theStore->alphaList;
threshold = fragSize-diffsAllowed;

numMatch = 0;
count = -fragSize;   // count = current database loc.

while( (dbChar = (long)(*(++dbString))) > 0x00 )
{
  // circular queue reset to center
  if( matchCur == matchTop )
  {
    matchCur += fragSize;
  }

  // check for a match to the fragment
  if((*matchCur+*(matchCur+fragSize)) >= threshold )
  {
    matchPosition[ numMatch ] = count;
    numMatch++;
  }

  // clear both old counts
  *matchCur = 0;
  *(matchCur+fragSize) = 0;

  // increment match counts for all possible
  //  fragment alignments
  hitOffset = aList + *(aTable + dbChar) - 1;
  while( (hitPos = (long)(*(++hitOffset))) >= 0 )
  {
```

```
      *(matchCur + hitPos ) += 1;
    }

    // The match count queue is walked in reverse
    //  order through memory because the alphaList
    //  indexes are positive.
    matchCur--;
    count++;
  }

  /*********
   * Check the remaining match count entries for
   * any fragments that extend beyond the end of
   * the database.
   */

  for( loop=0; loop < diffsAllowed; loop++)
  {
    if( matchCur == matchTop )
    {
      matchCur += fragSize;
    }

    if((*matchCur+*(matchCur+fragSize)) >= threshold )
    {
      matchPosition[ numMatch ] = count;
      numMatch++;
    }
    matchCur--;
    count++;
  }

  return(numMatch);
}

// end of DNA_Match.c
```