```
900 C=7280
2300 X=X+C
```

took 2.75 seconds to do the same number of repetitions.

3. If you must use the GOTO statement, jump forward in your program rather than back. If you must jump back, however, jump to the start of the program rather than back a few lines.

The same is true for GOSUB. On meeting a GOTO or GOSUB instruction the BASIC interpreter compares the target line number with the current line number. If the target is greater than the current, the interpreter simply searches forward, line by line, until it is found. But if the target is less than the current, then the search always begins from the very first line of the program. This means that it may be more efficient to place subroutines and frequently used sections at either end of a program. Add 56 REM lines at the start of the program, to make it up to typical length, and try:

```
2300 GOTO 2400
2400 GOTO 2500
2500 GOTO 2900
```

This took 2.33 seconds for 500 repetitions, whereas:

```
2300 GOTO 2500
2400 GOTO 2900
2500 GOTO 2400
```

took 4.85 seconds.

4. Initialise all variables in order of access frequency.

Variable names are stored by the interpreter in a symbol table in the order in which they first appear in a program. The later a variable occurs in the table, the longer it takes to find it and access its contents. For the same reason you should avoid using a new variable in a program where you can resort to one previously used by the program but currently not in use.

If a variable is used inside nested loops — as is common in sorting — that variable is accessed frequently, so initialise it at the start of the program before any other variable, with a dummy value if need be:

```
1000 L=500:C=7280:X=0:Z=1.1
2300 A=0
```

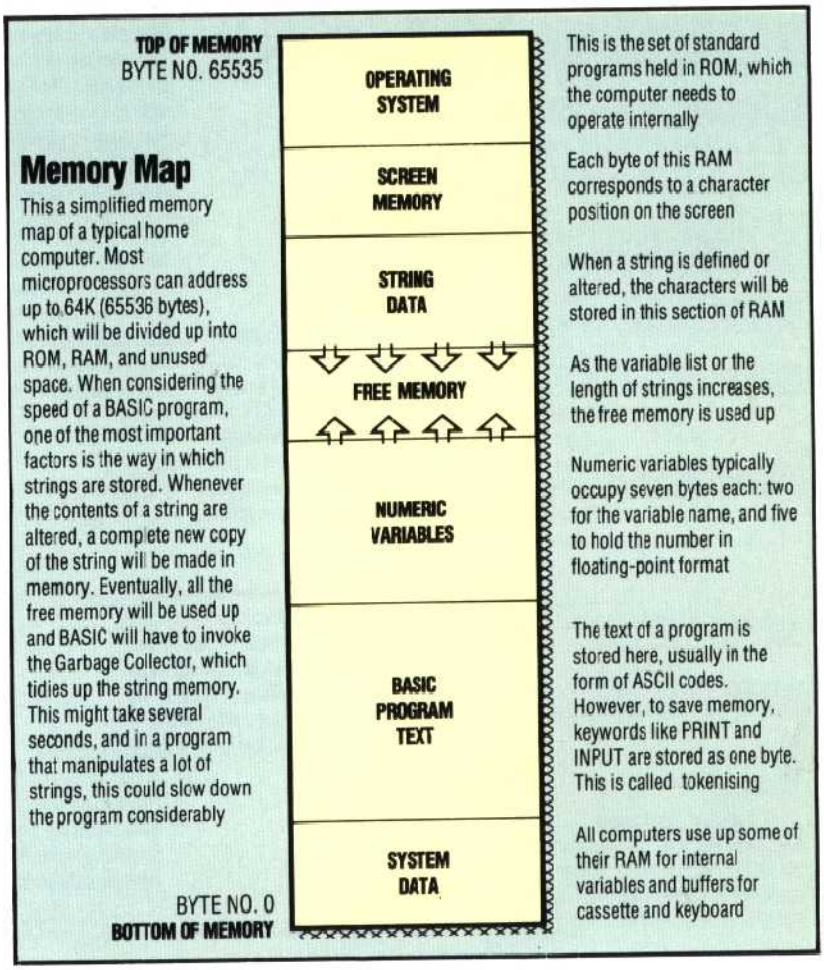
took 2.2 seconds for 500 repetitions, whereas:

```
1000 A=0:L=500:C=7280:X=0:Z=1.1
2300 A=0
```

took 2.06 seconds.

5. Avoid using strings.

String operations use up memory in ways that arithmetic does not, and a system program called the Garbage Collector may have to be called every now and again by the interpreter to tidy up the contents of string memory. This procedure can take a lot of time.



A general demonstration of this is difficult to write because computers vary so much in their memory management: you have to fill up most of the user memory with data — a large numeric array will do — then perform string manipulations that will cause the Garbage Collector to be called. On our machine we entered:

```
40 POKE 52,32:POKE 56,32:CLR
```

to reduce severely the amount of memory available to BASIC programs, and then entered:

```
1000 L=500:DIMT$(L)
1100 FOR K=1 TO L
1200 T$(K)="A"+"B"
1300 PRINT K
1400 NEXT K
```

which uses up a lot of string memory and provides a string array for later use. The PRINT statement is executed in every iteration, displaying the value of the loop counter. When we ran this version of the testbed program, the printing repeatedly paused as the Garbage Collector was called to rearrange memory. Sometimes the pause lasted more than three seconds. The program continues:

```
2300 A$=LEFT$(T$(L),1):B$=A$+RIGHT$(T$(L),1)
```

and this took 30.03 seconds for 500 repetitions. When we ran the same program with much more memory available, garbage collection was not visible, and the timed loop took 8.66 seconds.