Manuscript, dated 5th January 1999.

CSM03 & CS210: Complexity Theory

Survival Kit for Log Users

Horst Holstein [1]

Version: 8 January 1999

[1]Department of Computer Science, University of Wales, Aberystwyth, Ceredigion, SY23 3DB, UK.

1 Abstract

The mathematical logarithm function occurs frequently in a discussion of algorithmic complexity. A working knowledge of this function is therefore required by computer scientists concerned with algorithm efficiency issues.

Such a definition of logarithms is given, which is sufficient to allow (a) numerical estimates of the function to be made, and (b) some standard theoretical results to be obtained, or illustrated.

The definition is based on counting the number of divisions required to reduce a given number to a certain target. The concept of repetition and loop counting should be familiar to computer science students. The result is accurate to within a whole number.

The definition is applied to some case studies, related to tree and binary searches, and sorting.

2 Introduction

The class of functions known as "logarithms" (or "logs" for short) exhibit *slow growth* with respect to the function argument.

Many algorithms have a complexity governed (possibly in part) by the log function. This is the reason for studying the log function in this course.

3 What is slow growth?

"Slow growth" is exhibited by a function which increases with its argument, but by less and less as the argument gets bigger.

An example is that of a tree, which gets taller every year, but less and less so as the years go by. The graph of the tree growth against time will 'tail off'.

Table 1. Slowth grow data for tree example

Age in years	2	3	4	5	6	7	8
Growth (ft)		3.0	2.1	1.4	0.9	0.7	0.3
Height (ft)	5.0	8.0	10.1	11.5	12.4	13.1	13.4

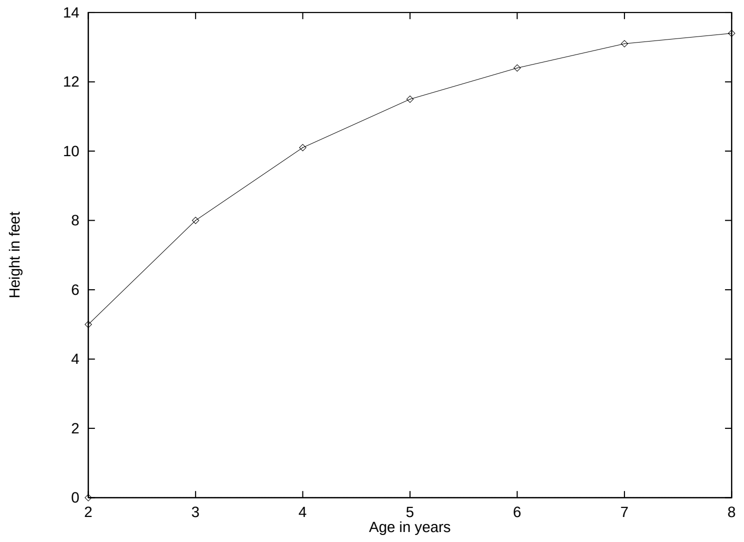

Figure 1: Growth of tree against time.

4 Informal definition of a logarithmic function

A logarithmic function has *two* parameters, the *base* and the *argument*. The function is usually written as

$$\log_b(x) \quad \text{or} \quad \log_b x \tag{1}$$

where b is the base and x is the argument. The usual restrictions are

$$\begin{aligned} b \ \ \text{(base)} &> 1 \\ x \ \ \text{(argument)} &> 0 \,. \end{aligned} \tag{2}$$

We are usually concerned with the value of a logarithm $\log_b N$ for values of N much greater that one. In that case, a useful definition which allows estimation of the value of $\log_b N$ for a given base b and argument N is

Informal Definition: The value of $\log_b N$ is the number of times N can be divided by b to reach unity.

Examples

$\log_{10} 1000$

1000	100	10	1
No of divisions by 10	1	2	3

Result: $\log_{10} 1000 = 3$ (exact).

$\log_8 1000$

1000	125	15.6	1.95	0.244
No of divisions by 8	1	2	3	4

Result: $3 < \log_8 1000 < 4$.

$\log_4 1000$

1000	250	62.5	15.6	3.90	0.976
No of divisions by 4	1	2	3	4	5

Result: $4 < \log_4 1000 < 5$.

$\log_3 1000$

1000	333	111	37.0	12.3	4.11	1.37	0.46
No of divisions by 3	1	2	3	4	5	6	7

Result: $6 < \log_3 1000 < 7$.

$\log_2 1000$

1000	500	250	125	62.5	31.3	15.6	7.81	3.90	1.95	0.977
No of divisions by 2	1	2	3	4	5	6	7	8	9	10

Result: $9 < \log_2 1000 < 10$.

When unity is reached exactly, the result obtained from this definition is the correct value, *e.g.*

$$\log_{10} 1000 = 3 \ . \tag{3}$$

When unity is not reached exactly, we can only give the integer interval in which the exact value of the logarithm lies. It is useful to use the 'floor' function $\lfloor . \rfloor$ and ceiling $\lceil . \rceil$ functions in that case, which respectively round down and up. Thus

$$\begin{aligned} \lfloor \log_2 1000 \rfloor &= 9 \\ \lceil \log_2 1000 \rceil &= 10 \end{aligned} \tag{4}$$

A more precise definition (not given here) is needed if fractional values in the integer interval are needed (but see the examples below).

4.1 Estimates of the log function

Some sample values of the log function are tabulated below, derived by repeated division as shown above.

A fractional base $e = 2.718\ldots$ is included.

Table 2. Estimates of the log function in various bases b, using repeated division.

b	$\log_b 1,000$	$\log_b 10,000$	$\log_b 100,000$	$\log_b 1,000,000$
10	3	4	5	6
8	3-4	4-5	5-6	6-7
4	4-5	6-7	8-9	9-10
3	6-7	8-9	10-11	12-13
e	6-7	9-10	11-12	13-14
2	9-10	13-14	16-17	19-20

The slow growth of the log functions with respect to their arguments are evident. This is further shown in Figure 2.

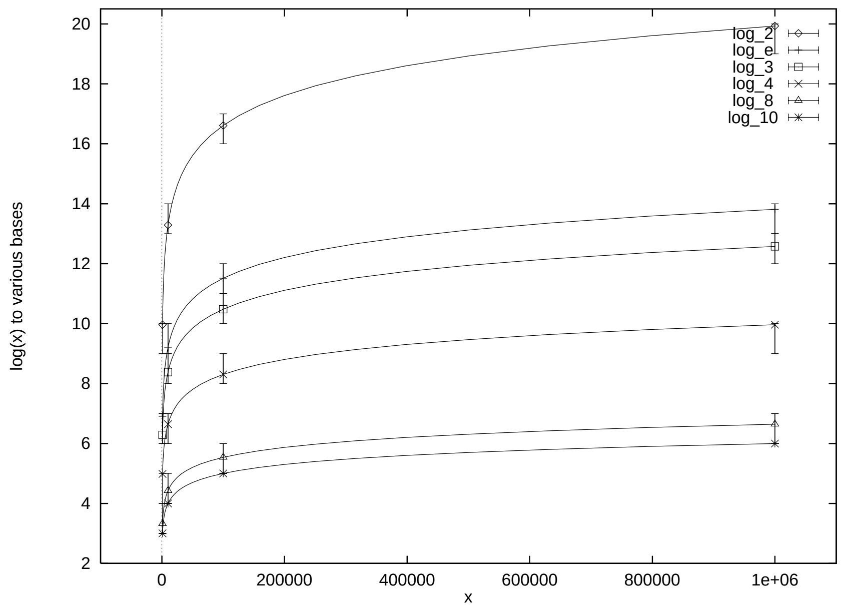

Figure 2: Graph of log functions. Vertical bars indicate estimates in Table 2

5 Properties of the log function

If we estimate the value of $\log_4 N$ by counting the repeated divisions of N by 4 until a result of order unity is reached, we would clearly reach the same target, but with twice the number of divisions, when using repeated division by 2, since one division by 4 is equivalent to two consecutive divisions by 2.

Similarly, in the estimation of $\log_8 N$ by repeated division of N by 8, the same target would be reached using repeated divisions by 2, but with three times the number of divisions, since division by 8 is equivalent to 3 consecutive divisions by 2.

Relating these facts back to the informal definition of logarithms given above, and realising that a count of the divisions of N by 2 gives an estimate of $\log_2 N$, we obtain

$$\frac{\log_2 N}{\log_4 N} = 2 \quad \text{for all } N, \tag{5}$$

$$\frac{\log_2 N}{\log_8 N} = 3 \quad \text{for all } N. \tag{6}$$

These relations are *exact*, although we have only demonstrated their plausibility with our informal definition of logs. In general, for any pair of bases a and b,

$$\frac{\log_b N}{\log_a N} \text{ is constant for all } N \tag{7}$$

where the constant depends only on the bases a and b. Thus, log functions of different bases are *proportional*. If we know one log function in one base, then the log function in any other base can be obtained by scaling.

What is the scaling factor? The examples above with bases 2 and 8 show that the factor 3 in equation (6) arose from the fact that 8 can be divided by 2 three times to reach unity, that is, the factor is $\log_2 8$ by our informal definition. This result also holds in general. Thus,

$$\frac{\log_b N}{\log_a N} = \log_b a. \tag{8}$$

This result essentially states that one division by a is equivalent to $\log_b a$ divisions by b. The same scaling factor is obtainable from equation (7) with N set to a, and the result $\log_a a = 1$ (see equation (12) below).

In the practical problem of converting the logarithm of one base into the logarithm of another, we re-express equation (8) into the form which groups terms of the same base on the same side of the equals sign, that is,

$$\frac{\log_b N}{\log_b a} = \log_a N. \tag{9}$$

Then, knowing the log function in the base b, we can evaluate the left hand side and obtain the log function for any N in the base a.

There is another way of looking at equation (9). The left hand side is a ratio of logarithms in the same base b, and the right hand side is independent of b. Therefore, *a ratio of logarithms* in the same base *is independent of the base.* Thus for bases $b, c, d, \ldots$,

$$\frac{\log_b N}{\log_b a} = \frac{\log_c N}{\log_c a} = \frac{\log_d N}{\log_d a} = \ldots . \tag{10}$$

This means that expressions such as $\log N / \log a$ are meaningful even when the base is not specified, because the result is the same whatever base is chosen.

5.1 Logarithms of powers

It takes exactly *zero* divisions to divide 1 by b to reach unity, exactly *one* division of b by b to reach unity, and exactly n divisions by b to reduce b^n to unity. The properties

$$\begin{aligned}\log_b 1 &= 0 \quad (\text{any base } b), &(11)\\ \log_b b &= 1, &(12)\\ \log_b(\underbrace{b * b * \ldots * b}_{n \text{ terms}}) &= n \ , \quad \text{or} \ \log_b(b^n) = n &(13)\end{aligned}$$

therefore follow directly from the informal definition. More generally, $b^n N$ takes n more divisions to reduce to unity than does N, so

$$\begin{aligned}\log_b(bN) &= 1 + \log_b N &(14)\\ \log_b(b^n N) &= n + \log_b N. &(15)\end{aligned}$$

When a base a is used in place of the base b, it takes $\log_a b$ divisions of b by a to reduce b to unity. Therefore, the number of divisions required to reduce N to unity under repeated division by a is $\log_a b$ times the number required to reduce N by unity under repeated division by b. Thus the generalisation of equation (13) is

$$\log_a b^n = n \log_a b. \qquad (16)$$

The same result is obtained more formally from the base independence formula (10),

$$\frac{\log_a(b^n)}{\log_a b} = \frac{\log_b(b^n)}{\log_b b} = n, \qquad (17)$$

which, upon multiplication by $\log_a b$, leads back to formula (16).

5.2 Natural logarithms

The properties (11-13) allow us to sketch the log graph for small arguments. This is done in Figure 3, in which the continuous lines are computed from exact functions. The log functions for the different bases all cross at the point (1,0), in accordance with result (11). The base of the log_e function is chosen such that the graph at the crossing point (1,0) has a slope of 45 degrees. This uniquely defines the base - it is seen to lie between between 2 and 3, being rather closer to 3 than to 2.

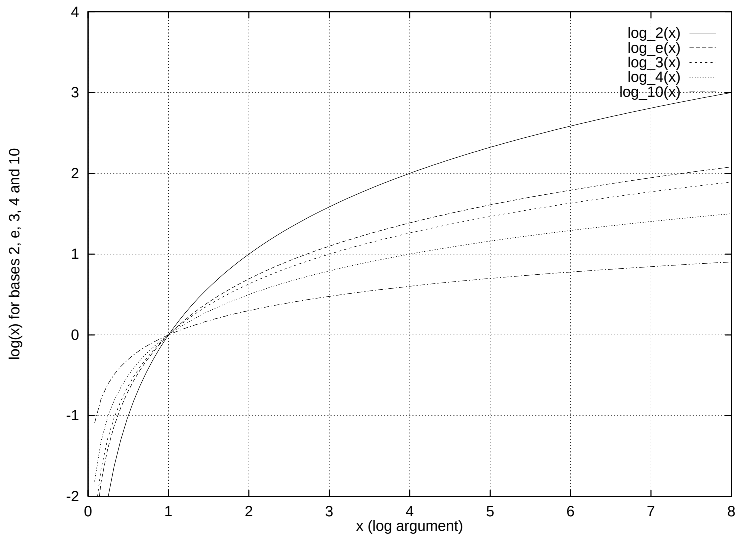

Figure 3: Graph of log functions, for small arguments

The log function with the 45 degree property is called the *natural logarithm* function. The base is denoted by the symbol e, where $e = 2.718\ldots$. Alternative notations are

$$\ln N = \log_e N = \log_{2.718\ldots} N. \tag{18}$$

For our purposes, the natural log function is simply a log function with a base close to 3. The number e arises in many mathematical contexts, rather like the number π.

Many computer languages will provide the ln function, but may not provide logarithms to any other base. In that case, formula (9) is used to compute the logarithms to any required base a, in the form

$$\log_a N = \frac{\ln N}{\ln a} \quad . \tag{19}$$

6 Logarithms in the hierarchy of growth functions

There are a number of standard functions which appear in complexity theory. These can be ordered by decreasing growth in the sense that a function higher up in the list will always overtake a function lower in the list at sufficiently high values of the argument, even if there is an unfavourable constant in front of it. Thus, $0.00001N^2$ will overtake the function N at sufficiently high values of N, namely when $N > 1/0.00001$, to be precise. In this sense, N^2 has a higher growth rate than N.

A table of standard functions ordered by decreasing growth is as follows.

Table 3. A hierarchy of growth function, in descending order

Function type	Function	Comments
Combinatorial	$N! = 1 * 2 \ldots * N$	The "try all possibilities" problem
Exponential	$\ldots, 4^N, 3^N, 2^N, \ldots$	Growth by a constant as N increases by 1
Polynomial	$\ldots, N^4, N^3, N^2, N$	
Fractional poly.	$\ldots N^{.9}, N^{.8}, \ldots$	
Logarithmic	$\ldots, \log_2 N, \ln N, \log_3 N, \ldots, \log_{10} N, \ldots$	Slow growths.
Log-log	$\ldots, \ln(\ln N), \ldots$	Very slow growth.

Mixed terms can appear in a complexity formula. Thus $N^{2.5}, N^2, N^{1.5}, N * \log_2 N, N$ form a decreasing sequence.

The different types are plotted in Figure 4. The vertical scale indicates powers of 10, so as to be able to accommodate a very large range.

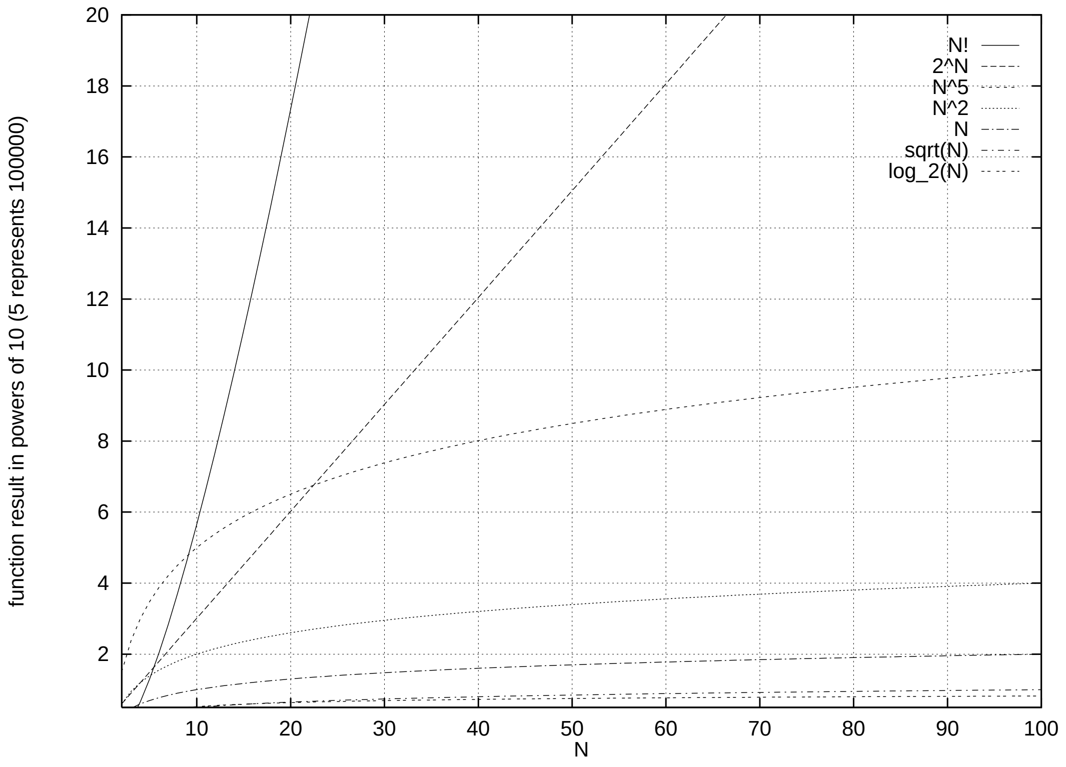

Figure 4: The tyranny of growth.

7 Some applications of log functions in complexity

7.1 How many page searches does it take to find a word in a dictionary?

There are many strategies for doing this. The binary search is a divide and conquer strategy which keeps halving the size of the problem.

Suppose a dictionary has N pages. How many times does the dictionary have to be opened to guarantee to locate a given word, or prove its absence from the dictionary?

If we use the binary search method, then with each lookup we can halve the number of pages in which the word is to be found. The number of times N can be halved before reaching order unity is $\log_2 N$.

The precise result is that it need take at most $\lceil \log_2 N \rceil$ page openings.

7.2 How many digits does a (large) number have?

The number of digits in the number N depends on the base used. In the decimal base, each division by 10 of N (throwing away any remainder) shortens the result by one digit. Repeating this until we get order unity, the result is seen to be $\log_{10} N$.

The precise result is $\lfloor \log_{10} N \rfloor + 1$ (try it!) .

If the base is some other value, *e.g.* 2, then the same reasoning gives $\lfloor \log_2 N \rfloor + 1$.

7.3 What is the height of a (complete) binary tree?

A complete binary tree has all its levels filled, except possibly the last.

If there are N nodes in total, then these can be thought of as as being shared out in the different levels according to the sum

$$N\left(\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\dots\right) \tag{20}$$

since each level has half the previous number of nodes. The total of the sum in brackets is just short of 1, no matter how many terms are used. Thus the sum accounts for a good estimate of the number of nodes N. The number of times N can be halved before reaching order unity is $\log_2 N$. This is the estimate of the height of the tree.

Again, an exact integer result can be given, but our estimate will never be wrong by more than one level.

Many algorithms, such as a binary tree search, move down one level of a tree from the root until they come to a leaf node. If the tree has N nodes, only $\log_2 N$ nodes will be encountered before a leaf node is reached.

7.4 Searching a B-tree

In a B-tree, the levels are complete, but each node can have between k and $2k-1$ children, where k is the order of the tree.

If $k = 10$ and there are N leaf nodes, height of the tree lies (approximately) between $\log_{10} N$ and $\log_{19} N$. So, searching for a node in a B-tree of $N = 100000$ nodes requires between 4 and 6 levels to be accessed, with constant work at each level.

7.5 Complexity of the quicksort algorithm

The Quicksort algorithm shuffles elements relative to a chosen pivot element. The pivot stays in its correct position thereafter.

Schematically, each stage is illustrated by

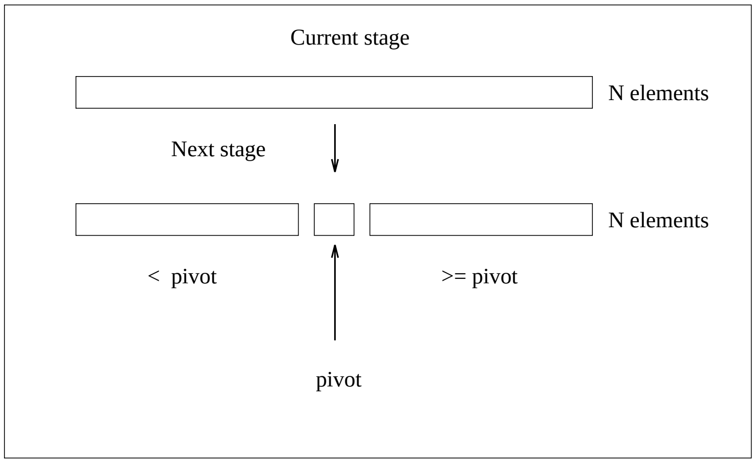

Figure 5: Partition process in the quicksort algorithm

The process is repeated on each partition. The amount of work needed to shuffle elements either side of the pivot, for N elements, is $O(N)$ steps, since each element has to be moved. At later stages, there will be many pivotal elements, but the remaining elements must all be shuffled either side of the local pivotal element. Thus no matter what level of partitioning we consider, the amount of work in shuffling elements is $O(N)$ steps at each partition level.

This estimate is poor (being an overestimate) only in the final stages, when partitions are very small and a substantial number of elements have become pivots, which are not themselves shuffled.

How many levels of partitions are needed? In an optimum partitioning strategy, we would have balanced partitioning - each level produces partitions of half the size in the previous level. The number of times N elements of the original unsorted partition can be halved in this way is $O(\log_2 N)$. Since the total work done at each partition level is $O(N)$, the total overall work is then $O(N \log_2 N)$.

Unfortunately, the optimum partioning cannot be carried out while maintaining $O(N)$ at each partition level. Instead, a heuristically 'sensible' partition strategy is adopted (such as between the median value of the values at the ends and in the middle).

One can show that the average of performance of quicksort under these conditions is still $O(N \log_2 N)$. The average is taken over all permutations of the data. It is, however, possible to find permutations for which the quicksort algorithm degenerates to $O(N^2)$. The probability of randomly choosing such an ordering, however, is very very small.

Given the $O(N \log_2 N)$ complexity of the quicksort average performance, how does the time grow when the data data size is doubled from N items to $2N$ items?

The ratio

$$\frac{2N \log_2 2N}{N \log_2 N} = 2\frac{1 + \log_2 N}{\log_2 N} = 2\left(1 + \frac{1}{\log_2 N}\right) \tag{21}$$

is only slightly greater than 2, when N is large. Thus, doubling the data size incurs slightly more than twice the amount of work, on average.

The details of the quicksort algorithm are quite involved. However, common to all sort algorithms, data must be moved around and must be compared. An experiment to sort random data for various data sizes N is reported below. Counters were put into the algorithms to monitor (a) moves - data assigned to a temporary variable (b) swaps - pairs of data values interchanged (c) comparison - pairs of data values compared. The values of the counters for these operations were recorded.

The theoretical growth functions for these quantities were estimated beforehand, and by dividing the counters by these functions it was possible to see whether the ratios 'settle down', indicating that the right growth function had been chosen.

```
    N  Moves  Swaps   Comps | Moves/N   Swaps/NlogN   Comps/NlogN

-----  -----  -----   ----- | -------   -----------   -----------

   15     45      4      45 | 3.00000   6.82555E-02   7.67874E-01

   31    107     19     138 | 3.45161   1.23714E-01   8.98554E-01

   63    208     62     371 | 3.30159   1.64645E-01   9.85212E-01

  127    389    171     817 | 3.06299   1.92662E-01   9.20498E-01
  255    799    390     2015 | 3.13333   1.91312E-01   9.88443E-01

  511   1642    902     5151 | 3.21331   1.96191E-01   1.12038E+00

 1023   3389   2049    10861 | 3.31281   2.00321E-01   1.06183E+00

 2047   6756   4594    23821 | 3.30044   2.04037E-01   1.05798E+00

 4095  13211  10275    52711 | 3.22613   2.09103E-01   1.07270E+00

 8191  26981  22322   114479 | 3.29398   2.09633E-01   1.07511E+00

16383  53327  48948   250685 | 3.25502   2.13411E-01   1.09297E+00

32767 107248 104863   543186 | 3.27305   2.13351E-01   1.10515E+00

65535 214308 226787  1150267 | 3.27013   2.16285E-01   1.09700E+00
```

It is seen that the last three columns do indeed vary by very little, showing that the correct complexity components were used in the different parts of the algorithm. The number of Moves grows as $O(N)$, while the number of Swaps and Comparisons grow as $O(N \log_2 N)$. Ultimately, the Comparisons and Swaps dominate the algorithm at large N.

7.6 The mergesort algorithm

The mergesort algorithm divides a list of N items into two sublists of approximately equal size, sorts these (recursively), and then merges the two halves together.

The important feature is that to merge two sorted lists each of length $N/2$, $O(N)$ steps are required. Basically, each element has to be (possibly compared) and then moved.

The lists of size $N/2$ are themselves sorted by merging 2 lists of size $N/4$. At this level there will be 4 such lists. Merging the two pairs still requires $O(N)$ work to produce at this stage two sorted lists of length $N/2$.

At each stage of the recursion, therefore, $O(N)$ work has to be done in merging the lists.

The total number of stages is $\log_2 N$. The total amount of work therefore grows as $O(N \log_2 N)$.

One may inquire where the sorting is actually carried out, since the entire algorithm is dominated by merging sorted sublists. The only sorting actually done is on sublists of 1 element. Even here the process of sorting is actually one of merging two lists of one element!

The mergesort algorithm can always be made to produce balanced partitions, and so is optimal. It relies on having dynamic lists, however (as is natural when the data is large and kept on files).

When the algorithm is adapted for array sorting, copy arrays are required. This introduces both storage and time overheads, which then make the algorithm less efficient, on average, than the quicksort algorithm.

7.7 How quickly can sorting be carried out?

It is possible to consider this problem abstractly, under the conditions that the only criterion on which to sort is that of key comparisons, and that no other information concerning the keys (such as the their range) is available.

This assumption is important, because there are various 'bin-sorting' type algorithms, in which an item can be placed in its final position just by looking at the key (consider sorting a complete deck of playing cards, for example). These algorithms use information about the keys, and allow of sorting N items in $O(N)$ steps.

Given N items to sort, using key comparisons only, there are $N! = 1 * 2 * \ldots * N$ possible configurations with which the sorting algorithm may be presented. The sorting algorithm must effectively decide which one of the $N!$ configurations it is presented. Having made this decision, it could use an $O(N)$ bin sort algorithm to allocate the items in their correct order.

How can the algorithm decide between the $N!$ combinations? We do not actually have to specify the process, but can state that for each key comparison made, we halve the search space. The number of key comparisons required is therefore $\log_2 N!$.

While this is the required answer, it is not in a very useful form. How quickly does $\log_2 N!$ grow? The following table shows that while $\log_2 N!$ has an enormous growth rate, it is matched almost exactly by the simpler function $N \log_2(N/e)$, since the ratio of it with $\log_2 N!$ approaches 1 for large N.

Table 4. Comparison of $\log_2 N!$ with Stirling's formula

N	$\lceil \log_2 N! \rceil$	$(\log_2 N!)/(N \log_2(N/e))$
10	22	1.159572043
100	525	1.008938155
1000	8530	1.000740197
10000	118459	1.000067282
100000	1516705	1.000006349
1000000	18488885	1.000000610

Table 4 shows that $N \log_2(N/e)$ - Stirling's formula - is a very good approximation for $\log_2 N!$. Since e is greater than 2, $N \log_2(N/2)$ is an overestimate of this quantity. The latter expression equals $N \log_2 N - N$, which is dominated by $N \log_2 N$. The base 2 only introduces a constant factor when replaced by any other base, so it is permitted to state the final result that $O(\log_2 N!) = O(N \log N)$. Since any superimposed binsort only introduces $O(N)$ work, the complexity of sorting based on key comparisons can be stated as $O(N \log N)$.

We have shown that within the restrictions of using key comparisons only, sorting algorithms faster than $O(N \log N)$ *do not exist.* The constant in front of the complexity function will vary for different sort algorithms of the $O(N \log_2 N)$ type. The quicksort algorithm has the smallest constant of all known array $O(N \log N)$ sorting algorithms.

8 Revision

8.1 The "Big O" notation.

A function $f(N)$ is said to be of order $g(N)$, *i.e.* $O(g(N)$, if $f(N) \leq cg(N)$ for some positive constant c, when N is large enough. In other words, the function f does not grow faster than the function g at sufficiently large arguments N.

8.2 Example

A program takes 10 seconds to process 100 data items. How long does it take to process (a) 1000 (b) 1000000 data items, if the time is proportional to $\log N$, N, $n \log N$, N^2, N^3?

Answer Let $T(N)$ be the time taken to perform a problem with N data items. Let the constant of proportionality be c.

$T(N) = c \log N$.

Case(a): $T(1000)/T(100) = \log 1000/\log 100.$

So $T(1000) = T(100) * 3/2 = 10 * 3/2 = 15$sec.

Case(b): $T(1000000)/T(100) = \log 1000000/\log 100.$

So $T(1000000) = T(100) * 6/2 = 10 * 6/2 = 30$sec.

$T(N) = cN$.

Case(a): $T(1000)/T(100) = 1000/100.$

So $T(1000) = T(100) * 1000/100 = 10 * 10 = 100\text{sec}$.

Case(b): $T(1000000)/T(100) = 1000000/100$.

So $T(1000000) = T(100) * 10000 = 100000\text{sec} \sim 1\text{day}$.

$T(N) = cN \log N$.

Case(a): $T(1000)/T(100) = (1000 \log 1000)/(100 \log 100) = 10 * 3/2$.

So $T(1000) = T(100) * 10 * 3/2 = 150\text{sec}$.

Case(b): $T(1000000)/T(100) = (1000000 \log 1000000)/(100 \log 100) = 10000 *$

$6/2$.

So $T(1000000) = T(100) * 30000 = 300000 \sim 3\text{days}$.

$T(N) = cN^2$.

Case(a): $T(1000)/T(100) = (1000^2)/(100^2) = (1000/100)^2 = 100$.

So $T(1000) = T(100) * 100 = 1000\text{sec} \sim 20\text{min}$.

Case(b): $T(1000000)/T(100) = (1000000^2)/(100^2) = 10000^2 = 10^8$.

So $T(1000000) = 10^8 T(100) = 10^9\text{sec} \sim 30\text{years}$.

$T(N) = cN^3$.

Case(a): $T(1000)/T(100) = (1000^3)/(100^3) = (1000/100)^3 = 1000$.

So $T(1000) = T(100) * 10^3 = 10^4\text{sec} \sim 2.5\text{hours}$.

Case(b): $T(1000000)/T(100) = (1000000^3)/(100^3) = 10000^3 10^1 2$.

So $T(1000000) = 10^{12} T(100) = 10^{13}\text{sec} \sim 3 * 10^5\text{years}$.

Note that the constant of proportionality does not feature in the answers, since an actual time for the case $N = 100$ is given.

8.3 Complexity of loops

If a loop has N passes, and the time for each loop pass never exceeds a certain constant value, then the single loop pass is an $O(1)$ process and the whole loop is an $O(N)$ process.

If the loop contains an inner loops of M passes, then the combined loops make MN inner passes, leading to an $O(MN)$ process.

In many situations the number of loop passes in an inner loop may be controlled by an outer loop. For example, pseudocode for the bubble sort may be written

```
for I in A'First .. A'Last-1 loop
  Flag := True;
  for J in reverse I+1 .. A'Last loop
     if A(J) < A(J-1) then  -- comparison
        Swap( A(J), A(J-1) );
        Flag := False;
     end if;
  end loop;
  exit when Flag;
```

```
        end loop;
```

If the outer loop performs N passes, then the inner loop performs $N-1, N-2, N-3, \ldots, 2, 1$ passes. This is a total of

$$1 + 2 + \ldots + (N-2) + (N-1) \tag{22}$$

loop passes. If added up twice, but in mutually reverse orders, we obtain

$$(1+(N-1))+(2+(N-2))+\ldots+((N-2)+2)+((N-1)+1) = (N-1)N \tag{23}$$

giving the correct result $(N-1)N/2$ for the sum (22). The result is $O(N^2)$.

If the maximum number of loops is not always obeyed, as in the above case with an exit in the loop, then there is generally a "best case", and "average case" and a "worst case".

The best case occurs when no swaps take place, so that the Flag variable remains true after the first inner pass. There are then no outer passes, and the algorithm is $O(N)$.

In the average case we may assume a 50% probability that the comparison is true. Therefore the probability of *no* swap taking place in the inner loop is $(1/2)^{k/2}$, where k is the number of loop passes at one one stage. Even for inner loop passes as low as 20, this probability is as low as $(1/2)^{10}$, or about one in one thousand. It can be seen that even for moderate values of N, the presence of the inner exit will have a negligible effect on the average run time. This the average and worst runtimes will be very similar, of $O(N^2)$. This is borne out by experiments.

The following example outputs illustrate these points. Randomly chosen data of different sizes N were sorted by the bubble sort routine (above) with counters monitoring comparisons and swaps. The results were:

```
   N     Swaps     Comps   Swaps/N^2     Comps/N^2

 ---     -----     -----   -----------   -----------

  15        32        90   1.42222E-01   4.00000E-01

  31       180       465   1.87305E-01   4.83871E-01

  63       864      1943   2.17687E-01   4.89544E-01

 127      3806      7998   2.35972E-01   4.95877E-01

 255     15020     32330   2.30988E-01   4.97193E-01

 511     63051    130277   2.41463E-01   4.98914E-01

1023    257493    522123   2.46045E-01   4.98909E-01

2047   1021374   2093730   2.43753E-01   4.99672E-01
```

The counter values for the Swaps are very close to half the counter values for the number of Comparisons, as suggested would be the average case. The swaps and comparisons clearly grow with N. The manner in which they grow is illustrated in the last two columns, in which the counters are divided by the expected growth functions N^2. The columns "settle down" with increasing N, showing that the number of swaps and comparisons do indeed grow as fast as N^2.

The last column shows that the number of comparisons is very close to $N^2/2$, which is the value expected when no early exit from the inner loop is made.

These counter values may be compared with the corresponding values for the quicksort algorithm, given above (section 7.5). The benefits of an $O(N \log N)$ algorithm over an $O(N^2)$ are clearly apparent. Quicksort can cope with 30 times the data size for the bubble sort before requiring in excess of one million comparisons.