$\mathcal{CS}$21120

Complexity of Algorithms

H. Holstein

References:

1. Data Structures in Java. TA Standish. Addison Wesley 1998.

2. Data structures and algorithms. Aho, Hopcroft and Ullman. Addison Wesley, 1983.

1.0 Survival Kit for Log Users

The mathematical logarithm function ($\log(x)$) occurs frequently in estimating algorithmic complexity (performance metric). A working knowledge of this function is required.

Such a definition should allow

- numerical estimates of the function to be made (for large arguments x),
- some standard theoretical results to be obtained, or illustrated.

The working definition given here for

$$\log_b(x)$$

is based on counting the number of times x can be divided by b before reaching a value around unity.

The concept of repetition and loop counting should be familiar. The result is accurate to within a whole number.

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

1.1 Introduction

The class of functions known as “logarithms” (or “logs”) 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.

1.2 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 as the years go by. The graph of the tree growth against time will ‘tail off’.

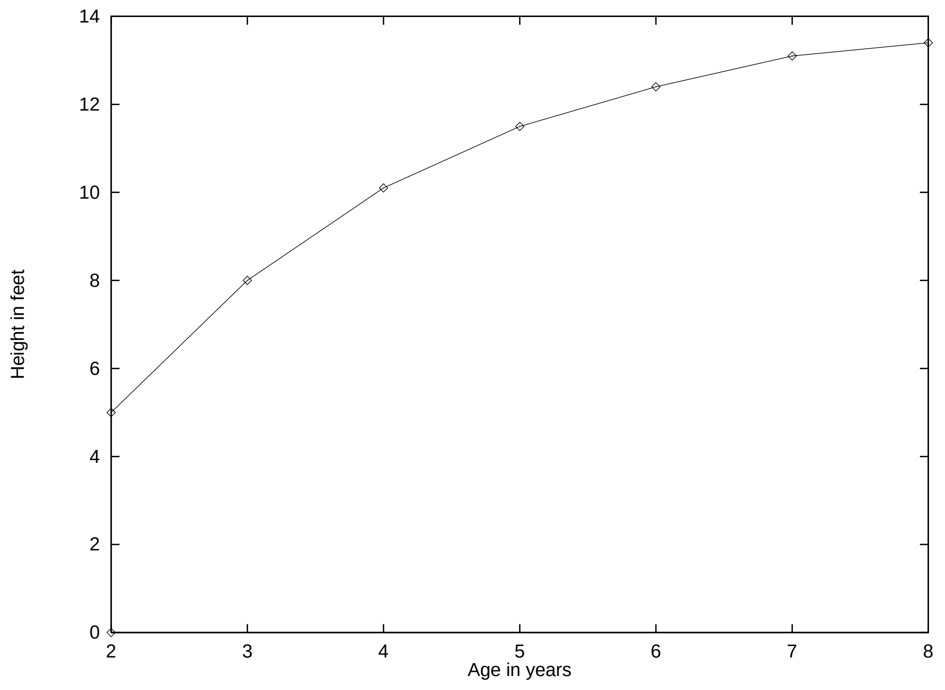

Fig. 1. Growth of tree against time.

Table 1. Slow growth data for tree example

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

2.0 Informal definition

A logarithmic function $\log_b(x)$ has *two* parameters, the *base b* and the *argument x*.

Notation:

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

Usually,

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

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.

2.1 Examples

$\log_{10} 1000$ - divide 1000 repeatedly by 10.

Divisions	1	2	3
Value	100	10	1

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

$\log_{8} 1000$ - divide 1000 repeatedly by 8.

Divisions	1	2	3	4
Value	125	15.6	1.95	0.244

Result: $3 < \log_{8} 1000 < 4$.

$\log_4 1000$ - divide 1000 repeatedly by 4.

Divisions	1	2	3	4	5
Value	250	62.5	15.6	3.90	0.976

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

$\log_3 1000$ divide 1000 repeatedly by 3.

Divisions	1	2	3	4
Value	333	111	37.0	12.3

4.11	1.37	0.46
5	6	7

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

$\log_2 1000$ - divide 1000 repeatedly by 2.

Divisions	1	2	3	4	5
Value	500	250	125	62.5	31.3

15.6	7.81	3.90	1.95	0.977
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.*

$$\log_{10} 1000 = 3\ . \qquad (3)$$

When unity is not reached exactly, this method gives only the integer interval in which the exact value of the logarithm lies.

The ‘floor’ function $\lfloor . \rfloor$ and ceiling $\lceil . \rceil$ functions 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.

2.2 Table of Estimates

Sample values of the log function are tabulated below, derived as above.

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

Table 2. Estimates of the log function.

b	1,000	10,000	100,000	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.

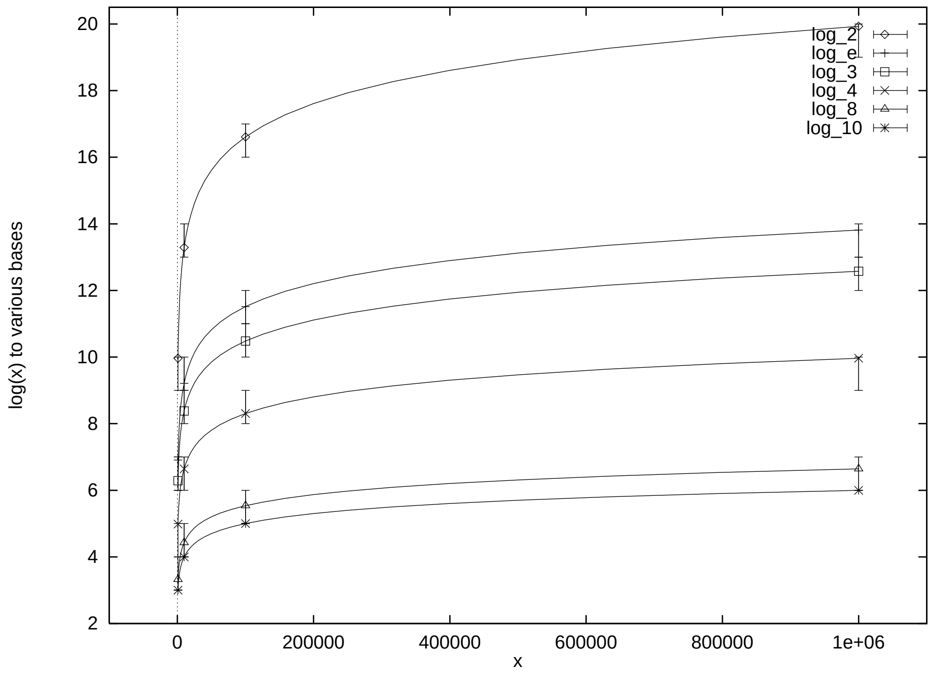

Fig 2. Graph of log functions. Vertical bars indicate estimates in Table 2.

3.0 Log function properties

Dividing repeatedly by 4 is clearly a process that is twice as fast as repeatedly dividing by 2.

Similarly, dividing repeatedly by 8 is clearly a process that is three times as fast as repeatedly dividing 2.

Relating these facts to section 2, it follows that for all N,

$$\frac{\log_2 N}{\log_4 N} = 2, \tag{5}$$

$$\frac{\log_2 N}{\log_8 N} = 3. \tag{6}$$

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,

$$\frac{\log_b N}{\log_a N} \text{ is constant for all } N \qquad (7)$$

where the constant depends only on the bases a and b.

Thus, log functions of different bases are *proportional*.

If we know a log function in one base, then the log function in any other base can be obtained by scaling.

What is the scaling factor?

Division by b is slower than division by a, if $b < a$, by a factor equal to the number of times a can be divided by b to reach unity, that is

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

This result holds in general.

The same scaling factor is obtainable from equation (7) with N set to a, and the identity $\log_a a = 1$ (see equation (12) below).

3.1 Base conversion

To convert 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 $b, c, d, \ldots$,

$$\frac{\log_b N}{\log_b a} = \frac{\log_c N}{\log_c a} = \frac{\log_d N}{\log_d a} = \ldots \; . \qquad (10)$$

This means that expressions such as

$$\log N / \log M$$

are meaningful, because the result is the same whatever base is chosen.

3.2 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,

exactly n divisions by b to reduce b^n to unity.

The properties

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

therefore follow directly from the informal definition.

More generally,
$b^n N$ takes n more divisions to reduce to unity than does N, so

$$\log_b(bN) = 1 + \log_b N \quad (14)$$
$$\log_b(b^n N) = n + \log_b N. \quad (15)$$

Formula (13) is also of interest when a base a is used in place of the base b.

The number of divisions required to reduce b^n to unity under repeated division by a is $\log_a b$ times the number required to reduce b^n by unity under repeated division by b.

Thus, the generalisation of equation (13) is

$$\log_a b^n = n \log_a b. \tag{16}$$

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).

4.0 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.

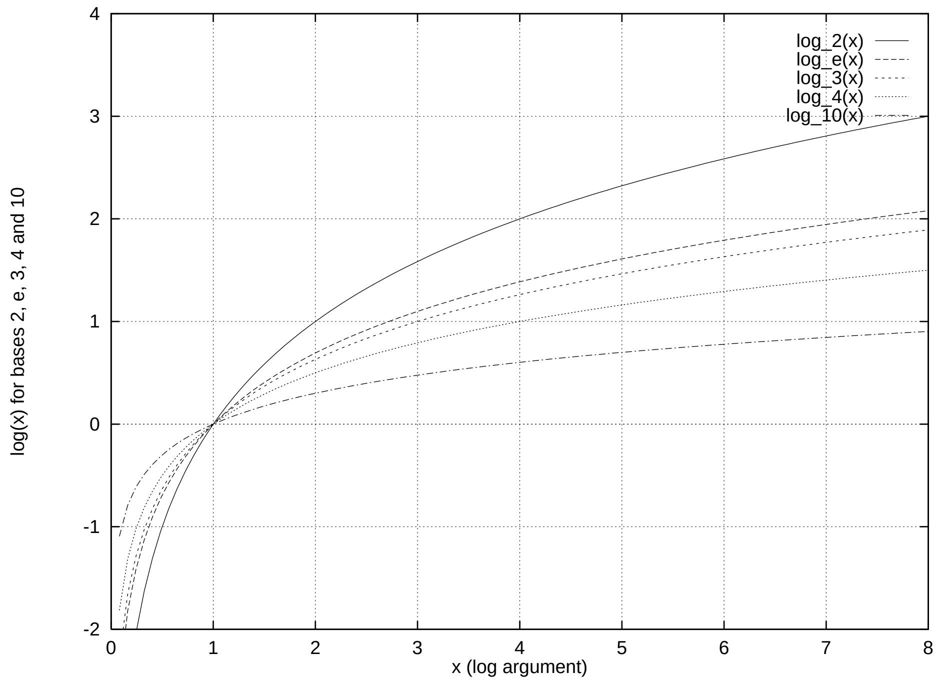

Fig 3. Graph of log functions for arguments near 1.

The log function with the 45 degree property is called the *natural logarithm* function.

The base is denoted by the symbol e, where $e = 2.718\ldots$, denoted by .

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.

Alternative notations are

$$\ln N = \log_e N = \log_{2.718...} N \qquad (18)$$

Some computer languages provide the ln function, but do 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

$$\log_a N = \frac{\ln N}{\ln a} \quad . \qquad (19)$$

$\mathcal{CS}$21120

Complexity of Algorithms

H. Holstein

Contents

5.0 Summary of log properties

Properties expressed in equations (9), (10), (11), (16) should be understood.

Other standard log properties, below, are derivable from the definition page 8.

- $\log x + \log y = \log(x * y)$

- $\log x - \log y = \log(x/y)$

6.0 A 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

A table of standard functions ordered by decreasing growth.

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

Function type	Function	Comments
Combinatorial	$N!$ $= 1 * 2 \ldots * N$	"try all " "possibilities"
Exponential	$\ldots, 4^N$ $3^N, 2^N, \ldots$ $2^N, \ldots$	Growth by a factor as N increases by 1
Polynomial	$\ldots N^4$ N^3, N^2, N	
Fractional polynomial	$\ldots N^{\cdot 9}, \ldots$ $\ldots N^{\cdot 5}, \ldots$	
Logarithmic	$\log_2 N, \ln N,$ $\log_3 N, \log_4 N,$ $\ldots, \log_{10} N, \ldots$	Slow growth
Log-log	$\ldots, \ln(\ln N), \ldots$	Very slow

Mixed terms can appear in a complexity formula. Thus

$$N^{2.5}(N^2 * N^{1/2}), N^2, N^{1.5}, N * \log_2 N, N$$

form a decreasing sequence.

The different growth functions are plotted in Figure 4. The vertical scale indicates powers of 10, to accommodate a very large range.

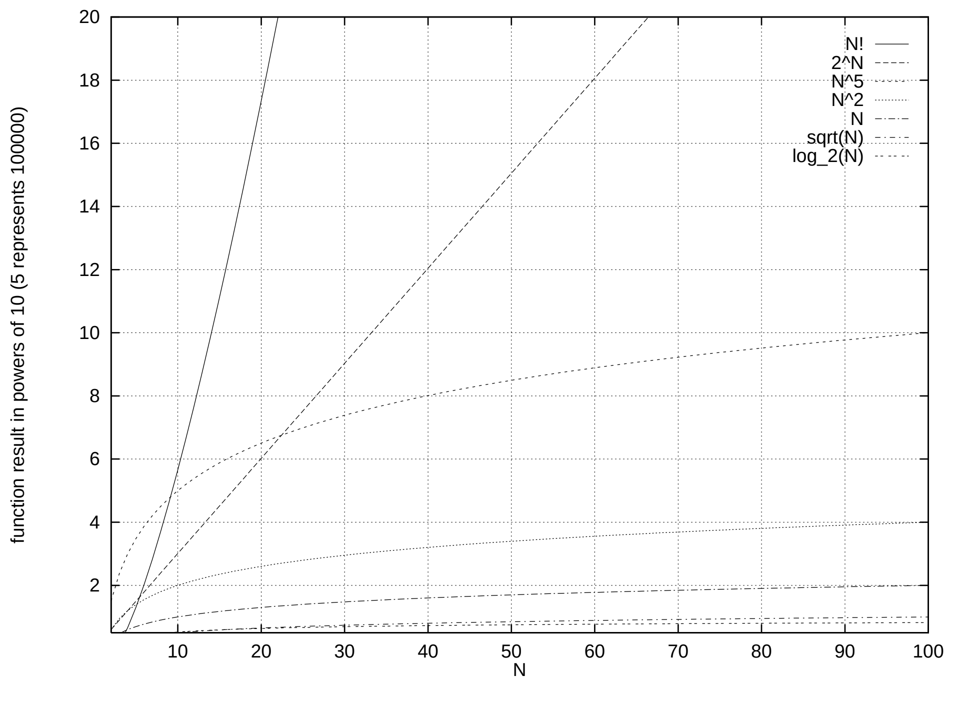

Fig 4. The tyranny of growth.

6.1 Example

Suppose the time $T(N)$ taken by an algorithm, as a function of the data size N, is

$$T(N) = c_0 + c_1 N \ln N + c_2 N^2 \tag{20}$$

where c_0, c_1 and c_2 are machine/environment dependent constants.

Equation (20) can be written as

$$T(N) = N^2 \left(c_2 + c_1 \{(\ln N)/N\} + c_0 \left\{1/N^2\right\}\right)$$

The terms { } each contain a function *divided by* a faster growing function. Therefore, as $N \to \infty$, their contribution $\to 0$, and () $\to c_2$.

The **dominant** term is $N^2 c_2$.

6.2 Example

Suppose, for $N = 100$, a machine executing an algorithm according to equation (20) takes 0.001s. Estimate the time it would take for the case $N = 10^6$.

Solution

The **dominant** term is N^2c_2. Therefore

$$T(10^6)/T(100) \approx c_2(10^6)^2/c_2(10^2)^2 = 10^8.$$

This gives

$$T(10^6) = T(100)10^8\text{s} = 10^5\text{s} = 1\text{day}$$

Note: the constant c_2 is not explicitly used - it is sufficient to know that the time $T(N)$ is *proportional* to N^2.

7.0 The Big-O notation

Notation

$T(N) = O(f(N))$ means that $f(N)$ is the dominant term in $T(N)$,
or "$T(N)$ is of order $f(N)$".

This in turn means that the absolute value of $T(N)/f(N)$ is bounded as $N \to \infty$, that is,

Definition when $\exists c, N_0$, such that

$$\forall N \geq N_0, \quad |T(N)/f(N)| \ \leq \ c \tag{21}$$

then $T(N) = O(f(N))$.

7.1 The Big-O notation: Comments

An almost equivalent definition (which holds even when $T(N)$ and $f(N)$ happen to be zero together - a not very common situation):

Definition when $\exists c, N_0$, such that

$$\forall N \geq N_0, \quad |T(N)| \;\leq\; c|f(N)| \tag{22}$$

then $T(N) = O(f(N))$.

In many situations, $T(N)$ and $f(N)$ are positive when $N \geq N_0$, so the modulus signs can be omitted:

Definition when $\exists c, c > 0, N_0$, such that

$$\forall N \geq N_0, \quad T(N) \;\leq\; cf(N) \tag{23}$$

then $T(N) = O(f(N))$.

7.1 Example

what is the order of $n(n+1)/2$?

Answer: $O(n^2)$, since

$$\begin{aligned} n(n+1)/2 &= n^2(1+1/n)/2 \\ &\leq n^2(1.1/2) \\ &\qquad \text{whenever } n \geq 10 \\ &= O(n^2) \end{aligned}$$

In this example, $c = 0.55$ and $n_0 = 10$.

7.2 Example

Suppose, for $N = 100$, a machine executing an $O(N^2)$ algorithm takes 0.001s. Estimate the time it would take for the case $N = 10^6$.

Solution

This is the same problem as in example **6.2**.

Although $T(N) = O(N^2)$ only tells us, for $N \geq N_0$ and for some constant c, that $T(N) \leq cN^2$, we shall assume for the purpose of estimation that $T(N) = cN^2$ holds with sufficient accuracy, for values of N greater or equal to 100 ($=N_0$).

Therefore, as before,

$$T(10^6)/T(100) \approx c(10^6)^2/c(10^2)^2 = 10^8.$$

This gives

$$T(10^6) = T(100)10^8\text{s} = 10^6\text{s} = 1\text{day}$$

Note: the constant c is not explicitly used.

It is sufficient to know that the time $T(N)$ is *approximately proportional* to N^2.

8.0 Some examples of log functions in complexity

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

Suppose a dictionary has N pages. How many double pages have to be looked at to locate or prove the absence of a given word 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 $\log_2 N$.

It should therefore be possible to construct an $O(\log N)$ algorithm to carry out the search.

8.2 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 $\lfloor \log_2 N \rfloor + 1$.

Exercise

Show that $\lfloor \log_2 N \rfloor + 1 = O(\log N)$

The result is intuitively obvious, but any proof must follow from the definition of Big-O.

$$\lfloor \log_2 N \rfloor + 1 \le \log_2 N + 1, \ \forall N, N > 0.$$

Further,
$\log_2 N + 1 = \log_2 N(1 + 1/\log_2 N) \le \log_2 N(1.1)$ whenever $N \ge 1024$. Combining:

$\lfloor \log_2 N \rfloor + 1 \le \log_2 N 1.1 = (1.1/\log_b 2) \log_b N$

where b is an arbitrary base ($b > 1$).

Thus both $N_0 = 1024$ and $c = (1.1/\log_b 2)$ can be chosen, to establish that

$$\lfloor \log_2 N \rfloor + 1 = O(\log_b N)$$

for any base b. The result follows.

8.3 What is the height of a (complete) binary tree of N nodes?

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

Let N be the number of nodes in the tree, counting only **one** node in the final level.
For trees of height 0,1,2,3,4, ..., the node count N is 1,2,4,8,16 ....

The height of such a tree is precisely the number of times its node count N, can be divided by 2 until reaching unity, that is, $\log_2 N$.

If we take N to be the actual node count, then we must round down any fractional log parts, since the height does not increase as the final level is populated from 1 to completion.

Thus, the height $= \lfloor \log_2 N \rfloor$.

8.4 Searching a B-tree

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

If $k = 19$, with $\lceil k/2 \rceil = 10$, 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
$N = 1000000$ nodes requires between
$\log_{10} 1000000 = 6$ and
$\lceil \log_{19} 1000000 \rceil = \lceil 6/\log_{10} 19 \rceil = \lceil 4.69 \rceil = 5$
levels to be accessed, with constant work at each level.

8.5 Complexity of quicksort

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

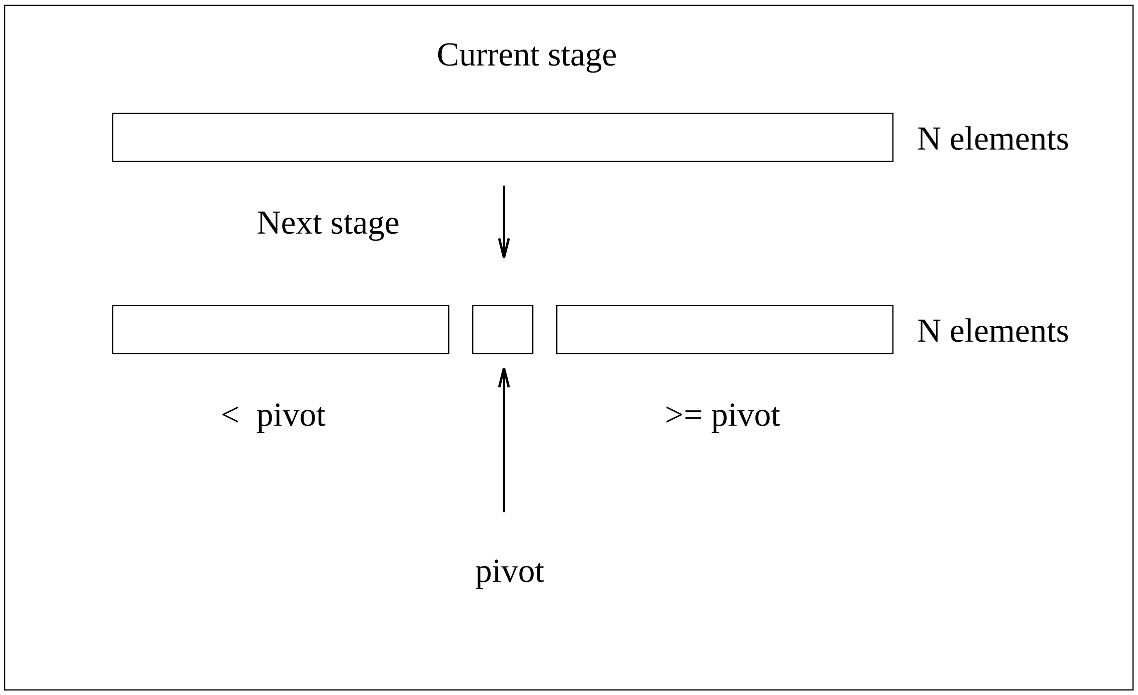

Fig 4. Partition process in the quicksort algorithm

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.

No matter what level of partitioning we consider (except near the end, when most elements are pivots), the amount of work in shuffling elements is $O(N)$ steps at each partition level.

For optimal pivot choice, each level produces partitions of half the size in the previous level.

The number of levels generated, in which N elements of the original unsorted partition can be sub-partitioned by halving 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 $O(N \log_2 N)$

Unfortunately, optimum partioning cannot be carried out while maintaining $O(N)$ at each partition level.

Instead, a heuristically ‘sensible’ partition strategy is adopted (e.g. the median 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.

8.5.1 Exercise

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

The ratio

$$\frac{2N \log 2N}{N \log N} = 2\frac{1 + \log_2 N}{\log_2 N} = 2\left(1 + \frac{1}{\log_2 N}\right) \tag{24}$$

is only slightly greater than 2, when N is large.

For example, when $N = 1024$, the ratio is 2.2.

Thus, doubling the data size incurs slightly more than twice the amount of work, on average.

8.5.2 Quicksort experiment

The quicksort algorithm involves data **moves** (copy of element to a variable, e.g to the pivot), **swaps** (element interchanges) and element **comparisons**.

By putting counters into the program, the number of moves, swaps and comparisons at the end of a sort run can be printed out.

This was done on various sizes of randomly ordered data. It is expected that the random cases will follow the average cases.

Table 4. Quicksort on random data

N	Moves	Swaps	Comps
15	45	4	45
31	107	19	138
63	208	62	371
127	389	171	817
255	799	390	2015
511	1642	902	5151
1023	3389	2049	10861
2047	6756	4594	23821
4095	13211	10275	52711
8191	26981	22322	114479
16383	53327	48948	250685

The expected counts were also estimated theoretically, as $O(N)$ and $O(N \log N)$. (See CS21120 website course notes)

Table 5. Quicksort – ratio of counts (Table 4) with growth functions

N	$\frac{\text{Moves}}{N}$	$\frac{\text{Swaps}}{N \log_2 N}$	$\frac{\text{Comps}}{N \log_2 N}$
15	3.00000	0.068256	0.76787
31	3.45161	0.123714	0.89855
63	3.30159	0.164645	0.98521
127	3.06299	0.192662	0.92050
255	3.13333	0.191312	0.98844
511	3.21331	0.196191	1.12038
1023	3.31281	0.200321	1.06183
2047	3.30044	0.204037	1.05798
4096	3.22613	0.209103	1.07270
8191	3.29398	0.209633	1.07511
16383	3.25502	0.213411	1.09297

The ratios “settle down”, as expected for the correct choice of growth function.

8.6 How quickly can sorting be carried out?

This can be answered by theory.

Assumptions: Only key comparison may be used. Data are distinct.

This rules out bin sort as the primary sort algorithm.

(**Bin Sort:** Put each card of a shuffled deck into a place holder - we know its position in the sorted deck. A single pass through the deck completes the sorting - an $O(N)$ process).

Given N items to sort, there are $N! = 1 * 2 * \ldots * N$ possible configurations with which the sorting algorithm may be presented.

The sorting algorithm must effectively decide, on the basis of key comparisons only, 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.

Every time a key comparison is made, (with outcome A comes before B, or A comes after B) the number of candidates of the $N!$ permutations which might match with the given sequence *is at most halved.*

Therefore, the number of key comparisons needed to identify the original sequence is *at least* $\log_2 N!$.

This is a *minimum* requirement.

8.6.1 Estimation of $\log_2 N!$

$$\log N! = \log 1 + \log 2 + \log 3 + \ldots + \log(N)$$

Since the log function has such a slow growth, it spends "most of the time near $\log N$". For example, when N=100,000, terms 10,000 to 100,000 (90% of them) in the summed expression for $\log_{10} 100,000!$ have values (inclusively) between 4 and 5.

A very rough bound is therefore given by

$$\log N! < N \log N.$$

A better approximation is **Stirling's formula**:

$$\log_2 N! \approx N \log_2(N/e).$$

From this the correct result is plausible, that

$$\log_2 N! = O(N \log N)$$

as illustrated Table 6:

Table 6. 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

Since $O(N \log N)$ sorting algorithms can be found, this complexity bound can be reached, but **cannot be improved**.

Complexity of Algorithms

H. Holstein

Exercises:

Over the next few days, I will be adding extra examples, such as

1. Analysis of the bubble sort, as covered in the lectures

2. Some other examples, such as set representation - how costly is it to determine set membership, the common elements of two sets, (set intersection)? What data structures are suitable, and what is their performance?

These slides are now avaiableL 58-73.

3. Set representation is dealt with in: Data structures and algorithms. Aho, Hopcroft and Ullman. Addison Wesley, 1983.

9.0 Complexity of program constructs

9.1 Complexity of sequential code Code that is truly sequential (no embedded loops, no function or subroutine calls that contain loops) can be executed in *constant* time, and has therefore a complexity $O(1)$.

9.2 Complexity of branched statements

An **if** statement, in which each limb is sequential code, will give rise to $O(1)$ complexity in each branch, and therefore has $O(1)$ complexity overall.

In more refined estimates, it might be necessary to allocate the different branches a different weighting - though still constant.

9.3 Complexity of loops

Sequential code contained within a singly nested loop (or a recursive call) has a complexity equal to the number of loop passes (or recursive calls). This is important when the number of loop passes grows with the data size.

In many situations, the extent of a loop (e.g. a **while** loop) may not be known in advance. In that case, some estimated average count is often sufficient to give a reasonable complexity estimate.

Other estimates lead to best case and worst case performances.

For example, a tree search can degenerate to a sequential search in a very unbalanced tree, in the worst case (making it $O(N)$).

9.4 Complexity of nested loops

Consider:

```
some code A;
for i from 1 to n do
  some code B;
  for j from 1 to m do
    some code C;
  end loop j;
end loop i;
```

In this segment, `code A`, `code B`, `code C` respectively have complexity $O(1), O(n)$, $O(n * m)$. The innermost loop, having $n * m$ passes, will dominate the computation (if m is not trivially small).

Nevertheless, these are all constant, and hence $O(1)$, unless n and m depend on the data size.

In cases where the inner loop passes do depend on data size, the complexity contribution is likely to be significant.

Example - Matrix operations

Consider two $n \times n$ matrices A and B.

The sum $C = A + B$ is given by the $O(n^2)$ code

```
for i from 1 to n do
for j from 1 to n do
    C[i,j] := A[i,j] + B[i,j];
end loop j;
end loop i;
```

The product $C = A * B$ is given by the $O(n^3)$ code

```
for i from 1 to n do
for j from 1 to n do
    cij := 0;                           // O(n^2)
for k from 1 to n do
    cij := cij + A[i,k] * B[k,j]; // O(n^3)
end loop k;
    C[i,j] := cij;                      // O(n^2)
end loop j;
end loop i;
```

Note that although the code has some $O(n^2)$ parts, its running time is dominated by the inner most $O(n^3)$ loop when n is large.

When the loops are nested, but the length of one loop depends on the length of the other, some estimate of the number of inner loop passes must be made. This occurs in the bubble sort:

```
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  -- compare
        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

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

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

$$\begin{aligned} &(1 + (N-1)) + (2 + (N-2)) + \ldots \\ &\ldots + ((N-2) + 2) + ((N-1) + 1) \\ &= (N-1)N \end{aligned} \tag{2}$$

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

The above count is in fact the maximum possible, and corresponds to the worst case (reverse sorted data).

If the data is already sorted, then the inner loop has only one pass, and the algorithm becomes $O(n)$. This is the best case.

On average, one may assume that the inner loop is performed about half the possible cases, because of the branching `if` statement. That still leaves $(N-1)N/4$ passes, which is again $O(N^2)$ complexity.

The code contains an early exit - if no swaps have taken place, then the data is sorted, and no further loop passes need take place.

While this can dramatically reduce the number of passes in individual data cases, the effect is negligible on the average running time.

Even for an inner loop pass as low as 20, the probability of failing the swap test 20 consecutive times is one part in 2^{20}, or one part in a million. The chance of this test having a significant saving early on in the sort process is negligible.

9.5 Case study bubble sort

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

Table 7. Bubble sort counts

```
   N     Swaps     Comps   Swaps/N^2  Comps/N^2
 ---     -----     -----   ---------  ---------
  15        32        90   1.422E-01  4.000E-01
  31       180       465   1.873E-01  4.839E-01
  63       864      1943   2.177E-01  4.895E-01
 127      3806      7998   2.360E-01  4.959E-01
 255     15020     32330   2.310E-01  4.972E-01
 511     63051    130277   2.415E-01  4.989E-01
1023    257493    522123   2.460E-01  4.989E-01
2047   1021374   2093730   2.438E-01  4.997E-01
```

The `Comps` heading counts the number of comparisons, that is, the number of executions of the `if` statement.

The number of swaps are indeed about half the number of comparisons, as suggested above.

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.

These counter values may be compared with the corresponding values for the quicksort algorithm, given above (section 8.5, Table 4).

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.

10.0 Further Complexity examples

How quickly can we find an item in some given data structure of N items?

Stated more abstractly, how quickly can we determine set membership?

- unsorted list $N/2$ lookups on average, $O(N)$.
- sorted list binary search - $O(\log N)$.
- binary search tree - $O(\log N)$.
- hashing - $O(1)$, assuming low load.

How quickly can we merge two data structures, each containing about N items?

Stated more abstractly, how quickly can we determine the union of two sets, while maintaining the data structure that stores them?

- unsorted list $O(1)$ - join the lists.
- sorted lists: merge sort - $O(N)$.
- binary search tree - $O(N \log N)$ (N insertions, each taking about $\log N$ steps.
- hashing - $O(N)$, assuming low load. (Ideally, each insert is $O(1)$ steps).

End of slides