The Growth of Functions

Rosen 3.2

What affects runtime of a program?

- the machine it runs on
- the programming language
- the efficiency of the compiler
- the size of the input
- the efficiency/complexity of the algorithm?

What has the greatest effect?

What affects runtime of an algorithm?

- the size of the input
- the efficiency/complexity of the algorithm?

We measure the number of times the "principal activity" of the algorithm is executed for a given input size n

One easy to understand example is search, finding a piece of data in a data set

N is the size of the data set
"principal activity" might be comparison of key with data

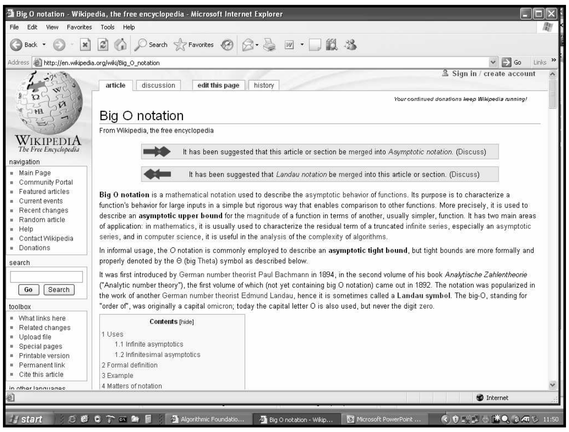
What are we interested in? Best case, average case, or worst case?

What affects runtime of an algorithm?

Therefore we express the complexity of an algorithm as a function of the size of the input

This function tells how the number of times the "principal activity" performed ***grows*** as the input size grows.

This does not give us an exact figure, but a growth rate.
This allows us to compare algorithms theoretically



Big O notation - Wikipedia, the free encyclopedia - Microsoft Internet Explorer

http://en.wikipedia.org/wiki/Big_O_notation

Big O notation

From Wikipedia, the free encyclopedia

It has been suggested that this article or section be merged into *Asymptotic notation.* (Discuss)

It has been suggested that *Landau notation* be merged into this article or section. (Discuss)

Big O notation is a mathematical notation used to describe the asymptotic behavior of functions. Its purpose is to characterize a function's behavior for large inputs in a simple but rigorous way that enables comparison to other functions. More precisely, it is used to describe an **asymptotic upper bound** for the magnitude of a function in terms of another, usually simpler, function. It has two main areas of application: in mathematics, it is usually used to characterize the residual term of a truncated infinite series, especially an asymptotic series, and in computer science, it is useful in the analysis of the complexity of algorithms.

In informal usage, the *O* notation is commonly employed to describe an **asymptotic tight bound**, but tight bounds are more formally and properly denoted by the Θ (big Theta) symbol as described below.

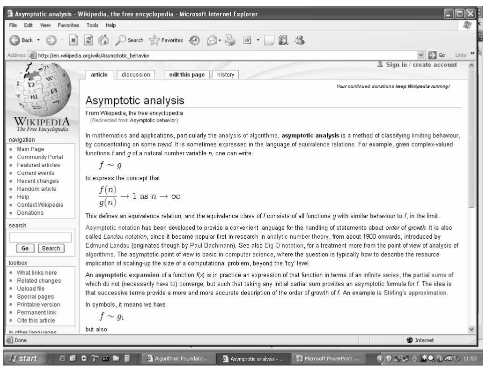
It was first introduced by German number theorist Paul Bachmann in 1894, in the second volume of his book *Analytische Zahlentheorie* ("Analytic number theory"), the first volume of which (not yet containing big O notation) came out in 1892. The notation was popularized in the work of another German number theorist Edmund Landau, hence it is sometimes called a **Landau symbol**. The big-O, standing for "order of", was originally a capital omicron; today the capital letter O is also used, but never the digit zero.

Contents [hide]

1 Uses
1.1 Infinite asymptotics
1.2 Infinitesimal asymptotics
2 Formal definition
3 Example
4 Matters of notation

The Big-O Notation

… gives the asymptotic behaviour of a function.



Asymptotic analysis - Wikipedia, the free encyclopedia - Microsoft Internet Explorer

http://en.wikipedia.org/wiki/Asymptotic_behavior

Asymptotic analysis

From Wikipedia, the free encyclopedia
(Redirected from Asymptotic behavior)

In mathematics and applications, particularly the analysis of algorithms, **asymptotic analysis** is a method of classifying limiting behaviour, by concentrating on some *trend*. It is sometimes expressed in the language of equivalence relations. For example, given complex-valued functions *f* and *g* of a natural number variable *n*, one can write

$$f \sim g$$

to express the concept that

$$\frac{f(n)}{g(n)} \to 1 \text{ as } n \to \infty$$

This defines an equivalence relation; and the equivalence class of *f* consists of all functions *g* with similar behaviour to *f*, in the limit.

Asymptotic notation has been developed to provide a convenient language for the handling of statements about *order of growth*. It is also called *Landau notation*, since it became popular first in research in analytic number theory, from about 1900 onwards, introduced by Edmund Landau (originated though by Paul Bachmann). See also Big O notation, for a treatment more from the point of view of analysis of algorithms. The asymptotic point of view is basic in computer science, where the question is typically how to describe the resource implication of scaling-up the size of a computational problem, beyond the 'toy' level.

An **asymptotic expansion** of a function *f(x)* is in practice an expression of that function in terms of an infinite series, the partial sums of which do not (necessarily have to) converge; but such that taking any initial partial sum provides an asymptotic formula for *f*. The idea is that successive terms provide a more and more accurate description of the order of growth of *f*. An example is Stirling's approximation.

In symbols, it means we have

$$f \sim g_1$$

but also

The Big-O Notation

f(x) is O(g(x))
if there are constants C and k such that
$|f(x)| \le C.|g(x)|$
whenever x > k

C and k are called "witnesses to the relationship"
There may be many witnesses

f(x) is big-O of g(x)
f(x) is order g(x)

The Big-O Notation

example

$$f(x) = x^2 + 2x + 1 \text{ is } O(x^2)$$

$$x^2 + 2x + 1 \le C.x^2 \quad \text{when } x > k$$

$$0 \le x^2 + 2x + 1 \le x^2 + 2x^2 + x^2 = 4x^2$$

Whenever x is greater than 1 the above holds,

consequently ... $f(x) = x^2 + 2x + 1 \text{ is } O(x^2)$

The Big-O Notation

example

$$f(x) = x^2 + 2x + 1 \text{ is } O(x^2)$$

Note also:

$$x^2 + 2x + 1 \text{ is } O(x^3)$$

But we prefer the former.

The Big-O Notation

Another example

$$f(n) = n^2 \text{ is not } O(n)$$

i.e. not linear

$\therefore n^2 \le C.n$ for some $n > k$
$\therefore \frac{n^2}{n} \le \frac{C.n}{n}$
$\therefore n \le C$
But n is a variable and C is a constant
This is impossible

The Big-O Notation

Yet another example

Show that $n!$ is $O(n^n)$

$\therefore n! \le C.n^n$ for some $n > k$
$\therefore 1.2.3.\cdots.n \le n.n.n.\cdots.n$
With $C = 1$ and $k = 1$ we have $n! = O(n^n)$

Complexity of bubble sort

```
bubbleSort(A:array,n:int)
  for i := 1 to n-1
    for j := 1 to n-I
      if A[j] > A[j+1]
      then swap(A,j,j+1)
```

See 3.1, Sorting pp 172 onwards

Complexity of bubble sort

```
bubbleSort(A:array,n:int)
 for i := 1 to n-1
   for j := 1 to n-i
     if A[j] > A[j+1]
     then swap(A,j,j+1)
```

First time round the outer loop (i=1) the inner loop executes the "if" statement 1 to n-1 times

Second time round the outer loop (i=2) the inner loop executes the "if" statement 1 to n-2 times

Third time round the outer loop (i=2) the inner loop executes the "if" statement 1 to n-3 times

n-1th time round the outer loop (i=n-1) the inner loop executes the "if" statement 1 to n-(n-1) times

How many times is the inner "if" conditional statement executed
(i.e. this is our principal activity)?

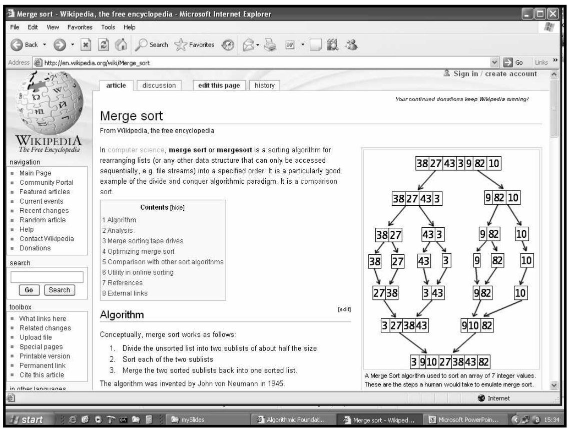
Complexity of bubble sort

```
bubbleSort(A:array,n:int)
 for i := 1 to n-1
   for j := 1 to n-i
     if A[j] > A[j+1]
     then swap(A,j,j+1)
```

$$
\begin{aligned}
&(n-1)+(n-2)+(n-3)+\cdots+(n-(n-2))+(n-(n-1))\\
&=1+2+3+\cdots+n-1\\
&=\sum_{1}^{n-1} i\\
&=\frac{(n-1)((n-1)+1)}{2}\\
&=\frac{n(n-1)}{2}\\
&=O(n^2)
\end{aligned}
$$

Complexity of merge sort

$$O(n.\log(n))$$


Merge sort - Wikipedia, the free encyclopedia - Microsoft Internet Explorer

http://en.wikipedia.org/wiki/Merge_sort

Merge sort

From Wikipedia, the free encyclopedia

In computer science, **merge sort** or **mergesort** is a sorting algorithm for rearranging lists (or any other data structure that can only be accessed sequentially, e.g. file streams) into a specified order. It is a particularly good example of the divide and conquer algorithmic paradigm. It is a comparison sort.

Contents [hide]

1 Algorithm
2 Analysis
3 Merge sorting tape drives
4 Optimizing merge sort
5 Comparison with other sort algorithms
6 Utility in online sorting
7 References
8 External links

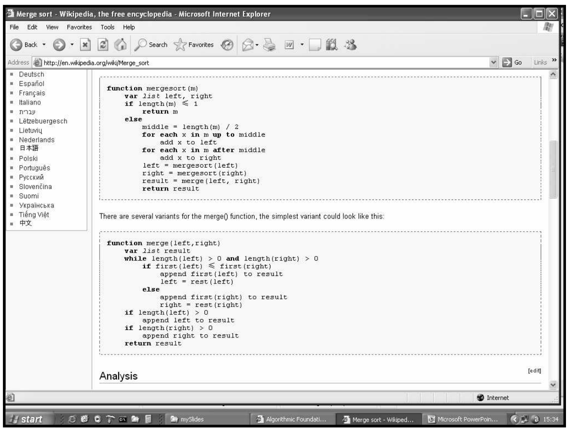
Algorithm

Conceptually, merge sort works as follows:

1. Divide the unsorted list into two sublists of about half the size
2. Sort each of the two sublists
3. Merge the two sorted sublists back into one sorted list.

The algorithm was invented by John von Neumann in 1945.

A Merge Sort algorithm used to sort an array of 7 integer values. These are the steps a human would take to emulate merge sort.


Merge sort - Wikipedia, the free encyclopedia - Microsoft Internet Explorer

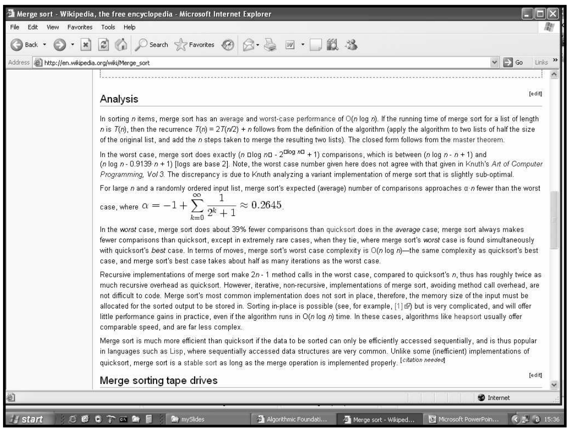
http://en.wikipedia.org/wiki/Merge_sort

```
function mergesort(m)
    var list left, right
    if length(m) ≤ 1
        return m
    else
        middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = mergesort(left)
        right = mergesort(right)
        result = merge(left, right)
        return result
```

There are several variants for the merge() function, the simplest variant could look like this:

```
function merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    if length(left) > 0
        append left to result
    if length(right) > 0
        append right to result
    return result
```

Analysis


Merge sort - Wikipedia, the free encyclopedia - Microsoft Internet Explorer

http://en.wikipedia.org/wiki/Merge_sort

Analysis

In sorting n items, merge sort has an average and worst-case performance of O(n log n). If the running time of merge sort for a list of length n is $T(n)$, then the recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists). The closed form follows from the master theorem.

In the worst case, merge sort does exactly (n ⌈log n⌉ - $2^{\lceil \log n \rceil}$ + 1) comparisons, which is between (n log n - n + 1) and (n log n - 0.9139·n + 1) [logs are base 2]. Note, the worst case number given here does not agree with that given in Knuth's *Art of Computer Programming, Vol 3*. The discrepancy is due to Knuth analyzing a variant implementation of merge sort that is slightly sub-optimal.

For large n and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches α·n fewer than the worst case, where $\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k+1} \approx 0.2645$.

In the *worst* case, merge sort does about 39% fewer comparisons than quicksort does in the *average* case; merge sort always makes fewer comparisons than quicksort, except in extremely rare cases, when they tie, where merge sort's *worst* case is found simultaneously with quicksort's *best* case. In terms of moves, merge sort's worst case complexity is O(n log n)—the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case.

Recursive implementations of merge sort make 2n - 1 method calls in the worst case, compared to quicksort's n, thus has roughly twice as much recursive overhead as quicksort. However, iterative, non-recursive, implementations of merge sort, avoiding method call overhead, are not difficult to code. Merge sort's most common implementation does not sort in place, therefore, the memory size of the input must be allocated for the sorted output to be stored in. Sorting in-place is possible (see, for example, [1]) but is very complicated, and will offer little performance gains in practice, even if the algorithm runs in O(n log n) time. In these cases, algorithms like heapsort usually offer comparable speed, and are far less complex.

Merge sort is much more efficient than quicksort if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some (inefficient) implementations of quicksort, merge sort is a stable sort as long as the merge operation is implemented properly. [citation needed]

Merge sorting tape drives

Or go see Rosen 4.4 Recursive Algorithms (page 317 onwards)

52.219 Complexity

Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n).

What effects run time of an algorithm?

```
(a) computer used, the harware platform
(b) representation of abstract data types (ADT's)
(c) efficiency of compiler
(d) competence of implementer (programming skills)
(e) complexity of underlying algorithm
(f) size of the input
```

We will show that of those above (e) and (f) are generally the most significant

Time for an algorithm to run t(n)

A function of input. However, we will attempt to characterise this by the size of the input. We will try and estimate the WORST CASE, and sometimes the BEST CASE, and very rarely the AVERAGE CASE.

What do we measure?

In analysing an algorithm, rather than a piece of code, we will try and predict the number of times "the principle activity" of that algorithm is performed. For example, if we are analysing a sorting algorithm we might count the number of comparisons performed, and if it is an algorithm to find some optimal solution, the number of times it evaluates a solution. If it is a graph colouring algorithm we might count the number of times we check that a coloured node is compatible with its neighbours.

Worst Case

... is the maximum run time, over all inputs of size n, ignoring effects (a) through (d) above. That is, we only consider the "number of times the principle activity of that algorithm is performed".

Best Case

In this case we look at specific instances of input of size n. For example, we might get best behaviour from a sorting algorithm if the input to it is already sorted.

Average Case

Arguably, average case is the most useful measure. It might be the case that worst case behaviour is pathological and extremely rare, and that we are more concerned about how the algorithm runs in the general case. Unfortunately this is typically a very difficult thing to measure. Firstly, we must in some way be able to define by what we mean as the "average input of size n". We would need to know a great deal about the distribution of cases throughout all data sets of size n. Alternatively we might make a possibly dangerous assumption that all data sets of size n are equally likely. Generally, in order to get a feel for the average case we must resort to an empirical study of the algorithm, and in some way classify the input (and it is only recently with the advent of high performance, low cost computation, that we can seriously consider this option).

The Growth rate of t(n)

Suppose the worst case time for algorithm A is

```
t(n) = 60*n*n + 5*n + 1
```

for input of size n.

Assume we have differing machine and compiler combinations, then it is safe to say that

```
t(n) = n*n + 5*n/60 + 1/60
```

The Growth rate of t(n)

Suppose the worst case time for algorithm A is

t(n) = 60*n*n + 5*n + 1

for input of size n.

Assume we have differing machine and compiler combinations, then it is safe to say that

t(n) = n*n + 5*n/60 + 1/60

That is, we ignore the coefficient that is applied to the most significant (dominating) term in t(n). Consequently this only affects the "units" in which we measure. It does not affect how the worst case time grows with n (input size) but only the units in which we measure worst case time Under these assumptions we can say ...

```
 "t(n) grows like n*n as n increases"

                or

          t(n) = O(n*n)
```

which reads "t(n) is of the order n squared" or as "t(n) is big-oh n squared"

In summary, we are interested only in the dominant term, and we ignore coefficients.

An Example (the tyranny of growth)

Tabulated below, are a number of functions against n (from 1 to 10)

In summary, we are interested only in the dominant term, and we ignore coefficients.

An Example (the tyranny of growth)

Tabulated below, are a number of functions against n (from 1 to 10)

```
A = (log2 n) (log to base 2 of n)
B = n (linear in n)
C = (* n (log2 n)) (n log n)
D = (* n n) (quadratic in n)
E = (* n n n) (cubic in n)
F = (expt 2 n) (exponential in n)
G = (expt 3 n) (exponential in n)
H = (fact n) (factorial in n)
```

Good (A–E)

Intractable (F–H)

n	A	B	C	D	E	F	G	H
1	0.0	1	0.0	1	1	2	3	1
2	1.0	2	2.0	4	8	4	9	2
3	1.0	3	4.0	9	27	8	27	6
4	2.0	4	8.0	16	64	16	81	24
5	2.0	5	11.0	25	125	32	243	120
6	2.0	6	15.0	36	216	64	729	720
7	2.0	7	19.0	49	343	128	2187	5040
8	3.0	8	24.0	64	512	256	6561	40320
9	3.0	9	28.0	81	729	512	19683	362880
10	3.0	10	33.0	100	1000	1024	59049	3628800

Think of this as algorithms A through H with complexities as defined above, showing growth rate versus input size n. Tabulated below are functions F, G and H from above (ie 2 to power n, 3 to power n, and n factorial). Problem size n varies from 10 to 100 in steps of 10. I have assumed that we have a machine that can perform "the principle activity of the algorithm" in a micro second (ie. if we are considering a graph colouring algorithm, it can compare the colour of two nodes in a millionth of a second). The columns give the number of years this machine would take to execute those algorithms on problems of size n (note: YEARS). This is expressed as 10^x, "10 raised to the power x".

9	3.0	9	28.0	81	729	512	19683	362880
10	3.0	10	33.0	100	1000	1024	59049	3628800

Think of this as algorithms A through H with complexities as defined above, showing growth rate versus input size n. Tabulated below are functions F, G and H from above (ie 2 to power n, 3 to power n, and n factorial). Problem size n varies from 10 to 100 in steps of 10. I have assumed that we have a machine that can perform "the principle activity of the algorithm" in a micro second (ie. if we are considering a graph colouring algorithm, it can compare the colour of two nodes in a millionth of a second). The columns give the number of years this machine would take to execute those algorithms on problems of size n (note: YEARS). This is expressed as 10^x, "10 raised to the power x".

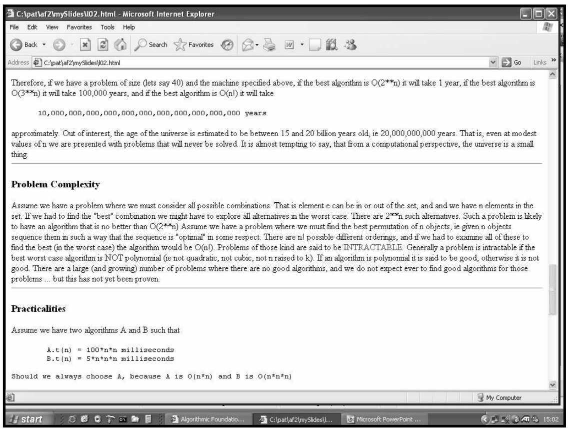
n	F	G	H
10	10^-10	10^-8	10^-6
20	10^-7	10^-3	10^4
30	10^-4	10^0	10^18
40	10^-1	10^5	10^34
50	10^1	10^10	10^50
60	10^4	10^15	10^68
70	10^7	10^19	10^86
80	10^10	10^24	10^105
90	10^13	10^29	10^124
100	10^16	10^34	10^144

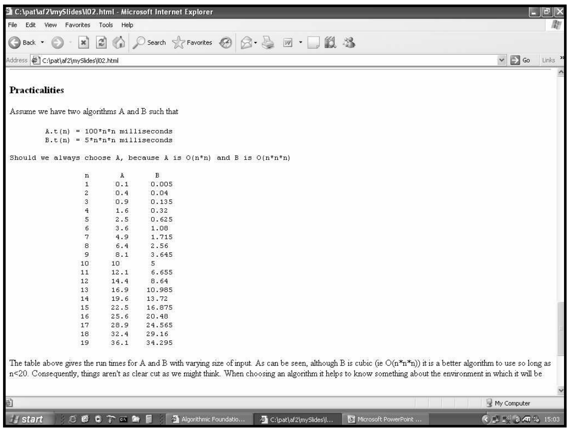
Therefore, if we have a problem of size (lets say 40) and the machine specified above, if the best algorithm is O(2**n) it will take 1 year, if the best algorithm is O(3**n) it will take 100,000 years, and if the best algorithm is O(n!) it will take

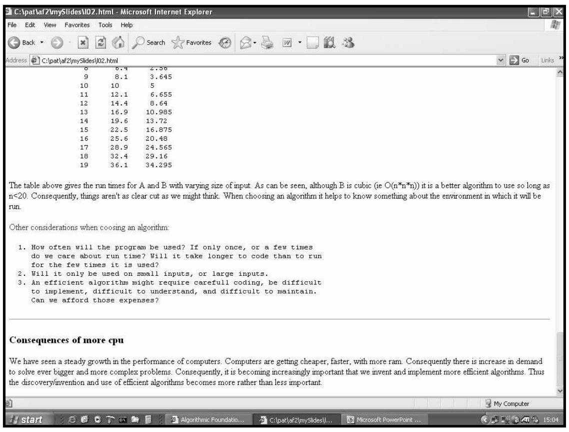
```
10,000,000,000,000,000,000,000,000,000,000,000,000 years
```

approximately. Out of interest, the age of the universe is estimated to be between 15 and 20 billion years old, ie 20,000,000,000 years. That is, even at modest values of n we are presented with problems that will never be solved. It is almost tempting to say, that from a computational perspective, the universe is a small thing.

Problem Complexity







A demo?

Sorting using the java demo's?

A rough cut calculation of runtimes on different data set sizes?