The Efficiency of Algorithms and Big O Notation

The Efficiency of Algorithms

An efficient algorithm is one that runs as *fast* as possible and requires as little computer *memory* as possible. You often have to settle for a trade-off between these two goals. For example, searching an array of *n* elements is *faster* than searching a linked-list of the same size. The linked-list, on the other hand, would require less memory.

So, what do we mean by faster? About the only measure of speed we can apply to an algorithm is to count the number of operations that it has to perform to produce its result. The fewer the number of operations, the faster it will execute. This obviously has a relationship with the *time* that it will take to execute, but so do a lot of other factors such as processor speed, amount of memory available etc. Therefore any attempt to give an absolute measure of time would be pretty much meaningless.

There are three ways of expressing performance: Best-case, worst-case and average case. Consider the task of performing a sequential search on some sort of list, e.g. an array. Best-case would be that your target value was found in the first element. Worst-case would be that the value was not there at all (so all elements would have to be compared and tested, including the last). Average-case would be mid-way between the two.

Of course, some operations would remain constant, regardless of the size of the list: **`write`** `myarray[n];` would always execute in one move, no matter how many elements there were.

Big O notation.

We have already seen that efficiency is defined as the number of operations an algorithm has to perform to achieve its result. Big O notation is simply a convenient theoretical way of measuring the execution of an algorithm, therefore expressing its efficiency.

A Big O expression always presents the worst-case scenario. (By the way, the “O” is really the upper case Greek letter omicron, but this looks just like the letter O!)

Big O notation says that the computing *time* of an algorithm grows **no faster** (as Big O presents a worst-case scenario) than a constant multiplied by f(n). The constant is machine specific (as we mentioned earlier it depends on such factors as processor speed) so is not usually included in a generalised Big O expression. Big O therefore is most useful as a comparative measure between algorithms rather than an absolute measure.

Linear search has a time efficiency (an **order**) of O(n), where n is the number of elements being searched. If the list doubles in size, the time taken also doubles. A bubble sort is determined to be of order $O(n^2)$ so the time taken quadruples if n doubles.

Here is a table of the relevant functions for different types of algorithm, expressed in order of efficiency from the fastest to the slowest:

Big O notation	**How performance varies with** n	Typical algorithms:
O (1)	Constant, size of n doesn’t matter	‘One-shot’ statements like the `write` example given.*

$O(\log_2 n)$	Logarithmic	Binary search
$O(n)$	Linear	Linear search
$O(n \log_2 n)$		Quicksort
$O(n^2)$	Quadratic	Bubble sort, selection sort.
$O(n^3)$	Cubic	Not examined at HL IB. Included only for interest and completeness!
$O(m^n)$**	Exponential	
O (n!)	Factorial	

n **is the number of items that must be handled by the algorithm.**

* Statements which always 'hit the mark' first time have this notation. A further example would be a perfect hashing algorithm, i.e. one that will always hash to the correct location without needing any overflow structure.

** m is an integer greater than 1.

Past paper question on efficiency:

(Higher Level Paper 1, November 1992)

(a) Given the subalgorithm below, state the exact number of statements in **terms of N** that will be executed by this subalgorithm. Describe how the answer is derived.

Assume array A is an array of integers and the number of entries in A is N. I, J and T are integers.

```
TEST
   I <-- 0
   While I <= N do
      J <-- 0
      T <-- 0
      While J <= N do
         T <-- T + A(J)
         J <-- J + 1
      enddo
       A(I) <-- T
      I <-- I + 1
```

enddo

end TEST

(b) The following questions relate to methods of searching. Briefly explain your reasoning in each case.

(i) Of the two search methods, binary and sequential, which one is usually the faster when the file is short? Which one is usually the faster when the file is long? *[2 marks]*

(ii) Which of these two methods requires the list to be sorted? *[2 marks]*

(iii) When performing a sequential search on a list of size N, for the worst possible case, what is the number of items that must be compared to locate the desired item?

[3 marks]

(iv) When performing a sequential search on a list of size N, for the average case, what is the number of items that must be compared to locate the desired item?

[3 marks]

(v) When performing a binary search on a list of size N, for the worst possible case, what is the approximate number of items that must be compared to locate the desired item? *[3 marks]*

(see also May '97 paper 1 question 5 and May '98 paper 2 question 1)

big-O notation

(definition)

Definition: A theoretical measure of the execution of an *algorithm*, usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation f(n) = O(g(n)) means it is less than some constant multiple of g(n). The notation is read, "f of n is big oh of g of n".

Formal Definition: f(n) = O(g(n)) means there are positive constants c and k, such that $0 \leq f(n) \leq cg(n)$ for all $n \geq k$. The values of c and k must be fixed for the function f and must not depend on n.

Also known as O.

See also Ω(*n*), ω(*n*), Θ(*n*), ∽, *little-o notation*, *asymptotic upper bound*, *asymptotically tight bound*, *NP*, *complexity*, *model of computation*.

Note: As an example, $n^2 + 3n + 4$ is $O(n^2)$, since $n^2 + 3n + 4 < 2n^2$ for all $n > 10$. Strictly speaking, $3n + 4$ is $O(n^2)$, too, but big-O notation is often misused to mean equal to rather than less than. The notion of "equal to" is expressed by Θ(n).

The importance of this measure can be seen in trying to decide whether an algorithm is adequate, but may just need a better implementation, or the algorithm will always be too slow on a big enough input. For instance, quicksort, which is O(n log n) on average, running on a small desktop computer can beat bubble sort, which is

$O(n^2)$, running on a supercomputer if there are a lot of numbers to sort. To sort 1,000,000 numbers, the quicksort takes 20,000,000 steps on average, while the bubble sort takes 1,000,000,000,000 steps!

Any measure of execution must implicitly or explicitly refer to some computation model. Usually this is some notion of the limiting factor. For one problem or machine, the number of floating point multiplications may be the limiting factor, while for another, it may be the number of messages passed across a network. Other measures which may be important are compares, item moves, disk accesses, memory used, or elapsed ("wall clock") time.

Strictly, the character is the upper-case Greek letter Omicron, not the letter O, but who can tell the difference?

Author: PEB

More information

A rough guide to big-oh notation by Mark Dunlop. Tutorial on complexity classes illustrated with several sort algorithms.

STUDENT OUTLINE

Lesson 20: Order of Algorithms

INTRODUCTION: The two criteria used for selecting a data structure and algorithm are the amount of memory required and the speed of execution. The analysis of the speed of an algorithm leads to a summary statement called the order of an algorithm.

The key topics for this lesson are:

A. Order of Algorithms
B. Constant Algorithms, O(1)
C. Log2N Algorithms, O(log2N)
D. Linear Algorithms, O(N)
E. N * Log2N Algorithms, O(N * log2N)
F. Quadratic Algorithms, (N2)
G. Other Orders
H. Comparison of Orders of Algorithms

VOCABULARY:

ORDER OF ALGORITHM
LOG2 N
N LOG2 N
CUBIC
CONSTANT
LINEAR
QUADRATIC
BIG O NOTATION

DISCUSSION:

A. Order of Algorithms

1. The order of an algorithm is based on the number of steps that it takes to complete a task. Time is not a valid measuring stick because computers have different processing speeds. We want a method of comparing algorithms that is independent of computing environment and microprocessor speeds.

2. Most algorithms solve problems involving an amount of data, N. The order of algorithms will be expressed as a function of N, the size of the data set.

3. The following chart summarizes the numerical relationships of common functions of N.

A	B	C	D	E
N	$O(\log_2 N)$	O(N)	$O(N*\log_2 N)$	$O(N^2)$
1	0	1	0	1
2	1	2	2	4
4	2	4	8	16
8	3	8	24	64
16	4	16	64	256
32	5	32	160	1024
64	6	64	384	4096
128	7	128	896	16384
256	8	256	2048	65536
512	9	512	4608	262144
1024	10	1024	10240	1048576
2048	11	2048	22528	4194304
4096	12	4096	49152	16777216
8192	13	8192	106496	67108864
16384	14	16384	229376	268435456
32768	15	32768	491520	1073741824
65536	16	65536	1048576	4294967296
131072	17	131072	2228224	17179869184
262144	18	262144	4718592	68719476736
524288	19	524288	9961472	2.74878E+11
1048576	20	1048576	20971520	1.09951E+12
2097152	21	2097152	44040192	4.39805E+12

a. The first column, N, is the number of items in a data set.
b. The other four columns are mathematical functions based on the size of N. In computer science, we write this with a capital O (order) instead of the traditional F (function) of mathematics. This type of notation is the order of an algorithm, or Big O notation.

c. You have already seen the last column in an empirical sense when you counted the number of steps in the quadratic sorting algorithms. The relationship between columns A and E is quadratic - as the value of N increases, the other column increases as a function of N2.

d. As we work through the rest of the student outline, assume the following array declaration of list applies:

```
int[] list = new int[4001];
```

Here are the specifications of array `list`:

1. Index position 0 keeps track of how many integers are stored as data.
2. Integers are stored from positions `list[1] ... list[list[0]]`.

B. Constant Algorithms, O(1)

1. This relationship was not included in the chart. Here, the size of the data set does not affect the number of steps this type of algorithm takes. For example:\

```
int howBig (int[] list)
{
    return list[0];
}
```

2. The number of data in the array could vary from 0..4000, but this does not affect the algorithm of howBig. It will take one step regardless of how big the data set is.

3. A constant time algorithm could have more than just one step, as long as the number of steps is independent of the size (N) of the data set.

C. Log2N Algorithms, O(log2N)

1. A logarithm is the exponent to which a base number must be raised to equal a given number.

2. A log2N algorithm is one where the number of steps increases as a function of log2N. If the number of data was 1024, the number of steps equals log2 1024, or 10 steps.

3. Algorithms in this category involve splitting the data set in half repeatedly. Several examples will be encountered later in the course.

4. Algorithms which fit in this category are classed as O(log N),

regardless of the numerical base used in the analysis.

D. Linear Algorithms, O(N)

1. This is an algorithm where the number of steps is directly proportional to the size of the data set. As N increases, the number of steps also increases.

```
long sumData (int[] list)
//  sums all the values in the array
{
  long total = 0;

  for (int loop = 1; loop <= list[0]; loop++)
  {
    total += list[loop];
  }
  return total;
}
```

2. In the above example, as the size of the array increases, so does the number of steps in the function.

3. A non-recursive linear algorithm, O(N), always has a loop involved.

4. Recursive algorithms are usually linear where the looping concept is developed through recursive calls. The recursive factorial function is a linear function.

```
long fact (int  n)
//  precondition:  n > 0
{
  if (1 == n)
    return 1;
  else
    return  n * fact(n - 1);
}
```

The number of calls of fact will be n. Inside of the function is one basic step, an if/then/else. So we are executing one statement n times.

E. N * Log2N Algorithms, O(N * log2N)

1. Algorithms of this type have a log N concept that must be applied N times.

2. When recursive MergeSort and Quicksort are covered, we will discover that they are O(N * log2N) algorithms.

3. These algorithms are markedly more efficient than our next

category, quadratics.

F. Quadratic Algorithms, (N^2)

1. This is an algorithm where the number of steps required to solve a problem increases as a function of N2. For example, here is bubbleSort.

```
void bubbleSort (int[][] list)
{
  for (int outer = 1; outer <= list[0]-1; outer++)
  {
    for (int inner = 1; inner <= list[0]-outer; inner+
+)
    {
      if (list[inner] > list[inner + 1])
      {
        // swap list[inner] & list[inner + 1]
        int temp = list[inner];
        list[inner] = list[inner + 1]);
        list[inner + 1] = temp;
      }
    }
  }
}
```

2. The **if** statement is buried inside nested loops, each of which is tied to the size of the data set, N. The **if** statement is going to happen approximately N2 times.

3. The efficiency of this bubble sort was slightly improved by having the inner loop decrease. But we still categorize this as a quadratic algorithm.

4. For example, the number of times the inner loop happens varies from 1 to (N-1). On average, the inner loop occurs (N/2) times.

5. The outer loop happens (N-1) times, or rounded off N times.

6. The number of times the if statement is executed is equal to this expression:

if statements = (Outer loop) * (Inner loop)
if statements = (N) * (N/2)
if statements = (N^2)/2

7. Ignoring the coefficient of 1/2, we have an algorithm that is quadratic in nature.

8. When determining the order of an algorithm, we are only concerned with its category, not a detailed analysis of the number of

steps.

G. Other Orders

1. A cubic algorithm is one where the number of steps increases as a cube of N, or N^3.

2. An exponential algorithm is one where the number of steps increases as the power of a base, like 2^N.

3. Both of these categories are astronomical in the number of steps required. Such algorithms cannot be implemented on small personal computers.

H. Comparison of Orders of Algorithms

1. We obviously want to use the most efficient algorithm in our programs. Whenever possible, choose an algorithm that requires the fewest number of steps to process data.

2. The transparency, T.A.22.1, *Order vs. Efficiency in Algorithms*, summarizes all the categories in this lesson. Note that both axes in this diagram are exponential in scale.

SUMMARY/ REVIEW: When designing solutions to programming problems, we are concerned with the most efficient solutions regarding time and space. We will consider memory requirements at a later time. Speed issues are resolved based on the number of steps required by algorithms.