# THE ANALYSIS OF ALGORITHMS

Professor D. E. Knuth

Department of Computer Science,
Stanford University,
Stanford,
California, 94305, U.S.A.

Abstract:

The analysis of algorithms can be considered from several points of view, namely, 1) the detailed mathematical analysis of particular algorithms; 2) the overall complexity analysis of groups of algorithms for a particular problem; and 3) an empirical analysis of algorithms using an instruction frequency count. Examples of some typical analyses are presented. The techniques used are relevant to the sort of mathematics that should be taught to computer science students.

Rapporteurs:

Mr. L. B. Wilson
Mr. L. Waller

# THE ANALYSIS OF ALGORITHMS

Professor D. E. Knuth

Department of Computer Science,
Stanford University,
Stanford,
California, 94305, U.S.A.

Abstract:

The analysis of algorithms can be considered from several points of view, namely, 1) the detailed mathematical analysis of particular algorithms; 2) the overall complexity analysis of groups of algorithms for a particular problem; and 3) an empirical analysis of algorithms using an instruction frequency count. Examples of some typical analyses are presented. The techniques used are relevant to the work of mathematics that should be taught to computer science students.

1. Introduction

Professor Knuth began by explaining that the title 'The Analysis of Algorithms' described the work he tries to do in Computer Science. In fact this title had originally been intended for his series of books (Knuth (1968), Knuth (1969)) but the publishers thought 'The Art of Computer Programming' would seel better.

2. The theoretical analysis of algorithms is divided into two types:

Type A. Analysing a particular algorithm from the quantitative point of view, in order to see how good it is. For example, it is possible to predict the execution time of various algorithms for sorting.

Type B. 'Complexity' - the study of classes of algorithms. Given a particular task a study is conducted, seeking the best possible way of doing it. For example, under certain assumptions the sorting problem can be examined and decisions about the 'best' possible algorithms made.

2.1 Type A

2.1.1 Analysis of algorithm for finding the minimum

An algorithm for finding the minimum of N elements $\{X_1, X_2 \ldots X_N\} = X_k$, $N \geq 1$, will be analysed. This example, which is similar to the analysis of the algorithm for finding the maximum in Knuth (1968), Volume 1, pp. 95-102, will show the kind of mathematical techniques needed for this work.

An algorithm is

```
MIN:= X[N]; k:= N;
for j:= N-1 step -1 until 1 do
 if  X[j] < MIN  then
     begin MIN:= X[j]; k:= j;
     end;
```

It will be assumed that the algorithm is correct (this is reasonably self-evident in this simple case). The analysis of how good it is can be done by a count of the number of times each line is performed. The first two lines are performed once, the third line N-1 times and the fourth line a variable number of times, say T. The question now is what is the value of T? The value of T is seen to be

dependent on the data, and the worst case, the best case and the average case are examined.

Worst case: max T = N−1 when data is in strictly increasing order.

Best case: min T = 0 when X[N] is the minimum.

Average case: The mean value for T and the variance are of interest, assuming that the X's are distinct and in random order. (Other assumptions about the input distribution could also be made; choice of the input assumptions is often an important consideration.)

Consider the case N = 3. Let the X's be 1, 2, 3. There are six possible orderings of these numbers and these are given below together with the appropriate T value.

| Permutation | | | T | |
|---|---|---|---|---|
| ① | ② | 3 | 2 | ) |
| ① | 3 | 2 | 1 | ) |
| 2 | ① | 3 | 1 | ) average value of |
| 2 | 3 | 1 | 0 | ) |
| 3 | ① | 2 | 1 | ) $T = \frac{5}{6}$ |
| 3 | 2 | 1 | 0 | ) |
| | | | 5 | |

In the permutations circles have been put around each number that causes the fourth line in the algorithm to be obeyed, i.e., that increased T by 1. The result may now be generalised to the case of N elements. Two methods are shown for doing this.

Method 1. This depends on a trick in that the permutations are examined and a note made of how many numbers are circled in each column. Working from right to left there are no numbers circled in the last column, whilst in the second last column every other number is circled. This is because the number in column N − 1 is less than that in column N and this happens half the time. In the next column the numbers are less than those in the two previous columns one third of the time. Similarly for the other columns, and for the first column the value is $\frac{N!}{N}$.

Thus
$$\text{Average } T = \left(\frac{N!}{2} + \frac{N!}{3} + \frac{N!}{4} + \ldots + \frac{N!}{N}\right)\frac{1}{N!}$$
$$= \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \ldots \frac{1}{N} \simeq \log_e N$$

This series occurs so often in algorithmic analysis it is denoted by $H_N$ where $H_N = 1 + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{N}$.

N.B. for N = 10000, T is about 8.8.

Method 2. This method is a more systematic technique which will be applicable in the analyses of other algorithms. It also has the advantage of giving more information about T, not just the average.

Let $A_{Nk}$ = number of permutations of N elements with T = k,

e.g. $A_{30} = 2$, $A_{31} = 3$, $A_{32} = 1$.

The technique is to write down a recurrence relation for $A_{N+1,k}$ and then solve it.

To do this the extension of each permutation from N to N+1 is considered, for example a permutation may be extended from N to N+1 by making N+1 copies of each permutation of N marks and letting N+1 be placed in every position.

e.g. the permutation 1 3 2 would lead to

4 ① 3 2
① 4 3 2
① 3 4 2
① 3 ② 4 ;

the circled elements are unchanged except that one more element is circled when N+1 falls into the final position.

An alternative way to go from N to N+1 would extend 1 3 2 as follows:

① ② 4 3
2 ① 4 3
3 ① 4 2
4 ① 3 2

where after the first column the remaining marks are in the order 1 3 2. Again exactly one new permutation has T increased by 1. Examining either of these constructions gives the recurrence relation

$$A_{N+1,k} = NA_{Nk} + A_{Nk-1} . \qquad (1)$$

The initial conditions are

$$\text{if } N = 1, \quad A_{1k} = \delta_{0k} ; \quad \text{and } A_{Nk} = 0 \text{ if } k < 0 .$$

Thus, the following table for $A_{Nk}$ can be developed.

| N \ k | 0 | 1 | 2 | 3 . . . |
|---|---|---|---|---|
| 0 | | | | |
| 1 | 1 | | | |
| 2 | 1 | 1 | | |
| 3 | 2 | 3 | 1 | |
| 4 | 6 | 11 | 6 | 1 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

(These are, in fact, Stirling Numbers of the first kind.)

Further investigation of $A_{Nk}$ can be done if the following generating function (G.F.) is set up

$$A_N(x) = \sum_k A_{Nk} x^k \quad .$$

The recurrence relation (1) can now be converted to a relation on the G.F.

$$A_{N+1}(x) = \sum_k (N A_{Nk} + A_{N,k-1}) x^k$$

$$= N \sum_k A_{Nk} x^k + \sum_k A_{N,k-1} x^k$$

$$= N \sum_k A_{Nk} x^k + x \sum_j A_{Nj} x^j$$

$$A_{N+1}(x) = (x + N) A_N(x)$$

This relation between G.E.s can be solved as follows

$$A_{N+1}(x) = (x + N)(x + N - 1) A_{N-1}(x)$$

$$\vdots$$

$$A_{N+1}(x) = (x + N)(x + N - 1)(x + N - 2) \ldots (x + 1)$$

Consider now the probabilities where $P_{Nk}$ = probability that a random permutation of N elements has T = k.

$$P_{Nk} = A_{Nk} \Big/ N!$$

$$P_N(x) = \sum_k P_{Nk} x^k = \frac{A_N(x)}{N!} = \frac{(x+N-1)}{N} \cdot \frac{(x+N-2)}{N-1} \cdots \frac{(x+1)}{2.1} \qquad (2)$$

The mean and variance can be derived from the above G.F. by looking at the G.F. whose coefficients are probabilities

$$P(x) = \sum_k P_k x^k$$

where $P_k$ is the probability that something specified is equal to k.

$$P(1) = \sum_k P_k = 1$$

$$P'(x) = \sum_k kP_k x^{k-1}$$

$$\text{mean } (P) = P'(1) = \sum_k kP_k = \text{mean value}$$

$$\text{var } (P) = P''(1) + P'(1) - (P'(1))^2 = \text{variance}$$

for a product of two G.F.'s

$$P(x) = Q(x)\ R(x) \qquad Q(1) = R(1) = 1$$

$$\text{mean } (P) = \text{mean } (Q) + \text{mean } (R)$$

$$\text{var } (P) = \text{var } (Q) + \text{var } (R).$$

Applying these general results to the probabilistic G.F. (2)

$$\text{mean } (P_N) = \text{mean } \left(\frac{x+N-1}{N}\right) + \text{mean } \left(\frac{x+N-2}{N-1}\right) + \ldots + \text{mean } \left(\frac{x+1}{2}\right)$$

$$= \frac{1}{N} + \frac{1}{N-1} + \ldots + \frac{1}{2} = H_N - 1$$

$$\text{var } (P_N) = \text{var } \left(\frac{x+N-1}{N}\right) + \text{var } \left(\frac{x+N-2}{N-1}\right) + \ldots + \text{var } \left(\frac{x+1}{2}\right)$$

$$= \left(\frac{1}{N} - \frac{1}{N^2}\right) + \left(\frac{1}{N-1} - \frac{1}{(N-1)^2}\right) + \ldots + \left(\frac{1}{2} - \frac{1}{2^2}\right) + 0$$

$$= H_N - \left(1 + \frac{1}{2^2} + \ldots \frac{1}{N^2}\right)$$

This example illustrates most of the techniques used in this type of algorithm analysis. In fact, the same recurrence relation occurs in other problems such as reservoir sampling techniques where the problem is to sample a large file; and the number of cycles in a random permutation.

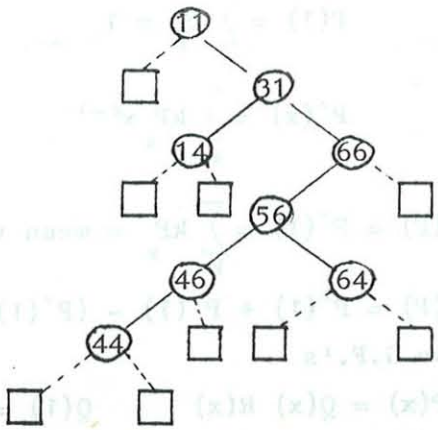
### 2.1.2 Analysis of an Information Retrieval Algorithm

The method of information retrieval using a binary tree is fairly well known (e.g. Hibbard 1962). Given a list of keys the tree is constructed so that first item is the root and a subsequent item is placed on the left if its key is less than the key of the current node, on the right if it is greater. The item is placed at the first unoccupied node.

Example: The following two digit keys were derived randomly using two dice.

11, 31, 66, 14, 56, 46, 64, 44

The binary search tree for these keys, entered in this order, is:




Below every node where there is a vacant place a square box is drawn. If There are n nodes there are n+1 square boxes; this can be seen by induction since it is true for n = 1 and every time a new node is added a square box is replaced by a node and two square boxes.

The search and insertion algorithm simply starts at the root testing the key against the key of the current node. If it is greater go to right, if it is less go to left, if it is the same the search is ended. If an unoccupied node is reached the new key is inserted. An algorithm for this might be as follows, where each node has a key, a left link, and a right link:

```
            key [0]:= - ∞ ;
            q:= 0; p:= right [0];
        while  p≠0 do
        begin  if x = key [p] then goto FOUND;
               q:= p;
               if x < key [q] then p:= left [q]
                              else p:= right [q]
        end;
NOT FOUND:  n:= n+1;  left [n]:= right [n]:= 0;
        if x < key [q] then left [q]:= n else right [q]:= n;
FOUND:
```

The algorithm is to be analysed with respect to the amount of time required to retrieve an item in the binary tree. This is proportional to the distance from the root to the particular node.

For the above tree

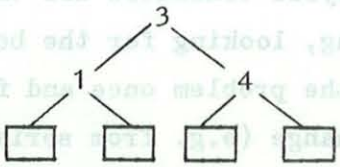
Average number of tests $= \frac{1 + 2 + 3 + 3 + 4 + 5 + 5 + 6}{8}$

This is equivalent to analysing the number of tests required to insert the item in the first place.

Consider a tree of n nodes.

Let $A_{nk}$ = number of permutations of 1 to n where the last element takes k comparisons to insert.

e.g. 3142.




It requires two comparisons to insert the final element 2.

In any binary search tree each square box is an equally likely node for the next item to be inserted, if we assume that the inputs are in random order. When a further node is inserted one square box is replaced by two square boxes at one level further down the tree.

Therefore, the recurrence relation is

$$A_{nk} = (n - 2)A_{n-1,k} + 2A_{n-1,k-1} \qquad (3)$$

It can be seen that this is the correct formula since (n - 2) of the square boxes are the same as in the n - 1 case and the other two have increased by one.

As in the minimum algorithm analysis the recurrence relation (3) is solved using a generating function

$$A_n(x) = \sum_k A_{nk} x^k = (n-2+2x)(n-3+2x) \ldots (0+2x), \quad n \geq 2 .$$

$$\text{mean}\left(\frac{A_n}{n!}\right) = \text{mean}\left(\frac{n-2+2x}{n}\right) + \text{mean}\left(\frac{n-3+2x}{n-1}\right) + \ldots$$

$$\ldots\ldots + \text{mean}\left(\frac{2x}{1}\right)$$

$$= \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots + \frac{2}{2}$$

$$= 2(H_n - 1)$$

Similarly, $\text{var}\left(\frac{A_n}{n!}\right) = 2H_n - 4\left(1 + \frac{1}{2^2} + \ldots + \frac{1}{n^2}\right)$

The variance is an interesting result in this example since it shows the method is fairly stable about the mean. This information is useful since the worst case analysis is not very promising.

Another example which has an instructive analysis is the algorithm for the addition of numbers in base b arithmetic. (See Knuth (1969) Vol.II, pp.242-243).

## 2.2 Type B

A second type of algorithmic analysis considers all the possible algorithms for doing a particular thing, looking for the best. Unfortunately, such analyses do not solve the problem once and for all since one of our fundamental premises may change (e.g. from serial to parallel operation).

### 2.2.1 Internal sorting

It seems reasonable in this case to base the analysis on the number of comparisons made by each method. Since there are n! possible outcomes and each comparison can at most reduce this by a factor of two then the minimum number of comparisons is $\log_2 n!$ for sorting n elements.

However, the relation

$$\min(x,y) = \frac{x + y - |x-y|}{2}$$

needs no formal comparisons. It is possible to sort n numbers without making a single conditional jump instruction in the program. Radix methods do the sorting with no comparisons. So there are difficulties in setting up the problem in such a way that methods can be compared by this 'complexity' analysis.

Another difficulty is finding exact values, and this is also well illustrated by internal sorting.

Consider the classical method of internal sorting known as binary insertion (H. Steinhaus (1950)). If the k-1 elements $x_1 x_2 \ldots x_{k-1}$ have been already sorted then the next element $x_k$ may be inserted in the correct position in the list by the use of binary search.

Number of comparisons to insert $k^{th}$ element $= \lceil \log_2 k \rceil$

$$\text{Total number of comparisons} = \sum_{k=1}^{n} \lceil \log_2 k \rceil = B(n) \qquad (5)$$

Let S(n) be the minimum number of comparisons achievable by any algorithm; then

$$B(n) \geq S(n) \geq \log_2 n! = \sum_{k=1}^{n} \log_2 k \qquad (6)$$

There is, therefore, only a small difference between the theoretical minimum and what can be achieved by binary insertion. However, this analysis is deceiving because the difficulty with binary insertion is that in order to insert the new element $x_k$ elements have to be moved in storage. This is often much more important than the time taken to do comparisons. There are methods that achieve the results in equation (5) without the extra penalty of moving elements, e.g. Merging.

The problem of obtaining the exact minimum value for the number of comparisons is now examined.

Consider n = 5. Using equations (5) and (6)

$$B(5) = 8$$
$$S(5) \geq 7.$$

There is a tricky construction which proves that S(5) = 7, due to Ford and Johnson (1959). They showed, in fact, that $S(n) = \lceil \log_2 n! \rceil$ for $n \leq 11$, $n = 20, 21$. M. Wells (1965) showed $S(12) = \lceil \log_2 12! \rceil + 1$. In general, the method of Ford and Johnson requires

$$F(n) = \sum_{k=1}^{n} \lceil \log_2 \tfrac{3}{4}k \rceil$$

comparisons, and so far this is the best result known.

Summing up this 'complexity' analysis of algorithms,

(i) There are great difficulties in formulating the problem.

(ii) The basic assumptions of the formulation can change and thus invalidate the whole analysis.

(iii) Exact solutions to most of the problems are not known and are difficult to find.

## 3. Instruction Frequency Counts

Detailed theoretical analyses of the types just discussed are only worthwhile if the algorithm is important, which means either frequently used or occupying a central place in computer science. For other algorithms an empirical approach may be adopted.

The idea of this analysis is to provide the algorithm designer with a count of the number of times various parts of his algorithms are passed. In detail this usually means providing him with a count of the number of times each independent statement in the algorithm is passed. This is much more helpful than the practice of many people today which is to supply running times for particular machines. Such times can be very compiler and machine dependent. Ideally, these instruction frequency counts should be supplied automatically by the compiler or software system and also there should be the facility for turning them off when not required. Professor Knuth gave an example of the use of frequency counts in which he had written a compiler and was running it on a simulator to find the bottlenecks. The early tests showed that 45% of the frequency counts were zero. Thus, quite a large part of compiler was untested and could still contain errors. Also the parts where the frequency counts were high showed where the computational effort was being employed and this was used to optimise the compiler. Therefore, both large frequency counts and zero counts provide valuable information. A related debugging method is to trace each instruction the first n times it is obeyed (typically n = 2). This selective trace technique has been used by Knuth for many years with interpreters, and it has recently been developed at Stanford by Satterthwaite for an Algol compiler. The use of instruction frequency counts is an excellent way of training computer scientists to write efficient algorithms. In fact, even experienced programmers are often surprised at the frequency count results. It seems, therefore, that such frequency counts should be a standard feature of compilers and operating systems.

## 4. The Role of Mathematics in Computer Science Teaching

Several sections of mathematics such as numerical analysis, mathematical logic and the theory of languages (including automata theory) have obvious relevance in the teaching of computer science students. However, there are other parts such as Combinatorial and Discrete Mathematics which have disappeared for many years from the mathematical syllabus but have much relevance to Computer Science. The analysis of algorithms has revealed the importance of such topics as permutation properties, counting, recurrence relations and generating functions. Partial ordering and tree structures are also important but not all of

classical graph theory is useful. Another point in the mathematical teaching of computer scientists is that the emphasis should be more on methods and less on theorems.

5. Discussion

Professor Randell asked if the type of empirical frequency count could not be obtained more easily by interrupting the program periodically and examining its position. Professor Knuth replied that this approach is also being fruitfully applied at Stanford and elsewhere. Unfortunately, such an interrupt often occurs in the logarithm routine or the I/O package and so the information derived is not quite so useful. In high level languages it is also difficult to relate the interrupt location to the position in the original program.

In reply to a question by Professor Perlis about getting information on the path followed by a program Professor Knuth said it was his personal opinion that it would not produce the spectacular payoffs that the initial provision of frequency counts had. In fact, he doubted if the extra information would be worth the difficulty of obtaining it.

Professor Knuth agreed with a remark of Professor Wirth that many programmers did not give any thought to the idea of frequency counts when writing their programs and consequently they would be surprised by the frequency counts obtained. However, he had personally been surprised by the frequency counts he had obtained even though he had considered efficiency in writing his own programs.

Professor Randell asked about the size and complexity of the programs on which the frequency count technique had been employed and Professor Knuth stated that programs of up to four to five thousand cards had been tackled.

Professor Michaelson posed the question that since compilers optimise uniformly would not frequency counts suggest that hand optimisation in the areas of the program with large frequency count is a more worthwhile technique. Professor Knuth agreed and said it would appear reasonable to develop a compiler which would optimise over the inner loop of a program. In typical programs the running time is governed by only a small percentage of the code. However, in practice it was found that the inner loops of programs were not infrequently rather large, and this makes the optimisation much more difficult.

Professor Hoare pointed out that the zeros of the frequency counts would appear to be of some advantage with respect to slave store techniques, in which backup storage would be used mainly for the parts of programs that were used very infrequently. Professor Knuth replied that unfortunately the distribution of zeros in the frequency counts was not usually uniform and it would be difficult to use them in the way suggested.

In reply to Professor King the speaker agreed that the probability distribution of the data could greatly affect the performance of an algorithm and therefore should if possible be included in the analysis. He quoted the example of a payroll program which would behave very differently at the end of the month than at any other time. The distribution of the data should be considered in the choice of a particular algorithm.

Professor Perlis asked the speaker how much of the analysis of algorithms he thought should be taught to engineers and physicists. Professor Knuth replied that they should be exposed to a few theoretical analyses, although most of the mathematical analysis would not be appropriate for such students. Frequency counts are a good way of introducing them to the idea of writing efficient programs.

## 6. References


Ford, L.R., Jr. and Johnson, S.M. (1959) 'A Tournament Problem'. Amer. Math. Monthly 66 (May 1959)., pp.387ff.

Hibbard, T.N. (1962) 'Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting'. JACM 9 (1962), pp.13-28.

Knuth, D.E. (1968) 'The Art of Computer Programming' Vol. I 'Fundamental Algorithms'. Addison-Wesley.

Knuth, D.E. (1969) Vol. II 'Semi-numerical Algorithms'. Addison-Wesley.

Steinhaus, H. (1950) 'Mathematical Snapshots'. Oxford University Press. pp.36ff.

Wells, M.B. (1965) 'Applications of a Language for Computing in Combinatorics'. Procs. of IFIP Congress 1965, Vol. 2, pp.497-498.