PROGRAMMING AND ALGORITHMS II

GABRIEL VALIENTE

ALGORITHMS, BIOINFORMATICS, COMPLEXITY AND FORMAL METHODS
RESEARCH GROUP, TECHNICAL UNIVERSITY OF CATALONIA

2023–2024

ANALYSIS OF ITERATIVE ALGORITHMS

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge MA, 4th edition, 2022

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.

You might consider resources such as memory, communication bandwidth, or energy consumption.

Most often, however, you'll want to measure computational time.

If you analyze several candidate algorithms for a problem, you can identify the most efficient one.

There might be more than just one viable candidate, but you can often rule out several inferior algorithms in the process.

Before you can analyze an algorithm, you need a model of the technology it runs on, including the resources of that technology and a way to express their costs.

We assume a generic one-processor, random-access machine (RAM) model of computation as the implementation technology, with the understanding that algorithms are implemented as computer programs.

In the RAM model, instructions execute one after another, with no concurrent operations.

The RAM model assumes that each instruction takes the same amount of time as any other instruction and that each data access (using the value of a variable or storing into a variable) takes the same amount of time as any other data access.

In other words, in the RAM model each instruction or data access takes a constant amount of time (even indexing into an array, assuming that each element of a given array occupies the same number of bytes and that the elements of a given array are stored in contiguous memory locations).

The RAM model contains instructions commonly found in real computers. arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return).

The data types in the RAM model are integer, floating point (for storing real-number approximations), and character.

We assume that each word of data has a limit on the number of bits (similar to the number of bits in a computer word).

For example, when working with inputs of size n, we typically assume that integers are respresented by $c \log_2 n$ bits for some constant $c \geqslant 1$.

We require $c \geqslant 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily.

The RAM model does not account for the memory hierarchy that is common in contemporary computers. It models neither caches nor virtual memory. However, RAM-model analyses are usually excellent predictors of performance on actual machines.

Although it is often straightforward to analyze an algorithm in the RAM model, sometimes it can be quite a challenge. You might need to employ mathematical tools such as the ability to identify the most significant terms in a formula.

Because an algorithm might behave differently for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Instead of timing a run, or even several runs, of an implementation of a given algorithm in a real programming language, we can determine how long it takes by analyzing the algorithm itself.

We'll examine how many times it executes each line of pseudocode and how long each line of pseudocode takes to run.

We'll first come up with a precise but complicated formula for the running time.

Then, we'll distill the important part of the formula using a convenient notation that can help us compare the running times of different algorithms for the same problem.

How do we analyze a given algorithm?

First, let's acknowledge that the running time depends on the input.

Even though the running time can depend on many features of the input, we'll focus on the one that has been shown to have the greatest effect, namely the size of the input, and describe the running time of a program as a function of the size of its input.

The best notion for input size depends of the problem being studied.

For many problems, the most natural measure is the number of items in the input.

For many other problems, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation.

The running time of an algorithm on a particular input is the number of instructions and data accesses executed.

How we account for these costs should be independent of any particular computer, but within the framework of the RAM model.

A constant amount of time is required to execute each line of our pseudocode.

One line might take more or less time than another line, but we'll assume that each execution of the kth line takes c_k time, where c_k is a constant.

For each $i = 2, 3, \ldots, n$, let t_i denote the number of times the while loop test is executed for that value of i.

Recall that the procedure INSERTION-SORT sorts array $A[1:n]$.

```
procedure INSERTION-SORT($A$)
    for $i = 2$ to $n$ do                                          cost $c_1$ times $n$
        $key = A[i]$                                                cost $c_2$ times $n-1$
        insert $A[i]$ into the sorted subarray $A[1:i-1]$          cost 0 times $n-1$
        $j = i - 1$                                                 cost $c_4$ times $n-1$
        while $j > 0$ and $A[j] > key$ do                           cost $c_5$ times $\sum_{i=2}^{n} t_i$
            $A[j+1] = A[j]$                                         cost $c_6$ times $\sum_{i=2}^{n}(t_i - 1)$
            $j = j - 1$                                             cost $c_7$ times $\sum_{i=2}^{n}(t_i - 1)$
        $A[j+1] = key$                                              cost $c_8$ times $n-1$
```

The running time of the algorithm is the sum of the running times for each statement executed.

A statement that takes c_k steps to execute and executes m times contributes $c_k m$ to the total running time.

We usually denote the running time of an algorithm on an input of size n by $T(n)$.

To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of cost and times, obtaining

$$\begin{aligned} T(n) = & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) \\ & + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) \end{aligned}$$

Even for inputs of a given size, the algorithm's running time may depend on which input of that size is given.

For INSERTION-SORT, the best case occurs when the array is already sorted, and the worst case arises when the array is in reverse sorted order.

In the best case, each time the **while** loop executes, the value originally in $A[i]$ is already greater than or equal to all values in $A[1 : i-1]$, so that the **while** loop always exits upon the first test.

Therefore, we have that $t_i = 1$ for $i = 2, 3, \ldots, n$, and the best-case running time is given by

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

We can express this best-case running time as $an + b$ for constants a and b that depend on the statement costs c_k.

The running time is thus a linear function of n.

In the worst case, the procedure must compare each element $A[i]$ with each element in the entire sorted subarray $A[1:i-1]$, and so $t_i = i$ for $i = 2, 3, \ldots, n$. Noting that

$$\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1 = \frac{n(n+1)}{2} - 1 \text{ and } \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

we find that the worst-case running time is given by

$$\begin{aligned} T(n) =& c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ & + c_6\left(\frac{n(n-1)}{2} - 1\right) + c_7\left(\frac{n(n-1)}{2} - 1\right) + c_8(n-1) \\ =& \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ & - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b and c that depend on the statement costs c_k.

The running time is thus a quadratic function of n.

Our analysis of insertion sort looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted.

We'll usually concentrate on finding only the worst-case running time, that is, the longest running time for any input of size n, for three reasons.

First, the worst-case running time of an algorithm gives an upper bound on the running time for any input.

Second, for some algorithms, the worst case occurs fairly often.

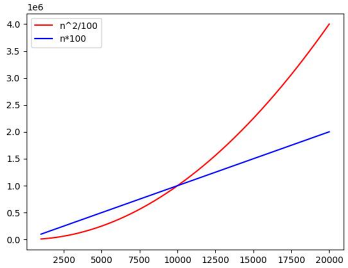
Third, the "average case" is often roughly as bad as the worst case.

Now, we expressed the best-case running time as $an + b$, and the worst-case running time as $an^2 + bn + c$, for constants a, b and c that depend on the statement costs.

Another simplifying assumption is the rate of growth, or order of growth, of the running time, where we consider only the leading term of a formula, since the lower-order terms are relatively insignificant for large values of n, and we also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.

For insertion sort's worst-case running time, when we ignore the lower-order terms and the leading term's constant coefficient, only the factor of n^2 from the leading term remains, and that factor is by far the most important part of the running time.

For example, suppose that an algorithm implemented on a particular maching takes $n^2/100 + 100n + 17$ microseconds on an input of size n. Although the coefficients of $1/100$ for the n^2 term and 100 for the n term differ by four orders of magnitude, the $n^2/100$ term dominates the $100n$ term once n exceeds 10,000.



To highlight the order of growth of the running time, we have a special notation that uses the Greek letter Θ (theta).

We say that insertion sort has a worst-case running time of $\Theta(n^2)$, which means "roughly proportional to n^2 when n is large."

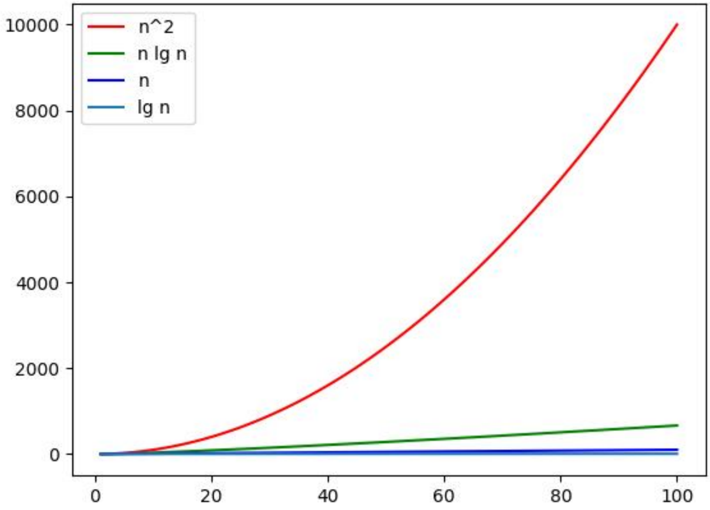
We also say that insertion sort has a best-case running time of $\Theta(n)$, which means "roughly proportional to n when n is large."

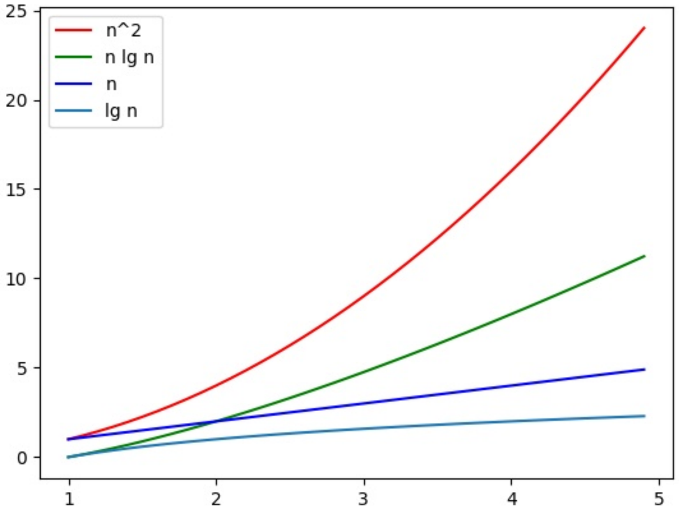
We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth.

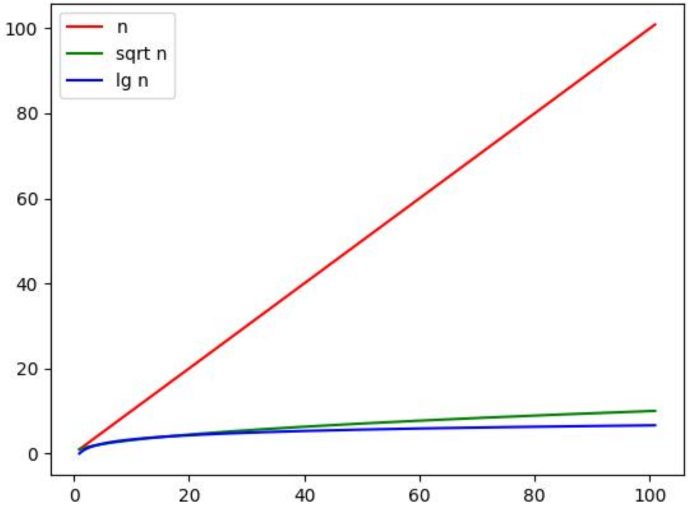
Due to constant factors and lower-order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth.

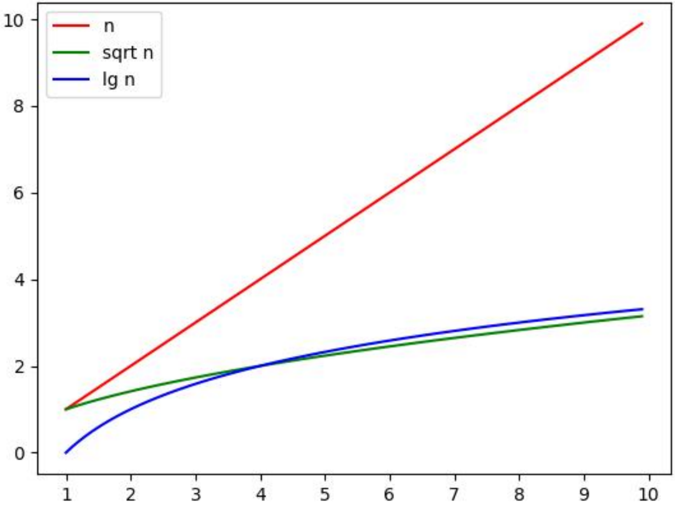
But on large enough inputs, an algorithm whose worst-case running time is $\Theta(n^2)$, for example, takes less time in the worst case than an algorithm whose worst-case running time is $\Theta(n^3)$.

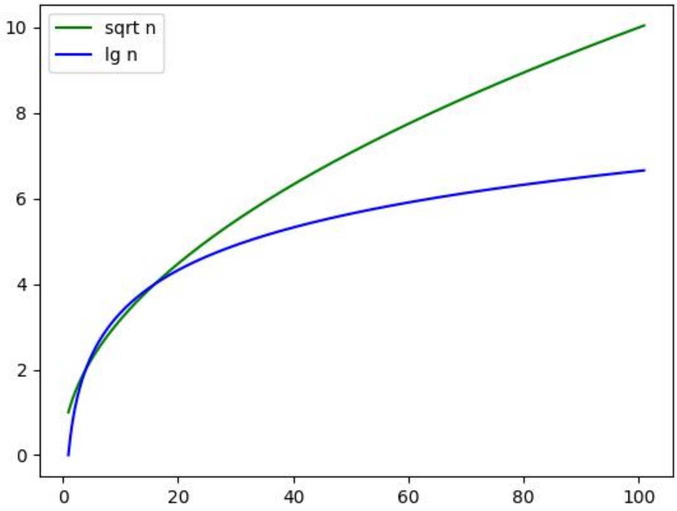
Regardless of the constants hidden by the Θ-notation, there is always some number, say n_0, such that for all input sizes $n \geqslant n_0$, the $\Theta(n^2)$ algorithm beats the $\Theta(n^3)$ algorithm in the worst case.


n^2
n lg n
n
lg n
10000
8000
6000
4000
2000
0
0
20
40
60
80
100


n^2
n lg n
n
lg n
0
5
10
15
20
25
1
2
3
4
5


n
sqrt n
lg n
100
80
60
40
20
0
0
20
40
60
80
100


n
sqrt n
lg n
0
2
4
6
8
10
1
2
3
4
5
6
7
8
9
10


sqrt n
lg n
0
2
4
6
8
10
0
20
40
60
80
100

EXAMPLES

The procedure LINEAR-SEARCH takes an array $A[1:n]$ and a value x.

```
function LINEAR-SEARCH($A, x$)
    $i = 1$
    while $i \leqslant n$ and $x \neq A[i]$ do
        $i = i + 1$
    if $i > n$ then
        return nil
    return $i$
```

In the best case, x appears in the first position of the array. Therefore, the running time of the algorithm is $\Theta(1)$ in the best case.

In the worst case, x appears only in the last position of the array, so that the entire array must be checked. Therefore, the running time of the algorithm is $\Theta(n)$ in the worst case.

The procedure BINARY-SEARCH takes a sorted array $A[1:n]$ and a value x.

```
function BINARY-SEARCH($A, x$)
    return BINARY-SEARCH($A, x, 1, n$)
```

The procedure BINARY-SEARCH takes a sorted array A, a value x, and a range $[low : high]$ of the array, in which we search for the value x.

```
function BINARY-SEARCH($A, x, low, high$)
    while $low \leqslant high$ do
        $mid = \lfloor (low + high)/2 \rfloor$
        if $x = A[mid]$ then
            return $mid$
        else if $x > A[mid]$ then
            $low = mid + 1$
        else
            $high = mid - 1$
    return nil
```

The running time of the algorithm is $\Theta(\lg n)$ in the worst case.

The procedure terminates the search unsuccessfully when the range is empty and terminates it successfully if the value x has been found.

Based on the comparison of x to the middle element in the searched range, the search continues with the range halved.

The recurrence for this procedure is therefore $T(n) = T(n/2) + \Theta(1)$, whose solution is $\Theta(\lg n)$.

We can use mathematical induction to show that when $n \geqslant 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 2 \\ T(n/2) + 1 & \text{if } n > 2 \end{cases}$$

is $T(n) = \lg n$.

The base case is when $n = 2$, and we have $\lg n = \lg 2 = 1$.

For the inductive step, our hypothesis is that $T(n/2) = \lg(n/2)$. Then

$$T(n) = T(n/2) + 1 = \lg(n/2) + 1 = \lg n - \lg 2 + 1 = \lg n - 1 + 1 = \lg n$$

which completes the inductive proof for exact powers of 2.

The procedure BUBBLE-SORT sorts array $A[1:n]$.

```
procedure BUBBLE-SORT($A$)
    for $i = 1$ to $n - 1$ do
        for $j = n$ downto $i + 1$ do
            if $A[j] < A[j-1]$ then
                exchange $A[j]$ with $A[j-1]$
```

The running time depends on the number of iterations of the inner **for** loop.

For a given value of i, this loop makes $n - i$ iterations, and i takes on the values $1, 2, \ldots, n-1$.

The total number of iterations, therefore, is

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Thus, the running time of the algorithm is $\Theta(n^2)$ for all cases.

The procedure SELECTION-SORT sorts array $A[1:n]$.

```
procedure SELECTION-SORT($A$)
    for $i = 1$ to $n - 1$ do
        $smallest = i$
        for $j = i + 1$ to $n$ do
            if $A[j] < A[smallest]$ then
                $smallest = i$
        exchange $A[i]$ with $A[smallest]$
```

The running time depends on the number of iterations of the inner **for** loop.

For a given value of i, this loop makes $n - i$ iterations, and i takes on the values $1, 2, \ldots, n - 1$.

The total number of iterations, therefore, is

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Thus, the running time of the algorithm is $\Theta(n^2)$ for all cases.