Chapter 3

July 28, 2021

1 Experimental Studies

Challenges of Experimental Analysis

There are three major limitations to their use for algorithm analysis:

- Experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments.
- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
- An algorithm must be fully implemented to execute it to study its running time experimentally.

1.1 Moving Beyond Experimental Analysis

Our goal is to develop an approach to analyzing the efficiency of algorithms that:

1. Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
2. Is performed by studying a high-level description of the algorithm without the need for implementation.
3. Takes into account all possible inputs

Primitive Operations

- Formally, a primitive operation corresponds to a low-level instruction with an execution time that is constant. Ideally, this might be the type of basic operation that is executed by the hardware, although many of our primitive operations may be translated to a small number of instructions.
- The implicit assumption in the approach of counting primitive operations as an indication of the running time is that the running times of different primitive operations will be fairly similar. Thus, the number, t, of primitive operations an algorithm performs will be proportional to the actual running time of that algorithm.

Focusing on the Worst-Case Input

Making the standard of success for an algorithm to perform well in the worst case necessarily requires that it will do well on every input.

2 The Seven Functions Used in The Book

2.1 The Constant Function

$$f(n) = c,$$

- It does not matter what the value of n is; f (n) will always be equal to the constant value c.
- As simple as it is, the constant function is useful in algorithm analysis, because it characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to some variable, or comparing two numbers.

2.2 The Logarithm Function

$$x = log_b n \; if \; and \; only \; if \; b^x = n.$$

- The most common base for the logarithm function in computer science is 2, as computers store integers in binary, and because a common operation in many algorithms is to repeatedly divide an input in half.
- In particular, we can easily compute the smallest integer greater than or equal to $log_b n$ (its so-called ceiling, $\lceil log_b n \rceil$). For positive integer, his value is equal to the number of times we can divide n by b before we geta number less than or equal to 1. For example, the evaluation of $log_3 27$ is 3, because $((27/3)/3)/3 = 1$. Likewise, $log_4 64$ is 3, because $((64/4)/4)/4 = 1$, and $log_2 12$ is 4, because $(((12/2)/2)/2)/2 = 0.75 \leq 1$.

Proposition 3.1 (Logarithm Rules): *Given real numbers* $a > 0, b > 1, c > 0$, and $d > 1$, we have:

- $log_b(ac) = log_b a + log_b c$
- $log_b(a/c) = log_b a - log_b c$
- $log_b(a^c) = c \; log_b a$
- $log_b a = \frac{log_d a}{log_d b}$
- $b^{log_d a} = a^{log_d b}$

2.3 The Linear Function

$$f(n) = n.$$

- This function arises in algorithm analysis any time we have to do a single basic operation for each of n elements.
- The linear function also represents the best running time we can hope to achieve for any algorithm that processes each of n objects that are not already in the computer's memory because reading in the n objects already requires n operations.

2.4 The N-Log-N Function

$$f(n) = n \; logn,$$

- This function grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function; therefore, we would greatly prefer an algorithm with a running time that is proportional to $nlogn$, than one with quadratic running time.

2.5 The Quadratic Function

$$f(n) = n^2,$$

- The main reason why the quadratic function appears in the analysis of algorithms is that many algorithms have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. Thus, in such cases, the algorithm performs $n \cdot n = n^2$ operations.
- The quadratic function can also arise in the context of nested loops where the number of operations performed inside the loop increases by one with each iteration of the outer loop.

2.6 The Cubic Function and Other Polynomials

A polynomial function has the form,

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdot \cdot \cdot + a_d n^d,$$

where $a_0, a_1, ..., a_d$ are constants, called the coefficients of the polynomial, and $a_d \neq 0$. Integer d, which indicates the highest power in the polynomial, is called the degree of the polynomial.

2.7 The Exponential Function

$$f(n) = b^n,$$

- As was the case with the logarithm function, the most common base for the exponential function in algorithm analysis is $b = 2$.

Proposition 3.4 (Exponent Rules): *Given positive integers* a, b, *and* c, *we have* :

1. $(b^a)^c = b^{ac}$
2. $b^a b^c = b^{a+c}$
3. $b^a / b^c = b^{a-c}$

- Given a positive integer k, we define $b^{1/k}$ to be k^{th} root of b, that is, the number r such that $r^k = b$. For example, $25^{1/2} = 5$, since $5^2 = 25$. Likewise, $27^{1/3} = 3$ and $16^{1/4} = 2$.
- $b^{a/c}$ is really just the c^{th} root of the integral exponent b^a.
- Given a negative exponent d, we define $b^d = 1/b^{-d}$, which corresponds to applying Exponent Rule 3 with $a = 0$ and $c = -d$. For example, $2^{-3} = 1/2^3 = 1/8$.

Geometric Sums

For any integer n 0 and any real number a such that $a > 0$ and $a \neq 1$, consider the summation:

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdot \cdot \cdot + a^n$$

(remembering that $a_0 = 1$ if $a > 0$). This summation is equal to:

$$\frac{a^{n+1}-1}{a-1}.$$

2.8 Comparing Growth Rates

Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or n-log-n time. Algorithms with quadratic or cubic running times are less practical, and algorithms with exponential running times are infeasible for all but the smallest sized inputs.

2.8.1 The Ceiling and Floor Functions

The analysis of an algorithm may sometimes involve the use of the floor function and ceiling function, which are defined respectively as follows:

- $\lfloor x \rfloor$ = the largest integer less than or equal to x.
- $\lceil x \rceil$ = the smallest integer greater than or equal to x.

3 Asymptotic Analysis

We analyze algorithms using a mathematical notation for functions that disregards constant factors. Namely, we characterize the running times of algorithms by using functions that map the size of the input, n, to values that correspond to the main factor that determines the growth rate in terms of n. This approach reflects that each basic step in a pseudo-code description or a high-level language implementation may correspond to a small number of primitive operations. Thus, we can perform an analysis of an algorithm by estimating the number of primitive operations executed up to a constant factor, rather than getting bogged down in language-specific or hardware-specific analysis of the exact number of operations that execute on the computer.

3.1 The "Big-Oh" Notation

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer n_0 1 such that:

$$f(n) \leq cg(n),\ for\ n \geq n_0.$$

- The Big-Oh notation allows us to say that a function $f(n)$ is "less than or equal her function $g(n)$ up to a constant factor and in the asymptotic sense as n grows toward infinity. This ability comes from the fact that the definition uses "$\leq$" to compare $f(n)$ to a $g(n)$ times a constant, c, for the asymptotic cases when $n \geq n_0$.

- It is considered poor taste to say "$f(n) \leq O(g(n))$," since the Big-Oh already denotes the "less-than-or-equal-to" concept. Likewise, although common, it is not fully correct to say "$f(n) = O(g(n))$," with the usual understanding of the "$=$" relation, because there is no way to make sense of the symmetric statement, "$O(g(n)) = f(n)$". It is best to say,"$f(n)$ *is* $O(g(n))$."

- We can say "$f(n)$ is order of $g(n)$." For the more mathematically $f(n) \in O(g(n))y$, it is also correct to say, "$f(n) \in O(g(n))$," for the Big-Oh notation, technilly speaking, denotes a whole collection of functions.

Characterizing Running Times Using the Big-Oh Notation

The Big-Oh notation is used widely to characterize running times and space bounds in terms of some parameter n, which varies from problem to problem, but is always defined as a chosen measure of the "size" of the problem.

Some Properties of the Big-Oh Notation

The big-Oh notation allows us to ignore constant factors and lower-order terms and focus on the main components of a function that affect its growth.

Proposition 3.9: If $f(n)$ is a polynomial of degree d, that is,

$$f(n) = a_0 + a_1n + \cdot \cdot \cdot + a_dn^d,$$

and $a_d > 0$, then $f(n)$ is $O(n^d)$. Justification:Note that, for n 1, we have 1 n n^2 $\cdot \cdot \cdot$ n^d; hence, $a_0 + a_1n + a_2n^2 + \cdot \cdot \cdot + a_dn^d$ $(|a_0| + |a_1| + |a_2| + \cdot \cdot \cdot + |a_d|)n^d$. We show that $f(n)$ is $O(n^d)$ by defining $c = |a_0| + |a_1| + \cdot \cdot \cdot + |a_d|$ and $n_0 = 1$.

Thus, the highest-degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial.

Characterizing Functions in Simplest Terms

- In general, we should use the big-Oh notation to characterize a function as closely as possible. While it is true that the function $f(n) = 4n^3+3n^2$ is $O(n^5)$ or even $O(n^4)$, it is more accurate to say that $f(n)$ is $O(n^3)$
- It is also considered poor taste to include constant factors and lower-order terms in the big-Oh notation. For example, it is not fashionable to say that the function $2n^2$ is $O(4n^2 + 6n\ logn)$, although this is completely correct. We should strive instead to describe the function in the Big-Oh in simple instead to describe the function in the Big-Oh in simplest terms.
- Indeed, we typically use the names of these functions to refer to the running times of the algorithms they characterize. So, for example, we would say that an algorithm that runs in worst-case time $4n^2 + n\ logn$ is a quadratic-time algorithm since it runs in $O(n2)$ time. Likewise, an algorithm running in at most $5n + 20logn + 4$ would be called a linear-time algorithm.

3.1.1 Big-Omega

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numhat $f(n)$ is $(g(n))$, pronounced " $f(n)$ is big-Omega of $g(n)$," if $g(n)$ is $O(f(n))$, that is, there is a real constant $c > 0$ and an integer constant n_0 1 such that

$$f(n) \geq cg(n),\ for\ n \geq n_0.$$

This definition allows us to say asymptotically that one function is greater than or equal to another, up to a constant factor.

3.1.2 Big-Theta

This notation allows us to say that two functions grow at the same rate, up to constant factors. We say that $f(n)$ is $(g(n))$, pronounced " $f(n)$ is big-Theta of $g(n)$," if $f(n)$ is $O(g(n))$ and $f(n)$

is $(g(n))$, that is, there are real constants $c^1 > 0$ and $c^n > 0$, and an integer constant n0   1 such that

$$c^1 g(n) \leq f(n) \leq c^n g(n), \ for \ n \geq n_0.$$

3.2 Comparative Analysis

An asymptotically slow algorithm is beaten in the long run by an asymptotically faster algorithm, even if the constant factor for the asymptotically faster algorithm is worse. Table 3.3: Maximum size of a problem that can be solved in 1 second, 1 minute, and 1 hour, for various running times, measured in microseconds.

Some Words of Caution

- Even when using the big-Oh notation, we should at least be somewhat mindful of the constant factors and lower-order terms we are "hiding."
- Any algorithm running in $O(n \ logn)$ time (with a reasonable constant factor) should be considered efficient. Even an $O(n^2)$-time function may be fast enough in some contexts, that is, when n is small. But an algorithm running in $O(2^n)$ time should almost never be considered efficient.
- If we must draw a line between efficient and inefficient algorithms, therefore, it is natural to make this distinction be that between those algorithms running in polynomial time and those running in exponential time.

3.3 Examples of Algorithm Analysis

Constant-Time Operations

- The list class maintains, for each list, an instance variable that records the current length of the list. This allows it to immediately report that length, rather than take time to iteratively count each of the elements in the list.
- The jth element of the list can be found, not by iterating through the list one element at a time, but by validating the index, and using it as an offset into the underlying array.

-
  ```
  def pre x average1(S):
      """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
      n = len(S)                          # create new list of n zeros
      A = [0] * n
      for j in range(n):
          total = 0                       # begin computing S[0] + ... + S[j]
          for i in range(j + 1):
              total += S[i]
          A[j] = total / (j+1)            # record the average
      return A
  ```

The statement, A = [0] * n, causes the creation and initialization of a Python list with length n, and with all entries equal to zero. This uses a constant number of primitive operations per element and thus runs in O(n) time.

-
  ```
  def pre x average2(S):
      """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
      n = len(S)                                  # create new list of n zeros
      A = [0] * n
      for j in range(n):
          A[j] = sum(S[0:j+1]) / (j+1)            # record the average
      return A
  ```

  Even though the expression, sum(S[0:j+1]), seems like a single command, it is a function call and an evaluation of that function takes $O(j + 1)$ time in this context. Technically, the computation of the slice, S[0:j+1], also uses $O(j+1)$ time, as it constructs a new list instance for storage.

Three-Way Set Disjointness

Suppose we are given three sequences of numbers, A, B, and C. We will assume that no individual sequence contains duplicate values, but that there may be some numbers that are in two or three of the sequences. The three-way set disjointness problem is to determine if the intersection of the three sequences is empty, namely, that there is no element x such that $x \in A, x \in B$, and $x \in C$.

- A simple algorithm to solve this problem loops through each possible triple of values from the three sets to see if those values are equivalent. If each of the original sets has size n, then the worst-case running time of this function is $O(n^3)$.

```
def disjoint1(A, B, C):
    """Return True if there is no element common to all three lists."""
    for a in A:
        for b in B:
            for c in C:
                if a == b == c:
                    return False        # we found a common value
    return True                         # if we reach this, sets are disjoint
```

- We can improve upon the asymptotic performance with a simple observation. Once inside the body of the loop over B, if selected elements a and b do not match each other, it is a waste of time to iterate through all values of C looking for a matching triple. An improved solution, disjoint2 taking advantage of this observation is shown below, we claim that the worst-case running time for disjoint2 is $O(n^2)$. There are quadratically many pairs (a, b) to consider. However, if A and B are each sets of distinct elements, there can be at most $O(n)$ such pairs with an equal to b. Therefore, the innermost loop, over C, executes at most n times.

```
def disjoint2(A, B, C):
    """Return True if there is no element common to all three lists."""
    for a in A:
        for b in B:
            if a == b:                  # only check C if we found a match from A and B
```

```
            for c in C:
                if a == c             # (and thus a == b == c)
                    return False      # we found a common value
    return True                       # if we reach this, sets are disjoint
```

Element Uniqueness

In the element uniqueness problem, we are given a single sequence S with n elements and asked whether all elements of that collection are distinct from each other.

- Our first solution to this problem uses a straightforward iterative algorithm. The unique1 function, given below, solves the element uniqueness problem by looping through all distinct pairs of indices $j < k$, checking if any of those pairs refer to elements that are equivalent to each other. Which we recognize as the familiar $O(n^2)$

```
def unique1(S):
    """Return True if there are no duplicate elements in sequence S."""
    for j in range(len(S)):
        for k in range(j+1, len(S)):
            if S[j] == S[k]:
                return False          # found duplicate pair
    return True                       # if we reach this, elements were unique
```

- **Using Sorting as a Problem-Solving Tool**: An even better algorithm for the element uniqueness problem is based on using sorting as a problem-solving tool. In this case, by sorting the sequence of elements, we are guaranteed that any duplicate elements will be placed next to each other. Thus, to determine if there are any duplicates, all we need to do is perform a single pass over the sorted sequence, looking for consecutive duplicates. A Python implementation of this algorithm with a running time of $O(n\ logn)$ is as follows:

```
def unique2(S):
    """Return True if there are no duplicate elements in sequence S."""
    temp = sorted(S)                 # create a sorted copy of S
    for j in range(1, len(temp)):
        if S[j-1] == S[j]:
            return False             # found duplicate pair
    return True                      # if we reach this, elements were unique
```

---

4 Simple Justification Techniques

4.1 By Example

Some claims are of the generic form, "There is an element x in a set S that has property P." To justify such a claim, we only need to produce a particular x in S that has property P. Likewise, some hard-to-believe claims are of the generic form, "Every element x in a set S has property P."

To justify that such a claim is false, we only need to produce a particular x from S that does not have property P. Such an instance is called a ***counterexample***. ***

4.2 The "Contra" Attack

This technique involves the use of the negative. The two primary such methods are the use of the ***contrapositive*** and the ***contradiction***.

- **Contrapositive**
  The use of the contrapositive method is like looking through a negative mirror. To justify the statement "if p is true, then q is true," we establish that "if q is not true, then p is not true" instead.

- **Contradiction**
  In applying the justification by contradiction technique, we establish that a statement q is true by first supposing that q is false and then showing that this assumption leads to a contradiction (such as q is false and then showing that this assumption leads to a contradiction ($2 \neq 2$ or $1 > 3$). By reaching such a contradiction, we show that no consistent situation exists with q being false, so q must be true. Of course, to reach this conclusion, we must be sure our situation is consistent before we assume q is false. ***

4.3 Induction and Loop Invariants

Induction

- This technique amounts to showing that, for any particular n 1, there is a finite sequence of implications that starts with something known to be true and ultimately leads to showing that q(n) is true.
- We should remember, however, the concreteness of the inductive technique. It shows that, for any particular n, there is a finite step-by-step sequence of implications that starts with something true and leads to the truth about n. In short, the inductive argument is a template for building a sequence of direct justifications.

Loop Invariants

To prove some statement L about a loop is correct, define L in terms of a series of smaller statements $L_0, L_1, ..., L_k$, where:

1. The ***initial*** claim, L_0, is true before the loop begins.
2. If L_{j-1} is true before iteration j, then L_j will be true after iteration j.
3. The final statement, L_k, implies the desired statement L to be true.