Time Complexity

Raniconduh

Introduction

Time complexity is a useful tool to measure how efficient a computer algorithm is. This is necessary information as computer programmers typically wish to employ the most efficient algorithms to reduce operating costs, running time, electricity usage, etc. While two algorithms may do the same thing at the end of the day, the one which is more efficient will be typically preferred.

Notation

There are three main ways to measure time complexity: Big-O, Big-Omega, and Big-Theta.

Big-Theta is an exact bound on the runtime of an algorithm. It states that the given algorithm will run no slower and no faster than the given bound.

Big-Omega is a lower bound on the runtime of an algorithm. It states that the algorithm will run no faster than the given bound.

Big-O is the last of the three and is an upper bound on the runtime of an algorithm. It states that the algorithm will run no slower than the bound.

Typically, only Big-O is used to express the runtime of an algorithm. Such usage is often inaccurate and it can be difficult to keep track of the three measures. For all intents and purposes, Big-O is treated as a general worst-case bound.

It is important to note that Big-O, Big-Theta, and Big-Omega typically represent bounds on the worst case runtime of an algorithm. The best time case of e.g. a recursive algorithm is typically constant time, since a recursive function always needs a base case. However, it is nonsensical to use this to represent the runtime of the algorithm. Thus, the worst case is considered.

Asymptotic Behavior

Big-O is only concerned with the asymptotic behavior of a function. That is to say that Big-O of a function `f(n)` (also written `O(f(n))`) considers the behavior as `n`, the input to the function, is relatively large.

When finding the runtime complexity of an algorithm, it is typically unnecessary to consider small inputs. Given a small enough list, any search function will perform relatively similarly. It's only at the big cases where the runtime complexities begin to diverge.

Mathematical Description

Here, only Big-O is considered, although it is trivial to come up with the descriptions of Big-Theta and Big-Omega.

The following statement: `f(n)` is `O(g(n))` can be expressed mathematically as such:

`f(n) is O(g(n))` if and only if for all `n > N`, there is some `k` such that `f(n) <= kg(n)`. Choosing appropriate `N` and `k` can allow us to directly prove if some function is Big-O of some other function.

`f(n) is O(g(n))` really means that at some sufficiently large input, `f(n)` is always less than or equal to `g(n)` for the same input.

Function Ranking

By using the above definition of Big-O, some common functions can be ranked such that each function is Big-O of the next function:

`1 < log(n) < n < nlog(n) < n^2 < n^99 < 2^n < 99^n < n!`

In this ranking there are a couple of different types of functions, all of which are important to know. `1` is the simplest and the fastest function. It is known as "constant time". An algorithm that is `O(1)` will always run in the same amount of time, no matter how large its input is.

The logarithmic function, `log(n)` is the next fastest. This is the runtime complexity of e.g. binary search.

`nlog(n)` is known as log-linear and is the next step up from logarithmic. Merge sort is a common example of a log-linear algorithm.

`n^2`, `n^99`, and other powers of `n` are known as polynomial time. It is very easy to develop a polynomial-time algorithm as it commonly arises from nested loops.

`2^n`, `99^n`, etc. are called exponential functions. Typically, an exponential time algorithm is unwanted as it is typically very inefficient for its task. It is very easy to devise a graph-coloring algorithm in exponential time.

The last common time complexity is factorial time, or `n!`. Algorithms running in factorial time are abysmally slow and entirely unwanted. BOGO sort is a good example of a factorial time algorithm.

Algorithm Runtime Complexity

It is clear that the fastest algorithm is typically the most desirable for any task, so it is important to be able to determine the runtime complexity of any given algorithm.

Constant Time

```
def f(n):
    return n + 1
```

The above algorithm is necessarily constant time, since the addition operation will complete in the same amount of time, no matter how large the input is. However, this requires a bit of extra work to write out. It is assumed that function calling and teardown are constant time operations, so they must take time `k` together. Similarly, it is assumed that addition is a constant time operation, so it must take time `a`. In order to determine the runtime complexity of the algorithm, we must sum the time it takes to complete each portion. In this case, the algorithm is `O(k + a)`. Since `k` and `a` are both constants, we can replace them with a single constant `K`. Now, the algorithm is `O(K)`. To determine the final runtime complexity, all constants must be dropped. In this case, the algorithm becomes `O(1)`, or constant time as per above.

```
def f(n):
    for i in 0..5:
        n += 1
```

The above algorithm now includes a for-loop. It is known that call and teardown are constant time operations, taking time `k`. Now the loop must be considered. The loop will iterate `5` times, and add `1` to the input each time. Since addition is a constant-time operation, it takes `5a` time to complete this loop. So, the complexity is `O(k + 5a)`. Since `5` is a constant, it can be combined with `k` and `a` such that we have `O(K)`. Now, dropping the constant leaves `O(1)`, another constant time algorithm.

Linear Time

```
def f(l: list):
    for i in 0..l.length:
        n[i] += 1
```

This algorithm just adds `1` to every element in the input list `l`. It is clear now that this algorithm scales with the length of `l`, which we will call `n`. Now, we must perform the same analysis as before. Call and teardown are `k` time, and addition is `a` time. However, this algorithm iterates over the entire length of the list. Each iteration will take `j` time and there are `n` iterations, so it takes `jn` time to iterate. All together, the function is `O(jn + k + a)`. Once again, constants can be combined and then dropped, so we end up with `O(n + 1)` time. In this case, the time complexity is comprised of two functions. Since `n` grows much faster than `1` (which doesn't grow at all), it is fair to drop the slower term so that the time complexity becomes `O(n)`, or linear time. It is valid to ignore slower terms since, as per above, Big-O is only concerned with asymptotic behavior. For large inputs, `n` is much larger than `1`, so the constant term is insignificant.

Polynomial Time

```
def f(l: list):
    for i in 0..l.length:
        for j in 0..l.length:
            l[i] += 1
```

This algorithm is a convoluted and inefficient way to add `n`, the length of `l`, to every element in `l`. Call and teardown require time `k`. Now, the first loop will iterate `n` times. Inside of it is another loop that iterates another `n` times. Addition requires time `a`, so the inside loop finishes in `an` time. The outer loop will then require `ann` time, or `an^2` time. Now, combining the terms gives `O(k + an^2)`. Dropping the coefficients leaves `O(n^2 + 1)`. Again, the slower terms can be ignored, so the algorithm is `O(n^2)`. Now we can see that this algorithm runs in polynomial time. For what it does, this algorithm is very inefficient and it is trivial to rewrite into a linear time algorithm. This analysis helps to show exactly how slow the algorithm is.

Recursive Algorithms

Recursive functions are described using the recursive relation. It takes the form of a base case (or however base cases are necessary) as well as a recursive function. The following is the mathematical form:

```
T(1) = f(n)
T(n) = bT(d(n)) + h(n)
```

Here, `T(1)` is the base case and requires `f(n)` time. `b` is the branching factor, i.e. how many times the recursion is called during each run. `d(n)` defines how the input is decreased every run, and `h(n)` is the amount of time it takes to complete each run. Now, consider the following algorithm:

```
def f(n):
    if n == 0 or n == 1:
        return 1
    return n * f(n - 1)
```

The algorithm is a simple recursive definition of the factorial function. Now, let's write the recursive relation:

The base cases are given and fairly simple: at `n = 0` and `n = 1`, the function takes only constant time `k` to run, since it is only returning a constant. So, `T(0) = T(1) = k`.

Since `f(n)` is called once inside itself (called as `f(n - 1)`), the branching factor `b` is `1`.

Each subsequent recursive call decreases the input by `1`, as evidenced by `f(n - 1)`, so the decreasing function `d(n)` is `n - 1`.

Finally, the algorithm uses only a multiplication outside of the recursive call, which is a constant time operation in time `m`.

Putting this all together, the recursive relation is as follows:

```
T(0) = k
T(1) = k
T(n) = T(n - 1) + m
```

From here, the runtime complexity can be found by expanding the recursive relation as follows:

```
T(n) = T(n - 1) + m
     = (T(n - 2) + m) + m
     = ((T(n - 3) + m) + m) + m
     = T(n - 3) + 3m
```

Each expansion represents increasing the recursive depth by `1`. Let's call the current recursive depth `r`. We can trivially substitute this in to get a general form of the recursive relation like so:

```
T(n) = T(n - r) + rm
```

To solve for the time complexity, we need to get to the base case. This happens when `n - r = 1`, so `r = n - 1`. Substituting:

```
T(n) = T(1) + m(n - 1)
     = k + mn - m
```

Combining and dropping all constants leaves us with `O(n + 1)`, which is essentially `O(n)` since `n` grows much faster than `1`. This means that this recursive factorial algorithm is linear time.

Logarithmic Time Recursive Algorithms

The analysis from before will apply exactly the same to the following algorithm:

```
def f(l: list, s):
    if l[0] == s: return 0
    middle = floor(l.length / 2)
    if l[middle] == s:
        return middle
    else if l[middle] < s:
        return f(l[middle + 1:], s)
    else if l[middle] > s:
        return f(l[:middle - 1], s)
```

This algorithm is an implementation of binary search. It requires that the input list is sorted low to high and that the element that is being searched for exists in the list. This analysis will only consider this case, as any other case will be considered a bug.

We can now begin to write our recursive relation:

```
T(1) = k
T(n) = T(n / 2) + q
```

Here, the base case is fairly obvious and runs in constant time. Assuming the list splicing is a constant time operation and the if-statements are also, we can call all these times `q`. Since the function will only ever do one recursive call each run, the branching factor is `1`. This can be seen since only one of the if statements will ever run, and we are only considering the worst case. The decreasing function can be seen as `n / 2` since each recursive call will input either the upper or lower half of the list.

Now, we can begin to unroll the relation with unroll level `r`:

```
T(n) = T(n / 2) + q
     = T(n / 4) + q + q
     = T(n / 8) + q + q + q
     = T(n / 2^r) + qr
```

The base case is reached when `n / 2^r = 1`, so `r = log2(n)`. Substituting:

```
T(n) = T(1) + qlog2(n)
     = k + qlog2(n)
```

Using properties of the logarithm we know that `log2(n) = log(n)/log(2)`. Since `1/log(2)` is a constant, it can be pulled out and combined with the other constants so that the algorithm is `O(K + log(n))`. In this case, `log(n)` still grows much faster than the constant `K`, so the algorithm is `O(log(n))`, or logarithmic time. Below is an approximate graph of linear time and logarithmic time.

```
 ^ time
 |     /
 |    /
 |   /   ---/
 |  / --/
 | /-/
 |/
-+-----------> n
 |
```

Since the linear curve is always above the logarithmic curve, a linear time algorithm will take more time to run than a logarithmic time algorithm for a similar sized input. This proves that the binary search algorithm is much faster than linear search for large enough inputs, provided that the conditions are met.

This same analysis can be applied for any recursive algorithm, although the algebra required to solve for the final runtime complexity may differ.

Conclusion

Time complexity is a powerful tool to rate the how efficient an algorithm may be. However, it is important to note that runtime complexity is not the entire picture when it comes to real time. This abstraction totally ignores things like the time and space it takes to call and destroy functions, which is especially important when looking at recursive algorithms. Typically, the algorithm with a faster runtime complexity is the one which should be used, although it is necessary to way the pros and cons of each algorithm. Merge sort, for instance, requires best and worst case time of `nlog(n)`, though it requires creating and destroying many sub-lists, which may take a fair bit of memory. Other sorts such as quick sort may have a worst case performance of `n^2`, though they operate in-place and typically run at around `nlog(n)` speed, much like merge sort.

It is also necessary to note that oftentimes, developing a slow algorithm is very simple while devising a faster algorithm can be much more difficult and, in some cases, may be impossible.

Big-O and time complexity are essential tools to have as a programmer and computer scientist, though sometimes it requires a little extra consideration to determine which algorithm is best suited for a given scenario.