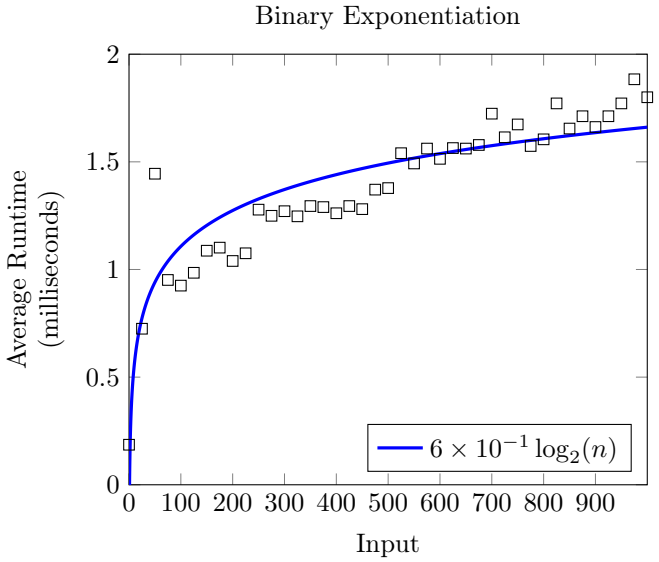
Analysis of Algorithms

Khalid Hourani

March 5, 2017

1 Experimental Studies

One way to study an algorithm's running time is by experiment: we run the algorithm on various inputs and record the time spent on execution. For example, the following implementation of binary exponentiation has the plot:



There are some notable problems with experimental analysis. For one, we can only test certain inputs. It may be the case that an algorithm appears to approach $f(n)$ for "smaller" values of n, but is actually $g(n)$ asymptotically. Moreover, runtime needs to be tested on the same configuration each time. Finally, the algorithm must be fully implemented *and* it must be fully executed each time in order for experimental values to be recorded. This can be prohibitively time-consuming.

2 Primitive Operations

In order to analyze algorithms, we define a set of **primitive operations** that correspond to a set of low-level instructions that run in a fixed amount of time. For example, this may include

- Assigning a value to a variable.
- Calling a function.
- Performing an arithmetic operation (e.g. addition).

Instead of trying to determine the specific execution time of an algorithm, we simply count the number of primitive operations of the algorithm. This then allows us to establish a runtime in terms of the number of primitive operations. We assume that primitive operations will run at approximately the same speed under different configurations.

In some cases, an algorithm will run faster on a specific input. To account for this, we can consider an **average-case** by defining a probability distribution on the set of possible inputs. However, this is often not possible (e.g. there is no uniform distribution on $\mathbb{N}$), so we will typically focus on the **worst case** input. This is generally much simpler than the average case, and has the requirement that in order for an algorithm to perform "well," it must perform so on **every** input.

3 Using Big O Notation

We say a function $f(n)$ is $O(g(n))$, written

$$f(n) \text{ is } O(g(n)) \text{ or}$$
$$f(n) = O(g(n)) \text{ or}$$
$$f(n) \in O(g(n))$$

if and only if there exists a real constant $c > 0$ and an integer constant $m \geq 1$ such that

$$f(n) \leq cg(n) \text{ for all } n \geq m$$

For example, the function $f(n) = n^2 + 4n + 9$ is $O(n^3)$, since

$$n^2 + 4n + 9 \leq 1n^3 \text{ for } n \geq 4$$

Informally, Big O notation can be thought of as describing an upper bound on the performance or memory space of an algorithm. For example, if an algorithm runs in $O(n^2)$ time, then doubling the input will **at most** quadruple the run time. This does not mean that it necessarily will, however. In this sense, it is best to think of Big O notation as a limit on the **worst-case** performance of an algorithm.

Consider, the following function to recursively evaluate x^n for non-negative integers n:

```
int pow(double x, int n)
{
        if (n == 0)
        {
                return 1;
        }
        else
        {
                return x * pow(x, n - 1);
        }
}
```

This function will perform **exactly** n operations, and is therefore $O(n)$. However, this function can be improved by performing what is called "binary exponentiation" instead. We demonstrate this by evaluating x^9: write $9 = 1001_b$ and observe that

$$\begin{aligned} x^9 &= x^{1001_b} \\ &= x^{1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0} \\ &= x^{1\cdot 2^3} x^{0\cdot 2^2} x^{0\cdot 2^1} x^{1\cdot 2^0} \\ &= x^8 x^1 \end{aligned}$$

This creates the far more efficient algorithm:

```
int pow(double x, int n)
{
        int result = 1;
        while (exponent > 0)
        {
                if (exponent % 2 == 1)
                {
                        result *= base;
                }
                exponent /= 2;
                base *= base;
        }
        return result;
}
```

We see that this version of the function will terminate when the exponent becomes 0, which will occur after integer division by 2 has occurred $\lceil \log(n) \rceil$ times. Thus, this version of the function is $O(\log(n))$.

As a rule of thumb, algorithms which are "faster" asymptotically are preferred. So an $O(n)$ algorithm is preferable to an $O(n^2)$ algorithm. This isn't *always* true, however, as an algorithm with an $O(10^{1000}n)$ runtime, while certainly faster **asymptotically** than an $O(n^2)$ algorithm, is hardly preferable. In fact, the $O(10^{1000}n)$ runtime algorithm will not see such results until $n > 10^{1000}$. In other words, it is important to contextualize this bound.