1 Asymptotic Analysis

This section deals with measuring the cost of algorithms. This cost is a function of the size $n$ of the data set on which the algorithm is applied. Size is usually something intuitive, like the number of elements to be sorted or the number of elements in the collection of data to be searched. Probably the only exception you will meet at an introductory level is when the problem deals with single numbers, e.g. an algorithm to multiply two positive integers each between 1 and $N$. Here the size is not $N$, but $\log_2 N$, since that is the number of bits required to represent each integer.

Algorithms usually consist of repeated applications of several types of operations e.g. addition, multiplication, swapping. The cost of an algorithm is usually measured in terms of the number of times the most expensive operation is run. For example, multiplication is more expensive than addition (though this is less true than it was 20 years ago owing to specialized multiplication units in processors) so the number of addition steps is often ignored in algorithms that do multiplications as well.

The other thing to note is that algorithm analysis is usually a worst-case analysis. This is because it’s easy to do, and because the results that come from doing it are usually consistent (since constants are ignored) with the result of average-case analysis. Average-case analysis, when done properly, involves sophisticated probability theory and assumptions on the distribution of input data, when it can be done at all. We will only encounter a few cases where average-case and worst-case analyses produce different results.

This sort of analysis is called asymptotic analysis for good reason – we only care what happens as $n \to \infty$. That’s one reason why constants can be ignored. Suppose you’ve got two algorithms, one which costs $a(n) = 1000n$ and the other which costs $b(n) = 15n \log_n n$. The first algorithm will be more expensive for all $n < 9 \cdot 10^{28}$, but asymptotic analysis would say that the second was better since $a(n) = \Theta(n)$ and $b(n) = \Theta(n \log n)$. Certainly $a$ is better as $n \to \infty$, but whether the problem is going to have a size above $n > 10^{29}$ is another
matter. Especially as the size of problems people want to solve increases as methods to solve them improve.

Despite the caveats, such analysis is still very useful.

1.1 Partitioning the space of functions

Let $\mathcal{F}$ be the set of functions from the non-negative integers $\mathbb{Z}^{\geq 0}$ to the non-negative real numbers $\mathbb{R}^{\geq 0}$. For any $f, g \in \mathcal{F}$, we have

\[ f(n) = O(g(n)) \iff \exists c > 0, N \in \mathbb{Z}^{\geq 0} : f(n) \leq cg(n) \forall n \geq N \]

\[ f(n) = \Omega(g(n)) \iff \exists c > 0, N \in \mathbb{Z}^{\geq 0} : f(n) \geq cg(n) \forall n \geq N \]

\[ f(n) = \Theta(g(n)) \iff (f(n) = O(g(n))) \text{ AND } (f(n) = \Omega(g(n))) \]
\[ \iff \exists c, c' \in \mathbb{R}^{\geq 0}, N \in \mathbb{Z}^{\geq 0} : cg(n) \leq f(n) \leq c'g(n) \forall n \geq N \]

We could say $f = \Theta(g)$ instead of $f(n) = \Theta(g(n))$. The latter is more standard and emphasizes the domains of $f$ and $g$.

We can think of $\Theta$ as a relation on elements of $\mathcal{F}$, and say $f \Theta g$ for $f = \Theta(g)$. The following statements can be easily proved:

\[ (f(n) = \Theta(g(n))) \text{ AND } (g(n) = \Theta(h(n))) \Rightarrow f(n) = \Theta(h(n)) \]

\[ f(n) = \Theta(f(n)) \]

\[ f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n)) \]

In relation notation, these are $f \Theta g$ AND $g \Theta h \Rightarrow f \Theta h$, $f \Theta f$ and $f \Theta g \Rightarrow g \Theta f$. In other words, $\Theta$ is a transitive, reflexive and symmetric — and thus an equivalence — relation on $\mathcal{F}$. It partitions $\mathcal{F}$ into functions of equal asymptotic cost.

$O$ and $\Omega$ can be viewed as relations on $\mathcal{F}$ as well. We concentrate on the former, for which the following statements hold:

\[ f(n) = O(g(n)) \text{ AND } g(n) = O(h(n)) \Rightarrow f(n) = O(h(n)) \]

\[ f(n) = O(f(n)) \]

\[ f(n) = O(g(n)) \text{ AND } g(n) = O(f(n)) \Rightarrow f(n) = \Theta(g(n)) \]

Thus $O$ is transitive, reflexive and antisymmetric relation, i.e. a partial order.
Thus $\Theta$, $O$ and $\Omega$ on functions are akin to $=, \leq$ and $\geq$ when comparing numbers. However, with any two numbers $x, y$, either $x \leq y$ or $x \geq y$ (or both if they’re equal) hold. But there are functions e.g. $f(n) = n, g(n) = n^{1+\frac{\log n}{n}}$, such that neither $f = O(g(n))$ nor $g = O(f(n))$.

1.2 Solving recursive cost descriptions

We often have to determine cost from recursive expressions, e.g.

$$C(n) = 2C\left(\frac{n}{2}\right) + \log n$$

To solve these, i.e. get them in a closed (non-recursive) form, we also need a base case giving the cost of the algorithm for very small problems, e.g. $C(1) = 1$. Such problems are usually solved by repeatedly applying the recursion till a pattern is seen. Whenever logarithms are involved, we take them to some base $b > 1$ and then limit ourselves to problems of size $b^k$. We leave out the rigorous details of showing how such analysis leads to the right answer in the general case. Note that the exact value of $b$ doesn’t matter, since $\log_b n = \log_b d \log_d n = \Theta(\log_d n)$ for any other base $d > 1$.

Here we take $b = 2$, and limit ourselves to the special case $n = 2^k$. Thus the recursion becomes $C(2^k) = 2C(2^{k-1}) + k$ and repeated expansions of it gives

\[
C(n) = C(2^k) \\
= 2C(2^{k-1}) + k \\
= 2(2C(2^{k-2}) + k - 1) + k \\
= 4C(2^{k-2}) + 2(k - 1) + k \\
= 4(2C(2^{k-3}) + k - 2) + 2(k - 1) + k \\
= 8C(2^{k-3}) + 4(k - 2) + 2(k - 1) + k \\
= 8(2C(2^{k-4}) + k - 3) + 4(k - 2) + 2(k - 1) + k \\
= 16C(2^{k-4}) + 8(k - 3) + 4(k - 2) + 2(k - 1) + k \\
= \ldots \\
= 2^k C(2^{k-k}) + 2^{k-1}(1) + 2^{k-2}(2) + \ldots + 2^3(k - 3) + 2^2(k - 2) + 2^1(k - 1) + 2^0(k - 0) \\
= 2^k C(2^0) + 2^{k-1}(1) + 2^{k-2}(2) + \ldots + 2^3(k - 3) + 2^2(k - 2) + 2^1(k - 1) + 2^0(k - 0) \\
= 2^k C(1) + (2^{k-1} + 2^{k-2} + \ldots 1 + 2^0) + (2^{k-2} + \ldots 2^1 + 2^0) + \ldots + (2^1 + 2^0) + (2^0) \\
= 2^k \cdot 1 + (2^k - 1) + (2^{k-1} - 1) + \ldots + (2^2 - 1) + (2^1 - 1) \\
= 2^k + (2^k + 2^{k-1} + \ldots 2^2 + 2^1) - (1 \cdot k) \\
= 2^k + (2^{k+1} - 2) - k \\
= 2^k + 2 \cdot 2^k - 2 - k \\
= 3 \cdot 2^k - (k + 2) \\
= 3n - \log_2 n - 2
Note how we get the final answer back in terms of $n$ at the end.

Here's another example. This time there are two variables involved — the number $N$ of elements to be sorted and a threshold $Q$.

\[ C(n) = \begin{cases} 
2C\left(\frac{n}{2}\right) + cn & \text{iff } n > Q \\
\frac{d n^2}{2} & \text{iff } n \leq Q
\end{cases} \]

To make the analysis easier, suppose we wish to find $C(N)$ for $N = 2^k Q$. Then

\[
C(N) = 2C\left(\frac{N}{2}\right) + cN \\
= 2 \left(2C\left(\frac{N}{4}\right) + c\frac{N}{2}\right) + cN \\
= 4C\left(\frac{N}{4}\right) + cN + cN \\
= 4 \left(2C\left(\frac{N}{8}\right) + c\frac{N}{4}\right) + cN + cN \\
= 8C\left(\frac{N}{8}\right) + cN + cN + cN \\
= \cdots = 2^k C\left(\frac{N}{2^k}\right) + k \cdot cN \\
= 2^k C(Q) + kcN \\
= 2^k dQ^2 + kcN \\
= \frac{N}{Q} Q^2 + \log \frac{N}{Q} \cdot cN \\
= dQN + cN \log \frac{N}{Q}
\]

Some of the last steps make use of algebraic manipulation of $N = 2^k Q$, such as $k = \log_2 \frac{N}{Q}$.

We can only simplify $dQN + cN \log \frac{N}{Q}$ further if we can be sure that one term will eventually become bigger than the other. But since we have no more information about $Q$, we cannot do this. For example, if $Q = c$ (c a constant), then the first and second terms would be $\Theta(N)$ and $\Theta(N \log N)$ respectively, but if $Q = c' N$ ($c' < 1$ also a constant) then they would be $\Theta(N^2)$ and $\Theta(N)$ respectively.
1.2.1 A common mistake

Consider the heapsort algorithm for sorting an array \( X[1...n] \) of integers in ascending or, more accurately, non-descending order, i.e.

\[
i < j \Rightarrow X[i] \neq X[j]
\]

More generally, we can sort objects of any type with an ordering relation \(<\).

Recall that it costs \( \Theta(\log n) \) to insert/remove an element to/from a heap with \( n \) elements. This gives rise to the following algorithm:

```plaintext
HeapSort (X)

n = size of X
create empty max-heap H
for i = 1 to n
  H.add (X[i])
for i = 1 to n
  X[i] = H.pop()
```

In a max-heap, the relation \( x < y \) holds for every node \( x \) and any of its ancestors (\( x \) is not an ancestor of itself, by definition). We could just as well require that the defining relation be \( y < x \) instead, in which case the structure is called a min-heap, and the second for loop becomes for \( i = n \) to 1 instead. For a fixed type of heap, changing the for loop determines whether the sorting is in ascending or descending order.

Now to figure out \( H(n) \), the cost of HeapSort ing an array with \( n \) elements. To do this, we need to know how much the add and pop operations cost. We are just told that they cost \( \Theta(\log n) \) each; we know nothing about the constants or smaller terms involved. However, we only need our final answer to be in \( \Theta \) form, so we can get away with hypothesizing that add costs \( c \log n \) and pop costs \( d \log n \). We can actually get away with more, assuming that \( c = d = 1 \), but we’ll leave them in for illustrative purposes.

The HeapSort algorithm above has two for loops, the first calling \( n \) times an operation that costs \( c \log n \) and the second calling \( n \) times an operation that costs \( d \log n \). Thus the total cost is \( H(n) = n \cdot c \log n + n \cdot d \log n = (c + d)n \log n \), i.e. \( H(n) = \Theta(n \log n) \).

This is wrong.

It’s the right answer, but the method is wrong. The problem is that we’re mixing up the \( n \)’s. When used in \( H(n) \), \( n \) describes the size of the array to be sorted. When used to describe the costs of add and pop, it refers to the size of the heap at the time those operations are called.
Let’s change notation and say that the cost of `add` and `pop` are both \( \log k \), where \( k \) is the size of the heap. Then the cost of the first `for` loop is \( c \log 1 + c \log 2 + c \ldots + c \log n = c \log(n!) \) and of the second is \( d \log n + d \log(n - 1) + \ldots + d \log 1 = d \log(n!) \). It turns out that \( \log(n!) = \Theta(n \log n) \), but only after making use of Stirling’s inequality, a non-trivial piece of math that you can look up elsewhere.