3. RECURSIVE ALGORITHMS AND RECURRENCE RELATIONS

In discussing the example of finding the determinant of a matrix an algorithm was outlined that defined det(M) for an nxn matrix in terms of the determinants of n matrices of size \((n-1)\times(n-1)\).

If \(D(n)\) is the work required to evaluate the determinant of an nxn matrix using this method then

\[
D(n) = n \cdot D(n-1)
\]

To solve this -- in the sense of ending up with an expression for \(D(n)\) that does not have reference to other occurrences of the function \(D()\) on the right hand side -- we can use *progressive substitutions* as follows:

\[
D(n) = n \cdot D(n-1)
\]

\[
= n \cdot (n-1) \cdot D(n-2)
\]

\[
= n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 \cdot D(1)
\]

\[
\text{a constant, the cost of returning a number}
\]

\[
\Rightarrow D(n) \in O(n!)
\]
Now consider

ALGORITHM Factorial(n)
// Recursively calculates n! for positive integer n
if n=1
    return 1
else
    return n*Factorial(n-1)

The only possible choice for the elementary operation here is multiplication, so we choose as a cost function

\[ F(n) = \text{number of multiplications to evaluate Factorial(n)} \]

The algorithm as defined then implies that

\[
\begin{align*}
F(1) &= 0 \\
F(n) &= F(n-1) + 1, \ n > 1
\end{align*}
\]

The general-n form can be rewritten as

\[ F(n) - F(n-1) = 1 \]

Letting \( n \to n-1 \) in the above, we also have

\[ F(n-1) - F(n-2) = 1 \]

We can continue to do this until the row beginning with \( F(2) \), which contains the first mention of the base case \( n-1 = 2-1 = 1 \).

\[
\begin{align*}
F(n) - F(n-1) &= 1 \\
+ F(n-1) - F(n-2) &= 1 \\
+ F(n-2) - F(n-3) &= 1 \\
& \quad \vdots \quad \vdots \quad \vdots \\
+ F(2) - F(1) &= 1 \\
F(n) - F(1) &= n-1
\end{align*}
\]

('Method of differences' or 'ladder method'.)
Adding up the rungs of the ladder there are cancellations on the left hand side for every term except \( F(n) \) and \( F(1) \), and for the right hand side the sum is just \( n-1 \).

\[
\begin{align*}
\rightarrow F(n) &= F(1) + n - 1 \\
\rightarrow F(n) &= n - 1, \text{ since } F(1) = 0 \\
\rightarrow F(n) &\in O(n).
\end{align*}
\]

*Aside:*

Is this reasonable, is evaluating \( n! \) really only in \( O(n) \)?

The analysis ignored the fact that large values will build up quickly and so it was not in this case ideal to count integer multiplications as elementary. However it was a simple example that showed the techniques that can also be used in much more complex cases.

* * *

Expressions like ‘\( D(n) = n \, D(n-1) \)’

‘\( F(n) = F(n-1) +1 \)’

are known as **recurrence relations**. A recurrence relation is a formula that allows us to compute the members of a sequence one after the other, given one or more starting values.

These examples are **first order recurrence relations** because a reference is made to only one smaller sized instance.

A first order recurrence requires only one ‘starting value’ -- in the first of these cases \( D(1) \), in the second \( F(1) \) -- in order to obtain a unique solution for general \( n \).

Recurrence relations arise naturally in the analysis of recursive algorithms, where the starting values are the work required to compute **base cases** of the algorithm.
Consider as another example of a first order recurrence relation

\[ f(0) = 1 \quad \text{(the base case is for } n=0) \]
\[ f(n) = c \cdot f(n-1) \quad n > 0 \]

(It doesn't matter here what hypothetical algorithm may have generated the definition of \( f(n) \). Most of the examples in this section will be starting directly from a recurrence relation and will be focused primarily on developing the mathematical tools for solving such relations.)

By inspection,

\[
\begin{align*}
  f(n) &= c \cdot f(n-1) \\
  &= c^2 \cdot f(n-2) \\
  &\quad \ldots \\
  &= c^n \cdot f(0) = c^n
\end{align*}
\]

\( f(n)=c^n \) is the solution of this recurrence relation.

It is purely a function of the input size variable \( n \), it does not make reference on the right hand side to the original cost function applied to any other input sizes.

The expression

\[ D(n)= n \cdot D(n-1) \]

is clearly also a first order recurrence of the same kind, but here the multiplying coefficient depends on \( n \). This is the next step up in complication, where for a general \( n \)-dependent multiplier \( b(n) \)

\[ f(n) = b(n)xf(n-1) \quad (1) \]

(defined for \( n > a \) with base case value \( f(a) \) given)

Again by inspection

\[
\begin{align*}
  f(n) &= b(n)xf(n-1) \\
  &= b(n)xb(n-1)xf(n-2) \\
  &= b(n)xb(n-1)x\ldots xb(a+1)xf(a)
\end{align*}
\]
(note it was not possible to go any further because there would then be a reference to \( f(a-1) \), which is not defined)

giving as a general solution to (1)

\[
f(n) = \left\{ \prod_{i=a+1}^{n} b(i) \right\} x f(a)
\]

in the determinant example

\( a=1, \; f(1)=1, \; b(i)=i \), giving \( f(n) = \prod_{i=a+1}^{n} b(i) = 2x3x...xn = n! \)

(1) is the most general case of a **first order** (it makes reference to only one smaller sized instance), **homogeneous** (there is nothing on the r.h.s. that doesn't multiply a smaller sized instance, such as a polynomial in \( n \)) recurrence relation.

What about the factorial function recurrence

\[
F(n) = F(n-1) + 1
\]

This is a simple example of a **first order, inhomogeneous** recurrence relation, for which the general form (with non-constant coefficients \( b(n), c(n) \)) is

\[
f(n) = b(n)xf(n-1) + c(n) \quad (n > a, \; f(a) \text{ given})
\]

(2)

If we try to do this one by progressive substitution as for the determinant case we quickly get into difficulties:

\[
f(n) = b(n)xb(n-1)x f(n-2) + b(n)xc(n-1) + c(n)
\]

\[
= b(n)xb(n-1)xb(n-2)xf(n-3) + b(n)xb(n-1)xc(n-2) + b(n)xc(n-1)+ c(n)
\]

\[
= ............
\]
In cases like this we need to define instead a new function \( g(n) \) by

\[
f(n) = b(a+1)x b(a+2)x \ldots x b(n)x g(n), \ n > a
\]

\[
f(a) = g(a) \quad \text{(the two functions agree on the base case)}
\]

Substituting in this way for \( f(n) \) in (2):

\[
f(n) = b(n)x f(n-1) + c(n)
\]

\[
b(a+1)x b(a+2)x \ldots x b(n)x g(n) = b(n)x b(a+1)x b(a+2)x \ldots x b(n-1)x g(n-1) + c(n)
\]

Then dividing by the product \( b(a+1)x b(a+2)x \ldots x b(n) \), the common multiplier of both instances of the function \( g() \), gives

\[
g(n) = g(n-1) + d(n)
\]

(3) where

\[
d(n) = \frac{c(n)}{b(a+1)x b(a+2)x \ldots x b(n)}.
\]

(3) is now in a much simpler form (the important thing is that the multiplier of the smaller-input instance of the function on the r.h.s. is now 1) and can be solved by the ladder method we used in the case of \( F(n) \) (where \( d(n) = 1 \), independent of \( n \)) to give the solution

\[
f(n) = \prod_{i=a+1}^{n} b(i) \left\{ f(a) + \sum_{j=a+1}^{n} d(j) \right\}
\]

in the factorial example \( a=1 \), \( f(1)=0 \), \( b(i)=1 \), \( d(j) = 1 \), giving \( f(n) = n-1 \) (as \( \sum_{j=2}^{n} d(j) = \sum_{j=2}^{n} 1 = n - 1 \))
It's possible, and not incorrect, to memorise and quote the above formula and substitute into it in order to solve a given recurrence relation, as the formula can be applied to inhomogeneous first order recurrences of any degree of complexity.

However it's more instructive, and often easier, to consider inhomogeneous recurrences on a case-by-case basis, noting that in the examples you would be given the base case input ‘a’ will normally be 0 or 1, and the d-summation above -- which could in principle be hard and require an approximation technique -- will use just the series summation formulae you have already seen.

**Example:** \( f(0)=0 \) \hspace{1cm} \text{(base case a=0)}

\[ f(n)= 3f(n-1) + 1, \hspace{0.5cm} n > 0 \]

Change variable:

\[ f(n) = 3^n g(n) \quad \text{ (b(i) = 3 for all i, so } \prod_{i=1}^{n} b(i) = 3^n \, \text{ )} \]

This gives

\[ 3^n g(n) = 3 \cdot 3^{n-1} g(n-1) + 1 \]

\[ \rightarrow g(n) = g(n-1) + \frac{1}{3^n} \]

Now use the method of differences (ladder method) to obtain

\[ g(n) = \sum_{i=1}^{n} \frac{1}{3^i} \]
In detail: 

\[ g(n) - g(n-1) = \frac{1}{3^n} \]
\[ g(n-1) - g(n-2) = \frac{1}{3^{n-1}} \]

... ...

\[ g(1) - g(0) = \frac{1}{3^1} \]

\[ g(n) - g(0) = \sum_{i=1}^{n} \frac{1}{3^i} \]

(When you are familiar with these techniques you don’t need to show all the details.)

Multiplying both sides of the solution for \( g(n) \) by \( 3^n \) gives the solution for \( f(n) \)

\[ f(n) = 3^n \sum_{i=1}^{n} \frac{1}{3^i} \]

We evaluate the sum using the formula for a geometric series:

\[ \sum_{i=1}^{n} \frac{1}{3^i} = \left[ \sum_{i=1}^{n} a \right]_{a=\frac{1}{3}} \]
\[ = \left[ \frac{a(1 - a^n)}{1 - a} \right]_{a=\frac{1}{3}} \]
\[ = \frac{1}{2} \left( 1 - \frac{1}{3^n} \right) \]

Hence

\[ f(n) = \frac{3^n}{2} \left( 1 - \frac{1}{3^n} \right) = \frac{1}{2} \left( 3^n - 1 \right) \in O(3^n) \]
SOLVING HIGHER ORDER RECURRENCES USING THE CHARACTERISTIC EQUATION

This is a technique which can be applied to linear recurrences of arbitrary order but is less general than some of the techniques illustrated for first order recurrences in that we will henceforth assume that

- anything multiplying an instance (for some input size) of the function is a constant
- the recurrence relations are homogeneous -- they don’t contain any term that isn’t a multiple of an instance of the function we are trying to obtain a 'closed' solution for

Hence, we are here trying to solve \textbf{kth order recurrences} of the general form

\[ a_k f(n) + a_{k-1} f(n-1) + \ldots + a_0 f(n-k) = 0 \]

where the \( \{a_i\} \) are constants.

(Putting everything on the l.h.s. and equating it to zero rather than writing 'f(n) = ...' is a convenience, as will be seen later.)

Try the solution \( f(n) = \alpha^n \) (see Brassard and Bratley p.65 for a justification of this choice):

\[ a_k \alpha^n + a_{k-1} \alpha^{n-1} + \ldots + a_0 \alpha^{n-k} = 0 \]

\( \alpha^{n-k} \) can be seen to be a common factor:

\[ \alpha^{n-k} (a_k \alpha^k + a_{k-1} \alpha^{k-1} + \ldots + a_0) = 0 \]

\[ \text{characteristic equation} \]

We assume that we are not interested in the trivial solution \( \alpha=0 \), as this would imply \( f(n)=0 \) -- in an algorithmic context that the work needed for an input of any size \( n \) was zero -- only in the non-trivial solutions of the characteristic equation above.
The characteristic equation is a polynomial of degree \( n \) and would in general be expected to have \( k \) distinct roots (the case that some of these may not be distinct will be considered later).

Assuming that the \( k \) roots \( \rho_1, \rho_2, \ldots, \rho_k \) of the characteristic equation are all distinct, any linear combination of \( \{\rho_i^n\} \)

\[
f(n) = \sum_{i=1}^{k} c_i \rho_i^n
\]

(Brassard and Bratley p.65) solves the homogeneous recurrence, where the \( \{c_i\} \) are constants determined by the \( k \) initial conditions that define the base cases of the recurrence.

**Example (k=2):**

\[
f(0) = 0 \\
f(1) = 1 \\
f(n) = 3f(n-1) - 2f(n-2), \quad n > 1
\]

Try a solution of the form \( f(n) = \alpha^n \):

\[
\alpha^n - 3\alpha^{n-1} + 2\alpha^{n-2} = 0
\]

Divide by \( \alpha^{n-2} \) to get the characteristic equation

\[
\alpha^2 - 3\alpha + 2 = 0 \\
\Rightarrow (\alpha - 1)(\alpha - 2) = 0
\]

The roots \( \rho_1 = 1, \rho_2 = 2 \) give a general solution of the form

\[
f(n) = c_1 \cdot 1^n + c_2 \cdot 2^n
\]

Using the initial conditions:

\[
\begin{align*}
c_1 + c_2 &= 0 \quad (n=0) \\
c_1 + 2c_2 &= 1 \quad (n=1)
\end{align*}
\]

\[
\Rightarrow c_1 = -1, \quad c_2 = 1
\]

\[
f(n) = 2^n - 1
\]
If the characteristic equation has a root \( \rho \) with multiplicity \( m \), then (Brassard and Bratley p.67)

\[ x_n = \rho^n, \ x_n = n \ \rho^n, \ ..., \ x_n = n^{m-1} \rho^n \]

are all possible solutions of the recurrence.

In this case the general solution is a linear combination (with \( k \) constant coefficients to be determined by the initial conditions) of these terms together with those contributed by the other roots of the characteristic equation.

**Example (k=2):**

\[
\begin{align*}
f(0) &= 1 \\
f(1) &= 5 \\
f(n) &= 2f(n-1) - f(n-2), \quad n > 1
\end{align*}
\]

The characteristic equation is

\[
\alpha^2 - 2\alpha + 1 = 0 \\
\rightarrow \quad (\alpha - 1)^2 = 0
\]

The repeated root \( \rho = 1 \) gives a general solution of the form

\[ f(n) = c_1 \cdot 1^n + c_2 \cdot n \cdot 1^n \]

Using the initial conditions:

\[
\begin{align*}
c_1 &= 1 \quad \text{(n=0)} \\
c_1 + c_2 &= 5 \quad \text{(n=1)}
\end{align*}
\]

\[ \rightarrow c_1 = 1, \ c_2 = 4 \]

\[ \rightarrow f(n) = 4n + 1 \]
CHANGE OF VARIABLE

Consider the problem of raising a number, \( a \), to some power \( n \). The naïve algorithm for doing this is

**ALGORITHM Exp1(a,n)**

// Computes \( a^n \) for positive integer \( n \)

```python
if n=1
    return a
else
    return a*Exp1(a,n-1)
```

Exp1 has the same general structure as the factorial function we looked at earlier and by a similar argument can be demonstrated also to be \( O(n) \) (counting multiplications at unit cost).

Can we improve on this simple exponentiation algorithm?

Consider the following sequence for computing \( a^{32} \):

\[
a \rightarrow a^2 \rightarrow a^4 \rightarrow a^8 \rightarrow a^{16} \rightarrow a^{32}
\]

Each term is obtained by squaring the previous one, so only 5 multiplications are required, not 31.

It's possible to base an alternate algorithm on this successive squaring idea:
ALGORITHM Exp2(a,n)
// Computes $a^n$ for positive integer n
if n=1
    return a
else
    if even(n)
        return $[\text{Exp2}(a,n/2)]^2$
    else
        return $a*[\text{Exp2}(a,(n-1)/2)]^2$
    (divisions in each case give integer result)

Exp2(a,n) gets broken into smaller (Exp2) instances and then squared to give the required result.

Let $E(n)$ be the work required to compute Exp2(a,n) and use multiplication as the unit-cost elementary operation:

$E(1) = 0$  

$squaring$ $of$ $smaller$-$input$ $instance$

$E(n) = \begin{cases} 
  E(n/2) + 1 & \text{if n even} \\
  E((n-1)/2) + 2 & \text{if n odd}
\end{cases}$

$squaring$ $of$ $smaller$-$input$ $instance$ $and$ 
multiplication $of$ $result$ $by$ $a$

Suppose for the moment that n is not only even but is a power of 2. Make the change of variable

\[ n \rightarrow 2^k \]
Then the even-\(n\) recurrence takes the form

\[
E(2^k) = E(2^{k/2}) + 1
= E(2^{k-1}) + 1
\]

or more compactly, writing \(E(2^k) = E_k\)

\[
E_k - E_{k-1} = 1
\]

This can now easily be solved by the ladder method:

\[
\begin{align*}
E_k - E_{k-1} &= 1 \\
+ E_{k-1} - E_{k-2} &= 1 \\
+ & \ldots \quad \ldots \quad \ldots \\
+ E_1 - E_0 &= 1 \\
E_k - E_0 &= k
\end{align*}
\]

So

\[
E_k = E_0 + k
\]

where \(E_0 = E(2^0) = E(1) = 0\)

\[
\rightarrow E(2^k) = k
\]

Since

\[
n = 2^k \iff k = \log_2 n
\]

\[
E(n) = \log_2 n
\]

Thus \(E(n) \in \mathcal{O}(\log n \mid n \text{ is a power of } 2)\) (remember that bases of logs can be dropped under ‘\(\mathcal{O}\)’)

conditional asymptotic notation
It can be shown (Brassard and Bratley, p.46) that

\[ f(n) \in O\left( g(n) \mid n \text{ is a power of } 2 \right) \Rightarrow f(n) \in O(g(n)) \]

if

(a) \( g \) is eventually non-decreasing
(b) \( g(2n) \in O\left( g(n) \right) \) -- 'smoothness property'

\( \log n \) is an increasing function for all \( n \), and since

\[ \log(2n) = \log 2 + \log n \in O(\log n) \]

we can say finally that \textbf{for all} \( n \)

\[ E(n) \in O(\log n) \]

Exp2 is thus significantly more efficient, as \( n \) grows, than the naïve algorithm Exp1 -- though it is always advisable to check that the computational overheads in implementing a more sophisticated algorithm don’t outweigh the benefits for the size of input instances you are likely to encounter.

\textbf{Example :}

\[ f(1) = 1 \]
\[ f(n) = 4 f(n/2) + n^2, \quad n > 1 \]

Change variables \( n \rightarrow 2^k \) to obtain the recurrence

\[ f_k = 4f_{k-1} + 4^k \]

Make the further substitution (because of the multiplier ‘4’ in front of \( f_{k-1} \)):

\[ f_k = 4^k g_k, \quad f_0 = g_0 \]

\[ 4^k g_k = 4 \cdot 4^{k-1} g_{k-1} + 4^k \]
Dividing by $4^k$ and constructing the ladder

\[
g_k - g_{k-1} = 1 \\
+ g_{k-1} - g_{k-2} = 1 \\
+ \ldots \quad \ldots \quad \ldots \\
+ g_1 - g_0 = 1 \\
\hline
\quad g_k - g_0 = k
\]

\[\rightarrow g_k = g_0 + k \quad \text{(where } k \text{ and } n \text{ are related by } n = 2^k, \ k = \log_2 n)\]

Multiplying the solution for $g$ by $4^k$:

\[f_k = 4^k f_0 + 4^k k\]

\[\rightarrow f(n) = n^2 f(1) + n^2 \log_2 n \quad (f_0 = f(2^0) = f(1); \ 4^k = (2^k)^2 = n^2)\]

\[f(n) \in O(n^2 \log n \mid n \text{ is a power of } 2)\]

Since $n^2 \log n$ is a non-decreasing function for all $n > 1$ (it has a turning point between 0 and 1), and

\[(2n)^2 \log(2n) = 4 n^2(\log2) + 4n^2(\log n) \in O(n^3 \log n)\]

it can be concluded that for all $n$

\[f(n) \in O(n^2 \log n)\]
**Example:**

\[
\begin{align*}
  f(1) &= 1 \\
  f(2) &= 2 \\
  f(n) &= 4f(n/2) - 4f(n/4), \quad n > 2
\end{align*}
\]

Change variables \( n \rightarrow 2^k \) (noting that as \( n/2 \rightarrow 2^{k-1}, n/4 \rightarrow 2^{k-2} \)) to obtain the recurrence for the new variable \( k \)

\[
f_k = 4 f_{k-1} - 4 f_{k-2}
\]

This is a **homogeneous 2nd order** recurrence, and should be solved using the **characteristic equation method**.

The characteristic equation for the above recurrence (variable \( k \)) is

\[
\alpha^2 - 4\alpha + 4 = 0 \\
\rightarrow (\alpha - 2)^2 = 0
\]

Repeated root \( \rho = 2 \), so the general solution for \( k \) is

\[
f_k = c_1 2^k + c_2 k 2^k
\]

In terms of the original variable \( n \), this general solution is

\[
f(n) = c_1 n + c_2 n \log_2 n
\]

Is this \( O(n \log n) \)? It depends on the base cases, which determine \( c_1 \) and \( c_2 \).

\[
\begin{align*}
  f(1) &= c_1 + c_2 \log_2 1 = c_1 = 1 \\
  f(2) &= 2c_1 + 2 c_2 \log_2 2 = 2(c_1 + c_2) = 2 \\
  &\rightarrow c_1 = 1, \quad c_2 = 0
\end{align*}
\]

\[
\rightarrow f(n) = n \quad O(n) \text{ not } O(n \log n)
\]