Asymptotic Analysis

Before we look at some nifty applications of recursion, and provided perhaps by our experiences with the Fibonacci-like sequence, we should look a little more closely at program efficiency.

What concerns us here is how quickly our programs will run. This matters more as the number of computations increases . . .

<table>
<thead>
<tr>
<th># Steps</th>
<th>Asymptotic behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>$150n + 17$</td>
<td>$n$</td>
</tr>
<tr>
<td>$2n^3 + 9000000n^2 - 1$</td>
<td>$n^3$</td>
</tr>
<tr>
<td>$3 \log(n^2) - 12$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>$5^n + 17n^{84}$</td>
<td>$5^n$</td>
</tr>
</tbody>
</table>

The important measure is what the LH formula looks like when $n$ is ginormous! Notice that in line 2, the nine million times $n^2$ is irrelevant when $n$ is "the square of the National Debt"!

Recall (from math) that $e^x$ blows up faster than any polynomial in $x$, and hence any power (even fractional) of $x$ blows up faster than $\log x$ to any (constant) base.
We use a capital $O$ to mean order, so for example...

$$2n^3 + 9000000n^2 - 1 = O(n^3)$$

means that the LH side is of order $n^3$; multiplying factors are ignored—it's growth that counts.

**Formal Details**

We might think of $n$ as being some parameter, such as the number of elements of an array; and $T(n)$ as some function of $n$, such as the number of computations in some algorithm. Then...

1. $T(n) = O(f(n))$ if $\exists c > 0$ with $T(n) \leq cf(n)$ $\forall n \geq N_0$.

The idea being that we're looking for some function $f(n)$ so that eventually (i.e., as $n$ gets progressively larger) $T(n)$ is bounded above by some multiple of $f(n)$. In other words, the growth of the graph of $f(n)$ describes the shape of the growth of $T(n)$—or at least provides a worst case scenario. For example...

$$2n^3 + 9000000n^2 - 1 = O(n^4).$$
There are three other related expressions...

2. \( T(n) = \Omega(f(n)) \) if \( \exists c > 0 \) with \( T(n) = c f(n) \ \forall n \geq N \)

- "bounded below by"

For example...

\[ 2n^3 + 9,000,000 n^2 - 1 = \Omega(n^2). \]

3. \( T(n) = \Theta(f(n)) \) if \( \exists c_1, c_2 > 0 \) with \( c_1 f(n) \leq T(n) \leq c_2 f(n) \ \forall n \geq N \)

- "asymptotic to"

So \( T(n) = \Theta(f(n)) \) holds provided both \( T(n) = O(f(n)) \) and \( T(n) = \Omega(f(n)) \). For example...

\[ 2n^3 + 9,000,000 n^2 - 1 = \Theta(n^3). \]

We also denote this by \( T(n) \sim f(n) \).

4. \( T(n) = o(f(n)) \) if \( \frac{T(n)}{f(n)} \to 0 \) as \( n \to \infty \).

- "negligible compared to"

So \( T(n) = o(f(n)) \) holds provided both \( T(n) = O(f(n)) \) and \( T(n) \neq \Theta(f(n)) \), although really the emphasis is on \( T(n) \) being eventually negligible when compared to \( f(n) \). For example...

\[ 2n^3 + 9,000,000 n^2 - 1 = o(n^4). \]
To be honest, we really use these formalisms as shorthand for formal mathematical statements — it's a matter of understanding what a particular function "smells like". It's not hard to see how these expressions combine...

\[ O(f(n)) + O(g(n)) = O(\text{whichever of } f \text{ or } g \text{ grows faster as } n \to \infty) \]

\[ \Omega(f(n)) + \Omega(g(n)) = \Omega(\text{whichever of } f \text{ or } g \text{ grows slower as } n \to \infty) \]

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

Similar sorts of expressions hold for the various algebraic manipulations.

Applying all this to estimate the running time of code is initially not too hard. As an example, we'll consider the following problem. Suppose we have an array of integers, for example...

\[ 3 \ 2 \ -25 \ 14 \ 8 \ -22 \ 10 \ 1 \ 3 \ 7 \ -1 \ 3 \ -12 \]

The exercise is to find the maximum value that can be obtained by adding the terms of any contiguous subsequence.
A rather brain-dead approach, essentially trying everything, would be to consider every starting point, and for each starting point consider every ending point, and for each pair of starting and ending points add up the values of this subsequence, and then compare this with the biggest so far.

Written as code, this gives...

```java
public static int maxSSSum (int[] A)
{
    int maxSum = 0;
    for (int i = 0; i < A.length; i++)
        for (int j = i; j < A.length; j++)
            int thisSum = 0;
            for (int k = i; k <= j; k++)
                thisSum += A[k];
            if (thisSum > maxSum)
                maxSum = thisSum;
    return maxSum;
}
```

This set of 3 nested for-loops is easy, but horribly inefficient!

Let's analyse this method to see how many steps are taken as a function of the number of terms in the array.
Analyse...

a. length = N, so the 1st for loop executes N times. For each run of the 1st loop, the 2nd loop runs N-i times — essentially this is a triangular process

\[ i = 0, 1, 2, \ldots, j, \ldots, N-1 \]

So the 1st two for loops run \( \frac{1}{2} N(N+1) \) times. Finally the last for loop means we’re counting the number of triples \( 0 \leq i \leq k \leq j \leq N-1 \) — essentially a tetrahedral process giving \( \frac{1}{24} N(N+1)(N+2) \) steps. Ignoring constants, the asymptotic behaviour is hence \( O(N^3) \).

A simple process, but a horrible number of steps if \( N \) is large!!

A slight tidying up of our program can lose the inner \( k \) for loop, reducing the running time to \( O(N^2) \).

This is a very valuable improvement.
```java
public static int max55Sum (int[] a) {
    int maxSum = 0;
    for (int i = 0; i < a.length; i++) {
        int thisSum = 0;
        for (int j = i; j < a.length; j++) {
            thisSum += a[j];
            if (thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }
    return maxSum;
}
```

Finally we can be a lot sneakier and reduce the running time to \(O(N)\).

Such success is rare in general...

```java
public static int max55Sum (int[] a) {
    int maxSum = 0, thisSum = 0;
    for (int i = 0, j = 0; j < a.length; j++) {
        thisSum += a[j];
        if (thisSum > maxSum) {
            maxSum = thisSum;
        } else if (thisSum < 0) {
            i = j + 1;
            thisSum = 0;
        }
    }
    return maxSum;
}
```
There's yet another way we could approach this maximum subsequence problem, we can build a recursive attack. Essentially the max subsequence will either
a. be in the left half of the array
b. ... ... right ... ... ... ...
or c. straddle both the left and right halves.
The corresponding recursive code is ...

```java
public static int maxSS (int i, j, a)
    & return maxSSR (a, 0, a.length - 1) }

private static int maxSSR (int i, j, L, R)
    }
    max border sums
    int maxL = 0, maxR = 0, Lbord = 0, Rbord = 0;
    int centre = (L + R) / 2;  // integer arithmetic
    if (L == R) return a[L] > 0 ? a[L] : 0;

    int maxLS = maxSSR (a, L, centre);
    int maxRS = maxSSR (a, centre + 1, R);
    for (int i = centre; i >= L; i--)
        & Lbord += a[i];
        if (Lbord > maxL) maxL = Lbord;
    }

    for (int j = centre + 1; j <= R; j++)
        & Rbord += a[j];
        if (Rbord > maxR) maxR = Rbord;
    }

    return max3 (maxLS, maxRS, maxL + maxR);
    }
```

returns biggest of 3 — easy to write!
Before we look at our last example, two principles are worth stating...

Repeated doubling principle: with \( x = 1 \), how often should \( x \) be doubled to reach a given \( N \)?

\[
2^y x = N \implies y = \log_2 N \quad (x = 1)
\]

Repeated halving principle: with \( x = N \), how often should \( x \) be halved to reach 1?

\[
\left(\frac{1}{2}\right)^y N = 1 \implies y = \log_2 N
\]

Since

\[
\log_2 N = c \log_r N, \quad \text{for any } r > 1 \implies c > 0,
\]

we're content to say that both of these are \( O(\log N) \) principles.

If we need to search a given batch of data, then checking each piece of data term-by-term for a match is clearly \( O(N) \). We can improve on sequential search by using the binary search algorithm, which assumes the data to be sorted in advance, and then restricts its attention to the likely half after each unsuccessful match. This repeated halving is of course \( O(\log N) \), which is a dramatic improvement....
public interface Comparable
{
    int compares(Comparable rhs);
    boolean lessThan(Comparable rhs);
}

This interface would need to be implemented by a class, however, assuming this...

public static int binSearch(Comparable[] a, Comparable x)
    throws ItemNotFound
{
    int low = 0, high = a.length - 1, mid;
    while (low <= high) // low & high can change here!
    {
        mid = (low + high) / 2;
        if (a[mid].compares(x) < 0) // Go Right!
            low = mid + 1;
        else if (a[mid].compares(x) > 0) // Go Left!
            high = mid - 1;
        else
            return mid; // Found it!
    }
    throw new ItemNotFound("Binary Search Failed!");
}

We'll look at this example in a lot more detail later on — for now it's enough to notice the repeated halving.