The $O()$ notation

When analyzing the runtime of an algorithm, we want to consider the time required for large $n$.

We also want to ignore constant factors (which often stem from “tricks” and do not indicate a better procedure).

**Definition:** Let $f(n), g(n)$ be functions of the natural (or real) numbers. We say that

$$f(n) = O(g(n))$$

if there exist constants $A, N$ such that

$$|f(n)| \leq |A \cdot g(n)| \quad \text{for all} \quad n \geq N.$$
For example

\[
1 + n = O(n) \\
9n^2 + 7n - 28 = O(n^2) \\
31415926 = O(1)
\]

We may also write \( f(n) = g(n) + O(h(n)) \), meaning that \( f(n) - g(n) = O(h(n)) \).

For example (remember that \( 1 + 2 + \cdots + n = n(n + 1)/2 = \frac{1}{2}n^2 + \frac{1}{2}n \))

\[
1 + 2 + \cdots + n = O(n^3) \\
1 + 2 + \cdots + n = O(n^2) \\
1 + 2 + \cdots + n = \frac{1}{2}n^2 + O(n)
\]
Lemma: Let \( f(n) = O(r(n)) \) and \( g(n) = O(r(n)) \). Then:

\[
\begin{align*}
f(n) + g(n) &= O(r(n) + s(n)) \\
f(n) \cdot g(n) &= O(r(n) \cdot s(n)) \\
f(n) + r(n) &= O(r(n))
\end{align*}
\]

but:

\[
O(f(n)) - O(g(n)) \neq O(f(n) - g(n))
\]
**Theorem:** Let \( f \) be a *monotonous* function (i.e. \( f(n_1) \geq f(n_2) \) whenever \( n_1 \geq n_2 \)). Then for every \( c > 0 \) and every \( a > 1 \) we have that

\[
(f(n))^c = O(a^{f(n)})
\]

In particular \( n^c = O(a^n) \).

Similarly \( \log n = O(n^d) \) for all \( d > 0 \).

The following list is in “ascending \( O \)-order”:

1 \( \log n \) \( \sqrt{n} \) \( n \) \( n \log n \) \( n^2 \)

\( n^3 \) \( \ldots \) \( 1.5^n \) \( 2^n \) \( \ldots \) \( n^n \)
Consider a vector space consisting of (complex-valued) functions. (If you like column vectors consider the discretized version with vectors of the form $(f(t_1), f(t_2), \ldots, f(t_n))$ obtained from function evaluations.)

One can define a scalar product between functions as

$$(f, g) = \int f(t)\overline{g(t)} \, dt$$

In the same way as with column vectors, it is now possible to define the projection of a function on a subspace spanned by basis functions $b_i$. 
We can use projection methods to decompose a function in the subspace into the basis elements:

\[ f(t) = \sum c_i b_i(t) \]

If the \( b_i \) form an orthonormal basis, the projection formulae give us that

\[ c_i = (f, b_i) = \int b_i(t) \overline{f(t)} \, dt \]

These coefficients \( c_i \) are called the *Fourier Coefficients* of \( f \).
The Fourier Transformation

Consider the functions $\sin(n \cdot t)$ and $\cos(n \cdot t)$ on the interval $[0, 2\pi]$. Then (for $n \neq m$):

$$0 = \int_0^{2\pi} \sin(nt) \sin(mt) \, dt = \int_0^{2\pi} \cos(nt) \cos(mt) \, dt$$

$$= \int_0^{2\pi} \sin(n_1 t) \cos(n_2 t) \, dt$$

$$\pi = \int_0^{2\pi} \sin(nt) \sin(nt) \, dt = \int_0^{2\pi} \cos(nt) \cos(nt) \, dt$$

Thus the function $\frac{1}{\sqrt{\pi}} \sin(nt)$, $\frac{1}{\sqrt{\pi}} \cos(nt)$ form an orthonormal basis for a subspace.

**Theorem:** This subspace consists of all periodic functions.
In general, one can consider not just integral, but real multiples. In the limit the infinite sum becomes an integral and we have

\[
(\ast) \quad f(t) = \int_0^\infty A(\omega) \sin(\omega t) \, d\omega \\
+ \int_0^\infty B(\omega) \cos(\omega t) \, d\omega
\]

with

\[
A(\omega) = \frac{1}{\pi} \int f(t) \sin(\omega t) \, dt \\
B(\omega) = \frac{1}{\pi} \int f(t) \cos(\omega t) \, dt
\]

These functions \( A \) and \( B \) indicate the weight of the various frequencies.
Applications

The Fourier transform has many applications in signal processing. We can use it for example to filter low frequencies:
\[ \cos(2\pi f_1 t) \]
Perceptual Coding

Another application is in audio encoding. The perception of loudness depends on the frequency.
Also the human ear cannot recognize frequencies with low amplitude, if another frequency "close by" has high amplitude:
When recording the audio signal, we can remove such "masked" frequencies, this data reduction cannot be spotted by the ear.

This principle is used for example in the MPEG standards (e.g. MP3 audio), similar techniques apply to pictures.
Interpretation over the complex numbers

Things become more uniform if we move to the complex numbers. Here we have: \( \sin(x) = \frac{e^{ix} - e^{-ix}}{2i} \), \( \cos(x) = \frac{e^{ix} + e^{-ix}}{2} \) and

\[
e^{a+ib} = e^a \cdot (\cos b + i \sin b)
\]

The Complex Fourier transform

Using these relations, we get the Fourier transform of \( f \) as:

\[
(F(f))(\omega) = g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{i\omega t} \, dt
\]

It fulfills (call \( f = \overline{F}(g) \) the inverse transform of \( g \))

\[
f(t) = (\overline{F}(g))(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega) e^{-i\omega t} \, d\omega
\]
Discretization

On a computer we cannot take infinitely many values but have to 
*discretize* an continuous process.

For every natural \( n \) and \( 0 \leq m < n \) the numbers:

\[
\frac{2\pi im}{n}
\]

lie equally distributed on the complex unit circle. We call these numbers *Roots of Unity.*
Let \( \omega := e^{\frac{2\pi i}{n}} \). Then \( \omega \) fulfills:

1. \( \omega^n = 1 \), \( \omega^m \neq 1 \) (\( 1 \leq m < n \))

2. \( \sum_{j=0}^{n-1} \omega^{jp} = 0 \) for \( 1 \leq p < n \)

A number \((\neq 1)\) fulfilling both properties is called a \textit{principal} \( n \)-th root
of unity.

If $\omega$ is a principal $n$-th root of unity, then $\omega^{-1}$ is as well. (This is all we need to know about complex numbers.)

The integrals of the Fourier transform thus become sums in the *Discrete Fourier transform* (DFT). Let $\mathbf{a} = [a_0, \ldots, a_{n-1}]$. Then $\mathcal{F}(\mathbf{a}) = [u_0, \ldots, u_{n-1}]$ with:

$$u_j = \sum_{k=0}^{n-1} a_k \omega^{jk}$$

If we define a polynomial $p(x) = \sum_{k=0}^{n-1} a_k x^k$, we have $u_j = p(\omega^j)$, the Fourier transformation thus consists of multiple polynomial evaluations.
Similarly we define the inverse discrete Fourier transform:

\[ \mathcal{F}(u) = [v_0, \ldots, v_{n-1}] \quad \text{with} \quad v_j = \frac{1}{n} \sum_{k=0}^{n-1} u_k \omega^{-jk}. \]

It is easily checked that \( \mathcal{F} \) and \( \mathcal{F}^{-1} \) are mutually inverse.

Furthermore the inverse can be computed by the same algorithm, replacing \( \omega \) by \( \omega^{-1} \).

If \( a \in \mathbb{R}^n \), a naïve evaluation of the DFT takes \( O(n^2) \) operations.

Our next aim will be to develop an \( O(n \log n) \) algorithm.
Fast Fourier Transformation

A particular efficient way to evaluate the discrete Fourier transform is the *Fast-Fourier transform* (FFT) algorithm of Cooley and Tukey:

Let \( \mathbf{u} = \mathcal{F}(\mathbf{a}) \). Then we can write \( \mathbf{u} = V \cdot \mathbf{a} \) with

\[
V = \begin{pmatrix}
1 & 1 & 1 & \ldots & 1 \\
1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\
1 & \omega^2 & \omega^4 & \ldots & \omega^{2(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)}
\end{pmatrix}
\]

a Vandermonde matrix.
Let us consider the case \( n = 8 \). Then

\[
\begin{pmatrix}
  u_0 \\
  u_1 \\
  u_2 \\
  u_3 \\
  u_4 \\
  u_5 \\
  u_6 \\
  u_7
\end{pmatrix} = \begin{pmatrix}
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\
  1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\
  1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\
  1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\
  1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega^1 & \omega^6 & \omega^3 \\
  1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\
  1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1
\end{pmatrix} \begin{pmatrix}
  a_0 \\
  a_1 \\
  a_2 \\
  a_3 \\
  a_4 \\
  a_5 \\
  a_6 \\
  a_7
\end{pmatrix}
\]
We swap the columns to sort by even/odd indices: $[0, 2, 4, 6, 1, 3, 5, 7]$:

\[
\begin{pmatrix}
  u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\
\end{pmatrix}
= 
\begin{pmatrix}
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  1 & \omega^2 & \omega^4 & \omega^6 & \omega & \omega^3 & \omega^5 & \omega^7 \\
  1 & \omega^4 & 1 & \omega^4 & \omega^2 & \omega^6 & \omega^2 & \omega^6 \\
  1 & \omega^6 & \omega^4 & \omega^2 & \omega^3 & \omega^1 & \omega^7 & \omega^5 \\
  1 & 1 & 1 & 1 & \omega^4 & \omega^4 & \omega^4 & \omega^4 \\
  1 & \omega^2 & \omega^4 & \omega^6 & \omega^5 & \omega^7 & \omega^1 & \omega^3 \\
  1 & \omega^4 & 1 & \omega^4 & \omega^6 & \omega^2 & \omega^6 & \omega^2 \\
  1 & \omega^6 & \omega^4 & \omega^2 & \omega^7 & \omega^5 & \omega^3 & \omega^1 \\
\end{pmatrix}
\begin{pmatrix}
  a_0 \\ a_2 \\ a_4 \\ a_6 \\ a_1 \\ a_3 \\ a_5 \\ a_7 \\
\end{pmatrix}
\]
We observe that we can write

\[
(\omega, \omega^3, \omega^5, \omega^7) = \omega(1, \omega^2, \omega^4, \omega^6) = \omega(1, \zeta, \zeta^2, \zeta^3)
\]
\[
(\omega^2, \omega^6, \omega^2, \omega^6) = \omega^2(1, \omega^4, 1, \omega^4) = \omega^2(1, \zeta^2, 1, \zeta^2)
\]
\[
(\omega^3, \omega, \omega^7, \omega^5) = \omega^3(1, \omega^6, \omega^4, \omega^2) = \omega^3(1, \zeta^3, \zeta^2, \zeta)
\]

where \(\zeta = \omega^2\) is a primitive 4-th root of unity and we have a 4-dimensional Fourier transform given by

\[
\begin{pmatrix}
    u_0 \\
    u_1 \\
    u_2 \\
    u_3
\end{pmatrix} =
\begin{pmatrix}
    1 & 1 & 1 & 1 \\
    1 & \zeta & \zeta^2 & \zeta^3 \\
    1 & \zeta^2 & 1 & \zeta^2 \\
    1 & \zeta^3 & \zeta^2 & \zeta
\end{pmatrix} \cdot 
\begin{pmatrix}
    a_0 \\
    a_1 \\
    a_2 \\
    a_3
\end{pmatrix}.
\]

In other words: We can write the transformation matrix \(F_8\) as a product:
where $F_4$ is the matrix for the 4-dimensional transformation.
We can now apply the same method recursively to the 4-dimensional transformation and so on.

Eventually, this eliminates all matrix multiplication by scalar multiplication.

For an $n$-dimensional problem we have $\log(n)$ iterations, each step of which has a cost of $O(n)$. The total cost is thus $O(n \log n)$.

The same process works for any power of 2 (filling up the data if there are fewer points).

The inverse transform uses the same algorithm with $\omega$ replaced by $\omega^{-1}$. 
function FFT(n, a, ω);  
(n a power of 2)
Output: u = F(a).
begin
    u := [];
    if n = 1 then
        u[0] := a[0];
    else
        m := n/2;
        c := FFT(m, a_e, ω^2); d := FFT(m, a_o, ω^2);
        for j ∈ [0..m − 1] do
        od;
    fi;
    return u;
end
Example: Sunspots

We want to analyze the number of sunspots per year. Astronomers have counted these for about three centuries, their number is called the *Wolfer number*.

We will use **MATLAB** to perform a fourier transform. If we log in remotely (from another X-windows workstation), we have to redirect graphics output:

```
$ setenv DISPLAY mycomputer.math.colostate.edu:0
$ matlab
```

Once in **MATLAB** we load the data (which is supplied as a sample) and split the list of pairs in a list of years and of counts. We also plot the curve.
load sunspot.dat
year=sunspot(:,1); wipher=sunspot(:,2);
plot(year, wipher)
title('Sunspot Data')
We now perform a fast fourier transformation. This stores the coefficients in a list. The first coefficient \( \cos(0x) = 1 \) gives an absolute shift (zero bias) that is usually not interesting, so we eliminate it.

\[
Y = \text{fft} \left( \text{wolfer} \right);
\]

\[
Y(1) = 0;
\]

The first \( \frac{n}{2} \) list entries correspond to positive frequencies, the last \( \frac{n}{2} \) corresponding to negative frequencies. Since our input signal is real, we need to consider only the entries corresponding to positive frequencies. (The rest will be their complex conjugates.)

Since we are only interested in the intensities for the various frequencies, we only consider the (absolute values of) the first \( \frac{n}{2} \) entries. Taking their square (proportional to the power per frequency) will emphasize peaks.

\[
n = \text{length} \left( Y \right);
\]
intens = abs(Y(1:n/2));
power = intens.^2;;

To obtain the frequencies, we note that our frequency resolution will be \( \frac{1}{2} \) times the sampling frequency (Nyquist rate/Shannon's theorem). Thus we create a equidistributed list of length \( \frac{n}{2} \) which goes from 1 to \( \frac{\text{rate}}{2} \). Also, since the frequencies are very small, we instead will use their reciprocal values.

rate=1;
nyquist = 1/2;
\text{freq} = (1:n/2)/(n/2)*nyquist*rate;
\text{period}=1./\text{freq};
Finally we plot the frequency distribution and set the view window to $x \in [0, 40], y \in [0, 2 \cdot 10^7]$, which fits the plot.

```matlab
plot(period, power);
ylabel('Power');
xlabel('Period (Years/Cycle)');
axis([0 40 0 2e+7]);
```