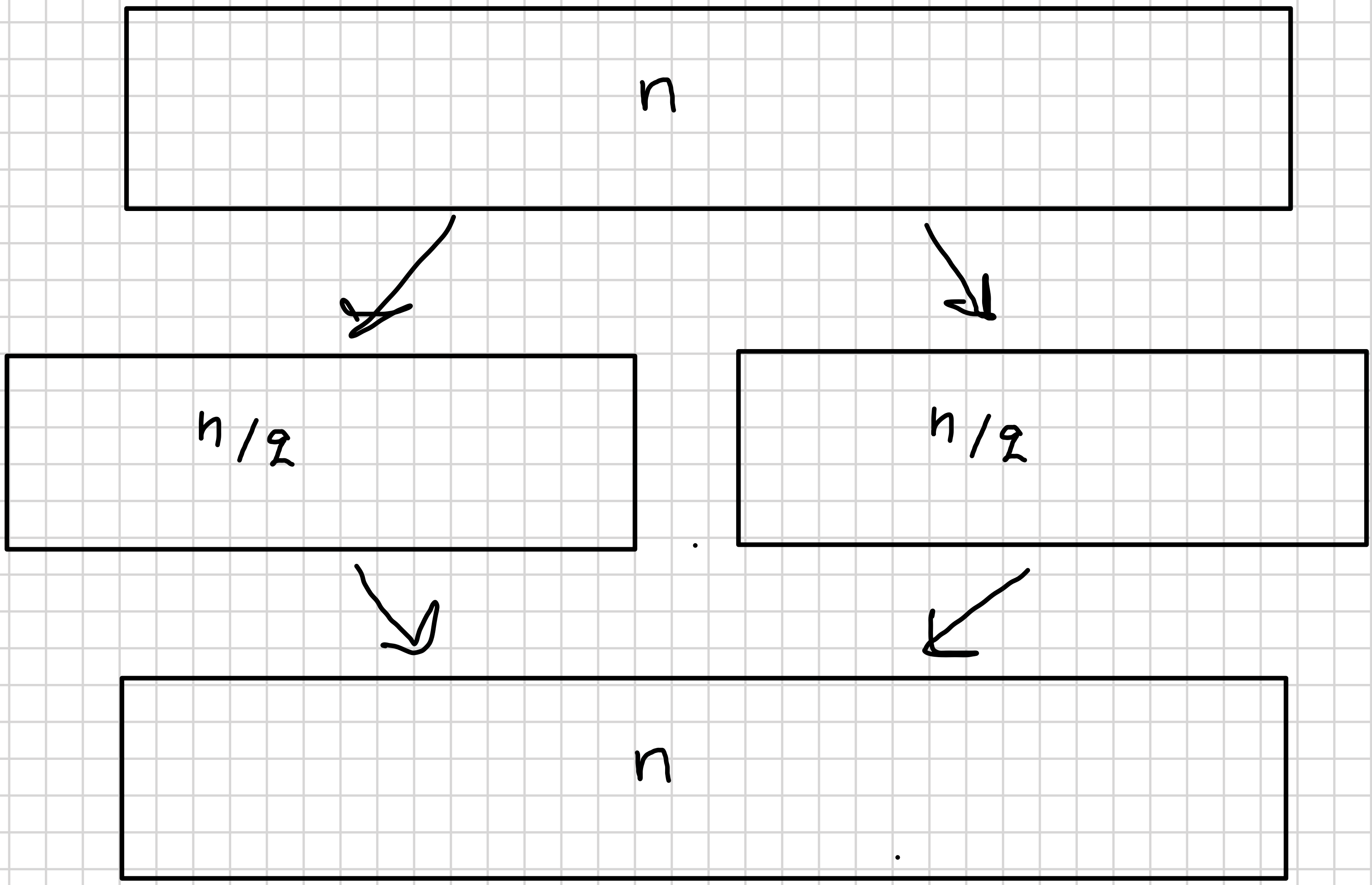
Divide and Conquer algorithms

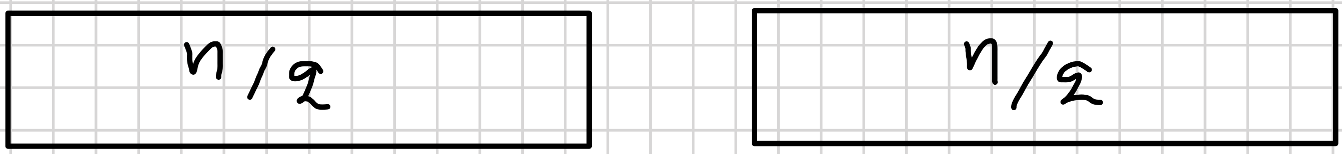
Another general method for constructing algorithms is given by the Divide and Conquer strategy. We assume that we have a problem with input that can be split into parts in a natural way.



Let $T(n)$ be the time-complexity for solving a problem of size n (using our algorithm). Then we have $T(n) = T(n/2) + T(n/2) + f(n)$
where $f(n)$ is the time for "making the split" and "putting the parts together.
This will be useful only if $f(n)$ is sufficiently small.

Mergesort

A famous example is Mergesort. Here we split a list of numbers into two parts, sort them separately, and merge the two lists.



How do we merge?

The question is how we merge two already sorted lists and what the complexity f(n) is?

We can use the following algorithm:

```
Merge[a[1, ..., p], b[1,...,q]]
    If a = Ø
        Return a
    End if
    If b = Ø
        Return b
    End if
    If a[1] ≤ b[1]
        Return a[1] . Merge[a[2,...,p],b[1,...,q]]
    End if
    Return b[1] . Merge[a[1,...,p],b[2,...,q]]
```

The complexity is $O(n)$.

The main Mergesort algorithm is:

MergeSort

MergeSort($v[i..j]$)
(1) **if** $i < j$
(2) $m \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
(3) MergeSort($v[i..m]$)
(4) MergeSort($v[m+1..j]$)
(5) $v[i..j] =$ Merge($v[i..m], v[m+1..j]$)

Let $T(N)$ be the time it takes to sort N numbers. then

$$T(N) = \begin{cases} O(1) & N = 1 \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + \Theta(N) & \text{else} \end{cases}$$

since Merge $\Theta(N)$ when input is arrays of length N.

But how do we decide the complexity? We are given a recursion equation. The following theorem often gives the solution:

Master Theorem

Theorem *If $a \geq 1$, $b > 1$ and $d > 0$ the equation*

$$T(1) = d$$
$$T(n) = aT(n/b) + f(n)$$

has the solution

- $T(n) = \Theta(n^{\log_b a})$ *if* $f(n) = O(n^{\log_b a - \epsilon})$ *for some* $\epsilon > 0$

- $T(n) = \Theta(n^{\log_b a} \log n)$ *if* $f(n) = \Theta(n^{\log_b a})$

- $T(n) = O(f(n))$ *if* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *for some* $\epsilon > 0$ *and* $af(n/b) \leq cf(n)$ *for some* $c < 1$ *for* n *large enough.*

When applied on Mergesort this theorem gives $\Theta(N \log N)$.

Let's look at some more advanced examples.

Multiplication of large numbers

We want to compute $x \cdot y$ for binary numbers x och y

$$x = \underbrace{x_{n-1} \cdots x_{n/2}}_{a} \underbrace{x_{n/2-1} \cdots x_1 x_0}_{b} = 2^{n/2}a + b$$

$$y = \underbrace{y_{n-1} \cdots y_{n/2}}_{c} \underbrace{y_{n/2-1} \cdots y_1 y_0}_{d} = 2^{n/2}c + d$$

For $n = 2^k$ we can split the product:

Mult(x, y)
(1) **if** $length(x) = 1$
(2) **return** $x \cdot y$
(3) **else**
(4) $[a, b] \leftarrow x$
(5) $[c, d] \leftarrow y$
(6) $prod \leftarrow 2^n Mult(a, c) + Mult(b, d)$
$+2^{n/2}(Mult(a, d) + Mult(b, c))$
(7) **return** $prod$

Time-complexity: $T(n) = 4T(n/2) + \Theta(n)$, $T(1) = \Theta(1)$ which gives us $T(n) = \Theta(n^2)$.

Here is a way of doing it that really uses D and C:

Karatsuba's algorithm

We use $(a+b)(c+d) = ac + bd + (ad + bc)$.
We can remove one of the four products:

Mult(x, y)
(1) **if** $length(x) = 1$
(2) **return** $x \cdot y$
(3) **else**
(4) $[a, b] \leftarrow x$
(5) $[c, d] \leftarrow y$
(6) $ac \leftarrow Mult(a, c)$
(7) $bd \leftarrow Mult(b, d)$
(8) $abcd \leftarrow Mult(a + b, c + d)$
(9) **return** $2^n \cdot ac + bd +$
$2^{n/2}(abcd - ac - bd)$

We get $T(n) = 3T(n/2) + \Theta(n)$, $T(1) = \Theta(1)$ with the solution $T(n) = \Theta(n^{\log_2 3}) \in O(n^{1.59})$. .

Here is an algorithm that fails to use D and C in a creative way.

Matrix multiplication

When we multiply $n \times n$-matrices we can use matrix blocks:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

by using the formulas

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

we get 8 products and

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 8T(n/2) + \Theta(n^2) & n > 1 \end{cases}$$

which gives us $T(n) = \Theta(n^3)$.

But this is D and C:

.

Strassen's algorithm

If we instead use the more complicated formulas

$$\begin{aligned}
M_1 &= (A_{12} - A_{22})(B_{21} + B_{22})\\
M_2 &= (A_{11} + A_{22})(B_{11} + B_{22})\\
M_3 &= (A_{11} - A_{21})(B_{11} + B_{12})\\
M_4 &= (A_{11} + A_{12})B_{22}\\
M_5 &= A_{11}(B_{12} - B_{22})\\
M_6 &= A_{22}(B_{21} - B_{11})\\
M_7 &= (A_{21} + A_{22})B_{11}
\end{aligned}$$

$$\begin{aligned}
C_{11} &= M_1 + M_2 - M_4 + M_6\\
C_{12} &= M_4 + M_5\\
C_{21} &= M_6 + M_7\\
C_{22} &= M_2 - M_3 + M_5 - M_7
\end{aligned}$$

we reduce the number of products to 7 which gives us $T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$.

An advanced application of D and C is the Fast Fourier Transform (FFT). We start by describing what the Discrete Fourier Transform (DFT) is:

Discrete Fourier Transform

We transform a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$. Essentially we do it by computing it's values for the complex unity roots $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ where $\omega_n = e^{2\pi i/n}$.

$$DFT_n(\langle a_0, \ldots, a_{n-1} \rangle) = \langle y_0, \ldots, y_{n-1} \rangle$$

where

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j e^{2\pi ijk/n}.$$

The n coefficients gives us n "frequencies". Compare with the continuous transform

$$\hat{f}(t) = \int_{-\infty}^{\infty} f(x) e^{-itx} dx$$

This simplest way of computing this transform has complexity $O(n^2)$.
The FFT is a more efficient way of doing it.

FFT: An efficient way of computing DFT

We have $y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j e^{2\pi ijk/n}$. We separate odd and even degrees in A:

For $k < n/2$ We have

$$
\begin{aligned}
A^{[0]}(\omega_n^{2k}) &= \sum_{j=0}^{n/2-1} a_{2j} e^{4\pi ijk/n} \\
&= \sum_{j=0}^{n/2-1} a_{2j} \omega_{n/2}^{jk} \\
&= DFT_{n/2}(\langle a_0, a_2, \ldots, a_{n-2} \rangle)_k
\end{aligned}
$$

where $DFT_n(\langle a_0, \ldots, a_{n-1} \rangle)_k$ is the k:th element of the transform.

In the same way, for $k < n/2$,

$$A^{[1]}(\omega_n^{2k}) = DFT_{n/2}(\langle a_1, a_3, \ldots, a_{n-1}\rangle)_k$$

For $k \geq n/2$ we can easily see that

$$A^{[0]}(\omega_n^{2k}) = DFT_{n/2}(\langle a_0, a_2, \ldots, a_{n-2}\rangle)_{k-n/2}$$
$$A^{[1]}(\omega_n^{2k}) = DFT_{n/2}(\langle a_1, a_3, \ldots, a_{n-1}\rangle)_{k-n/2}$$
$$\omega_n^k = -\omega_n^{k-n/2}$$

In order to decide $DFT_n(\langle a_0, \ldots, a_{n-1}\rangle)$ we use $DFT_{n/2}(\langle a_0, a_2, \ldots, a_{n-2}\rangle)$ and $DFT_{n/2}(\langle a_1, a_3, \ldots$ and combine values.

FFT is a Divide Conquer algorithm — the base case is $DFT_1(\langle a_0\rangle) = \langle a_0\rangle$.

Algorithm for computing FFT

We assume that n is a power of 2.

$DFT_n(\langle a_0, a_1, \ldots a_{n-1} \rangle)$

if $n = 1$
    return $\langle a_0 \rangle$
$\omega_n \leftarrow e^{2\pi i/n}$
$\omega \leftarrow 1$
$y^{[0]} \leftarrow DFT_{n/2}(\langle a_0, a_2, \ldots, a_{n-2} \rangle)$
$y^{[1]} \leftarrow DFT_{n/2}(\langle a_1, a_3, \ldots, a_{n-1} \rangle)$
for $k = 0$ **to** $n/2 - 1$
    $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$
    $y_{k+n/2} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$
    $\omega \leftarrow \omega \cdot \omega_n$
return $\langle y_0, y_1, \ldots, y_{n-1} \rangle$

The time-complexity $T(n)$ is given by

$$T(n) = \begin{cases} O(1) & n = 1 \\ 2T(n/2) + \Theta(n) & n > 1 \end{cases}$$

with solution $T(n) = \Theta(n \log n)$.

Inverse to DFT

The relation $y = DFT_n(a)$ can be written in matrix form

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

To get the inverse transformation $a = DFT_n^{-1}(y)$ we invert the matrix It can be shown that

$$DFT_n^{-1}(\langle y_0, y_1, \ldots, y_{n-1} \rangle) = \langle a_0, a_1, \ldots, a_{n-1} \rangle$$

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-jk}$$

so the FFT-algorithm can also be used to compute DFT^{-1}.

Polynomial multiplication using FFT

We want to compute $C(x) = \sum_{j=0}^{2n-2} c_i x^i = A(x)B(x)$ when $A(x)$ and $B(x)$ are polynomials of degree $n-1$. Since $C(x)$ has $2n-1$ coefficients we will look at $A(x)$ and $B(x)$ as polynomials of degree $2n-1$ as well.

Algorithm:

$$\langle y_0, \ldots y_{2n-1} \rangle \leftarrow DFT_{2n}(\langle a_0, \ldots, a_{n-1}, 0, \ldots, 0 \rangle)$$
$$\langle z_0, \ldots z_{2n-1} \rangle \leftarrow DFT_{2n}(\langle b_0, \ldots, b_{n-1}, 0, \ldots, 0 \rangle)$$
$$\langle c_0, \ldots c_{2n-1} \rangle \leftarrow DFT_{2n}^{-1}(\langle y_0 z_0, \ldots, y_{2n-1} z_{2n-1} \rangle)$$

(We assume that n is a power of two.)

We have to do compute three DFT vectors of size $2n$ and compute $2n$ products in the transform plane. That gives us the complexity $\Theta(n \log n)$.