FFT, fast implementation (OUCH! IT'S TOO LATE.)

1) Choice of algorithm: Choose recursive FFT, not iterative FFT

2) Locality optimization:

\[ \text{DFT}_{km} = (\text{DFT}_n \otimes I_m) T_{km} (I_n \otimes \text{DFT}_m) L_{km} \]

(schematic) =

- Compute in many
  \[ \text{DFT}_{km} \quad \text{part of } \quad \text{DFT}_n \]
- at stride \( m \)
  \[ \text{at stride } m \]
- with input and output
  \[ \text{input and output} \]
- writes to the same
  \[ \text{writes to the same} \]
- location it reads from
  \[ \text{location it reads from} \]
- in-place
  \[ \text{in-place} \]

DFTscaled(k, xx, x*0, stride)

\[ \text{size input diagonal} \]
\[ \text{output elements vector} \]

This interface cannot handle arbitrary recursions

→ in FFTW a base case

Pseudocode:

\[
\text{DFT}(n, x, y) = \text{DFTrec}(n, x, y, 1, 1) \\
\text{for (int } i = 0; i < k; ++i) \\
\quad \text{DFTrec}(m, y + m*i, x + i, k, 1); \quad // \text{implemented as } \text{DFT}(\ldots) \text{ is} \\
\text{for (int } j = 0; j < m; ++j) \\
\quad \text{DFTscaled}(k, y + j, t[j], m); \quad // \text{always a base case} \\
\]

\[ y \leftarrow \text{stage 2} \]
\[ y \leftarrow \text{stage 1} \]
\[ x \]
3.) Constants:
The matrix \( T_n^m \) yields multiplications by constants:
\[
y_i = \frac{1}{m} e^{\frac{2\pi}{n} xi}.
\]
where \( x \) is a root of unity.

which in the code, on real numbers, gives multiplications by sines and cosines
\[
y_i = \sin\left(\frac{2\pi}{n} i \cdot x_i\right) \text{ etc.}
\]

Problem: Computed \( \sin(\cdot) \) is very expensive (HW 2)

Solution:
- precompute once
- reuse many times
- assumes a specific form for \( \sin \)
- is used many times

Changes library interface:
\[
d = \text{dft-plan}(1024); \quad \text{// precompute constants}
d(\cdot, x, y); \quad \text{// computes DFT, size 1024}
\]

4.) Fast base blocks:
We do not want to reuse all the way up to \( n = 2^k \)
- function call overhead
- suboptimal register use

Solution:
- unroll recursion for small enough \( n \)
- practice shows \( n \leq 32 \) is sufficient
- requires 62 functions! Why?

FFTW: "codeless" generator for small size FFT

\[
\begin{array}{c}
n \rightarrow \text{DAG generator} \rightarrow \text{Simplifier} \rightarrow \text{Scheduler} \rightarrow \text{straight-line code for DFTrec(n)}
\end{array}
\]

a.) DAG generator recursively
- generates DAG from several algorithms
- DAGs have only adds/subs/muls by const

Example:
\[
x_0 \rightarrow x \oplus x_0 \rightarrow (y_0) = (1 \cdot 1) \cdot (c)(x_i)
x_i \rightarrow x \oplus x_i \rightarrow (y_1) = (1 \cdot 1) \cdot (c)(x_i)
\]
6. Simplification
- Simplifies multiplies by 0, 1, -1
- Distributive law: \( kx + ky = k(x + y) \)
- Canonicalization: \( x - y, y - x \rightarrow x - y \), - \( (x - y) \)
- Common subexpression elimination (CSE)
- All constants are made positive:
  - Reduces register pressure
- CSE also on transposed DAG

C. Scheduler:

Theoretical result: 2-power FFT needs
\[ R \left( \frac{n \log(n)}{R^2} \right) \] register spills
for \( R \) registers

The following algorithm achieves that:

```
   in: \[
   \begin{array}{c}
   \text{in} \\
   \end{array}
   \]
   out: \[
   \begin{array}{c}
   \text{out} \\
   \end{array}
   \]
```

- Step 1: cut DAG in modules (how to do that)
- Step 2: DAG, \( \text{DAG}_i \) decompose into independent components

```
\text{DAG}_i: \[
\begin{array}{c}
\text{DAG}_i \\
\end{array}
\]
```

- Schedule these recursively

Finally: output straightline, SSA code