FFT, fast implementation (please try \( f(n) \approx x \cdot n \) )

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

2) Locality optimization:

\[
\text{DFT}_{km} = (\text{DFT}_m \otimes \text{I}_m) \cdot T_m = (\text{I}_k \otimes \text{DFT}_m) \cdot L_k
\]

(schematic) =

\[
\begin{array}{c}
\text{compute in many } \\
\text{DFT}_m \cdot D \uparrow \\
\text{part of } \\
\text{Legendre } T_m \\
\text{at stride } m \\
\text{Input and output} \\
\rightarrow \text{writes to the same location it reads from} \\
\rightarrow \text{inplace}
\end{array}
\]

\[
\begin{array}{c}
\text{compute } k \text{ many } \\
\text{DFT}_m \text{ with input stride } k \text{ and} \\
\text{output stride } 1 \\
\rightarrow \text{writes to different location} \\
\rightarrow \text{out-of-place}
\end{array}
\]

\[
\begin{array}{c}
\text{DFTscaled}(k, xx, xd, stride) \\
\uparrow \uparrow \uparrow \\
\text{slice input diagonal} \\
\text{output elements} \\
\text{vector}
\end{array}
\]

This interface cannot handle arbitrary recursions

→ in FFTW a base case

Pseudocode:

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

precomputed twiddles
1.) Constants:

The matrix $T^\pi$ yields multiplications by constants:

$$y_i = \psi^2_i x_i$$

where $\psi_i$ is some root of unity, which in the code, on real numbers, gives multiplications by sines and cosines.

$$y_i = \sin \left( \frac{i \pi}{127} \right) x_i \quad \text{etc...}$$

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

Solution:
- precompute once
- reuse many times
- assumes a transform for one size is used many times

Changes library interface:

```c
d = dft-plan(1024); // precompute constants
```

```c
d(x, y); // computes DFT, size 1024
```

4.) Fast basis blocks

We do not want to recompute all the way $y_0 \to y_n$ in $n = 2$

- 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: "coolest" generator for small size FFT

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

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

Example:

\[ x_0 \quad \Box \quad Y_0, \quad \leftarrow (Y_0) = (1-1)(1+c)(x_0) \]

\[ x_1 \quad \Box \quad Y_1, \quad \leftarrow (Y_1) = (1-1)(1+c)(y_1) \]
6. Simplifier
- simplifies mutlty by 0, 1, -1
- distributivity 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 $\frac{\log_2(n)}{R} \text{register spills}$ for $R$ registers

The following algorithm achieves that:

\[
\begin{array}{c}
\text{in} \quad \text{DAG} \quad \text{out} \\
\end{array}
\]

Step 1: cut DAG in modules (how do do that

\[
\begin{array}{c}
\text{DAG}_1 \quad \text{DAG}_2 \\
\end{array}
\]

Step 2: DAG$_1$, DAG$_2$ decompose into independent components

\[
\begin{array}{c}
\text{DAG}_1: \quad \text{DAG}_2: \\
\end{array}
\]

schedule these recursively

Finally: output straightline, SSA code