# Oblig 4 Rapport - Optimalisering

## Implementasjon av CT:

### Kjapp forklaring av minneallokering i C:

I programmeringspråket Java allokeres det minne når "new" blir skrevet. I C derimot, brukes malloc (slik som i den naive implementasjonen). Når den naive implementasjonens funksjon blir kjørt, vil det da allokeres nytt minne hver gang funksjonen kalles. Noe som er tids- og plasskrevende.

### Litt om Cooley-Turkey implementasjonen (hentet fra Cooley–Tukey FFT algorithm - Wikipedia):

Den mest vanlige formen for Cooley-Tukey algoritmen er "the radix-2 DIT case". Her benytter vi oss av rekursive kall i stedet for malloc (minne allokering) og funskjonskallene get_even og get_odd. I disse kallene sender vi med parametrene N/2 og 2*s (altså 2*stride-faktoren). Radix-2 tar først seg av par-indekserte inputs, deretter odde-indekserte inputs, for å til slutt kombinere disse to. I den naive implementasjonen hadde vi funksjoner som get_odd og get_even, som vi i prinsippet ikke trenger i Cooley-Tukey. Algoritmen får sin effektivitet fra gjenbruken av resultat fra mellomregninger i rekursjonene. Vi går ikke dypere inn på detaljene her, i og med at hvordan algoritmen tar for seg utregningene er ganske komplisert.

```
X_{0,...,N-1} ← ditfft2(x, N, s):              DFT of (x_0, x_s, x_2s, ...,
x_(N-1)s):
    if N = 1 then
        X_0 ← x_0                                          trivial size-1
DFT base case
    else
        X_{0,...,N/2-1} ← ditfft2(x, N/2, 2s)              DFT of (x_0, x_2s,
x_4s, ...)
        X_{N/2,...,N-1} ← ditfft2(x+s, N/2, 2s)            DFT of (x_s,
x_s+2s, x_s+4s, ...)
        for k = 0 to N/2-1 do                              combine DFTs of
two halves into full DFT:
            p ← X_k
            q ← exp(-2πi/N k) X_{k+N/2}
            X_k ← p + q
            X_{k+N/2} ← p - q
        end for
    end if
```

**Gjennomgang av pseudokode:**

Her ser vi en tredje parameter, nemlig s. S står for "stride of an array". Stride er en indikasjon på hvor mange bytes det er mellom starten av to suksessive elementer i en array. Den naive implementasjonen tar for seg odde og par elementer som kommer etter hverandre. Med stride implementert har vi muligheten til å hente ut disse direkte, ved at vi vet nøyaktig hvor neste verdi befinner seg. Første even er plassert på (in) mens første odd er plassert på (in+s). I tillegg dobler vi stride-verdien for hvert kall. Da unngår vi å måtte bruke tid og plass på minneallokering og kalkuleringer av odde/par i funksjonskall. Alt dette gjøres i de nye rekursive kallene.


Alt i alt vil Cooley-Tukey teoretisk sett gjøre FFT raskere fordi den bruker rekursjon som gjør at vi kan kvitte oss med unødvendige funskjoner (som get_odd og get_even), og den benytter seg av en stridefaktor. Som forklart over gjør det kjøringen mindre tids- og plasskrevende.

**Endringer som burde gjøres:**

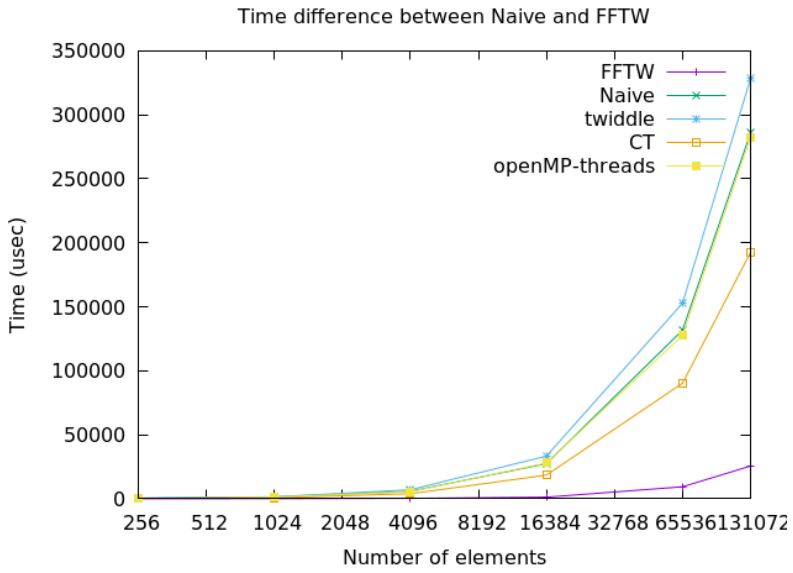
I algoritmen kjøres det rekursive kall i stedet for minneallokering (som det ble gjort i det naive eksempelet av FFT). For å implementere algoritmen trenger vi i tillegg å legge til en ekstra parameter i funskjonskallet, s, som vi setter lik 1 (s = 1) i hovedprogrammet. I tillegg kan vi kvitte oss med funskjonskall som get_odd og get_even, nettopp fordi rekursjonskallet tar seg av odde og par operasjoner. Det par-rekursive kallet tar inn (in, out, half, 2s) mens det odde-rekursive kallet tar inn (in+s, out+half, half, 2s). De største endringene blir da i odde-kallet. Vi dropper altså to funksjonskall og 4 minneallokeringer, noe som sparer tid og arbeid for datamaskinen.

**Utsnitt av koden implementert:**

```
// Include for math functions and definition of PI
#include <math.h>
// Included to get access to `malloc` and `free`
#include <stdlib.h>

void fft_compute(const complex* in, complex* out, const int n, int const s) {
    if(n == 1) {
        out[0] = in[0];
    } else {
        const int half = n / 2;
        // Recursively calculate the result for bottom and top half
        fft_compute(in, out, half, 2*s);
        fft_compute(in+s, out+half, half, 2*s); //Possible logical error with in+s, as
        // Combine the output of the two previous recursions
        for(int i = 0; i < half; ++i) {

            // q becomes the symbol for twiddle factors
            const complex e = out[i];
            const complex o = out[i + half];
            const complex w = cexp(0 - (2. * M_PI * i) / n * I); // May have to change
            out[i]        = e + w * o;
            out[i + half] = e - w * o;
    }
}
}

```

**Tidsbruk:**

Vi ser at CT-implementasjonen reduserer tidsforbruket drastisk. CT gir en tidsreduksjon på ca 37.5%. Dette stemmer med forventningen vi hadde om implementasjonen.



# Teoretisk implementasjon av twiddle:

**Kjapt om Twiddle-faktoren:**

Twiddle faktor, representert som w, er verdier som brukes for å gjøre FFT algoritmen raskere. I følge Twiddle factors in DSP for calculating DFT, FFT and IDFT (technobyte.org) har algoritmen uten twiddle faktor en tidskompleksitet på O(n^2), mens når det implementeres, vil vi få O(N log N). I den naive løsningen finner vi twiddle-faktoren. Denne regnes ut hver gang for-løkken kjøres. Ifølge oppgaveteksten vil et forslag være å cache verdiene for twiddle-faktoren før vi henter den ut

**Litt om hvordan cache fungerer (How Caching Works | HowStuffWorks):**

Datamaskiner har som oftest 2-3 "lag" med cacher. Disse kalles for L1, L2 og L3. Når vi skal hente ut en verdi fra minnet sjekker datamaskinen cache først. Først sjekkes L1, så L2 og til slutt L3. Hvis vi finner verdien vi leter etter i for eksempel L1, får vi en cache-hit. Dersom den ikke er i L1 får vi et cache-miss, og L2 sjekkes, osv. Datamaskinen sjekker stegvis om verdien vi skal hente ut er i de ulike "lagene" med minnet. Nytt eksempel: la oss si at verdien vi skal hente ut ligger i harddisken. Da vil datamaskinen først sjekke cachene, RAM og så til slutt harddisken. I dette tilfellet vil det å sjekke alle lagene med cache gjøre prosessen tregere. Som oftest er ikke dette et stort problem. L1 befinner seg nær cpuen, og L3 lengst fra cpuen (av alle lagene). To av tingene som gjør cache kjapt er at det er nær cpu-en, altså lett tilgjengelig, og at det er lite plass i forhold til for eksempel RAM. Dette gjør at det er færre verdier å sjekke, som igjen gir lite tidsbruk. Til slutt er det viktig å nevne at cache lagrer verdier etter behov. Forklart vil dette si at er det data vi bruker eller henter ofte; legges det i cachen for å redusere tidsbruken, og at det tar kortest tid før vi får en cache hit.

**Caching av twiddlefaktoren:**

```
void fft_compute(const complex* in, complex* out, const int n) {
    if(n == 1) {
        out[0] = in[0];
    } else {
        const int half = n / 2;
        // First we declare and allocate arrays
        // Allocate enough room for half the input values
        complex* even = malloc(sizeof(complex) * half);
        complex* odd  = malloc(sizeof(complex) * half);
        complex* even_out = malloc(sizeof(complex) * half);
        complex* odd_out  = malloc(sizeof(complex) * half);
        // Extract even and odd indexed numbers using methods above
        get_even(in, even, n);
        get_odd(in, odd, n);
        // Recursively calculate the result for bottom and top half
        fft_compute(even, even_out, n / 2);
        fft_compute(odd, odd_out, n / 2);
        // Combine the output of the two previous recursions
        for(int i = 0; i < half; ++i) {
            const complex e = even_out[i];
            const complex o = odd_out[i];
            const complex w = cexp(0 - (2. * M_PI * i) / n * I);
            out[i]        = e + w * o;
            out[i + half] = e - w * o;
        }
```

La oss se på caching av twiddle-faktoren. Twiddle-faktoren beregnes hver gang vi kjører for-løkken. Det vil si at hver gang for-løkken kjøres vil den bruke ekstra tid på å regne ut w. La oss se på even_out og odd_out. Her ser vi at disse tildeles minne malloc og verdiene beregnes før for-løkken, som vil si at datamaskinen ikke bruker masse tid på beregningene under for-løkken. Med andre ord kommer effektiviteten av at even_out (e) og odd_out (o) er beregnet på forhånd. Dermed kan vi si at disse verdiene er "cachet". De er lett tilgjengelige i minnet (vi vet ikke nøyaktig hvor i minnet, men mest sannsynlig i et av cache-lagene eller i RAM). Dersom vi ser på twiddle-faktoren (w), ser vi at det er en komplisert beregning som på liten skala tar kort tid, mens på stor skala kan ta lang tid. Velger vi å cache disse tre verdiene (e, o og w) før vi kjører for-løkken, vil det redusere tidsbruken, og effektivisere algoritmen sin helhet. Hvordan disse verdiene regnes ut og settes inn i array er avgjørende for kjøretiden.

**Baksiden ved å implementere denne forbedringen:**

For det første har vi en tanke rundt den såkalte "cachingen". Vi vil ha twiddle-faktorene ferdig utregnet og rask tilgjengelig. Det er ingen garanti på at verdiene faktisk legges i de første minne-lagene. Altså hadde det vært raskest og mer optimalt om de ble lagt direkte inn i L1 cache, men det er ingen garanti for at programmet eksekverer dette for oss. Helst ønsker vi færrest cache miss. En annen tanke er at vi utfører mange minne-allokeringer. Malloc kjøres allerede 4 ganger per rekursjon, og dersom vi hadde lagt til en ekstra malloc for twiddle-faktoren, ville dette igjen ta lengre tid og være plasskrevende. I tillegg må alle twiddle-faktorene beregnes på forhånd. Dette krever også tid og mange itereringer, og blir en tilnærmet lik prosess som i den naive implementasjonen. Dersom implementasjonen har en raskere måte å regne ut disse faktorene, i tillegg til at vi har garanti for at verdiene faktisk caches optimalt, vil caching twiddle i teorien gi raskere algoritme.

SLUTT