fft: optimizations

Vladimir Smykalov

Toulouse, 2017

Non-recursion realization

let's look again at recursive realization

```
def fft(a, N): # computes values of polynomial (sum a_i * x^i) in roots of x^N - 1 = 0
    if N == 1:
        return [a[0]]

    # split a to a_odd and a_even
    a_even = [a[0], a[2], ...]
    a_odd = [a[1], a[3], ...]

    # run fft recursively
    f_even = fft(a_even, N/2)
    f_odd = fft(a_odd, N/2)

    # reconstruct f values
    for i in 0 .. N/2-1:
        f[i] =     f_even[i] + z[i] * f_odd[i]
        f[i+N/2] = f_even[i] + z[i+N/2] * f_odd[i]

    return f
```

Non-recursion realization

In which order elements are are used?

```
id                          binary            reversed binary
0                       <-- 00000000000    <-- 00000000000
2^{k-1}                 <-- 10000000000    <-- 00000000001
2^{k-2}                 <-- 01000000000    <-- 00000000010
2^{k-1}+2^{k-2}         <-- 11000000000    <-- 00000000011
```

... in reversed-binary order!

Non-recursion realization

Let's first learn to quickly reverse bits in number

```
rev[0] = 0
for i in 1 .. N-1:
    rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (logN - 1))
```

Non-recursion realization

Now, let's write code that will run all calculations of `fft`-tree from bottom to top

```
fft(a, f): # calculate results of A and store in F
    # reversed order
    for i in 0 .. N-1:
        f[i] = a[rev[i]]

    for (k = 1; k < N; k = k * 2):
        for (i = 0; i < N; i = i + 2 * k):
            for j in 0 .. k-1:
                z = root(2*PI*j/(2*k)) * f[i + j + k]
                f[i + j + k] = f[i + j] - z
                f[i + j] = f[i + j] + z
```

`fft` became much shorter

But how to quickly get `root(...)` ?

It's too slow to run every time `cos` and `sin`

Let's precalculate them once!

```
for i in 0 .. N-1:
    alp = i * 2 * PI / N
    root[i] = (cos(alp), sin(alp))
```

Now just use `root[j * (N/(2*k))]` instead of `root(2*PI*j / (2*k))`

roots: `hardcore` level

Inside `fft` we have

```
        ...
            for j in 0 .. k-1:
                z = root[j * (N/(2*k))] * f[i + j + k]
                f[i + j + k] = f[i + j] - z
                f[i + j] = f[i + j] + z
```

This access `root[j * (N/(2*k))]` provides to much memory jumps and is not cache-efficient

roots: `hardcore` level

We can fix it by re-ordering roots.

First, let's notice that we don't use roots with `alp >= PI`

Now, let's set `root[k .. 2*k-1]` to upper roots order `2*k` (from `2*PI*0 / (2*k)` to `2*PI*(k-1) / (2*k)`)

Easy initialization:

```
for i in 0 .. N/2-1:
    alp = 2*PI*i / N
    root[i+N/2] = (cos(alp), sin(alp))
for (i = N/2-1; i >= 1; i = i - 1):
    root[i] = root[2 * i]
```

roots: `hardcore` level

Now we can use it in `fft` as pretty as it can be

```
    ...
        for j in 0 .. k-1:
            z = root[j + k] * f[i + j + k]
            f[i + j + k] = f[i + j] - z
            f[i + j] = f[i + j] + z
```

cache-efficient now, no memory jumps! 🚀🚀🚀

roots: `ultra hardcore` level

We still have `O(N)` evaluations of `cos` and `sin`.

We can reduce this number to `O(log)` almost **without loss** of precision!

Bad solution:

```
root[N/2] = (1, 0)
root[N/2+1] = (cos(2*PI/N), sin(2*PI/N))
for i in 2 .. N/2-1:
    root[N/2+i] = root[N/2+i-1] * root[N/2+1]
...
```

Calculating just one root and powering it is **very-very** bad!

That's the reason of most precision errors in `fft`-implementatoins!

Even implementation on `e-maxx.ru/algo` has this error!

roots: `ultra hardcore` level

Good solution:

```
root[1] = (1, 0)
for k in 1 .. logN-1:
    alp = 2 * PI / (1 << (k+1))
    z = (cos(alp), sin(alp))
    for i in (1 << (k-1)) ..  (1 << k)-1:
        root[2 * i] = root[i]
        root[2 * i + 1] = root[i] * z
```

Now each root is calculated by multiplication of at most `logN` other roots.

Suprisingly, this is almost as accurate as `O(n)` evaluations of `(cos, sin)`.

Tested on `N=2^20`, error is just `5.5511e-016` which is approximate `double-error`.

2-in-1 trick

Usual workflow of `fft`:

```
mult(a, b):
    ....
    fft(a, f)
    fft(b, g)
    for i in 0 .. N-1:
        h[i] = f[i] * g[i] / N
    reverse(h + 1, h + N)
    fft(h, c)
    ....
```

If `a` and `b` are real-value arrays, then we can **merge** them into one
(this will reduce total number of `fft`-runs from `3` to `2`)

2-in-1 trick

Let's set `IN[i] = (a[i], b[i])` (`a[i]` as real part and `b[i]` as image part)

```
fft(IN, OUT)
```

But how to reconstruct `f` and `g` from `OUT`?

`OUT(z^k) = f(z^k) + i * g(z^k)`
`OUT(z^-k) = f(z^-k) + i * g(z^-k)`

Let's run `conj(...)` to second equality

`conj(OUT(z^-k)) = conj(f(z^-k)) + conj(i * g(z^-k))`
`conj(OUT(z^-k)) = f(conj(z^-k)) - i * g(conj(z^-k))`
`conj(OUT(z^-k)) = f(z^k) - i * g(z^k)`

So, `f(z^k) = (OUT(z^k) + conj(OUT(z^-k))) / 2`
Same way, `g(z^k) = (OUT(z^k) - conj(OUT(z^-k))) / 2i`

2-in-1 trick

```
mult(a, b):
    IN = [(a[0], b[0]), (a[1], b[1]), ...]
    fft(IN, OUT)

    for i in 0 .. N-1:
        reconstruct f and g
        h[i] = f[i] * g[i] / N
    ...
```

... but what is `f[i] * g[i]` ?

`let j be (N-i) & (N-1)` (it's like `-i`)

`f[i] * g[i] = (OUT[i] + conj(OUT[j])) / 2 * (OUT[i] - conj(OUT[j])) / 2i`

`f[i] * g[i] = (OUT[i] * OUT[i] - conj(OUT[j] * OUT[j])) / 4i`

2-in-1 trick

Updated workflow

```
mult(a, b):
    IN = [(a[0], b[0]), (a[1], b[1]), ...]
    fft(IN, OUT)
    for i in 0 .. N-1:
        j = (N-i) & (N-1)
        h[i] = (OUT[i]*OUT[i] - conj(OUT[j]*OUT[j])) * (0, -0.25 / N)
    reverse(h + 1, h + N)
    fft(h, c)
```

2-in-1 trick

minor fix (`reverse` moved inside `for`)

```
mult(a, b):
    IN = [(a[0], b[0]), (a[1], b[1]), ...]
    fft(IN, OUT)
    for i in 0 .. N-1:
        j = (N-i) & (N-1)
        h[i] = (OUT[j]*OUT[j] - conj(OUT[i]*OUT[i])) * (0, -0.25 / N)
    fft(h, c)
```

Now let's move to `fft-mod`

Previously described method allows multiplication of `int` numbers if number in output is no more than `10^15` (due to precision problems)

Switching from `double` to `long double` doesn't really helps!

What to do if we need to multiply polynomials modulo `10^9+7` ?

Let's use old `fft` to solve `fft-mod`

Suppose `s` is approx. $\sqrt{mod}$

Then we can divide `a` to `a_small` and `a_large`, like this:
`a = a_small + S * a_large`, so both `a_small` and `a_large` are less then `S`

Now multiplication `f * g` can be solved in `4` `double multiplications`-s, that makes a total of `12` `fft` (or `8` if using `2-in-1` optimization)

That amount can be reduced to just `4` `fft`-runs!

fft-mod

Workflow:

- run `fft` on pairs `(a_small[i], a_large[i])`
- run `fft` on pairs `(b_small[i], b_large[i])`
- reconstruct `(f_small[i], f_large[i])` and `(g_small[i], g_large[i])`
- let `h0 = f_small * g_small`
- let `h1 = f_small * g_large + f_large * g_small`
- let `h2 = f_large * g_large`
- run inverse `fft` on `h0 + i * h1`
- run inverse `fft` on `h2`

fft-mod

about this:

- run inverse `fft` on `h0 + i * h1`

This is a `2-in-1 merge` in inverse `fft` !

If we run inverse `fft(f + i*g, OUT)` we will have `OUT = (a, b)` iff `a` and `b` are real

fft-mod

That all makes it just `4` `fft`-runs to multiply to polynomials over ANY module!

... isn't that awesome?

To community also known `fft-int` method that allows to run `fft` completly in integer numbers, but that algo works only on `mod = x * 2^k + 1` and works approx. the same time as `fft-mod`