DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 39

1. COMPILING RUST TO WEBASSEMBLY
2. BONUS: FAST ALGORITHM FOR COMPUTING FIBONACCI NUMBERS

OFFICE HOURS

- Today: 4:30–6 (will finish slightly early)

OFFICE HOURS

- Today: 4:30–6 (will finish slightly early)
- Proposed final project coding jams:
  - Today: 7–???
  - Thursday: 8–???
  - (and another one on Monday?)

[REVIEW OF SOLUTIONS TO HOMEWORKS 9 & 10]

1. COMPILING RUST TO WEBASSEMBLY

2. BONUS: FAST ALGORITHM FOR COMPUTING FIBONACCI NUMBERS

RUNNING ARBITRARY CODE IN A BROWSER

Traditional way:

- JavaScript
- Java
- Adobe Flash

Shortcomings (various degrees):

- far from native speed
- portability
- many other

WEBASSEMBLY

- WebAsembly ≡ "assembler" for the web
- Near–native speed
- Clear definition
- Every major web browser can run it
- Can be compiled to from many languages, including Rust

WEBASSEMBLY

- WebAsembly ≡ "assembler" for the web
- Near–native speed
- Clear definition
- Every major web browser can run it
- Can be compiled to from many languages, including Rust

Additional features:

- memory directly accessible to JavaScript (to avoid foreign function interface translation)
- no garbage collection
  - adding it is being considered
  - cannot be compiled directly into from languages that use garbage collection

HOW DOES ONE USE WEBASSEMBLY IN PRACTICE?

- It does not replace JavaScript completely
- Find out which parts of your code are slow
- Rewrite them in Rust and compile to WebAssembly
- Use JavaScript on your webpage to interact with WebAssembly binaries

HOW TO DEPLOY IT

Some basics:

- Write a library in Rust
- Compile as a dynamic library
- Set the compilation target ("architecture") to `wasm32-unknown-unknown`
- Library binaries: `.wasm`
- See the tutorial how to make Rust code and JavaScript talk to each other

Tutorial: https://rustwasm.github.io/docs/book/

1. COMPILING RUST TO WEBASSEMBLY

2. BONUS: FAST ALGORITHM FOR COMPUTING FIBONACCI NUMBERS

FIBONACCI NUMBERS: ALGORITHMS WE SAW SO FAR

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-2} + F_{k-1} & \text{if } k > 1 \end{cases}$$

Assuming $O(1)$ time arithmetic operations:

- $O(F_k)$ time by directly recursively following the definition
- $O(k)$ time by storing values and computing F_k from F_{k-2} and F_{k-1}

COMPUTING VIA MATRIX OPERATIONS

Useful matrix

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

COMPUTING VIA MATRIX OPERATIONS

Useful matrix

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Observation:

$$A \begin{bmatrix} F_k \\ F_{k-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_k \\ F_{k-1} \end{bmatrix} = \begin{bmatrix} F_{k+1} \\ F_k \end{bmatrix}$$

COMPUTING VIA MATRIX OPERATIONS

Useful matrix

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Observation:

$$A \begin{bmatrix} F_k \\ F_{k-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_k \\ F_{k-1} \end{bmatrix} = \begin{bmatrix} F_{k+1} \\ F_k \end{bmatrix}$$

Hence, by induction:

$$A^k \begin{bmatrix} 1 \\ 0 \end{bmatrix} = A^k \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} F_{k+1} \\ F_k \end{bmatrix}$$

COMPUTING VIA MATRIX OPERATIONS

Useful matrix

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Observation:

$$A \begin{bmatrix} F_k \\ F_{k-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_k \\ F_{k-1} \end{bmatrix} = \begin{bmatrix} F_{k+1} \\ F_k \end{bmatrix}$$

Hence, by induction:

$$A^k \begin{bmatrix} 1 \\ 0 \end{bmatrix} = A^k \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} F_{k+1} \\ F_k \end{bmatrix}$$

Can we compute A^k efficiently?

EXPONENTIATION BY SQUARING

Halving the exponent:

Even k:

- Compute recursively $A_\star = A^{k/2}$
- Return $A_\star^2$

Odd k:

- Compute recursively $A_\star = A^{(k-1)/2}$
- Return $A_\star^2 \times A$

EXPONENTIATION BY SQUARING

Halving the exponent:

Even k:

- Compute recursively $A_\star = A^{k/2}$
- Return $A_\star^2$

Odd k:

- Compute recursively $A_\star = A^{(k-1)/2}$
- Return $A_\star^2 \times A$

Total: $O(\log k)$ arithmetic operations

EXPONENTIATION BY SQUARING

Halving the exponent:

Even k:

- Compute recursively $A_\star = A^{k/2}$
- Return $A_\star^2$

Odd k:

- Compute recursively $A_\star = A^{(k-1)/2}$
- Return $A_\star^2 \times A$

Total: $O(\log k)$ arithmetic operations

Let's implement it!

AVOIDING BIG NUMBERS

- To avoid dealing with big numbers, let's just compute it modulo a large prime
- First, we have to find a big prime

AVOIDING BIG NUMBERS

- To avoid dealing with big numbers, let's just compute it modulo a large prime
- First, we have to find a big prime

In [2]:
```
let lower_bound = 1_234_567_890_000u128;
(lower_bound..2*lower_bound)
    .filter(
        |x| (2..*x)
            .take_while(|y| y * y <= *x)
            .all(|y| *x % y != 0)
     ).next()
```

Out[2]:
```
Some(1234567890007)
```

AVOIDING BIG NUMBERS

- To avoid dealing with big numbers, let's just compute it modulo a large prime
- First, we have to find a big prime

In [2]:
```
let lower_bound = 1_234_567_890_000u128;
(lower_bound..2*lower_bound)
    .filter(
        |x| (2..*x)
            .take_while(|y| y * y <= *x)
            .all(|y| *x % y != 0)
     ).next()
```

Out[2]: Some(1234567890007)

In [3]:
```
const BIG_PRIME: u128 = 1_234_567_890_007;
```

"SLOW" $O(k)$-TIME IMPLEMENTATION

In [4]:
```
fn fib_mod_linear(x: u128) -> u128 {
    if x == 0 {
        return 0;
    }
    let mut y = 1;
    let mut fib_prev = 0;
    let mut fib = 1;
    // invariant:
    //   * fib_prev == F(y - 1) mod BIG_PRIME
    //   * fib == F(y) mod BIG_Prime
    while y < x {
        y += 1;
        (fib_prev,fib) = (fib,(fib_prev+fib) % BIG_PRIME)
    }
    return fib
}
```

MATRIX OPERATIONS

In [5]:

```
// Matrix shape
// 0 1
// 2 3

type MyMatrix = [u128;4];

const A: MyMatrix = [1,1,1,0];

fn multiply(x: MyMatrix, y: MyMatrix) -> MyMatrix {
    let mut solution = [0;4];
    solution[0] = x[0] * y[0] + x[1] * y[2];
    solution[1] = x[0] * y[1] + x[1] * y[3];
    solution[2] = x[2] * y[0] + x[3] * y[2];
    solution[3] = x[2] * y[1] + x[3] * y[3];
    solution.iter_mut().for_each(|x| *x = *x % BIG_PRIME);
    solution
}
```

IMPLEMENTATION OF THE FAST ALGORITHM

In [6]:

```
// exponentiation of A by squaring (module BIG_PRIME)
fn exponentiate_fib_matrix(exponent: u128) -> MyMatrix {
    if exponent == 0 {
        return [1,0,0,1];
    }
    let tmp = exponentiate_fib_matrix(exponent / 2);
    if exponent % 2 == 0 {
        multiply(tmp, tmp)
    } else {
        multiply(multiply(tmp, tmp), A)
    }
}
```

IMPLEMENTATION OF THE FAST ALGORITHM

In [6]:

```
// exponentiation of A by squaring (module BIG_PRIME)
fn exponentiate_fib_matrix(exponent: u128) -> MyMatrix {
    if exponent == 0 {
        return [1,0,0,1];
    }
    let tmp = exponentiate_fib_matrix(exponent / 2);
    if exponent % 2 == 0 {
        multiply(tmp, tmp)
    } else {
        multiply(multiply(tmp, tmp), A)
    }
}
```

In [7]:

```
// Fibonacci computation
fn fib_mod_logarithmic(x: u128) -> u128 {
    if x == 0 {
        0
    } else {
        let matrix = exponentiate_fib_matrix(x - 1);
        matrix[0]
    }
}
```

BENCHMARKING

In [8]:

```
use std::time::SystemTime;
// see how long something is executing
fn time_it(f: impl FnOnce() -> u128) {
    let before = SystemTime::now();
    let result = f();
    let after = SystemTime::now();
    println!("Time: {:.3?}", after.duration_since(before).unwrap());
    println!("Computed number: {}\n", result);
}
```

BENCHMARKING

In [8]:

```
use std::time::SystemTime;
// see how long something is executing
fn time_it(f: impl FnOnce() -> u128) {
    let before = SystemTime::now();
    let result = f();
    let after = SystemTime::now();
    println!("Time: {:.3?}", after.duration_since(before).unwrap());
    println!("Computed number: {}\n", result);
}
```

In [10]:

```
let k: u128 = 10;

time_it(|| fib_mod_linear(k));

time_it(|| fib_mod_logarithmic(k));
```

```
Time: 970.000ns
Computed number: 55

Time: 1.651µs
Computed number: 55
```

BENCHMARKING

In [8]:

```
use std::time::SystemTime;
// see how long something is executing
fn time_it(f: impl FnOnce() -> u128) {
    let before = SystemTime::now();
    let result = f();
    let after = SystemTime::now();
    println!("Time: {:.3?}", after.duration_since(before).unwrap());
    println!("Computed number: {}\n", result);
}
```

In [11]:

```
let k: u128 = 1000;

time_it(|| fib_mod_linear(k));

time_it(|| fib_mod_logarithmic(k));
```

```
Time: 13.500µs
Computed number: 202736284353

Time: 3.105µs
Computed number: 202736284353
```

BENCHMARKING

In [8]:

```
use std::time::SystemTime;
// see how long something is executing
fn time_it(f: impl FnOnce() -> u128) {
    let before = SystemTime::now();
    let result = f();
    let after = SystemTime::now();
    println!("Time: {:.3?}", after.duration_since(before).unwrap());
    println!("Computed number: {}\n", result);
}
```

In [14]:

```
let k: u128 = 1_000_000;

time_it(|| fib_mod_linear(k));

time_it(|| fib_mod_logarithmic(k));
```

```
Time: 12.549ms
Computed number: 863350906745

Time: 5.494µs
Computed number: 863350906745
```

BENCHMARKING

In [8]:
```
use std::time::SystemTime;
// see how long something is executing
fn time_it(f: impl FnOnce() -> u128) {
    let before = SystemTime::now();
    let result = f();
    let after = SystemTime::now();
    println!("Time: {:.3?}", after.duration_since(before).unwrap());
    println!("Computed number: {}\n", result);
}
```

In [16]:
```
let k: u128 = 1_000_000_000;

time_it(|| fib_mod_linear(k));

time_it(|| fib_mod_logarithmic(k));
```

```
Time: 12.108s
Computed number: 129171585224

Time: 8.383µs
Computed number: 129171585224
```