Lesson 2. Foundation of Algorithms: Importance of Analysis

I111E – Algorithms and Data Structures

Ryuhei Uehara & Giovanni Viglietta
uehara@jaist.ac.jp & johnny@jaist.ac.jp

JAIST – October 21, 2019

All material is available at
www.jaist.ac.jp/~uehara/couse/2019/i111e

I111E – Q&A time

Question:
What if I am not at ease with C? Can I still take this course?
Am I required to write projects or exams in C?
Can I use Python/Java/C# instead?

I111E – Q&A time

Question:
What if I am not at ease with C? Can I still take this course?
Am I required to write projects or exams in C?
Can I use Python/Java/C# instead?

Answer:
This course is not about programming: it is about algorithms.
So, you are not required to write *any* code in this course.
You can describe an algorithm in any language you want: either a computer language of your choice, or in English (or Japanese), or even in a pseudo-code that you invented!
If you can understand the C code in the slides and the mathematics behind it, you are good to go. :)

Goals of today's lecture

- Understand the importance of designing efficient algorithms
- Familiarize with big-O notation, e.g., $5n^2 + 3n + 6 = O(n^2)$
- Learn how to analyze the complexity of an algorithm

Algorithms and programs

The word "algorithm" comes from the Latinization of the Persian mathematician Al-Khwarizmi.

An algorithm is an abstract description of how to solve a problem:

- it halts on every input,
- it returns the correct answer on every input,
- its description is not ambiguous (operations are well defined).

A program is the description of an algorithm in some computer language: the same algorithm can be *implemented* in many languages.

Design of good algorithms

Approach for designing a good algorithm:

- There are some general *design methods*
  (divide and conquer, greedy, dynamic programming, ...)
- Estimate the time complexity (running time) and the space complexity (amount of memory used) of the algorithm
- Verify and prove the correctness of the algorithm

Bad algorithm:

- Instant idea: no design method
- Just made it: no analysis of complexity and/or correctness

The Collatz function

Recall from Lesson 1...

```
void collatz(int n) {
    printf("%d\n", n);
    if (n == 1) return;
    if (n%2 == 0) collatz(n/2);
    else collatz(3*n + 1);
}
```

The Collatz function

Recall from Lesson 1...

```
void collatz(int n) {
    printf("%d\n", n);
    if (n == 1) return;
    if (n%2 == 0) collatz(n/2);
    else collatz(3*n + 1);
}
```

`collatz(5)` calls
`collatz(16)`, which calls
`collatz(8)`, which calls
`collatz(4)`, which calls
`collatz(2)`, which calls
`collatz(1)`, which returns.

The factorial function

Let's compute the factorial function:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

Equivalently,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{otherwise} \end{cases}$$

The factorial function

Let's compute the factorial function:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

Equivalently,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{otherwise} \end{cases}$$

```
int fact(int n) {
    if (n == 0) return 1;
    return fact(n-1) * n;
}
```

The Fibonacci sequence

$$0, \quad 1, \quad 1, \quad 2, \quad 3, \quad 5, \quad 8, \quad 13, \quad 21, \quad 34, \quad 55, \quad 89, \quad \ldots$$

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

The Fibonacci sequence

$$0,\ \ 1,\ \ 1,\ \ 2,\ \ 3,\ \ 5,\ \ 8,\ \ 13,\ \ 21,\ \ 34,\ \ 55,\ \ 89,\ \ \ldots$$

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

The Fibonacci sequence: running time

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

Problem: on my computer, `fib(50)` takes more than a minute...

And `fib(100)` would take more than 30,000 years on today's fastest computer!

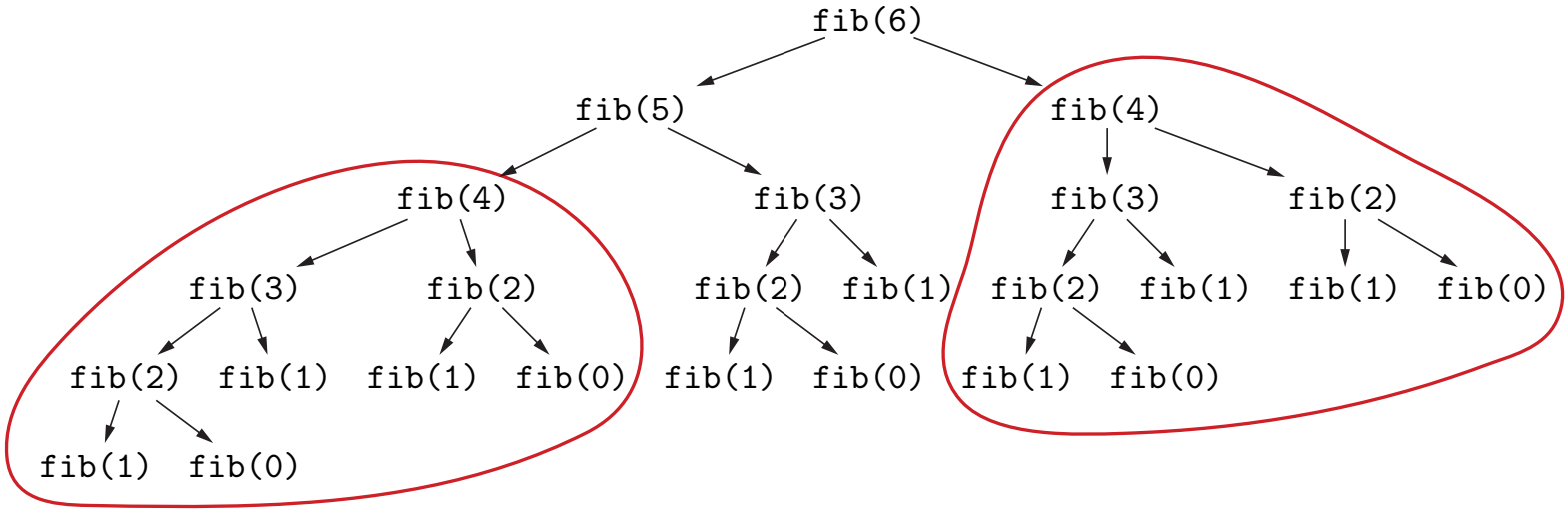
But a human can easily compute F_{100} by hand in a few hours!

Weren't computers supposed to be faster than people??

The Fibonacci sequence: running time

It turns out that we used a very inefficient algorithm:

each call to `fib` calls `fib` again, twice.



This algorithm re-computes the same numbers over and over!

The Fibonacci sequence: a better version

What would a human do instead?

Start from the bottom: write down F_0, F_1, F_2, F_3, and compute the next Fibonacci number by looking up the last two.

This way, each Fibonacci number is computed just once!

The Fibonacci sequence: a better version

What would a human do instead?

Start from the bottom: write down F_0, F_1, F_2, F_3, and compute the next Fibonacci number by looking up the last two.

This way, each Fibonacci number is computed just once!

```
int fib2(int n) {
    int f[n+1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
```

The Fibonacci sequence: a better version

What would a human do instead?

Start from the bottom: write down F_0, F_1, F_2, F_3, and compute the next Fibonacci number by looking up the last two.

This way, each Fibonacci number is computed just once!

```
int fib2(int n) {
    int f[n+1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
```

Problem: `fib2(1000000)` gives a "stack overflow" error!

We are using too much memory to store the Fibonacci numbers.

The Fibonacci sequence: an even better version!

What can we do to use less memory?

We only ever need the last two Fibonacci numbers to compute the next one, so we do not have to store them all!

```
int fib3(int n) {
    int last1 = 0;
    int last2 = 1;
    for (int i = 0; i < n; i++) {
        int next = last1 + last2;
        last1 = last2;
        last2 = next;
    }
    return last1;
}
```

Big-O notation

When we reason about the efficiency of an algorithm, we want to abstract from the actual implementation details, programming language, and machine on which it is executed.

All these elements introduce speedups or slowdowns by constant factors only (e.g., accessing a C++ array on a PC is 1.3 times faster than accessing a Java array on a smartphone).

For the essence of an algorithm, these factors do not matter.

Big-O notation

When we reason about the efficiency of an algorithm, we want to abstract from the actual implementation details, programming language, and machine on which it is executed.

All these elements introduce speedups or slowdowns by constant factors only (e.g., accessing a C++ array on a PC is 1.3 times faster than accessing a Java array on a smartphone).

For the essence of an algorithm, these factors do not matter.

So, we will "identify" all functions that differ only by additive and multiplicative constants.

For example, $5n + 3$ is "the same" as $8n + 100$:

we say that both these functions are $O(n)$, because they are "the same" as n up to constant factors.

Comparing common functions

From smaller to larger, we have:

- **Constant:** $O(1)$ (e.g., 10)
- **Logarithmic:** $O(\log n)$
- **Linear:** $O(n)$ (e.g., $6n + 8$)
- **Quasi-linear:** $O(n \log n)$
- **Quadratic:** $O(n^2)$ (e.g., $3n^2 + 8n - 2$)
- **Cubic:** $O(n^3)$
- **Polynomial:** $O(n^c)$ (e.g., $n^{37} + 5n^{10} + 8$)
- **Exponential:** $O\left(c^n\right)$ (e.g., 2^{n+7})

Comparing common functions

From smaller to larger, we have:

- **Constant:** $O(1)$ (e.g., 10)
- **Logarithmic:** $O(\log n)$
- **Linear:** $O(n)$ (e.g., $6n + 8$)
- **Quasi-linear:** $O(n \log n)$
- **Quadratic:** $O(n^2)$ (e.g., $3n^2 + 8n - 2$)
- **Cubic:** $O(n^3)$
- **Polynomial:** $O(n^c)$ (e.g., $n^{37} + 5n^{10} + 8$)
- **Exponential:** $O\left(c^n\right)$ (e.g., 2^{n+7})

An algorithm with a quasi-linear running time is *practical*.

An algorithm with a polynomial running time is *feasible*.

Otherwise, it is *unfeasible*.

Logarithmic bases

Note: we do not have to specify the base of logarithms, because logarithms in different bases only differ by a constant factor:

$$\log_a n \quad = \quad \underline{\log_a b} \cdot \log_b n$$

So, $\log_a n = O(\log_b n)$.

For logarithms, we just write $O(\log n)$ without the base.

Running time of `fib`

For measuring running times, we use a "simplistic" model: each "elementary instruction" such as an assignment, an arithmetic operation, a Boolean test, etc. takes unit time.

Running time of `fib`

For measuring running times, we use a "simplistic" model: each "elementary instruction" such as an assignment, an arithmetic operation, a Boolean test, etc. takes unit time.

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

Let $T(n)$ be the running time of `fib(n)`:

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2 & \text{if } n = 1 \\ T(n-1) + T(n-2) + 5 & \text{otherwise} \end{cases}$$

Running time of `fib`

For measuring running times, we use a "simplistic" model: each "elementary instruction" such as an assignment, an arithmetic operation, a Boolean test, etc. takes unit time.

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

Let $T(n)$ be the running time of `fib(n)`:

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2 & \text{if } n = 1 \\ T(n-1) + T(n-2) + 5 & \text{otherwise} \end{cases}$$

It is now easy to show that $T(n) > F_n$.

But $F_n = O(1.62^n)$, so the running time of `fib` is exponential!

Running time of `fib2`

```
int fib2(int n) {
    int f[n+1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
```

We have 3 initial operations, plus a loop that is executed $n-1$ times, and each time it performs 6 elementary instructions: test for $i \leq n$, $i++$, $i-1$, $i-2$, addition, assignment.

The total running time is therefore $T(n) = 6(n-1)+3 = O(n)$.

We use an array of size $n+1$ plus the variable `i`, hence the total space is $n+2 = O(n)$.

Time and space are both linear.

Running time of `fib3`

```
int fib3(int n) {
    int last1 = 0;
    int last2 = 1;
    for (int i = 0; i < n; i++) {
        int next = last1 + last2;
        last1 = last2;
        last2 = next;
    }
    return last1;
}
```

We have 2 initial operations, plus a loop that is executed n times, each time performing 6 operations: $T(n) = 6n + 2 = O(n)$.

This time we only use 4 variables: $O(1)$ space.

`fib3` runs in linear time and constant space.

Is exponential time really bad?

Moore's law: the speed of computers doubles every 18 months.

Since the speed of computers increases exponentially, maybe in a couple of years we will be able to run `fib` in a reasonable time (?)

Is exponential time really bad?

Moore's law: the speed of computers doubles every 18 months.

Since the speed of computers increases exponentially, maybe in a couple of years we will be able to run `fib` in a reasonable time (?)

Unfortunately not:

Suppose today we can execute `fib(100)` in a reasonable time.

In 12 months, computers will be about 1.6 times faster.

But `fib(101)` takes about 1.6 times more than `fib(100)`!

So, next year we will only be able to execute `fib(101)`.

In 2 years, we will only be able to execute `fib(102)`.

In 3 years, we will only be able to execute `fib(103)`...

Only one Fibonacci number per year:
this is the "curse" of exponential running times! :(

Another example: exponentiation

Let us compute the function a^n.

Naive approach:

```
int expon(int a, int n) {
    int res = 1;
    for (int i = 1; i <= n; i++)
        res = res * a;
    return res;
}
```

This algorithm performs n multiplications and runs in time $O(n)$.

Can we do better?

Exponentiation: a better approach

What would we do if we had to compute a^{16}?

```
int a2 = a * a;
int a4 = a2 * a2;
int a8 = a4 * a4;
int a16 = a8 * a8;
return a16;
```

This performs only 4 multiplications:

much better than the 16 multiplications of the previous algorithm!

Can we generalize this observation?

Exponentiation: a better approach

To compute a^{20}, do this (observe that $a^{20} = a^4 \cdot a^{16}$):

```
int a2 = a * a;
int a4 = a2 * a2;
int a8 = a4 * a4;
int a16 = a8 * a8;
return a4 * a16;
```

Only 5 multiplications instead of 20!

What about a^n?

We have to look at the binary digits of n...

Binary digits

Every positive integer has a unique binary representation:

$$37 \;=\; 32+4+1 \;=\; \mathbf{1}\cdot 2^5+\mathbf{0}\cdot 2^4+\mathbf{0}\cdot 2^3+\mathbf{1}\cdot 2^2+\mathbf{0}\cdot 2^1+\mathbf{1}\cdot 2^0$$

$$37_{(10)} = 100101_{(2)}$$

Binary digits

Every positive integer has a unique binary representation:

$$37 \;=\; 32+4+1 \;=\; \mathbf{1}\cdot 2^5+\mathbf{0}\cdot 2^4+\mathbf{0}\cdot 2^3+\mathbf{1}\cdot 2^2+\mathbf{0}\cdot 2^1+\mathbf{1}\cdot 2^0$$

$$37_{(10)} = 100101_{(2)}$$

How do we compute the binary digits of a number?

Repeatedly check if the number is even or odd, and divide it by 2:

this gives its binary digits in reverse order.

```
do {
    printf("%d", n % 2);
    n = n / 2;
} while (n > 0);
```

Since $n = 2^{\log_2 n}$, the loop is repeated $O(\log n)$ times.

Exponentiation: a better approach

```
int a2 = a * a;
int a4 = a2 * a2;
int a8 = a4 * a4;
int a16 = a8 * a8;
return a4 * a16;
```

If $n = 20 = 2^2 + 2^4$, we have $n = 10100_{(2)}$.

The 1-digits of n correspond to $2^{\mathbf{2}}$ and $2^{\mathbf{4}}$...

And we multiplied $a^{2^{\mathbf{2}}}$ and $a^{2^{\mathbf{4}}}$.

Exponentiation: final algorithm

In general, if the ith binary digit of n is 1, multiply the result by a^{2^i}

```
int expon2(int a, int n) {
    int res = 1;
    int pow = a;
    do {
        if (n%2 == 1) res = res * pow;
        pow = pow * pow;
        n = n / 2;
    } while (n > 0);
    return res;
}
```

The loop is repeated $O(\log n)$ times,

so the total running time is $O(\log n)$.

A big improvement over the previous $O(n)$!