Introduction to Algorithms and Data Structures

2. Foundation of Algorithms (2) Simple Basic Algorithms

Professor Ryuhei Uehara,

School of Information Science, JAIST, Japan.

uehara@jaist.ac.jp

http://www.jaist.ac.jp/~uehara

http://www.jaist.ac.jp/~uehara/course/2020/myanmar/

Algorithm?

- Algorithm: abstract description of how to solve a problem (by computer)
  - It returns correct answer for any input
  - It halts for any input
  - Description is not ambiguity
    - (operations are well defined)

- Program: description of algorithm by some computer language
  - (Sometimes it never halt)

Al-Khwarizmi

Design of Good Algorithms

- There are some design method
- Estimate time complexity (running time) and space complexity (quantity of memory)
- Verification and Proof of Correctness of Algorithm

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

Goal of this morning

- 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

Examples:

SOME FUNCTIONS AND ALGORITHMS

The Collatz function

```
collatz(unsigned int n) {
  print(n);  // output n
  if (n == 1) return;
  if (n%2==0) collatz(n/2);
  else        collatz(3n+1);
}
```

- collatz(5) calls collatz(16), which calls collatz(8), … , collatz(1), which returns.

C.f.: Collatz conjectured that for *any* positive integer k, collatz(k) converges to 1, which is still open!

The factorial function

- Let's compute the factorial function:

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n$$

Equivalently,

$$n! = \begin{cases} 1 & \text{If } n = 0 \\ (n-1)! \times n & \text{Otherwise} \end{cases}$$

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

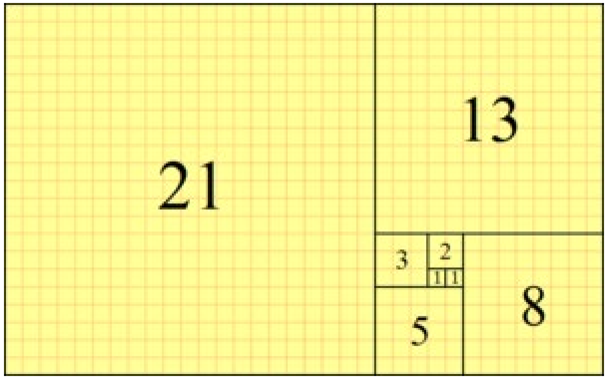
The Fibonacci sequence

- Let's compute the Fibonacci sequence:

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

Equivalently,

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



Check the Wikipedia for (interesting) Fibonacci sequence…

The Fibonacci sequence

- Let's compute the Fibonacci sequence:

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

Equivalently,

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

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

The Fibonacci sequence: computation time

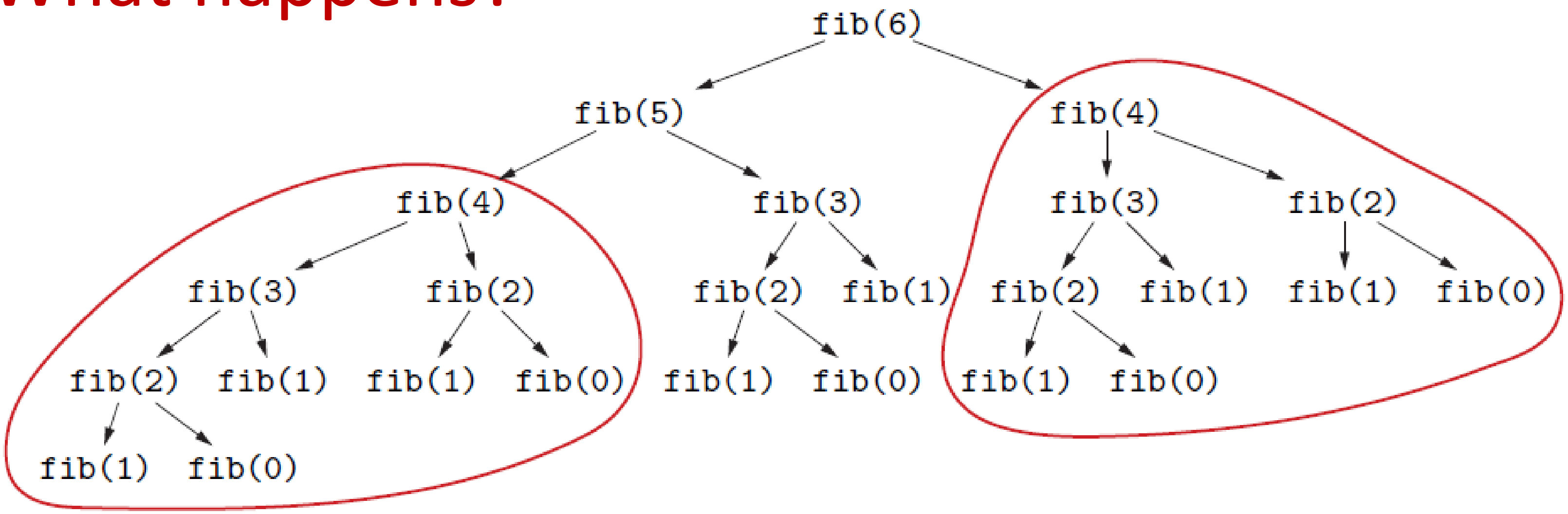
```
int fib(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1)+fib(n-2);
}
```

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!!

However, 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: computation time

What happens?



We used a very inefficient algorithm! Each call to fib calls fib again, twice or more. That is, the algorithm re-computes the same numbers over and over!!

The Fibonacci sequence: a better version

What would a human do instead?

We 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(unsigned 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?

```
int fib2(unsigned 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: on my computer, fib2(1000000) gives “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(unsigned 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;
}
```

Tool for estimation of algorithms:

BIG-O NOTATION

Big-O notation

Why we use 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 model on which it is executed.
- All these elements introduce speedups or slowdowns by constant factors only (e.g., accessing a C++ array on my PC is 2.5 times faster than accessing a Java array on your smartphone).
- For the essence of an algorithm, these factors do not matter.

Big-O notation

Why we use big-O notation?

- So, we will "identify" all functions that differs only by additive and multiplicative constants.
  - For example, $5n + 3$ is "the same" as $100n + 800$
  - We say that both these functions are $O(n)$, because they are "the same" as n up to constant factors.

Representative functions in big-O notation

- **Constant**: $O(1)$ (E.g., 10)
- **Logarithmic**: $O(\log n)$ (E.g., $3\log n + 23$)
- **Linear**: $O(n)$
- **Quasi-linear**: $O(n \log n)$
- **Quadratic**: $O(n^2)$
- **Cubic**: $O(n^3)$
- **Polynomial**: $O(n^c)$ (E.g., $35n^{80} + 800n^{20} + 23n^{15}$)
- **Exponential**: $O(c^n)$ (E.g., 2^{n+80})

Small exercise:
Show that for any integers *a* and *b*,
$\log_a n = O(\log_b n)$

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

An algorithm with a polynomial time is tractable.

Otherwise, it is intractable.

Cf. Definition of Big-O notation

In this class, I will not give the formal definition:

Definition: For functions f and g on natural numbers, if

$\exists c, n_0 > 0, \forall n \geqq n_0 \, [f(n) \leqq c\, g(n)]$

then we say $f(n)$ is <u>in the order of $g(n)$</u> and denote it by $f(n) = O(g(n))$.

<u>Remark</u>: the constants c and n_0 must be determined <u>independently</u> of n.

Ex. 1: The followings hold for any functions f, g and h on natural numbers:

1. $\forall n [\, f(n) \leqq g(n)] \rightarrow f(n) = O(g(n))$
2. $[\, f(n) = O(g) \text{ and } g(n) = O(h(n))] \rightarrow f(n) = O(h(n))$

Ex. 2: Prove the following:

1. $5n^3+4n^2+n=O(n^3)$
2. $5n^3+4n^2+n=O(n^4)$
3. $5n^3+4n^2+n \neq O(n^2)$

[Comment] Some people write as $f(n) \in O(g(n))$

- If you are interested in, please check textbook!

Computation of Fibonacci sequence:

ANALYSIS OF ALGORITHMS

The Fibonacci sequence: Running time of fib

We use a "simplistic" model for estimation:

Each "elementary instruction" such as an assignment, an arithmetic operation, a Boolean test, etc. takes unit time.

```
int fib(unsigned 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)`:

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

The Fibonacci sequence: Running time of fib

We use a "simplistic" model for estimation:

Each "elementary instruction" such as an assignment, an arithmetic operation, a Boolean test, etc. takes unit time.

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

It is (well) know that $F_n = O(\varphi^n)$, where φ is "the golden ratio" $\varphi = (1+\sqrt{5})/2 \fallingdotseq 1.61803$, so the running time of `fib(n)` is exponential!

The Fibonacci sequence: Running time of fib2

```
int fib2(unsigned 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 $T(n)$ 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.

The Fibonacci sequence: Running time of fib3

```
int fib3(unsigned 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, and each time it performs 6 operations.
- The total running time $T(n)$ is therefore $T(n) = 6n + 2 = O(n)$.
- We only use 4 variables: $O(1)$ space.

This `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(n)` 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 10 years, we will only be able to execute `fib(110)` …
- Only one Fibonacci number per year: this is the "curse" of exponential running times!!

Representative functions in big-O notation

- **Constant**: $O(1)$ (E.g., 10)
- **Logarithmic**: $O(\log n)$ (E.g., $3 \log n + 23$)
- **Linear**: $O(n)$
- **Quasi-linear**: $O(n \log n)$
- **Quadratic**: $O(n^2)$
- **Cubic**: $O(n^3)$
- **Polynomial**: $O(n^c)$ (E.g., $35n^{80} + 800n^{20} + 23n^{15}$)
- **Exponential**: $O(c^n)$ (E.g., 2^{n+80})

An algorithm with a quasi-linear running time is practical.
An algorithm with a polynomial time is tractable.
Otherwise, it is intractable.