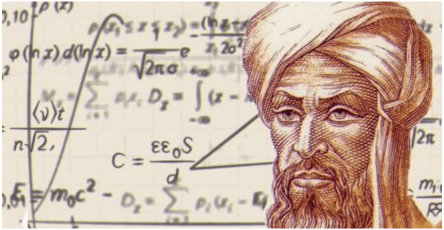

THE UNIVERSITY OF
AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Algorithms and Data Structures

Lecture 5 More issues on algorithm analysis

Jiamou Liu
The University of Auckland



Plan for Today

There are a number of caveats which concern the validity of algorithm analysis. We finalise this section by discussing these issues.

- What amounts to an elementary operation?
- How is input size measured?
- What happens if there are many inputs of a given size?

Elementary Operations

In previous lecture:

- Adding two numbers takes $O(1)$ time
- Multiplying two numbers takes $O(1)$ time

In real life:

- Only when integers a, b can fit into a machine word can they be added/multiplied in $O(1)$ time.
- Nowadays a machine word length is typically 64 bits, so the integers needs to be no bigger than $2^{63} \approx 9.22 \times 10^{18}$.

Question. When do we need to add/multiply large numbers?

- **Cryptography:** Public-key cryptography algorithms typically operate on integers with hundreds of digits.
- **Scientific computing:** Arbitrary-precision arithmetic are performed on numbers whose digits of precision are limited only by the available memory of the host system

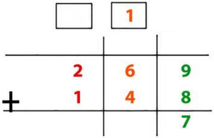
Question. What should be done when adding/multiplying large numbers?

- Use variable-length arrays to store digits in a number.
- Design algorithms that perform arithmetic operations on the variable-length arrays.

Example. Adding two large numbers is not an elementary operation:

Addition Table

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18



Example. Suppose that integers a, b are stored as two arrays $a[0..n-1]$ and $b[0..n-1]$.

The following algorithm performs addition and outputs an array $s[0..n]$ storing $a+b$:

```
function ADDITION(arrays a[0..n − 1], b[0..n − 1])
    Initialise array s[0..n]
    c ← 0                                              ▷ carry bit
    for j ← 0 to n − 1 do
        Look up addition table for
            s[j] ← a[j] + b[j] + c and
            update the corresponding carry bit c
    if c = 1 then set s[n] = 1.
    return array s
```

The algorithm above takes time $\Theta(n)$.

Example. Consider the FASTFIB algorithm introduced in Lecture 2.

```
function FASTFIB(integer n)
    if n < 0 then return 0
    else if n = 0 then return 0
    else if n = 1 then return 1
    else
        a ← 1                          ▹ stores F(i) at bottom of loop
        b ← 0                          ▹ stores F(i − 1) at bottom of loop
        for i ← 2 to n do
            t ← a
            a ← a + b
            b ← t
    return a
```

Fact. Each Fibonacci number $F(n)$ contains roughly $\Theta(n)$ digits.

If the numbers a, b are stored as arrays, and + performs ADDITION algorithm, then FASTFIB will have running time

$$1 + 2 + 3 + \cdots + n \text{ which is } \Theta(n^2)$$

Nevertheless, in this course, we will assume "+", "×" are elementary

by default,

unless stated otherwise.

Input Size

Definition

The input size of an algorithm is the number of bits taken to store the input of the algorithm.

Example. ADDITION algorithm: The input size n is the length of the numbers a and b[1].

function ADDITION(arrays $a[0..n-1]$, $b[0..n-1]$)
  Initialise array $s[0..n]$
  $c \leftarrow 0$ ▹ carry bit
  for $j \leftarrow 0$ to $n-1$ **do**
    Look up addition table for
      $s[j] \leftarrow a[j] + b[j] + c$ and
      update the corresponding carry bit c
  if $c = 1$ **then** set $s[n] = 1$.
  return array s

The running time $\Theta(n)$ is stated with respect to the input size n.

[1] Assuming they have the same length.

Recall. Two algorithms for computing Fibonacci numbers:

Algorithm	Running Time (asymptotic)
SLOWFIB	$\Omega(1.618^n)$
FASTFIB	$\Theta(n)$

Recall. Two algorithms for computing Fibonacci numbers:

Algorithm	Running Time (asymptotic)
SLOWFIB	$\Omega(1.618^n)$
FASTFIB	$\Theta(n)$

Mistake: The "n" above is in fact input value, not its size.

E.g. We usually use binary encoding for the input:

Value	Binary	Size
2	10	2
3	11	2
5	101	3
15	1111	4
x	–	$\Theta(\lg x)$

Example.

- We use a different symbol, x, to denote the input value (i.e., the previous n).
- We now use n to denote the size of x, i.e., say $\lg x$.
- Then $x = 2^n$.

The running time of SLOWFIB and FASTFIB algorithms:

Algorithm	Running Time (in x)	Running (in n)
SLOWFIB	$\Omega(1.618^x)$	$\Omega(1.618^{2^n})$
FASTFIB	$\Theta(x)$	$\Theta(2^n)$

Question.

- Now FASTFIB is in fact an exponential time algorithm!
- Can you design a polynomial time algorithm to solve this problem?

In this course, from now on, we will use n to denote

the size of the input,

not the value of input.

Different Inputs of a Given Size

- Input value $\neq$ input size
- With the same input size, there may be many input values:
  E.g., $n = 3$ corresponds to values
  $$4(100), 5(101), 6(110), 7(111)$$
- An algorithm may have different running time on different inputs of size n.

Definition

Let `algo` be an algorithm and n denote the input size.

- The worst-case running time of `algo` maps n to the maximum running time of `algo` on any input with size n.
- The average-case running time of `algo` maps n to the average running time of `algo` on all inputs with size n.
- The best-case running time of `algo` maps n to the minimum running time of `algo` on any input with size n.

Example. In many cases, the worst-case, average-case, and best-case running time are the same.

```
function ADDITION(arrays a[0..n − 1], b[0..n − 1])
    Initialise array s[0..n]
    c ← 0                                          ▹ carry bit
    for j ← 0 to n − 1 do
        Look up addition table for
            s[j] ← a[j] + b[j] + c and
            update the corresponding carry bit c
    if c = 1 then set s[n] = 1.
    return array s
```

The worst-case, average-case, and best-case running times are all $\Theta(n)$.

Example (count leading zero). We want to solve this problem:

- **INPUT:** A 0/1-valued array $a[0..n-1]$,
- **OUTPUT:** The number of 0s before the first 1; or the length of the array if there is no 1.

E.g.

INPUT	OUTPUT
$[0, 0, 1, 0, 0, 1]$	2
$[0, 0, 0, 0, 0, 0]$	6
$[1, 0, 0, 1, 0, 1]$	0

We can solve the problem using a simple algorithm:

function ZERO(arrays $a[0..n-1]$)
  $j \leftarrow 0$
  $count \leftarrow 0$
  while $a[j] = 0$ **do**
    $count \leftarrow count + 1$
    $j \leftarrow j + 1$
  return $count$

Example. Continued from above

```
function ZERO(arrays a[0..n − 1])
    j ← 0
    count ← 0
    while a[j] = 0 do
        count ← count + 1
        j ← j + 1
    return count
```

Asymptotic analysis of running time:

- **Best-case**: When $a[1] = 1$, the **while**-loop terminates straightaway. So running time $\Theta(1)$.
- **Worst-case**: When $a[0..n-1]$ contains no 1, the **while**-loop repeats n iterations. So running time $\Theta(n)$.
- **Average-case**: We need to analyse the running time for all possible input of size n.

Average-case: We need to analyse the running time for all possible input of size n.

- There are in total 2^n possible inputs (0/1-valued arrays $a[0..n-1]$)
- For $i = 1, \ldots, n$, there are precisely 2^{n-i} arrays of the form $[\underbrace{0, \ldots, 0, 1}_{i}, \underbrace{\star, \ldots, \star}_{n-i}]$. Each array will run i iterations of **while**-loop.
- There is 1 array with n 0s $[0, 0, \ldots, 0]$.

Sum of the number of **while**-loop iterations over all inputs:

$$n + \sum_{i=1}^{n} i2^{n-i} = n + 2^{n-1} + 2 \times 2^{n-2} + 3 \times 2^{n-3} + \cdots + n \times 2^0$$

- One can easily prove by induction that $\sum_{i=1}^{n} i2^{n-i} = 2^{n+1} - n - 2$ (you can try for yourself).
- Thus the average running time is

$$\frac{n + \sum_{i=1}^{n} i2^{n-i}}{2^n} = \frac{2^{n+1} - 2}{2^n} \leq 2 \text{ which is } O(1)$$

Question. What are the pros and cons of worst and average case analysis?

Worst-case running time :

- Worst-case bounds are valid for all instances. Important for mission-critical applications.
- Worst-case bounds are often easier to derive mathematically.

Worst-case running time :

- Worst-case bounds can hugely exceed expected running time and have little predictive or comparative value.
- Average-case running time is often more realistic, provided the algorithm will run on “random” data and we are risk-tolerant.

In this course, we will mostly perform

worst-case running time analysis,

and we will discuss average-case only for special algorithms.

Final word on algorithm analysis:

- Algorithms are meant to be implemented and used.
- The mathematical analysis of running time give us insights on the theoretical limitations of the algorithm under idealised assumptions.
- But how the algorithm actually performs in practice can only be seen empirically.

Summary

Here is a list of the main points covered in this lecture

- What amounts to an elementary operation?

  Operations over data that fit into a machine word.
- How is input size measure?

  n denotes the number of bits used to store the input, not the value of the input.
- What happens if there are many inputs of a given size?

  Best-case, worst-case, average-case running time analysis.


THE UNIVERSITY OF
AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND