Run-Time Analysis

Tao Hou

Run-Time Analysis

- Goal: Measure the *efficiency* (running time) of algorithms for comparing which one is *faster* among different algorithms

Run-Time Analysis

- Goal: Measure the *efficiency* (running time) of algorithms for comparing which one is *faster* among different algorithms
- Difficulty:
  - The running time of an algorithm varies with the *size* of input; even for different inputs of the *same* size, running time may vary.
  - Different *implementations* of an algorithms can run differently; the same implementation on different machines also runs differently
  - Parallelism; caching; hyper-threading

Run-Time Analysis

- Goal: Measure the *efficiency* (running time) of algorithms for comparing which one is *faster* among different algorithms
- Difficulty:
  - The running time of an algorithm varies with the *size* of input; even for different inputs of the *same* size, running time may vary.
  - Different *implementations* of an algorithms can run differently; the same implementation on different machines also runs differently
  - Parallelism; caching; hyper-threading
- Solution:
  - Measure the *growth* of the running time w.r.t input size, where the growth is roughly like an order of magnitude

Example: Counting the number of iterations

Insertion Sort

```
for i=2 to n do
    key ← A[i];
    j = i − 1;
    while (j > 0) and (A[j] > key) do
        A[j + 1] ← A[j];
        j ← j − 1;
    end
    A[j + 1] ← key;
end
```

Input:	6, 4, 3, 8, 5		
$i = 2$:	6, 4, 3, 8, 5	$\Rightarrow$	4, 6, 3, 8, 5
$i = 3$:	4, 6, 3, 8, 5	$\Rightarrow$	3, 4, 6, 8, 5
$i = 4$:	3, 4, 6, 8, 5	$\Rightarrow$	3, 4, 6, 8, 5
$i = 5$:	3, 4, 6, 8, 5	$\Rightarrow$	3, 4, 5, 6, 8

Example: Counting the number of iterations

Insertion Sort

```
for i=2 to n do
    key ← A[i];
    j = i − 1;
    while (j > 0) and (A[j] > key) do
        A[j + 1] ← A[j];
        j ← j − 1;
    end
    A[j + 1] ← key;
end
```

Idea:

- Before each iteration i, we have an invariant that $A[1, \ldots, i-1]$ is already sorted
- At iteration i, insert $A[i]$ after the the first element in $A[1, \ldots, i-1]$ (counting from the right) which is no greater than $A[i]$

Example: Counting the number of iterations

Insertion Sort

```
for i=2 to n do
    key ← A[i];
    j = i − 1;
    while (j > 0) and (A[j] > key) do
        A[j + 1] ← A[j];
        j ← j − 1;
    end
    A[j + 1] ← key;
end
```

Number of iterations in the best and worst case:

Example: Counting the number of iterations

Insertion Sort

```
for i=2 to n do
    key ← A[i];
    j = i − 1;
    while (j > 0) and (A[j] > key) do
        A[j + 1] ← A[j];
        j ← j − 1;
    end
    A[j + 1] ← key;
end
```

Number of iterations in the best and worst case:

Input Size: n
Best case: $n - 1$
Worst Case: $1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$

Time complexity function

Definition: The time complexity function $T : \mathbb{N} \to \mathbb{R}$ of an algorithm is a function s.t. $T(n)$ equals the *maximum* running time of any input with size n.

Definition taken from: Michael R. Garey, David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*

Time complexity function

Definition: The time complexity function $T : \mathbb{N} \rightarrow \mathbb{R}$ of an algorithm is a function s.t. $T(n)$ equals the *maximum* running time of any input with size n.

Definition taken from: Michael R. Garey, David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*

Notice:

- The above defined is indeed the *worst-case* time complexity, which we care about the most in computer science
- If we replace 'maximum' with 'average', then this becomes the definition of *average* time complexity, which we occasionally do
- If we replace 'running time' with 'memory', then this becomes the definition of memory/space complexity function

Input size

Best notion for 'input size' depends on specific problems:

- For most problems, n is the number of items in input, e.g., array size
- Sometimes, the size of input is measured with two numbers rather than one, e.g., for graph inputs, the input size is typically number of vertices (n) and number of edges (m)
- Some other problems (e.g., multiplying two integers) take input size as the total number of bits needed to represent the input in ordinary binary notation: we may only very occasionally do this in this course

Problem with previous time complexity function

Difficulty: It is hard or even impossible to really define what T is

- e.g., what is $T(10)$ for input size 10?

Problem with previous time complexity function

Difficulty: It is hard or even impossible to really define what T is

- e.g., what is $T(10)$ for input size 10?

Solution: We measure the running time T **asymptotically** using O-, Θ-, and Ω-analysis

Asymptotic Notations

Let $f, g : \mathbb{N} \to \mathbb{R}$ be asymptotically positive functions
($f(n), g(n)$ are always positive for large enough n)

Asymptotic Notations

Let $f, g : \mathbb{N} \to \mathbb{R}$ be asymptotically positive functions
($f(n), g(n)$ are always positive for large enough n)

Definition (Big-O): $f(n) \in O(g(n))$ if $\exists c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq cg(n) \quad \forall n \geq n_0$; we also say that $g(n)$ is an *asymptotic upper bound* of $f(n)$

Asymptotic Notations

Let $f, g : \mathbb{N} \to \mathbb{R}$ be asymptotically positive functions
($f(n), g(n)$ are always positive for large enough n)

Definition (Big-O): $f(n) \in O(g(n))$ if $\exists c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq cg(n) \quad \forall n \geq n_0$; we also say that $g(n)$ is an *asymptotic upper bound* of $f(n)$

Examples:
$n \in O(n^2)$
$n \log n \notin O(n)$
$2n + 5 \in O(n)$
$\frac{1}{2}n^2 + 2n + 10 \in O(n^2)$
$\log_{100} n \in O(n^{0.0001})$
$n^{100} \in O(2^n)$

Asymptotic Notations

Definition (Big-Omega): $f(n) \in \Omega(g(n))$ if $\exists c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \geq cg(n) \quad \forall n \geq n_0$; we also say that $g(n)$ is an *asymptotic lower bound* of $f(n)$

Asymptotic Notations

Definition (Big-Omega): $f(n) \in \Omega(g(n))$ if $\exists c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \geq cg(n) \quad \forall n \geq n_0$; we also say that $g(n)$ is an *asymptotic lower bound* of $f(n)$

Remark: We have $f \in \Omega(g)$ iff $g \in O(f)$

Asymptotic Notations

Definition (Big-Omega): $f(n) \in \Omega(g(n))$ if $\exists c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \geq cg(n) \quad \forall n \geq n_0$; we also say that $g(n)$ is an *asymptotic lower bound* of $f(n)$

Remark: We have $f \in \Omega(g)$ iff $g \in O(f)$

Definition (Big-Theta): $f(n) \in \Theta(g(n))$ iff $f(n) \in \Omega(g(n))$ and $f(n) \in O(g(n))$; we also say that $g(n)$ is an *asymptotic tight bound* of $f(n)$

Asymptotic Notations

Definition (Big-Omega): $f(n) \in \Omega(g(n))$ if $\exists c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \geq cg(n) \quad \forall n \geq n_0$; we also say that $g(n)$ is an *asymptotic lower bound* of $f(n)$

Remark: We have $f \in \Omega(g)$ iff $g \in O(f)$

Definition (Big-Theta): $f(n) \in \Theta(g(n))$ iff $f(n) \in \Omega(g(n))$ and $f(n) \in O(g(n))$; we also say that $g(n)$ is an *asymptotic tight bound* of $f(n)$

Remark: We have $f \in \Theta(g)$ iff $g \in \Theta(f)$ (try to think of why!)

Asymptotic Notations

Definition (Big-Omega): $f(n) \in \Omega(g(n))$ if $\exists c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \geq cg(n) \quad \forall n \geq n_0$; we also say that $g(n)$ is an *asymptotic lower bound* of $f(n)$

Remark: We have $f \in \Omega(g)$ iff $g \in O(f)$

Definition (Big-Theta): $f(n) \in \Theta(g(n))$ iff $f(n) \in \Omega(g(n))$ and $f(n) \in O(g(n))$; we also say that $g(n)$ is an *asymptotic tight bound* of $f(n)$

Remark: We have $f \in \Theta(g)$ iff $g \in \Theta(f)$ (try to think of why!)

Examples:
$n^{100} + 2n^{90} + n^{70} + n^2 + 1 \in \Theta(n^{100})$
$\log(n!) \in \Theta(n \log n)$
Stirling's Approximation: $n! \approx \frac{n^n}{e^n}\sqrt{2\pi n}$

Asymptotic Notations

Note:

- We use '$\in$' to denote the asymptotic relations for a reason: $O(g(n))$ can be thought of as the *set* of functions having $g(n)$ as an asymptotic upper bound (the same for Big-Θ and -Ω)
- Sometimes we simply write $f(n) \in O/\Omega/\Theta(g(n))$ as $f(n) = O/\Omega/\Theta(g(n))$, e.g., $n = O(n^2)$, $\log(n!) = \Theta(n \log n)$

Properties

Transitivity:

- If $f = O(g)$ and $g = O(h)$, then $f = O(h)$
- If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$
- If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$

Additivity:

- If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$
- If $f = \Omega(h)$ and $g = \Omega(h)$, then $f + g = \Omega(h)$
- If $f = \Theta(h)$ and $g = \Theta(h)$, then $f + g = \Theta(h)$

Using limits to determine asymptotic order

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \quad \Rightarrow \quad f(n) \in O(g(n))$ but $g(n) \notin O(f(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \quad \Rightarrow \quad f(n) \in \Omega(g(n))$ but $g(n) \notin \Omega(f(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0 \ (c \neq \infty) \quad \Rightarrow \quad f(n) \in \Theta(g(n))$

L'Hopital's rule: convenient for determining the limit

L'Hopital's rule

For two functions $f(n), g(n)$, if $\lim_{n \to a} f(n)$ and $\lim_{n \to a} g(n)$ are both 0 or both ∞ (notice that a could be ∞), then

$$\lim_{n \to a} \frac{f(n)}{g(n)} = \lim_{n \to a} \frac{f'(n)}{g'(n)}$$

Example:

$$\lim_{n \to \infty} \frac{n}{e^n} = \lim_{n \to \infty} \frac{1}{e^n} = 0$$

(i.e., $n \in O(e^n)$)

Asymptotic Bounds for Some Common Functions

Polynomials.

- $a_d n^d + a_{d-1} n^{d-1} + \cdots a_1 n + a_0 \in \Theta(n^d)$ for $a_d > 0$

Logarithms.

- $\log_a n = \Theta(\log_b n)$ for any base $a, b > 1$
- For every $a > 0$, $\log n = O(n^a)$

Exponentials.

- For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$

Asymptotic Bounds for Some Common Functions

Polynomials.

- $a_d n^d + a_{d-1} n^{d-1} + \cdots a_1 n + a_0 \in \Theta(n^d)$ for $a_d > 0$

Logarithms.

- $\log_a n = \Theta(\log_b n)$ for any base $a, b > 1$
- For every $a > 0$, $\log n = O(n^a)$

Exponentials.

- For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$

So,

- logarithm "$<$" polynomial "$<$" exponential

Some Notes

- Whenever we say the time complexity of an algorithm is $O(f(n))$, what we really mean is that the *time complexity function* of the algorithm $\in O(f(n))$
- E.g., an algorithm is $O(n \log(n))$, or an algorithm is $\Omega(n^2)$
- Question: When we want to know the lower bound for the time complexity of an algorithm, do we consider the worst-case time complexity or the best-case?

Some common running time

- Linear Time: $O(n)$
- '$n \log n$' time, near-linear time: $O(n \log n)$
- Quadratic Time: $O(n^2)$
- Cubic Time: $O(n^3)$
- Polynomial Time: $O(n^k)$, for $k > 0$
- Exponential Time: $O(a^n)$, for $a > 1$

"Efficient" algorithms

Definition : An algorithm is called *efficient* if its time complexity function $T(n) \in O(n^k)$ for a fixed integer k; the algorithm is also called a *polynomial-time algorithm*

“Efficient” algorithms

Definition : An algorithm is called *efficient* if its time complexity function $T(n) \in O(n^k)$ for a fixed integer k; the algorithm is also called a *polynomial-time algorithm*

Question: Is $O(n \log n)$ polynomial time algorithm?

"Efficient" algorithms

Definition : An algorithm is called *efficient* if its time complexity function $T(n) \in O(n^k)$ for a fixed integer k; the algorithm is also called a *polynomial-time algorithm*

Question: Is $O(n \log n)$ polynomial time algorithm?

Why we have a definition like this?

- Although an $O(N^{20})$ algorithm is useless in practice, the polynomial time algorithms that people develop almost *always* have low constants and exponents
- Breaking through the exponential barrier of brute force typically exposes some *crucial structure* of the problem

Exceptions

- Some polynomial-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare (simplex algorithm, `grep`)

Asymptotic Growth

Sort the following functions in a non-decreasing order of their asymptotic growth

(1) $2n^3 - 5n$	(6) $4 \lg n - 1$
(2) $5n - 3$	(7) $n!$
(3) $n^n - 2$	(8) $2n(\lg n)^2 + 3n$
(4) $3n^2 - 3n + 1$	(9) $10n - 2$
(5) $2^n + n + 1$	(10) 10^{100}

Solution: (10), (6), (2)=(9), (8), (4), (1), (5), (7), (3)

Example of asymptotic analysis (in full detail)

Insertion Sort

```
1 for i=2 to n do
2     key ← A(i);
3     j = i − 1;
4     while (j > 0) and (A(j) > key) do
5         A(j + 1) ← A(j);
6         j ← j − 1;
7     end
8     A(j + 1) ← key;
9 end
```

- Assume Line i takes c_i time to execute
- Line 1, 2, 3, 8 executes $n - 1$ times
- In worst case, Line 4 executes i times, Line 5 and 6 executes $i - 1$ times for each i

Example of asymptotic analysis (in full detail)

$$
\begin{aligned}
T(n) =& (c_1 + c_2 + c_3 + c_8) * (n-1) + \sum_{i=2}^{n} (c_4 * i + (c_5 + c_6) * (i-1)) \\
=& (c_1 + c_2 + c_3 + c_8) * (n-1) + \sum_{i=2}^{n} (c_4 + c_5 + c_6) * i \\
& + (c_5 + c_6) * (n-1) \\
=& (c_4 + c_5 + c_6)(n+2)(n-1)/2 \\
& + (c_1 + c_2 + c_3 + c_8 + c_5 + c_6) * (n-1) \\
=& \alpha n^2 + \beta n + c \in O(n^2)
\end{aligned}
$$

Note: You don't need to provide such level of details in hw/exams

Example of asymptotic analysis (in short)

We know that the running time of the insertion sort is dominated by the inner loop (Line 4–6), which runs for $\leq n^2$ times in the worst case, so we have:

$$T(n) \leq c * n^2 \in O(n^2)$$

Example of asymptotic analysis (in short)

We know that the running time of the insertion sort is dominated by the inner loop (Line 4–6), which runs for $\leq n^2$ times in the worst case, so we have:

$$T(n) \leq c * n^2 \in O(n^2)$$

Note: You will be asked to give an upper bound (Big-O) which should be *as tight as possible*, e.g., $O(n^2)$ is a tight upper bound for insertion sort but $O(n^{100})$ is not

Example of asymptotic analysis (in short)

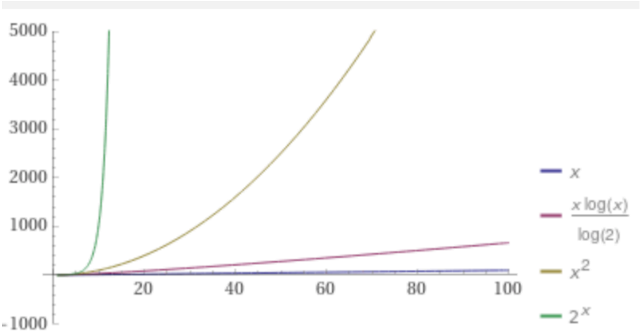
We know that the running time of the insertion sort is dominated by the inner loop (Line 4–6), which runs for $\leq n^2$ times in the worst case, so we have:

$$T(n) \leq c * n^2 \in O(n^2)$$

Note: You will be asked to give an upper bound (Big-O) which should be *as tight as possible*, e.g., $O(n^2)$ is a tight upper bound for insertion sort but $O(n^{100})$ is not

Question: Is the time complexity of insertion sort $\Omega(n^2)$? (If the answer is yes, then insertion sort is indeed $\Theta(n^2)$ so n^2 is the *tightest* possible bound)

Asymptotic Growth



https:

//www.wolframalpha.com/input?i=x%2C+x+log_2%28x%29%2C+x%5E2%2C+2%5Ex%2C+x+from+1+to+100%2C+y+from+1+to+5000

Asymptotic Growth

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

(Figure from *Algorithm design* by Kleinberg and Tardos)

An Example: The Fibonacci Sequence

- A well-known sequence of numbers

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

- Mathematical definition:

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

Our First Algorithm

```
FIBONACCI(n)
if n == 0
        return 0
elseif n == 1
        return 1
else return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

Our First Algorithm

```
FIBONACCI(n)
if n == 0
    return 0
elseif n == 1
    return 1
else return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

The ***three fundamental questions*** for algorithmists:

1. Is the algorithm *correct?*
   - for every valid input, does it terminate?
   - if so, does it do the right thing?
2. How much *time* does it take to complete?
3. Can we do better?

Complexity of Our First Algorithm

- Let $T(n)$ be the number of ***basic steps*** needed to compute **Fibonacci**(n)

```
FIBONACCI(n)
if n == 0
        return 0
elseif n == 1
        return 1
else return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

$T(0) = 2$; $T(1) = 3$

$T(n) = T(n-1) + T(n-2) + 3$

Complexity of Our First Algorithm

- Let $T(n)$ be the number of ***basic steps*** needed to compute **Fibonacci**(n)

```
FIBONACCI(n)
if n == 0
        return 0
elseif n == 1
        return 1
else return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

$T(0) = 2$; $T(1) = 3$

$T(n) = T(n-1) + T(n-2) + 3 \quad \Rightarrow T(n) \geq F_n$

Complexity of Our First Algorithm (2)

- So, let's try to understand how F_n grows with n

$$T(n) \geq F_n = F_{n-1} + F_{n-2}$$

Now, since $F_n \geq F_{n-1} \geq F_{n-2} \geq F_{n-3} \geq \ldots$

$$F_n \geq 2F_{n-2} \geq 2(2F_{n-4}) \geq 2(2(2F_{n-6})) \geq \ldots \geq 2^{\frac{n}{2}}$$

This means that

$$T(n) \geq (\sqrt{2})^n \approx (1.4)^n$$

Complexity of Our First Algorithm (2)

- So, let's try to understand how F_n grows with n

$$T(n) \geq F_n = F_{n-1} + F_{n-2}$$

Now, since $F_n \geq F_{n-1} \geq F_{n-2} \geq F_{n-3} \geq \ldots$

$$F_n \geq 2F_{n-2} \geq 2(2F_{n-4}) \geq 2(2(2F_{n-6})) \geq \ldots \geq 2^{\frac{n}{2}}$$

This means that

$$T(n) \geq (\sqrt{2})^n \approx (1.4)^n$$

- $T(n)$ ***grows exponentially*** with n
- Can we do better?

A Better Algorithm

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

Idea: we compute F_n only from the previous two numbers!

```
SmartFibonacci(n)
if n == 0
        return 0
elseif n == 1
        return 1
else pprev = 0
        prev = 1
        for i = 2 to n
                f = prev + pprev
                pprev = prev
                prev = f
return f
```

A Better Algorithm

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

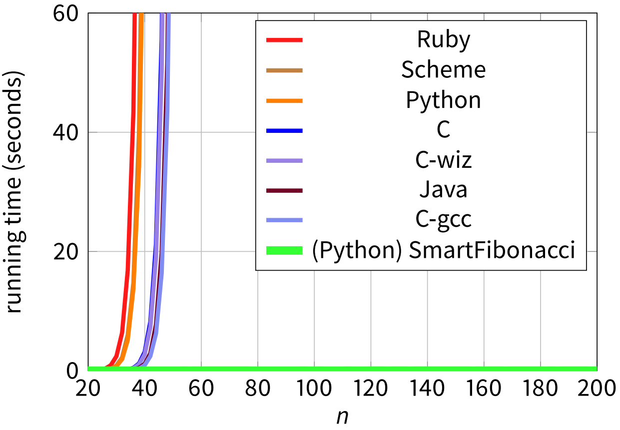
Idea: we compute F_n only from the previous two numbers!

```
SmartFibonacci(n)
if n == 0
        return 0
elseif n == 1
        return 1
else pprev = 0
        prev = 1
        for i = 2 to n
                f = prev + pprev
                pprev = prev
                prev = f
return f
```

$T(n) = 6 + 6(n - 1) = 6n$

The *complexity* of **SmartFibonacci**(n) is ***linear*** in n

Results



running time (seconds)
60
40
20
0
20
40
60
80
100
120
140
160
180
200
n
Ruby
Scheme
Python
C
C-wiz
Java
C-gcc
(Python) SmartFibonacci