Analysis of Algorithms

CS 141, Fall 2015
Analysis of Algorithms: Issues

• Correctness/Optimality
• Running time ("time complexity")
• Memory requirements ("space complexity")
• Power
• I/O utilization
• Ease of implementation
• ...

Correctness

An algorithm is **correct** if, for every input size,

it holts

with the correct output.
Analysis of Algorithms

• **Primitive Operations**: Low-level computations independent from the programming language can be identified in pseudo-code

• Examples:
  • calling a method and returning from a method
  • arithmetic operations (e.g. addition)
  • comparing two numbers, etc.

• By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm
### Input size and basic operation examples

<table>
<thead>
<tr>
<th><strong>Problem</strong></th>
<th><strong>Input size measure</strong></th>
<th><strong>Basic operation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Searching for key in a list of $n$ items</td>
<td>Number of items in the list, i.e., $n$</td>
<td>Key comparison</td>
</tr>
<tr>
<td>Multiplication of two matrices</td>
<td>Matrix dimensions or total number of elements</td>
<td>Multiplication of two numbers</td>
</tr>
<tr>
<td>Checking primality of a given integer $n$</td>
<td>size of $n = \text{number of digits (in binary representation)}$</td>
<td>Division</td>
</tr>
<tr>
<td>Typical graph problem</td>
<td>$#\text{vertices and/or } #\text{edges}$</td>
<td>Visiting a vertex or traversing an edge</td>
</tr>
</tbody>
</table>
Why Running Time?
Definition of the Fibonacci function

\[
F(n) = \begin{cases} 
0 & n = 0 \\
1 & n = 1 \\
F(n - 1) + F(n - 2) & n > 1
\end{cases}
\]

Recursive implementation

```python
function fib1(n)
if n == 0: return 0
if n == 1: return 1
return fib1(n - 1) + fib1(n - 2)
```

Time complexity?

\[
T(n) = \begin{cases} 
n \leq 1 & \\
n > 1
\end{cases}
\]
Figure 0.1 The proliferation of recursive calls in `fib1`. 

```

```

```
function fib1(n)
if n = 0:   return 0
if n = 1:   return 1
return fib1(n - 1) + fib1(n - 2)

\[ T(n) = \begin{cases} 
2 & \text{for } n \leq 1 \\
T(n - 1) + T(n - 2) + 3 & \text{for } n > 1 
\end{cases} \]

\[ T(n) \geq F_n \approx 2^{0.694n} \]
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i - 1] + f[i - 2]
return f[n]

T(n) is linear in n !!
Average Case vs. Worst Case

• An algorithm may run faster on certain data sets than on others (e.g., for the sorting problem, the input is partially sorted)

• Finding the average case can be very difficult, so typically algorithms are measured by the worst case time complexity
Average Case vs. Worst Case

- In time-critical application domains (e.g., air traffic control, surgery, IP lookup, ...) knowing the worst case time complexity is crucial.

![Graph showing worst-case, best-case, and average-case performance for inputs A to G.](same input size)
Worst Case Time-Complexity

• Definition: The worst case time-complexity of an algorithm A is the asymptotic running time of A as a function of the size of the input, when the input is the one that makes the algorithm slower in the limit.

• How do we measure the running time of an algorithm?
Example

```python
def rMax(A):
    if len(A) == 1:
        return A[0]
    return max(rMax(A[1:]), A[0])

def iMax(A):
    currentMax = A[0]
    for i in range(len(A)):
        if currentMax < A[i]:
            currentMax = A[i]
    return currentMax
```

Max recursive

Max iterative

Time-complexity is $O(n)$
Asymptotic notation

Section 0.3 of the textbook
The “Big-Oh” Notation

- **Definition**: Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if and only if there are positive constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$
The “Big-Oh” Notation

Figure 1.3: Illustrating the “big-Oh” notation. The function \( f(n) \) is \( O(g(n)) \), for \( f(n) \leq c \cdot g(n) \) when \( n \geq n_0 \).
Asymptotic Notation

Big - O

Theorem

Suppose that $f_1(x) = O(g_1(x))$ and $f_2(x) = O(g_2(x))$. Then

(a) $f_1(x) + f_2(x) = O(g_1(x) + g_2(x))$

$$= O(\max(g_1(x), g_2(x)))$$

(b) $f_1(x)f_2(x) = O(g_1(x)g_2(x))$
Example

\[ f(n) = 2n + 6 \]
\[ g(n) = n \]

For functions \( f(n) \) and \( g(n) \) (to the right) there are positive constants \( c \) and \( n_0 \) such that:
\[ f(n) \leq c \cdot g(n) \text{ for } n \geq n_0 \]

Note: the picture itself is NOT a proof, but it may help you finding the parameters.
Proof

- $f(n) = 2n + 6$
- $g(n) = n$
- $2n + 6 \leq 4n$  ???
- $2n + 6 \leq 4n$ when $n \geq 3$
- So, if we choose $c = 4$, then $n_0 = 3$ satisfies $f(n) \leq c \cdot g(n)$ for $n \geq n_0$
- **Conclusion**: $2n + 6$ is $O(n)$
Asymptotic Notation

Big - O

**Theorem**

Let \( f(x) = \sum_{i=0}^{k} a_i x^i \). Then \( f(x) = O(x^k) \).

**Proof:** Let \( A = \max |a_i| \), be the maximum absolute value of the coefficient in \( f(x) \). We can estimate \( f(x) \) as follows. For \( x \geq 1 \) we have

\[
    f(x) = a_k x^k + a_{k-1} x^{k-1} + \ldots + a_1 x + a_0 \\
    \leq A(x^k + x^{k-1} + \ldots + x + 1) \\
    \leq A(k + 1)x^k.
\]

Thus \( f(x) \leq cx^k \) for \( c = A(k + 1) \) and \( x \geq 1 \). The theorem follows. \( \square \)
Asymptotic Notation

• **Note**: Even though it is **correct** to say “7n - 3 is \( O(n^3) \)”, a **more precise** statement is “7n - 3 is \( O(n) \)”, that is, one should make the approximation **as tight as possible**

• **Simple Rule**: Drop lower order terms and constant factors

  7n-3 is \( O(n) \)

  \( 8n^2 \log n + 5n^2 + n \) is \( O(n^2 \log n) \)
Asymptotic Notation

Big - O

Theorem

Let $a > 0$, $b > 0$, $c > 1$. Then

(a) $1 = O(\log^a n)$. (b) $\log^a n = O(n^b)$. (c) $n^b = O(c^n)$.

Proof: (c) Let $d = c^{1/b}$, then $d > 1$, and

\[
\begin{align*}
n &\leq 1 + d + d^2 + \ldots + d^{n-1} \\
&= \frac{d^n - 1}{d - 1} \\
&\leq Ad^n, \\
&= Ac^{(1/b)n} \\
n^b &\leq Bc^n \\
n^b &= O(c^n)
\end{align*}
\]

, since $d > 1$

- summation of the geom. sequences

, where $A = 1 / (d - 1)$

, where $B = A^b$
Asymptotic Notation

• Special classes of algorithms
  – constant: \( O(1) \)
  – logarithmic: \( O(\log n) \)
  – linear: \( O(n) \)
  – quadratic: \( O(n^2) \)
  – cubic: \( O(n^3) \)
  – polynomial: \( O(n^k), k \geq 1 \)
  – exponential: \( O(a^n), n > 1 \)
Asymptotic Notation

• “Relatives” of the Big-Oh
  – $\Omega(f(n))$: Big Omega
    • asymptotic lower bound
  – $\Theta(f(n))$: Big Theta
    • asymptotic tight bound
Big Omega

• **Definition**: Given two functions \( f(n) \) and \( g(n) \), we say that \( f(n) \) is \( \Omega(g(n)) \) if and only if there are positive constants \( c \) and \( n_0 \) such that \( f(n) \geq c \ g(n) \) for \( n \geq n_0 \)

• **Property**: \( f(n) \) is \( \Omega(g(n)) \) iff \( g(n) \) is \( O(f(n)) \)
Big Theta

- **Definition**: Given two functions $f(n)$ and $g(n)$, we say that $f(n)$ is \( \Theta(g(n)) \)
  *if and only if*
  there are positive constants \( c_1, c_2 \) and \( n_0 \) such that \( c_1 g(n) \leq f(n) \leq c_2 g(n) \) for \( n \geq n_0 \)

- **Property**: \( f(n) \) is \( \Theta(g(n)) \) if and only if
  “\( f(n) \) is \( O(g(n)) \) AND \( f(n) \) is \( \Omega(g(n)) \)”
Summary

• $A \in O(f(n))$ means “the algorithm $A$ won’t take longer than $f(n)$, give or take a constant multiplier and lower order terms” (upper bound)

• $A \in \Theta(f(n))$ means “the algorithm $A$ will take as long as $f(n)$, give or take a constant multiplier and lower order terms” (tight bound)

• $A \in \Omega(f(n))$ means “the algorithm $A$ will take longer than $f(n)$, give or take a constant multiplier and lower order terms” (lower bound)
Establishing order of growth using limits

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} 
0 & \text{order of growth of } f(n) < \text{order of growth of } g(n) \\
c > 0 & \text{order of growth of } f(n) = \text{order of growth of } g(n) \\
\infty & \text{order of growth of } f(n) > \text{order of growth of } g(n)
\end{cases}
\]

Examples:

• 10n vs. \(n^2\)

• \(n(n+1)/2\) vs. \(n^2\)
Orders of growth: some important functions

- All logarithmic functions $\log_a n$ belong to the same class $\Theta(\log n)$ no matter what the logarithm’s base $a > 1$ is.
- All polynomials of the same degree $k$ belong to the same class: $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$ in $\Theta(n^k)$.
- Exponential functions $a^n$ have different orders of growth for different $a$’s.
- Order $\log n < \text{order } n < \text{order } n \log n < \text{order } n^k$ ($k \geq 2$ constant) $< \text{order } a^n < \text{order } n! < \text{order } n^n$.
- Caution: Be aware of very large constant factors.
Suppose each operation takes 1 nanoseconds (10^{-9} seconds)

<table>
<thead>
<tr>
<th>n</th>
<th>lg n</th>
<th>n</th>
<th>n lg n</th>
<th>n^2</th>
<th>2^n</th>
<th>n!</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.003μs</td>
<td>0.01μs</td>
<td>0.033μs</td>
<td>0.1μs</td>
<td>1μs</td>
<td>3.63ms</td>
</tr>
<tr>
<td>20</td>
<td>0.004μs</td>
<td>0.02μs</td>
<td>0.086μs</td>
<td>0.4μs</td>
<td>1ms</td>
<td>77.1years</td>
</tr>
<tr>
<td>30</td>
<td>0.005μs</td>
<td>0.02μs</td>
<td>0.147μs</td>
<td>0.9μs</td>
<td>1sec</td>
<td>&gt;10^{15}years</td>
</tr>
<tr>
<td>100</td>
<td>0.007μs</td>
<td>0.1μs</td>
<td>0.644μs</td>
<td>10μs</td>
<td>&gt;10^{13}years</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>0.013μs</td>
<td>10μs</td>
<td>130μs</td>
<td>100ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.020μs</td>
<td>1ms</td>
<td>19.92μs</td>
<td>16.7min</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- For n < 10, the difference is insignificant.
- Θ (n!) algorithms are useless well before n = 20.
- Θ (2^n) algorithms are practical for n < 40.
- Θ (n^2) and Θ (n lg n) are both useful, but Θ (n lg n) is significantly faster.
Time analysis for iterative algorithms

Steps

• Decide on parameter $n$ indicating input size
• Identify algorithm’s basic operation
• Determine worst case(s) for input of size $n$
• Set up a sum for the number of times the basic operation is executed
• Simplify the sum using standard formulas and rules
Example

Give the number $f(n)$ of letters “Z” printed by Algorithm PrintZs below:
(first using a summation notation, and then - a closed-form formula for $f(n)$ )

Analyze the worst-case time complexity of the following algorithm, and give a tight bound using the big-theta notation

**Algorithm** PRINTZS ($n : integer$)

```
for $i \leftarrow 1$ to $3n + 1$ do
  for $j \leftarrow 1$ to $i^2 + 2$ do print(“Z”)
```
Example

Give the number $f(n)$ of letters “Z” printed by Algorithm PrintZs below:
(first using a summation notation, and then - a closed-form formula for $f(n)$)

Analyze the worst-case time complexity of the following algorithm, and give a tight bound using the big-theta notation

**Algorithm PRINTZS** $(n : \text{integer})$

```
for $i \leftarrow 1$ to $3n + 1$ do
  for $j \leftarrow 1$ to $i^2 + 2$ do print(“Z”)
```

\[
\sum_{i=1}^{3n+1} (i^2 + 2) = 9n^3 + \frac{27}{2}n^2 + \frac{25}{2}n + 3.
\]