Algorithm Analysis

CSC 349

Christopher Siu

Algorithms

"[W]hat we can do once, we can do twice, and by induction we fool ourselves into believing that we can do it as many times as needed!"
— Edsger Dijkstra, *Structured Programming*

Definition

An **algorithm** is a sequence of precise instructions for performing a computation or for solving a problem.

- Muhammad ibn Mūsā al-Khwārizmī (*c.* 780 – *c.* 850) was a Persian scholar of the House of Wisdom in Baghdad.
- He wrote *On the Calculation with Hindu Numerals*, later translated into Latin as *De numero Indorum*, by "Algorithmi".

Algorithms

Consider the following problem:

- Given a sequence of comparable elements sorted in ascending order, find a given value in that sequence.

Example

Find an Element in a Sorted Sequence

1. Start with the middle element in the sequence.
2. If that element is equal to the given value, return “true”.
3. If it’s too large, repeat with the first half of the sequence.
4. If it’s too small, repeat with the second half of the sequence.
5. If there are no more elements in the sequence, return “false”.

Algorithms

Example *(cont.)*

The above is (subjectively) a poor description of an algorithm.

An algorithm should...

- ...take as **input** instances of a specified problem, then produce as **output** solutions to that problem.
- ...be **definite**, having precisely defined steps.
- ...be **correct**, producing the desired output for each input.
- ...be **finite**, producing each output in a finite number of steps.
- ...be **general**, being applicable to any input of a specified form.

Mathematical-Style Pseudocode

- Language-specific implementation details are abstracted away.
  - Memory management, libraries, backing data structures...
  - Input validation and error handling...
- Good pseudocode is human-readable.

Example

Assignment	**let** x be $x + 1$
Conditionals	**if** $a_i > x$ **then**... **if** $A \subseteq B$ **then**...**else**...
Loops	**for** all $v \in V$ **do**... **for** i **from** 0 **to** $n - 1$ **do**... **while** $S \neq \emptyset$ **do**...

Mathematical-Style Pseudocode

$\textsc{BinarySearch}(x, A = (a_0, a_1, \ldots, a_{n-1}))$

Input: An integer x and a finite, non-empty sequence A of n integers sorted in ascending order

Output: Whether or not x is an element of A

let mid be $\lfloor {}^{n}/_{2} \rfloor$
if $a_{\text{mid}} = x$ **then**
    return T
else if $n = 1$ **then**
    return F
else if $a_{\text{mid}} > x$ **then**
    return $\textsc{BinarySearch}(x, (a_0, a_1, \ldots, a_{\text{mid}-1}))$
else
    return $\textsc{BinarySearch}(x, (a_{\text{mid}}, a_{\text{mid}+1}, \ldots, a_{n-1}))$

Algorithm Analysis

Example

Suppose we have two algorithms that solve the same problem:

- BinarySearch starts in the middle of a sequence.
- LinearSearch starts at the beginning of a sequence.

Which algorithm should we implement?

- Are both algorithms correct?
  - Are there situations where only one algorithm is applicable?
- How much time does each algorithm require?
  - Does the size of the sequence affect this time?
  - Do the best-, average-, and worst-case times differ?
- How much space does each algorithm require?

Correctness

Theorem

$\textsc{BinarySearch}$ is correct.

Lemma

Let A be a finite, non-empty sequence of integers sorted in ascending order. $\textsc{BinarySearch}(x, A)$ returns T iff $x \in A$.

- Recursively defined algorithms can be proven with induction.
- A proof by induction is a "recursively defined proof".

Definition

An **recursive** algorithm is one that solves a problem by reducing it to smaller instances of the same problem.

Complexity

- The **time complexity**, $T(n)$, is typically given as a **Big-$\boldsymbol{O}$ estimate** of operations performed as a function of input size.

Definition

Suppose that a recursive algorithm reduces a problem of size n into a subproblems, where each subproblem is of size $m < n$. Then:

$$T(n) = a \cdot T(m) + O(g(n))$$

- In other words, the total work done, $T(n)$, is equal to:
  - The number of recursive calls, a...
  - ...times the work done by each recursive call, $T(m)$...
  - ...plus the work done non-recursively, $O(g(n))$.
- The complexity of a recursive algorithm is a recurrence relation.

The Iterative Method

Example

The complexity of BinarySearch is given by:

$$
\begin{aligned}
T(n) &= a \cdot T(m) + O(g(n)) \\
&= 1 \cdot T(n/2) + O(1) \\
&= T(n/2) + O(1) \\
&= T(n/4) + O(1) + O(1) \\
&= T(n/8) + O(1) + O(1) + O(1) \\
&\dots \\
&= \log_2(n) \cdot O(1) \; = O(\log n)
\end{aligned}
$$

The Tree Method

Example

The complexity of BinarySearch is $T(n) = T(n/2) + O(1)$:

	Problem Size	Work
$\log_2(n)$	n	1
	$n/2$	1
	$n/4$	1
	$\vdots$	
	1	1

$$T(n) = O(1 + 1 + 1 + \ldots + 1) = O(\log n)$$

The Master Theorem

Theorem

Suppose that a recursive algorithm has complexity given by:

$$T(n) = a \cdot T\left(\left\lceil \frac{n}{b} \right\rceil\right) + O\left(n^d\right)$$

Then for all $a > 0$, $b > 1$, $d \geq 0$:

$$T(n) = \begin{cases} O\left(n^d\right) & d > \log_b(a) \\ O\left(n^d \log n\right) & d = \log_b(a) \\ O\left(n^{\log_b(a)}\right) & d < \log_b(a) \end{cases}$$

The Master Theorem

Example

The complexity of BinarySearch is $T(n) = T(n/2) + O(1)$:

1. $a = 1, \ a > 0$
2. $b = 2, \ b > 1$
3. $d = 0, \ d \geq 0$
4. $0 = \log_2(1), \ d = \log_b(a)$

Thus:

5. $T(n) = O\left(n^d \log n\right) = O(\log n)$

Common Algorithm Complexities

Example

Suppose the complexity of an algorithm is given by $T(n) = 2T({}^{n}/_{2}) + O\left(n^2\right)$. Then the algorithm is $O\left(n^2\right)$.

Example

Suppose the complexity of an algorithm is given by $T(n) = 2T({}^{n}/_{2}) + O(1)$. Then the algorithm is $O(n)$.

Example

Suppose the complexity of an algorithm is given by $T(n) = T(n-1) + O(n)$. Then the algorithm is $O\left(n^2\right)$.

Common Algorithm Complexities

- Colloquially, "Big-O" refers to a **Big-Θ estimate**.
- A smaller estimate indicates a faster algorithm.

Complexity	**Terminology**	**Example**
$\Theta(1)$	"Constant"	Addition
$\Theta(\log n)$	"Logarithmic"	Binary Search
$\Theta(n)$	"Linear"	Maximum Element
$\Theta(n \log n)$	"Linearithmic"	Merge Sort
$\Theta\left(n^b\right),\ b > 1$	"Polynomial"	Matrix Multiplication
$\Theta(b^n),\ b > 1$	"Exponential"	Satisfiability
$\Theta(n!)$	"Factorial"	Brute Force TSP