18:52 7/28/2011

Chapter 3

Algorithms and Integers

3.1 ALGORITHMS

DEF: An ***algorithm*** is a finite set of precise instructions for performing a computation or for solving a problem.

Example 3.1.1: A computer program is an algorithm.

Remark: From a mathematical perspective, an algorithm represents a function. The British mathematician Alan Turing proved that some functions cannot be represented by an algorithm.

CLASSROOM PERSPECTIVE

Every computable function can be represented by many different algorithms. Naive algorithms are almost never optimal.

TERMINOLOGY: A ***good pseudocoding*** of an algorithm provides a clear prose representation of the algorithm and also is transformable into one or more target programming languages.

Algo 3.1.1: Find Maximum

Input: unsorted array of integers $a_1, a_2, \ldots, a_n$
Output: largest integer in array

{*Initialize*} $max := a_1$

For $i := 2$ **to** n
  If $max < a_i$ **then** $max := a_i$
  Continue with next iteration of for-loop.

Return (max)

Remark: For a sorted array, there would be a much faster algorithm to find the maximum. In general, the representation of the data profoundly affects both the choice of an algorithm and the execution time.

Algo 3.1.2: Unsorted Sequential Search

Input: unsorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

{*Initialize*} $i := 1$

While $i \leq n$ **and** $x \neq a_i$
$i := i + 1$
Continue with next iteration of while-loop.

If $i \leq n$ **then** $loc := i$ **else** $loc := 0$
Return (loc)

Remark: If the array were presorted into ascending (or descending) order, then faster algorithms could be used.

(1) linear search could stop sooner

(2) 2-level search could avoid many comparisons

(3) binary search could divide-and-conquer

Algo 3.1.3: Sorted Sequential Search

Input: sorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

{*Initialize*} $i := 1$

While $i \leq n$ **and** $x < a_i$
$i := i + 1$
Continue with next iteration of while-loop.

If $(i \leq n$ **cand** $x = a_i)$ **then** $loc := i$ **else** $loc := 0$
Return (loc)

DEF: The logical expression ***conditional-and***

$$boolean1 \textbf{ cand } boolean2$$

is like conjunction, except that $boolean2$ is not evaluated if $boolean1$ is false.

Example 3.1.2: In Algorithm 3.1.3, if $i > n$ then variable a_i does not exist. Since the conditional-and does not evaluate such an a_i, problems are avoided.

Algo 3.1.4: Two-level Search

Input: sorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

{*Initialize*} $i := 10$
{*Find target sublist of 10 entries*}
While $i \leq n$ **and** $x < a_i$
$i := i + 10$
Continue with next iteration of while-loop.

{*Linear search target sublist of 10 entries*}
{*Initialize*} $j := i - 9$
While $j < i$ **and** $x < a_j$
$j := j + 1$
Continue with next iteration of while-loop.

If $(j \leq n$ **cand** $x = a_j)$ **then** $loc := j$ **else** $loc := 0$
Return (loc)

Algo 3.1.5: Binary Search

Input: sorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

$\{Initialize\}\ left := 1; right := n$

While $left < right$
$mid := \lfloor (left + right)/2 \rfloor$
If $x > a_{mid}$ **then** $left := mid$ **else** $right := mid$
Continue with next iteration of while-loop.

If $x = a_{left}$ **then** $loc := left$ **else** $loc := 0$
Return (loc)

3.2 GROWTH OF FUNCTIONS

DEF: Let f and g be functions $\mathbb{R} \to \mathbb{R}$. Then f is ***asymptotically dominated*** by g if

$$(\exists K \in \mathbb{R})\,(\forall x > K)\,[f(x) \leq g(x)]$$

NOTATION: $f \preceq g$.

Remark: This means that there is a location $x = K$ on the x-axis, after which the graph of the function g lies above the graph of the function f.

BIG OH CLASSES

DEF: Let f and g be functions $\mathbb{R} \to \mathbb{R}$. Then f is in the ***class*** $\mathcal{O}(g)$ ("***big-oh of g***") if

$$(\exists C \in \mathbb{R})\,[f \preceq Cg]$$

NOTATION: $f \in \mathcal{O}(g)$.

DISAMBIGUATION: Properly understood, $\mathcal{O}(g)$ is the class of all functions that are asymptotically dominated by any multiple of g.

TERMINOLOGY NOTE: The idiomatic phrase
"f is big-oh of g"
makes sense if one imagines either
that the word "in" precedes the word "big-oh",
or that "big-oh of g" is an adjective.

Example 3.2.1: $4n^2 + 21n + 100 \in \mathcal{O}(n^2)$

Pf: First suppose that $n \geq 0$. Then

$$\begin{aligned} 4n^2 + 21n + 100 &\leq 4n^2 + 24n + 100 \\ &\leq 4(n^2 + 6n + 25) \\ &\leq 8n^2 \text{ which holds whenever} \end{aligned}$$

$n^2 \geq 6n + 25$, which holds whenever

$n^2 - 6n + 9 \geq 34$, which holds whenever

$n - 3 \geq \sqrt{34}$, which holds whenever $n \geq 9$.

Thus,

$$(\forall n \geq 9)[4n^2 + 21n + 100 \leq 8n^2] \qquad \diamondsuit$$

Remark: We notice that n^2 itself is asymptotically dominated by $4n^2 + 21n + 100$. However, we proved that $4n^2 + 21n + 100$ is asymptotically dominated by $8n^2$, a multiple of n^2.

WITNESSES

This operational definition of membership in a big-oh class makes the definition of asymptotic dominance explicit.

DEF: Let f and g be functions $\mathbb{R} \to \mathbb{R}$. Then f is in the ***class*** $\mathcal{O}(g)$ ("***big-oh of g***") if

$$(\exists C \in \mathbb{R})\,(\exists K \in \mathbb{R})\,(\forall x > K)\,[f(x) \leq Cg(x)]$$

DEF: In the definition above, the multiplier C and the location K on the x-axis after which $Cg(x)$ dominates $f(x)$ are called the ***witnesses*** to the relationship $f \in \mathcal{O}(g)$.

Example 3.2.1, continued: The values

$$C = 8 \quad \text{and} \quad K = 9$$

are witnesses to the relationship

$$4n^2 + 21n + 100 \in \mathcal{O}(n^2)$$

Larger values of C and K could also serve as witnesses. However, a value of C less than or equal to 4 could not be a witness.

CLASSROOM EXERCISE

If one chooses the witness $C = 5$, then $K = 30$ could be a co-witness, but $K = 9$ could not.

Lemma 3.2.1. $(x+1)^n \in \mathcal{O}(x^n)$.

Pf: Let C be the largest coefficient in the (binomial) expansion of $(x+1)^n$, which has $n+1$ terms. Then

$$(x+1)^n \ \leq \ C(n+1)x^n \qquad \diamondsuit$$

Example 3.2.2: The proof of Lemma 3.2.1 uses the witnesses

$$C = \binom{n}{\lfloor \frac{n}{2} \rfloor} \ \text{ and } \ K = 0$$

Theorem 3.2.2. *Let $p(x)$ be any polynomial of degree n. Then $p(x) \in \mathcal{O}(x^n)$.*

Pf: Apply the method of Lemma 3.2.1. $\diamondsuit$

Example 3.2.3: $100n^5 \in \mathcal{O}(e^n)$. Observing that $n = e^{\ln n}$ inspires what follows.

Pf: Taking the upper Riemann sum with unit-sized intervals for $\ln x = \int_1^n \frac{dx}{x}$ implies for $n > 1$ that

$$\begin{aligned}
\ln(n) &< \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} \\
&\leq \left(\frac{1}{1} + \cdots + \frac{1}{5}\right) + \frac{1}{6} + \cdots + \frac{1}{n} \\
&\leq \left(\frac{1}{1} + \cdots + \frac{1}{5}\right) + \frac{1}{6} + \cdots + \frac{1}{6} \\
&\leq 5 + \frac{n-5}{6}
\end{aligned}$$

Therefore, $6 \ln n \leq n + 25$, and accordingly,

$$100n^5 = 100 \cdot e^{5 \ln n} < 100 \cdot e^{n+25} < e^{32} \cdot e^n$$

We have used the witnesses $C = e^{32}$ and $K = 0$. $\diamondsuit$

Example 3.2.4: $2^n \in \mathcal{O}(n!)$.

Pf:

$$\begin{aligned}
\overbrace{2 \cdot 2 \cdots 2}^{n \text{ times}} &= 2 \cdot 1 \cdot \overbrace{2 \cdot 2 \cdots 2}^{n-1 \text{ times}} \\
&\leq 2 \cdot 1 \cdot 2 \cdot 3 \cdots n = 2n!
\end{aligned}$$

We have used the witnesses $C = 2$ and $K = 0$. $\diamondsuit$

BIG-THETA CLASSES

DEF: Let f and g be functions $\mathbb{R} \to \mathbb{R}$. Then f is ***in the class*** $\Theta(g)$ ("***big-theta of g***") if $f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(f)$.

3.3 COMPLEXITY

DISAMBIGUATION: In the early 1960's, Chaitin and Kolmogorov used ***complexity*** to mean measures of complicatedness. However, most theoretical computer scientists have used it in a jargon sense that means measures of resource consumption.

DEF: ***Algorithmic time-complexity measures*** estimate the time or the number of computational steps required to execute an algorithm, given as a function of the size of the input.

TERMINOLOGY: The resource for a complexity measure is implicitly time, unless space or something else is specified.

DEF: A ***worst-case complexity measure*** estimates the time required for the most time-consuming input of each size.

DEF: An ***average-case complexity measure*** estimates the average time required for input of each size.

Example 3.3.1: In searching and sorting, complexity is commonly measures in terms of the number of comparisons, since total computation time is typically a multiple of that.

Algo 3.1.1: Find Maximum

Input: unsorted array of integers $a_1, a_2, \ldots, a_n$
Output: largest integer in array

$\{Initialize\}$ $max := a_1$

For $i := 2$ **to** n
  If $max < a_i$ **then** $max := a_i$
  Continue with next iteration of for-loop.

Return (max)

Big-Oh:

Always takes $n - 1$ comparisons.

Time complexity is in $\mathcal{O}(n)$.

Example 3.3.2:

Algo 3.1.2: Unsorted Sequential Search

Input: unsorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

$\{Initialize\}$ $i := 1$

While $i \leq 2$ **and** $x \neq a_i$
$i := i + 1$
Continue with next iteration of while-loop.

If $i \leq n$ **then** $loc := i$ **else** $loc := 0$
Return (loc)

Target in or not in Array:

Worst case takes n comparisons.

Average case takes $n/2$ comparisons.

Target not in Array:

Every case takes n comparisons.

Big-Oh:

Time complexity is in $\mathcal{O}(n)$.

Example 3.3.3:

Algo 3.1.3: Sorted Sequential Search

Input: sorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

{*Initialize*} $i := 1$

While $i \leq n$ **and** $x < a_i$
$i := i + 1$
Continue with next iteration of while-loop.

If $(i \leq n$ **cand** $x = a_i)$ **then** $loc := i$ **else** $loc := 0$
Return (loc)

Target in or not in Array:

Worst case takes n comparisons.

Average case takes $n/2$ comparisons.

Big-Oh:

Time complexity is in $\mathcal{O}(n)$.

Example 3.3.4:

Algo 3.1.4: Two-level Search

Input: sorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

{*Initialize*} $i := 10$
{*Find target sublist of 10 entries*}
While $i \leq 2$ **and** $x \leq a_i$
$i := i + 10$
Continue with next iteration of while-loop.
{*Linear search target sublist of 10 entries*}
{*Initialize*} $j := i - 9$
While $j \leq i$ **and** $x < a_j$
$j := j + 1$
Continue with next iteration of while-loop.
If $(j \leq n$ **cand** $x + a_j)$ **then** $loc := j$ **else** $loc := 0$
Return (loc)

Target in or not in Array:

Worst case takes $(n/10) + 10$ comparisons.

Big-Oh: Time complexity is in $\mathcal{O}(n)$.

To optimize the two-level search, minimize

$$\frac{n}{x} + x$$

as in differential calculus.

$$\frac{-n}{x^2} + 1 = 0 \quad \Rightarrow \quad x = \sqrt{n}$$

Target in or not in Array:

Worst case takes $2\sqrt{n}$ comparisons.

Big-Oh: Time complexity is in $\mathcal{O}(\sqrt{n})$.

Increasing to k levels further decreases the execution time to $\mathcal{O}(\sqrt[k]{n})$,
provided that k is not too large.

Example 3.3.5:

Algo 3.1.5: Binary Search

Input: sorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

$\{Initialize\}$ $left := 1; right := n$

While $left < right$
$mid := \lfloor (left + right)/2 \rfloor$
If $x > a_{mid}$ **then** $left := mid$ **else** $right := mid$
Continue with next iteration of while-loop.

If $x = a_{left}$ **then** $loc := left$ **else** $loc := 0$
Return (loc)

Target in or not in Array:

Every case takes $\lg n$ comparisons.

Big-Oh: Time complexity is in $\mathcal{O}(\lg n)$.

COMPLEXITY JARGON

DEF: A problem is ***solvable*** if it can be solved by an algorithm.

Example 3.3.6: Alan Turing defined the ***halting problem*** to be that of deciding whether a computational procedure (e.g., a program) halts for all possible input. He proved that the halting problem is unsolvable.

DEF: A problem is in ***class* P** if it is solvable by an algorithm that runs in polynomial time.

DEF: A problem is ***tractable*** if it is in class **P**.

DEF: A problem is in ***class* NP** if an algorithm can decide in polynomial time whether a putative solution is really a solution.

Example 3.3.7: The problem of deciding whether a graph is 3-colorable is in class **NP**. It is believed not to be in class **P**.