COMPLEXITY

ALGORITHMS & DATA STRUCTURES – I

COMP 221

Algorithm Efficiency

What to measure?

Space utilization: amount of memory required

☆***Time efficiency***: amount of time required to process the data

Depends on many factors:

- size of input
- speed of machine
- quality of source code
- quality of compiler

These factors vary from one machine/compiler (platform) to another

$\Rightarrow$ Count the *number of times instructions are executed*

So, measure computing time as

T(n) = computing time of an algorithm for input of size n

= number of times the instructions are executed

Example: Calculating the Mean

p. 350

/* Algorithm to find the mean of n real numbers.
Receive: integer n >= 1 and an array $x[0], \ldots, x[n-1]$ of real numbers
Return: The mean of $x[0], \ldots, x[n-1]$
---*/

1. Initialize *sum* to 0. ———————— 1
2. Initialize index variable *i* to 0. ———————— 1
3. While $i < n$ do the following: ———————— n+1
4. a. Add $x[i]$ to *sum*. ———————— n
5. b. Increment *i* by 1. ———————— n
6. Calculate and return $mean = sum / n$. —— 1

T(n) = **3n + 4**

Big Oh Notation

The *computing time* of an algorithm *on input of size n, T(n),* is said to have ***order of magnitude f(n)***, written **T(n) is O(f(n))**
if **there is some constant C such that**

$T(n) \le C \cdot f(n)$ for all sufficiently large values of n.

Another way of saying this:

The ***complexity*** of the algorithm is O(f(n)).

f(n) is usually simple:
$n, n^2, n^3, \ldots$
2^n
$1, \log_2 n$
$n \log_2 n$
$\log_2 \log_2 n$

Example: For the Mean-Calculation Algorithm:

T(n) is **O(n)**

Note that constants and multiplicative factors are ignored.

T(n) is also $O(n^2)$, $O(n^3)$, etc., but use smallest f(n) $\Rightarrow$ most info

Worst-case Analysis

The arrangement of the input items may affect the computing time.
How then to measure performance?

best case – not very informative

average - too difficult to calculate

worst case - *usual measure*

/* Linear search of the list $a[0], \ldots, a[n-1]$.

Receive: An integer n an array of n elements and *item*

Return: *found* = true and *loc* = position of *item* if the search is successful; otherwise, *found* is false. */

1. *found* = false.
2. *loc* = 0.
3. While (*loc* < *n* && !*found*)
4. If *item* = *a*[*loc*] *found* = true // *item* found
5. Else Increment *loc* by 1 // keep searching

Worst case: Item not in the list: $T_L(n)$ is **O(n)**

Average case (assume equal distribution of values) is **O(n)**

Binary Search

/* Binary search of the list $a[0], \ldots, a[n-1]$ in which the items are in ascending order.

Receive: integer n and an array of n elements and *item*.

Return: *found* = true and *loc* = position of *item* if the search successful otherwise, *found* is false. */

1. *found* = false.
2. *first* = 0.
3. *last* = $n - 1$.
4. While (*first* <= *last* && !*found*)
5.   Calculate *loc* = (*first* + *last*) / 2.
6.   If *item* < *a*[*loc*] then
7.   *last* = *loc* – 1.   // search first half
8.   Else if *item* > *a*[*loc*] then
9.   *first* = *loc* + 1.   // search last half
10.   Else
        found = true.   // *item* found

p. 255

Worst case: Item not in the list: $T_B(n) =$ **$O(\log_2 n)$**

Makes sense: each pass cuts search space in half!

THANK YOU

?