

: Assignment: 3.8: 1,5,13,15,17,19,23,27

(THU). Algorithms And Their Efficiency

Defn.I.0: A *computable* expression can be completely specified by a *finite* list of well-defined commands.

Example (1): The most basic set which is **computable** is Z^+ , (the positive integers) since the set can be generated by the following procedure, requiring only 3 commands:

- A. Let $n = 1$
- B. $n := n + 1$
- C. Return to B.

- **Note:** The "=" in B. Means: "assign the value of."
- **Note:** Commands A. Through C. can be repeated arbitrarily long , if never allowed to stop, all of Z^+ would be generated. These commands

Example (2): The real numbers aren't computable! For there exists no procedure to generate R which can be specified by a finite numbr of well defined commands.

Defn.I.1: An *effective procedure* is a procedure which is *computable*.

Defn I.2 An *algorithm* is a solution to a problem which can be formulated by an *effective procedure*.

Not all solutions to problems can be expressed as algorithms: this is a more general way of noting that not all mathematical objects are computable. As mentioned, discrete sets are countable, therefore computable. Why? Because the act of counting (generating an exhaustive list) can be seen as a simple algorithm. (Consider commands A. - C. listed above, generating the positive integers.)

Algorithms lead historically and naturally into the concept of a machine which will do what they say; i.e., a computer. Because computers are naturally bound by limits of memory, time, space, we can rate algorithms in terms of their "efficiency;" how much does it "cost" (memory, time, etc) to run a particular procedures. Just as currency (\$) is divided into fundamental units of change (in the case of \$, the cent), so the fundamental unit of "cost" of an algorithm is the **Elementary Operation (EO)**.

Defn. I.3: An *EO* is either one of the four arithmetic operations (+, \times , - , /) or one of the three fundamental comparisons (=, <, \leq).

- **Note:** Strictly speaking, since \times is iterated + , and \leq involves both = and < , the above definition seems to oversimplify by treating all of these operations on equal footing. Nonetheless, the simplification is acceptable, and only breaks down when we want to analyze basic algorithms written in assembly-language.
- **Note:** "=" only costs something when it's used as a comparison. If "=" is used to assign a variable a new value (like in command B. Above), then it's not an EO and therefore is "free."

Defn. I.4: The *complexity* of an algorithm A (denoted COMP(A)) is its total number of elementary operations.

This is what we mean by total cost. For example, in the above algorithm generating Z+:

- A. Let $n = 1$ (0 EOs, since = is assign)
- B. $n = n + 1$ (1 EO, the +)
- C. Return to B. (0 EO)

So the total complexity of this procedure = 1 (if we ran it once)
 If we cycled n times, the total complexity = n

Example(2): (Algorithm to evaluate x^n)

Command	Number of EO
A. Input x, n	0
B. $K = 1$	0
C. If $K = n$, STOP. Otherwise, continue.	1 (comparison between K and n)
D. $x = x * x$	1 (one multiplication)
E. $K = K + 1$	1 (one addition)
F. Go to C.	0

We're tempted to say that the complexity of the procedure = 3. But this is false! We have recursion in command blocks C. - F. For example, if the algorithm evaluated x^4 , notice that block C. would be visited 4 times, while D and E would be visited 3 times. We'd loop 3 times in C. - F. Hence the complexity of the procedure in this case = $4 + 3 + 3 = 10$. And in general, for any n , the complexity = $n + (n - 1) + (n - 1) = 3n - 2$. Therefore: $COMP(x^n) = 3n - 2$.

Example 2 ("POLYN," an algorithm evaluating any polynomial of degree n)

Given the coefficients $A_0, A_1, \dots, A_n; x$, POLYN will evaluate: $p_n = A_0 + A_1x^1 + A_2x^2 + \dots + A_nx^n$

- A Input $A_0, A_1, \dots, A_n; x, n$
- B. $Y = A_0$
- C $k = 1$
- D If $k > n$, then write output Y , STOP. Otherwise, continue
- E $Y = Y + A_kx^k$
- F $k = k + 1$
- G Go to D

As in the previous case, this algorithm entails n loops through commands D – G. (Try doing a sample run for the some 3 or 4th degree polynomial you invented to see how the algorithm works) Therefore, for any n :

- (#EOS at D) = $n + 1$ ($n + 1$ comparisons)
- (#EOS at F) = n (n additions)

But what about E? we have one multiplication and one addition to perform at every pass, but notice that for $1 \leq k \leq n$, we must evaluate x^k . We saw in the previous example that this has complexity = $3k + 1$. Therefore in a total number of n executions, the complexity of E is:

$$n \text{ (additions)} + n \text{ (multiplications)} + \{(3 * 1 - 2) + (3 * 2 - 2) + \dots + (3 * n - 2)\}$$

The last expression $\{(3 * 1 - 2) + (3 * 2 - 2) + \dots + (3 * n - 2)\}$ is of course an arithmetic series. Using sigma notation, this series can be written:

$$\sum_{k=1}^n (3k - 2)$$

How do we evaluate and get an answer? Recall the formula $\frac{n(a_1 + a_n)}{2}$

In this case, the first term $a_1 = 3 \cdot 1 - 2 = 1$ the last term $a_n = 3n - 2$

Therefore the series sums to: $\frac{n(1 + 3n - 2)}{2} = \frac{3}{2}n^2 - \frac{1}{2}n$

- The remaining steps in the procedure are: $(n + 1)$ comparisons (step **D**), n additions (steps **E**, **F**) and n multiplications (step **E**)
- So the overall complexity is $\frac{3}{2}n^2 - \frac{1}{2}n + (3n + 1) = \frac{3}{2}n^2 + \frac{5}{2}n + 1$ which is an $O(n^2)$ procedure

Hoerner Procedure:

A Input $A_0, A_1, \dots, A_n; x, n$

B $Y = A_n$

C $k = 1$

D If $k > n$, then write output Y , **STOP**. Otherwise, continue

E $Y = Yx + A_{n-k}$

F $k = k + 1$

G Go to **D**

This procedure does the same thing (evaluate the answer of a degree n polynomial) however:

- There are $n + 1$ comparisons at **D**
- There are $2n$ additions (**E** and **F**)
- There are n subtractions (**F**) (the subscript of A)
- There are n multiplications (**E**)

So the overall complexity is $5n + 1$, an $O(n)$ procedure