Towers of Hanoï Revisited

Let us use the notation $H(i, x, y, z)$ to denote the instructions for solving a Towers of Hanoï problem with i disks, starting on peg x, moving them to peg y with z as spare. Then

one disk $H(1, x, y, z) =$ "Move (the top) disk from x to y"

two disks

$$H(2, x, y, z) = \begin{cases} H(1, x, z, y) \\ H(1, x, y, z) \\ H(1, z, y, x) \end{cases}$$

three disks

$$H(3, x, y, z) = \begin{cases} H(2, x, z, y) \\ H(1, x, y, z) \\ H(2, z, y, x) \end{cases}$$

...501 disks

$$H(501, x, y, z) = \begin{cases} H(500, x, z, y) \\ H(1, x, y, z) \\ H(500, z, y, x) \end{cases}$$

Time Complexity Analysis

("Time Complexity" is a calculation, in Big- Oh notation, of how many steps are needed in the solution of the problem (as a function of the input size)).

Let $N(n)$ be the number of disk moves required to solve a problem with n disks. Then we know $N(0) = 0$ and $N(1) = 1$.

From the above analysis we can write

$$N(n) = \begin{cases} N(n-1) \\ +1 \\ +N(n-1) \end{cases}$$

so

$$N(n) = 1 + 2N(n-1) \tag{1}$$

We can re- use equation (1) to get

$$\begin{aligned} N(n) &= 1 + 2N(n-1) \\ &= 1 + 2[1 + 2N(n-2)] \\ &= 1 + 2 + 4N(n-2) \\ &= 1 + 2 + 4[1 + 2N(n-3)] \\ &= 1 + 2 + 4 + 8N(n-3) \\ &= 1 + 2 + 4 + \cdots + 2^k N(n-k) \end{aligned}$$

This stops eventually when $n - k$ becomes small: Specifically, when $n - k = 1$ we know $N(1) = 1$, so at this point

$$N(n) = 1 + 2 + 4 + \cdots + 2^{n-1} N(1)$$

This is a *geometric series*:

$$a + ar + ar^2 + ar^3 + \cdots + ar^{n-1} = \frac{a(1 - r^n)}{1 - r}$$

In our case, $a = 1$ and $r = 2$ so the sum is

$$\frac{1(1 - 2^n)}{1 - 2} = 2^n - 1$$

The exact number of moves required to solve a Towers of Hanoï Problem with n disks is $2^n - 1$. To this date, no one has come up with a cleverer algorithm to do this quicker. (The original Towers of Hanoï Problem was posed with $n = 64$ disks, which takes 18,446,744,073,709,551,615 moves to solve. Hanoï is the capital of modern day Vietnam.)

Uses of Big- Oh Notation

At a practical level, Big- Oh Notation allows us do two things:

1. Compare two different algorithms for solving a problem, to see which is “faster”
2. Compare the performance of an algorithm to itself for different sizes of input data: i.e., answer questions such as “if I double the size of the input, will the program take twice as long”?

Example 1

Question

If a given algorithm takes 10 seconds to execute on an input data of size 1,000, and the algorithm is given as being $\mathbf{O}(n^2)$, estimate how long it will take to execute if we double the input size to 2,000.

Answer

Since the algorithm is $\mathbf{O}(n^2)$, it means the time it will take is proportional to n^2. We can write this mathematically as

$$t = kn^2$$

for some constant k.

Example 1 Continued

For the first run we are told $t_1 = 10$ and $n_1 = 1,000$, so

$$10 = k(1,000)^2, \text{ or } k = 10/(1,000)^2.$$

For the second run

$$t_2 = kn_2^2 = \left(\frac{10}{1,000^2}\right) 2,000^2 = 40 \text{ seconds}$$

Example 2

Question

For a quicksort of a million items, how much slower would you expect the worst case behaviour to be compared to the best case behaviour?

Answer

Quicksort is $\mathbf{O}(n^2)$ in worst case and $\mathbf{O}(n \log_2 n)$ in best case. So we can write

best case Time $t_1 = k10^6 \log_2 10^6$

worst case Time $t_2 = k(10^6)^2$

Example 2 Continued

So the ratio is

$$\frac{\text{worst}}{\text{best}} = \frac{k(10^6)^2}{k10^6 \log_2 10^6} = \frac{10^6}{\log_2 10^6} = \frac{10^6}{6 \log_2 10}$$

$$= \frac{10^6}{(6)(3.322)} \approx 50,171$$

...so the best case is about fifty thousand times faster!

Example 3

Question

Two algorithms A1 and A2 are known to be $\mathbf{O}(n^3)$ and $\mathbf{O}(3^n)$ respectively. For an input of $n = 10$ both run in 5 milliseconds. Estimate the time each will take for $n = 11$.

Answer

A1

$$t_1 = kn_1^3 \implies 5 = k10^3 \implies k = \frac{5}{10^3}$$

$$t_2 = kn_2^3 = \frac{5}{10^3}(11^3) = 5\left(\frac{11}{10}\right)^3$$

$$= 5(1.1)^3 = 6.655 \text{ milliseconds}$$

Example 3 Continued

A2

$$t_1 = k3^{n_1} \implies 5 = k3^{10} \implies k = \frac{5}{3^{10}}$$

$$t_2 = k3^{n_2} = \left(\frac{5}{3^{10}}\right) 3^{11} = (5)(3) = 15 \text{ milliseconds}$$

Pseudocode

- A “language” halfway between a programming language (e.g. C) and a human language (e.g. english).
- Used to describe an algorithm in more precise terms than in english.
- Not compilable! But if you have the algorithm in pseudocode, you should be able to write a corresponding program in any programming language (C, Java, Python, PHP,...)

BEGIN...END

```
BEGIN
{put-your-comments-here}
END
```

- We start every algorithm with `BEGIN` and finish with `END`.
- `BEGIN` and `END` can also be used to block in a block of code.
- Comments in pseudocode are put inside curly brackets.

Assignment Statements

syntax	`variable_name` ← `expression`
semantics	Assign the valus of `expression` to `variable_name`
examples	a ← 5 b ← b+3 color ← 'orange'

IF...THEN...ELSE

syntax `IF condition THEN statement1 ELSE statement2`

semantics `condition` is a boolean expression (i.e. something that is true or false): If it is true, we execute statement `statement1`. If it is false we execute statement `statement2`.

IF- Example 1

```
BEGIN
{an algorithm to fine the absolute value
(modulus) of a number}
IN(x)
IF (x<0) THEN x ← -x
OUT(x)
END
```

Note we introduce here the pseudocode key-words `IN` and `OUT` to represent (unformatted) input/output.

IF- Example 2

```
BEGIN
{an algorithm to fine the larger of two input
numbers}
IN(a,b)
IF (a>b) THEN c ← a ELSE c ← b
OUT(c)
END
```

FOR loop

syntax
```
FOR variable ←
    initial_value TO
    final_value DO statement(s)
```

semantics

1. Assign initial_value to `variable`.
2. If variable $\leq$ final_value, execute statement(s), otherwise exit loop.
3. Increment (increase) variable by 1.
4. Go to step 2.

N.B.:

- We write all (reserved) keywords in pseudocode in block capitals: here we introduce the three new keywords FOR, TO, DO.
- initial_value and final_value should both be integers, with final_value > initial_value (i.e. we count up).
- statement(s) could be a single statement or a block of statements. If it is a block, it should be enclosed with `BEGIN...END`.

FOR- Example 1

```
BEGIN
{an algorithm to print out 10 numbers}
FOR x ← 1 TO 10 DO OUT(x)
END
```

FOR- Example 2

```
BEGIN
FOR x ← 1 TO 10 DO OUT(3)
END
```

FOR- Example 3

```
BEGIN
FOR x ← 1 TO 10 DO
OUT(2*x)
OUT(3)
END
```

FOR- Example 4

```
BEGIN
  FOR x ← 1 TO 10 DO
  BEGIN
  OUT(2*x)
  OUT(3)
  END
END
```

FOR- Example 5

```
BEGIN
{alogrithm to sum the first 10 natural
numbers}
sum ← 0
FOR i ← 1 TO 10 DO
sum ← sum + i
OUT(sum)
END
```

Exercise: Re-write this algorithm so it finds the sum of the first *n* natural numbers.

FOR- Example 6

```
BEGIN
{alogrithm to calculate n!}
f ← 1
IN(n)
FOR i ← 1 TO n DO
f ← f*i
OUT(f)
END
```

check run- through on board for n=10

WHILE loop

syntax

```
WHILE condition DO
    statement(s)
```

semantics

1. Evaluate the `condition`.
2. If true, execute statement(s) and then go to step 1.

3. If false, exit the loop.

REPEAT...UNTIL loop

syntax

```
REPEAT statement(s) UNTIL
condition
```

semantics

1. Execute the statement(s).
2. Evaluate the `condition`.
3. If it is true, exit the loop.
4. If it is false, go to step 1.

WHILE- Example 1

```
BEGIN
{alogrithm to determine the integer cube-root
of x:  i.e.
** largest integer y with y^3 ≤ x
** integer y such that y^3 ≤ x < (y+1)^3
** calculate cube-root of x and truncate}
IN(x)
i ← 0
WHILE (i*i*i ≤ x) DO i ← i+1
OUT(i-1)
END
```

Comparison of Loops

- Number of times the statements in a WHILE loop are executed: Between 0 and ∞.

- Number of times the statements in a REPEAT...UNTIL loop are executed: Between 1 and ∞.
- Number of times the statements in a FOR loop are executed: Exactly (final_value - initial_value +1).