Running times of algorithms

Operation	Running Time
Integer add/subtract	$\Theta(1)$
Integer multiply/divide	$\Theta(1)$
Float add/subtract	$\Theta(1)$
Float multiply/divide	$\Theta(1)$
Trigonometric Functions (sine, cosine, ..)	$\Theta(1)$
Variable Declaration	$\Theta(1)$
Assignment Operation	$\Theta(1)$
Logical Operations ($<, >, \leq, \geq$, etc)	$\Theta(1)$
Array Access	$\Theta(1)$
Array Length	$\Theta(1)$
1D array allocation	$\Theta(n)$
2D array allocation	$\Theta(n^2)$
Substring extraction	$\Theta(1)$ or $\Theta(n)$
String concatenation	$\Theta(n)$

Consecutive statements

Let two *independent* consecutive statements are P_1 and P_2. Let t_1 be the cost of running P_1 and t_2 be the cost of running P_2. The total cost of the program is the addition of cost of individual statement i.e. $t_1 + t_2$. In asymptotic notation the total time is $\Theta(\max(t_1, t_2))$(we ignore the non significant term).

Example: Consider the following code.

```
int main() {
    // 1. some code with running time n
    // 2. some code with running time n^2
    return 0;
}
```

Assume that statement 2 is independent of statement 1 and statement 1 executes first followed by statement 2. The total running time is

$$\Theta(\max(n, n^2)) = \Theta(n^2)$$

`for` loops

It is relatively easier to compute the running time of `for` loop than any other loops. All we need to compute the running time is how many times the statement inside the loop body is executed. Consider a simple `for` loop in C.

```c
for (i = 0; i < 10; i++) {
    // body
}
```

The loop body is executed 10 times. If it takes m operations to run the body, the total number of operations is $10 \times m = 10m$. In general, if the loop iterates n times and the running time of the loop body are m, the total cost of the program is $n * m$. Please note that we are ignoring the time taken by expression $i < 10$ and statement $i++$. If we include these, the total time becomes

$$1 + 2 \times n + mn = \Theta(mn)$$

Nested `for` loops

Suppose there are p nested `for` loops. The p `for` loops execute $n_1, n_2, \ldots, n_p$ times respectively. The total cost of the entire program is

$$n_1 \times n_2 \times, \ldots, \times n_p \times \text{cost of the body of innermost loop}$$

Consider nested `for` loops as given in the code below

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        // body that runs in linear time n
    }
}
```

There are two `for` loops, each goes `n` times. So the total cost is

$$n \times n \times n = n^3 = \Theta(n^3)$$

`while` loops

`while` loops are usually harder to analyze than `for` loops because there is no obvious a priori way to know how many times we shall have to go round the loop. One way of analyzing `while` loops is to find a variable that goes increasing or decreasing until the terminating condition is met. Consider an example given below

```
while (i > 0) {
    // some computation of cost n
    i = i / 2
}
```

How many times the loop repeats? In every iteration, the value of `i` gets halved. If the initial value of `i` is 16, after 4 iterations it becomes 1 and the loop terminates. The implies that the loop repeats $\log_2 i$ times. In each iteration, it does the n work. Therefore the total cost is $\Theta(n \log_2 i)$.

Recursive calls

To calculate the cost of a recursive call, we first transform the recursive function to a recurrence relation and then solve the recurrence relation to get the complexity. There are many techniques to solve the recurrence relation. These techniques will be discussed in details in the next article.

```
int fact(int n) {
    if (n <= 2) {
        return n;
    }

    return n * fact(n - 1);
}
```

We can transform the code into a recurrence relation as follows.

$$T(n) = \begin{cases} a & \text{if } n \leq 2 \\ b + T(n-1) & \text{otherwise} \end{cases}$$

When `n` is 1 or 2, the factorial of `n` is n itself. We return the result in constant time a. Otherwise, we calculate the factorial of $n-1$ and multiply the result by n. The multiplication takes a constant time b. We use one of the techniques called back substitution to find the complexity.

$$\begin{aligned}
T(n) &= b + T(n-1) \\
&= b + b + T(n-2) \\
&= b + b + b + T(n-3) \\
&= 3b + T(n-3) \\
&= kb + T(n-k) \\
&= nb + T(0) \\
&= nb + a \\
&= \Theta(n)
\end{aligned}$$

Example 1

```
1   int sum(int a, int b) {
2       int c = a + b;
3       return c
4   }
```

The `sum` function has two statements. The first statement (line 2) runs in constant time i.e. $Theta(1)$ and second statement (line 3) also runs in constant time $\Theta(1)$. These two statements are consecutive statements, so the total running time is $\Theta(1) + \Theta(1) = \Theta(1)$

Example 2

```
1  int array_sum(int a, int n) {
2      int i;
3      int sum = 0;
4      for (i = 0; i < n; i++) {
5          sum = sum + a[i]
6      }
7      return sum;
8  }
```

Analysis

1. Line 2 is a variable declaration. The cost is $\Theta(1)$
2. Line 3 is a variable declaration and assignment. The cost is $\Theta(2)$
3. Line 4 - 6 is a `for` loop that repeats n times. The body of the for loop requires $\Theta(1)$ to run. The total cost is $\Theta(n)$.
4. Line 7 is a `return` statement. The cost is $\Theta(1)$.

1, 2, 3, 4 are consecutive statements so the overall cost is $\Theta(n)$

Example 3

```
 1  int sum = 0;
 2  for (i = 0; i < n; i++) {
 3      for (j = 0; j < n; j++) {
 4          for (k = 0; k < n; k++) {
 5              if (i == j == k) {
 6                  for (l = 0; l < n*n*n; l++) {
 7                      sum = i + j + k + l;
 8                  }
 9              }
10          }
11      }
12  }
```

Analysis

1. Line 1 is a variable declaration and initialization. The cost is $\Theta(1)$
2. Line 2 - 11 is a nested for loops. There are four `for` loops that repeat n times. After the third `for` loop in Line 4, there is a condition of `i == j == k`. This condition is true only n times. So the total cost of these loops is $\Theta(n^3) + \Theta(n^4) = \Theta(n^4)$

The overall cost is $\Theta(n^4)$.

Running Time	**Examples**
Constant	1, 2, 100, 300, ...
Logarithmic	$\log n$, $5 \log n$, ...
Linear	n, $n + 3$, $2n + 3$, ...
$n \log n$	$n \log n$, $2n \log n + n$, ...
Polynomial	Quadratic, Cubic, or higher order
Exponential	2^n, 3^n, $2^n + n^4$, ...
Factorial	n!, n! + n, ...

Asymptotic Notations

Example:

$$5n^2 + 3n = \Theta(n^2)$$

because there exist two constants c_1 and c_2 such that

$$c_1 \cdot n^2 \leq 5n^2 + 3n \leq c_2 \cdot n^2$$

Simplyfying:

$$c_1 \leq 5 + \frac{3}{n} \leq c_2$$

We can choose $c_1 = 5, c_2 = 8$.

Asymptotic Notations

Example:

$$3^{2n+3} \neq O(3^n)$$

because there does not exist a constant c such that

$$3^{2n+3} \leq c \cdot 3^n$$

Simplyfying:

$$3^{2n} \cdot 3^3 \leq c \cdot 3^n$$

$$3^n \cdot 3^3 \leq c$$

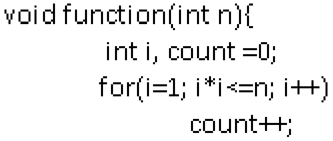
Since

$$\lim_{n \Rightarrow \infty} 3^n \cdot 3^3 = \infty,$$

there does not exist such a constant c.

Exercises

1. Check if the following statements are true!
   1.1 $4^n + 3n^2 + 200 = O(4^n)$
   1.2 $2^{n+1} = O(2^n)$
   1.3 $2^{2n+1} = O(2^n)$
   1.4 $(n+k)^m = \Theta(n^m)$, where k and m are constants
2. Rank the following functions by order of growth:
   $(n+1)!, n!, 4^n, n\cdot 3^n, 3^n+n^2, \frac{3}{4}n^2, n^2+200, 20n+500, 2^{lgn}, n^{\frac{2}{3}}, 1$
3. Find the complexity of the following function!



```
void function(int n){
        int i, count =0;
        for(i=1; i*i<=n; i++)
                count++;
```