

# Calculating the running time of Algorithms

## Basic operations

Knowing the cost of basic operations helps to calculate the overall running time of an algorithm. The table below shows the list of basic operations along with their running time. The list not by any means provides the comprehensive list of all the operations. But I am trying to include most of the operations that we come across frequently in programming.

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.

```
1  int main() {
2      // 1. some code with running time n
3      // 2. some code with running time n^2
4      return 0;
5  }
```

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.

```
1  for (i = 0; i < 10; i++) {
2      // body
3  }
```

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

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

In this analysis, we made one important assumption. We assumed that the body of the loop doesn't depend on  $i$ . Sometimes the runtime of the body does depend on  $i$ . In that case, our calculation becomes a little bit difficult. Consider an example shown below.

```
1  for (i = 0; i < n; i++) {  
2      if (i % 2 == 0) {  
3          // some operations of runtime  $n^2$   
4      }  
5  }
```

In the for loop above, the control goes inside the if condition only when  $i$  is an even number. That means the body of if condition gets executed  $n/2$  times. The total cost is therefore  $n/2 * n^2 = n^3/2 = \Theta(n^3)$ .

## Nested for loops

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

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

Consider nested for loops as given in the code below

```
1  for (i = 0; i < n; i++) {  
2      for (j = 0; j < n; j++) {  
3          // body that runs in linear time n  
4      }  
5  }
```

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

```
1  while (i > 0) {  
2      // some computation of cost n  
3      i = i / 2  
4  }
```

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. This 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.

```
1  int fact(int n) {  
2      if (n <= 2) {  
3          return n;  
4      }  
5  
6      return n * fact(n - 1);  
7  }
```

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

Let us put together all the techniques discussed above and compute the running time of some example programs.

### 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  $\Theta(n^4)$ .

## References

---

1. Brassard, G., & Bratley, P. (2008). Fundamentals of Algorithmics. New Delhi: PHI Learning Private Limited.