# Analyzing the Runtime of Functions

This document provides a few examples of analyzing the runtime of functions. Instead of using C++ style code, we use common pseudocode. It is important to be able to analyze functions regardless of language specifics. The main focus is on the contribution of loops (for-loops and while-loops) to the runtime. Please report any confusions or typos to `ben.chugg@alumni.ubc.ca`.

The basic idea behind analyzing function runtime is to count the number of instructions executed which *depend on the input to the function*. For all intensive purposes, any work done by a function which is independent of the input—even if it seems like a rather large amount of work—does not contribute to the analysis.

Consider, for example, the following function.

**function** USELESS($n$)
  $x \leftarrow 0$
  **for** $i \leftarrow 1$ to $10^5$, $i++$ **do**
    $x \leftarrow 1$
  **end for**
  **return** $x$
**end function**

What's the runtime of this function? Well, regardless of the input $n$, the for-loop runs $10^5$ times, and two additional instructions are executed (setting $x$ and returning $x$). Therefore, for any $n$, this function executes $10^5 + 2 \in \Theta(1)$ instructions, so this function runs in constant time!

It is important to be comfortable concerning what counts as a constant time instruction. Once you decide what takes constant time in your function, all that remains is to count up how many times this constant time procedure is performed with respect to the input, and boom! we have our runtime. As a general rule, variable assignments (e.g., $x \leftarrow b$), comparisons (e.g., $x == 10$), arithmetic operations (e.g., $x + x$, $x^2$ mod 4) and return statements take constant time.

---

**Example 1.** For the following three code blocks, suppose the function in question has variables $x, a$, an array $A$, and has input $n$. How long does the following code take to run?

**if** $x * a^2 == \sqrt{n}$ **then**
  $x \leftarrow A[i]$
**end if**

Well, $x * a^2 == \sqrt{n}$ is a combination of arithmetic operations and array access is constant time, so $x \leftarrow A[i]$ is constant time. Therefore this entire if statement takes constant time, $\Theta(1)$. What about the following, where $c$ is some constant?

**if** $n == 0$ **then** $x \leftarrow x^2$
**else if** $n == 1$ **then** $x \leftarrow 0$
**else if** $n == 2$ **then** $x \leftarrow x^2$
**else if** $n == 3$ **then** $x \leftarrow 0$
  $\vdots$
**else if** $n == c$ **then** $x \leftarrow 1$
**end if**

Again, regardless of the input, each of these if statements takes constant time, $\Theta(1)$, and we execute at most $c$ of them, where $c$ is a constant, so this entire block still takes constant time ($\Theta(c) = \Theta(1)$). Okay, last example. Suppose DOSOMETHING is a function and consider

**if** DOSOMETHING($n$) == 5 **then**
  $x = a$
**end if**

The runtime of this statement depends on the runtime of DOSOMETHING($n$). To evaluate, DOSOMETHING($n$) == 5, we need to evaluate DOSOMETHING($n$). If its runtime is $O(T(n))$, then this code block will also have

running time $O(T(n))$ (since everything else takes constant time).
**End Example.**

---

Okay, now that we've got that sorted, let's move on to some more interesting examples. Many of the functions you will analyze in CPSC 221 will have time complexities determined by loops; either for-loops or while-loops which depend in some way on the input size $n$.

---

**Example 2.** Let's just consider one for-loop for now. Again, suppose the input to all these functions is $n$. Consider,

**for** $i \leftarrow 1$ to $n$, $i++$ **do**
  **if** $x \mod 2 == 0$ **then**
    $x \leftarrow x + 7$
  **end if**
**end for**

Now that we're experts with operations that take constant time, we know that this if-statement falls in that category. Therefore, we think of this function as equivalent in runtime to

**for** $i \leftarrow 1$ to $n$, $i++$ **do**
  c
**end for**

for some constant $c$. Awesome, that's not bad. We execute the inside of the loop exactly $n$ times, since that's how many times the for-loop iterates. Hence, the runtime is $nc \in \Theta(n)$. A more explicit way to count the number of operations is to add everything together:

$$\underbrace{c + c + \ldots + c}_{\#\text{ of times the for loop executes}} = \underbrace{c + c + \ldots + c}_{n \text{ times}} = \sum_{i=1}^{n} c = c \sum_{i=1}^{n} 1 = cn \in \Theta(n)$$

Sums are a very natural way to count the number of executions of a loop so you should try and get comfortable with the notation.

Maybe we got lucky for the last example. What happens if $i$ doesn't increment by one each time we iterate? For example,

**for** $i \leftarrow 2$ to $n$, $i+ = i$ **do**
  c
**end for**

Here we're multiplying $i$ by 2 every iteration. So how many times will the loop execute? The sequence of values of $i$ looks like $2, 4, 8, \ldots, 2^j$, where $i = 2^j$ on the $j$-th iteration. We stop as soon as $i = 2^j > n$. So what's $j$? Manipulating $2^j > n$, we get $j > \log(n)$, so we iterate $\lfloor \log n \rfloor$ times. However, floors and ceilings don't matter when it comes to asymptotic analysis. Therefore, the total runtime is $\Theta(c \log n) = \Theta(\log n)$, since $c$ is a constant.

We got the answer, but it's useful to try and do this example with summation notation, in an attempt to get familiar with this method. We iterate until $j > \log n$, and at each iteration we do $c$ work, therefore our runtime is:

$$\sum_{j=1}^{\log n} c = c \sum_{j=1}^{\log n} 1 = c \log n \in \Theta(\log n).$$

It is important to remark that the index of the summation represents how many times the for-loop iterates, not the index of the for-loop itself. For example, it would be incorrect to write that the runtime is given by

$$\sum_{i=1}^{n} c,$$

because there's no way to let the summation know that $i$ is actually being multiplied by 2 each iteration and not just incremented by one.

Finally, it was kind of convenient that we started at $i = 2$ for the last example. What if we start at $i = 1$, or $i = a$ for any constant $a$? It doesn't change the running time! This is good—our intuition at this point should be telling us that constant factors shouldn't contribute to anything.

If we started at $i = a$, then the sequence of values of $i$ looks like $a, 2a, 4a, \ldots, 2^{j-1}a$, where on the $j$-th iteration $i = a2^{j-1}$. Now, we stop when $a2^{j-1} > n$. Since $2^{j-1} = 2^j/2$ we can write this as $2^j > 2n/a$, so $j > \log(2n/a)$. However, 2 and $a$ are just constants, so $\log(2n/a) \in \Theta(\log n))$. Therefore this will lead to the same running time.
**End Example.**

---

Now that we see that sums will be useful in our analysis, we should get comfortable with a few common results.

**Fact 1** (Arithmetic Series)**.** Also known as Gauss' formula,

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1).$$

Moreso than the constants involved, the important observation is that this sum is in $\Theta(n^2)$.

**Fact 2** (Geometric Series)**.** For a constant $r$, we have

$$\sum_{k=1}^{n} r^k = \frac{r(r^n - 1)}{r - 1}.$$

Again, what's important is that this sum is in $\Theta(r^n)$, i.e., it is exponential (in $n$).

It is also useful to observe that $\sum_{j=i}^{n} = n - i + 1$, and that sums are linear. That is, if $a, b$ are constants, then

$$\sum_{k=1}^{R}(aT(k) + b) = a\sum_{k=1}^{R} T(k) + b\sum_{k=1}^{R} 1.$$

If that seems confusing right now, don't worry about it. It will become clear why we need it later.

We should notice at this point that nothing in our analyses rely on the fact that loop indices are being incremented as opposed to decremented. Furthermore, the analyses of while loops are for-loops are the same. Convince yourself that

**for** $i \leftarrow n$ to 1, $i--$ **do**
    c
**end for**

still has running time $\Theta(n)$, and that

$i \leftarrow n$
**while** $i > 1$ **do**
    $c$
    $i = i/2$
**end while**

still has running time $\Theta(\log n)$. For this last one, if you're lost, consider again the sequence of values of $i$. It looks like $n, n/2, n/4, \ldots, n/2^j$, until $n/2^j \leq 1$. Which $j$ solves this equation?

Time to get a little more complicated—let's look at some nested for loops.

---

**Example 3.** Consider two nested for loops:

**for** $i \leftarrow 1$ to $n$,$i++$ **do**
  **for** $j \leftarrow i$ to $n$, $j++$ **do**
    $x \leftarrow x + 1$
  **end for**
**end for**

Again, the operation being executed inside the two-loops is a constant time operation, so we can replace it with a constant $c$. Let's focus on the inner for loop for now. Fix some arbitrary $1 \leq i \leq n$. The inner for loop looks like:

**for** $j \leftarrow i$ to $n$, $j++$ **do**
  $c$
**end for**

We've analyzed something like this before, the index just started at 1 instead of $i$. This time, therefore, we start from $i$, and get

$$\sum_{j=i}^{n} c = c \sum_{j=i}^{n} 1 = c(n - i + 1).$$

This is the amount of work being done on each iteration of the outer for loop. The outer for loop goes from $i = 1$ to $n$, with an increment of one, therefore the total work done is

$$\sum_{i=1}^{n} \text{Amount of work done by inner for loop} = \sum_{i=1}^{n} c(n - i + 1),$$

and since sums are linear we can write this as

$$c\left(n\sum_{i=1}^{n} - \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1\right) = cn^2 - \frac{cn(n+1)}{2} + cn = \frac{cn^2}{2} + cn/2 \in \Theta(n^2),$$

where we've used the arithmetic series formula.

Interestingly, even if $j$ starts at 1 instead of $i$ during each loop, we get the same asymptotic running time:

**for** $i \leftarrow 1$ to $n$,$i++$ **do**
  **for** $j \leftarrow 1$ to $n$, $j++$ **do**
    c
  **end for**
**end for**

Regardless of $i$, the inner for loop then does $cn$ work. To get the total running time, we sum up over all $i$,

$$\sum_{i=1}^{n} cn = cn^2 \in \Theta(n^2).$$

**End Example.**

---

It might be tempting to think that $k$ nested for loops gives a running time of $\Theta(n^k)$. But this isn't necessarily the case. For example, what if one for loop does only a constant amount of work, i.e., it iterates only a constant number of times? Its contribution then won't contribute to the asymptotic running time.

Okay, now what if one of the loops is incremented in a way which is slightly more complicated?

---

**Example 4.** Consider:

$i \leftarrow 1$

**while** $i < n$ **do**
  **for** $j \leftarrow 1$ to $i$, $i++$ **do**
    $c$
  **end for**
  $i = i * 2$
**end while**

Let's get to work. We've dealt with inner loop before. $j$ is being incremented by one on each iteration, so the amount of work done by the inner loop, for any $i$, is

$$\sum_{j=1}^{i} c = ci.$$

Now, how many times does the outer while-loop iterate? Based on an analysis above, $\log(n)$ times. Consider the $k$-th iteration of the while-loop, at which point $i = 2^{k-1}$. The amount of work done by the inner loop for $i = 2^{k-1}$ is $c2^{k-1}$, as per above. Summing over all $\log(n)$ iterations gives the total work of

$$\sum_{k=1}^{\log n} c2^{k-1} = c\frac{2}{2}(2^{\log n} - 1) = c(n-1) \in \Theta(n),$$

where we've used the geometric series formula and recalled that $2^{\log n} = n$.

Let's see how modifying this example slightly gives a different running time. Consider,

$i \leftarrow n$
**while** $i > 1$ **do**
  **for** $j \leftarrow i$ to $n$, $i++$ **do**
    $c$
  **end for**
  $i = i/2$
**end while**

Now we're pros—the work done by the inner for-loop is

$$\sum_{j=i}^{n} c = c(n - i + 1).$$

A similar logic to above applies: the outer while-loop iterates $\log n$ times, and during the $k$-th iteration, $i = n/2^{k-1}$ (write out the sequence of values of $i$ if this seems confusing). Therefore, the total work done is

$$\begin{aligned}\sum_{k=1}^{\log n} c(n - \frac{n}{2^{k-1}} + 1) &= cn\sum_{k=1}^{\log n} 1 - \frac{cn}{2}\sum_{k=1}^{\log n}\frac{1}{2^k} + \sum_{k=1}^{\log n} 1 \\ &= cn\log n - \frac{cn}{2}(1 - 2^{-\log n}) + \log n\end{aligned}$$

The dominant term here is $cn\log n$, hence the running time is $\Theta(n\log n)$ (isn't this fun!). Here we've used the fact that

$$\sum_{i=1}^{n}\frac{1}{2^i} = 1 - 2^{-n},$$

which is very close to 1 as $n$ gets large.
**End Example.**

---

The main point of these examples is to demonstrate that you can handle sequences of nested loops by looking at each loop individually, deriving a nice closed form solution of its runtime and then proceeding to the next loop. Constants, floors and ceilings won't generally change the runtime, so you should feel comfortable throwing those out.