

1 Efficiency

When we talk about the efficiency of a function, we are often interested in the following: as the size of the input grows, how does the runtime of the function change? And what do we mean by “runtime”?

- `square(1)` requires one primitive operation: `*` (multiplication). `square(100)` also requires one. No matter what input `n` we pass into `square`, it always takes one operation.

input	function call	return value	number of operations
1	<code>square(1)</code>	$1 \cdot 1$	1
2	<code>square(2)</code>	$2 \cdot 2$	1
\vdots	\vdots	\vdots	\vdots
100	<code>square(100)</code>	$100 \cdot 100$	1
\vdots	\vdots	\vdots	\vdots
n	<code>square(n)</code>	$n \cdot n$	1

- `factorial(1)` requires one multiplication, but `factorial(100)` requires 100 multiplications. As we increase the input size of `n`, the runtime (number of operations) increases linearly proportional to the input.

input	function call	return value	number of operations
1	<code>factorial(1)</code>	$1 \cdot 1$	1
2	<code>factorial(2)</code>	$2 \cdot 1 \cdot 1$	2
\vdots	\vdots	\vdots	\vdots
100	<code>factorial(100)</code>	$100 \cdot 99 \cdots 1 \cdot 1$	100
\vdots	\vdots	\vdots	\vdots
n	<code>factorial(n)</code>	$n \cdot (n - 1) \cdots 1 \cdot 1$	n

Here are some general guidelines for finding the order of growth for the runtime of a function:

- If the function is recursive or iterative, you can **subdivide** the problem as seen above:
 - Count the number of **recursive calls/iterations** that will be made in terms of input size n .
 - Find how much work is done **per recursive call or iteration** in terms of input size n .

The answer is usually the product of the above two, but be sure to pay attention to control flow!

2 Efficiency

- If the function calls helper functions that are not constant-time, you need to take the runtime of the helper functions into consideration.
- We can ignore **constant** factors. For example $1000000n$ and n steps are both linear.
- We can also ignore smaller factors. For example if h calls f and g , and f is Quadratic while g is linear, then h is Quadratic.
- For the purposes of this class, we take a fairly coarse view of efficiency. All the problems we cover in this course can be grouped as one of the following
 - **Constant:** the amount of time does not change based on the input size. Rule: $n \rightarrow 2n$ means $t \rightarrow t$.
 - **Logarithmic:** the amount of time changes based on the logarithm of the input size. Rule: $n \rightarrow 2n$ means $t \rightarrow t + k$.
 - **Linear:** the amount of time changes based on the logarithm of the input size. Rule: $n \rightarrow 2n$ means $t \rightarrow 2t$.
 - **Quadratic:** the amount of time changes based on the logarithm of the input size. Rule: $n \rightarrow 2n$ means $t \rightarrow 4t$.
 - **Exponential:** the amount of time changes based on the logarithm of the input size. Rule: $n \rightarrow n + 1$ means $t \rightarrow 2t$.

Questions

1.1 What is the efficiency of `bonk`?

```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

1.2 Previously, we looked at the `is_prime` function. Here's the code for it:

```
def is_prime(n):
    if n == 1:
        return False
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

What is the efficiency of `is_prime`?

1.3 What is the efficiency of `mod_7`?

```
def mod_7(n):  
    if n % 7 == 0:  
        return 0  
    else:  
        return 1 + mod_7(n - 1)
```