Overview. This worksheet has worked examples for the running time analysis of small pieces of code, by counting the number of performed operations.

To get the full benefit, go through the examples in the same order they are presented and avoid looking at the solutions and explanations until you have tried to find the running time on your own.

You will find at the end of the worksheet a summary of basic *summations* and *logarithms* rules, which are helpful when analyzing running times for simple pieces of code.

Question. For each of the following pieces of code, find the number of times **op()** is called as a function of the input size **n**. Express your answer in terms of the **Big-Theta** notation.

Single Loops

Ex.1

```
for (i = 10; i < n + 5; i += 2)
      op();
```

Solution. $\Theta(n)$.
op() is called exactly $(n + 5 - 10)/2$ times.

Ex.2

```
for (i = 1; i < n; i *= 2)
      op();
```

```
for (i = n; i > 1; i /= 2)
      op();
```

Solution. $\Theta(\log n)$.
The number of steps needed to get from 1 to n by doubling or from n to 1 by halving is $\log_2 n$.
The base is not important when using the order of growth notation, since logs with different bases are a constant factor away from each other.
(See **A.6** in the Math Cheat Sheet at the end).

Ex.3

```
for (i = 10; i < n + 5; i *= 3)
      op();
```

Solution. $\Theta(\log n)$.
The loop will multiply $i = 10$ by 3 until $10 \times 3^i \geq n + 5$. Solving for i, we get $i = \log_3 \frac{n+5}{10}$.

Ex.4

```
for (i = 0; i < n * n * n; i *= 2)
      op();
```

Solution. $\Theta(\log n)$.
$\log_2 n^3 = 3 \log_2 n$. We can drop the coefficient and ignore the base.

Ex.5

```
for (i = 0; i * i < n; i++)
      op();
```

Solution. $\Theta(\sqrt{n})$.
This is because $\sqrt{n} \times \sqrt{n} = n$.

Nested Independent Loops

Ex.6

```
for (i = 10; i < n; i++)
      for (j = 0; j < n; j += 2)
            op();
```

Solution. $\Theta(n^2)$.
`op()` is called exactly $(n-10) \times \frac{n}{2} = \frac{1}{2}n^2 - 5n$ times.

Ex.7

```
for (i = 0; i < n; i++)
      for (j = 0; j < 100; j++)
            op();
```

Solution. $\Theta(n)$.
`op()` is called $100n$ times, but we ignore the coefficient.

Ex.8

```
for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++)
            op();

      for (j = 1; j < n; j *= 2);
            op();
}
```

Solution. $\Theta(n^2)$.
At each iteration of the outer loop, the first inner loop runs and then the second inner loop runs. Therefore, **`op()`** is called $n \times (n + \log n)$ $= n^2 + n \log n$ times, which is in the order of n^2.

Nested Dependent Loops

Ex.9

```
for (i = 1; i <= n; i++)
      for (j = 1; j <= i; j ++)
            op();
```

Solution. $\Theta(n^2)$.
The inner loop performs 1 iteration when $i = 1$, and 2 iterations when $i = 2$, etc.

Therefore, `op()` is called $1 + 2 + \ldots + n$ times. This can be represented as a summation:

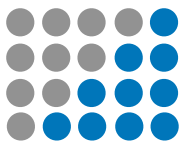
$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \qquad \textbf{Eq. 1}$$

Explanation. To understand why **Eq. 1** is true, note that if we add the numbers twice, we get:

$$\begin{array}{ccccccccccc} & 0 & + & 1 & + & \ldots & + & (n-1) & + & n \\ + & n & + & (n-1) & + & \ldots & + & 1 & + & 0 \\ \hline = & n & + & n & + & \ldots & + & n & + & n \end{array} \longleftarrow n+1 \text{ terms}$$

In other words, $2 \times (0 + 1 + \ldots + n) = n \times (n+1)$, which can be re-organized to give **Eq. 1**

We can also think of this visually as follows:



How many blue circles for $n = 4$?
n rows and $n+1$ columns
area $= n \times (n+1)$
blue circles $= \frac{1}{2} \times n \times (n+1)$

Ex.10

```
for (i = 1; i <= n * n - 10; i++)
      for (j = 1; j <= i; j ++)
            op(m);
```

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

The inner loop performs 1 iteration when $i = 1$, and 2 iterations when $i = 2$, etc.

Therefore, **op()** is performed $1 + 2 + \ldots + (n^2 - 10)$ times. This can be represented as a summation:

$$\sum_{i=0}^{n^2-10} i \text{ , which equals to } = \frac{(n^2-10)\times((n^2-10)+1)}{2} \text{ (using } \textbf{Eq. 1}\text{)}$$

Simplifying the result and dropping the coefficients and lower order terms leads to n^4.

Ex.11

```
for (i = 1; i <= n; i++)
      for (j = 1; j <= i; j++)
            for (k = 1; k <= i; k++)
                  op();
```

Solution. $\Theta(n^3)$.

When $i = 1$, **op()** is performed 1×1 times, when i=2, **op()** is performed 2×2 times, etc. Hence the total number of times **op()** is performed is:

$$1^2 + 2^2 + 3^2 + \ldots + n^2 = \sum_{i=1}^{n} i^2 \sim \frac{1}{3}n^3 \qquad \textbf{Eq. 2}$$

Explanation.

To understand why this summation is in the order of n^3, consider the following two observations:

1. *Upper bound:*

$$\begin{aligned} 1^2 + 2^2 + \ldots + n^2 &\leq n^2 + n^2 + \ldots + n^2 \\ &\leq n \times n^2 \end{aligned}$$

Therefore, the order of growth is not more than n^3.

2. *Lower bound:*

$$\begin{aligned} 1^2 + 2^2 + \ldots + n^2 &\geq (\tfrac{n}{2})^2 + ((\tfrac{n}{2}+1)^2) + \ldots + n^2 \longleftarrow \text{Considering only the 2nd half of the summation.} \\ &\geq (\tfrac{n}{2})^2 + (\tfrac{n}{2})^2 + \ldots + (\tfrac{n}{2})^2 \\ &\geq \tfrac{n}{2} \times (\tfrac{n}{2})^2 \end{aligned}$$

Therefore, the order of growth is not less than $\frac{1}{8}n^3$.

Since we drop constant coefficients, the upper and lower bounds imply that the order of growth is n^3.

See **B.6** in the math cheat sheet for the exact closed form of this summation.

Ex.12

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= i; j++)
        for (k = 1; k <= j; k++)
            op();
```

Solution. $\Theta(n^3)$.
`op()` is called 1 time when $j = 1$, 2 times when $j = 2$, etc. Therefore, `op()` is called $\sum_{j=1}^{i} j$ times.

However, i takes all the values from $1 \rightarrow n$. Therefore, the total number of times `op()` is called is:

$$\sum_{i=1}^{n}\sum_{j=1}^{i} j = \sum_{i=1}^{n} \frac{i(i+1)}{2} \quad \text{(using **Eq. 1**)}$$

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

Dropping lower order terms, we are left with $\frac{1}{2}\sum_{i=1}^{n} i^2$, which is in the order of n^3 (using **Eq. 2**).

Ex.13

```
for (i = 1; i <= n; i *= 2)
    for (j = 1; j <= i; j++)
        op();
```

Solution. $\Theta(n)$.
The inner loop performs 1 iteration when $i = 1$, and 2 iterations when $i = 2$, etc.

Therefore, `op()` is called $1 + 2 + 4 + \ldots + n$ times, because i doubles in each iteration of the outer loop. This equals to $2^0 + 2^1 + \ldots + 2^k$, where $2^k = n$ and can be represented as a summation:

$$\sum_{k=0}^{\lg n} 2^k = 2^{\lg n + 1} - 1 \sim 2n \qquad \textbf{Eq. 3}$$

Using the identity $a^m \times a^n = a^{m+n}$, we can represent the answer as $2^1 \times 2^{\lg n} - 1$

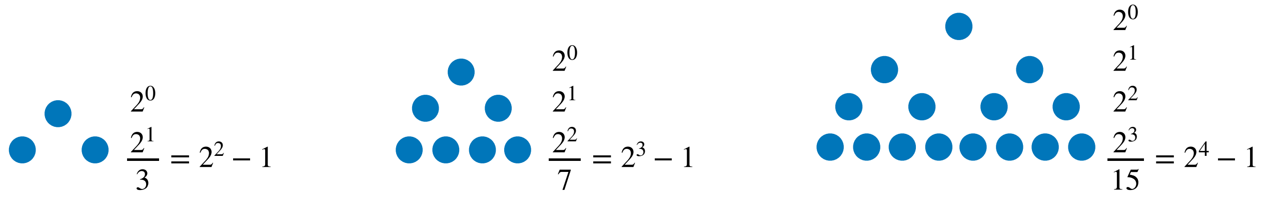
Using the identity $x^{\lg y} = y^{\lg x}$, we can represent the answer as $2^1 \times n^{\lg 2} - 1 = 2^1 \times n^1 - 1$, which is in the order of n.

Explanation.

This is a geometric sum that can be calculated using the geometric sum formula:

$$\sum_{i=0}^{m} r^i = \frac{r^{m+1} - 1}{r - 1} \qquad \textbf{Eq. 4}$$

where $m = \lg n$ and $r = 2$ in this exercise. The following is a visual explanation for this special case:



Ex.14

```
for (i = 1; i <= n; i++)
     for (j = 1; j < i; j *= 2)
          op();
```

Solution. $\Theta(n \log n)$.
The inner loop performs $\lg 1$ iterations when $i = 1$, and $\lg 2$ iterations when $i = 2$, etc.

Therefore, `op()` is called $\lg 1 + \lg 2 + \ldots + \lg n$ times. Using the identity $\log(ab) = \log a + \log b$, we can rewrite the summation as: $\lg 1 + \lg 2 + \ldots + \lg n = \lg(1 \times 2 \times \ldots \times n) = \lg(n!)$.

Using Stirling's Approximation:

$$\lg(n!) \sim n \lg n$$

Eq. 5

Ex.15

```
for (i = 1; i <= n; i++)
     for (j = 1; j <= n; j += i)
          op();
```

Solution. $\Theta(n \log n)$.
The inner loop performs $\frac{n}{1}$ iterations when $i = 1$, and $\frac{n}{2}$ iterations when $i = 2$, etc.

Therefore, `op()` is called $\frac{n}{1} + \frac{n}{2} + \ldots + \frac{n}{n}$, which can be represented as the summation:

$$\sum_{i=1}^{n} \frac{n}{i} = n \times \sum_{i=1}^{n} \frac{1}{i}$$

This is n multiplied by *a Harmonic Number.* Harmonic numbers can be approximated by turning the summation into an integral:

$$\sum_{i=1}^{n} \frac{1}{i} \sim \int_{1}^{n} \frac{1}{i}\, \mathrm{d}i = \ln n$$

Eq. 6

Hence, the total is $\sim n \ln n$.

Ex.16

```
for (i = 1; i <= n; i++) {
     for (j = 1; j <= n; j += i)
          op();

     for (j = 1; j <= i; j++)
          op();
}
```

Solution. $\Theta(n^2)$.
There are two independent inner loops:

- The first inner loop combined with the outer loop are identical to the one in **Ex.15**, which runs in the order of $n \log n$.
- The second inner loop combined with the outer loop are identical the one in **Ex.9**, which runs in the order of n^2.

The total is $n \log n + n^2$, which is order n^2.

Recursive Algorithms

Ex.17

```
void f(int n) {
      if (n == 0) return;
      op();
      f(n-1);
}
```

Solution. $\Theta(n)$.
`op()` is executed once in each of the recursive calls for $n, n-1, \ldots, 2, 1$.
These are exactly n calls.

Ex.18

```
void f(int n) {
      if (n == 1) return;
      op();
      f(n/2);
}
```

Solution. $\Theta(\log n)$.
`op()` is executed once in each of the recursive calls for $n, \frac{n}{2}, \frac{n}{4}, \ldots, 2, 1$.
These are $\lg n$ calls.

Ex.19

```
void f(int n) {
      if (n == 0) return;
      for (int i = 0; i < n; i++)
            op();
      f(n-1);
}
```

Solution. $\Theta(n^2)$.
`op()` is executed n times for `f(n)` and $n-1$ times for `f(n-1)`, etc. This means that `op()` is executed $n + (n-1) + \ldots + 1$ times.
From **Eq. 1**, we know this is in the order of n^2.

Ex.20

```
void f(int n) {
      if (n == 0) return;
      for (int i = 0; i < n; i++)
            op();
      f(n/2);
}
```

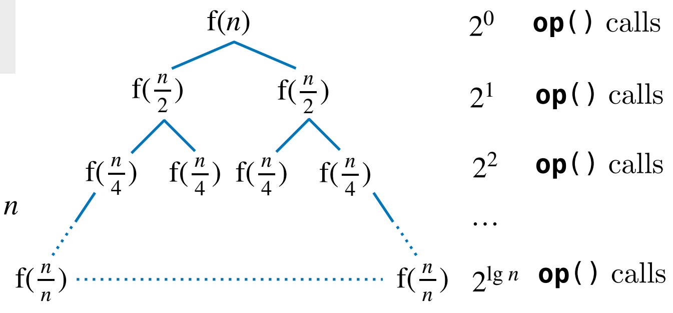
Solution. $\Theta(n)$.
`op()` is executed n times for `f(n)` and $\frac{n}{2}$ times for `f(n/2)`, etc. This means that `op()` is executed $n + \frac{n}{2} + \frac{n}{4} + \ldots + 1$ times. This can be written as $n \times (1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{n})$, which is in the order of n (See **B.8** in the cheat sheet at the end).

Ex.21

```
void f(int n) {
      if (n == 0) return;
      op();
      f(n/2);
      f(n/2);
}
```

`op()` is performed once in each recursive call. The total number of calls to `op()` is $2^0 + 2^1 + 2^2 + \ldots + 2^{\lg n}$, which is in the order of n (see **Eq.3**).

Solution. $\Theta(n)$.
Consider the following visualization for the recursive calls:



Ex.22

```
void f(int n) {
      if (n == 0) return;
      op();
      f(n-1);
      f(n-1);
}
```

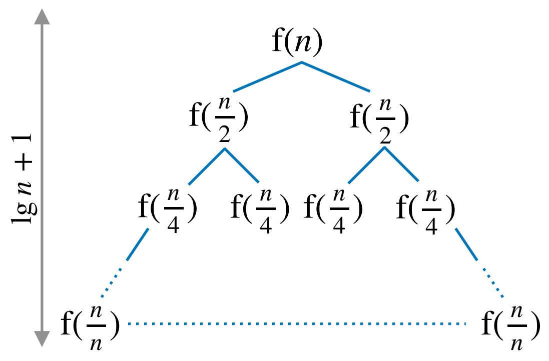
Solution. $\Theta(2^n)$.
As in **Ex.21**, `f(n)` produces a binary tree, but with $n+1$ levels instead of $\lg n + 1$ levels. Since `op()` is performed once in each recursive call, the total number of times `op()` is performed is:
$2^0 + 2^1 + 2^2 + \ldots + 2^n$.

From **Eq.4**, we know this is $2^{n+1} - 1 = 2^1 \times 2^n - 1$, which is in the order of 2^n.

Ex.23

```
void f(int n) {
      if (n == 0) return;
      for (int i = 0; i < n; i++)
            op();
      f(n/2);
      f(n/2);
}
```

Solution. $\Theta(n \lg n)$.
In each call to `f(n)`, `op()` is performed a number of times that depends on the value of the argument n. The visualization below shows the value of n in each of the recursive calls and the number of times `op()` is performed at each level of the recursion tree.
The recursion tree has $\lg n + 1$ levels, and `op()` is performed a total of n times at each level. Therefore, the order of growth is $n \lg n$.


$\lg n + 1$
$f(n)$
$f(\frac{n}{2})$ $f(\frac{n}{2})$
$f(\frac{n}{4})$ $f(\frac{n}{4})$ $f(\frac{n}{4})$ $f(\frac{n}{4})$
$f(\frac{n}{n})$ $f(\frac{n}{n})$

n	`op()` calls
$\frac{n}{2} + \frac{n}{2} = n$	`op()` calls
$4 \times \frac{n}{4} = n$	`op()` calls
$\ldots$	
$n \times \frac{n}{n} = n$	`op()` calls

Ex.24

```
void f(int n) {
      if (n == 0) return;
      for (int i = 0; i < n; i++)
            op();
      for (int i = 0; i < 4; i++)
            f(n/4);
}
```

Solution. $\Theta(n \log n)$.
This is the same as **Ex.23**, but the height of the tree is $\log_4 n$ instead of $\log_2 n$. The number of times `op()` is performed at each level is still n. Therefore, the total is $n \log_4 n$ and the order of growth is $n \log n$.

A. Logarithms

1. $a^b = c \rightarrow b = \log_a c$ ⟵ Definition
2. $\log_b b = 1, \log_b 1 = 0$ ⟵ Special cases
3. $\log_b(\frac{x}{y}) = \log_b x - \log_b y$
4. $\log_b(x \times y) = \log_b x + \log_b y$
5. $\log_b x^y = y \times \log_b x$ ⟵ Follows directly from the previous rule.
6. $\log_b x = \dfrac{\log_c x}{\log_c b}$ ⟵ Changing bases
7. $x^{\log_b y} = y^{\log_b x}$
8. $b^{\log_b x} = x$ ⟵ Follows directly from the previous rule.
9. $\lg(n!) = \sim n \lg n$ ⟵ Stirling's Approximation

Notation:

- $\log n$ (no base) $\rightarrow$ used with orders of growth to indicate that the base is not important. Logarithms with different bases differ by a constant factor as shown in the listed identities.
- $\lg n \rightarrow$ base 2.
- $\ln n \rightarrow$ natural logarithm (base is e)

B. Summations

1. $\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$ ⟵ Definition
2. $\sum_{i=1}^{n} c = c + c + \ldots + c = c \times n$ ⟵ If c does not depend on i.
3. $\sum_{i=1}^{n} c \times f_i = c \times \sum_{i=1}^{n} f_i$
4. $\sum_{i=1}^{n} f_i + g_i = \sum_{i=1}^{n} f_i + \sum_{i=1}^{n} g_i$
5. $\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = \dfrac{n(n+1)}{2}$
6. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \ldots + n^2 = \dfrac{n(n+1)(2n+1)}{6}$
7. $\sum_{i=0}^{n} r^i = r^0 + r^1 + r^2 + \ldots + r^n = \dfrac{r^{n+1} - 1}{r - 1}, \quad r \neq 1$ ⟵ Geometric Sum.
8. $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1\ , \quad \sum_{i=0}^{n} (\frac{1}{2})^i = 1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^n} = \sim 2$ ⟵ Special cases of a geometric sum ($r = 2$ and $r = 0.5$).
9. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \sim \int_1^n \frac{1}{i}\, \mathrm{d}i = \ln n$ ⟵ Harmonic Number H_n.

Prepared by Ibrahim Albluwi (isma@cs.princeton.edu). Many thanks to Kevin Wayne for providing very valuable feedback.