CSCI 570: Asymptotic Notation

Additional reading: K/T textbook Ch. 2, G/T textbook Ch. 1, Rosen textbook Ch. 3

`Find-Max`
Input: an array A of n comparable values, denoted $A_1 \dots A_n$
Output: the value of the largest element of A.

```
max = A_1
for i = 2 → n do
    if max < A_i then
        max = A_i
return max
```

How many lines of code get executed when `Find-Max` is run, as a function of n?

Formal Definition: Given worst-case runtime of $T(n)$ for size n, we say that $T(n)$ is $O(f(n))$ if $T(n)$ has some constant $\times$ $f(n)$ as an upper bound for sufficiently large n.
More formally, $T(n)$ is $O(f(n))$ if $\exists_{c>0,n_0\geq 0}$ such that $\forall_{n\geq n_0}$, $T(n) \leq cf(n)$

- What is the running time of `Find-Max` in O-notation?
- What are the advantages and disadvantages of stating the running time for an algorithm in O-notation instead of giving an exact value?

What is the running time of this function?

`Closest-Pair`
Input: n points in $2D$-space
Output: The pair that has the smallest distance between them.

```
min = ∞
for i = 2 → n do
    for j = 1 → i − 1 do
        if (x_j − x_i)^2 + (y_j − y_i)^2 < min then
            min = (x_j − x_i)^2 + (y_j − y_i)^2
            closestPair = ((x_i, y_i), (x_j, y_j))
return closestPair
```

Other Definitions:

- $T(n)$ is $\Omega(f(n))$ is similar for lower bound.
  $T(n)$ is $\Omega(f(n))$ if $\exists_{\varepsilon>0, n_0 \geq 0}$ such that $\forall n \geq n_0$, $T(n) \geq \varepsilon \times f(n)$.
- $T(n)$ is $\Theta(f(n))$ if it is both $O(f(n))$ and $\Omega(f(n))$.
- Equivalently, $T(n)$ is $\Theta(f(n))$ if the following limit exists and:

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = c > 0$$

Properties of Asymptotic Growth Rates

- Transitivity:
  - If f is $O(g)$ and g is $O(h)$, then f is $O(h)$
  - If f is $\Omega(g)$ and g is $\Omega(h)$, then f is $\Omega(h)$
- Sums of functions
  - If f is $O(h)$ and g is $O(h)$, then $f + g$ is $O(h)$

Desired properties of algorithms

Ideally, we want every algorithm we write to have the following two characteristics:

- **Efficient**: if we run the algorithm on inputs that are larger, the running time scales well. For example, if we double the input size, and it causes the program to take twice as long to run, that's fine. It's even okay if it's some other constant. What we *don't* want is for small manipulations (such as an input only one bit larger) to cause the running time to double.
  - Problems that can be solved by polynomial time algorithms tend to have solutions with very small polynomials; while there are exceptions, "polynomial time" tends to be a practical equivalent to "efficient".
- **Correct/Optimal**: the algorithm returns the correct answer for every input. If multiple outputs are valid, it returns the "best" valid solution (or one of many that are equally best, if multiple qualify).
  - How do we prove an algorithm **is not** correct?

Review: Order the following functions from smallest asymptotic running time to largest. In addition, identify all pairs of functions f_x and f_y such that $f_x = \Theta(f_y)$, or state that none exist.

1. $f_a = 4n^{3/2}$
2. $f_b = \lfloor 2n \log^2 n \rfloor$
3. $f_c = n^{0.007}$
4. $f_d = \sum_{k=1}^{n} 100k^2$
5. $f_e = \log^{10} n$
6. $f_f = \pi^{100}$
7. $f_g = n^3/10000$
8. $f_h = n^2 \log n$
9. $f_i = e^n$