Problem 1: Comparing Algorithm A and B

To determine when Algorithm A becomes faster than Algorithm B, we need to solve the inequality:

$1000n^3 < 2^n$

So, this problem examines polynomial vs exponential growth. Algorithm A runs in **$O(n^3)$** time, while the growth of Algorithm B is exponential **$O(2^n)$.** Although A seems bad with the big constant (1000), it is true that exponential functions eventually outpace every polynomial (goes white hot in the output). So, we can check which value of **n** (reverting to Algorithm A is faster (i.e. lower total number of steps):

- At **n = 5**:

  A = 1000 * 125 = 125,000

  B = 2^5 = 32

- At **n = 10**:

  A = 1000 * 1000 = 1,000,000

  B = 2^{10} = 1024

Clearly, Algorithm B is still faster at small n.

Keep testing:

- At **n = 15**:

  A = 1000 * 3375 = 3,375,000

  B = 2^{15} = 32,768

- At **n = 20**:

  A = 1000 * 8000 = 8,000,000

  B = 2^{20} = 1,048,576

Now, B is growing rapidly. Finally:

- At **n = 29**:

  A = 1000 * 24,389 = 24,389,000

  B = 2^{29} = 536,870,912

Algorithm A uses fewer steps when compared to Algorithm B, so we can see that when **n ≥ 29**, Algorithm A runs faster than Algorithm B. This is consistent with the theoretical conclusion that functions that are exponential will eventually outgrow a polynomial for sufficiently high values of n, no matter the constant (Cormen et al., 2009).

Problem 2: Time Complexity of Nested Loops

Let's analyze the time complexity of the code:

```java
for(int i = 0; i < n; i++) {
   for(int j = 0; j < i; j++){
      //do swap stuff, constant time
   }
```

```
}
```

The outer loop runs **n** times. The inner loop runs **i** times for each iteration of **i**. Therefore, the total number of operations is:

$0 + 1 + 2 + ... + (n - 1) = n (n - 1)/2$

This adds up to **$O(n^2)$,** which is the **upper bound** of this code fragment. Each instantiation of the inner loop is thus guaranteed to run a maximum of **i - 1** times, each time performing a constant-time operation, and so no fewer than this in the worst case.

As for the **lower bound**, it is also **$\Omega(n^2)$** as this is the exact summation pattern of the total operations. This is why the upper and lower bounds are equal. Thus, pretty much the **tight bound (Θ notation)** to give is **$\Theta(n^2)$**, which means that algorithm's growth rate is bounded quadratically both from above and below.

This assumption is justified by the consistent growth behavior. The takeaway is complexity is not just an abstract measure, performance directly scales with it as input sizes are increased, a basic principle from computation theory (Goodrich & Tamassia, 2014).

Conclusion:

By analyzing both problems, we've explored how exponential growth eventually outgrows polynomial growth, and how a pair of nested loops contribute to quadratic time complexity. There are also many examples to solidify both theoretical and practical understanding of what asymptotic analysis is and how it helps to better understand the

characteristic of an algorithm. Working together and cross-comparing varies methodologies, we can get a better understanding of how algorithms act when input sizes increase.

Wordcount: 531

References:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. https://mitpress.mit.edu/9780262533058/introduction-to-algorithms/

Goodrich, M. T., & Tamassia, R. (2014). *Algorithm Design and Applications*. Wiley. https://www.wiley.com/en-us/Algorithm+Design+and+Applications%2C+1st+Edition-p-9781118335918