Let $f(n) = 3n^2 + 17n \log_2 n + 1000$. Which of the following are true?

A. $f(n)$ is $O(n^2)$.

B. $f(n)$ is $O(n^3)$.

C. Both A and B.

D. Neither A nor B.

Analysis of algorithms: quiz 1

Choose $c = 10^2$, $n_0 = 1$.
Properties of Big-O Notation

Claim (Transitivity): If \( f \) is \( O(g) \) and \( g \) is \( O(h) \), then \( f \) is \( O(h) \).

Proof: we know from the definition that
\[
\begin{align*}
& f(n) \leq c g(n) \text{ for all } n \geq n_0 \\
& g(n) \leq c' h(n) \text{ for all } n \geq n_0'
\end{align*}
\]
Therefore
\[
\begin{align*}
& f(n) \leq c g(n) \text{ if } n \geq n_0 \\
& \leq c \cdot c' h(n) \text{ if } n \geq n_0 \text{ and } n \geq n_0' \\
& = \frac{c'}{c} h(n) \text{ if } n \geq \max\{n_0, n_0'\}
\end{align*}
\]

Know how to do proofs using Big-O definition.

Consequences of Additivity

- OK to drop lower order terms. E.g., if
  \[
  f(n) = 4.1n^3 + 23n + n \log n
  \]
  then \( f(n) \) is \( O(n^3) \)
- Polynomials: Only highest degree term matters. E.g., if
  \[
  f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d, \quad a_d > 0
  \]
  then \( f(n) \) is \( O(n^d) \)

Other Useful Facts: Log vs. Poly vs. Exp

Fact: \( \log_b(n) \) is \( O(n^d) \) for all \( b, d > 0 \)
All polynomials grow faster than logarithm of any base

Fact: \( n^d \) is \( O(r^n) \) when \( r > 1 \)
Exponential functions grow faster than polynomials

Exercise: Prove these facts!

Logarithm review

Definition: \( \log_b(a) \) is the unique number \( c \) such that \( b^c = a \)
Informally: the number of times you can divide \( a \) into \( b \) parts until each part has size one

Properties:
- Log of product \( \rightarrow \) sum of logs
  - \( \log(xy) = \log x + \log y \)
  - \( \log(x^e) = e \log x \)
- \( \log_b(\cdot) \) is inverse of \( b^{(\cdot)} \)
  - \( \log_b(b^c) = c \)
  - \( b^{\log_b(n)} = n \)
  - \( \log_b a \cdot \log_b b = \log_b n \) (logs in any two bases are proportional)

When using big-O, it’s OK not to specify base. Assume \( \log_2 \) if not specified.

Big-O comparison

Which grows faster?
\[
\begin{align*}
n(\log n)^3 & \text{ vs. } n^{4/3} \\
simplifies to \quad (\log n)^3 & \text{ vs. } n^{1/3} \\
simplifies to \quad \log n & \text{ vs. } n^{1/9}
\end{align*}
\]
- We know \( \log n = O(n^d) \) for all \( d \)
- \( \Rightarrow \log n = O(n^{1/9}) \)
- \( \Rightarrow n(\log n)^3 = O(n^{1/3}) \)

Apply transformations (monotone, invertible) to both functions.
Try taking log.
Exponential time

An algorithm is *exponential time* if it is $O(2^{nk})$ for some $k > 0$

*Useful fact:* (Stirling’s approximation)

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$  (ratio tends to 1)

*Exercise:* What can you claim from here for big-O (and later big-Θ)?

---

**Analysis of algorithms: quiz 4**

Which is an equivalent definition of exponential time?

A. $O(2^n)$

B. $O(2^{cn})$ for some constant $c > 0$.

C. Both A and B.

D. Neither A nor B.

---

**Big-Ω Motivation**

Algorithm `foo`

```python
for i = 1 to n do
    for j = 1 to n do
        do something...
    end for
end for
```

Fact: run time is $O(n^3)$

Conclusion: `foo` and `bar` have the same asymptotic running time. What is wrong?

---

**More Big-Ω Motivation**

Algorithm `sum-product`

```python
sum = 0
for i = 1 to n do
    for j = i to n do
    end for
end for
```

What is the running time of `sum-product`?

Easy to see it is $O(n^2)$. Could it be better? $O(n)$?

---

**Big-Ω**

Informally: $T$ grows at least as fast as $f$

*Definition:* The function $T(n)$ is $\Omega(f(n))$ if there exist constants $c \geq 0$ and $n_0 \geq 0$ such that

$$T(n) \geq cf(n) \text{ for all } n \geq n_0$$

$f$ is an *asymptotic lower bound* for $T$.

---

**Analysis of algorithms: quiz 2**

Which is an equivalent definition of big Omega notation?

A. $f(n)$ is $\Omega(g(n))$ iff $g(n)$ is $O(f(n))$.

B. $f(n)$ is $\Omega(g(n))$ iff there exist constants $c > 0$ such that $f(n) \geq c \cdot g(n) \geq 0$ for infinitely many $n$.

C. Both A and B.

D. Neither A nor B.
Big-Ω

Exercise: let $T(n)$ be the running time of sum-product.
Show that $T(n)$ is $\Omega(n^2)$

Algorithm sum-product

```plaintext
sum = 0
for $i = 1$ to $n$
  for $j = i$ to $n$
  end for
end for
```

Big-Θ

Definition: the function $T(n)$ is $\Theta(f(n))$ if there exist positive constants $c_1$, $c_2$ and $n_0$ such that

$0 \leq c_1 f(n) \leq T(n) \leq c_2 f(n)$ for all $n \geq n_0$

$f$ is an asymptotically tight bound of $T$

Exercise: solution

Hard way

- Count exactly how many times the loop executes

$1 + 2 + \ldots + n = \frac{n(n+1)}{2} = \Omega(n^2)$

Easy way

- Ignore all loop executions where $i > n/2$ or $j < n/2$
- The inner statement executes at least $(n/2)^2 = \Omega(n^2)$ times

Analysis of algorithms: quiz 3

Which is an equivalent definition of big Theta notation?

A. $f(n)$ is $\Theta(g(n))$ iff $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$.

B. $f(n)$ is $\Theta(g(n))$ iff $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$ for some constant $0 < c < \infty$.

C. Both A and B.

D. Neither A nor B.

Big-Θ example

How do we correctly compare the running time of these algorithms?

Algorithm foo

```plaintext
for $i = 1$ to $n$
  for $j = 1$ to $n$
    do something
    do something else
  end for
end for
```

Algorithm bar

```plaintext
for $i = 1$ to $n$
  for $j = 1$ to $n$
    do something
  end for
end for
```

Answer: foo is $\Theta(n^2)$ and bar is $\Theta(n^3)$.
They do not have the same asymptotic running time.
Additivity Revisited

Suppose $f$ and $g$ are two (non-negative) functions and $f$ is $O(g)$

Old version: Then $f + g$ is $O(g)$

New version: Then $f + g$ is $\Theta(g)$

Example: $\frac{n^2 + 42n + n \log n}{g}$ is $\Theta(n^2)$

Running Time Analysis

Mathematical analysis of worst-case running time of an algorithm as function of input size. Why these choices?

- Mathematical: describes the algorithm. Avoids hard-to-control experimental factors (CPU, programming language, quality of implementation), while still being predictive.
- Worst-case: just works. ("average case" appealing, but hard to analyze)
- Function of input size: allows predictions. What will happen on a new input?

Efficiency

When is an algorithm efficient?

Stable Matching Brute force: $\Omega(n!)$
Propose-and-Reject?: $O(n^2)$

We must have done something clever
Question: Is it $\Omega(n^2)$?

Polynomial Time

Definition: an algorithm runs in polynomial time if its running time is $O(n^d)$ for some constant $d$

Polynomial Time: Examples

These are polynomial time:
- $f_1(n) = n$
- $f_2(n) = 4n + 100$
- $f_3(n) = n \log(n) + 2n + 20$
- $f_4(n) = 0.01n^2$
- $f_5(n) = n^2$
- $f_6(n) = 20n^2 + 2n + 3$

Not polynomial time:
- $f_7(n) = 2^n$
- $f_8(n) = 3^n$
- $f_9(n) = n!$

Why Polynomial Time?

Why is this a good definition of efficiency?

- Matches practice: almost all practically efficient algorithms have this property.
- Usually distinguishes a clever algorithm from a “brute force” approach.
- Refutable: gives us a way of saying an algorithm is not efficient, or that no efficient algorithm exists.