Algorithm Analysis

Which takes longer?

```
void pushAll(int k) {
   for (int i=0;
         i<= 100*k;
         i++)
   {
      stack.push(i);
   }
}
```

```
void pushAdd(int k) {
   for (int i=0; i<= k; i++)
   {
      for (int j=0; j<= k; j++){
             stack.push(i+j);
      }
   }
}
```

100k push operations

k^2 push operations

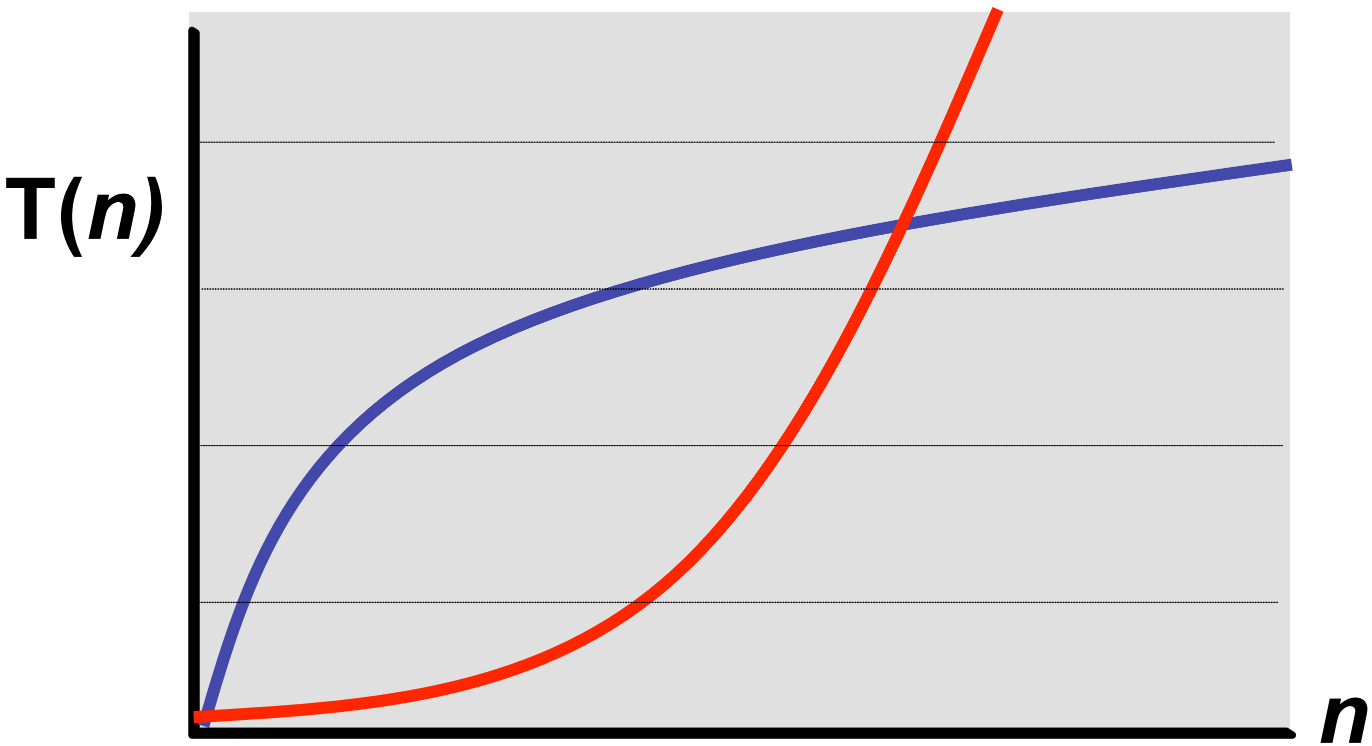
Which grows faster?

$f(k) = 100k$	$f(k) = k^2$
$f(0) = 0$	$f(0) = 0$
$f(1) = 100$	$f(1) = 1$
$f(100) = 10{,}000$	$f(100) = 10{,}000$
$f(1000) = 100{,}000$	$f(1000) = 1{,}000{,}000$

Big-O Notation

How does an algorithm scale?

- For large inputs, what is the running time?
- T(n) = running time on inputs of size n



Why Big-O notation?

Example:

- Downloading a file:
  - 3 seconds to setup a connection
  - 1.5Kbytes/second

- Document Distance: quadratic vs linear.

- Exponential time algorithms cannot run for n>100.

Big-O Notation

Definition: T(n) = O(f(n)) if T grows no faster than f

$T(n) = O(f(n))$ if:

- there exists a constant $c > 0$
- there exists a constant $n_0 > 0$

such that for all $n > n_0$:

$T(n) \leq cf(n)$

Big-O Notation

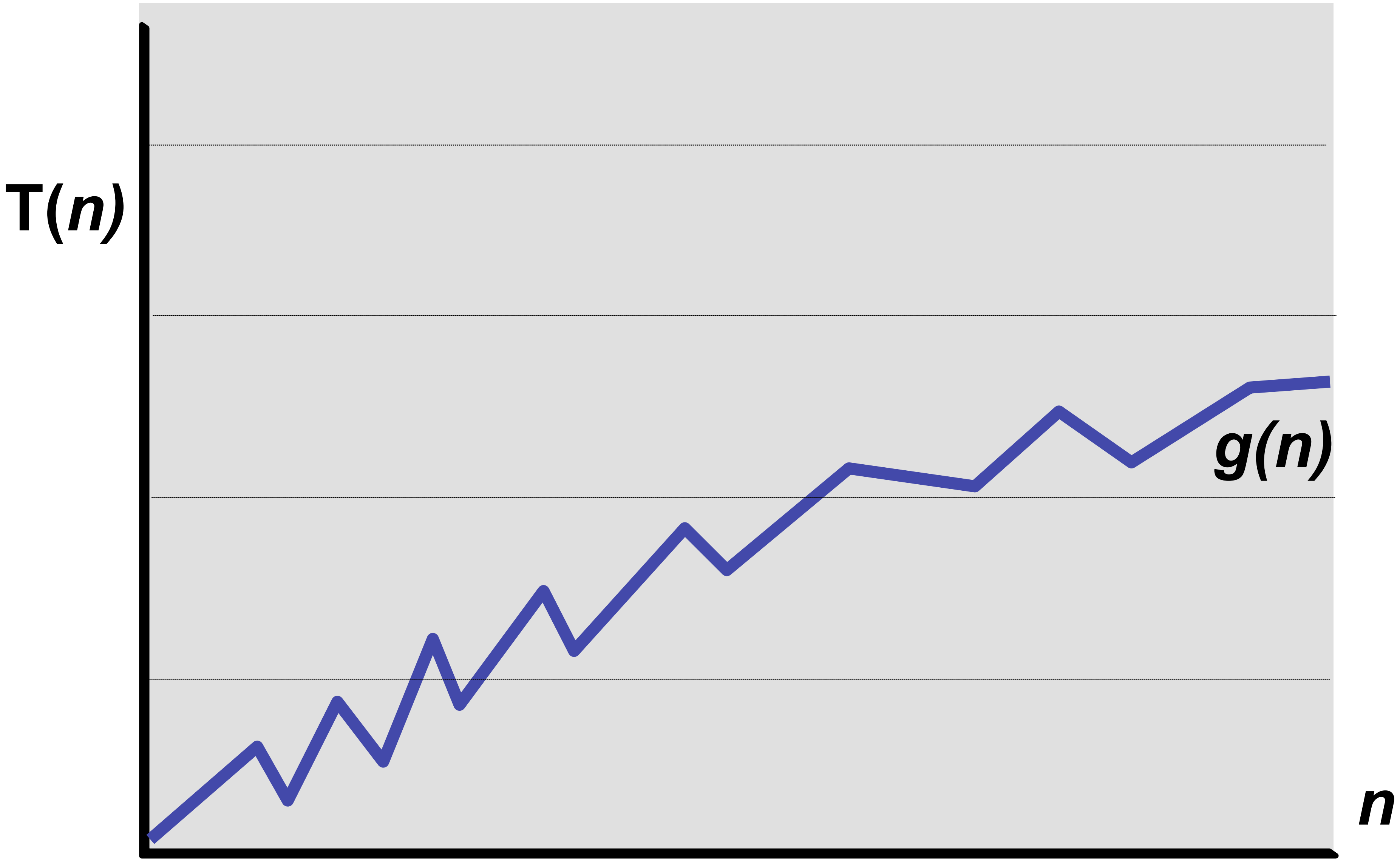
Example:

$T(n) = 4n^2 + 24n - 16$

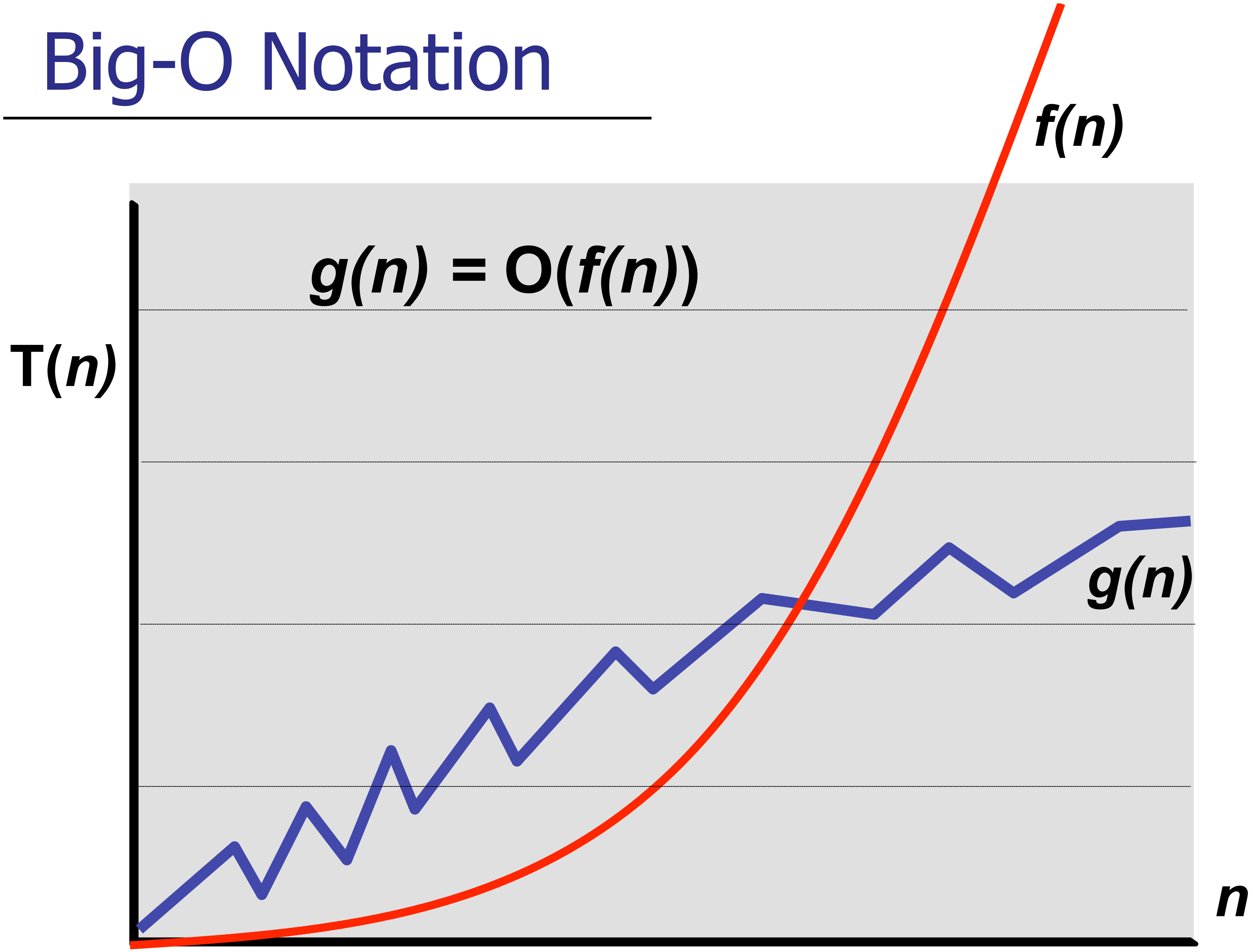
$< 28n^2 \quad$ (for $n_0 > 0$)

$= O(n^2) \quad$ (for $c = 28$)

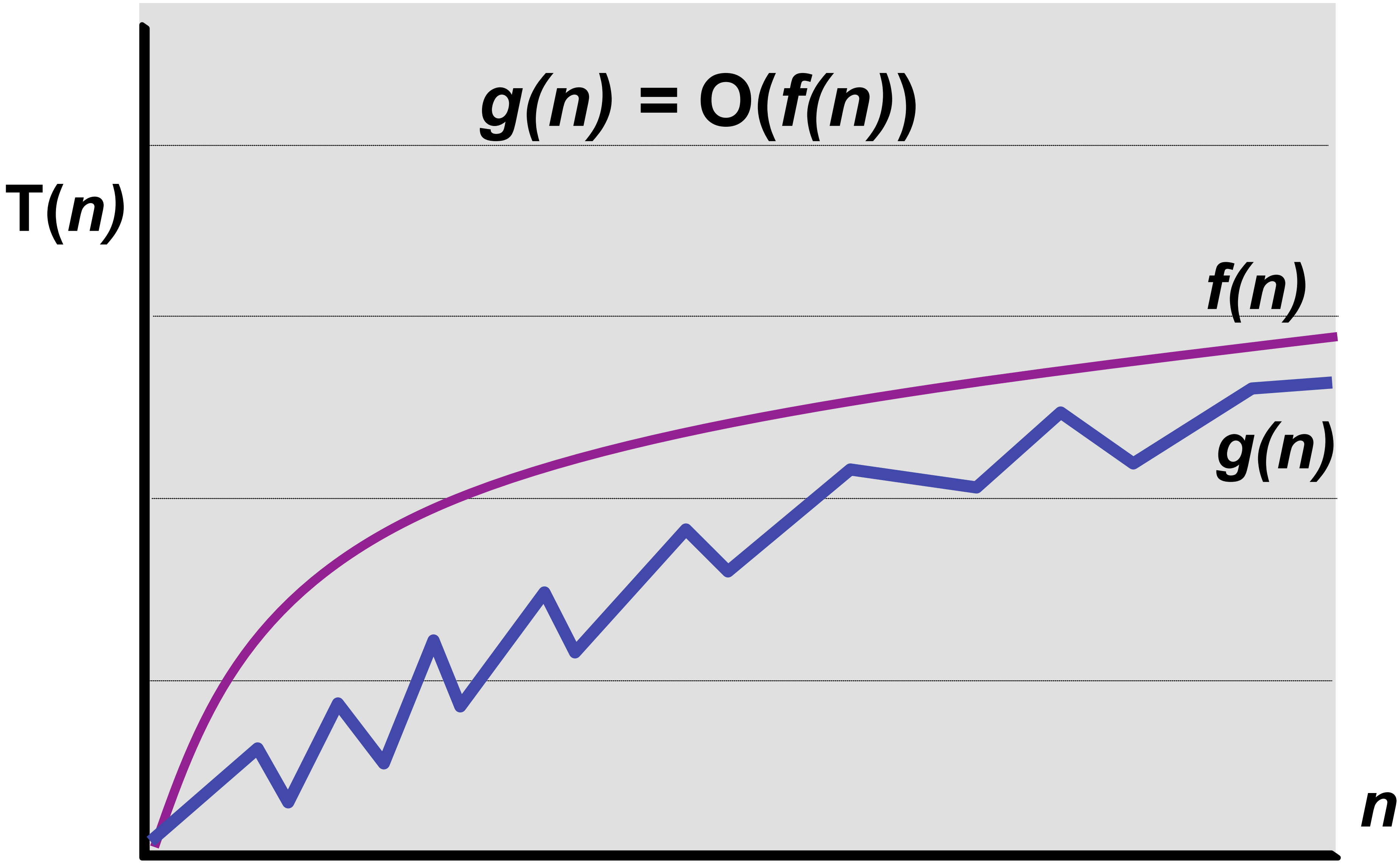
Big-O Notation



Big-O Notation



Big-O Notation



Example

T(n)	f(n)	big-O
$T(n) = 1000n$	$f(n) = n$	$T(n) = O(f(n))$
$T(n) = 1000n$	$f(n) = n^2$	$T(n) = O(f(n))$
$T(n) = n^2$	$f(n) = n$	$T(n) \neq O(f(n))$
$T(n) = 13n^2 + n$	$f(n) = n^2$	$T(n) = O(f(n))$

Example

$T(n)$	$f(n)$	big-O
$T(n) = 1000n$	$f(n) = n$	$T(n) = O(n)$
$T(n) = 1000n$	$f(n) = n^2$	$T(n) = O(n^2)$
$T(n) = n^2$	$f(n) = n$	$T(n) \neq O(n)$
$T(n) = 13n^2 + n$	$f(n) = n^2$	$T(n) = O(n^2)$

Big-O Notation

Definition: $T(n) = O(f(n))$ if T grows no faster than f

$\boldsymbol{T(n) = O(f(n))}$ if:

- there exists a constant $c > 0$
- there exists a constant $n_0 > 0$

such that for all $n > n_0$:

$$\boldsymbol{T(n) \leq cf(n)}$$

Big-O Notation

Definition: $T(n) = \Omega(f(n))$ if T grows no slower than f

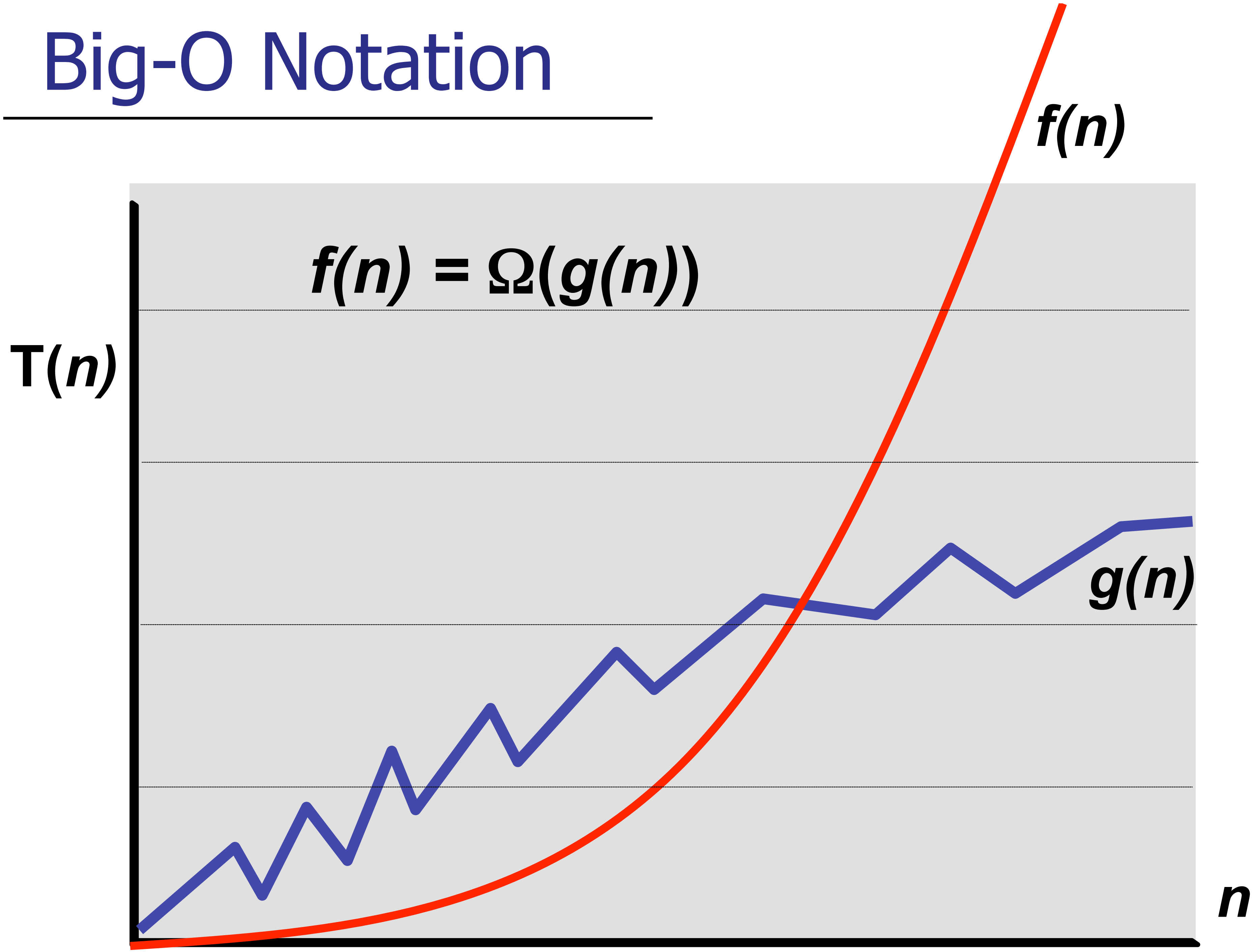
$T(n) = \Omega(f(n))$ if:

- there exists a constant $c > 0$
- there exists a constant $n_0 > 0$

such that for all $n > n_0$:

$T(n) \geq cf(n)$

Big-O Notation



Big-O Notation

Exercise:

True or false:

“$f=O(g)$ if and only if $g = \Omega(f)$”

Prove that your claim is correct using the definitions of O and Ω or by giving an example.

Example

T(n)	f(n)	big-O
$T(n) = 1000n$	$f(n) = 1$	$T(n) = \Omega(1)$
$T(n) = n$	$f(n) = n$	$T(n) = \Omega(n)$
$T(n) = n^2$	$f(n) = n$	$T(n) = \Omega(n)$
$T(n) = 13n^2 + n$	$f(n) = n^2$	$T(n) = \Omega(n^2)$

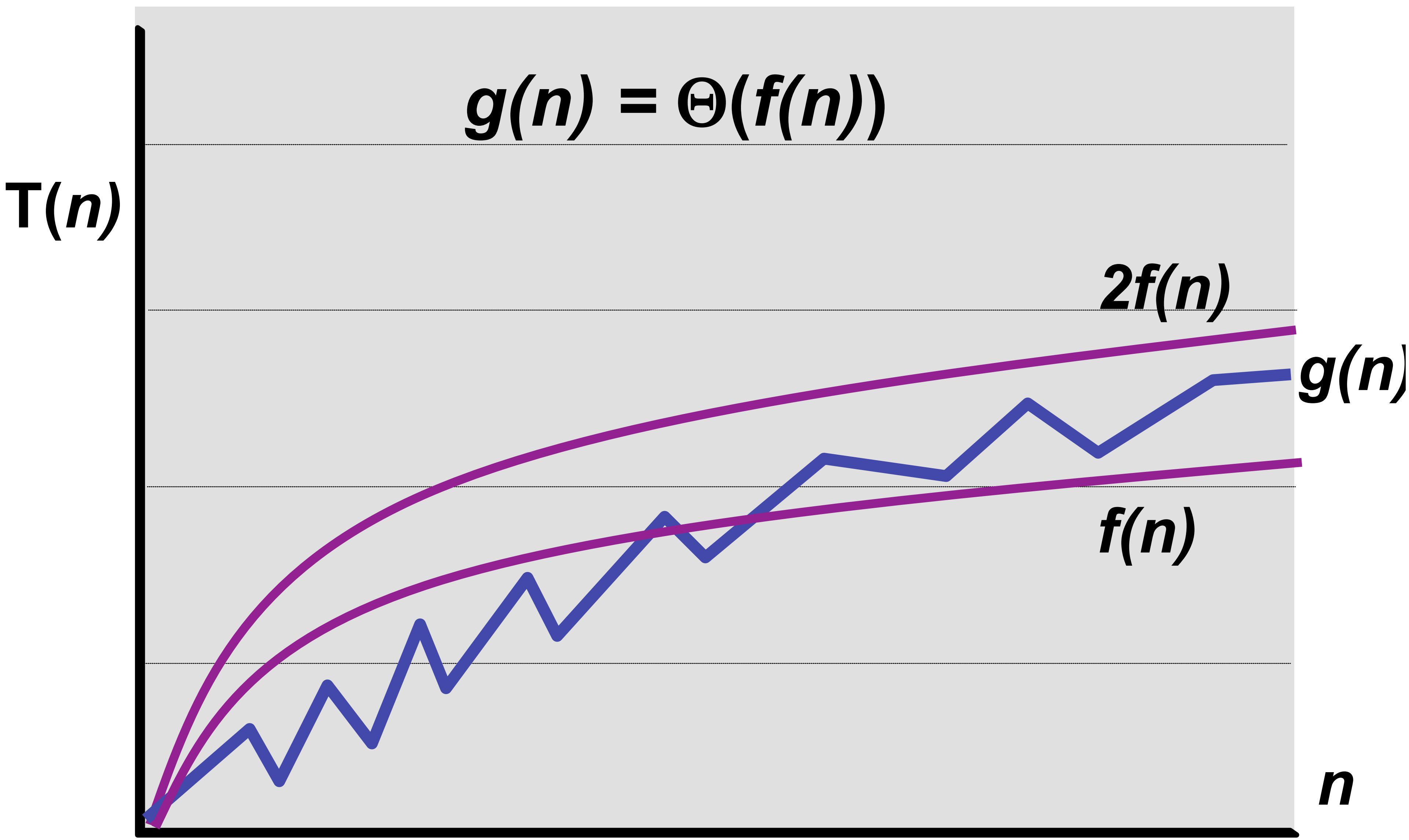
Big-O Notation

Definition: $T(n) = \Theta(f(n))$ if T grows at the same rate as f

$\boldsymbol{T(n) = \Theta(f(n))}$ if and only if:

- $T(n) = O(n)$
- $T(n) = \Omega(f(n))$

Big-O Notation



Example

$T(n)$	$f(n)$	big-O
$T(n) = 1000n$	$f(n) = n$	$T(n) = \Theta(n)$
$T(n) = n$	$f(n) = 1$	$T(n) \neq \Theta(1)$
$T(n) = 13n^2 + n$	$f(n) = n^2$	$T(n) = \Theta(n^2)$
$T(n) = n^3$	$f(n) = n^2$	$T(n) \neq \Theta(n^2)$

Big-O Notation

Rules:

If $T(n) = O(f(n))$ and $S(n) = O(g(n))$ then:

$T(n) + S(n) = O(f(n) + g(n))$

Example:

$10n^2 = O(n^2)$

$5n = O(n)$

$10n^2 + 5n = O(n^2 + n) = O(n^2)$

Big-O Notation

Rules:

If $T(n) = O(f(n))$ and $S(n) = O(g(n))$ then:

$T(n)*S(n) = O(f(n)*g(n))$

Example:

$10n^2 = O(n^2)$

$5n = O(n)$

$50n^3 = (10n^2)(5n) = O(n*n^2) = O(n^3)$

Big-O Notation

Rules:

If $T(n)$ is a polynomial of degree k then:

$$T(n) = O(n^k)$$

Example:

$$10n^5 + 50n^3 + 10n + 17 = O(n^5)$$

Big-O Notation

Rules:

If $T(n) = \log^k n$ for any k then:

$$T(n) = O(n)$$

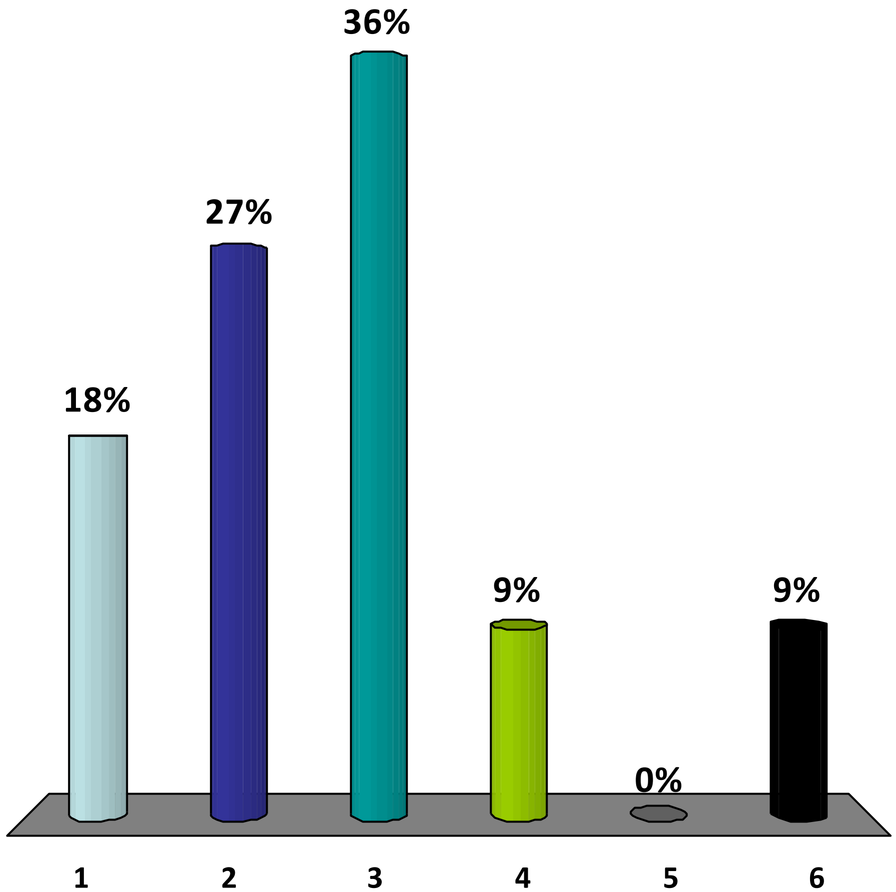
Example:

$\log^5 n = O(n)$

$4n^2\log(n) + 8n + 16 =$

1. O(log n)
2. O(n)
3. O(nlog n)
4. $\boxed{O(n^2 \log n)}$
5. $O(2^n)$
6. Still confused…

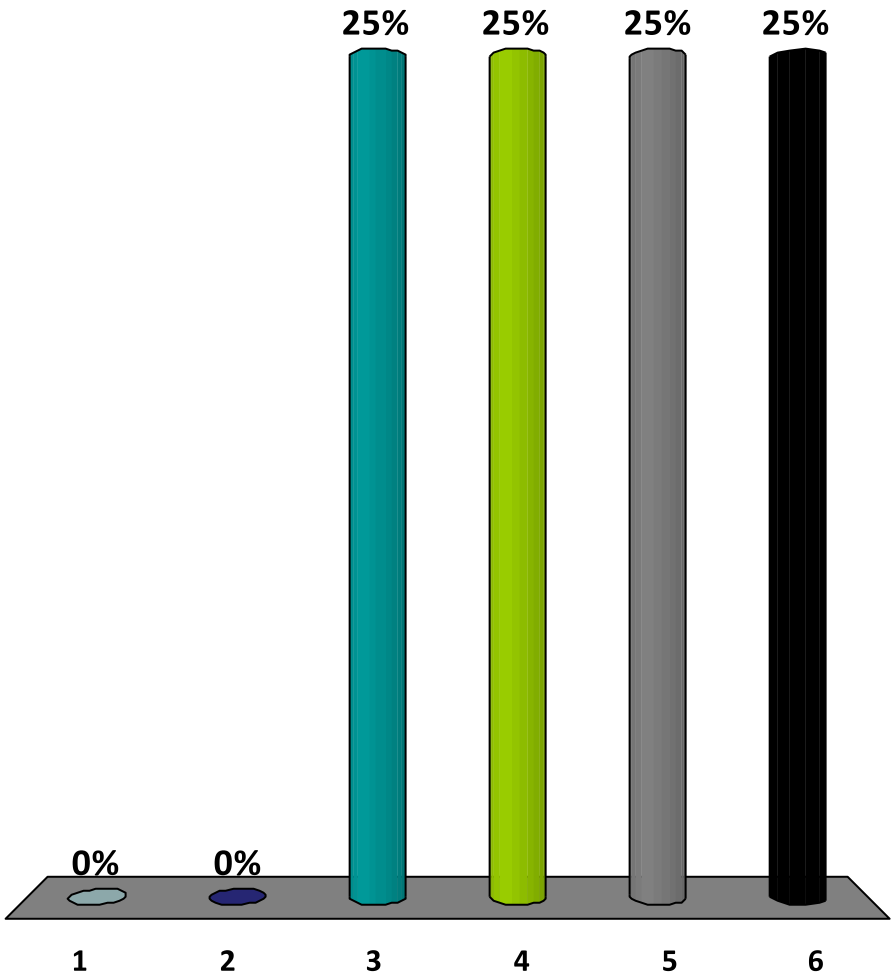
Response Counter



$$2^{2n} + 2^n + 2 =$$

1. O(n)
2. O(n^6)
3. O(2^n)
4. $\boxed{O(2^{2n})}$
5. O(n^n)
6. Still confused…

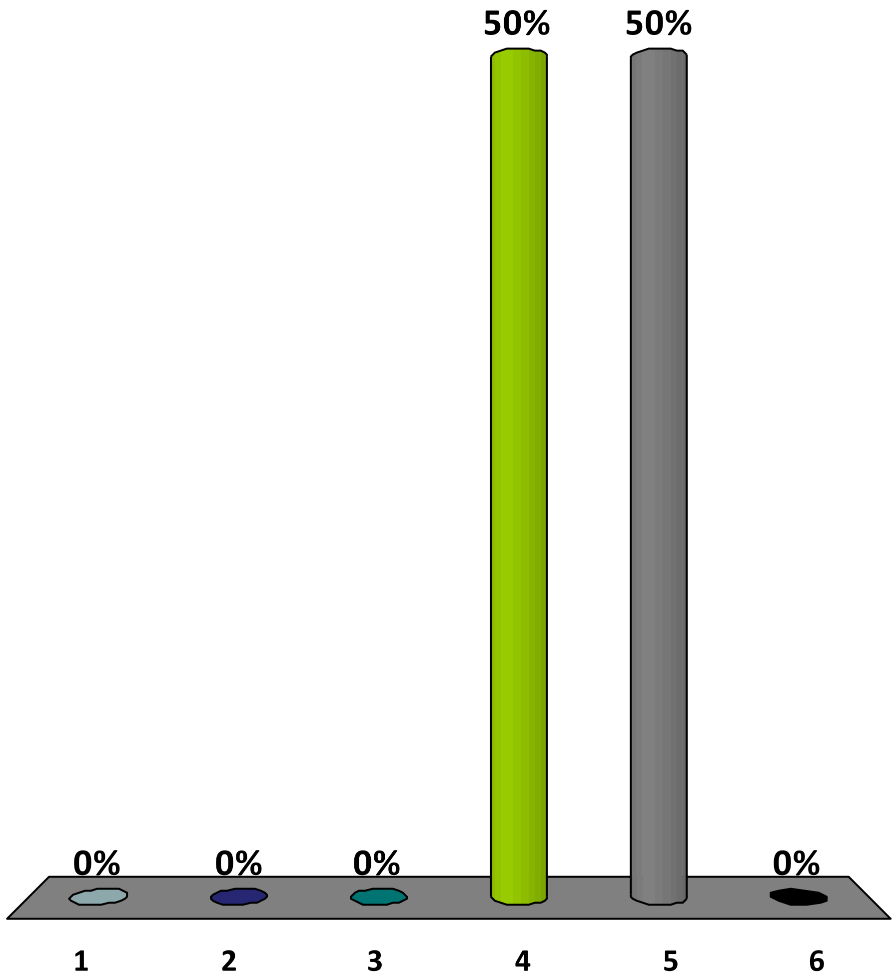
Response Counter



$\log(8n^2 + 4n) =$

1. O(1)
2. O(log n)
3. O($\log^2$n)
4. O(n)
5. O(n^2)
6. Still confused…

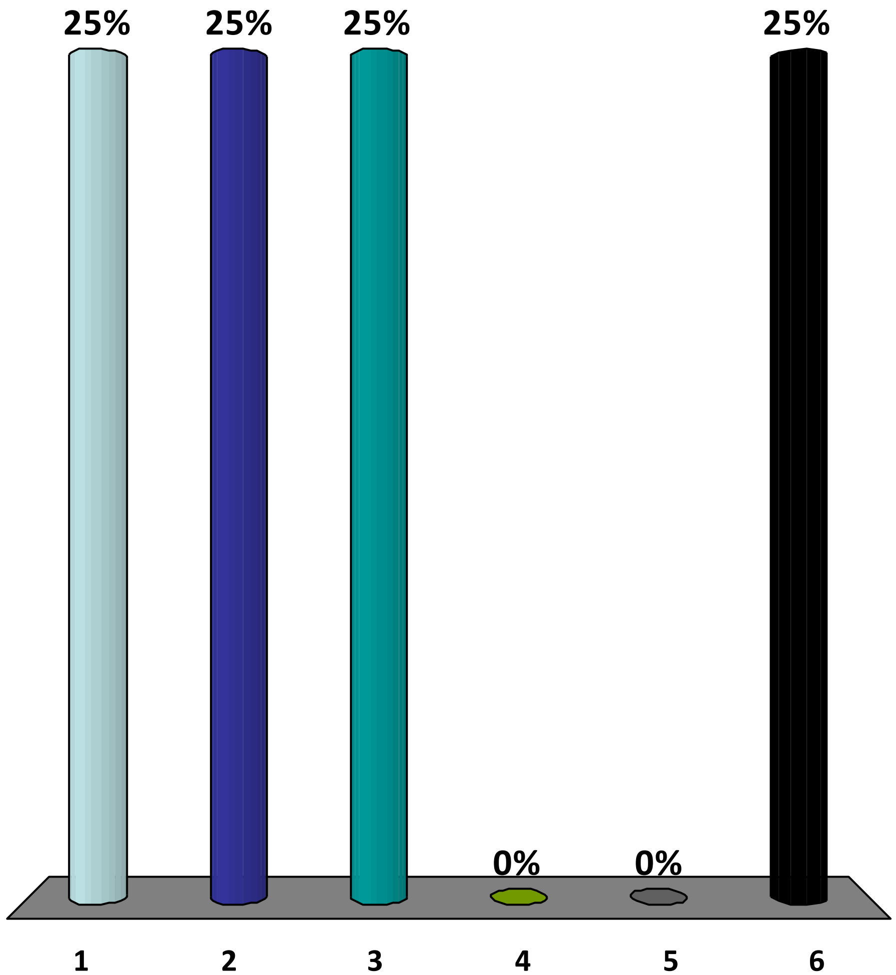
Response Counter



log(n!) =

1. O(log n)
2. O(n)
3. O(n log n)
4. O(n^2)
5. O(2^n)
6. Still confused…

Response Counter



Model of Computation

Sequential Computer

- One thing at a time

- All operations take constant time

  Addition, subtraction, multiplication, comparison

Algorithm Analysis

Example:

```
void sum(int k, int[] intArray) {
   int total=0;
   for (int i=0; i<= k; i++){
        total = total + intArray[i];
   }
   return total;
}
```

1 assignment

1 assignment
k+1 comparisons
k increments

k array access
k addition
k assignment

1 return

Total: 1 + 1 + (k+1) + 3k + 1 = 4k+4 = O(k)

Rules

Loops

- cost = (# iterations)(max cost of one iteration)

```
int sum(int k, int[] intArray) {
   int total=0;
   for (int i=0; i<= k; i++){
       total = total + intArray[i];
   }
   return total;
}
```

Rules

Nested Loops

- cost = (# iterations)(max cost of one iteration)

```
int sum(int k, int[] intArray) {
    int total=0;
    for (int i=0; i<= k; i++){
      for (int j=0; j<= k; j++){
          total = total + intArray[i];
      }
   }
   return total;
}
```

Rules

Sequential statements

- cost = (cost of first) + (cost of second)

```
int sum(int k, int[] intArray) {
   for (int i=0; i<= k; i++)
        intArray[i] = k;
   for (int j =0; j<= k; j++)
       total = total + intArray[i];
   return total;
}
```

Rules

if / else statements

- cost = max(cost of first, cost of second)

  <= (cost of first) + (cost of second)

```
void sum(int k, int[] intArray) {
   if (k > 100)
       doExpensiveOperation();
   else
      doCheapOperation();
   return;
}
```

Recurrences

$$T(n) = 1 + T(n-1) + T(n-2)$$

```
int fib(int n) {
   if (n <= 1)
      return n;
   else
      return fib(n-1) + fib(n-2);
}
```

Recurrences

$$T(n) = 1 + T(n-1) + T(n-2)$$
$$= O(2^n)$$

```
int fib(int n) {
  if (n <= 1)
      return n;
  else
      return fib(n-1) + fib(n-2);
}
```

What is the running time?

```
for (int i = 0; i<n; i++)
    for (int j = 0; j<i; j++)
        store[i] = i + j;
```

1. O(1)
2. O(n)
3. O(n log n)
4. $\boxed{O(n^2)}$
5. $O(2^n)$