Week 01: Introduction to Algorithms

FanFly

March 8, 2020

Computational Problems

What is a computational problem?

Computational Problems

What is a computational problem?

- A **computational problem** is a relation between inputs and outputs.

The Primality Test Problem

Here is an example.

The Primality Test Problem

Here is an example.

The Primality Test Problem

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Let's try to solve it for some instances by hand.

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Let's try to solve it for some instances by hand.

- Is 11 a prime number?

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Let's try to solve it for some instances by hand.

- Is 11 a prime number? Yes, it is.

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Let's try to solve it for some instances by hand.

- Is 11 a prime number? Yes, it is.
- Is 111 a prime number?

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Let's try to solve it for some instances by hand.

- Is 11 a prime number? Yes, it is.
- Is 111 a prime number? No, it isn't since $111 = 3 \times 37$.

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Let's try to solve it for some instances by hand.

- Is 11 a prime number? Yes, it is.
- Is 111 a prime number? No, it isn't since $111 = 3 \times 37$.
- Is 1111 a prime number?

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Let's try to solve it for some instances by hand.

- Is 11 a prime number? Yes, it is.
- Is 111 a prime number? No, it isn't since $111 = 3 \times 37$.
- Is 1111 a prime number? No, it isn't since $1111 = 11 \times 101$.

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Let's try to solve it for some instances by hand.

- Is 11 a prime number? Yes, it is.
- Is 111 a prime number? No, it isn't since $111 = 3 \times 37$.
- Is 1111 a prime number? No, it isn't since $1111 = 11 \times 101$.
- Is 11111 a prime number?

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Let's try to solve it for some instances by hand.

- Is 11 a prime number? Yes, it is.
- Is 111 a prime number? No, it isn't since $111 = 3 \times 37$.
- Is 1111 a prime number? No, it isn't since $1111 = 11 \times 101$.
- Is 11111 a prime number? No, it isn't since $11111 = 41 \times 271$.

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Let's try to solve it for some instances by hand.

- Is 11 a prime number? Yes, it is.
- Is 111 a prime number? No, it isn't since $111 = 3 \times 37$.
- Is 1111 a prime number? No, it isn't since $1111 = 11 \times 101$.
- Is 11111 a prime number? No, it isn't since $11111 = 41 \times 271$.
- Is 1111111111111111111 a prime number?

The Primality Test Problem

Here is an example.

The Primality Test Problem

- Input: A positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Let's try to solve it for some instances by hand.

- Is 11 a prime number? Yes, it is.
- Is 111 a prime number? No, it isn't since $111 = 3 \times 37$.
- Is 1111 a prime number? No, it isn't since $1111 = 11 \times 101$.
- Is 11111 a prime number? No, it isn't since $11111 = 41 \times 271$.
- Is 1111111111111111111 a prime number? Yes, it is.

Algorithms

It's too difficult to test if an integer is prime by hand.
Thus, people write programs to solve it.

Algorithms

It's too difficult to test if an integer is prime by hand.
Thus, people write programs to solve it.

```
def is_prime(m):
    k = 2
    while k < m:
        if m % k == 0:
            return False
        k += 1
    return True
```

Algorithms

It's too difficult to test if an integer is prime by hand.
Thus, people write programs to solve it.

```
def is_prime(m):
    k = 2
    while k < m:
        if m % k == 0:
            return False
        k += 1
    return True
```

This function (in Python) produces a correct output for each possible input of the problem.

Algorithms

It's too difficult to test if an integer is prime by hand.
Thus, people write programs to solve it.

```
def is_prime(m):
    k = 2
    while k < m:
        if m % k == 0:
            return False
        k += 1
    return True
```

This function (in Python) produces a correct output for each possible input of the problem.

- If a function produces a correct output for each possible input of a problem, we say that it **solves** the problem.
- Formally, we call such a function an **algorithm**.

Hardness of a Problem

In most cases, we would like to measure the "hardness" of a problem.

Hardness of a Problem

In most cases, we would like to measure the "hardness" of a problem.

- Assume that there is a program (i.e., an algorithm) that solves the problem.

Hardness of a Problem

In most cases, we would like to measure the "hardness" of a problem.

- Assume that there is a program (i.e., an algorithm) that solves the problem.
- We can measure how long it takes to execute the program.

Hardness of a Problem

In most cases, we would like to measure the "hardness" of a problem.

- Assume that there is a program (i.e., an algorithm) that solves the problem.
- We can measure how long it takes to execute the program.
- We can measure how much memory it uses to execute the program.

Hardness of a Problem

In most cases, we would like to measure the “hardness” of a problem.

- Assume that there is a program (i.e., an algorithm) that solves the problem.
- We can measure how long it takes to execute the program.
- We can measure how much memory it uses to execute the program.

However, the measurement may not be consistent due to some reasons.

Hardness of a Problem

In most cases, we would like to measure the “hardness” of a problem.

- Assume that there is a program (i.e., an algorithm) that solves the problem.
- We can measure how long it takes to execute the program.
- We can measure how much memory it uses to execute the program.

However, the measurement may not be consistent due to some reasons.

- The input may change.

Hardness of a Problem

In most cases, we would like to measure the “hardness” of a problem.

- Assume that there is a program (i.e., an algorithm) that solves the problem.
- We can measure how long it takes to execute the program.
- We can measure how much memory it uses to execute the program.

However, the measurement may not be consistent due to some reasons.

- The input may change.
- The programming language may change.

Hardness of a Problem

In most cases, we would like to measure the “hardness” of a problem.

- Assume that there is a program (i.e., an algorithm) that solves the problem.
- We can measure how long it takes to execute the program.
- We can measure how much memory it uses to execute the program.

However, the measurement may not be consistent due to some reasons.

- The input may change.
- The programming language may change.
- The power of the computer (on which the program runs) may change.

Computation Models

In order to make the measurement consistent in any situation, we have to specify a computation model.

Computation Models

In order to make the measurement consistent in any situation, we have to specify a computation model.

For example, we can define the following operations as **primitive** ones, each consumes one unit of time.

Computation Models

In order to make the measurement consistent in any situation, we have to specify a computation model.

For example, we can define the following operations as **primitive** ones, each consumes one unit of time.

- Assigning a value to a variable.

Computation Models

In order to make the measurement consistent in any situation, we have to specify a computation model.

For example, we can define the following operations as **primitive** ones, each consumes one unit of time.

- Assigning a value to a variable.
- Performing an arithmetic or logical operation.

Computation Models

In order to make the measurement consistent in any situation, we have to specify a computation model.

For example, we can define the following operations as **primitive** ones, each consumes one unit of time.

- Assigning a value to a variable.
- Performing an arithmetic or logical operation.
- Indexing into an array (i.e., a `list` in Python).

Computation Models

In order to make the measurement consistent in any situation, we have to specify a computation model.

For example, we can define the following operations as **primitive** ones, each consumes one unit of time.

- Assigning a value to a variable.
- Performing an arithmetic or logical operation.
- Indexing into an array (i.e., a `list` in Python).
- Performing a jump due to branch, loop or function call.

Computation Models

In order to make the measurement consistent in any situation, we have to specify a computation model.

For example, we can define the following operations as **primitive** ones, each consumes one unit of time.

- Assigning a value to a variable.
- Performing an arithmetic or logical operation.
- Indexing into an array (i.e., a `list` in Python).
- Performing a jump due to branch, loop or function call.

Then the executation time can be measured regardless of the environment.

Analysis of Time Complexity

Let us consider this program again.

```python
def is_prime(m):
    k = 2
    while k < m:
        if m % k == 0:
            return False
        k += 1
    return True
```

Analysis of Time Complexity

Let us consider this program again.

```
def is_prime(m):
    k = 2
    while k < m:
        if m % k == 0:
            return False
        k += 1
    return True
```

Given the input m, it can be shown that the statements inside the `while` loop is executed at most $m - 2$ times.

Analysis of Time Complexity

Let us consider this program again.

```
def is_prime(m):
    k = 2
    while k < m:
        if m % k == 0:
            return False
        k += 1
    return True
```

Given the input m, it can be shown that the statements inside the `while` loop is executed at most $m - 2$ times.

- Note that the `return` statement inside the `while` loop is not executed if m is prime.

Analysis of Time Complexity

Let us consider this program again.

```
def is_prime(m):
    k = 2
    while k < m:
        if m % k == 0:
            return False
        k += 1
    return True
```

Given the input m, it can be shown that the statements inside the `while` loop is executed at most $m - 2$ times.

- Note that the `return` statement inside the `while` loop is not executed if m is prime.
- In this case, the running time is “approximately” proportional to m.

Analysis of Time Complexity

Let us consider this program again.

```
def is_prime(m):
    k = 2
    while k < m:
        if m % k == 0:
            return False
        k += 1
    return True
```

Given the input m, it can be shown that the statements inside the `while` loop is executed at most $m - 2$ times.

- Note that the `return` statement inside the `while` loop is not executed if m is prime.
- In this case, the running time is "approximately" proportional to m.
- If the running time of a program is "approximately" proportional to m, we say that the **time complexity** of this program is $O(m)$.

The Big-O Notation

What does $O(\cdot)$ mean? We state the definition formally.

The Big-O Notation

What does $O(\cdot)$ mean? We state the definition formally.

Definition (Big-O Notation)

Let $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$ be functions. If there exists a constant $c > 0$ and an integer N such that

$$0 \leq f(n) \leq cg(n)$$

for each $n \geq N$, then we write $f(n) = O(g(n))$.

Examples of Big-O Notation

Following are some examples.

Examples of Big-O Notation

Following are some examples.

- Is $1365n = O(n)$ true?

Examples of Big-O Notation

Following are some examples.

- Is $1365n = O(n)$ true? Yes, it is since $1365n \leq cn$ with $c = 1365$.

Examples of Big-O Notation

Following are some examples.

- Is $1365n = O(n)$ true? Yes, it is since $1365n \leq cn$ with $c = 1365$.
- Is $n^2 = O(n)$ true?

Examples of Big-O Notation

Following are some examples.

- Is $1365n = O(n)$ true? Yes, it is since $1365n \leq cn$ with $c = 1365$.
- Is $n^2 = O(n)$ true? No, it isn't since $n^2 > cn$ if $n > \lceil c \rceil$.

Input Size

Now we have an algorithm that solves the primality test problem in $O(m)$ time.

The Primality Test Problem

- Input: An n-bit positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Input Size

Now we have an algorithm that solves the primality test problem in $O(m)$ time.

The Primality Test Problem

- Input: An n-bit positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Can we use n (instead of m) to represent the time complexity of the algorithm?

Input Size

Now we have an algorithm that solves the primality test problem in $O(m)$ time.

The Primality Test Problem

- Input: An n-bit positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Can we use n (instead of m) to represent the time complexity of the algorithm?

- An n-bit binary number can represent up to $2^n - 1$.

Input Size

Now we have an algorithm that solves the primality test problem in $O(m)$ time.

The Primality Test Problem

- Input: An n-bit positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Can we use n (instead of m) to represent the time complexity of the algorithm?

- An n-bit binary number can represent up to $2^n - 1$.
- One should use at least $n = \lceil \log_2(m + 1) \rceil$ bits to represent a positive integer m.

Input Size

Now we have an algorithm that solves the primality test problem in $O(m)$ time.

The Primality Test Problem

- Input: An n-bit positive integer $m \geq 2$.
- Output: True if m is prime, false if m is composite.

Can we use n (instead of m) to represent the time complexity of the algorithm?

- An n-bit binary number can represent up to $2^n - 1$.
- One should use at least $n = \lceil \log_2(m+1) \rceil$ bits to represent a positive integer m.
- Thus the time complexity of the algorithm is $O(2^n)$.

Comparison between Algorithms

Suppose that we have two algorithms that solve the same problem.

Comparison between Algorithms

Suppose that we have two algorithms that solve the same problem.

- Algorithm A runs in $O(n^2)$ time.

Comparison between Algorithms

Suppose that we have two algorithms that solve the same problem.

- Algorithm A runs in $O(n^2)$ time.
- Algorithm B runs in $O(n)$ time.

Comparison between Algorithms

Suppose that we have two algorithms that solve the same problem.

- Algorithm A runs in $O(n^2)$ time.
- Algorithm B runs in $O(n)$ time.

Can we thus say that Algorithm B is better than Algorithm A?

Comparison between Algorithms

Suppose that we have two algorithms that solve the same problem.

- Algorithm A runs in $O(n^2)$ time.
- Algorithm B runs in $O(n)$ time.

Can we thus say that Algorithm B is better than Algorithm A?

No! Note that we also have the following results.

Comparison between Algorithms

Suppose that we have two algorithms that solve the same problem.

- Algorithm A runs in $O(n^2)$ time.
- Algorithm B runs in $O(n)$ time.

Can we thus say that Algorithm B is better than Algorithm A?

No! Note that we also have the following results.

- Algorithm A runs in $O(n^3)$ time.

Comparison between Algorithms

Suppose that we have two algorithms that solve the same problem.

- Algorithm A runs in $O(n^2)$ time.
- Algorithm B runs in $O(n)$ time.

Can we thus say that Algorithm B is better than Algorithm A?

No! Note that we also have the following results.

- Algorithm A runs in $O(n^3)$ time.
- Algorithm B runs in $O(n^4)$ time.

Comparison between Algorithms

Suppose that we have two algorithms that solve the same problem.

- Algorithm A runs in $O(n^2)$ time.
- Algorithm B runs in $O(n)$ time.

Can we thus say that Algorithm B is better than Algorithm A?

No! Note that we also have the following results.

- Algorithm A runs in $O(n^3)$ time.
- Algorithm B runs in $O(n^4)$ time.

Thus, using big-O notation is not enough to compare the efficiency of different algorithms.

The Big-Omega and Big-Theta Notations

Let us introduce new notations, $\Omega(\cdot)$ and $\Theta(\cdot)$.

The Big-Omega and Big-Theta Notations

Let us introduce new notations, $\Omega(\cdot)$ and $\Theta(\cdot)$.

Definition (Big-Omega and Big-Theta Notations)

Let $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$ be functions.

The Big-Omega and Big-Theta Notations

Let us introduce new notations, $\Omega(\cdot)$ and $\Theta(\cdot)$.

Definition (Big-Omega and Big-Theta Notations)

Let $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$ be functions.

- We write $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.

The Big-Omega and Big-Theta Notations

Let us introduce new notations, $\Omega(\cdot)$ and $\Theta(\cdot)$.

Definition (Big-Omega and Big-Theta Notations)

Let $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$ be functions.

- We write $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.
- We write $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

The Big-Omega and Big-Theta Notations

Let us introduce new notations, $\Omega(\cdot)$ and $\Theta(\cdot)$.

Definition (Big-Omega and Big-Theta Notations)

Let $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$ be functions.

- We write $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.
- We write $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Suppose again that we have two algorithms that solve the same problem.

The Big-Omega and Big-Theta Notations

Let us introduce new notations, $\Omega(\cdot)$ and $\Theta(\cdot)$.

Definition (Big-Omega and Big-Theta Notations)

Let $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$ be functions.

- We write $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.
- We write $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Suppose again that we have two algorithms that solve the same problem.

- Algorithm A runs in $\Omega(n^2)$ time.

The Big-Omega and Big-Theta Notations

Let us introduce new notations, $\Omega(\cdot)$ and $\Theta(\cdot)$.

Definition (Big-Omega and Big-Theta Notations)

Let $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$ be functions.

- We write $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.
- We write $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Suppose again that we have two algorithms that solve the same problem.

- Algorithm A runs in $\Omega(n^2)$ time.
- Algorithm B runs in $O(n)$ time.

The Big-Omega and Big-Theta Notations

Let us introduce new notations, $\Omega(\cdot)$ and $\Theta(\cdot)$.

Definition (Big-Omega and Big-Theta Notations)

Let $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$ be functions.

- We write $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.
- We write $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Suppose again that we have two algorithms that solve the same problem.

- Algorithm A runs in $\Omega(n^2)$ time.
- Algorithm B runs in $O(n)$ time.

Now we can say that Algorithm B is better than Algorithm A (in time complexity).

Exercise #1

We know that there is an $O(2^n)$-time algorithm that solves the primality test problem for n-bit inputs as follows.

```
def is_prime(m):
    k = 2
    while k < m:
        if m % k == 0:
            return False
        k += 1
    return True
```

Exercise #1

We know that there is an $O(2^n)$-time algorithm that solves the primality test problem for n-bit inputs as follows.

```
def is_prime(m):
    k = 2
    while k < m:
        if m % k == 0:
            return False
        k += 1
    return True
```

In fact, it runs in time $\Theta(2^n)$ if we consider the worst-case complexity.

Exercise #1 (cont.)

Now we have an improvement on this algorithm.

```python
def is_prime(m):
    k = 2
    # Only test k <= sqrt(m)
    while k * k <= m:
        if m % k == 0:
            return False
        k += 1
    return True
```

Exercise #1 (cont.)

Now we have an improvement on this algorithm.

```python
def is_prime(m):
    k = 2
    # Only test k <= sqrt(m)
    while k * k <= m:
        if m % k == 0:
            return False
        k += 1
    return True
```

What is the worst-case time complexity of this algorithm?

- $\Theta(n)$
- $\Theta(2^{\sqrt{n}})$
- $\Theta(2^{n/2})$
- $\Theta(2^n)$

Exercise #1 (cont.)

Now we have an improvement on this algorithm.

```
def is_prime(m):
    k = 2
    # Only test k <= sqrt(m)
    while k * k <= m:
        if m % k == 0:
            return False
        k += 1
    return True
```

What is the worst-case time complexity of this algorithm?

- $\Theta(n)$
- $\Theta(2^{\sqrt{n}})$
- $\Theta(2^{n/2})$
- $\Theta(2^n)$

Solution

The worst-case time complexity is $\Theta(\sqrt{2^n}) = \Theta(2^{n/2})$.

Exercise #2

Let $f : \mathbb{N} \to \mathbb{R}$ be a function with

$$f(n) = \sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}$$

for any positive integer n.

Exercise #2

Let $f : \mathbb{N} \to \mathbb{R}$ be a function with

$$f(n) = \sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}$$

for any positive integer n.

Find an elementary function $g : \mathbb{N} \to \mathbb{R}$ such that $f(n) = \Theta(g(n))$.

Exercise #2

Let $f : \mathbb{N} \to \mathbb{R}$ be a function with

$$f(n) = \sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}$$

for any positive integer n.

Find an elementary function $g : \mathbb{N} \to \mathbb{R}$ such that $f(n) = \Theta(g(n))$.

Solution

We have $f(n) = \Theta(\ln n)$ since

$$\int_1^{n+1} \frac{1}{t}\, dt \quad \leq \quad f(n) \quad \leq \quad 1 + \int_1^{n} \frac{1}{t}\, dt.$$

Exercise #3

Consider the following function.

```python
def hello(n):
    if n == 0:
        return
    elif n == 1:
        print("Hello!")
    else:
        hello(n - 2)
        hello(n - 1)
```

Exercise #3

Consider the following function.

```python
def hello(n):
    if n == 0:
        return
    elif n == 1:
        print("Hello!")
    else:
        hello(n - 2)
        hello(n - 1)
```

Let $H(n)$ be the number of lines of `"Hello!"` printed by this function, given the input n.

Exercise #3

Consider the following function.

```
def hello(n):
    if n == 0:
        return
    elif n == 1:
        print("Hello!")
    else:
        hello(n - 2)
        hello(n - 1)
```

Let $H(n)$ be the number of lines of `"Hello!"` printed by this function, given the input n.

Then we have

$$H(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ H(n-2) + H(n-1), & \text{otherwise.} \end{cases}$$

Exercise #3 (cont.)

Which of the following is true?

- $H(n) = O(n)$.
- $H(n) = \Omega(n)$ and $H(n) = O(n^2)$.
- $H(n) = \Omega(n^2)$ and $H(n) = O(2^n)$.
- $H(n) = \Omega(2^n)$.

Exercise #3 (cont.)

Which of the following is true?

- $H(n) = O(n)$.
- $H(n) = \Omega(n)$ and $H(n) = O(n^2)$.
- $H(n) = \Omega(n^2)$ and $H(n) = O(2^n)$.
- $H(n) = \Omega(2^n)$.

Solution

Note that $H(n)$ is the nth Fibonacci number.

Exercise #3 (cont.)

Which of the following is true?

- $H(n) = O(n)$.
- $H(n) = \Omega(n)$ and $H(n) = O(n^2)$.
- $H(n) = \Omega(n^2)$ and $H(n) = O(2^n)$.
- $H(n) = \Omega(2^n)$.

Solution

Note that $H(n)$ is the nth Fibonacci number. Thus we have

$$H(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right),$$

implying $H(n) = \Omega(n^2)$ and $H(n) = O(2^n)$.