

Summary – Halving the Search

The best strategy for Higher or Lower, the most guesses it could ever need, and why it only works when the numbers are in order.

The game

One player, the **thinker**, picks a secret whole number from 1 to 100. The other player, the **guesser**, says a number. The thinker answers **higher**, **lower** or **right**. The guesser tries to find the number in as few guesses as possible.

The strategy

1. Guess the middle of the numbers that are still possible.
2. The answer tells you which half the secret number is in. Throw away the other half.
3. Repeat until you find it.

The best first guess is **50** (or 51), the middle of 1 to 100. Whatever the answer, about half of the numbers are gone. A first guess of 10 might rule out only ten numbers.

Example 1

The secret number is 37, from 1 to 100.

Guess	Answer	Still possible	Left
		1 to 100	100
50	lower	1 to 49	49
25	higher	26 to 49	24
37	right		

Found in **3** guesses.

Example 2

What is the most guesses you could ever need for a number from 1 to 100?

Each guess leaves at most half of the numbers. Starting from 100, the most numbers that can be left are:

100, 50, 25, 12, 6, 3, 1

After six guesses, at most one number is left, so the seventh guess always finds it. The most is **7** guesses.

The most guesses you could ever need

The same reasoning as in Example 2 gives every column of this table.

Numbers from	1 to 10	1 to 20	1 to 100	1 to 200	1 to 1000	1 to 1,000,000
Most guesses	4	5	7	8	10	20

- **Doubling** the numbers costs exactly **one** more guess: 1 to 10 needs 4, and 1 to 20 needs 5.
- **Ten times** as many numbers costs about **three** more guesses: 4, then 7, then 10. A million numbers need only 20.

Why it works

Halving only works because the thinker says **higher** or **lower**, and that only makes sense because the numbers are **in order**.

If the thinker only ever said **right** or **wrong**, each wrong guess would rule out just one number. You would be back to needing up to 100 guesses for 1 to 100.

Two searches, two names

Playing with **higher** and **lower** is a **binary search**: every guess halves what is left. Playing with only **right** and **wrong** is a **linear search**: you check the numbers one at a time, because nothing you hear lets you skip any.

Computer scientists compare algorithms by asking how the number of steps grows as the list gets longer. This is called **time complexity**. Call the number of possible values n . Two symbols describe the two extremes:

- **Big Omega**, written Ω , is the **best case**: the fewest steps the algorithm could ever take.
- **Big O**, written O , is the **worst case**: the most steps it could ever take.

	Linear search (right or wrong)	Binary search (higher or lower)
Best case	$\Omega(1)$: the first guess is right	$\Omega(1)$: the first guess is right
Worst case	$O(n)$: up to every number, one by one	$O(\log n)$: about $\log_2 n$ guesses
Worst case, 100 numbers	100 guesses	7 guesses
Worst case, 1,000,000 numbers	1,000,000 guesses	20 guesses

Both searches share the same best case. Getting lucky on the first guess is always possible, so $\Omega(1)$ says nothing about which search is better. The worst case is what separates them: $O(n)$ grows as fast as the list, while $O(\log n)$ grows by only one step each time the list doubles. That is why, when someone asks how fast an algorithm is, they usually mean its Big O.

Big O and Big Omega go beyond what the AP Computer Science Principles exam asks, but they are the everyday vocabulary computer scientists use to talk about how fast an algorithm is, so it is worth knowing them. The exam wants the idea behind them: binary search needs sorted data, and it needs far fewer steps than checking one at a time. Binary search comes back properly in Thread 3.