Program testing can be used very effectively to show the presence of bugs but never to show their absence.

E.W. Dijkstra, EWD 303

All by itself, a program is no more than half a conjecture. The other half of the conjecture is the functional specification the program is supposed to satisfy. The programmer's task is to present such complete conjectures as proven theorems.

E.W. Dijkstra, EWD 1036

Bounded linear search returns

$$\begin{array}{ll} -1 & \text{if } \forall\ i \in [0, n), \sim P(\texttt{sequence}[i]) \\ k & \text{otherwise, where } P(\texttt{sequence}[k]) \\ & \qquad\qquad\quad \text{and } \forall\ i \in [0, k), \sim P(\texttt{sequence}[i]) \end{array}$$

Invariant (Loop of `bounded_linear_search`.)

(a) $\forall\ j \in [0, i-1), \sim P(\texttt{sequence}[j])$

(b) `found` *iff* $P(\texttt{sequence}[i-1])$

(c) *i is the number of iterations completed.*

Initialization.

(a) Since i is initially 0, the range $[0, i) = [0, 0)$ which is empty. Hence the proposition is vacuously true.

(b) Assuming $\sim P(\mathit{undef})$ makes this hold.

(c) There have been 0 iterations, and $i = 0$.

Maintenance. Distinguish i_{pre} and i_{post}, namely $i_{\text{post}} = i_{\text{pre}} + 1$. Similarly distinguish $\texttt{found}_{\text{pre}}$ and $\texttt{found}_{\text{post}}$

(a) It must be that $\sim \texttt{found}_{\text{pre}}$ or else the guard would have failed. Thus $\sim P(\texttt{sequence}[i_{\text{pre}} - 1])$, by *inductive hypothesis*, part b. Together with the fact that that $\forall\ j \in [0, i_{\text{pre}} - 1), \sim P(\texttt{sequence}[j])$, we now have $\forall\ j \in [0, i_{\text{pre}}), \sim P(\texttt{sequence}[j])$, that is $\forall\ j \in [0, i_{\text{post}} - 1), \sim P(\texttt{sequence}[j])$.

(b) Immediate from the assignment to `found`.

(c) Immediate from the update to i. $\square$

Correctness claim (`bounded_linear_search.`)

After at most n iterations, `bounded_linear_search` will return as specified.

Proof. By Invariant 1.c, after at most n iterations, $i = n$ and the guard will fail. Moreover, when the guard fails, either `found` or $i = n$.
Case 1. Suppose `found`. Then we return $i - 1$. Invariant 1.a tells us that nothing in $[0, i - 1)$ satisfies P. Invariant 1.b tells us that $i - 1$ does. Together these fulfill the second part of the specification: $i - 1$ is the first item satisfying P, and we return it.
Case 2. Suppose $\sim$ `found`. By elimination $i = n$. Invariant 1.a tells us that nothing in $[0, n - 1)$ satisfies P. Invariant 1.b tells us that $i - 1$, that is, $n - 1$, also does not satisfy P. We return -1, fulfilling the first part of the specification. $\square$

Binary search returns

$$\begin{array}{ll} -1 & \text{if } \forall\ i \in [0, n), \texttt{sequence}[i] \neq \texttt{item} \\ k & \text{otherwise, where } \texttt{sequence}[k] = \texttt{item} \end{array}$$

Invariant (Loop of `binary_search`.)

(a) *If* $\exists\ j \in [0, n)$ *such that* $\texttt{item} = \texttt{sequence}[j]$, *then* $\exists\ j \in [\texttt{low}, \texttt{high})$ *such that* $\texttt{item} = \texttt{sequence}[j]$.

(b) *After* i *iterations,* $\texttt{high} - \texttt{low} \leq \frac{n}{2^i}$.

Invariant (Class `Exercise2`.)

(a) *`head == null` iff `tail == null` iff `size == 0`.*
(b) *If `tail != null` then `tail.next == null`.*
(c) *If `head != null` then `tail` is reached by following `size - 1` `next` links from `head`.*

```
def binary_searfch(sequence, TO, item):                              c1
    low = 0
    high = len(sequence)                                             c2
     while high - low > 1 :                                          c3
        mid = (low + high) / 2
        compar = TO(item, sequence[mid])
        if compare < 0:
            high = mid                                               c4
        elif compar > 0:                                             c5
            low = mid + 1                                            c6
        else:                                                        c7
            low = mid
            high = mid + 1                                           c8
     if low < high and TO(item, sequence[low]) == 0:
        return low                                                   c9
    else:
        return -1                                                    c10
```

$$
\begin{aligned}
T_{bs}(n) &= c_1 + c_2(\lg n + 1) + (c_3 + \max(c_4, c_5 + c_6, c_5 + c_7)) \lg n \\
&\quad + c_8 + \max(c_9, c_{10}) \\
&= d_0 + d_1 \lg n
\end{aligned}
$$

Every time you run a program, you are performing a scientific experiment that relates the program to the natural world and answers one of our core questions: How long will my program take?
The running time [of many programs] is relatively insensitive to the input itself; it depends primarily on the problem size.

Sedgewick, pg 173

$g(n) \sim f(n)$ means the functions are asymptotically *equal*, that is, that $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 1$. Thus $\frac{n^3}{6} - \frac{n^2}{2} + \frac{n}{3} \sim \frac{n^3}{6}$.

$g(n) = O(f(n))$, which really should be written $g(n) \in O(f(n))$, means that a scaled version of $f(n)$ asymptotically *bounds* g above. It means there exists a c such that when n is large enough, $g(n) \leq cf(n)$. Thus $\frac{n^3}{6} - \frac{n^2}{2} + \frac{n}{3} = O(\frac{n^3}{6})$ but also $\frac{n^3}{6} - \frac{n^2}{2} + \frac{n}{3} = O(n^3)$ and $\frac{n^3}{6} - \frac{n^2}{2} + \frac{n}{3} = O(n^4)$.

With big-oh, you can throw away the lower ordered terms *and* throw away the constant factor of the highest order term *and* overshoot.

With tilde, you only can throw away the lower ordered terms.