Searching

An important use of computers is to look up data—how often have you turned on your computer just to google for something?

The process of looking up data is called *searching*. Common examples would be to look up a person in the telephone book or to search for the meaning of a word in a dictionary. The efficiency of searching depends on whether the data being searched is sorted or not.

Linear search

When the data is not sorted or indexed in some way, we have no choice but to look at each item one by one. This is called *linear search* or *sequential search*, because we are stepping through a list of items sequentially in linear order, comparing each of them with the item we are looking for.

This is you would have to do in the case of the telephone book when we search a person by his/her phone number: While this is an impossible task for a human, it is just slow (and boring) for a computer.

Linear search takes $O(n)$ time on a list of length n. However, it is often possible to improve this in practice, by keeping often-needed elements at the beginning of the list: the best case search time is only $O(1)$.

Binary search

More interesting is the case where the data is sorted. To make effective use of this, we need to be able to access each element quickly, so we will assume that the data is in a Python list (not in a linked list), and formulate the problem as follows:

> Given a Python list a with a *non-decreasing* sequence of elements and an element x, find the smallest index i such that $a[i] \geqslant x$. If all elements of a are smaller than x, return `len(a)`.

The definition is perhaps a bit more complicated than expected, since we also want to say precisely what should happen when the element x we are searching is *not in the array*. This is often important, for instance because we then want to insert it at this position, or because we want to find the nearest element in the array.

Once again, the most elegant and clearest solution is recursive. We compare x with the middle element of the list a, and recursively search in the left half or the right half. This algorithm is called *binary search*, since we are making a binary decision in every step. Here is code for recursive binary search.

```
def find_rec(a, x, i, j):
  if j < i:
    return i
  mid = (i + j) // 2
  if a[mid] < x:
    return find_rec(a, x, mid+1, j)
  else:
    return find_rec(a, x, i, mid-1)
```

It should be called originally as

```
def find(a, x):
  return find_rec(a, x, 0, len(a) - 1)
```

Even though the idea of binary search is simple, getting the code right is not so easy. Jon Bentley, in his classic book *Programming Pearls*, says that even given ample time, only about ten percent of professional programmers were able to get this small program right. And while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962!

So let us convince ourselves that our function `find` really is correct. How do we convince ourselves (or someone else)? The only fully convincing argument is a mathematical proof of correctness.

Proof of correctness. As usual, when we want to prove correctness or running time of a recursive function, we use induction. This time, however, it is not so clear what the inductive claim should be: The simple claim that `find_rec` finds the correct index for `x` in `a` does not allow us to perform the inductive step.

What we need here is a stronger inductive claim. In correctness proofs, we set this up using *preconditions* and *postconditions*. A precondition is a condition or predicate that must always be true just prior to the execution of some section of code, a postcondition is a condition that must be true just after the code has executed.

In our `find_rec` function, we set up the precondition that $a[k] < x$ for $k < i$ and $a[k] \geqslant x$ for $k > j$. This condition must hold on entry to the function `find_rec`. The postcondition is that `find_rec` returns an integer r in the range $\{i, \ldots, j+1\}$, and that this is the correct answer. Namely, $a[k] < x$ for $k < r$, and $a[k] \geqslant x$ for all $k \geqslant r$.

We now prove the following *claim*: If the precondition holds when `find_rec` is called, then the postcondition holds when it returns.

The proof is by induction on the difference $j - i$.

The base case occurs when $j < i$. By the precondition, this is only possible if $j + 1 = i$, since otherwise the preconditions $a[k] < x$ for $k < i$ and $a[k] \geqslant x$ for $k > j$ contradict. As a result, the result $r = i = j + 1$ is the smallest index such that $a[i] \geqslant x$, and the postcondition holds.

Let us assume now that the claim holds when $j - i < d$, and consider an invokation of `find` with $j - i = d$. Let $m = \lfloor \frac{i+j}{2} \rfloor$ be the value of `mid`. We have $i \leqslant m \leqslant j$, so $a[m]$ is well-defined. We consider two cases:

If $a[m] < x$, then the sortedness of the list implies that $a[k] < x$ for all $k < m+1$. By our precondition, we also know that $a[k] \geqslant x$ for $k > j$. Together this means that the precondition of `find_rec(a, x, mid+1, j)` holds. Also $j - (m+1) < j - i = d$, and so by the inductive assumption `find_rec(a, x, mid+1, j)` returns a number $r \in \{m + 1, j + 1\}$ such that $a[k] < x$ for $k < r$ and $a[k] \geqslant x$ for $k \geqslant r$. This means that the postcondition of `find_rec(a, x, i, j)` holds.

If $a[m] \geqslant x$, then the sortedness of the list implies that $a[k] \geqslant x$ for all $k \geqslant m$. By our precondition, we also know that $a[k] < x$ for all $k < i$. Together these imply the preconditions of `find_rec(a, x, i, mid-1)`. Since $m - 1 - i < j - i = d$, the inductive assumption tells us that `find_rec(a, x, i, mid-1)` returns a number $r \in \{i, m\}$ such that $a[k] < x$ for $k < r$ and $a[k] \geqslant x$ for $k \geqslant r$. This means that the postcondition of `find_rec(a, x, i, j)` holds.

This completes the proof of the claim. Now it remains to observe that when we first call `find(a, x)`, this calls `find_rec(a, x, 0, len(a)-1)`. Therefore, the preconditions to `find_rec(a, x, 0, len(a)-1)` hold: $a[k] < x$ for all $k < 0$ (because there is no such k), and $a[k] \geqslant x$ for all $k \geqslant n$ (where n is the length of `a`, because there again is no such k). Therefore `find(a, x)` correctly returns the answer we are looking for, and it lies in the range $\{0, 1, \ldots, n\}$.

Runtime analysis. Let $T(n)$ denote the number of primitive operations for `find_rec(a, x, i, j)` when $j - i + 1 = n$, that is, when looking at an interval in the list of length n.

When n is zero, a constant number c_1 of primitive operations suffice. Otherwise, the program executes a constant number c_2 of primitive operations, and then makes a recursive call to a problem of size at most $\lceil \frac{n-1}{2} \rceil$. We get the following *recurrence relation*

$$T(n) \leqslant \begin{cases} c_1 & \text{if } n = 0, \\ c_2 + T(\lceil \frac{n-1}{2} \rceil) & \text{otherwise.} \end{cases}$$

We will show that $T(n) \leqslant c_2 k + c_1$ if $2^{k-1} \leqslant n < 2^k$. We use induction on k. If $k = 1$, we have $n = 1$, and so the recursion gives us $T(1) = c_2 + T(0) = c_2 + c_1$. Let us therefore consider $k > 1$, and assume the claim is true for all smaller values of k. For any n with $2^{k-1} \leqslant n < 2^k$, we have $\lceil \frac{n-1}{2} \rceil < 2^{k-1}$, and so the inductive assumption gives us

$$T(n) \leqslant c_2 + T\big(\lceil \frac{n-1}{2} \rceil\big) \leqslant c_2 + c_2(k-1) + c_1 = c_2 k + c_1.$$

We can rewrite the expression in terms of n as $T(n) \leqslant c_1 + c_2 \lceil \log_2(n+1) \rceil$, or $T(n) = O(\log n)$.

Note that in algorithm analysis, the symbol log always means logarithms in base 2, because that is what we need nearly all of the time. However, in big-Oh notation, it really makes no difference: For any $a > 1$, we have $\log_a x = \frac{\log_2 x}{\log_2 a}$, and therefore $\log_a x = O(\log_2 x)$ and $\log_2 x = O(\log_a x)$.

An iterative version. We have seen earlier that it can be useful to convert a recursive method into an iterative method, to avoid stack overflows and to make it a bit faster.

It turns out that it is very easy to rewrite binary search as an iterative function:

```
def binary_search_iterative(a, x):
  i = 0
  j = len(a) - 1
  while i <= j:
    # a[k] < x for k < i and a[k] >= x for k > j
    mid = (i + j) // 2
    if a[mid] < x:
      i = mid + 1
    else:
      j = mid - 1
  return i
```

We can argue that this function is correct because it does exactly the same as the recursive version. Alternatively, we can perform the correctness proof on the iterative version. Since it contains a loop, we need to write a condition that we guarantee to hold at any time during the execution of the loop—a *loop invariant*.

Our loop invariant is that at the beginning of the loop (before we evaluate the `while` condition), we have $a[k] < x$ for all $k < i$, and that $a[k] \geqslant x$ for all $k > j$. We prove that the loop invariant holds in every iteration of the loop by induction over the number of times d that the loop has already been executed.

The base case is when $d = 0$, which means that we evaluate the `while` condition for the first time. Then $i = 0$ and $j = n - 1$, and so the loop invariant holds automatically.

Assume now that the loop invariant holds in one iteration. We will argue that it will then also hold in the next iteration. We pick m with $i \leqslant m \leqslant j$, so $A(m)$ is well defined.

If $a[m] < x$, then the sortedness of the array guarantees that $a[k] < x$ for all $k \leqslant m$. The loop invariant guarantees that $a[k] \geqslant x$ for all $k > j$. In this case, we set $i = m + 1$ and leave j unchanged. This implies that the loop invariant holds in the next iteration.

If $a[m] \geqslant x$, then the sortedness of the array guarantees that $a[k] \geqslant x$ for all $k \geqslant m$. The loop invariant guarantees that $a[k] < x$ for all $k < i$. In this case, we set $j = m - 1$ and leave i unchanged. This again implies that the loop invariant holds in the next iteration.

It follows that the loop invariant holds when we evaluate the `while` condition for the last time. This means that $i > j$. By the loop invariant, this can only be true if $i = j + 1$, and we have that $a[k] < x$ for all $k < i$, and $a[k] \geqslant x$ for $k \geqslant i$. It follows that i is the first index where $a[i] \geqslant x$, and the result of the function is correct.

Was is necessary to convert this function to avoid stack overflow? As we saw above, the depth of recursion of the recursive `find` function was $\lceil \log_2(n+1) \rceil$, so even for astronomical values of n the depth remains very small—no stack overflow can happen.

Tail recursion. Converting the recursive function was so easy because it used *tail recursion*. Recursion is called tail recursion when the recursive call is the last thing that happens in the function. In some programming languages the compiler implements tail-recursive functions by jumping back to the beginning of the function, without creating a new stack frame. In such a language, the compiled code for our recursive and iterative functions would be nearly the same, and there would be no difference in running time. Unfortunately, Python is not one of those programming languages.