17.2 Average-Case Running Time of Linear Search

In this section, we'll learn how to perform an *average-case running time analysis*. Recall the *linear search* algorithm, which searches for an item in a list by checking each list element one at a time.

```
def search(lst: list, x: Any) -> bool:
    """Return whether x is in lst."""
    for item in lst:
        if item == x:
            return True
    return False
```

We've previously seen that the worst-case running time of this function is $\Theta(n)$, where n is the length of the list. What can we say about its average-case running time?

Well, for one thing, we need to precisely define what we mean by "all possible inputs of length n" for this function. Because we don't have any restrictions on the elements stored in the input list, it seems like there could be an infinite number of lists of length n to choose from, and we cannot take an average of an infinite set of numbers!

In an average-case running-time analysis, dealing with an infinite set of inputs is a common problem. To resolve it, we choose a particular set of allowable inputs, and then compute the average running time for that set. The rest of this section will be divided into two example analyses of `search`, using two different input sets.

A first example

Let $n \in \mathbb{N}$. We'll choose our input set to be the set of inputs where:

- `lst` is always the list `[0, 1, 2, ... n - 1]`
- `x` is an integer between `0` and `n - 1`, inclusive.

In other words, we'll consider always searching in the same list (`[0, 1, ..., n - 1]`), and search for one of the elements in the list. For convenience, we'll use $\mathcal{I}_n$ to denote this set, rather than $\mathcal{I}_{\texttt{search},n}$. Now let's see how to analyse the average-case running time for this input list.

Average-case running time analysis. For this definition of $\mathcal{I}_n$, we know that $|\mathcal{I}_n| = n$, since there are n different choices for `x` (and just one choice for `lst`). From our definition of average-case running time, we have

$$Avg_{\texttt{search}}(n) = \frac{1}{n} \sum_{\texttt{(lst, x)} \in \mathcal{I}_n} \text{running time of } \texttt{search(lst, x)}$$

To calculate the sum, we need to compute the running time of `search(lst, x)` for every possible input. Let $x \in \{0, 1, 2, \dots, n-1\}$. We'll calculate the running time in terms of x.

Since `lst = [0, 1, 2, ... n - 1]`, we know that there will be exactly $x + 1$ loop iterations until x is found in the list, at which point the early return will occur and the loop will stop. Each loop iteration takes constant time (1 step), for a total of $x + 1$ steps.

So the running time of `search(lst, x)` equals $x + 1$, and we can use this to calculate the average-case running time:

$$\begin{aligned}
Avg_{\texttt{search}}(n) &= \frac{1}{n} \times \sum_{\texttt{(lst, x)} \in \mathcal{I}_n} \text{running time of } \texttt{search(lst, x)} \\
&= \frac{1}{n} \times \sum_{x=0}^{n-1} (x+1) \\
&= \frac{1}{n} \times \sum_{x'=1}^{n} x' && (x' = x + 1) \\
&= \frac{1}{n} \times \frac{n(n+1)}{2} \\
&= \frac{n+1}{2}
\end{aligned}$$

And so the average-case running time of `search` on this set of inputs is $\frac{n+1}{2}$, which is $\Theta(n)$.[1]

[1] Notice that we do not need to compute an upper and lower bound separately, since in this case we have computed an exact average.

At the end of this analysis, it was tempting for us to say that the average-case running time of `search` is $\Theta(n)$, but keep in mind that our analysis depended on the choice of allowable inputs. For this *specific* choice of inputs, the average running time was $\Theta(n)$.

Like worst-case and best-case running times, the average-case running time is a *function* which relates input size to some measure of program efficiency. In this particular example, we found that for the given set of inputs $\mathcal{I}_n$ for each n, the average-case running time is asymptotically equal to that of the worst-case.

This might sound a little disappointing, but keep in mind the positive information this tells us: the worst-case input family here is not so different from the average case, i.e., it is fairly representative of the algorithm's running time as a whole.

And as our next example will show, it isn't always the case that the average-case and worst-case running times are equal!

A second example: searching in binary lists

For our next example, we'll keep the same function (`search`) but choose a different input set to consider. Let $n \in \mathbb{N}$, and let $\mathcal{I}_n$ be the set of inputs `(lst, x)` where:

- `lst` is a list of length n that contains only `0`'s and `1`'s.[2]

[2] We call these lists *binary* lists, analogous to the binary representation of numbers we've discussed earlier.

- `x = 0`

This input set is a little more complicated than the first. Now `x` is fixed, and `lst` has multiple possibilities. But also, there are far more than n possible inputs: `lst` could be any list of length n that consists of `0`'s and `1`'s. But intuitively, because there are fewer possibilities for each element, it should be faster "on average" to find a `0`. But how much faster?

We'll now do an analysis to confirm and quantify our intuition, and introduce a more formal technique for average-case running time analysis. This technique will build on what we did in our previous example, but provide a more concrete structure that generalizes to more complex algorithms as well.

Average-case running time analysis. This running-time analysis is divided into five steps.

1. Compute the number of inputs, $|\mathcal{I}_n|$.
2. Divide the set of inputs into partitions based on the running time of each input.
3. Compute the size of each partition.
4. Using Steps 2 and 3, calculate the sum of the running times for all the inputs in $\mathcal{I}_n$.
5. Using Steps 1 and 4, calculate the average-case running time for $\mathcal{I}_n$.

Step 1: compute the number of inputs. Since `x` is always `0`, the size of $\mathcal{I}_n$ is equal to the number of lists of `0`'s and `1`'s of length n. This is 2^n, since there are n independent choices for each of the n elements of the list, and each choice has two possible values, `0` and `1`.[3]

[3] Note that the "all `0`" and "all `1`" lists are included in $\mathcal{I}_n$.

Step 2: partitioning the inputs by their running time. We now have far more inputs to worry about in this example than our previous one. To add up the running times for each input, we'll first group them based on their running time.

Looking at the implementation of `search`, we know that if `lst` first has a 0 at index i, then the loop will take $i + 1$ iterations, and therefore this many steps.[4] In other words, we can

[4] Note that `lst` could have other `0`'s after index i; the loop stops the *first* time it sees a `0`.

divide up the lists based on the index of the first `0` in the list, and treat the list `[1, 1, ..., 1]` as a special case, since it has no `0`'s. Formally, for each $i \in \{0, 1, \ldots, n-1\}$, we define

$S_{n,i} \subset \mathcal{I}_n$ to be the set of binary lists of length n where their first `0` occurs at index i. For example:

- $S_{n,0}$ is the set of binary lists `lst` where `lst[0] == 0`.
    - Every element in $S_{n,0}$ takes 1 step for `search`.
- $S_{n,1}$ is the set of binary lists `lst` where `lst[0] == 1` and `lst[1] == 0`.
    - Every element in $S_{n,1}$ takes 2 steps for `search`.
- $S_{i,i-1}$ is the set of binary lists `lst` where `lst[j] == 1` for all natural numbers j that are $< i$, and `lst[i] == 0`.
    - Every element in $S_{n,i}$ takes $i + 1$ steps for `search`.

We can't leave out the all-`1`'s list! We'll define a special set $S_{n,n}$ that contains just the list `[1, 1, ..., 1]`. This input causes `search` to take $n + 1$ steps: n steps for the loop, and then 1 step for the `return False` at the end of the function.[5]

[5] In fact, this input is the only one in $\mathcal{I}_n$ where `False` gets returned!

Step 3: compute the size of each partition. Now that we have our partitions and their running times, to add them all up we need to compute the size of each partition. Let's try to establish a pattern first.

- First consider $S_{n,0}$. A list `lst` in this set must have `lst[0] == 0`, but its remaining elements could be anything (either `0` or `1`). So there are 2^{n-1} possible lists, since there are 2 choices for each of the remaining $n - 1$ spots. Therefore $|S_{n,0}| = 2^{n-1}$.
- Next, consider $S_{n,1}$. A list `lst` in this set has two "fixed" elements, `lst[0] == 1` and `lst[1] == 0`. So there are $n - 2$ remaining spots that could contain either a `0` or `1`, and so 2^{n-2} such lists total. Therefore $|S_{n,1}| = 2^{n-2}$.
- In general, for $i \in \{0, 1, \dots, n-1\}$, lists in $S_{n,i}$ have $i + 1$ fixed spots, and the remaining $n - i - 1$ spots could be either 0 or 1. So $|S_{n,i}| = 2^{n-i-1}$.
- Finally, $|S_{n,n}| = 1$, since we defined that partition to only contain the all-1's list, and there is only one such list.

Step 4: calculate the sum of the running times. To calculate the sum of the running times of all inputs in $\mathcal{I}_n$, we can group the inputs into their partitions:

$$\begin{aligned}
&\sum_{\texttt{(lst, x)} \in \mathcal{I}_n} \text{running time of } \texttt{search(lst, x)} \\
={}& \sum_{i=0}^{n} \sum_{\texttt{lst} \in S_{n,i}} \text{running time of } \texttt{search(lst, 0)} && \text{(note } x = 0\text{)}
\end{aligned}$$

We haven't changed the values being summed, just rearranged the terms so that they are grouped together. Now the big payoff: within a single partition $S_{n,i}$, each list has the same running time, and so the body of the inner summation depends only on i, and is constant for the inner summation. We can use our work in Steps 2 and 3 to simplify this summation:

$$\begin{aligned}
&\sum_{\texttt{(lst, x)} \in \mathcal{I}_n} \text{running time of } \texttt{search(lst, x)} \\
&= \sum_{i=0}^{n} \sum_{\texttt{lst} \in S_{n,i}} \text{running time of } \texttt{search(lst, 0)} && \text{(note } x = 0\text{)} \\
&= \sum_{i=0}^{n} \sum_{\texttt{lst} \in S_{n,i}} (i+1) && \text{(from Step 2)} \\
&= \sum_{i=0}^{n} |S_{n,i}| \cdot (i+1) \\
&= \left(\sum_{i=0}^{n-1} |S_{n,i}| \cdot (i+1) \right) + |S_{n,n}| \cdot (n+1) && \text{(splitting off last term)} \\
&= \left(\sum_{i=0}^{n-1} 2^{n-i-1} \cdot (i+1) \right) + 1 \cdot (n+1) && \text{(from Step 3)}
\end{aligned}$$

Using this grouping technique, we've successfully managed to obtain a purely algebraic expression for the total running time of this function. To finish up, we'll use the *arithmetico-geometric* summation formula from Appendix C.1 (valid for all $r \in \mathbb{R}$ if $r \neq 1$):[6]

[6] That formula uses a lower bound of $i = 0$ instead of $i = 1$, but these are equivalent since the summation body equals 0 when $i = 0$.

$$\sum_{i=0}^{m-1} i \cdot r^i = \frac{m \cdot r^m}{r-1} - \frac{r(r^m - 1)}{(r-1)^2}$$

$$\begin{aligned}
&\left(\sum_{i=0}^{n-1} 2^{n-i-1} \cdot (i+1) \right) + (n+1) \\
&= \left(\sum_{i'=1}^{n} 2^{n-i'} \cdot i' \right) + (n+1) && (i' = i+1) \\
&= 2^n \left(\sum_{i'=1}^{n} \left(\frac{1}{2} \right)^{i'} \cdot i' \right) + (n+1) \\
&= 2^n \left(\frac{\frac{n+1}{2^{n+1}}}{-\frac{1}{2}} - \frac{\frac{1}{2}\left(\frac{1}{2^{n+1}} - 1\right)}{\frac{1}{4}} \right) + (n+1) && \text{(Using the formula)} \\
&= 2^n \left(-\frac{n+1}{2^n} - \frac{1}{2^n} + 2 \right) + (n+1) \\
&= -(n+1) - 1 + 2^{n+1} + (n+1) \\
&= 2^{n+1} - 1
\end{aligned}$$

That algebraic simplification was a bit tricky, but well worth it: our total running time for all inputs in $\mathcal{I}_n$ is just $2^{n+1} - 1$.

Step 5: Putting it all together. The hard work is out of the way now, and we can wrap up by calculating the average-case running time of `search` for this set of inputs:

$$\begin{aligned} Avg_{\texttt{search}}(n) &= \frac{1}{|\mathcal{I}_n|} \times \sum_{x \in \mathcal{I}_n} \text{running time of } \texttt{search}(x) \\ &= \frac{1}{2^n} \cdot (2^{n+1} - 1) \\ &= 2 - \frac{1}{2^n} \end{aligned}$$

Amazing! What we've found is that our average-case running time is $2 - \frac{1}{2^n}$ steps, which is $\Theta(1)$. This not just confirms our intuition that the average running of `search` is faster on this input set than our initial example, but show just how dramatically faster it is.

Average-case analysis and choosing input sets

We've now performed two average-case running time analyses for `search`, and gotten two different results, a $\Theta(n)$ and a $\Theta(1)$. So you might be wondering, what's the "right" average-case running time for `search`? It turns out that this is a subtle question. On one hand, we can simply say that there *is* no right answer, and that the average-case running time is always dependent on the set of inputs we choose to analyse. This is technically true, and what we've illustrated in the past two examples.

On the other hand, this response isn't necessarily very helpful, and leads to asking why we would care about average-case running time at all. In practice, we use our judgments of what input sets are realistic, and use those in our analysis. This often relies on empirical observation, for example by recording the inputs to an algorithm over a period of time and analysing their distribution. In a program P_1 that calls `search`, we might observe that the input lists often don't contain many duplicates, in which case an average-case analysis based on having n distinct elements is more applicable. In another program P_2, we might find that the input lists to `search` have elements that are within a fixed range (e.g., the digits `0` to `9`), and so might try to generalize our binary list analysis. Neither of these analyses is more "correct" than the other, but instead differ in how accurately they reflect the algorithm's actual inputs in practice.

In future courses, you'll explore more complex average-case analyses, on algorithms like quicksort and data structures like *hash tables*, which are used to implement sets and dictionaries in Python. You'll also learn how to use tools from probability theory to frame such analyses in terms of probabilities and likelihood, rather than simply counting inputs. This will allow you to simplify some calculations, and give particular inputs certain *weights* to indicate that they are more frequently occurring than others. Fun stuff!