Learning Objectives

By the end of this worksheet, you will:

- Analyse the worst-case running time of an algorithm.
- Find, with proof, an input family for a given algorithm that has a specified asymptotic running time.

---

1. **Worst-case analysis.** Consider the following function, which takes in a list of numbers and determines whether the list contains any duplicates.

```
1 def has_duplicates(lst: list) -> bool:
2     n = len(lst)
3     for i in range(n):              # Loop 1: i goes from 0 to n-1
4         for j in range(i + 1, n):  # Loop 2: j goes from i+1 to n-1
5             if lst[i] == lst[j]:
6                 return True
7     return False
```

(a) Find a tight asymptotic upper bound on the worst-case running time of this function.

Solution

First, the lines `n = len(lst)` and `return False` are constant time; we'll count these as *at most 2 steps* (a subtle detail: the `return False` might not actually execute).

For a fixed iteration of Loop 1: Loop 2 runs *at most* $n-1-i$ iterations ($j = i+1, i+2, \ldots, n-1$), where i is the loop variable for Loop 1. Each iteration of Loop 2 takes constant time (1 step), for a total running time of *at most* $n-1-i$ steps.

Loop runs *at most* n iterations ($i = 0, 1, \ldots, n-1$), and each iteration takes $n-1-i$ steps. So the total cost of Loop 1 is *at most*

$$\sum_{i=0}^{n-1}(n-1-i) = n(n-1) - \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

So the total number of steps of this algorithm is *at most* $2 + \frac{n(n-1)}{2}$, which is $\mathcal{O}(n^2)$.

(b) Prove a matching lower bound on the worst-case running time of this function, by finding (with proof) an input family whose asymptotic runtime matches the bound you found in the previous part.

For an extra challenge, find an input family for which this function *does* return early (i.e., the `return` on line 6 executes), but the runtime is still Theta of the upper bound you found in the previous part.

Solution

There's actually several possibilities—here's one idea which does have the early return.

Let $n \in \mathbb{N}$ and assume $n \geq 2$. Let $lst = [0, 1, 2, \ldots, n-2, n-2]$; it has length n, but none of its elements are equal except $lst[n-2]$ and $lst[n-1]$.

In this case, all iterations of the outer loop occur except the last one: when $i = n-2$ and $j = n-1$, the inner loop returns early. The analysis is essentially the same as part (a), except now i only goes from 0 to $n-2$, and the `return False` does *not* execute. The running time of Loop 1 for this input is

$$\sum_{i=0}^{n-2}(n-1-i) = \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{n(n-1)}{2}$$

So the total runtime is $\frac{n(n-1)}{2} + 1$ (here we added 1 step for the first line, but not the `return False`), which is $\Theta(n^2)$, matching the bound we got in part (a).

(c) Find, with proof, an input family whose running time is $\Theta(n)$, where n is the length of the input list.

Solution

Again, there are several possibilities. One is to keep the same idea of $lst = [0, 1, 2, \dots]$ from the previous part, but change the last element to something smaller.

2. **Analysing binary search**. As you've discussed in CSC108 and/or CSC148, one of the most fundamental advantages of sorted data is that it makes it easier to search this data for a particular item. Rather than searching sequentially (item by item) through the entire list of data, we can employ the *binary search algorithm*:

```
1  def binary_search(lst: List[int], x: int) -> bool:
2      i = 0                    # i is the lower bound of the search range (inclusive)
3      j = len(lst)             # j is the upper bound of the search range (exclusive)
4      while i < j:
5          mid = (i + j) // 2   # mid is the midpoint of the search range
6          if lst[mid] == x:
7              return True
8          elif lst[mid] < x:
9              i = mid + 1      # New search range is (mid+1) to j
10         else:
11             j = mid          # New search range is i to mid
12
13     return False
```

The intuition behind analysing the running time of binary search is to say that at each loop iteration, the size of the range being searched decreases by a factor of 2. At the same time, our more formal techniques of analysis seem to have trouble. We don't have a predictable formula for the values of variables i and j after k iterations, since how the search range changes depends on the contents of *lst* and the item being searched for.

We can reconcile the intuition with our more formal approach by explicitly introducing and analysing the behaviour of a new variable. Specifically: let $r = j - i$ be a variable representing the size of the search range.

(a) Let n represent the length of the input list *lst*. What is the initial value of r, in terms of n?

Solution

Since i starts at 0 and j starts at n, the initial value of r is $n - 0 = n$.

(b) For what value(s) of r will the loop *terminate*?

Solution

The loop terminates when $j \leq i$, which is equivalent to when $r \leq 0$.

(c) Prove that at each loop iteration, if the item `x` is not found then the value of r decreases by at least a factor of 2. More precisely, let r_k and r_{k+1} be the values of r immediately before and after the k-th iteration, respectively, and prove that $r_{k+1} \leq \frac{1}{2} r_k$. You can use external properties of floor/ceiling in this question.

Solution

Let i_k, j_k, i_{k+1}, and j_{k+1} be the values of i and j immediately before and after the k-th iteration, respectively. So then $r_k = j_k - i_k$ and $r_{k+1} = j_{k+1} - i_{k+1}$. There are two possibilities for how i and j change in the loop, so we'll split this analysis up into two cases.

Case 1: i stays the same, and j changes.

In this case, $i_{k+1} = i_k$ and $j_{k+1} = \left\lfloor \frac{j_k + i_k}{2} \right\rfloor$. So then we can calculate:

$$\begin{aligned} r_{k+1} &= j_{k+1} - i_{k+1} \\ &= \left\lfloor \frac{j_k + i_k}{2} \right\rfloor - i_k \\ &= \left\lfloor \frac{j_k + i_k}{2} - i_k \right\rfloor \\ &= \left\lfloor \frac{j_k - i_k}{2} \right\rfloor \\ &= \left\lfloor \frac{1}{2} r_k \right\rfloor \\ &\leq \frac{1}{2} r_k \end{aligned}$$

Case 2: j stays the same, and i changes.
In this case, $i_{k+1} = \left\lfloor \frac{j_k + i_k}{2} \right\rfloor + 1$ and $j_{k+1} = j_k$. So then we can calculate:

$$\begin{aligned} r_{k+1} &= j_{k+1} - i_{k+1} \\ &= j_k - \left(\left\lfloor \frac{j_k + i_k}{2} \right\rfloor + 1 \right) \\ &= j_k - \left\lfloor \frac{j_k + i_k}{2} \right\rfloor - 1 \\ &= - \left\lfloor \frac{i_k - j_k}{2} \right\rfloor - 1 \\ &= \left\lceil \frac{j_k - i_k}{2} \right\rceil - 1 && (\text{since } -\lfloor x \rfloor = \lceil -x \rceil) \\ &= \left\lceil \frac{1}{2} r_k \right\rceil - 1 \\ &\leq \frac{1}{2} r_k && (\text{since } \lceil x \rceil \leq x + 1) \end{aligned}$$

(d) Find the exact maximum number of iterations that could occur (in terms of n), and use this to show that the worst-case running time of `binary_search` is $\mathcal{O}(\log n)$.

Solution

Applying what we know from part (c), we can conclude that after k iterations, the value of r is at most $\left\lfloor \frac{n}{2^k} \right\rfloor$. Since the loop terminates when $r \leq 0$, this happens as soon as $2^k > n$, i.e., when $k = \lfloor \log n \rfloor + 1$.
So then there are at most $\lfloor \log n \rfloor + 1$ loop iterations, with each iteration taking constant time. The total number of steps is at most $\lfloor \log n \rfloor + 1$, which is $\mathcal{O}(\log n)$.

(e) Prove that the worst-case running time of `binary_search` is $\Omega(\log n)$. Note that your description of the input family should talk about both the input list, lst, and the item being searched for, x.

Solution

Let $n \in \mathbb{N}$, and let $lst = [0, 1, \ldots, n-1]$, and let $x = -1$.
In this case, because we have concrete values for lst and x, we *can* predict exactly how i and j will change at each loop iteration (we leave the details as an exercise).