Searching

Linear Search

- **Exercise 2.1-4**. This problem asks us to write the pseudocode for linear search algorithm, which scans an input array from beginning to the end, looking for a given element x. This is the simplest search algorithm and does not require that the input array is sorted.

```
LINEAR-SEARCH(A, x)
// Input: a sequence of numbers A[1:n] and a value x
// Output: An index i such that x equals A[i] or NIL is x is not found
1. i = 1
2. while i <= n and A[i] != x
3.    i = i + 1
4. if i == n + 1
5.    return NIL
6. else return i
```

- **Loop Invariant**: At the start of each iteration of the while loop, x is not present in A[1:i-1]
  - **initialization**: Before the start of the while loop, $i = 1$ and $A[1 : i-1] = A[1 : 0]$, an empty subarray, which does not contain x.
  - **maintenance**: Now we show that each iteration maintains the invariant. If $i = n+1$, then the loop condition fails and it terminates and the invariant for A[1:n] holds. This is the special case when the value x is not found in the array. The second condition for the while loop checks if $A[i] != x$ and only then goes into another iteration. Since by induction, x wasn't found in $A[1 : i-1]$, we can infer that $A[1 : i]$ does not contain x and the invariant holds true for the next iteration.
  - **termination**: The loop terminates if $i = n+1$, then the invariant says that x was not found in $A[1 : n]$. If the loop terminates with $i <= n$, then $A[i]$ must be equal to x for the loop condition to fail.
- **Runtime analysis**: In the worst-case, the while loop walks through the entire array. Each iteration of the loop takes $\Theta(1)$ time, so the worst-case run time is $\Theta(n)$.
- **Space analysis**: In the worst-case, the code uses one extra variable so the space complexity is $\Theta(1)$.
- **Recommended Exercise**: Sentinel: To simplify the condition in the while loop, we can put the value x at location $A[n+1]$ as a sentinel. Rewrite the above pseudocode using the sentinel.

Binary Search

If the input array is sorted, we can check the midpoint of the subarray against x and eliminate half the subarray from further consideration. Then we can apply this idea iteratively or recursively until we have either found the location of x or determined that it it doesn't exist in the input array. This algorithm is known as the binary search algorithm.

This is another example of a divide-and-conquer algorithm.

```
BINARY-SEARCH(A, x)
//Input: A[1:n] is a sorted array, x is the element we are searching for
//Output: An index i such that x equals A[i] or NIL is x is not found
1.  left = 1
2.  right = n
3.  while left <= right
4.     mid = (left + right)/2  // left + (right - left)/2 --> to avoid overflow
5.     if A[mid] < x
6.         left = mid + 1
7.     else if A[mid] > x
8.         right = mid - 1
9.     else //equal
10.        return mid
11. return NIL
```

A recursive version is actually easier to write.

```
BINARY-SEARCH-RECURSIVE(A, left, right, x)
//Input: A[low:high] is a sorted array, x is the element we are searching for
//Output: An index i such that x equals A[i] or NIL is x is not found
1.  if left > right
2.      return NIL //array is empty so not found
3.  mid = (left + right)/2
4.  if A[mid] < x
5.      return BINARY-SEARCH-RECURSIVE(A, mid + 1, right, x)
6.  else if A[mid] > x
7.      return BINARY-SEARCH-RECURSIVE(A, left, mid - 1, x)
8.  else //equal
9.      return mid
```

Notes:

- **Runtime Analysis**. The iterative version and the recursive versions are equivalent so we can analyze the recursive one for the worst case.

We can analyze the worst-case runtime of recursive binary search algorithm by drawing the recurrence tree and calculating the worst-case path length. Or we can write the recurrence equation for it as follows (each recursive call takes $\Theta(1)$ time as it has no loops):

$$\begin{aligned} T(n) &= T(n/2) + \Theta(1) \\ &= T(n/2^2) + \Theta(1) + \Theta(1) \\ &= T(n/2^3) + \Theta(1) + \Theta(1) + \Theta(1) \\ &\dots \\ &= T(n/2^k) + k \times \Theta(1) \\ &\dots \\ &= T(1) + \lg n \times \Theta(1) \\ &= \Theta(1) + \lg n \times \Theta(1) \\ &= \Theta(\lg n) \end{aligned}$$

The worst case of the recursion is unsuccessful search, which stops when $n/2^k = 1$, which implies $k = \lg n$. Note that we have assumed that n is always divisible by 2. However, in almost all cases, the result extends to a general value of n (which we can prove by induction if required).

- **Space Analysis**: Binary search requires three variables in addition to the input array. Therefore, the space complexity is $\Theta(1)$.
- **Variations**:
  - **Exponential search**: Exponential search extends binary search to unbounded lists. It starts by finding the first element with an index that is both a power of two and greater than the target value. Afterwards, it sets that index as the upper bound, and switches to binary search. A search takes $\lfloor \log_2 x + 1 \rfloor$ iterations before binary search is started and at most $\lfloor \log_2 x \rfloor$ iterations of the binary search, where x is the position of the target value.
  - **Interpolation search**:Instead of calculating the midpoint, interpolation search estimates the position of the target value, taking into account the lowest and highest elements in the array as well as length of the array. It works on the basis that the midpoint is not the best guess in many cases. For example, if the target value is close to the highest element in the array, it is likely to be located near the end of the array.
- **Implementations**:
  - **Java**: Java offers a set of overloaded `binarySearch()` static methods in the classes `Arrays` and `Collections` in the standard `java.util` package for performing binary searches on Java arrays and on Lists, respectively.

- **Python**: Python provides the `bisect` module.
- **C**: provides the function `bsearch()` in its standard library, which is typically implemented via binary search, although the official standard does not require it so.