10 Searching Static Collections

This chapter describes standard techniques for locating the items that match a specified criterion in a collection of items. This problem is known as searching. We will look at two standard algorithms—linear search and binary search—and discuss their implementations for simple static collections. After that, we will look at their use beyond searching such simple collections.

Searching is undoubtedly a basic operation in computing. As an example, suppose we have an array of student records (objects), as shown below. An ellipsis (...) indicates that more attributes exist than shown here.

```
[
  {"id": 58001, "name": "John Green", "height": 182, ...},
  {"id": 58011, "name": "Karen Best", "height": 149, ...},
  {"id": 57009, "name": "Amy Brown", "height": 169, ...},
  ...
  {"id": 59018, "name": "Jake Jacob", "height": 189, ...}
]
```

On this collection of data, questions such as the following often arise: given an id, what is the name? Or given a name, what is the height? These are examples of searching questions. When we are searching by a value k (i.e., locating an item whose certain attribute matches this value), this k is known as the search *key*. In general, searching is used to test if a key belongs in a collection and also to retrieve a record corresponding to that key.

Searching. Searching is much like finding a needle (an item) in a haystack (in a collection or a "logical" collection). This is one example where locating where the item is generally requires more work than simply looking to see if the item is really at that location. For a collection that has n items, we will see the linear search algorithm that assumes essentially nothing about the collection and runs in $O(n)$ time. We will also see the much faster binary search algorithm that requires the collection to be sorted but runs in just $O(\log n)$ time!

10.1 Linear Search

Linear search works practically universally. It makes no assumptions about the collection being searched nor about the key. The only assumption is we can go over each and every member of the collection to search. To begin, we will work with Java arrays, which are easy to traverse.

As a warm-up, we will implement a simple variant that simply tests whether or not a key belongs to an array. Specifically, we are writing a function

```
boolean exists(int[] A, int key)
```

that takes as input an array `A` of `int`s, together with a key `key`, and returns whether the key `key` exists in the array `A`. Implementing this is straightforward:

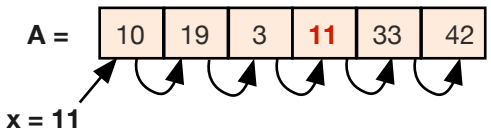
```
boolean exists(int[] A, int key) {
    for (int elt : A)
        if (elt == key) return true;
    return false;
}
```

This code uses a for-each construct but is equally natural to write it using a loop that traverses the array using an index.

For a concrete example, suppose the function is called using the following parameters:

```
int[] A = {10, 19, 3, 11, 33, 42};
boolean r = exists(A, 11);
```

Then, pictorially, the code is following the arrow in the diagram below until either it hits that key or reaches the end of the array, each time testing if the item there matches the key:



If, instead, we're interested in returning the index where that key is found. The specification becomes to write a function

```
int find(int[] A, int key)
```

that takes as input an array `A` of `int`s, together with a key `key`, and returns a number `i` such that `A[i] == key` or -1 if that key does not exist in `A`.

To update the code, when we consider an element, we will want to know the index in addition to what that item is. In this case, accessing the array using indexing seems cleaner:

```
int find(int[] A, int x) {
    for (int i=0;i<A.length;i++)
        if (A[i] == x) return i;
    return -1;
}
```

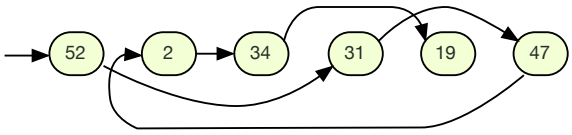
Could we have done better? The answer is no. Without any additional structure or information about the collection or the key being looked up, we cannot hope to do much better than going through every one of them. This argument gives a lower bound of $\Omega(n)$.

Running Time Analysis. What is the running time of these implementations? It is easy to see that they all have the same complexity of $O(n)$: In addition to the setup code (which takes constant time), the bulk of the work is the **for** loop that goes over all of the items in the array, a total of $n = \texttt{len}(A)$ times. Each time, it does a constant amount of work, only doing comparison and increment operations. Hence, the total cost is $O(n)$.

Does this algorithm really going to take $O(n)$*?* At least it does for some input. Consider, for example, inputs where the key is not present in `A`. In order for it to know that the key is not in `A`, the algorithm has to look through all the items—and that is $\Theta(n)$ time spent.

Generalization

The only assumption we made about linear search is simply that we can go over all of the members of that collection. In other words, the linear search algorithm "unfolds" the collection into a linear structure despite the underlying representation. For example, we could have a linked list where the underlying physical organization requires pointer chasing, but the conceptual view is that it is a linear list and linear search will look at each item in its logical—i.e., linear list—order.



This means that in addition to arrays, any structure where such traversal is possible can use linear search. Incidentally, in Java, this quality is known as being iterable. The following code offers a generalization of our linear search code to any iterable collection:

Code 10.1: Linear search on an `Iterable`

```
int find(Iterable<T> A, T key) {
    int i=0;
    for (T elt : A) {
        if (elt.equals(key)) return i;
        ++i;
    }
    return -1;
}
```

Several things are different from before. Because `A` is an iterable, we can use the for-each syntax to loop over `A`. At the same time, we store our position in the variable `i`, which is incremented every time we move on to the next element. Furthermore, we have updated the key type to be a generic type `T`. This means we need to use `.equals` to compare whether two given items are equal. Note that there are other ways to traverse an iterable collection. Readers should look into the related concept of `Iterators`.

10.2 Binary Search

As was argued earlier, searching requires $\Omega(n)$ time in general because all items in the collection have to be examined. We will now see if improvements can be made if we make certain assumptions about the collection.

The assumption we will be making is a simple one: the collection being searched keeps the items sorted by the search key. For ease of exposition, we will think of an array ordered from small to large by the search key.

Sorted Arrays. If an array $A = [a_1, a_2, \ldots, a_n]$ is sorted, then $a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$.

The hope is that by sacrificing generality (only operating on sorted arrays), we can bring the running time down significantly: We will show that $O(\log n)$ per search is possible even in the worst case. Concretely, we're aiming to write a function

```
int find(int[] A, int key)
```

that takes as input a sorted array `A` of `int`s, together with a key `key`, and returns a number `i` such that `A[i] == key` or -1 if that key does not exist in `A`. This is the same specification as before; we only want to do it faster!

Recursive Thinking Plan

We will design a recursive algorithm for the task. To design the recursive process, we will use the same blueprint as seen earlier. Let us answer the following questions:

Nothing can appear in the empty array!

(Q1) **How to solve small instances?** We begin by setting how we measure the input size. We anticipate working with smaller and smaller arrays, so it makes sense to use the array size as our measure. Therefore, when calling `find(A, key)`, we say the size of this problem is $n = $ `len(A)`.

As usual, we have the liberty of choosing the smallest instance that won't be solved recursively. It is natural to pick the smallest possible input—an empty array. We know for fact that an empty array contains nothing, so no key is in that array. Hence, we have `if (A.length==0) return -1`.

(Q2) **How to tackle an instance in terms of smaller instances?** Consider a call to `find(A, key)`, where `len(A)` $= n$. Say, for any $0 \leqslant$ `len(T)` $< n$ and `key`, we already know how to compute `find(T, key)`. The question to answer now is: *can we compute* `find(A, x)` *in terms of* `find` *calls on smaller inputs?*

We wish to split the problem in half, and it seems like we could something better than scanning the whole array because it is sorted.

Sorted Array. In a sorted array $A = [a_1, a_2, \ldots, a_n]$, if the middle index is $m =$ `len(A)/2`, then,

$\ldots \leqslant$	a_m	$\leqslant \ldots$

Tips

If we compare the search key `key` with the element in the middle of the array, then we can decide which half to look at next depending on the comparison outcome.

More specifically, if `m = len(A)/2`, then because `A` is sorted in ascending order, we know that

- If `key == A[m]`, the key is found at `m`
- If `key < A[m]`, the search key appears *before* `m`, so we should look at `A[:m]` next.
- If `key > A[m]`, the search key appears *after* `m`, so we should look at `A[m+1:]` next.

Note that in no case do we need to look at `A[m]` again because failing the first condition, we have determined that `A[m] != key`. Importantly, the problem size becomes smaller in all cases—either immediately done or the problem size has shrunk in half.

We translate this idea into code as follows:

```
int find(int[] A, int key) {
    if (A.length == 0) return -1;

    int m = A.length/2;
    if (A[m] == key) return m; // found at m
    if (key < A[m]) // key is less than A[m]
        return find(Arrays.copyOfRange(A, 0, m), key);
    // otherwise, key is more than A[m]
    int r = find(Arrays.copyOfRange(A,m+1,A.length), key);
    if (r < 0) return -1;
    else return m + 1 + r;
}
```

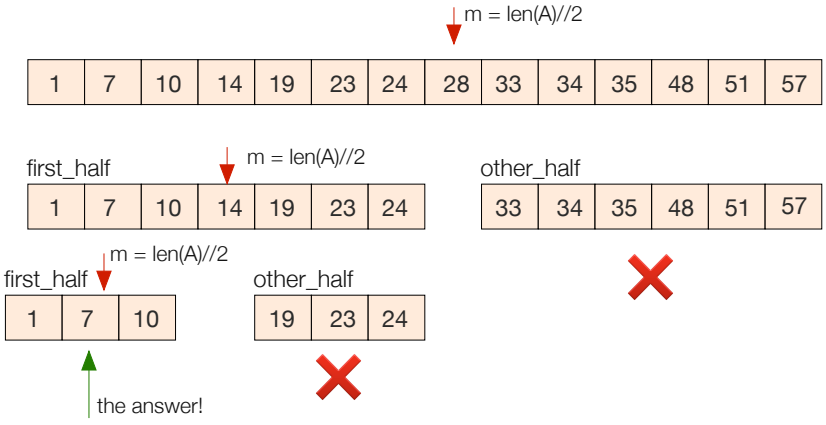
Notice how the case where the search key is more than `A[m]` is handled. Because the recursive call can have two outcomes—found at some index in the sliced array or not found (-1)—we need to check the result of `find` and adjust it accordingly before returning the value.

Let us take a quick look at an execution of the algorithm.

Example. The sorted input array is

```
[1, 7, 10, 14, 19, 23, 24, 28, 33, 34, 35, 48, 51, 57]
```

and the search key is 7. Then, once `find` is called, the following sequence of operations will unfold:



Does it work with arbitrary arrays? We expect the answer to be no, but why? If the array is unordered, unlike in the example above, the problem is we cannot say for certain which side to look into next—because either side could contain the key we are looking for. For a concrete example, try searching for the key 5 in the following array:

```
[10, 19, 3, 11, 33, 42, 5]
```

Running Time

This code above follows the recipe of reducing the problem size in half. *But does it help and how much does it really help?* We will analyze the time complexity of this program. The plan is to write a recurrence in the problem size.

> Let $T(n)$ denote the time `find` takes to search for any key in a sorted array of length n.

Then, it follows that $T(0) = O(1)$. For $n > 0$, we determine the cost of $T(n)$ as follows: First, we perform constant work (computing the middle index, testing the middle element with the key) to arrive at the conclusion about the relative position between `A[m]` and `key`. Every step is $O(1)$ so far. After that, three things could happen:

- If `key == A[m]`, the key is at m. We return right away—the cost is $O(1)$.
- If `key < A[m]`, we look at `A[:m]` next. The cost here is the cost of calling `find(A[:m])` and constructing `A[:m]`. The length of `A[:m]`, as was determined before, is m, so the former cost is $T(m)$, and the latter cost is $O(m)$ via `Arrays.copyOfRange`. Knowing that $m = n/2$, we have $T(m) = T(n/2)$ and $O(m) = O(n)$.
- If `key > A[m]`, we look at `A[m+1:]` next. The cost here is the cost of calling `find(A[m+1:])` and constructing `A[m+1:]`. The length of `A[m+1:]`, as was determined before, is $n - (m+1) + 1 = n - m$, so the former cost is $T(n-m)$, and the latter cost is $O(n-m)$. But $m = n/2$, so $T(n-m) = T(n/2)$ and $O(n-m) = O(n)$.

The cost of $T(n)$ is therefore bounded from above by the largest of the three possibilities. Hence, $T(n) \leqslant T(n/2) + O(n)$, which solves to $T(n) \in O(n)$.

Unfortunately, with these "upgrades," we still haven't beaten the complexity of linear search!

Eliminating the Most Costly Operations

To improve upon this, we need to identify the "bottlenecks" in our algorithm—the most costly steps that drag everything else down with them. It turns out this is not difficult to locate: making a copy of `A` (via `copyOfRange`) is costly—it has linear complexity.

The copy operation makes a copy of half of the array, so it necessarily spends linear time. To avoid this costly step, we will use two indices to demarcate where the search will take place. We'll revise the algorithm by adding a helper function

```
private int findHelper(int[] A, int lo, int hi, int key)
```

which looks for `key` between `A[lo],...,A[hi-1]`. We introduce a helper function because the parameters of the function that we need internally differ from the interface we want to expose to the user—we hide the internal "mess" by making `find` call the helper function. This way, our users will enjoy the same interface to `find` despite what has been changed internally.

Rather than redoing all the steps again, we will simply note the major differences from the previous version:

- While the previous version measures the problem size in the length of `A`, this version will measure the size in $n =$ `hi - lo`, which is the true problem size. Our code will not look outside the range `lo,...,hi-1`).
- Moreover, we no longer need to compensate for the index offset due to slicing. Hence, for the `key > A[m]` case, we no longer need to add $m+1$ to the result.

Code 10.2: Fast binary search

```
1  private int findHelper(int[] A, int lo, int hi, int key) {
2      if (lo >= hi) return -1;
3
4      int m = (lo+hi)/2;
5
6      if (A[m] == key) return m;
7
8      if (key < A[m])
9          return findHelper(A, lo, m, key);
10     return findHelper(A, m+1, hi, key);
11 }
12
13 int find(int[] A, int key) {
14     return findHelper(A, 0, A.length, key);
15 }
```

The revised code is shown in Code 10.2. Notice that inside `find`, we call `findHelper` with the range 0 through `A.length`, meaning the real valid indices are $0, 1, \ldots,$ `A.length-1`. Also, the effect size of the array (i.e., the number of items still under consideration) is $n =$`hi - lo`. This makes the midpoint of the this range `m = (lo + hi)/2`*.

Running Time. To analyze the complexity of the new `find`, we resort to writing a recurrence relation for `findHelper`, which is the real workhorse. Let $n =$ `hi` $-$ `lo`. Then,

$$T(0) = O(1) \text{ and}$$
$$T(n) = T(n/2) + O(1)$$

This is true because for $n > 0$, in all cases, we only perform a constant number of arithmetic and comparison operations. This recurrence solves to $T(n) \in O(\log n)$. Therefore, using binary search, finding any key at all in a sorted collection takes $O(\log n)$ time.

Remarks

In Java, a family of functions

`Arrays.binarySearch` `Collections.binarySearch`

are readily available. Though, there will be situations that require specialized versions of binary search. These variants of binary search can often be implemented using the very same idea but with some modifications. For example, if the array we are searching has multiple entries of the key, we could customize binary search to look for the first element or to look for the last element. The exercises at the end of the chapter explore these further.

*To prevent overflow, it is better to write `m = lo + (hi - lo)/2`

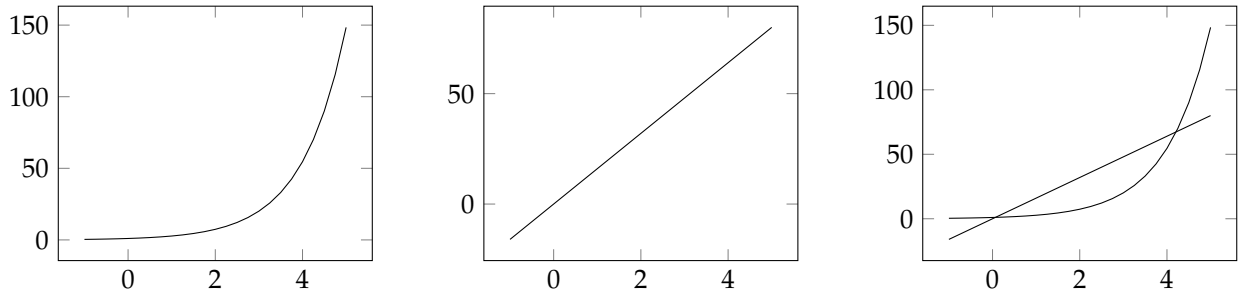
10.3 Applications

There are numerous applications of these common searching techniques. We will look at two, which at first glance don't seem like searching problems.

Equation Solving

What does the curve e^x look like? What about $16x$? Considering the trends of these curves, we realize they must intersect each other somewhere. But where exactly is this? Mathematically, we wish to solve for an x for which

$$e^x = 16x.$$

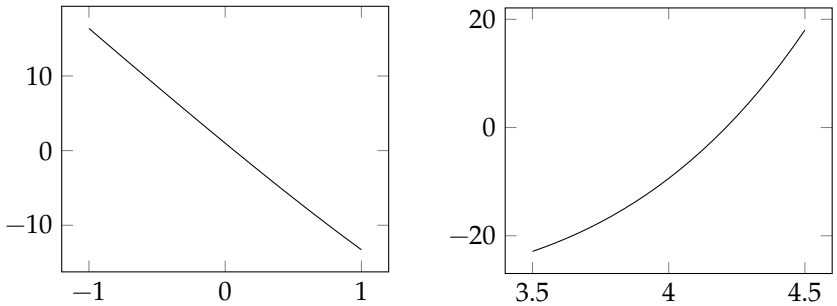


At first glance, this may seem like a problem that one was told to solve back in an early algebra course. But a few attempts at manipulating this equation don't seem to give an analytical solution. As a computer scientist, you may wonder, *what we can do with a fast computer and laziness?*

Once again, at first glance, this does not look much like a search problem. But we can transform the problem slightly. When we ask for an x for which $e^x = 16x$, we can equivalently ask for an x such that $e^x - 16x = 0$. Call this expression $f(x)$, so

$$f(x) = e^x - 16x.$$

When we plotted it above, we saw that there are two places where these curves intersect: one between -1 and 1 (to be super crude), and one between 3.5 and 4.5. Let us now zoom in on $f(x)$ on these two ranges separately.



Suppose we would like to focus on finding the x between 3.5 and 4.5. What can we do?

Linear Search. We cannot quite solve it exactly. But we can approximate the answer. If we are happy with reaching the 100-th (i.e., correct within ± 0.01), we could try every value between 3.5 and 4.5 in increments of 0.01. There are many values to try: $(4.5 - 3.5)/0.01 = 100$. But this is still manageable. In general, if we are looking in the range $[a, b]$ and we are hoping to have precision within $\pm\varepsilon$, then the total time required will be $O((b-a)/\varepsilon)$ because there are $\frac{b-a}{\varepsilon}$ values to try.

In applying linear search, we subdivide $[a, b]$ into evenly-spaced points and look for a point x that minimizes $|f(x)|$. The function f needs *not* be monotone in this range.

Binary Search. One nice property of $f(x)$ in the range 3.5 to 4.5 is that $f(3.5) < 0$ and $f(4.5) > 0$. Furthermore, the function f is increasing in that range. This means in that range, if $x < x'$, then $f(x) < f(x')$. In other words, it's sorted! Hence, we could apply binary search to look for an x where $f(x) = 0$. If the precision sought is ± 0.01, there are 100 values in the space, so it takes about $\log_2(100)$ probes. Therefore, in general, it costs us around $O(\log((b-a)/\varepsilon))$ time to look for such an answer, a marked improvement over linear search for sure.

We are technically working with the same evenly-spaced set of points but by assuming that f is monotone in this range, we can apply binary search to find an x that minimizes $|f(x)|$.

This idea of applying binary search to root finding is also known as the bisection method, one of the basic methods for finding roots numerically. Readers interested in this type of algorithms should look further into a course/book on numerical methods.

The Stuttering-Substring Problem

Consider two strings A and B. We say that A is a *substring* of B if it is possible to find the letters of A inside of B in the same order that they appear originally in A, potentially skipping some letters of B. More formally, we say that A is a substring of B if there are indicies $0 \leqslant i_0 < i_1 < i_2 < \cdots < i_{n-1} < m$ into B such that

$$B[i_k] = A[k], \qquad \forall\, k = 0, 1, 2, \ldots, n-1.$$

As examples, "cat" is a substring "excavate", but "vet" isn't a substring of "excavate".

It is a simple exercise to check whether A is a substring of B. This can actually be done in $O(\texttt{len}(A) + \texttt{len}(B))$ time. For the rest of this discussion, we will assume a function `boolean isSubstr(String A, String B)` that checks whether `A` is a substring of `B` in the desired running time.

Putting that aside for now, if A is made up of $A = a_0 a_1 a_2 \ldots a_{n-1}$, we define the concept of *stuttering* as follows: $A^{(k)}$ expands the string A by repeating each a_i consecutively k times. We define $A^{(0)}$ to generate the empty string, which is a substring of any B. For example, if $A =$ `"hello"`, then $A^{(2)} =$ `"hheellllloo"`. And if $A =$ `"hi"`, then $A^{(5)} =$ `"hhhhhiiiii"`.

Given strings A and B, the *stuttering-substring problem* is to find the largest $k \geqslant 0$ such that $A^{(k)}$ is a substring of B. This problem is well-defined because for sure $A^{(0)}$ is a substring of B. But what is the largest such k?

As an example, consider $A =$ `"ho"`. and $B =$ `"hi and hello jello"`. It is easy to see that each of $A^{(0)}$, $A^{(1)}$, and $A^{(2)}$ is a substring of B while $A^{(3)}$ is not. Therefore, in this particular example, the answer is 2.

Linear Search. To solve this problem, we could try linear search, probing all k in increasing order ($k = 0, 1, 2, \ldots$) until we find one that is no longer a substring. We will analyze this algorithm briefly. Suppose $n = \texttt{len}(A)$ and $m = \texttt{len}(B)$. Then, since for each k that we try, we pay $O(k \cdot n + m)$ to use `isSubstr`. This means that if t is the smallest point where it is no longer a substring, then the total cost will be

$$\sum_{k=0}^{t} (k \cdot n + m) = n \cdot \frac{t(t+1)}{2} + mt \in O(nt^2 + mt).$$

We further know that t can never exceed $m/n + 1$ because at that point the length of $A^{(\frac{m}{n}+1)}$ is already $m + n > m$ and cannot be a substring of B for sure. Hence, with the knowledge that $t \leqslant m/n + 1$, we conclude that the worst-case running time of our linear search is

$$O\left(n \cdot \left(\tfrac{m}{n} + 1\right)^2 + m(\tfrac{m}{n} + 1)\right) = O(m^2/n).$$

Binary Search? (How) can we apply binary search? As we saw already, an important requirement is that the space being searched has to be ordered. But this is already so. Moreover, we can think of the space as a conceptual array `a` where the entries are only 1s and 0s—and all the 1s come before all the 0s. In this setup, `a[i]` is 1 if $A^{(i)}$ is a substring of B. Therefore, we can readily apply binary search to find the last 1. We further know that the smallest possible k is 0 and the largest possible k is $m/n + 1$, hence giving us the lower- and upper- bounds. Details are left to readers to figure out (in Exercise 10.10.). But it suffices to say that using binary search in a space of size $(m/n + 1) - 0 + 1 = m/n + 2$ will take at most $O(\log(m/n))$ tests, where each test is a call to `isSubstr`. But the length of $A^{(k)}$ and B never exceed $m + n$, so each test (via `isSubstr`) is $O(m + n)$. This means the total running time is $O((n + m)\log(m/n))$ in the worst case.

Exercises

Exercise 10.1. Suppose the input sequence in Code 10.1 is instead given as an `Iterator<T>`. Rewrite `find` accordingly.

Exercise 10.2. Consider the sorted array

```
[1, 7, 10, 14, 19, 23, 24, 28, 33, 34, 35, 48, 51, 57]
```

Step through the algorithm in Code 10.2 looking for key `key = 48`.

Exercise 10.3. Let $n \geqslant 2$ be a power of two. The binary search algorithm in Code 10.2 returns as soon as it finds the search key. Design an array of length n and a key that exists in that array so that our binary search implementation takes at least $2\log_2(n) - 1$ comparisons. Count the comparison `A[m] == key` as one and `key < A[m]` as another.

Exercise 10.4. Reimplement the logic of `findHelper` in Code 10.2 without using recursion.

Exercise 10.5. Suppose we are to reimplement binary search from Code 10.2 for use with a `LinkedList` as opposed to a fixed-size array. Would it still be correct? What about the running time in terms of the length of the list n?

Exercise 10.6. Our binary search algorithm, as discussed, has the property that if the key we're looking for appears multiple times, the algorithm may return the index of any of them.

Adapt the binary search algorithm so that it always returns the index of the last occurrence of the search key. More precisely, let a be a sorted array of elements (i.e., $a[0] \leqslant a[1] \leqslant \cdots \leqslant a[n-1]$, where n is the length of a). You are to implement a function `Integer binarySearchLast(int[] a, int k)` that returns the value of $\max\{i \mid a[i] == k\}$; or otherwise, returns `null` because the key `k` is not present in the input array. For example,

- `binarySearchLast({1,2,2,2,4,5}, 2)` returns 3.
- `binarySearchLast({1,2,2,2,4,5}, 0)` returns `null`.
- `binarySearchLast({1,2,2,2,4,5}, 5)` returns 5.

Whatever the input may be, your implementation must take at most $O(\log n)$ time, where $n =$ `len(a)`.

(Hints: Your code must run in $O(\log n)$ *even in the case when there is a long streak (say of length about* n*) of the key we're searching for. It won't be fast enough to use standard binary search and keep moving right from that point by one until we reach the last key.*

Exercise 10.7. The *rank* of an element e with respect to an array L, denoted by $r(L, e)$, is the number of elements in L that are less than e. For example, if L is the array {`10`, `21`, `32`, `53`, `54`}, then

- $r(L, 9)$ is 0
- $r(L, 10)$ is 0
- $r(L, 33)$ is 3 because 10, 21, and 32 are *all* smaller than 33.

Let $n =$ `len(A)` and $m =$ `len(B)`. Denote by $A + B$ the array obtained by concatenating A with B. We guarantee that $A + B$ *contains no repeated numbers.*

Your Task: Write a function

```
int rank(int[] A, int[] B, int e)
```

that takes two *sorted* arrays A and B and an integer e, and computes the rank $r(A + B, e)$ *without* actually expanding out $A + B$. Your code must run in at most $O(\log(n + m))$ time or faster.

Exercise 10.8. Continuing from Exercise 10.7., write a function

```
Integer select(int[] A, int[] B, int k)
```

that locates and returns the rank `k` element from $A + B$. To be precise, your code must return the same answer as `sorted`$(A + B)$`[k]` (only faster!), where `sorted` returns the sorted version of the input. If the rank `k` element is not present, you will return `null`. Your code must run in at most $O(\log^2(n + m))$ time or faster.

BONUS: If you are up for an extra challenge, improve the running time to $O(\log(n+m))$ or faster.

Exercise 10.9. Let $f(x) = e^x - 16x$. Write a function

```
double solve(double eps)
```

that uses binary search to determine an x between 3.5 and 4.5 where $|f(x)| <$ `eps`. This means solving $f(x)$ up to $\pm\varepsilon$, where $\varepsilon =$ `eps`.

Exercise 10.10. The stuttering-substring problem is as defined earlier in the chapter. You will solve this problem in a few steps:

(i) Implement a function `boolean isSubstr(String a, String b)` that takes two strings `a` and `b` as input, and returns a Boolean indicating whether `a` is a substring of `b`. It must run in $O(\texttt{len(a)} + \texttt{len(b)})$ time.

(ii) You will analyze why your function meets in running time bound. *(Hint: Argue that each letter is "touched" at most once. Look at the* `merge` *function in merge sort for inspiration.)*

(iii) Write a function `String stutter(String A, int k)` that takes a string `A` and a number $k \geqslant 0$ and produces $A^{(k)}$. For `A` of length n, we expect this function to run within $O(nk)$ time.

(iv) Implement (perhaps by reusing the logic from a previous exercise as well) a function `int maxStutter(String a, String b)` that takes as input two *nonempty* strings `a` and `b` and returns a number that is the answer to the stuttering-substring problem for the pair of strings.

(v) Analyze the running time of your implementation. We expect your code to run in at most $O((m+n)\log(\frac{m}{n}))$ time, where $m = \texttt{len(B)}$ and $n = \texttt{len(A)}$.

(Hint: To analyze the running time, write a recurrence to determine the number of "probes." How should we define a probe? If the number of probes is $O(P(n))$ *and each probe costs at most* $O(f(n))$*, then the total running time is at most* $O(P(n)f(n))$*.)*

Chapter Notes

Techniques for finding a matching item in a static collection dates back to antiquity. Use of linear search in computing is probably as old as the computing field itself. Introducing ordering (i.e., sorting the collection) is an ancient idea that brings about more structure to the collection, enabling search to be carried out more efficiently. Further optimization to binary search is discussed in, for example, Sedgewick and Wayne [SW11]. Readers should know that besides searching a collection as discussed in this chapter, binary search is also used in the context of root finding in numerical computations (often known the bisection method in that context). The stuttering-substring problem was studied previously in the literature. Mirzaian [Mir87] presents an optimal linear-time algorithm for the problem, a faster running time than what is asked for in the exercise.