Lecture 7: Searching

built on 2019/05/14 at 13:08:41

The main theme of this lecture is, how to find a needle (an element) in a haystack (in a collection or a "logical" collection)? This is a basic operation in computing. For instance, say we have a list of IDs and preferred names at an unnamed college:

58001	John
58016	Best
58009	Boss
58006	Bank
58021	Kelly
58011	Mint

And we're interested in answering questions like, (1) given an ID, what's the preferred name? and (2) given a preferred name, what is the ID?

In the literature, when we're searching by a value x, we usually call that x a *key*. Typically, we use this to test if a key belongs to a collection and also to retrieve a value corresponding to that key.

In the example above, if we're looking up 58011, that is the key we're looking for, and the collection we're looking into contains keys and their associated values.

1 Linear Search

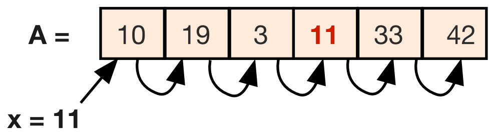
The first variant we're going to look at is also the most generic one: we make no assumptions about the collection being searched nor the key. The only assumption we're making is we can go over each and every member of that collection.

We'll start with lists. It is easy to traverse a list, either referring to each element in the list using an index or using a **for** construct that goes over the list in the list order.

Before we attempt the version that returns the corresponding value or the version that returns the location, let's write a simple version which tests if a key belongs to a list. Writing this is easy:

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

Pictorially, we're following the arrow in the diagram below until either we hit that key or we reach the end of the list, each time testing if the element we're seeing matches the key:



If, instead, we're interested in returning the index where that key is found. To update the code, when we consider an element, we'll want to know the index in addition to what that element is. There are many ways to go about this. Let's look at on way to do it:

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

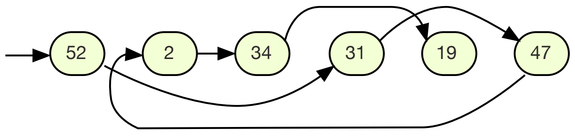
What's the running time of these implementations? It's easy to see that they all have the same complexity of $O(n)$. The reason is that in addition to the setup code (which takes constant time), the bulk of the work is the **for** loop that go overs all of the elements in the list, a total of $n = \texttt{len}(A)$ 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 x is not present in A. In order for it to know that x isn't in A, the algorithm will have to look through all the elements—that's $O(n)$ work.

Could we have done better? The answer is no. Without any additional structure/information about the collection or the key being looked up, we can't hope to do much better than going through every one of them.

1.1 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:



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:

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

Because A is iterable, we can use the "foreach" 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. Perhaps, a less technical piece of code would be:

```
public static <T> int find(Iterable<T> A, T key) {
  Iterator<T> it=A.iterator();
  int i=0;
  while (it.hasNext()) {
    T elt=it.next();
    if (elt==key) return i;
    i++;
  }
  return -1;
}
```

2 Binary Search

We'll continue with the theme of searching for a particular element in a collection. This time we'll make an assumption that the collection we're looking into is a list and is sorted, either from small to large, or large to small. For ease of presentation, we'll only think of a list ordered from small to large.

We hope to improve upon the running time of $O(n)$ which we saw for linear search that works on any list at all. By sacrificing generality (only operating on sorted lists), we wish to do better. In fact, we'll bring the running time down to $O(\log n)$ per look up.

For concreteness, the problem we're solving is as follows:

> `lookup`(A, x) that takes an array A sorted in ascending order and produces an index i such that $A[i] = x$ or `-1` if $x \notin A$.

We'll design a recursive algorithm for the task. To design the recursive process, we'll answer the following questions:

(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 `lookup`(A, x), we say the size of this problem is $n = \texttt{len}(A)$.
As usual, we have the liberty of choosing the smallest instance that won't be solved recursively. It's natural to pick the smallest possible input—an empty array. We know for fact that an empty array contains nothing, so x is not 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 `lookup`(A, x), where $\texttt{len}(A) = n$. Say, for any $0 \leqslant \texttt{len}(T) < n$ and x, we already know how to compute `lookup`(T, x). The question to answer now is: *can we compute* `lookup`(A, x) *in terms of* `lookup` *on smaller inputs?*
To solve this question, we take inspiration from previous lectures—we wish to split the problem in half. However, it seems like we could something better than scanning the whole array because the list is ordered.
The crux: If we compare the search key x 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 $m = \texttt{len}(A)/2$, then because A is sorted in ascending order, we know that

- If $x == A[m]$, the key we're looking for is at m
- If $x < A[m]$, the key we're looking for appears *before* $A[m]$, so we should look at `A[:m]` next.
- If $x > A[m]$, the key we're looking for appears *after* $A[m]$, so we should look at `A[m+1:]` next. Note that we don't need to look at $A[m]$ again because we have already determined that $A[m] \neq x$.

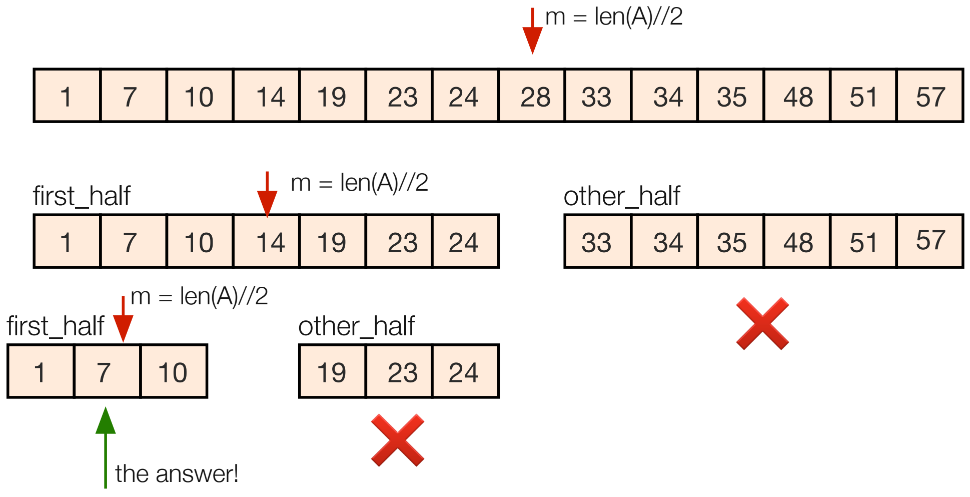
Importantly, the problem size becomes smaller in all cases (It's a simple exercise to check that).

We translate this idea into code as follows:

```java
public static int lookup(int[] A, int k) {
  if (A.length==0) return -1;
  else {
    int m=A.length/2;
    if (A[m]==k) return m;
    else if (k < A[m]) return lookup(Arrays.copyOfRange(A, 0, m), k);
    else {
      int r=lookup(Arrays.copyOfRange(A,m+1,A.length), k);
      if (r<0) return -1;
      else return m + 1 + r;
    }
  }
}
```

2.1 How This Works?

We'll try to understand how this code actually works. Consider the following example. Say we're given the following list and we're looking for 7:



Does it work with arbitrary lists? Let's see for ourselves. If the list is unordered, like in the example above, the problem is we can't say for certain which side to look into next—because either side could contain the key we're looking for. Try searching for 5 in the following list:

```
10 19 3 11 33 42 5
```

2.2 Running Time

This code above uses the recipe from our previous lesson where reducing the problem size in half is supposed to help. *But does it help and how much does it really help?* We'll analyze the time complexity of this program. The plan is to write a recurrence in the problem size.

Let $T(n)$ denote the time `lookup` 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 between $A[m]$ and x. That's $O(1)$ so far. After that, three things could happen:

- If $x == A[m]$, the key we're looking for is at m. We return right away—the cost is $O(1)$.
- If $x < A[m]$, we look at `A[:m]` next. The cost here is the cost of calling `lookup(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 $O(m)$. Knowing that $m = n/2$, we have $T(m) = T(n/2)$ and $O(m) = O(n)$.
- If $x > A[m]$, we look at `A[m+1:]` next. The cost here is the cost of calling `lookup(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 according to our table, is $T(n) \in O(n)$.

Unfortunately, with a lot of thinking so far, we still haven't beaten the complexity of linear search!

2.3 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 (`copyOfRange`) is bulky——it has linear complexity.

The copy operation makes a copy of half of the list, so it necessarily spends linear time. To avoid this costly step, we'll use two indices (like in binary search) to demarcate where the search will take place. We revise the algorithm, adding a helper function `lookup_helper(A, lo, hi, x)`, which looks for x 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 `lookup` call the helper function. This way, our users will enjoy the same interface to `lookup` despite what's been changed internally.

In lieu of redoing all the steps again, we'll 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 (we aren't interested in looking outside the range `lo`, ..., `hi-1`). Moreover, we no longer need to compensate for the index offset due to slicing. Hence, for the $x > A[m]$ case, we no longer need to add $m + 1$ to the result. In code:

```
int lookup_helper(int[] A, int lo, int hi, int x) {
  if (lo >= hi) return -1;
  else {
    int m = (lo+hi)/2;
    if (A[m]==x) return m;
    else if (x < A[m]) return lookup_helper(A, lo, m, x);
    else return lookup_helper(A, m+1, hi, x);
  }
}

int lookup(A, k) { return lookup_helper(A, 0, A.length, k); }
```

To analyze the complexity of the new `lookup`, we resort to writing a recurrence relation for `lookup_helper`, which is the real workhorse. Let $n =$ `hi` $-$ `lo`. Then, $T(0) = O(1)$ 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 $T(n) \in O(\log n)$ via a table lookup.

Exercise: Prove that this works as intended.

In real life, there's a family of functions `Arrays.binarySearch` and `Collections.binarySearch` that you can readily use in Java.

2.4 Remarks

There are variants of binary search that build on the very same idea. For example, if the array we're 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. You'll explore more of these in your assignment.

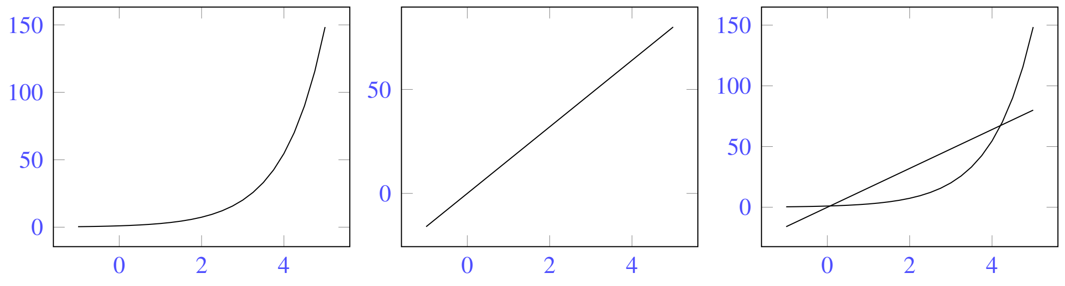
3 Applications

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

3.1 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

$$e^x = 16x.$$

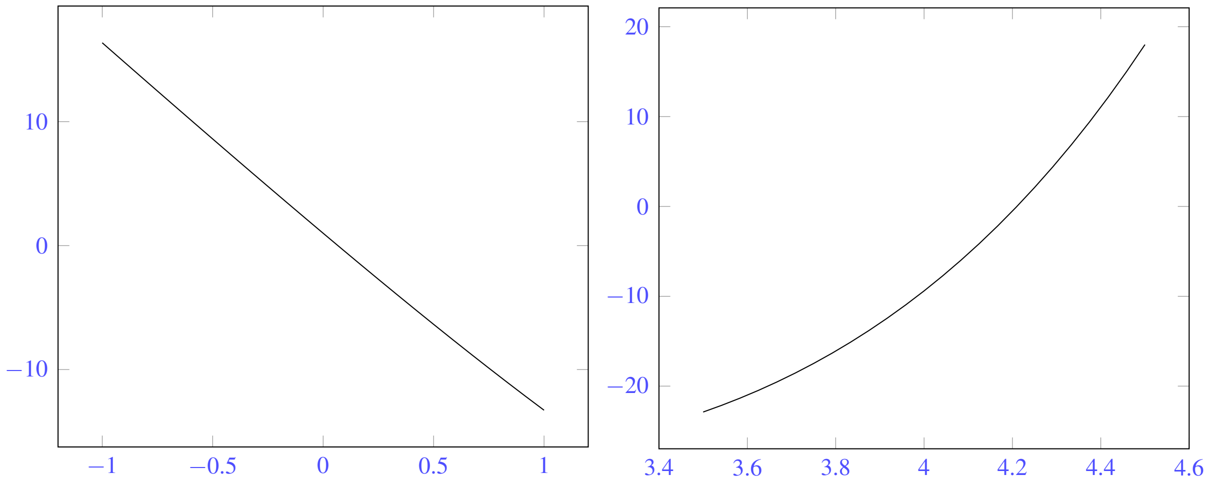


At first glance, this may seem like a problem that one was told to solve back in an early Algebra course. But a few attempts at manipulating this equation don't seem to give an analytical solution. As a computer scientist, you may wonder, *what we can do with a fast computer and laziness?*

Once again, at first glance, this doesn't look much like a search problem. But let's transform the problem slightly. When we ask for an x for which $e^x = 16x$, we can equivalently ask for an x such that $e^x - 16x = 0$. Let's 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's zoom in on $f(x)$ on these two ranges separately.



Let's say we're interested in finding the x between 3.5 and 4.5. What can we do?

Linear Search: We can't quite solve things exactly. So let us approximate it. 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. That's quite a lot of values to try: $(4.5 - 3.5)/0.01 = 100$. But it's still manageable. In general, if we are looking in the range $[a, b]$ and we're hoping to have precision within $\pm\varepsilon$, then the total time required will be $O((b - a)/\varepsilon)$.

Binary Search: One characteristic of $f(x)$ on the range 3.5 and 4.5 is that $f(3.5) < 0$ and $f(4.5) > 0$. Furthermore, the function f is increasing on that range. This means on that range, if $x < x'$, then $f(x) < f(x')$. Hence, we could apply binary search to look for an x where $f(x) = 0$. This is a total of 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.

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. We'll study a lot more of such algorithms, including much more sophisticated ones, in a numerical methods course.

3.2 The Stuttering-Substring Problem

Let's consider two strings A and B. As we have seen before, 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 some examples: "cat" is a substring "excavate", but "vet" isn't a substring of "excavate"

It is a simple exercise (which you have done in Intro to Programming) to check if A is a substring of B. This can actually be done in $O(\texttt{len}(A) + \texttt{len}(B))$, so let's assume that we have a function `isSubstr(A, B)` that returns True or False with the claimed running time bound.

Exercise: Implement and analyze `isSubstr(A, B)` that runs in $O(\texttt{len}(A) + \texttt{len}(B))$.

Putting that aside for now, let's say 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 $A^{(3)}$ isn't. 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. Let's 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 first attempt is

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

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. Therefore, we can readily apply binary search. 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 you to figure out (in Assignment 3). 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 $O(\log(m/n))$ "probes." Now each probe involves a call to `isSubstr` but the length of $A^{(k)}$ and B each never exceed $m + n$, so the total running time is $O((n + m)\log(m/n))$ in the worst case.