1

Binary Search

Thomas Seidl and Jost Enderle

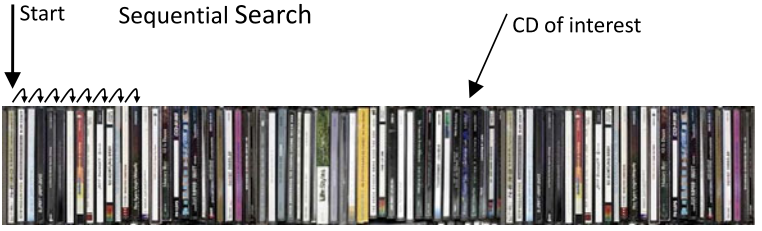
RWTH Aachen University, Aachen, Germany

Where has the new Nelly CD gone? I guess my big sister Linda with her craze for order has placed it in the CD rack once again. I've told her a thousand times to leave my new CDs outside. Now I'll have to check again all 500 CDs in the rack one by one. It'll take ages to go through all of them!

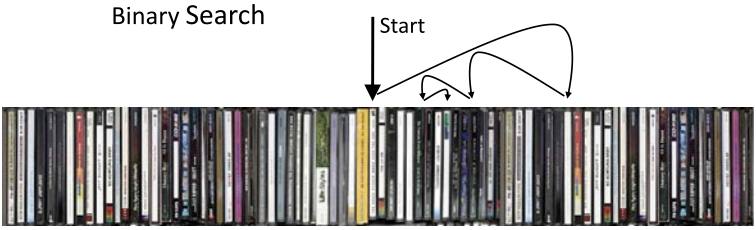
Okay, if I'm lucky, I might possibly find the CD sooner and won't have to check each cover. But in the worst case, Linda has lent the CD to her friend again: then I'll have to go through all of them and listen to the radio in the end.

Aaliyah, AC/DC, Alicia Keys ... hmmm, Linda seems to have sorted the CDs by artist. Using that, finding my Nelly CD should be easier. I'll try right in the middle. "Kelly Family"; must have been too far to the left; I have to search further to the right. "Rachmaninov"; now that's too far to the right, let's shift a bit further to the left ... "Lionel Hampton." Just a little bit to the right ... "Nancy Sinatra" ... "Nelly"!

Well, that was quick! With the sorting, jumping back and forth a few times will suffice to find the CD! Even if the CD hadn't been in the rack, this would have been noticed quickly. But when we have, say, 10,000 CDs, I'll probably have to jump back and forth a few hundred times to examine the CDs. I wonder if one could calculate that.



B. Vöcking et al. (eds.), *Algorithms Unplugged*,
DOI 10.1007/978-3-642-15328-0_1,



Sequential Search

Linda has been studying computer science since last year; there should be some documents of hers lying around providing useful information. Let's have a look ... "search algorithms" may be the right chapter. It describes how to search for an element of a given set (here, CDs) by some key value (here, artist). What I tried first seems to be called "sequential search" or "linear search." As already expected, half of the elements have to be scanned on average to find the searched key value. The number of search steps increases proportionally to the number of elements, i.e., doubling the elements results in double search time.

Binary Search

My second search technique seems also to have a special name, "binary search." For a given search key and a sorted list of elements, the search starts with the middle element whose key is compared with the search key. If the searched element is found in this step, the search is over. Otherwise, the same procedure is performed repeatedly for either the left or the right half of the elements, respectively, depending on whether the checked key is greater or less than the search key. The search ends when the element is found or when a bisection of the search space isn't possible anymore (i.e., we've reached the position where the element should be). My sister's documents contain the corresponding program code.

In this code, A denotes an "array," that is, a list of data with numbered elements, just like the CD positions in the rack. For example, the fifth element in such an array is denoted by A[5]. So, if our rack holds 500 CDs and we're searching for the key "Nelly," we have to call BinarySearch (rack, "Nelly", 1, 500) to find the position of the searched CD. During the execution of the program, left is assigned 251 at first, and then right is assigned 375, and so on.

The function BinarySearch returns the position of "key" in array "A" between "left" and "right"

```
1  function BinarySearch (A, key, left, right)
2  while left ≤ right do
3      middle := (left + right)/2      // find the middle, round the result
4      if A[middle] = key then return middle
5      if A[middle] > key then right := middle − 1
6      if A[middle] < key then left := middle + 1
7  endwhile
8  return not found
```

Recursive Implementation

In Linda's documents, there is also a second algorithm for binary search. But why do we need different algorithms for the same function? They say the second algorithm uses *recursion*; what's that again?

I have to look it up ...: "A recursive function is a function that is defined by itself or that calls itself." The *sum function* is given as an example, which is defined as follows:

$$\text{sum}(n) = 1 + 2 + \cdots + n.$$

That means, the first n natural numbers are added; so, for $n = 4$ we get:

$$\text{sum}(4) = 1 + 2 + 3 + 4 = 10.$$

If we want to calculate the result of the sum function for a certain n and we already know the result for $n - 1$, n just has to be added to this result:

$$\text{sum}(n) = \text{sum}(n - 1) + n.$$

Such a definition is called a *recursion step*. In order to calculate the sum function for some n in this way, we still need the base case for the smallest n:

$$\text{sum}(1) = 1.$$

Using these definitions, we are now able to calculate the sum function for some n:

$$\begin{aligned}
\text{sum}(4) &= \text{sum}(3) + 4 \\
&= (\text{sum}(2) + 3) + 4 \\
&= ((\text{sum}(1) + 2) + 3) + 4 \\
&= ((1 + 2) + 3) + 4 \\
&= 10.
\end{aligned}$$

The same holds true for a recursive definition of binary search: Instead of executing the loop repeatedly (*iterative* implementation), the function calls itself in the function body:

The function BinSearchRecursive returns the position of "key" in array "A" between "left" and "right"

```
1  function BinSearchRecursive (A, key, left, right)
2  if left > right return not found
3  middle := (left + right)/2      // find the middle, round the result
4  if A[middle] = key then return middle
5  if A[middle] > key then
6      return BinSearchRecursive (A, key, left, middle − 1)
7  if A[middle] < key then
8      return BinSearchRecursive (A, key, middle + 1, right)
```

As before, A is the array to be searched through, "key" is the key to be searched for, and "left" and "right" are the left and right borders of the searched region in A, respectively. If the element "Nelly" has to be found in an array "rack" containing 500 elements, we have the same function call, BinSearchRecursive (rack, "Nelly", 1, 500). However, instead of pushing the borders towards each other iteratively by a program loop, the BinSearchRecursive function will be called recursively with properly adapted borders. So we get the following sequence of calls:

BinSearchRecursive (rack, "Nelly", 1, 500)
BinSearchRecursive (rack, "Nelly", 251, 500)
BinSearchRecursive (rack, "Nelly", 251, 374)
BinSearchRecursive (rack, "Nelly", 313, 374)
BinSearchRecursive (rack, "Nelly", 344, 374)
...

Number of Search Steps

Now the question remains, how many search steps do we actually have to perform to find the right element? If we're lucky, we'll find the element with the first step; if the searched element doesn't exist, we have to keep jumping until we have reached the position where the element should be. So, we have to consider how often the list of elements can be cut in half or, conversely, how many elements can we check with a certain number of comparisons. If we presume the searched element to be contained in the list, we can check two elements with one comparison, four elements with two comparisons, and eight elements with only three comparisons. So, with k comparisons we are able to check $2 \cdot 2 \cdot \cdots \cdot 2$ (k times) $= 2^k$ elements. This will result in ten comparisons for 1,024 elements, 20 comparisons for over a million elements,

and 30 comparisons for over a billion elements! We will need an additional check if the searched element is not contained in the list. In order to calculate the converse, i.e., to determine the number of comparisons necessary for a certain number of elements, one has to use the inverse function of the power of 2. This function is called the "base 2 logarithm" and is denoted by $\log_2$. In general, the following holds true for logarithms:

$$\text{If } a = b^x, \text{ then } x = \log_b a. \tag{1.1}$$

For the base 2 logarithm, we have $b = 2$:

$$\begin{aligned}
2^0 &= 1, & \log_2 1 &= 0\\
2^1 &= 2, & \log_2 2 &= 1\\
2^2 &= 4, & \log_2 4 &= 2\\
2^3 &= 8, & \log_2 8 &= 3\\
&\vdots & &\vdots\\
2^{10} &= 1{,}024, & \log_2 1{,}024 &= 10\\
&\vdots & &\vdots\\
2^{13} &= 8{,}192, & \log_2 8{,}192 &= 13\\
2^{14} &= 16{,}384, & \log_2 16{,}384 &= 14\\
&\vdots & &\vdots\\
2^{20} &= 1{,}048{,}576, & \log_2 1{,}048{,}576 &= 20.
\end{aligned}$$

So, if $2^k = N$ elements can be checked with k comparisons, $\log_2 N = k$ comparisons are needed for N elements. If our rack contains 10,000 CDs, we have $\log_2 10{,}000 \approx 13.29$. As there are no "half comparisons," we get 14 comparisons! In order to further reduce the number of search steps of a binary search, one can try to guess more precisely where the searched key may be located within the currently inspected region (instead of just using the middle element). For example, if we are searching in our sorted CD rack for an artist's name whose initial is close to the beginning of the alphabet, e.g., "Eminem," it's a good idea to start searching in the front part of the rack. Accordingly, a search for "Roy Black" should start at a position in the rear part. For a further improvement of the search, one should take into account that some initials (e.g., D and S) are much more common than others (e.g., X and Y).

Guessing Games

This evening I'll put Linda to the test and let her guess a number between 1 and 1,000. If she didn't sleep during the lectures, she shouldn't need more than ten "yes/no" questions for that. (The figure below shows a possible approach for guessing a number between 1 and 16 with just four questions.)

In order to avoid asking the same boring question "Is the number greater/less than ...?" over and over again, one can throw in something like "Is the number even/odd?". This will also exclude one half of the remaining possibilities. Another question could be "Is the number of tens/hundreds even/odd?" which would also result in halving the search space (approximately). However, when all digits have been checked, we have to return to our regular halving method (while taking into account the numbers that have already been excluded).

The procedure becomes even easier if we use the binary representation of the number. While numbers in the decimal system are represented as sums of multiples of powers of 10, e.g.,

$$\begin{aligned} 107 &= \mathbf{1} \cdot 10^2 + \mathbf{0} \cdot 10^1 + \mathbf{7} \cdot 10^0 \\ &= \mathbf{1} \cdot 100 + \mathbf{0} \cdot 10 + \mathbf{7} \cdot 1, \end{aligned}$$

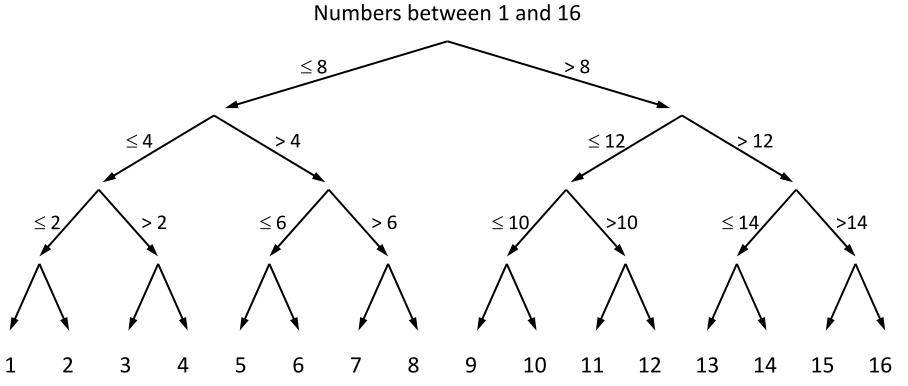
numbers in the binary system are represented as sums of multiples of powers of 2:

$$\begin{aligned} 107 &= \mathbf{1} \cdot 2^6 + \mathbf{1} \cdot 2^5 + \mathbf{0} \cdot 2^4 + \mathbf{1} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{1} \cdot 2^0 \\ &= \mathbf{1} \cdot 64 + \mathbf{1} \cdot 32 + \mathbf{0} \cdot 16 + \mathbf{1} \cdot 8 + \mathbf{0} \cdot 4 + \mathbf{1} \cdot 2 + \mathbf{1} \cdot 1. \end{aligned}$$

So the binary representation of 107 is 1101011. To guess a number using the binary representation, it is sufficient to know how many binary digits the number can have at most. The number of binary digits can easily be calculated using the base 2 logarithm. For example, if a number between 1 and 1,000 has to be guessed, one would calculate that

$$\log_2 1000 \approx 9.97 \text{ (round up!)},$$

i.e., ten digits, are required. Using that, ten questions will suffice: "Does the first binary digit equal 1?", "Does the second binary digit equal 1?", "Does the third binary digit equal 1?", and so on. After that, all digits of the binary representation are known and have to be converted into the decimal system; a pocket calculator will do this for us.



Further Reading

1. Donald Knuth: *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*. 3rd edition, 1997.
   This book describes the binary search on pages 409–426.
2. Implementation of the binary search algorithm:
   http://en.wikipedia.org/wiki/Binary_search
3. Binary search in the Java SDK:
   http://download.oracle.com/javase/6/docs/api/java/util/Arrays.html#binarySearch(long[],long)
4. To perform a binary search on a set of elements, these elements have to be in sorted order. The following chapters explain how to sort the elements quickly:
   - Chap. 2 (Insertion Sort)
   - Chap. 3 (Fast Sorting Algorithms)
   - Chap. 4 (Parallel Sorting)