Code it

Theory workshop

Efficient search tools

Get the most out of binary search algorithms with a basic halving strategy

In this issue...

▶ WHAT'S COVERED

Binary search is possibly the most well-known and simplest algorithm in the computer science lexicon. It can be said to be one of those algorithms you learn at your at an early age and yet it has been shown to be difficult to get right: the **edge cases** can trip you up badly.

Nugget

Binary search can help with the game of guessing a number less than one million using 20 questions. You guess the halfway point of your current range. If it's equal, great, you're done. If it's less (or greater) then you modify your range accordingly and continue the algorithm.

In essence, by following this algorithm, you are guessing the digits of the binary representation of the number the other has selected from the most to the least significant. Because one million is less than two to the twentieth power, you'll only need at most 20 questions to determine all the binary digits and therefore the actual number.

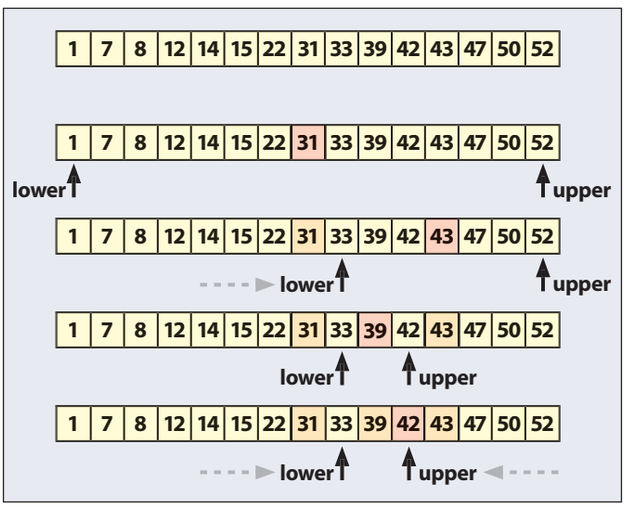
Consider this problem: you are given a phone book and you have to find the name associated with a particular phone number. If you were inclined to even start this task (and, believe me, I wouldn't), the only way you could do it would be to start at the beginning of the book and read each phone number until you find the one requested, at which point you'll know the name of the person who has that phone number. This simple algorithm is called sequential search.

The opposite problem though, finding out the phone number for a given name, we can easily solve. In general, we'd flip quickly through the book until we find the first letter of the surname, and then hone into the name we want, using the surnames printed at the top of the pages (in essence, narrowing in on subsequent letters in the surname) until we reach the right page. At this point what we'd probably do is to look at the middle name in that page. If it's not the one we want, we can immediately state which half of the page contains our requested name, and just concentrate on that half. We'd essentially do the same thing: look at the middle name in the half of the page, and then know which half of the half page we need to concentrate on. Eventually we'd zero in on the surname we want. I even find that once I've roughly found out where the name is, I'll sequentially search through that small part of the page for the one I want.

Binary search

That process, once formalized and implemented for a computer, is known as binary search.

Binary search is one of those algorithms that seems to have been around as long as Euclid's algorithm for finding the Greatest Common Divisor of two numbers,



▲ **The steps taken in an example of a successful search for 42. Note how the upper and lower bounds zero in on it.**

and my example of the phone book seems to prove it. In fact it was only formally stated in 1946 by John Mauchly (1907-1980), but a complete implementation was produced in 1960 by D.H. Lehmer (1905-1991), the number theorist also famous for primality testing and generating random numbers.

Problems with bugs

In fact, for some reason, binary search is one of those algorithms whose implementation always seems to cause problems. The main bugs that surface seem to be off-by-one errors or even an infinite loop. Let's implement a bug-free version in C♯.

First we must define the algorithm itself. Suppose we have an array of elements that are in sorted order and an item that we are trying to find in this array. If the array is empty, we can immediately return and report that the item can't be found. Otherwise, we find the middle element in the array. There are three possibilities: this middle element is equal to the item we're looking for, it's larger, or it's smaller. If it's equal, great, we can exit the algorithm and report the index of the array where we found the item. If the middle element is smaller, the item we're looking for must be in the half of the array that's greater than this middle element; otherwise it's in the half of the array that's smaller than the middle element. We now apply the same algorithm again to the half array, whichever half we determined the item to be in.

In implementing this algorithm, rather than copying the initial array into smaller and smaller arrays, we make use of two indexes, traditionally called lower and upper, to determine the bounds of the sub-array we're working on. We initially set lower to 0 and upper to the index of the last element in the array.

To be pedantic and not use exits from loops or from routines, we make use of a Boolean variable, still looking to determine when we should exit the loop (that is, when we determine the item is not present or the index of the element in the array if it is). We use an integer variable foundAtIndex for ▶

▶ the index at which we find our item; with the assumption that if we don't find it, it'll be -1. It is this value we'll be returning.

```
int BinarySearch(int[] array, int item)
{
  int lower = 0;
  int upper = array.Length - 1;
  int middle;
  int foundAtIndex = -1; // this
means "not found"
  bool stillLooking = true;
  while (stillLooking) {
    ... code that searches ...
  }
  return foundAtIndex;
}
```

We now enter the implementation of the algorithm proper. The first thing to check is if the array is empty. When using our index variables, we have an empty array when lower is greater than upper. A common error is to assume that the array is empty when these variables are equal, but when they are, the array has one element.

So suppose lower is less than or equal to upper. We can calculate the middle index, middle, to be (lower + upper)/2. (The division operator is assumed here to be integer division, so that 6/2 is 3 and 7/2 is also 3).

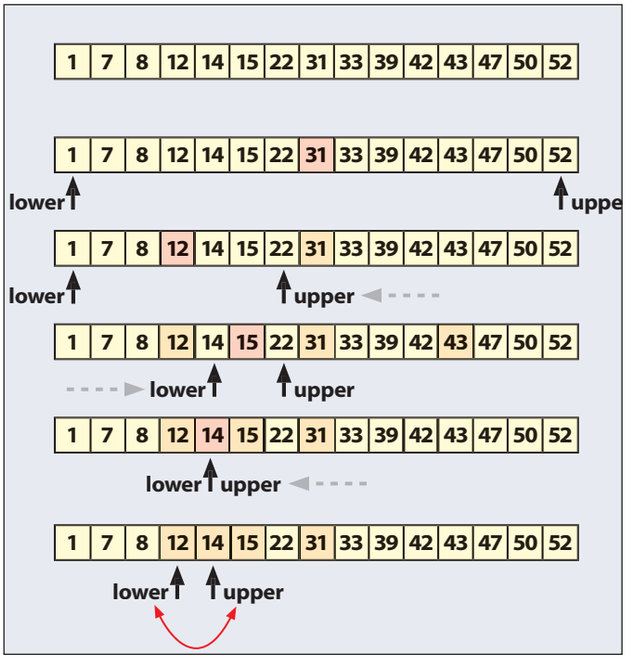
We can now check to see whether the element at middle is less than, equal to, or greater than the item we're trying to find. If equal, we return the value of middle. If it's less than the item we're searching for, then we set lower to middle+1, and go around to try again. In adding one to the value of middle, we are deliberately excluding that element from the sub-array that we're going to search next, and quite rightly too since this is the prime cause of inadvertent infinite loops.

If the middle element is greater than the item we're searching for, we set upper to middle-1, and go around and try again. And again, we are deliberately excluding the middle element from the subsequent sub-array.

```
if (lower > upper) {
stillLooking = false;
}
else {
middle = (lower + upper) / 2;
if (array[middle] == item) {
stillLooking = false;
foundAtIndex = middle;
}
else if (array[middle] < item) {
lower = middle + 1;
}
else {
upper = middle - 1;
}
}
```

Although this code will work, it is a little verbose, and there are ways in which it can be made tighter. The important point to note is that it works, and any changes you make should be rigorously tested against the original.

The figures show the operation of the binary search algorithm in action. Figure 1 shows a successful search for the item 42 in a sorted array. Notice how the bounds of the range quickly close in on the element we're searching for. Figure 2, shows an unsuccessful search for the item 13 in the array. This time notice how the bounds cross over in the last step to indicate that the item is not present in the array.



▲ **In this example, the bounds cross in, but finally cross over – an unsuccessful binary search in action.**

Numerical analysis

Binary search is ubiquitous in programming. I show elsewhere in the callouts a couple of examples of its use. Another example from the field of numerical analysis is when you are trying to find the root of a mathematical function in x (that is, for which value of x does the function equal zero?). One way is to use the bisection method: find a value of x for which the function is negative, and one for which it is positive. Hence the value for x for which it is zero is between them. Find the middle value of x. If the function evaluates to zero we've found the answer, otherwise we can reduce one of the bounds for x to this middle value. Repeat until we find the answer.

Binary search is also used in several data structures, especially sorted binary trees and the B-trees used by database engines to index data. In fact, any time you hear the phrase 'divide-and-conquer' when talking about an algorithm, you're probably looking at a use for binary search. ■

Julian Bucknall is a program manager and CTO for companies from Turbo Power to Microsoft ***feedback@pcplus.co.uk****.*

Spotlight on… Bugs in standard binary

Although the code I've presented in the main article works, there is a subtle bug that appears with large numbers of elements. The bug is extremely subtle: the code shown has been accepted to be bug-free for a very long time and it was a complete surprise when Joshua Bloch, a Software Engineer at Google, published a blog post last year that revealed this subtle bug.

The problem lies with the statement that calculates the value of middle. It looks innocuous enough, and for the vast majority of values it's perfectly correct. The problem occurs when the numbers get too large.

The largest value that can be held by a 32-bit integer is just over two billion. Let's imagine that both upper and lower are about 1.5 billion. When we add them together prior to dividing by two, we'll overflow the range of an integer. To put it another way, we cannot hold the result in an integer variable. In fact, the overflowed sum will be negative, causing all kinds of bugs in the algorithm.

The solution is to use the following statement instead:

```
middle = lower + (upper – lower)
/ 2
```

This avoids the overflow. An error like this will often fail silently, throwing your maths when you least expect it. Always test your code,keeping both eyes peeled for anomalous results. ■

Nugget

Binary search turns up everywhere. Do you want to calculate the number of rightmost zero bits for an unsigned 8-bit integer? (So, for 10111000 the answer is three). Use binary search. Set a counter to zero. Check if the four rightmost bits are zero by ANDing with 0x0F. If the result is zero increment the counter by four and shift the number right by four. Now check if the two rightmost bits are zero by ANDing with 0x03. If the result is zero, increment the counter by two and shift the number left by two. Finally check if the rightmost bit is zero by ANDing with 0x01, and if the result is zero, increment the counter by one. The counter is now equal to the number of rightmost zeros. This binary search can be extended to 32-bit integers quite easily. ■