Binary Search

Binary search is one of the fundamental algorithms in computer science. In order to explore it, we'll first build up a theoretical backbone, then use that to implement the algorithm properly.

Finding a value in a sorted sequence

In its simplest form, binary search is used to quickly find a value in a sorted sequence (consider a sequence an ordinary array for now). We'll call the sought value the *target* value for clarity. Binary search maintains a contiguous subsequence of the starting sequence where the target value is surely located. This is called the *search space*. The search space is initially the entire sequence. At each step, the algorithm compares the median value in the search space to the target value. Based on the comparison and because the sequence is sorted, it can then eliminate half of the search space. By doing this repeatedly, it will eventually be left with a search space consisting of a single element, the target value.

For example, consider the following sequence of integers sorted in ascending order and say we are looking for the number 55:

```
0   5   13   19   22   41   55   68   72   81   98
```

We are interested in the location of the target value in the sequence so we will represent the search space as indices into the sequence. Initially, the search space contains indices 1 through 11. Since the search space is really an interval, it suffices to store just two numbers, the low and high indices. As described above, we now choose the median value, which is the value at index 6 (the midpoint between 1 and 11): this value is 41 and it is smaller than the target value. From this we conclude not only that the element at index 6 is not the target value, but also that no element at indices between 1 and 5 can be the target value, because all elements at these indices are smaller than 41, which is smaller than the target value. This brings the search space down to indices 7 through 11:

```
55   68   72   81   98
```

Proceeding in a similar fashion, we chop off the second half of the search space and are left with:

```
55  68
```

Depending on how we choose the median of an even number of elements we will either find 55 in the next step or chop off 68 to get a search space of only one element. Either way, we conclude that the index where the target value is located is 7.

If the target value was not present in the sequence, binary search would empty the search space entirely. This condition is easy to check and handle. Here is some code to go with the description:

```
binary_search(A, target)
{
   lo = 1, hi = size(A);
   while (lo <= hi)
   {
      mid = lo + (hi-lo)/2
      if A[mid] == target:
         return mid;
      else if (A[mid] < target)
         lo = mid+1;
      else
         hi = mid-1;

    // target was not found
    }
 }
```

Complexity

Since each comparison binary search uses halves the search space, we can assert and easily prove that binary search will never use more than (in big-oh notation) $O(lgN)$ comparisons to find the target value.

The logarithm is an awfully slowly growing function. In case you're not aware of just how efficient binary search is, consider looking up a name in a phone book containing a million names. Binary search lets you systematically find any given name using at most 21 comparisons. If you could manage a list containing all the people in the world sorted by name, you could find any person in less than 35 steps. This may not seem feasible or useful at the moment, but we'll soon fix that.

Note that this assumes that we have random access to the sequence. Trying to use binary search on a container such as a linked list makes little sense and it is better use a plain linear search instead.

The Discrete Binary Search

Here we start abstract binary search. A sequence (array) is really just a function which associates integers (indices) with the corresponding values. However, there is no reason to restrict our usage of binary search to tangible sequences. In fact, we can use the same algorithm described above on any monotonic function f whose domain is the set of integers. The only difference is that we replace an array lookup with a function evaluation: we are now looking for some x such that $f(x)$ is equal to the target value. The search space is now more formally a subinterval of the domain of the fun4ction, while the target value is an element of the codomain. The power of binary search begins to show now: not only do we need at most $O(logN)$ comparisons to find the target value, but we also do not need to evaluate the function more than that many times. Additionally, in this case we aren't restricted by practical quantities such as available memory, as was the case with arrays.

Taking it further: The Main Theorem

When you encounter a problem which you think could be solved by applying binary search, you need some way of proving it will work. I will now present another level of abstraction which will allow us to solve more problems, make proving binary search solutions very easy and also help implement them. This part is a tad formal, but don't get discouraged, it's not that bad.

Consider a predicate p defined over some ordered set S (the search space). The search space consists of candidate solutions to the problem. In this article, a predicate is a function which returns a boolean value, true or false (we'll also use yes and no as boolean values). We use the predicate to verify if a candidate solution is legal (does not violate some constraint) according to the definition of the problem.

What we can call the *main theorem* states that **binary search can be used if and only if for all x in S, $p(x)$ implies $p(y)$ for all $y > x$** . This property is what we use when we discard the second half of the search space. It is equivalent to saying that $\neg p(x)$ implies $\neg p(y)$ for all $y < x$ (the symbol $\neg$ denotes the logical not operator), which is what we use when we discard the first half of the search space. Prove the theorem as an exercise.

What actually is stated here is that, if you had a yes or no question (the predicate), getting a yes answer for some potential solution x means that you'd also get a yes answer for any element after x. Similarly, if you got a no answer, you'd get a no answer for any element before x. As a consequence, if you were to ask the question for each element in the search space (in order), you would get a series of no answers followed by a series of yes answers.

Careful readers may note that binary search can also be used when a predicate yields a series of yes answers followed by a series of no answers. This is true and complementing that predicate will satisfy the original condition. For simplicity we'll deal only with predicates described in the theorem.

If the condition in the main theorem is satisfied, we can use binary search to find the smallest legal solution, i.e. the smallest x for which $p(x)$ is true. The first part of devising a solution based on binary search is designing a predicate which can be evaluated and for which it makes sense to use binary search: we need to choose what the algorithm should find. We can have it find either the first x for which $p(x)$ is true or the last x for which $p(x)$ is false. The difference between the two is only slight, as you will see, but it is necessary to settle on one. For starters, let us seek the first yes answer (first option).

The second part is proving that binary search can be applied to the predicate. This is where we use the main theorem, verifying that the conditions laid out in the theorem are satisfied. The proof doesn't need to be overly mathematical, you just need to convince yourself that $p(x)$ implies $p(y)$ for all $y > x$ or that $\neg p(x)$ implies $\neg p(y)$ for all $y < x$. This can often be done by applying common sense in a sentence or two.

When the domain of the predicate are the integers, it suffices to prove that $p(x)$ implies $p(x + 1)$ or that $\neg p(x)$ implies $\neg p(x - 1)$, the rest then follows by induction.

These two parts are most often interleaved: when we think a problem can be solved by binary search, we aim to design the predicate so that it satisfies the condition in the main theorem.

One might wonder why we choose to use this abstraction rather than the simpler-looking algorithm we've used so far. This is because many problems can't be modeled as searching for a particular value, but it's possible to define and evaluate a predicate such as "*Is there an assignment which costs x or less?*", when we're looking for some sort

of assignment with the lowest cost. For example, the usual traveling salesman problem (TSP) looks for the *cheapest round-trip which visits every city exactly once*. Here, the target value is not defined as such, but we can define a predicate "*Is there a round-trip which costs x or less?*" and then apply binary search to find the smallest x which satisfies the predicate. This is called *reducing* the original problem to a decision (yes/no) problem. Unfortunately, we know of no way of efficiently evaluating this particular predicate and so the TSP problem isn't easily solved by binary search, but many optimization problems are.

Let us now convert the simple binary search on sorted arrays described in the introduction to this abstract definition. First, let's rephrase the problem as: "*Given an array A and a target value, return the index of the first element in A equal to or greater than the target value.*" Incidentally, this is more or less how lower_bound behaves in C++.

We want to find the index of the target value, thus any index into the array is a candidate solution. The search space S is the set of all candidate solutions, thus an interval containing all indices. Consider the predicate "*Is $A[x]$ greater than or equal to the target value*?". If we were to find the first x for which the predicate says yes, we'd get exactly what decided we were looking for in the previous paragraph.

The condition in the main theorem is satisfied because the array is sorted in ascending order: if $A[x]$ is greater than or equal to the target value, all elements after it are surely also greater than or equal to the target value.

If we take the sample sequence from before:

```
0    5    13    19    22    41    55    68    72    81    98
```

With the search space (indices):

```
1    2     3     4     5     6     7     8     9    10    11
```

And apply our predicate (with a target value of 55) to it we get:

```
N    N     N     N     N     N     Y     Y     Y     Y     Y
```

This is a series of *no* answers followed by a series of *yes* answers, as we were expecting. Notice how index 7 (where the target value is located) is the first for which the predicate yields yes, so this is what our binary search will find.

Example

Suppose, a number of workers need to examine a number of filing cabinets. The cabinets are not all of the same size and we are told for each cabinet how many folders it contains. We are asked to find an assignment such that each worker gets a sequential series of cabinets to go through and that it minimizes the maximum amount of folders that a worker would have to look through.

After getting familiar with the problem, a touch of creativity is required. Imagine that we have an unlimited number of workers at our disposal. The crucial observation is that, for some number `MAX`, we can calculate the minimum number of workers needed so that each worker has to examine no more than `MAX` folders (if this is possible). Let's see how we'd do that. Some worker needs to examine the first cabinet so we assign any worker to it. But, since the cabinets must be assigned in sequential order (a worker cannot examine cabinets `1` and `3` without examining `2` as well), it's always optimal to assign him to the second cabinet as well, if this does not take him over the limit we introduced `(MAX)`. If

it would take him over the limit, we conclude that his work is done and assign a new worker to the second cabinet. We proceed in a similar manner until all the cabinets have been assigned and assert that we've used the minimum number of workers possible, with the artificial limit we introduced. Note here that the number of workers is inversely proportional to `MAX`: the higher we set our limit, the fewer workers we will need.

Now, if you go back and carefully examine what we're asked for in the problem statement, you can see that we are really asked for the smallest `MAX` such that the number of workers required is less than or equal to the number of workers available. With that in mind, we're almost done, we just need to connect the dots and see how all of this fits in the frame we've laid out for solving problems using binary search.

With the problem rephrased to fit our needs better, we can now examine the predicate Can the workload be spread so that each worker has to examine no more than x folders, with the limited number of workers available? We can use the described greedy algorithm to efficiently evaluate this predicate for any x. This concludes the first part of building a binary search solution, we now just have to prove that the condition in the main theorem is satisfied. But observe that increasing x actually relaxes the limit on the maximum workload, so we can only need the same number of workers or fewer, not more. Thus, if the predicate says yes for some x, it will also say yes for all larger x.

The overall complexity of the solution is $O(n\ lgW)$, where W is the size of the search space. This is very fast.

As you see, we used a greedy algorithm to evaluate the predicate. In other problems, evaluating the predicate can come down to anything from a simple math expression to finding a maximum cardinality matching in a bipartite graph.

Conclusion

If you've gotten this far without giving up, you should be ready to solve anything that can be solved with binary search. Try to keep a few things in mind:

- Design a predicate which can be efficiently evaluated and so that binary search can be applied.
- Decide on what you're looking for and code so that the search space always contains that (if it exists).
- If the search space consists only of integers, test your algorithm on a two-element set to be sure it doesn't lock up.
- Verify that the lower and upper bounds are not overly constrained: it's usually better to relax them as long as it doesn't break the predicate.