Binary search algorithm

In computer science, a binary search or half-interval search algorithm locates the position of an item in a sorted array. Binary search works by comparing an input value to the middle element of the array. The comparison determines whether the element equals the input, less than the input or greater. When the element being compared to equals the input the search stops and typically returns the position of the element. If the element is not equal to the input then a comparison is made to determine whether the input is less than or greater than the element. Depending on which it is the algorithm then starts over but only searching the top or bottom subset of the array's elements. If the input is not located within the array the algorithm will usually output a unique value indicating this. Binary search algorithms typically halve the number of items to check with each successive iteration, thus locating the given item (or determining its absence) in logarithmic time. A binary search is a dichotomic divide and conquer search algorithm.

Overview

It is useful to find where an item is in a sorted array. For example, to search an array for contact information, with people's names, addresses, and telephone numbers sorted by name, binary search could be used to find out a few useful facts: whether the person's information is in the array, what the person's address is, and what the person's telephone number is.

Binary search will take far fewer comparisons than a linear search, but there are some downsides. Binary search can be slower than using a hash table. If items are changed, the array will have to be re-sorted so that binary search will work properly, which can take so much time that the savings from using binary search aren't worth it. If you can tell ahead of time that a few items are disproportionately likely to be sought, putting those items first and using a linear search could be much faster.

Examples

Number guessing game

This rather simple game begins something like "I'm thinking of an integer between forty and sixty inclusive, and to your guesses I'll respond 'High', 'Low', or 'Yes!' as might be the case." Supposing that \( N \) is the number of possible values (here, twenty-one as "inclusive" was stated), then at most \( \lceil \log_2 N \rceil \) questions are required to determine the number, since each question halves the search space. Note that one less question (iteration) is required than for the general algorithm, since the number is already constrained to be within a particular range.
Even if the number to guess can be arbitrarily large, in which case there is no upper bound $N$, the number can be found in at most $2 \left\lceil \log_2 k \right\rceil$ steps (where $k$ is the (unknown) selected number) by first finding an upper bound by repeated doubling. \footnote{citation needed} For example, if the number were 11, the following sequence of guesses could be used to find it: 1, 2, 4, 8, 16, 12, 10, 11

One could also extend the method to include negative numbers; for example the following guesses could be used to find $-13$: 0, $-1$, $-2$, $-4$, $-8$, $-16$, $-12$, $-14$, $-13$.

**Word lists**

People typically use a mixture of the binary search and interpolative search algorithms when searching a telephone book, after the initial guess we exploit the fact that the entries are sorted and can rapidly find the required entry. For example when searching for Smith, if Rogers and Thomas have been found, one can flip to a page about halfway between the previous guesses. If this shows Samson, it can be concluded that Smith is somewhere between the Samson and Thomas pages so these can be divided.

**Applications to complexity theory**

Even if we do not know a fixed range the number $k$ falls in, we can still determine its value by asking $2 \left\lceil \log_2 k \right\rceil$ simple yes/no questions of the form "Is $k$ greater than $x$?" for some number $x$. As a simple consequence of this, if you can answer the question "Is this integer property $k$ greater than a given value?" in some amount of time then you can find the value of that property in the same amount of time with an added factor of $\log_2 k$. This is called a reduction, and it is because of this kind of reduction that most complexity theorists concentrate on decision problems, algorithms that produce a simple yes/no answer.

For example, suppose we could answer "Does this $n \times n$ matrix have determinant larger than $k$?" in $O(n^2)$ time. Then, by using binary search, we could find the (ceiling of the) determinant itself in $O(n^2 \log d)$ time, where $d$ is the determinant; notice that $d$ is not the size of the input, but the size of the output.

**Performance**

With each test that fails to find a match at the probed position, the search is continued with one or other of the two sub-intervals, each at most half the size. More precisely, if the number of items, $N$, is odd then both sub-intervals will contain $(N-1)/2$ elements, while if $N$ is even then the two sub-intervals contain $N/2$ and $N/2$ elements.

If the original number of items is $N$ then after the first iteration there will be at most $N/2$ items remaining, then at most $N/4$ items, at most $N/8$ items, and so on. In the worst case, when the value is not in the list,
the algorithm must continue iterating until the span has been made empty; this will have taken at most \( \lfloor \log_2 N \rfloor + 1 \) iterations, where the \( \lfloor \cdot \rfloor \) notation denotes the **floor function** that rounds its argument down to an integer. This **worst case analysis** is tight: for any \( N \) there exists a query that takes exactly \( \lfloor \log_2 N \rfloor + 1 \) iterations. When compared to **linear search**, whose worst-case behaviour is \( N \) iterations, we see that binary search is substantially faster as \( N \) grows large. For example, to search a list of one million items takes as many as one million iterations with linear search, but never more than twenty iterations with binary search. However, a binary search can only be performed if the list is in sorted order.

### Average performance

\( \log_2(N) - 1 \) is about the expected number of probes in an average successful search, and the worst case is \( \log_2(N) \), just one more probe. If the list is empty, no probes at all are made. Thus binary search is a **logarithmic algorithm** and executes in \( O(\log N) \) time. In most cases it is considerably faster than a linear search. It can be implemented using **iteration**, or **recursion**. In some languages it is more elegantly expressed recursively; however, in some C-based languages tail recursion is not eliminated and the recursive version requires more stack space.

Binary search can interact poorly with the memory hierarchy (i.e. **caching**), because of its random-access nature. For in-memory searching, if the span to be searched is small, a linear search may have superior performance simply because it exhibits better locality of reference. For external searching, care must be taken or each of the first several probes will lead to a disk seek. A common method is to abandon binary searching for linear searching as soon as the size of the remaining span falls below a small value such as 8 or 16 or even more in recent computers. The exact value depends entirely on the machine running the algorithm.

Notice that for multiple searches **with a fixed value for** \( N \), then (with the appropriate regard for integer division), the first iteration always selects the middle element at \( N/2 \), and the second always selects either \( N/4 \) or \( 3N/4 \), and so on. Thus if the array's key values are in some sort of slow storage (on a disc file, in virtual memory, not in the cpu's on-chip memory), keeping those three keys in a local array for a special preliminary search will avoid accessing widely separated memory. Escalating to seven or fifteen such values will allow further levels at not much cost in storage. On the other hand, if the searches are frequent and not separated by much other activity, the computer's various storage control features will more or less automatically promote frequently-accessed elements into faster storage.

When multiple binary searches are to be performed for the same key in related lists, **fractional cascading** can be used to speed up successive searches after the first one.

### Inserting and deleting elements
The main disadvantage of sorted arrays is that inserting and deleting random elements executes in $O(N)$ time. This is not true in all cases however. The search algorithm can be changed to check the last element first, which allows loading a sorted list at $O(1)$ time per element.

Extensions

There is no particular requirement that the array being searched has the bounds $1$ to $N$. It is possible to search a specified range, elements $first$ to $last$ instead of $1$ to $N$. All that is necessary is that the initialisation of the bounds be $L:=first - 1$ and $R:=last + 1$, then all proceeds as before.

The elements of the list are not necessarily all unique. If one searches for a value that occurs multiple times in the list, the index returned will be of the first-encountered equal element, and this will not necessarily be that of the first, last, or middle element of the run of equal-key elements but will depend on the positions of the values. Modifying the list even in seemingly unrelated ways such as adding elements elsewhere in the list may change the result. To find all equal elements an upward and downward linear search can be carried out from the initial result, stopping each search when the element is no longer equal. Thus, e.g. in a table of cities sorted by country, we can find all cities in a given country.

Several algorithms closely related to or extending binary search exist. For instance, noisy binary search solves the same class of projects as regular binary search, with the added complexity that any given test can return a false value at random. (Usually, the number of such erroneous results are bounded in some way, either in the form of an average error rate, or in the total number of errors allowed per element in the search space.) Optimal algorithms for several classes of noisy binary search problems have been known since the late seventies, and more recently, optimal algorithms for noisy binary search in quantum computers (where several elements can be tested at the same time) have been discovered.

Variations

There are many, and they are easily confused. Also, binary search vs. selection sort is debatable.

Exclusive or inclusive bounds

The most significant differences are between the "exclusive" and "inclusive" forms of the bounds. In the "exclusive" bound form the span to be searched is $(L + 1)$ to $(R - 1)$, and this may seem clumsy when the span to be searched could be described in the "inclusive" form, as $L$ to $R$. Although the details differ the two forms are equivalent as can be seen by transforming one version into the other. The inclusive bound form may be attained by replacing all appearances of "$L$" by "$(L - 1)$" and "$R$" by "$(R + 1)$" then
rearranging. Thus, the initialisation of $L:=0$ becomes $(L-1):=0$ or $L:=1$, and $R:=N+1$ becomes $(R+1):=N+1$ or $R:=N$. So far so good, but note now that the changes to $L$ and $R$ are no longer simply transferring the value of $p$ to $L$ or $R$ as appropriate but now must be $(R+1):=p$ or $R:=p-1$, and $(L-1):=p$ or $L:=p+1$.

Thus, the gain of a simpler initialisation, done once, is lost by a more complex calculation, and which is done for every iteration. If that is not enough, the test for an empty span is more complex also, as compared to the simplicity of checking that the value of $p$ is zero. Nevertheless, the inclusive bound form is found in many publications, such as Donald Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition.

**Deferred detection of equality**

Because of the syntax difficulties discussed below, so that distinguishing the three states $<$, $=$, and $>$ would have to be done with two comparisons, it is possible to use just one comparison and at the end when the span is reduced to zero, equality can be tested for. The solution distinguishes only $<$ from $\geq$.

**Midpoint and width**

An entirely different variation involves abandoning the $L$ and $R$ pointers in favour of a current position $p$ and a width $w$ where at each iteration, $p$ is adjusted by $+$ or $-$ $w$ and $w$ is halved. Professor Knuth remarks "It is possible to do this, but only if extreme care is paid to the details" – Section 6.2.1, page 414 of The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition, outlines an algorithm, with the further remark "Simpler approaches are doomed to failure!"

**The algorithm**

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky… — Professor Donald Knuth

When Jon Bentley assigned it as a problem in a course for professional programmers, he found that an astounding ninety percent failed to code a binary search correctly after several hours of working on it, and another study shows that accurate code for it is only found in five out of twenty textbooks (Kruse, 1999). Furthermore, Bentley's own implementation of binary search, published in his 1986 book Programming Pearls, contains an error that remained undetected for over twenty years.
Divide in half

A fast way to search a sorted array is to use a binary search. The idea is to look at the element in the middle. If the key is equal to that, the search is finished. If the key is less than the middle element, do a binary search on the first half. If it's greater, do a binary search of the second half.

Performance

The advantage of a binary search over a linear search is astounding for large numbers. For an array of a million elements, binary search, O(log N), will find the target element with a worst case of only 20 comparisons. Linear search, O(N), on average will take 500,000 comparisons to find the element. Probably the only faster kind of search uses hashing, a topic that isn't covered in these notes.

This performance comes at a price - the array must be sorted first. Because sorting isn't a fast operation, it may not be worth the effort to sort when there are only a few searches.

Example

```c
int binarySearch(int sortedArray[], int first, int last, int key) {
    // function:
    //   Searches sortedArray[first]..sortedArray[last] for key.
    // returns: index of the matching element if it finds key,
    //         otherwise -(index where it could be inserted)-1.
    // parameters:
    //   sortedArray in  array of sorted (ascending) values.
    //   first, last in  lower and upper subscript bounds
    //   key         in  value to search for.
    // returns:
    //   index of key, or -insertion_position -1 if key is not
    //   in the array. This value can easily be
    //   transformed into the position to insert it.
    while (first <= last) {
        int mid = (first + last) / 2;  // compute mid point.
        if (key > sortedArray[mid])
            first = mid + 1;  // repeat search in top half.
    ```
else if (key < sortedArray[mid])
    last = mid - 1;  // repeat search in bottom half.
else
    return mid;     // found it. return position /////<

return -(first + 1);    // failed to find key

---

**C++ Note: Algorithms: Recursive Binary Search**

**Recursive Binary Search**

Iterative algorithms, ie those with a loop, can usually be easily rewritten to use recursive function calls instead of loops. This is almost always a bad idea because the iterative version is usually simpler, faster, and uses less memory.

Some problems, eg traversing a tree, are better solved recursively because the recursive solution is so clear (see Binary Tree Traversal). Binary search is really is better in the non-recursive form, but it is one of the more plausible algorithms to use as an illustration of recursion.

This recursive version checks to see if we're at the key (in which case it can return), otherwise it calls itself so solve a smaller problem, ie, either the upper or lower half of the array.

**Example**

```cpp
int rBinarySearch(int sortedArray[], int first, int last, int key) {
    // function:
    //   Searches sortedArray[first]..sortedArray[last] for key.
    // returns: index of the matching element if it finds key,
    //           otherwise -(index where it could be inserted)-1.
    // parameters:
    //   sortedArray in  array of sorted (ascending) values.
    //   first, last in  lower and upper subscript bounds
    //   key         in  value to search for.
    // returns:
    //   index of key, or -insertion_position -1
    //       if key is not in the array.
```
if (first <= last) {
    int mid = (first + last) / 2;  // compute mid point.
    if (key == sortedArray[mid])
        return mid;   // found it.
    else if (key < sortedArray[mid])
        // Call ourself for the lower part of the array
        return rBinarySearch(sortedArray, first, mid-1, key);
    else
        // Call ourself for the upper part of the array
        return rBinarySearch(sortedArray, mid+1, last, key);
}
return -(first + 1);    // failed to find key

Sources: [Wikipedia](https://en.wikipedia.org) and [C++ Note of Fred Swartz](http://www.cplusplus.com/reference/algorithms/binary-search/)