WIKIPEDIA

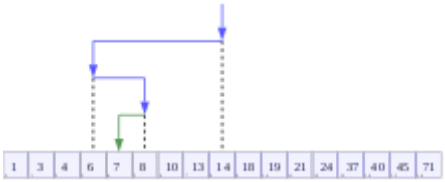
Binary search algorithm

In computer science, **binary search**, also known as **half-interval search**,[1] **logarithmic search**,[2] or **binary chop**,[3] is a search algorithm that finds the position of a target value within a sorted array.[4][5] Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Binary search runs in logarithmic time in the worst case, making $O(\log n)$ comparisons, where n is the number of elements in the array, the O is Big O notation, and **log** is the logarithm.[6] Binary search is faster than linear search except for small arrays. However, the array must be sorted first to be able to apply binary search. There are specialized data structures designed for fast searching, such as hash tables, that can be searched more efficiently than binary search. However, binary search can be used to solve a wider range of problems, such as finding the next-smallest or next-largest element in the array relative to the target even if it is absent from the array.

There are numerous variations of binary search. In particular, fractional cascading speeds up binary searches for the same value in multiple arrays. Fractional cascading efficiently solves a number of search problems in computational geometry and in numerous other fields. Exponential search extends binary search to unbounded lists. The binary search tree and B-tree data structures are based on binary search.

Binary search algorithm



Visualization of the binary search algorithm where 7 is the target value

Class	Search algorithm
Data structure	Array
Worst-case performance	$O(\log n)$
Best-case performance	$O(1)$
Average performance	$O(\log n)$
Worst-case space complexity	$O(1)$

Contents

Algorithm

Binary search works on sorted arrays. Binary search begins by comparing an element in the middle of the array with the target value. If the target value matches the element, its position in the array is returned. If the target value is less than the element, the search continues in the lower half of the array. If the target value is greater than the element, the search continues in the upper half of the array. By doing this, the algorithm eliminates the half in which the target value cannot lie in each iteration.[7]

Procedure

Given an array A of n elements with values or records $A_0, A_1, A_2, \ldots, A_{n-1}$ sorted such that $A_0 \leq A_1 \leq A_2 \leq \cdots \leq A_{n-1}$, and target value T, the following subroutine uses binary search to find the index of T in A.[7]

1. Set L to 0 and R to $n-1$.
2. If $L > R$, the search terminates as unsuccessful.
3. Set m (the position of the middle element) to the floor of $\frac{L+R}{2}$, which is the greatest integer less than or equal to $\frac{L+R}{2}$.
4. If $A_m < T$, set L to $m+1$ and go to step 2.
5. If $A_m > T$, set R to $m-1$ and go to step 2.
6. Now $A_m = T$, the search is done; return m.

This iterative procedure keeps track of the search boundaries with the two variables L and R. The procedure may be expressed in pseudocode as follows, where the variable names and types remain the same as above, `floor` is the floor function, and `unsuccessful` refers to a specific value that conveys the failure of the search.[7]

```
function binary_search(A, n, T):
    L := 0
    R := n - 1
    while L <= R:
        m := floor((L + R) / 2)
        if A[m] < T:
            L := m + 1
        else if A[m] > T:
            R := m - 1
        else:
            return m
    return unsuccessful
```

Alternatively, the algorithm may take the ceiling of $\frac{L+R}{2}$, or the least integer greater than or equal to $\frac{L+R}{2}$. This may change the result if the target value appears more than once in the array.

Alternative procedure

In the above procedure, the algorithm checks whether the middle element (m) is equal to the target (T) in every iteration. Some implementations leave out this check during each iteration. The algorithm would perform this check only when one element is left (when $L = R$). This results in a faster comparison loop, as one comparison is eliminated per iteration. However, it requires one more iteration on average.[8]

Hermann Bottenbruch published the first implementation to leave out this check in 1962.[8][9]

1. Set L to 0 and R to $n - 1$.
2. While $L \neq R$,
    1. Set m (the position of the middle element) to the ceiling of $\frac{L+R}{2}$, which is the least integer greater than or equal to $\frac{L+R}{2}$.
    2. If $A_m > T$, set R to $m - 1$.
    3. Else, $A_m \leq T$; set L to m.
3. Now $L = R$, the search is done. If $A_L = T$, return L. Otherwise, the search terminates as unsuccessful.

Where `ceil` is the ceiling function, the pseudocode for this version is:

```
function binary_search_alternative(A, n, T):
    L := 0
    R := n - 1
    while L != R:
        m := ceil((L + R) / 2)
        if A[m] > T:
            R := m - 1
        else:
            L := m
    if A[L] == T:
        return L
    return unsuccessful
```

Duplicate elements

The procedure may return any index whose element is equal to the target value, even if there are duplicate elements in the array. For example, if the array to be searched was $[1, 2, 3, 4, 4, 5, 6, 7]$ and the target was 4, then it would be correct for the algorithm to either return the 4th (index 3) or 5th (index 4) element. The regular procedure would return the 4th element (index 3) in this case. It does not always return the first duplicate (consider $[1, 2, 4, 4, 4, 5, 6, 7]$ which still returns the 4th element). However, it is sometimes necessary to find the leftmost element or the rightmost

element for a target value that is duplicated in the array. In the above example, the 4th element is the leftmost element of the value 4, while the 5th element is the rightmost element of the value 4. The alternative procedure above will always return the index of the rightmost element if such an element exists.[9]

Procedure for finding the leftmost element

To find the leftmost element, the following procedure can be used:[10]

1. Set L to 0 and R to n.
2. While $L < R$,
    1. Set m (the position of the middle element) to the floor of $\frac{L+R}{2}$, which is the greatest integer less than or equal to $\frac{L+R}{2}$.
    2. If $A_m < T$, set L to $m+1$.
    3. Else, $A_m \geq T$; set R to m.
3. Return L.

If $L < n$ and $A_L = T$, then A_L is the leftmost element that equals T. Even if T is not in the array, L is the rank of T in the array, or the number of elements in the array that are less than T.

Where `floor` is the floor function, the pseudocode for this version is:

```
function binary_search_leftmost(A, n, T):
    L := 0
    R := n
    while L < R:
        m := floor((L + R) / 2)
        if A[m] < T:
            L := m + 1
        else:
            R := m
    return L
```

Procedure for finding the rightmost element

To find the rightmost element, the following procedure can be used:[10]

1. Set L to 0 and R to n.
2. While $L < R$,
    1. Set m (the position of the middle element) to the floor of $\frac{L+R}{2}$, which is the greatest integer less than or equal to $\frac{L+R}{2}$.
    2. If $A_m > T$, set R to m.
    3. Else, $A_m \leq T$; set L to $m+1$.
3. Return $L-1$.

If $L > 0$ and $A_{L-1} = T$, then A_{L-1} is the rightmost element that equals T. Even if T is not in the array, $n-L$ is the number of elements in the array that are greater than T.

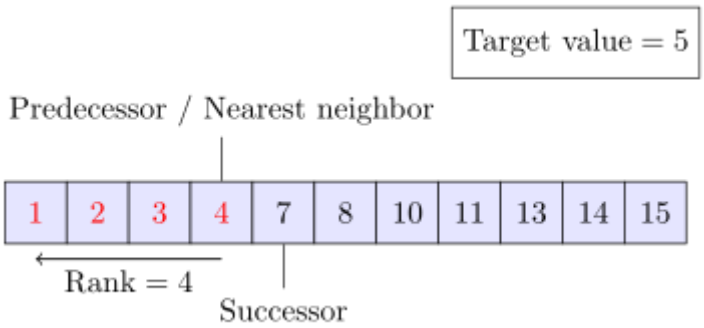
Where `floor` is the floor function, the pseudocode for this version is:

```
function binary_search_rightmost(A, n, T):
    L := 0
    R := n
    while L < R:
        m := floor((L + R) / 2)
        if A[m] > T:
            R := m
```

```
        else:
            L := m + 1
    return L - 1
```

Approximate matches

The above procedure only performs *exact* matches, finding the position of a target value. However, it is trivial to extend binary search to perform approximate matches because binary search operates on sorted arrays. For example, binary search can be used to compute, for a given value, its rank (the number of smaller elements), predecessor (next-smallest element), successor (next-largest element), and nearest neighbor. Range queries seeking the number of elements between two values can be performed with two rank queries.[11]

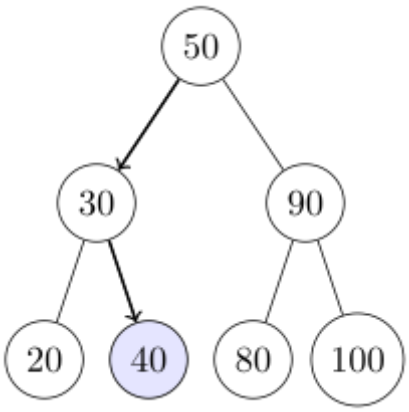


Binary search can be adapted to compute approximate matches. In the example above, the rank, predecessor, successor, and nearest neighbor are shown for the target value 5, which is not in the array.

- Rank queries can be performed with the procedure for finding the leftmost element. The number of elements *less than* the target value is returned by the procedure.[11]
- Predecessor queries can be performed with rank queries. If the rank of the target value is r, its predecessor is $r - 1$.[12]
- For successor queries, the procedure for finding the rightmost element can be used. If the result of running the procedure for the target value is r, then the successor of the target value is $r + 1$.[12]
- The nearest neighbor of the target value is either its predecessor or successor, whichever is closer.
- Range queries are also straightforward.[12] Once the ranks of the two values are known, the number of elements greater than or equal to the first value and less than the second is the difference of the two ranks. This count can be adjusted up or down by one according to whether the endpoints of the range should be considered to be part of the range and whether the array contains entries matching those endpoints.[13]

Performance

In terms of the number of comparisons, the performance of binary search can be analyzed by viewing the run of the procedure on a binary tree. The root node of the tree is the middle element of the array. The middle element of the lower half is the left child node of the root, and the middle element of the upper half is the right child node of the root. The rest of the tree is built in a similar fashion. Starting from the root node, the left or right subtrees are traversed depending on whether the target value is less or more than the node under consideration.[6][14]



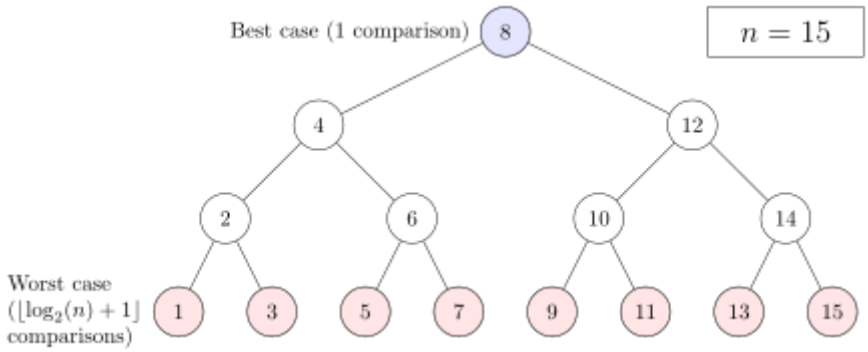
A tree representing binary search. The array being searched here is $[20, 30, 40, 50, 80, 90, 100]$, and the target value is 40.

In the worst case, binary search makes $\lfloor \log_2(n) + 1 \rfloor$ iterations of the comparison loop, where the $\lfloor \rfloor$ notation denotes the floor function that yields the greatest integer less than or equal to the argument, and $\log_2$ is the binary logarithm. This is because the worst case is reached when the search reaches the deepest level of the tree, and there are always $\lfloor \log_2(n) + 1 \rfloor$ levels in the tree for any binary search.

The worst case may also be reached when the target element is not in the array. If n is one less than a power of two, then this is always the case. Otherwise, the search may perform $\lfloor \log_2(n) + 1 \rfloor$ iterations if the search reaches the deepest level of the tree.

However, it may make $\lfloor \log_2(n) \rfloor$ iterations, which is one less than the worst case, if the search ends at the second-deepest level of the tree.[15]

On average, assuming that each element is equally likely to be searched, binary search makes



The worst case is reached when the search reaches the deepest level of the tree, while the best case is reached when the target value is the middle element.

$\lfloor \log_2(n) \rfloor + 1 - (2^{\lfloor \log_2(n) \rfloor + 1} - \lfloor \log_2(n) \rfloor - 2)/n$ iterations when the target element is in the array. This is approximately equal to $\log_2(n) - 1$ iterations. When the target element is not in the array, binary search makes $\lfloor \log_2(n) \rfloor + 2 - 2^{\lfloor \log_2(n) \rfloor + 1}/(n+1)$ iterations on average, assuming that the range between and outside elements is equally likely to be searched.[14]

In the best case, where the target value is the middle element of the array, its position is returned after one iteration.[16]

In terms of iterations, no search algorithm that works only by comparing elements can exhibit better average and worst-case performance than binary search. The comparison tree representing binary search has the fewest levels possible as every level above the lowest level of the tree is filled completely.[a] Otherwise, the search algorithm can eliminate few elements in an iteration, increasing the number of iterations required in the average and worst case. This is the case for other search algorithms based on comparisons, as while they may work faster on some target values, the average performance over *all* elements is worse than binary search. By dividing the array in half, binary search ensures that the size of both subarrays are as similar as possible.[14]

Space complexity

Binary search requires three pointers to elements, which may be array indices or pointers to memory locations, regardless of the size of the array. However, it requires at least $\lceil \log_2(n) \rceil$ bits to encode a pointer to an element of an array with n elements.[17] Therefore, the space complexity of binary search is $O(\log n)$. In addition, it takes $O(n)$ space to store the array.

Derivation of average case

The average number of iterations performed by binary search depends on the probability of each element being searched. The average case is different for successful searches and unsuccessful searches. It will be assumed that each element is equally likely to be searched for successful searches. For unsuccessful searches, it will be assumed that the intervals between and outside elements are equally likely to be searched. The average case for successful searches is the number of iterations required to search every element exactly once, divided by n, the number of elements. The average case for unsuccessful searches is the number of iterations required to search an element within every interval exactly once, divided by the $n + 1$ intervals.[14]

Successful searches

In the binary tree representation, a successful search can be represented by a path from the root to the target node, called an *internal path*. The length of a path is the number of edges (connections between nodes) that the path passes through. The number of iterations performed by a search, given that the corresponding path has length l, is $l+1$ counting the initial iteration. The *internal path length* is the sum of the lengths of all unique internal paths. Since there is only one path from the root to any single node, each internal path represents a search for a specific element. If there are n elements, which is a positive integer, and the internal path length is $I(n)$, then the average number of iterations for a successful search $T(n) = 1 + \frac{I(n)}{n}$, with the one iteration added to count the initial iteration.[14]

Since binary search is the optimal algorithm for searching with comparisons, this problem is reduced to calculating the minimum internal path length of all binary trees with n nodes, which is equal to:[18]

$$I(n) = \sum_{k=1}^{n} \lfloor \log_2(k) \rfloor$$

For example, in a 7-element array, the root requires one iteration, the two elements below the root require two iterations, and the four elements below require three iterations. In this case, the internal path length is:[18]

$$\sum_{k=1}^{7} \lfloor \log_2(k) \rfloor = 0 + 2(1) + 4(2) = 2 + 8 = 10$$

The average number of iterations would be $1 + \frac{10}{7} = 2\frac{3}{7}$ based on the equation for the average case. The sum for $I(n)$ can be simplified to:[14]

$$I(n) = \sum_{k=1}^{n} \lfloor \log_2(k) \rfloor = (n+1) \lfloor \log_2(n+1) \rfloor - 2^{\lfloor \log_2(n+1) \rfloor + 1} + 2$$

Substituting the equation for $I(n)$ into the equation for $T(n)$:[14]

$$T(n) = 1 + \frac{(n+1) \lfloor \log_2(n+1) \rfloor - 2^{\lfloor \log_2(n+1) \rfloor + 1} + 2}{n} = \lfloor \log_2(n) \rfloor + 1 - (2^{\lfloor \log_2(n) \rfloor + 1} - \lfloor \log_2(n) \rfloor - 2)/n$$

For integer n, this is equivalent to the equation for the average case on a successful search specified above.

Unsuccessful searches

Unsuccessful searches can be represented by augmenting the tree with *external nodes*, which forms an *extended binary tree*. If an internal node, or a node present in the tree, has fewer than two child nodes, then additional child nodes, called external nodes, are added so that each internal node has two children. By doing so, an unsuccessful search can be represented as a path to an external node, whose parent is the single element that remains during the last iteration. An *external path* is a path from the root to an external node. The *external path length* is the sum of the lengths of all unique external paths. If there are n elements, which is a positive integer, and the external path length is $E(n)$, then the average number of iterations for an unsuccessful search $T'(n) = \frac{E(n)}{n+1}$, with the one iteration added to count the initial iteration. The external path length is divided by $n+1$ instead of n because there are $n+1$ external paths, representing the intervals between and outside the elements of the array.[14]

This problem can similarly be reduced to determining the minimum external path length of all binary trees with n nodes. For all binary trees, the external path length is equal to the internal path length plus $2n$.[18] Substituting the equation for $I(n)$:[14]

$$E(n) = I(n) + 2n = \left[(n+1) \lfloor \log_2(n+1) \rfloor - 2^{\lfloor \log_2(n+1) \rfloor + 1} + 2\right] + 2n = (n+1)(\lfloor \log_2(n) \rfloor + 2) - 2^{\lfloor \log_2(n) \rfloor + 1}$$

Substituting the equation for $E(n)$ into the equation for $T'(n)$, the average case for unsuccessful searches can be determined:[14]

$$T'(n) = \frac{(n+1)(\lfloor \log_2(n) \rfloor + 2) - 2^{\lfloor \log_2(n) \rfloor + 1}}{(n+1)} = \lfloor \log_2(n) \rfloor + 2 - 2^{\lfloor \log_2(n) \rfloor + 1}/(n+1)$$

Performance of alternative procedure

Each iteration of the binary search procedure defined above makes one or two comparisons, checking if the middle element is equal to the target in each iteration. Assuming that each element is equally likely to be searched, each iteration makes 1.5 comparisons on average. A variation of the algorithm checks whether the middle element is equal to the target at the end of the search. On average, this eliminates half a comparison from each iteration. This slightly cuts the time taken per iteration on most computers. However, it guarantees that the search takes the maximum number of iterations, on average adding one iteration to the search. Because the comparison loop is performed only $\lfloor \log_2(n) + 1 \rfloor$ times in the worst case, the slight increase in efficiency per iteration does not compensate for the extra iteration for all but very large n.[b][19][20]

Running time and cache use

In analyzing the performance of binary search, another consideration is the time required to compare two elements. For integers and strings, the time required increases linearly as the encoding length (usually the number of bits) of the elements increase. For example, comparing a pair of 64-bit unsigned integers would require comparing up to double the bits as comparing a pair of 32-bit unsigned integers. The worst case is achieved when the integers are equal. This can be significant when the encoding lengths of the elements are large, such as with large integer types or long strings, which makes comparing elements expensive. Furthermore, comparing floating-point values (the most common digital representation of real numbers) is often more expensive than comparing integers or short strings.

On most computer architectures, the processor has a hardware cache separate from RAM. Since they are located within the processor itself, caches are much faster to access but usually store much less data than RAM. Therefore, most processors store memory locations that have been accessed recently, along with memory locations close to it. For example, when an array element is accessed, the element itself may be stored along with the elements that are stored close to it in RAM, making it faster to sequentially access array elements that are close in index to each other (locality of reference). On a sorted array, binary search can jump to distant memory locations if the array is large, unlike algorithms (such as linear search and linear probing in hash tables) which access elements in sequence. This adds slightly to the running time of binary search for large arrays on most systems.[21]

Binary search versus other schemes

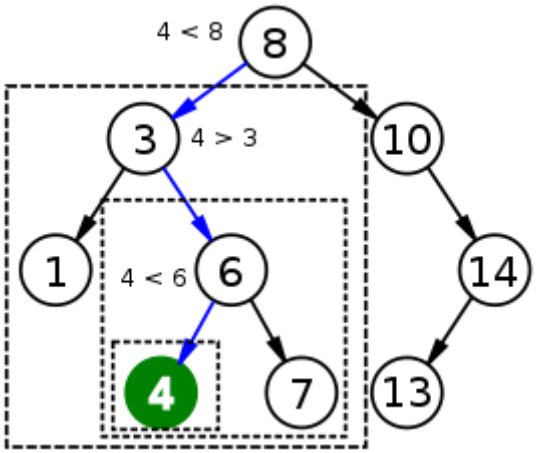
Sorted arrays with binary search are a very inefficient solution when insertion and deletion operations are interleaved with retrieval, taking $O(n)$ time for each such operation. In addition, sorted arrays can complicate memory use especially when elements are often inserted into the array.[22] There are other data structures that support much more efficient insertion and deletion. Binary search can be used to perform exact matching and set membership (determining whether a target value is in a collection of values). There are data structures that support faster exact matching and set membership. However, unlike many other searching schemes, binary search can be used for efficient approximate matching, usually performing such matches in $O(\log n)$ time regardless of the type or structure of the values themselves.[23] In addition, there are some operations, like finding the smallest and largest element, that can be performed efficiently on a sorted array.[11]

Linear search

Linear search is a simple search algorithm that checks every record until it finds the target value. Linear search can be done on a linked list, which allows for faster insertion and deletion than an array. Binary search is faster than linear search for sorted arrays except if the array is short, although the array needs to be sorted beforehand.[c][25] All sorting algorithms based on comparing elements, such as quicksort and merge sort, require at least $O(n \log n)$ comparisons in the worst case.[26] Unlike linear search, binary search can be used for efficient approximate matching. There are operations such as finding the smallest and largest element that can be done efficiently on a sorted array but not on an unsorted array.[27]

Trees

A binary search tree is a binary tree data structure that works based on the principle of binary search. The records of the tree are arranged in sorted order, and each record in the tree can be searched using an algorithm similar to binary search, taking on average logarithmic time. Insertion and deletion also require on average logarithmic time in binary search trees. This can be faster than the linear time insertion and deletion of sorted arrays, and binary trees retain the ability to perform all the operations possible on a sorted array, including range and approximate queries.[23][28]



Binary search trees are searched using an algorithm similar to binary search.

However, binary search is usually more efficient for searching as binary search trees will most likely be imperfectly balanced, resulting in slightly worse performance than binary search. This even applies to balanced binary search trees, binary search trees that balance their own nodes, because they rarely produce the tree with the fewest possible levels. Except for balanced binary search trees, the tree may be severely imbalanced with few internal nodes with two children, resulting in the average and worst-case search time approaching n comparisons.[d] Binary search trees take more space than sorted arrays.[30]

Binary search trees lend themselves to fast searching in external memory stored in hard disks, as binary search trees can be efficiently structured in filesystems. The B-tree generalizes this method of tree organization. B-trees are frequently used to organize long-term storage such as databases and filesystems.[31][32]

Hashing

For implementing associative arrays, hash tables, a data structure that maps keys to records using a hash function, are generally faster than binary search on a sorted array of records.[33] Most hash table implementations require only amortized constant time on average.[e][35] However, hashing is not useful for approximate matches, such as computing the next-smallest, next-largest, and nearest key, as the only information given on a failed search is that the target is not present in any record.[36] Binary search is ideal for such matches, performing them in logarithmic time. Binary search also supports approximate matches. Some operations, like finding the smallest and largest element, can be done efficiently on sorted arrays but not on hash tables.[23]

Set membership algorithms

A related problem to search is set membership. Any algorithm that does lookup, like binary search, can also be used for set membership. There are other algorithms that are more specifically suited for set membership. A bit array is the simplest, useful when the range of keys is limited. It compactly stores a collection of bits, with each bit representing a

single key within the range of keys. Bit arrays are very fast, requiring only $O(1)$ time.[37] The Judy1 type of Judy array handles 64-bit keys efficiently.[38]

For approximate results, Bloom filters, another probabilistic data structure based on hashing, store a set of keys by encoding the keys using a bit array and multiple hash functions. Bloom filters are much more space-efficient than bit arrays in most cases and not much slower: with k hash functions, membership queries require only $O(k)$ time. However, Bloom filters suffer from false positives.[f][g][40]

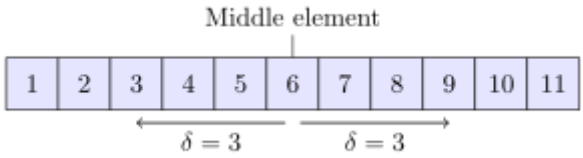
Other data structures

There exist data structures that may improve on binary search in some cases for both searching and other operations available for sorted arrays. For example, searches, approximate matches, and the operations available to sorted arrays can be performed more efficiently than binary search on specialized data structures such as van Emde Boas trees, fusion trees, tries, and bit arrays. These specialized data structures are usually only faster because they take advantage of the properties of keys with a certain attribute (usually keys that are small integers), and thus will be time or space consuming for keys that lack that attribute.[23] As long as the keys can be ordered, these operations can always be done at least efficiently on a sorted array regardless of the keys. Some structures, such as Judy arrays, use a combination of approaches to mitigate this while retaining efficiency and the ability to perform approximate matching.[38]

Variations

Uniform binary search

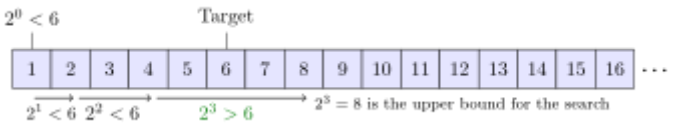
Uniform binary search stores, instead of the lower and upper bounds, the difference in the index of the middle element from the current iteration to the next iteration. A lookup table containing the differences is computed beforehand. For example, if the array to be searched is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], the middle element (m) would be 6. In this case, the middle element of the left subarray ([1, 2, 3, 4, 5]) is 3 and the middle element of the right subarray ([7, 8, 9, 10, 11]) is 9. Uniform binary search would store the value of 3 as both indices differ from 6 by this same amount.[41] To reduce the search space, the algorithm either adds or subtracts this change from the index of the middle element. Uniform binary search may be faster on systems where it is inefficient to calculate the midpoint, such as on decimal computers.[42]



Uniform binary search stores the difference between the current and the two next possible middle elements instead of specific bounds.

Exponential search

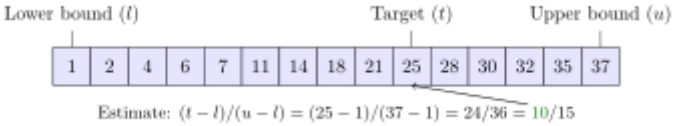
Exponential search extends binary search to unbounded lists. It starts by finding the first element with an index that is both a power of two and greater than the target value. Afterwards, it sets that index as the upper bound, and switches to binary search. A search takes $\lfloor \log_2 x + 1 \rfloor$ iterations before binary search is started and at most $\lfloor \log_2 x \rfloor$ iterations of the binary search, where x is the position of the target value. Exponential search works on bounded lists, but becomes an improvement over binary search only if the target value lies near the beginning of the array.[43]



Visualization of exponential searching finding the upper bound for the subsequent binary search

Interpolation search

Instead of calculating the midpoint, interpolation search estimates the position of the target value, taking into account the lowest and highest elements in the array as well as length of the array. It works on the basis that the midpoint is not the best guess in many cases. For example, if the target value is close to the highest element in the array, it is likely to be located near the end of the array.[44]

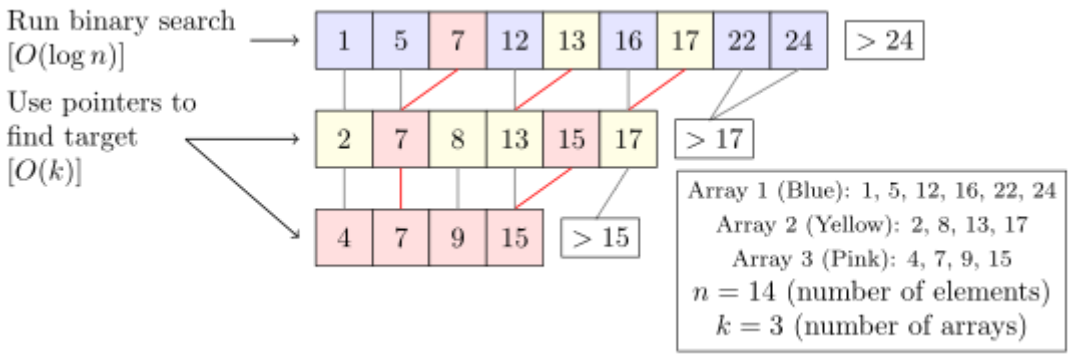


Visualization of interpolation search. In this case, no searching is needed because the estimate of the target's location within the array is correct. Other implementations may specify another function for estimating the target's location.

A common interpolation function is linear interpolation. If A is the array, L, R are the lower and upper bounds respectively, and T is the target, then the target is estimated to be about $(T - A_L)/(A_R - A_L)$ of the way between L and R. When linear interpolation is used, and the distribution of the array elements is uniform or near uniform, interpolation search makes $O(\log \log n)$ comparisons.[44][45][46]

In practice, interpolation search is slower than binary search for small arrays, as interpolation search requires extra computation. Its time complexity grows more slowly than binary search, but this only compensates for the extra computation for large arrays.[44]

Fractional cascading



In fractional cascading, each array has pointers to every second element of another array, so only one binary search has to be performed to search all the arrays.

Fractional cascading is a technique that speeds up binary searches for the same element in multiple sorted arrays. Searching each array separately requires $O(k \log n)$ time, where k is the number of arrays. Fractional cascading reduces this to $O(k + \log n)$ by storing specific information in each array about each element and its position in the other arrays.[47][48]

Fractional cascading was originally developed to efficiently solve various computational geometry problems. Fractional cascading has been applied elsewhere, such as in data mining and Internet Protocol routing.[47]

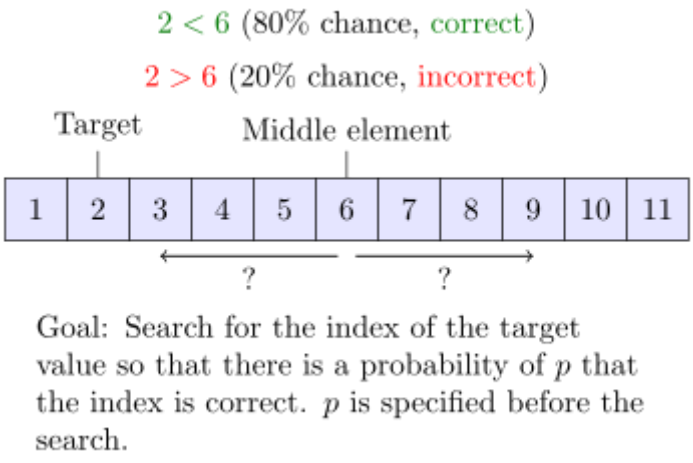
Generalization to graphs

Binary search has been generalized to work on certain types of graphs, where the target value is stored in a vertex instead of an array element. Binary search trees are one such generalization—when a vertex (node) in the tree is queried, the algorithm either learns that the vertex is the target, or otherwise which subtree the target would be located in. However, this can be further generalized as follows: given an undirected, positively weighted graph and a target vertex, the algorithm learns upon querying a vertex that it is equal to the target, or it is given an incident edge that is on

the shortest path from the queried vertex to the target. The standard binary search algorithm is simply the case where the graph is a path. Similarly, binary search trees are the case where the edges to the left or right subtrees are given when the queried vertex is unequal to the target. For all undirected, positively weighted graphs, there is an algorithm that finds the target vertex in $O(\log n)$ queries in the worst case.[49]

Noisy binary search

Noisy binary search algorithms solve the case where the algorithm cannot reliably compare elements of the array. For each pair of elements, there is a certain probability that the algorithm makes the wrong comparison. Noisy binary search can find the correct position of the target with a given probability that controls the reliability of the yielded position. Every noisy binary search procedure must make at least $(1-\tau)\frac{\log_2(n)}{H(p)} - \frac{10}{H(p)}$ comparisons on average, where $H(p) = -p\log_2(p) - (1-p)\log_2(1-p)$ is the binary entropy function and τ is the probability that the procedure yields the wrong position.[50][51][52] The noisy binary search problem can be considered as a case of the Rényi-Ulam game,[53] a variant of Twenty Questions where the answers may be wrong.[54]



In noisy binary search, there is a certain probability that a comparison is incorrect.

Quantum binary search

Classical computers are bounded to the worst case of exactly $\lfloor \log_2 n + 1 \rfloor$ iterations when performing binary search. Quantum algorithms for binary search are still bounded to a proportion of $\log_2 n$ queries (representing iterations of the classical procedure), but the constant factor is less than one, providing for a lower time complexity on quantum computers. Any *exact* quantum binary search procedure—that is, a procedure that always yields the correct result—requires at least $\frac{1}{\pi}(\ln n - 1) \approx 0.22 \log_2 n$ queries in the worst case, where $\ln$ is the natural logarithm.[55] There is an exact quantum binary search procedure that runs in $4\log_{605} n \approx 0.433 \log_2 n$ queries in the worst case.[56] In comparison, Grover's algorithm is the optimal quantum algorithm for searching an unordered list of elements, and it requires $O(\sqrt{n})$ queries.[57]

History

The idea of sorting a list of items to allow for faster searching dates back to antiquity. The earliest known example was the Inakibit-Anu tablet from Babylon dating back to c. 200 BCE. The tablet contained about 500 Sexagesimal numbers and their reciprocals sorted in Lexicographical order, which made searching for a specific entry easier. In addition, several lists of names that were sorted by their first letter were discovered on the Aegean Islands. *Catholicon*, a Latin dictionary finished in 1286 CE, was the first work to describe rules for sorting words into alphabetical order, as opposed to just the first few letters.[9]

In 1946, John Mauchly made the first mention of binary search as part of the Moore School Lectures, a seminal and foundational college course in computing.[9] In 1957, William Wesley Peterson published the first method for interpolation search.[9][58] Every published binary search algorithm worked only for arrays whose length is one less than a power of two[h] until 1960, when Derrick Henry Lehmer published a binary search algorithm that worked on all