# Average-case analysis of binary search

```
Algorithm BinarySearch (data, key):
  Input: a sorted array of integers (data) and a key (key)
  Output: the position of the key in the array (-1 if not found)

  low ← 0
  high ← data.length - 1

  while low <= high do
    int mid ← (low + high) / 2
    if (data[mid] < key)
      low ← mid + 1
    else if (data[mid] > key)
      high ← mid - 1
    else
      return mid

  // couldn't find the key
  return -1
```

We wish to understand the average case running time of binary search. What do we mean by average case? Let us consider the case where each key in the array is equally likely to be searched for, and what the average time is to find such a key.

To make the analysis simpler, let us assume that $n = 2^k - 1$, for some integer $k \geq 1$. (Why does it make the analysis simpler?)

We first notice that in the two lines of pseudocode before the while loop, 3 primitive operations always get executed (an assignment, a subtraction, and then an assignment). However, since these operations happen no matter what the input is, we will ignore them for now.

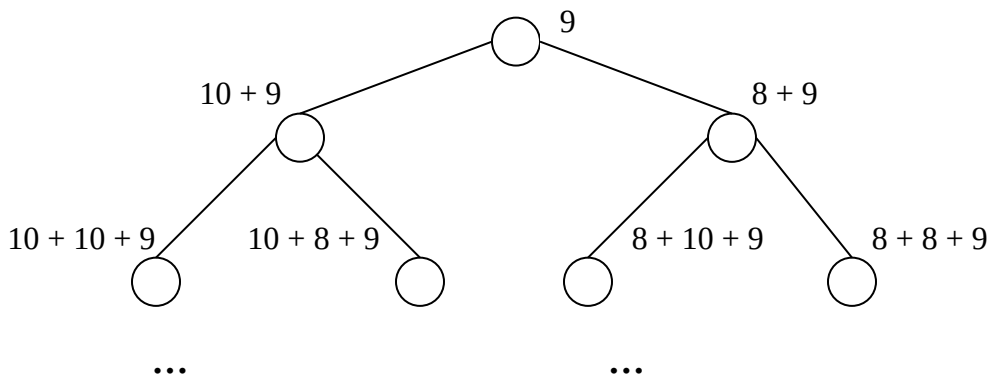
Focusing our attention on the while loop, we notice that each time the program enters the while loop, we execute 6 primitive operations (a <= comparison, an addition, a divide, an assignment, an array index, and a < comparison), before the program might branch depending on the result of the first if statement.

Depending on the result of the conditional, the program will execute different numbers of primitive operations. If data[mid] < key, then the program executes 2 more primitive operations. If data[mid] > key, then the program executes 4 more primitive operations (an array index and a < comparison in the next if, and then a subtraction and assignment). If data[mid] = = key, then we execute 3 more primitive operations (2 operations for the if and then 1 operation to return mid). In other words, the number of primitive operations executed in an iteration of the loop is

10 , if data[mid] > key,
9 , if data[mid] = = key, and
8 , if data[mid] < key.

We can now construct a tree that summarizes how many operations are executed in the while loop, depending on the number of times the while loop is executed and the result of

the comparisons between data[mid] and key in each iteration. Each node in the tree represents the exit from the algorithm because data[mid] = = key. A node is a left child of its parent if in the previous iteration of the while loop, the search was restricted to the left part of the subarray (that is, when data[mid] > key). A node is a right child of its parent if in the previous iteration of the while loop, the search was restrictred to the right part of the subarray (that is, when data[mid] < key). Here are the first three levels of the tree:




The root of the tree corresponds to the situation where data[mid] = = key in the first iteration of the while loop. Nodes on the *i*th level of the tree correspond to finding the key after executing *i* iterations of the while loop (level 1 is the root of the tree). Notice that the number of numbers at each node is equal to what level it is on. Since $n = 2^k - 1$, then the number of levels in the tree is $k = \log (n + 1)$.

Now consider the number of operations that occur at each level. Notice that every node has a 9 in it (which comes from the last iteration of the while loop when the item is found) and that for every node with some number of 10s and 8s in it, there is a corresponding node on the same level that has exactly same number of 8s and 10s, respectively. Since we are only interested in adding all of the numbers in the tree, we can simplify our calculations by simply making all of the numbers 9s.

Let T be the sum of all of the numbers at all of the nodes in the tree. Except for the 3 operations we ignored earlier, T is the total amount of time it would take to execute binary search *n* times, one for each item in the array. So, T / *n* is the average time to find a key.

Each node at level *i* has *i* 9s in it, and there are $2^{i-1}$ nodes in level *i*. It is easy to see that T / *n* has the value $\frac{9}{n} \sum_{i=1}^{\log(n+1)} i\, 2^{i-1}$.

We now simply need to compute what $\sum_{i=1}^{\log(n+1)} i\, 2^{i-1}$ is. Let us write out the terms in the summation:

$$\sum_{i=1}^{\log(n+1)} i\, 2^{i-1} = 1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + 4 \cdot 2^3 + \ldots + \log (n + 1) \cdot 2^{\log (n + 1) - 1}$$

$$= 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + \ldots + 1 \cdot 2^{\log (n + 1) - 1} \qquad (1)$$

$$+ 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + \ldots + 1 \cdot 2^{\log (n+1) - 1} \quad (2)$$
$$+ 1 \cdot 2^2 + 1 \cdot 2^3 + \ldots + 1 \cdot 2^{\log (n+1) - 1} \quad (3)$$
$$+ 1 \cdot 2^3 + \ldots + 1 \cdot 2^{\log (n+1) - 1} \quad (4)$$
$$\ldots$$
$$+ 1 \cdot 2^{\log (n+1) - 1}$$

Note how the summation has been broken up. What we do now is notice that the terms on line (1) are the form of a geometric series that grows by a factor of 2.

$$1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + \ldots + 1 \cdot 2^{\log (n+1) - 1}$$
$$= \sum_{i=0}^{\log(n+1)-1} 2^i = \sum_{i=-\infty}^{\log(n+1)-1} 2^i - \sum_{i=-\infty}^{-1} 2^i = 2^{\log (n+1)} - 2^0$$

Line (2) can be calculated in a similar way:

$$1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + \ldots + 1 \cdot 2^{\log (n+1) - 1}$$
$$= \sum_{i=1}^{\log(n+1)-1} 2^i = \sum_{i=-\infty}^{\log(n+1)-1} 2^i - \sum_{i=-\infty}^{0} 2^i = 2^{\log (n+1)} - 2^1$$

In general, the sum on line $i$ is $2^{\log (n+1)} - 2^{i-1} = (n+1) - 2^{i-1}$, and there are $\log (n+1)$ lines. To compute the entire sum, we need to compute $\sum_{i=1}^{\log(n+1)} ((n+1) - 2^{i-1})$.

$$\sum_{i=1}^{\log(n+1)} ((n+1) - 2^{i-1}) = \sum_{i=1}^{\log(n+1)} (n+1) - \sum_{i=1}^{\log(n+1)} 2^{i-1}$$
$$= (n+1) \cdot \log (n+1) - (2^{\log (n+1)} - 1)$$
$$= (n+1) \cdot \log (n+1) - ((n+1) - 1)$$
$$= (n+1) \cdot \log (n+1) - n$$

So, $T = 9\left(\frac{n+1}{n}\right)\log(n+1) - 9 \approx 9 \log n - 9.$

This means that that average number of primitive operations on a successful search is about $9 \log n - 6$. (We need to add the three operations we ignored at the beginning of our analysis.)

Note that in the worst case, a successful search executes approximately $10 \log n$ steps, since the bottommost, leftmost node of the tree has value $10 \log (n+1) - 1$. The average time for a successful search is only $\log n$ steps smaller than the worst case because almost exactly half of the nodes are at the bottom of the tree, and those nodes have an average value of $9 \log n$. So, even though there are many nodes near the top of the tree that have a very small value, they influence the average by only a constant ($9 \log n - 9$ on average for all nodes, as opposed to $9 \log n$ on average for just the leaves of the tree) because there are so many more nodes at the bottom of the tree.