Binary Search Algorithm:

Binary search is probably one of the most common algorithms that we all use without even realizing we are using it.

Guess the Number Game:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 3 | 9 | 11 | 18 | 34 | 40 | 45 | 50 | 60 |

Let say you have an array sorted in numerical order and you want to see if your guessed number exists in the array. If you start from the front and check every element in the array the complexity will be O(n) if the number is very big and vice versa.

However, if you start from the middle number of the sorted array and find the condition if middle number is greater than or smaller than the given number then it will reduce to half time. And again, start from middle number. This is called binary search.

In summary:

* Binary search is a search algorithm where we find the position of a target value by comparing the middle value with this target value.
* If the middle value is equal to the target value, then we have our solution (we have found the position of our target value).
* If the target value comes before the middle value, we look for the target value in the left half.
* Otherwise, we look for the target value in the right half.
* We repeat this process as many times as needed, until we find the target value.

Time Complexity Binary Search:

it won't always be *exactly* half the numbers that get discarded. If you have an even number of elements, you will have to check either the lower or higher of the middle two elements—and this means you'll rule out either half of the array, \frac{n}{2}2*n*​, or one more than half the array, \frac{n}{2} + 12*n*​+1. But when we calculate time complexity using big O notation, we tend to ignore such small details, because they have negligible impact on the efficiency. Usually, we are concerned with large input sizes—on the order of, say, 10^5105. Imagine an array of size 10^5105! It doesn’t really matter if each step rules out exactly half of the array, \frac {10^5}{2}2105​ or slightly more than half of the array, \frac {10^5} {2} +12105​+1. So to keep things simple here, we will ignore the +1+1.



Here's what we're doing at each step:

* In the first step, we discard half of the numbers—that is, \frac{n}{2}2*n*​ numbers. So, the total number of remaining integers is also half, or \frac{n}{2}2*n*​.
* In the second step, we discard half of the numbers that were left with us from the first step. We had \frac{n}{2}2*n*​ integers, so we discard half of these numbers and hence are left with \frac{n}{4}4*n*​ integers.
* Similarly, in the next step we again discard half of the numbers that were left in the last step. Thus, we are now left with \frac{n}{8}8*n*​ integers.
* We'll continue this process until, in the final step, we will have only one integer left. We will compare with this integer and check whether this is our target.

In case you want to follow the math, here are the steps:

n \* \frac {1}{2} ^s = 1*n*∗21​*s*=1

Use the properties of negative exponents to rearrange the fraction:

n \* 2^{-s} = 1*n*∗2−*s*=1

Divide both sides by {2^{-s}}2−*s*

\frac {n \* 2^{-s}} {2^{-s}} = \frac {1} {2^{-s}}2−*sn*∗2−*s*​=2−*s*1​

n = \frac {1} {2^{-s}} *n*=2−*s*1​

Again, use the properties of negative exponents to rearrange the fraction:

n = 2^s*n*=2*s*

Take the logarithm (base 2) of both side:

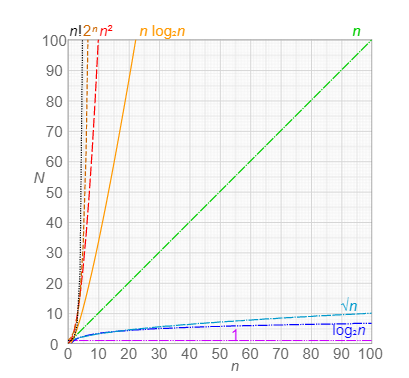
log\_2(n) = log\_2(2^s) *log*2​(*n*)=*log*2​(2*s*)

log\_2(n) = s*log*2​(*n*)=*s*

The bottom lines? The number of steps is equal to the logarithm of the input size:  
s = log\_2(n)*s*=*log*2​(*n*)

This is the number of steps it will require to find the target number in the worst-case scenario. In big-O notation, we would say that the time complexity is O(log\_2(n)) *O*(*log*2​(*n*)).

If we look back at our comparison of computational complexities, we can see that this is extremely efficient:



[**"Comparison of computational complexity"**](https://commons.wikimedia.org/wiki/File:Comparison_computational_complexity.svg) by **[Cmglee](https://commons.wikimedia.org/wiki/User:Cmglee" \t "_blank)**. Used under [**CC BY-SA 4.0**](https://creativecommons.org/licenses/by-sa/4.0/deed.en).

In fact, this efficiency is the second best on the graph, with only constant time complexity, O(1)*O*(1) performing better!

Even as the input size grows very large, the number of steps required is still surprisingly small.

Going back to our guess-the-number game, you should now see why it's possible—using binary search—to correctly guess a number, out of 100. with only a handful of tries. If you like, go back and try inputs of 200 or even 1000. The algorithm will still perform quite well! (For an input of 1000, you might need slightly more than 7 tries.)

Another Advanced binary search algorithm is **Trie**: The general use of Trie algorithm is to check the spelling and find that the word is correct or not.

**Heaps:** It is another specific type of tree with some of its own additional rules. In heap elements are arranged in decreasing or increasing order, such that the root element is either the maximum or minimum value in the tree.

There are two different types of heaps.

1. Max heaps: A parent must always have a greater value than its child so, the roots ends up being biggest element.

2. Min heaps: The opposite is true for min heap and the parent will be the smallest element.

Heaps do not need to be binary trees, so parent can have any number of children. Operations like search, insert and delete can vary a lot based on the type of heap.

MAX Binary Heap:

Here we are going to keep the two children rule and the root be the maximum element, in addition a binary heap must be a complete tree. Meaning all levels except the last one is completely full. If the last level isn’t full, values are added from left to right. The right most leaf will be empty until the whole row has been filled. I the following heap, a function that gets the maximum value,

O(1)

5 Steps,

5 elements O(n)

In this heap, a function gets the maximum value, also called peek, happens in constant time.

How search looks like, should we start our search by going to left or the right? In the BST, we knew which direction to go at each step by doing comparisons. Here there is no guarantee either way. Thus, searching ends up being a linear time operation since normally, we can’t rely on tricks and we’ll searching the entire tree. One thing to note is that we can actually use the maxi properties to our advantage in a search.

For, example we can quit our search immediately if the element we’re searching for is bigger than the root. In general, if our node value is bigger than the one, we’re comparing to, we don’t need to check anything in its sub-tree since we know that it’s the biggest. The worst case remains but in the average case, we don’t actually need to search every element. The average case is closer to n over two but that still approximated to linear time.

Search: worst O(n), Average about O(n/2) = O(n)

**Heapify:** Next, let’s try to insert the element 21:

We could take the approach we used with BST’s, start at the root then move down the tree one level at a time and do comparisons until we find the open spot. However, if our element is bigger than most of the parent nodes or even the root will need to shuffle the tree around a lot. As, such we take a different approach to inserting here. We can stick to new element in the next open spot in the tree. Then we Heapify. Heapify is the operation in which we reorder the tree based on the heap property. Since, we care that our parent element is bigger then its child, we just need to keep comparing our new element with its parent and swapping them when the child is bigger. We can take a similar approach to an extract operation where the root is removed from the tree, we stick to right most leaf in the root spot then just compare it to its children and swap where necessary.

Worst case: O(log(n)) Height of the tree

The runtime of insert and delete, a more general case of extract ends up being O of log in the worst case. Ultimately the worst case would involve moving an element all the way up or down the tree and would roughly be as many operations as the height of the tree.

Heap Implementation: The heaps are represented as trees, they actually often stored as arrays. Since we know how many children each parent has, two, and thus, how many nodes will be at each level, we can use a little math to figure out where the next node will fall in the array and then traverse the tree.

19->17->16->15->11->6->4->2->1

19 will go to the root since it’s the biggest. And will follow the numbers.

Self-Balanced Tree: In the world, a balanced tree has nodes condensed to only a few levels, while an unbalanced tree has nodes spread out among many levels. The most extreme type of unbalanced tree is really just a linked list, where every node has only one child. By the definition a self-balancing tree is one that tries to minimize the number of levels that it uses. It does some algorithm during insertion and deletion to keep itself balanced and the nodes themselves might have some additional properties. The most common example is the **Red – Black Tree**, which is an extension of binary search tree. In this kind of tree, nodes are assigned an additional color property where the value must be either red or black. The use of color red or black is just a convention. The second property of red and black tree is the existence of null leaf nodes. Every node in the tree that doesn’t otherwise have two leaves must have null children. Additional optional rule is the root node must be black. Lastly, for the rule that makes these trees actually useful, every path from a node to its descendent null nodes must contain the same number of black nodes.

**Red – Black Tree Insertion:** A heap is another specific type of tree with some of its own additional rules.

There are several different states of the tree and the node you are inserting that require different courses of action. The resulting tree need to follow both the Red-Black and the BST rules. One overall rule of insertion is that you should try to insert node as a red node, and then change its color as needed. The first situation is where you’re inserting the first node in to the tree. Since the root, you can change the color to black if you are adhering to the root must be black. Otherwise, you’ll have nothing to do. If the new parent node is black, you don’t have to do anything. Since, you are adding the red node you haven’t upset the balance of black nodes in any path or violated any of the other rules. Now, if the parent is red, there are several cases with more complicated solution. If the parent and its sibling are both red then they should be changed to black, and their parents, the grandparents, let’s say the node you are inserting, becomes red. We switch the colors of the node to maintain the number of black nodes in a given path.

**Tree Rotations:**

**Case 4:** In case 4 and 5, the node’s parent is red and its sibling is black

Grand Parent

Parent Sibling

Parent

16 is inserted node

In both, you will need to perform rotation.

In a Rotation you shift a group of nodes around in a way that changes the structure of the tree, but not the order of the nodes. This is still a BST, so we need to keep our elements in strict order.

In case 4 our red node and its red parent are not on the same side of their parents. Our node is a right child and its parent is a left child. Here we will perform a left rotation since the nodes shift one place to the left while maintaining their order.

In case 5: At this point we have a set up that looks exactly like case 5, where both the red node and their parent are both on the same side of their parents. We will do a right rotation here, but this time involving the grandparents and both of its children.

We need to swap the colors of these two nodes as well. And we have rearranged the node without changing the number of black nodes