Module 13 - Searching

The Searching Problem 1.2

Let's talk about two fundamental searching algorithms. The first one is the most general searching algorithm; it will implement a function search that is given an array, basically the starting address when of an array and its logical size, and in addition it would also get value, an integer value, val. And this function should return an index inside this array where val first appears. Or if val doesn't appear as one of the elements in A R R this function would just return a negative one. So, for example, if array contains five, eight, twelve, seven, eight, and ten; for example, if we call the search function with this array of size six searching for the value eight, we're expecting to get the index of the first time where eight appears. That's the second position so the index would be one; this function in this case should return one. If we call the search function on the same array of size six searching for four, which as you can see doesn't show in the array, the function should then return a negative one. So let's implement the search function.

Searching Code Sample 1.3

Very intuitively we should basically iterate over the elements of the array. And check each one whether its value is valid or not. Since we're not sure how many iterations we're going to have, cause may be not all we won't need to go over the entire array, we'll see val somewhere in the middle, we can just break out. I think it's better to use a while loop, in this case. Let's use a while loop and have indexes I, in this case, iterating over the array, let's initialize I to zero. Each iteration will increment I by one, I plus plus, and we'll go while I is still inside the logical portion of the array the while I is less then there A R R's size. And then for each element let's check if there A R R I is value is val, in this case let's break the function and return this index, let's return I. if will go over the entire array in none of the elements match value, in this case outside of the a while loop we'll return negative one. Basically meaning we've gone over the entire range and none of the elements are val, so valid doesn't appear negative one should be the return value. Very basic implementation so let's try to analyze the running time. So, the instructions, we have the expressions we have before and after the while loop or constants, so they each take a Taito of one. The while loop cost; let's try to figure out. So, the body of the while loop is a constant, an if statement and the I plus plus are one, two, three statements each iteration, so let's say Taito of a one. Then the entire while loop, since we're either iterating, so let's figure out how many iterations we have for the while loop. So, it would cost us as the order of the number of iterations. So, if T of N, is as we said Taito of number of iterations, we notice that the worst case scenario is when the number of iterations is basically the size of the array, basically N. So, in this case, T of N for worst case would be paid over. So, this search as you can see is a linear search; we're going over the entire set of elements of the array and it is in a linear time, so T of N is Taito of N.

The Sorted-Search Problem 1.4

Okay. So, let's take a look at another search algorithm; basically it's a variation of the general searching problems algorithms, named the sorted search problem. in this case we'll implement the function, a sorted search, where we're given a sorted array, sorted A R R, that's the name of our array. Which means that the elements are ordered in this array in an increasing order, so we get the sorted array and its logical size and again a val to search for. And same thing, so if val appears in the sorted array, let's just return one of the indexes where it appears. We're not really interested in finding the first appearance of value in a sorted array, so let's just return an index where val appears. And once again, if val is not one of A R R elements, we'll just return a negative one. For example, if our sorted array is five, seven, eight, eight, ten, and twelve, again notice that the elements are ordered in increasing order, then when we call this sorted search function for this array of size six searching for eight, we'll just return an index where eight appears; in this case eight appears in index two and three, we could arbitrarily return three.

The Sorted-Search Problem 1.5

So let's try to think how we can solve this problem. First, most basic idea would say do the same thing as we've done before, basically searching the array element by element. So let's see what the first value of the array is, in this case it is one. So, we're searching, for example val equals seventy nine, so it's not one. Let's go over to the next one three; that's not seventy nine. It's go over for the next one, four that's not seventy nine. Once again eight; that's not seventy nine and so on. We can continue on iterating over the elements with an array. Yeah, basically could work, once again the running time of this approach would be linear because we'll worst case have to iterate over the entire range of elements, basically Taito of N.

But this approach here doesn't take advantage of the fact that our array is given to us, basically, in an increasing order and we can take advantage of that if we do something a bit differently. So, for example if we’re again searching for seventy nine. Let's take a look at the entire range and look only at the element that appears in the middle of the array. Let's test that element for its value. for example, this element could be ninety six; since we know we're searching for seventy nine and at the middle there is ninety six, we don't really have to go over the entire left side of the array right because after all on the left side of ninety six would appear only values that are greater than ninety six. So, just by looking at ninety six, we can rule out all the greater part there or the left part of ninety six and keeping us with only the smaller values smaller than ninety six to look for seventy nine. So, with one check we basically ruled out half of our range. And then once again for the half of the numbers we still have to search for; again we can look at only the middle point of that range, testing the value of the element that is there. For example, twenty eight and then once again since we're looking for seventy nine the numbers are in an increasing order, if we know that in some position there is twenty eight we can rule out all the elements there are smaller than twenty eight; we can basically rule out all of these elements and keeping us with only the range of elements between these twenty eight and ninety six. Once again, we can check what's in the middle of that range; for example, eighty four and we can rule out all the elements that are greater than eighty four. We can, we’re then left with the inner range over there looking at the middle point and luckily seventy nine; so we've just found our seventy one. So, basically we are starting with the regional range, and each time we're ruling out, we're taking off half of the elements, basically closing on the element we are searching for. Let's try to implement this approach on the computer.

Sorted Search Implementation 1.6

Let's implement the sorted search function. So I have here a main program that I created, a sorted array. Notice that the elements in this array are increasing order, one, three, five, seven, nine, and so on. And I've made it call to the sorted search function, that is given the array, sorted array and its logical size of ten. and for example we're searching for thirteen here, and thirteen does appear over here so we're expecting to get the index of thirteen, which is zero, one, two, three, four, five, six; so, we're expecting to get a six out in this print statement. And let's just implement the sorted search function so this program can execute. I have this prototype up here; let’s copy it and do our implementation, so something like that. And okay. So, let's get started.

So, we have our sorted array, something like that, and each time we are limiting half of the range. So, let's have two indexes to indicate the valid range we're searching initially; let's name the low and high. So, initially they are set to the entire range of the array, so let's create two variables low and high and set them initially low to zero and high to the last index of the array, which is the sorted array size minus one. And then, we should start iterating each time eliminating half of the valid range. For that we would use a while loop, right. We don't know exactly how many iterations we're going to have so let's use a while loop here. And I'll keep the Boolean condition here empty for now, till we figure out exactly what's the body of the loop it would be easier for us to state that really in condition.

Okay. Another thing I think it's good to set from the beginning would be a Boolean flag that indicates if we found the value val we are searching for or still haven't done that. So, let's have Boolean variable name found, that would be set initially found to false and when we find our value we'll update it to true, but initially it would be false. And then let's start iterating, so each iteration we are searching the range from low to high; and we said we're not searching it linearly, we are taking the middle point and comparing the value we have in here, to the value we're searching for. So, for example if our value were searching is as we had before seventy nine, let's take the middle point and compare it to seventy nine. So let's have a mid index; so we have mid. And let's set mid to the middle between a low and high. So mid would be… how can we figure out what mid should be?

So mid, in order to be the middle between low and high is basically the average between low and high. So low plus high over two; that's the value of mid, so let's set mid to low plus high over two. Notice that since low and high are both integers, obviously, low plus high is also an integer and since two is an integer literal then these two operands of the dividing slash here are integers basically saying that C++ would interpret this slash as a div not as a real division. Which is very good for us because we don't really want mid to be the real average between low and high; it should be a valid index, an integer value.
So, div basically takes it floored down which is an estimation of the point; it could be offsetted one index to the left. So, we have the middle position and now we should compare val to the array in the middle position. So let's see if, and in here there could be a few values, so for example if we're lucky the value in sorted array at the middle position is val, exactly the value we're looking for, in this case we should raise the flag basically say that found is true. Right? So if in here we have, for example seventy nine, then basically we're done. We should just say that val, that found is true, and break out of the function return the index we're at. Actually, I don't want to break the while loop by a return statement so I'll just save the position that val is found so I'll create another integer named ind where I'll store this index and I'll set ind to be the mid point basically where value is right that is at the mid index. So if we found val, we are storing the index and setting found to true. Hopefully that would help us breakout of the while loop; we'll take care of it in a few minutes when we state the Boolean condition.

But most likely we won't be so lucky and we won't get seventy nine right here in the beginning.
So, let's see, else if we're as we said not so lucky and value isn't exactly as sorted array mid, we should compare it and see how what relational order they have. So, for example if we have say ninety six, or in other words if our val is less then sorted A R R at the middle position, right; like we have here where seventy nine is less than ninety six. In this case, as we said, we can eliminate all of that range right. In order to do that, we should then just said high over here right; so, we can basically create that as our valid range. To do that let's say that high in this case should equal to mid minus one right. We ruled out all the elements there are to the left, to the right, actually of ninety six. Okay, so that takes care of that.

But another case could be if we have here a value instead of ninety six that is a value that is less than seventy nine. So if we have instead of ninety six, let's say fifty one; so we're in an else right. So sorted A R R mid is not seventy nine, sorted A R R mid is not greater than seventy nine, not greater than val. In this case, that mid … maybe comment and say that we are in the case where val is greater, seventy nine is greater, than the sorted array in the middle position.

In this case, we can eliminate all of these elements and to do that we just set low to here, right. So we should just say low equals mid plus one, right; setting the valid range to this half of our searching range. Let's take a look at the body we have for the while loop so each iteration we're figuring out where the middle index is and then we're checking whether the middle element is exactly the value we're searching for, in this case where saving this position and hopefully raising the flag that would lead us to break out of the while. Otherwise if we are not so lucky we are comparing val to the element that is in the middle position; if val is less than we are eliminating all the right values, the values that are greater than the middle position basically setting high to mid minus one. Otherwise, we're eliminating all the values to the left of the middle position, basically saying that low is mid plus one and we do that over and over. While, so obviously, while found is still false right. We initially set found to false, so we want to keep on iterating while found is still false. But there are cases where found would never become true, right. So, if the value val appears in sorted A R, eventually we'll close on it and find its position, but in case value doesn't appear in sorted A R R at all found would never become true, and we want to be able to break out of this while loop in that case as well.

So, in order to figure out what we need to do here. Let's take a closer look; I'll maybe try to demonstrate it on a small numerical example. So let's take a new array of three elements, let's have, I don't know, two, five, seven; these are our elements. There are indexes: zero, one, two. So, initially we have low and high over here, right. And let's try to execute this code and see what happens if we are searching for a value that is not in this array, let's search for four, right. Obviously, we should figure out that it doesn't appear there. Let's try to execute the code. So, initially mid would be low plus high div two, basically zero plus two. That's two div two, two div two that's one. So mid would be here, right. And then we're trying to compare if in index one element equals val, equals four, that's not true right. Our value here, five, does not equal four. In this case, we should then continue on. So if val four is less then what we have in the middle position which is five, so four is less than five which is true, then we should set high to be mid minus one, which makes a lot of sense right. If four would appear here, it wouldn't be in the right half, it would be in the left half.

So, let's set high to being mid minus one; so high would then be right here. So high would be zero as well. Okay. So we have both low and high at zero position; another iteration mid would be low plus high div two, zero plus zero div two that would be zero as well. So, our mid would also be here. Once again we're checking whether the element we're searching in, which is basically the only element in our valid range, the one and only element in our range. So, two is it equal to val, to four? It is not, so or we should check whether val four is less than the element we have in the middle; is four less than two, that's not true as well. So it means that the val is greater than our middle element, which is true four is greater than two, in this case we said that we should make low mid plus one.

So our low would go over here. Okay. So as you can see, basically, low crossed over high; always we have low to the left of high, right. Initially low is to the left of high then we change high to that half and again, low is to the left of high and low would go over here and it would be to the left of high and high would go over here and a low would be to the left high. But if we have one single element here and it's not that element, low would go over high and that basically means that our element is not found.
So, in addition to looking whether the flag is turned out to true or false. We should always check whether our range is valid, basically if low is less or equal to high. So, if both we still haven't found our element, found is still false, and our range is a still valid range, low is still to the left of high, low is less or equal to high, we want to keep on iterating.

But if found becomes true, we want to break out. If low crosses over high and low is not less or equal to high, that means our val doesn't appear, we want to break out. Okay. So after this while loop, we should basically check what's the reason we broke out; if we found this element? So if found equals true that means that val appears in sorted array so we should just return the index we saved earlier, this index here. But if found is false, basically that means that we broke out of the loop not because we found the element but because our range is invalid, low is not less or equal to high, in this case we should just return negative one.

Okay, yeah. That seems to be the implementation for this function. Let's try to execute it and see if we get a six if we're searching for thirteen in this sorted array. Okay. We've got a six here. Let's try to call it one more time with an element that does not appear. For example, it's fourteen and see that we get a negative one here. Yeah. For fourteen we got the negative one which seems to be right. Yeah. So, just you know maybe it would be a good idea if you try on your own, to take an array with values and to trace it on your own, but the idea here is just as we've talked earlier. Let's try to analyze the running time of this algorithm, right now.

Implementation 1.7

Let's try to analyze the running time of the sorted search function. Hopefully, it would give us a better result than linear running time; we've worked quite a lot in order to get this coded. It's much more complicated; I hope it's worth it. Let's take a closer look. So, once again if we look at the code we have a few statements before the while loop but they are constants. So, let's consider them as Taito of one. There is the if statement after the while loop, also constant, Taito one. And I think the dominant part here is basically the while loop, so let's try to analyze the running time of this iterative process.

Once again if we look closer at the body of this a while loop, there are a few statements here but we can count them; they are not depending on the size of the input, no matter what the size of the range the number of operations we're doing there remains the same. that's why we're staying saying it's constant, it's also Taito of one. Which basically implies that the number of operations, the total number of operations for the while loop, is once again the number of iterations is the order of the number of iterations. But in this case it's a bit more tricky to figure out what's the number of iterations that this algorithm can do.

The Sorted-Search Problem 1.8

Let's try to make a table that would help us estimate how many iterations this algorithm does. So let's try to find the relationship between the iteration number and the size of the searching range. We have each time, right. We are basically cutting half of the range each time. So, let's try to find the relationship between these two sizes. So first iteration, the size of the searching range is the original size basically N right. Second iteration, we cut the searching range by two, by half. So, that gives us, leaves us a searching range of size N by two. Next iteration the range is again cut by half, so we'll have a range of N by four and next iteration the range would be N by eight and so on.

In a general iteration number for example K, let's try and figure out or formulate the size of this searching range. So if we take a closer look at the first four numbers will see that N by eight is also can be written as N by two to the power of three and N by four can be written as N over two square and N by two can be written as N over two the power of one. They're all N over two to some power, even the first N can be thought as N over one or in other words N over two the power of zero, which is one. And we can see that each iteration we have N over two to the power of one less than the iteration number, right; two to the three is two to the power of four minus one. And two to the two is to to the power of three minus one or in general form we can say that in a ration number K, the size of the searching range is N over two to the power of K minus one, right. It matches the pattern we've found before. So the biggest question is: what's the number of the last iteration, right? That's basically would tell us the worst case scenario right. Obviously we can stop at the first iteration and find the value we're searching for, but the worst case is we'll keep on reducing the sorting range more and more and more till we have only one element in that certain range and then low and the high would cross over one another.

So we should figure out what the iteration number where our searching range basically equals one. This question mark: what's the number of this iteration where the searching range is one? If we figure that out that would be basically our running time; that would be the number of iterations.
So let's do the math; let's ask ourselves for what value of K, N over two to the K minus one equals one right. If the general form of the size of the searching range is N over two to the power of K minus one for which K this size equals the size of the last searching range, basically one.

So let's do some math tricks here; let’s multiply by two to the K minus one that says that N equals two to the power of K minus one. Let's apply log base to both sides of the equation, basically saying that Log two of N equals log two of two to the power of K minus one. I hope you are familiar with the log rules here; where you can say that log two of two to the power of K minus one basically equals to K. minus one times log two of two. So that means that look two of N. is then equal to K. minus one times Log two of two. Again, Log two of two is one so that means that Log two of N basically equals to K minus one. Let's add one to both ends; that means that K, which is what we're looking for, equals to one plus Log two of N. In order of growth means that is K. is theta of Log two of N, saying that the number of iterations is theta of log two of N. And that concludes the worst case running time of the sorted search here. A lot of people also name it ‘Binary Search;’ the running time of this binary search algorithm is T of N equals theta of Log two of N. The next video we’ll talk about how great improvement log two of N versus a linear algorithm is.

Linear vs. Logarithmic 1.9

Okay. So we've seen two searching algorithms: the general search algorithm, which was linear time, and sorted search, the binary search algorithm which was a log of N time. Now obviously a theta of log N is significantly better than a theta of N, but I want to show you or give you a better intuition what's the meaning of this difference of log time versus linear time. So, you would really appreciate these kinds of algorithms: the logarithmic time algorithms. Let's try to compare the value of N versus the value of log base two of N for several values. So, for example for N equals two, log two of N would be log two of two which is one, slightly better than N, one is obviously smaller than two but doesn't seem such a big difference not something to be too happy about. Let's go on, for N equals four, I'm skipping three by the way just because I want the log of N to be integer values. So, any value between two and four, in this case three, would be somewhere in that range and you see that later on as well. So, for N equals four, log two of N is log two of two squared which is four, that equals two. So again log two of four is less than four. But again doesn't seem such a big difference; let's go on. Let's keep two eight two to the power of three; let's take a look at log two of eight. That would be log two of two to the power three, that is three again better than eight not something to get too crazy about.

Let's go to a larger value: two to the power of ten, around one thousand right. So one thousand on the N but log two of two to the power of N, is only ten that starts to be a greater difference here; a thousand versus ten. And let's go on, two to the power of thirty two; to the power of thirty two, that's around four point three billion. When we're taking the log two of N that would be a log two of two to the power of thirty two, that it would be thirty two. That already is quite a big difference: four point three billion versus only thirty two. So an algorithm that had to do four point three operations versus an algorithm that has to do only thirty two operations, the thirty to one would run much faster. But let's even go further; let's talk about N equals two to the power of one thousand. Now that is a huge number, it represents like the quantity of two the power of one thousand is greater than the number of the total number of practicals in the universe. So that's a huge number, obviously it would take practically infinite time for an algorithm to run, to execute two to the power of one thousand instructions. But a logarithmic algorithm, on the other end, which has two log two of two to the power of one thousand, would basically need to execute only one thousand instructions and that is less than the second.

So a logarithmic algorithm is much faster; obviously as N grows but it's much faster than the linear one. And that's the thing to take into account where we are implementing an algorithm, if we can create a logarithmic algorithm that we would give us a much better performance than a linear one. Just you can see that a linear function, if we look at the graph of F of N equals N versus a function equals to log of N, you can see that they are both kind of growing. I don't know if you can see that log of N also grows to infinity very slow, but it also grows to infinity. If we zoom out this difference would even be more visual. Again, you can see that F of N equals N, the linear function, grows very fast where the logarithmic function actually seems very much like a constant, almost doesn't grow at all. It's kind of fascinating but it would grow to infinity eventually but very very slowly. So a logarithmic algorithm is it's a very good resource, is very good algorithm.