**<Binary Search(이진 탐색)>**

**1. intro to algorithm**

What is an algorithm and why should you care?

- algorithm이란 어떤 문제를 해결하기 위한 절차들의 집합. In computer programming, 문제 해결을 위한 "명령어"들의 집합.

What makes a good algorithm?

- Correctness

- Efficiency

**2. Binary Search**

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

**Describing binary search**

The main idea of binary search is to keep track of the current range of reasonable guesses. Let's say that I'm thinking of a number between one and 100, just like the guessing game. If you've already guessed 25 and I told you my number was higher, and you've already guessed 81 and I told you my number was lower, then the numbers in the range from 26 to 80 are the only reasonable guesses. Here, the red section of the number line contains the reasonable guesses, and the black section shows the guesses that we've ruled out:



In each turn, you choose a guess that divides the set of reasonable guesses into two ranges of roughly the same size. If your guess is not correct, then I tell you whether it's too high or too low, and you can eliminate about half of the reasonable guesses. For example, if the current range of reasonable guesses is 26 to 80, you would guess the halfway point, (26 + 80) / 2, or 53. If I then tell you that 53 is too high, you can eliminate all numbers from 53 to 80, leaving 26 to 52 as the new range of reasonable guesses, halving the size of the range.



For the guessing game, we can keep track of the set of reasonable guesses using a few variables. Let the variable *min* be the current minimum reasonable guess for this round, and let the variable *max* be the current maximum reasonable guess. The input to the problem is the number *n*, the highest possible number that your opponent is thinking of. We assume that the lowest possible number is one, but it would be easy to modify the algorithm to take the lowest possible number as a second input.

Here's a step-by-step description of using binary search to play the guessing game:

1. Let min = 1 and max = n.

2. Guess the average of max and min rounded down so that it is an integer.

3. If you guessed the number, stop. You found it!

4. If the guess was too low, set min to be one larger than the guess.

5. If the guess was too high, set max to be one smaller than the guess

6. Go back to step two.

**3. Implementing binary search of an array**

var primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97];

Suppose we want to know whether the number 67 is prime. If 67 is in the array, then it's prime.

We might also want to know how many primes are smaller than 67. If we find the position of the number 67 in the array, we can use that to figure out how many smaller primes exist.

Once we know that the prime number 67 is at index 18, we can identify that it is a prime. We can also quickly identify that there are 18 elements which come before 67 in the array, meaning that there are 18 prime numbers smaller than 67.

Did you see how many steps that took? A binary search might be more efficient. Because the array primes contains 25 numbers, the indices into the array range from 0 to 24. Using our pseudocode from before, we start by letting min = 0 and max = 24. The first guess in the binary search would therefore be at index 12 (which is (0 + 24) / 2). Is primes[12] equal to 67? No, primes[12] is 41.

텍스트, 시계이(가) 표시된 사진

자동 생성된 설명

Is the index we are looking for higher or lower than 12? Since the values in the array are in increasing order, and 41 < 67, the value 67 should be to the right of index 12. In other words, the index we are trying to guess should be greater than 12. We update the value of min to 12 + 1, or 13, and we leave max unchanged at 24.

텍스트, 시계이(가) 표시된 사진

자동 생성된 설명

What's the next index to guess? The average of 13 and 24 is 18.5, which we round down to 18, since an index into an array must be an integer. We find that primes[18] is 67.

G:\내 드라이브\Programming\myProjects\khan-academy\computer-science\algorithm\binary-search.js텍스트, 시계이(가) 표시된 사진

자동 생성된 설명

The binary search algorithm stops at this point, since it has found the answer. It took only two guesses, instead of the 19 guesses that linear search would have taken. You can step through that again in the visualization below:

**Pseudocode**

Here's the pseudocode for binary search, modified for searching in an array. The inputs are the array, which we call *array* the number *n* of elements in *array* and *target*, the number being searched for. The output is the index in *array* of *target*

1. Let *min* = 0 and *max* = n-1.

2. Compute guess as the average of *max* and *min*, rounded down (so that it is an integer).

3. If *array[guess]* equals *target*, then stop. You found it! Return guess.

4. If the guess was too low, that is, *array[guess]* < *target*, then set *min* = guess + 1.

5. Otherwise, the guess was too high. Set *max* = guess - 1.

6. Go back to step 2.

**Implementing pseudocode**

How would we turn that pseudocode into a JavaScript program? We should create a function, because we're writing code that accepts an input and returns an output, and we want that code to be reusable for different inputs. The parameters to the function—let's call it *binarySearch*— will be the array and target value, and the return value of the function will be the index of the location where the target value was found.

Now let's go into the body of the function, and decide how to implement that. Step 6 says to go back to step 2. That sounds like a loop. Should it be a for-loop or a while-loop? If you really wanted to use a for-loop, you could, but the indices guessed by binary search don't go in the sequential order that a for-loop makes convenient. First we might guess the index 12, and then 18, based on some computations. So a while-loop is the better choice.

There's also an important step missing in that pseudocode that didn't matter for the guessing game, but does matter for the binary search of an array. What should happen if the number you are looking for is not in the array? Let's start by figuring out what index the *binarySearch* function should return in this case. It should be a number that cannot be a legal index into the array. We'll use -1, since that cannot be a legal index into any array. (Actually, any negative number would do.)

The target number isn't in the array if there are no possible guesses left. In our example, suppose that we're searching for the target number 10 in the primes array. If it were there, 10 would be between the values 7 and 11, which are at indices 3 and 4. If you trace out the index values for min and max as the binarySearch function executes, you would find that they eventually get to the point where min equals 3 and max equals 4. The guess is then index 3 (since (3 + 4) / 2 equals 3.5, and we round down), and primes[3] is less than 10, so that min becomes 4. With both min and max equaling 4, the guess must be index 4, and primes[4] is greater than 10. Now max becomes 3. What does it mean for min to equal 4 and max to equal 3? It means that the only possible guesses are at least 4 and at most 3. There are no such numbers! At this point, we can conclude that the target number, 10, is not in the primes array, and the binarySearch function would return -1. In general, once max becomes strictly less than min, we know that the target number is not in the sorted array. Here is modified pseudocode for binary search that handles the case in which the target number is not present:

1. Let *min* = 0 and *max* = n-1.

2. If *max* < *min*, then stop: *target* is not present in *array*. *Return -1*.

3. Compute guess as the average of *max* and *min*, rounded down (so that it is an integer).

4. If *array[guess]* equals *target*, then stop. You found it! Return guess.

5. If the guess was too low, that is, *array[guess]* < *target*, then set *min* = guess + 1.

6. Otherwise, the guess was too high. Set *max* = guess - 1.

7. Go back to step 2.

**4. Running time of binary search**

Let's see how to analyze the maximum number of guesses that binary search makes.

If we start with an array of length 8, then incorrect guesses reduce the size of the reasonable portion to 4, then 2, and then 1. Once the reasonable portion contains just one element, no further guesses occur; the guess for the 1-element portion is either correct or incorrect, and we're done. So with an array of length 8, binary search needs at most four guesses.

What do you think would happen with an array of 16 elements? If you said that the first guess would eliminate at least 8 elements, so that at most 8 remain, you're getting the picture. So with 16 elements, we need at most five guesses.

By now, you're probably seeing the pattern. Every time we double the size of the array, we need at most one more guess. Suppose we need at most *m* guesses for an array of length *n*. Then, for an array of length *2n*, the first guess cuts the reasonable portion of the array down to size *n*, and at most *m* guesses finish up, giving us a total of at most *m+1* guesses.

Fortunately, there's a mathematical function that means the same thing as the number of times we repeatedly halve, starting at *n*, until we get the value 1: **the base-2 logarithm of** *n*. That's most often written as but you may also see it written as in computer science writings.

Here's a table showing the base-2 logarithms of various values of *n*:

|  |  |
| --- | --- |
| n |  |
| 1 | 0 |
| 2 | 1 |
| 4 | 2 |
| 8 | 3 |
| 16 | 4 |
| 32 | 5 |
| 64 | 6 |
| 128 | 7 |
| 256 | 8 |
| 512 | 9 |
| 1024 | 10 |
| 1,048,576 | 20 |
| 2,097,152 | 21 |

That makes it easy to calculate the runtime of a binary search algorithm on an *n* that's exactly a power of 2. If *n* is 128, binary search will require at most 8 ( + 1) guesses.

What if *n* isn't a power of 2? In that case, we can look at the closest lower power of 2. For an array whose length is 1000, the closest lower power of 2 is 512, which equals . We can thus estimate that is a number greater than 9 and less than 10, or use a calculator to see that its about 9.97. Adding one to that yields about 10.97. In the case of a decimal number, we round down to find the actual number of guesses. Therefore, for a 1000-element array, binary search would require at most 10 guesses.