Bonus Material - Nodle's Notes on How To Binary Search

We've seen binary search during the first few lectures. And we've talked a little bit about how bug prone it is. In this note, I am going to share with you a slightly helpful perspective on array indexing, how to view indices, and how that helps with binary search.

These notes are bonus material that I made. Because this is more of a programming methodology thing, rather than algorithms. But at this level, one could see this as both relevant to both programming **and** algorithms. So you may find these to be extremely helpful, not just for CS2040S, but for CS3230, or even your personal programming experience or CS2030S. Those who've attended my lecture on Week 1 will know, I've failed binary search during an interview before. Perhaps it's best my mistakes are not repeated by my students.

For CS2040S, we do expect you to know how to binary search, that is an explicit learning outcome. The goal of these notes are to help you write bug-free binary searches. And to generalise beyond searching in an array.

Here's a rough overview:

1. A new perspective on indices
2. Basic binary search
3. Generalised binary search
4. An example of how you might apply this kind of binary search

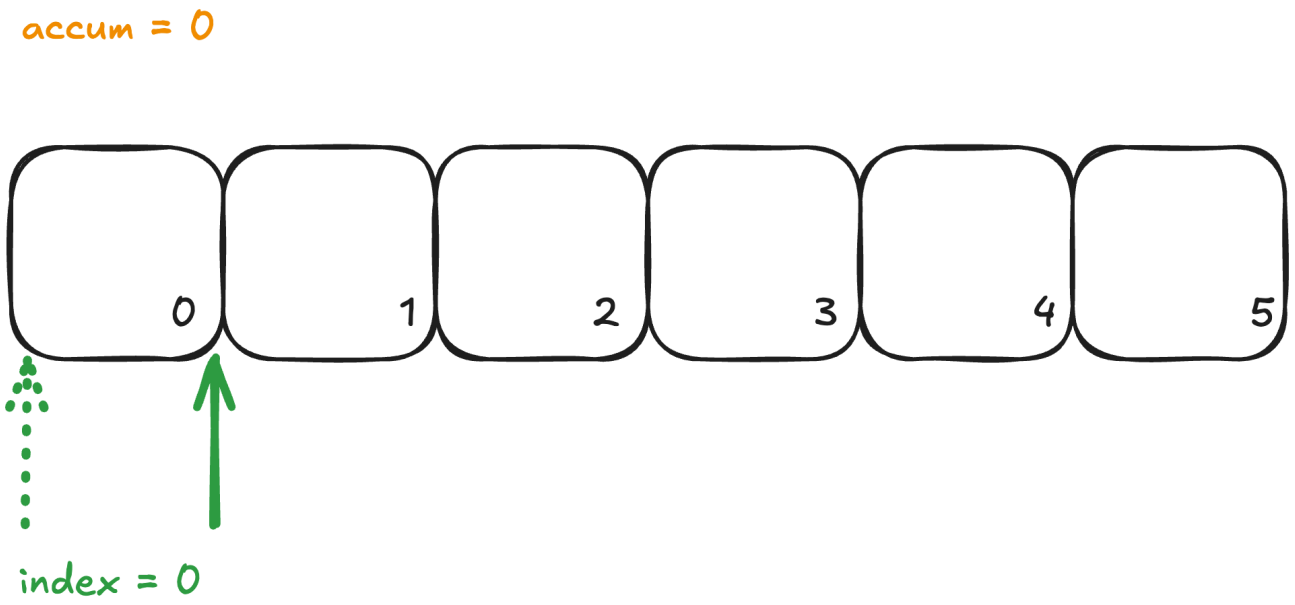
Note: This is bonus material. For the sake of ease of exposition, I'll be using C++ in the examples. Mostly because I want the indices to be very explicitly shown (not like in Python). But full disclaimer: Experienced C++ programmers will probably notice that I am using a very basic subset of the language. No, I will not be using the advanced language features. The goal here is to show a new concept in the best way possible, not teach pragmatic/real-life C++.

A new perspective on indices

Some of you might already be used to arrays, some of you might not. But we can probably all understand how something like this works:

```
int total_sum(int[] arr, size_t arr_length) {
int accum = 0;

for(size_t index = 0; index < arr_length; ++index){
        accum += arr[index];
}
```

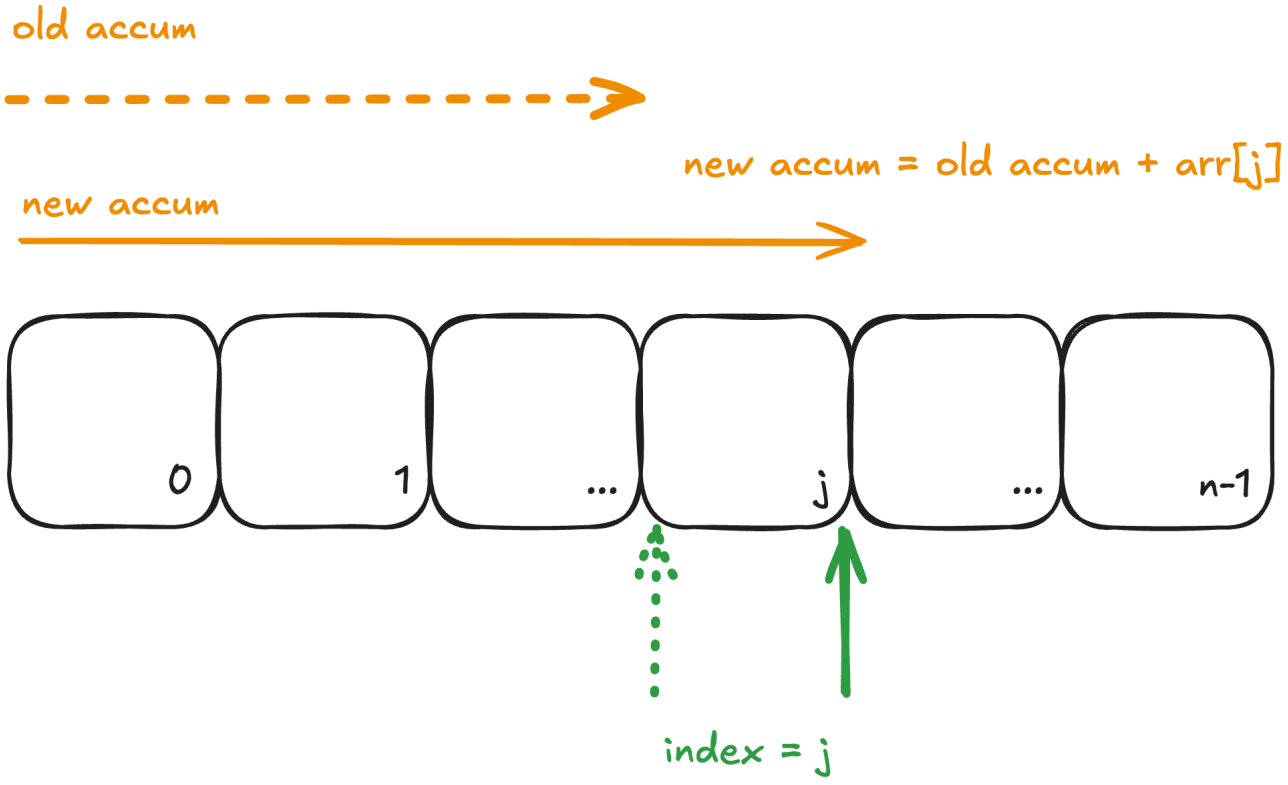
Here, the variable `index` is... well aptly named. It's our index into the array. Let's think about correctness for a little bit. I'd argue, that we can actually relate `accum` and `index`. How?



Before the very first iteration, `index` is at 0. And `accum` is also set to `0`. Think about how this represents the sum of the array up until the **dotted green line** in the array.

After the very first iteration, and before the second, `index` is at 0. And `accum` is basically `0 + arr[0]`. Think about how this represents, the sum of the array up until the **solid green line** in the array.

So in general, it looks something like this:



Before the $(j+1)^{th}$ **iteration**, and the `accum` variable holds the sum of the values values `arr[0]` up until `arr[j-1]`. And after that, by definition of our algorithm/program, `accum` is updated to hold the sum of `arr[0]` up until `arr[j]`.

> So what's this new perspective on indexing? This is just a straightforward for-loop.

Take note again (pictorially) of how the variable `index` was related to `accum`. It was pointing to a position **in-between** elements. It wasn't pointing at an element specifically, it was pointing at a **boundary/border** between two array elements.

> Okay... so what's so special about that?

Just bear in mind that indices can point in-between elements for now. Yes I know the value is a concrete value, and you can use it to get some element from the array. After all, something like `arr[0]` gives me the first element of the array, I know. But **conceptually speaking** we can look as an index as **representing** a border between two elements.

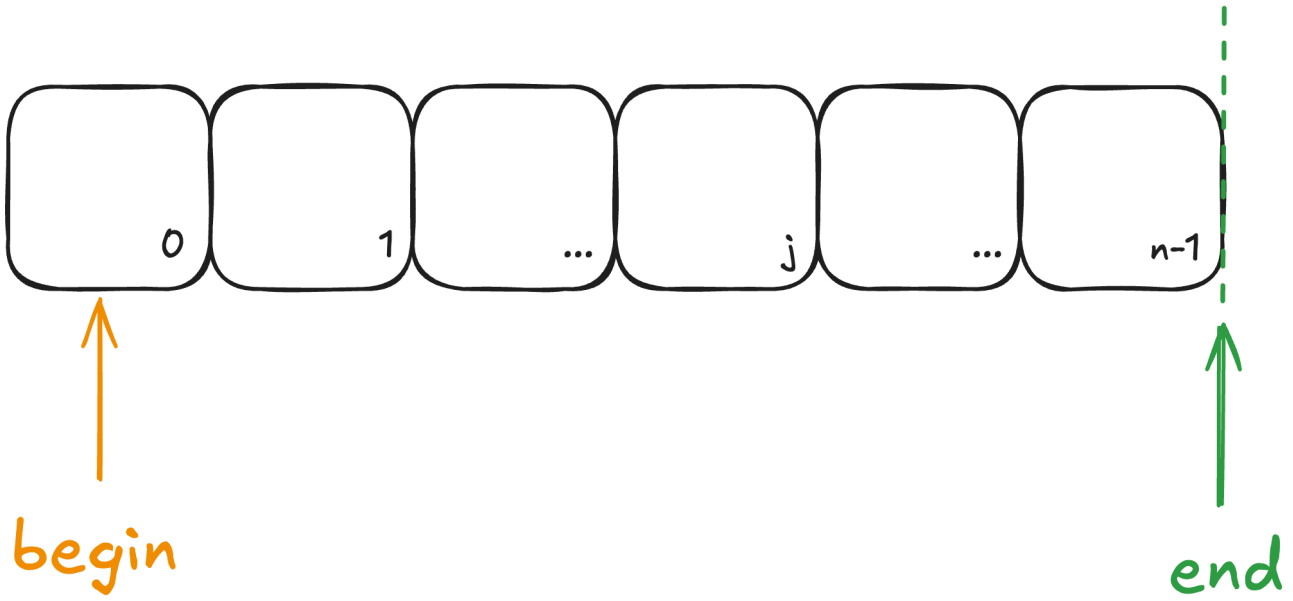
Basic Binary Search

Okay, let's look at basic binary search again. This time around, to fit my narrative, I'll give the following code (which may differ from the slides). Bear in mind there's more than one way to write the code. **How you interpret the indices changes how you use the indices, which changes how you write the code**.

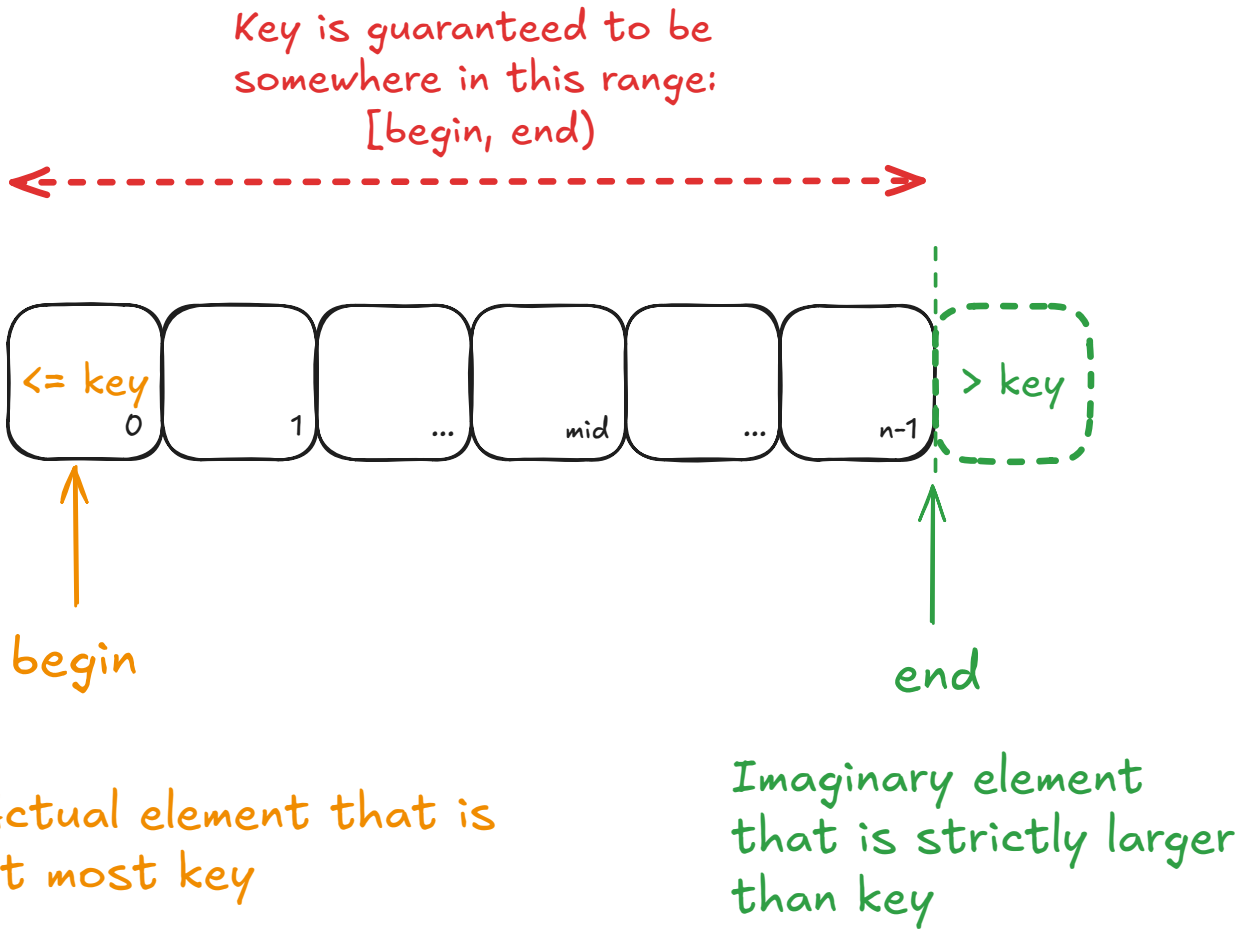
```
// assumes: arr is sorted
std::optional<size_t> binary_search(int arr[], size_t arr_length, int key)
{
        size_t n = arr_length; // just so I can use n in the drawings

        if(arr[0] > key) {
                // return a null value to if it doesn't exist
                return std::nullopt;
        }

        if(arr[n - 1] < key) {
                // return a null value to if it doesn't exist
                return std::nullopt;
        }

        // After this line, guaranteed arr[0] <= key
        // also guaranteed arr[n - 1] >= key

        size_t begin = 0, end = n; // (1)

        while(begin + 1 < end) { // (2)
            size_t mid = (end - begin) / 2 + begin;
            if(arr[mid] <= key) { // (3)
                    begin = mid;
            } else { // (4)
                    end = mid;
            }
        }


        // if value at begin is key return it, otherwise indicate
        // it doesn't exist in the array
        return (arr[begin] == key) ? begin : std::nullopt;
}
```

(Again this isn't the only way to write binary search, but the easiest for me to explain.)

So pictorially, I want you to think of the binary search at point (1) in the code as basically doing the following:

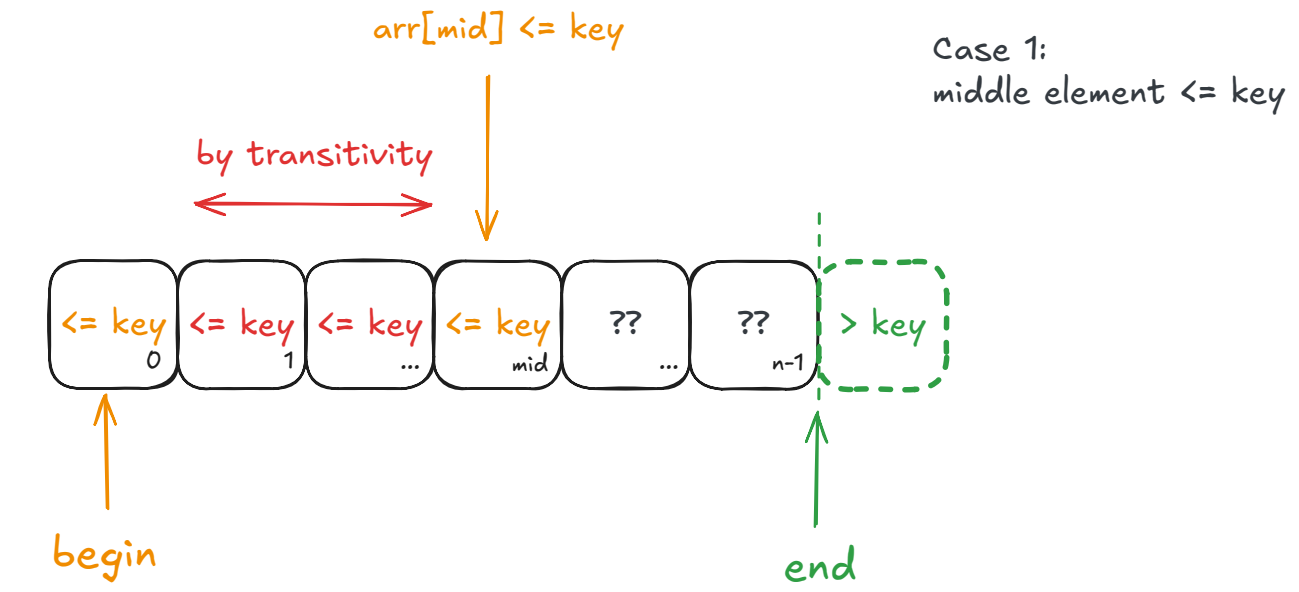


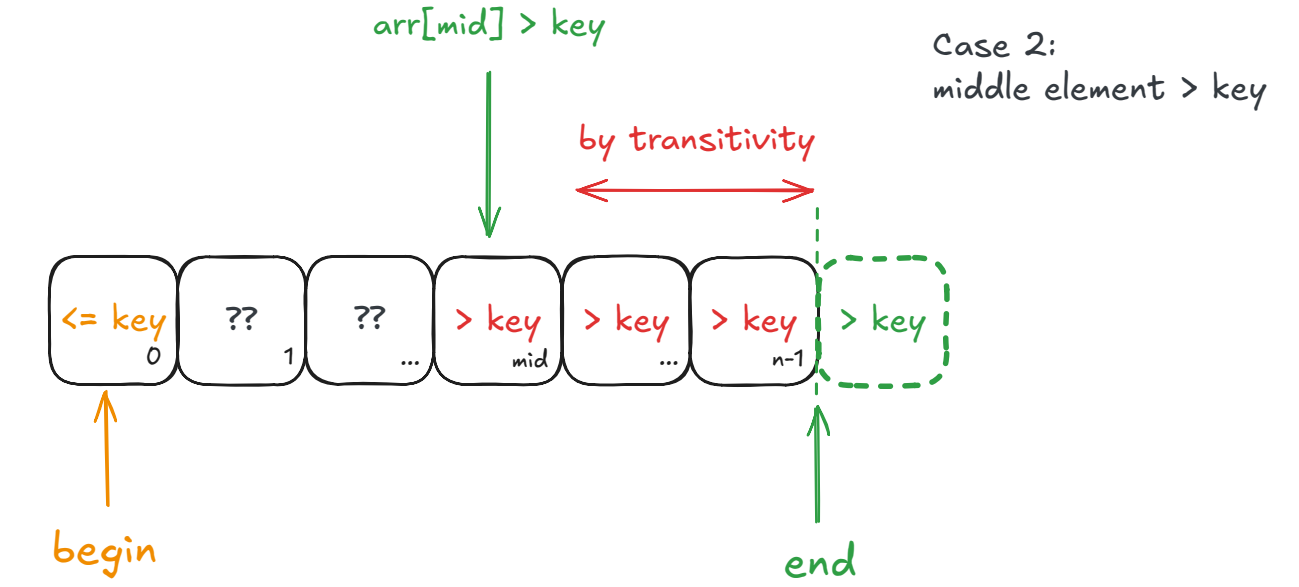
`begin` is an index that points **at** a position in the array. `end` is an index that points **in between** two positions in the array. You can also think of this as a *half open* interval. I.e. We wish to search for the key, in range `[begin, end)`. (Hello CS1231S notation!)



If you look at what the indices are doing, `begin` is pointing to an element we know is `<= key`. `end` in our implementation is pointing past the end. One possible (but not unique) interpretation is that it's bordering elements that are `> key`. After all, we know `arr[n - 1] >= key` (but we don't know if it's equal).

Okay, so two possible cases: either `arr[mid] <= key` or `arr[mid] > key`:

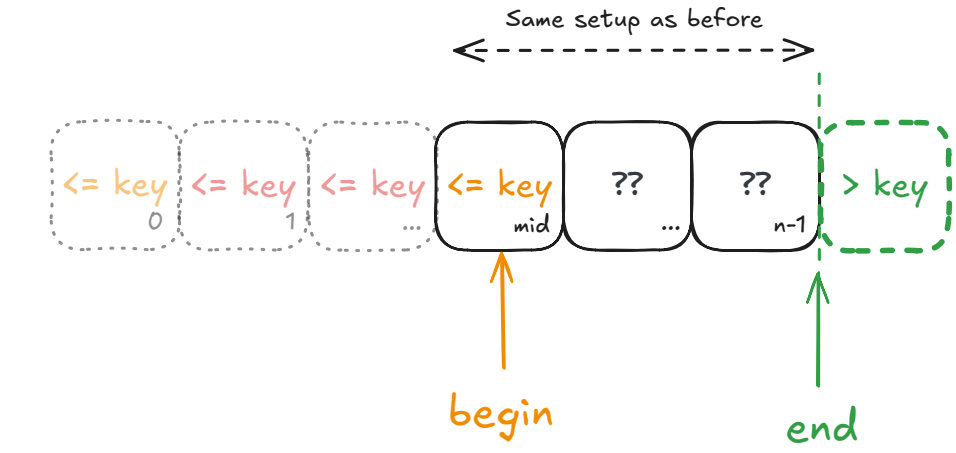


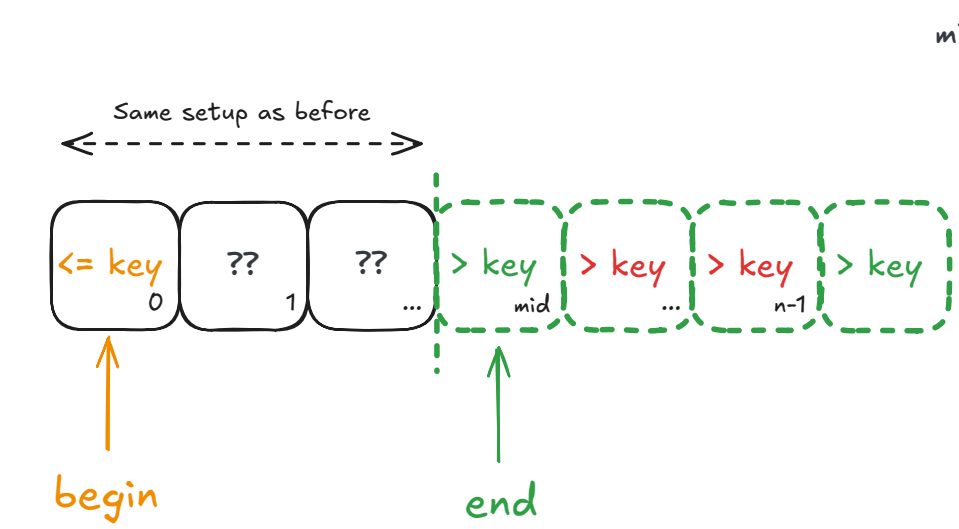


In the first case, **because** we tested the middle element, we know by transitivity everything to its left is also `<= key`. Based on this, how should we update `begin`? Set it to `mid`!

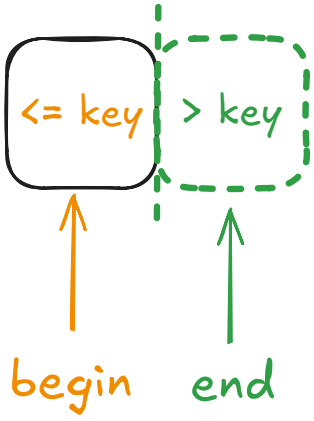
Similarly, in the second case, **because** we tested the middle element, we know by transitivity everything to its right is also `> key`. So, in the second case, we should set `end` to `mid`

See how it maintains it?





Okay so our update looks fine, so what is our termination condition? We keep going when `begin + 1 < end`. Let's think about what this means. When `begin + 1 == end`, the array looks like this:



Which is to say that there is a single element left for us to check, and we do so on the very final line of our algorithm.

Are we done? Not quite, we need to also argue that we will always work towards the case where $begin + 1 == end$ (otherwise, our algorithm never terminates!). To do this, we can try to argue that at every iteration, the difference between `begin` and `end` shrinks by at least 1 each time.

Let $begin, end$ be the values at the beginning of the loop iteration. Let $begin', end'$ be the values after the loop iteration has ended. We want to argue that $end' - begin' < end - begin$. (This means that we always move towards the case where $begin + 1 == end$.)

Pseudo-proof:

1. Assume $begin + 1 < end$ (that's why we're doing an iteration in the first place)
2. Let $mid = \lfloor \frac{(end - begin)}{2} \rfloor + begin$
3. There are 2 cases, either we update $begin' = mid$, or $end' = mid$.
4. Case 1: We update $begin' = mid$
    1. Since we know $end - begin \geq 2$: $\lfloor \frac{(end - begin)}{2} \rfloor \geq 1$. So $begin' \geq begin + 1$.
    2. Meanwhile $end' = end$.
    3. So $end' - begin' = end - begin' \leq end - begin - 1 < end - begin$
5. Case 2: We update $end' = mid$
    1. $mid = \lfloor \frac{(end - begin)}{2} \rfloor + begin$
    2. Since $begin < end - 1$:
    $\lfloor \frac{(end - begin)}{2} \rfloor + begin \leq \frac{(end - begin)}{2} + begin = \frac{end + begin}{2} < end - \frac{1}{2}$.
    3. So $end' = mid < end - \frac{1}{2}$.
    4. Since end' is an integer, $end' < end$.
    5. Meanwhile $begin' = begin$.
    6. So $end' - begin' = end' - begin < end - begin$
6. In both cases, $end' - begin' < end - begin$.

Generalised Binary Search

Okay, moving on, let's get back to the "programming methods" part of this. I believe you probably have some familiarity with taking in functions as inputs (from CS1101S). But in case you did not come from CS1101S, here's a quick run-down: "We can take functions in as inputs, and call them."

So for example, we could have a function that applies some kind of binary operator on the first two elements of some list, like so:

```
def apply_binary_op(arr, operator):
        return operator(arr[0], arr[1])
```

Also I said I would use C++, so here's the same example in C++:

```
#include <functional>
int apply_binary_op(int[] arr,std::function<int(int, int)> binary_op) {
        return binary_op(arr[0], arr[1]);
}
```

CS1101S also has great notes on this, if you need a refresher: Functions as First Class Citizens.

Okay, so why am I bringing this up?

Because now we're going to see a whole new world where we aren't just looking for whether or not a value is in an array (and where it is).

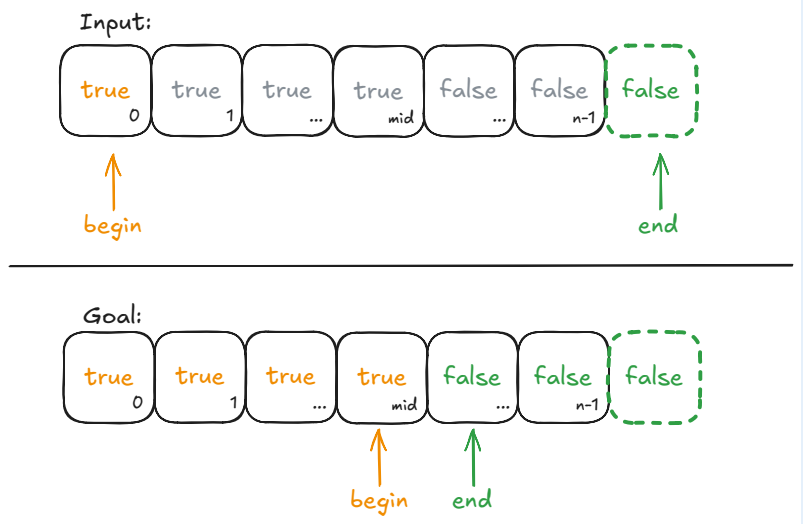
The idea is the following:

1. If the problem we're trying to solve has a function we can evaluate on a range of values.
2. If the function is "monotonic", then we can binary search.

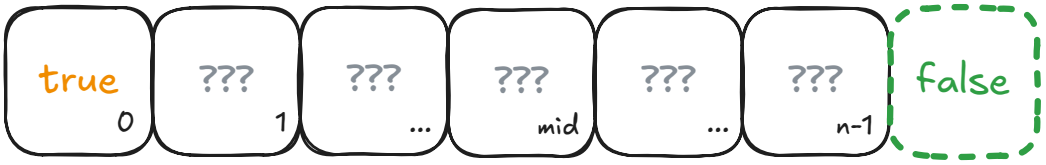
In doing so, if the range of possible values is n, then we only have to evaluate the function $O(\log(n))$ times.

Goal

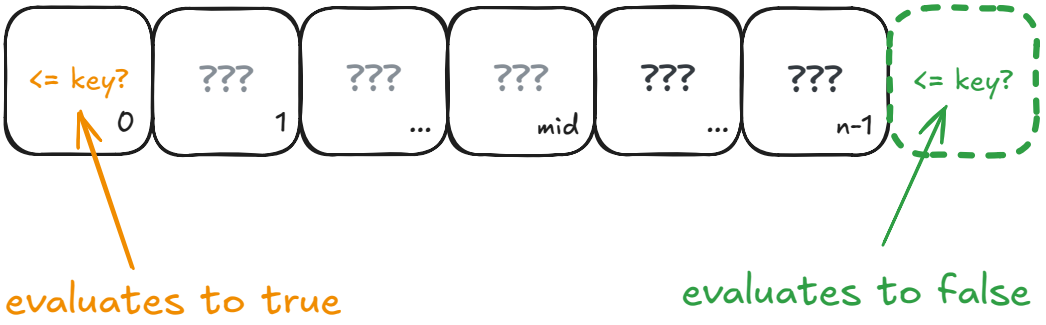
"Monotonic" here is in quotes because the idea is a little vague. But basically here's what we want to to in general. We're given as input a problem that can be represented as starting with a value that evaluates to `true`, and we will treat it so that the one value past the possible range evaluates to `false`. Then we basically want to find the highest index for which the value evaluates to `true`.



So in picture form, what I mean is the following:



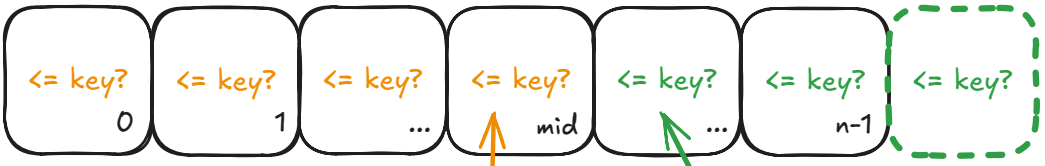
Example: When searching for a key



evaluates to true

evaluates to false

Goal: Find the point for which:



evaluates to true

evaluates to false

In general, we have a problem where we want to be assured the first element evaluates to true, and we're going to pretend that the last element evaluates to false. For example, when searching for a value `key`, we use a function `compare(value)` whose definition is:

```
auto create_predicate(int key){
        // return a closure
        return [=](int value) -> bool {
                return value <= key;
        };
}
```

So we *could* use a generalised binary search like so:

```
#include <functional>

using pred_t = std::function<bool(int)>;

size_t gen_bin_ser(int[] arr, size_t arr_len, pred_t predicate){
        size_t begin = 0, end = arr_len;
        while(begin + 1 < arr_len) {
                size_t mid = (end - begin) / 2 + begin;
                if(predicate(arr[mid])) {
                        begin = mid;
                } else {
                        end = mid;
                }
        }
        // at this point, we have the highest index such that
        // pred(arr[begin]) evaluates to true
        return begin;
}
```

> But wait! Given an array like `[1, 5, 10]`, and we made `predicate` where `key = 2`, then we would return `begin = 1`, even though `5` is not `2`.

Yep, you're right, I did a rug pull here and changed the program specification. This however, is a **very common pattern**. For example, if you look up std::lower_bound from the C++ STL library, you'll notice that they do that too. And they leave the burden of checking if it's the actual value up to the caller. Why do this though? Because, we want to free ourselves up to solving even more problems with binary search. In fact, let me show you how we should have written it:

```
#include <functional>
#include <cassert>

using pred_t = std::function<bool(int)>;

size_t gen_bin_ser(size_t begin, size_t end, pred_t predicate){
        assert(predicate(begin));
        while(begin + 1 < end) {
                size_t mid = (end - begin) / 2 + begin;
                if(predicate(mid)) {
                        begin = mid;
                } else {
                        end = mid;
                }
        }
        return begin;
}
```

Notice here that we call the function on `begin` and `end`. Which means we don't have access to the array anymore. We should actually pack that into the function `predicate` as well. For example:

```
auto create_predicate(int[] array, int key){
        // return a closure
        return [=](size_t index) -> bool {
                return array[index] <= key;
        };
}
```

So calling `create_predicate(arr, 5)` will return a function `pred()` for example, where let's say `arr = {0, 1, 20}`, is such that `pred(0)` and `pred(1)` will evaluate to `true`, but `pred(2)` will return `false`.

An example of how you might apply this kind of binary search

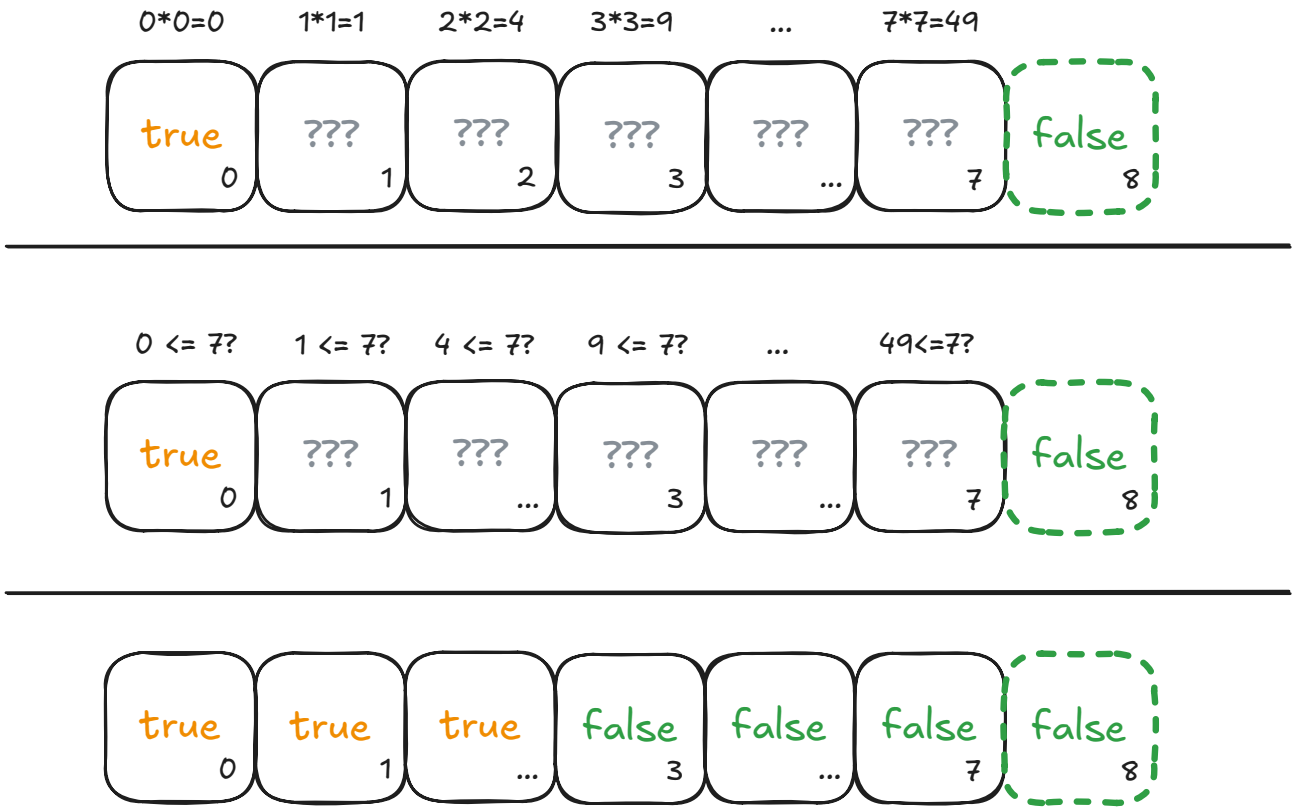
Let's say you need to find the largest integer that is not larger than the square root of some input number n. So here's a table:

n	desired output
0	0
1	1
2	1
3	1
4	2
5	2
6	2
7	2
8	2
9	3

Do you see where this is going? What happens if we made the following predicate?

```
auto create_predicate(int value_to_sqrt){
        // return a closure
        return [=](size_t test_value) -> bool {
                return test_value * test_value <= value_to_sqrt;
        };
}
```

So and for example, our input value is 7. Then we know we can the output answer is at least 0, and the output answer is at most 7, so `begin = 0`, `end = 7`. Pictorially, here's what would happen if we evaluated the result at each value in the range:



It's **monotonic**!

Which means, all we have to do, solve our problem like this:

```
#include <functional>
#include <cassert>

using pred_t = std::function<bool(int)>;

size_t gen_bin_ser(size_t begin, size_t end, pred_t predicate){
        assert(predicate(begin));
        while(begin + 1 < end) {
                size_t mid = (end - begin) / 2 + begin;
                if(predicate(mid)) {
                        begin = mid;
                } else {
                        end = mid;
                }
        }
        return begin;
}

auto create_predicate(int value_to_sqrt){
        // return a closure
        return [=](size_t test_value) -> bool {
                return test_value * test_value <= value_to_sqrt;
        };
}
```

```
int main(int argc, char **argv){
        int n = 7;
        int result = gen_bin_ser(0, n + 1, create_predicate(n));

        return 0;
}
```

Wait, one more thing. Why are we setting end to be $n + 1$? Recall that based on the setup we have, we **must have** that testing $end^2 \leq n$ evaluates to $false$. If we had instead called it like so: `gen_bin_ser(0, n, create_predicate(n));`, think about what happens when $n = 0$ or $n = 1$. When $n = 0$, while the function returns the correct value of 0, the range was actually empty to begin with. We didn't represent our solution space properly. On the other hand, for $n = 1$, the function returns value 0, because the only possible solution in the solution space of $[0, 1)$ is literally 0. So the algorithm thinks "okay I've hit base case, here's your solution". For it to be correct, we should start with $[0, n + 1)$ because n is a viable solution for when $n = 0$ or $n = 1$.

Do you see how this is a little bit more of a programming methodology thing? The concepts are similar though. We have pre-conditions, post-conditions, and invariants. Here, we still want to argue that the "answer" is within the range $[begin, end)$. And the post-condition is that the return value is the "highest value such that `predicate` on that value evaluates to $true$".