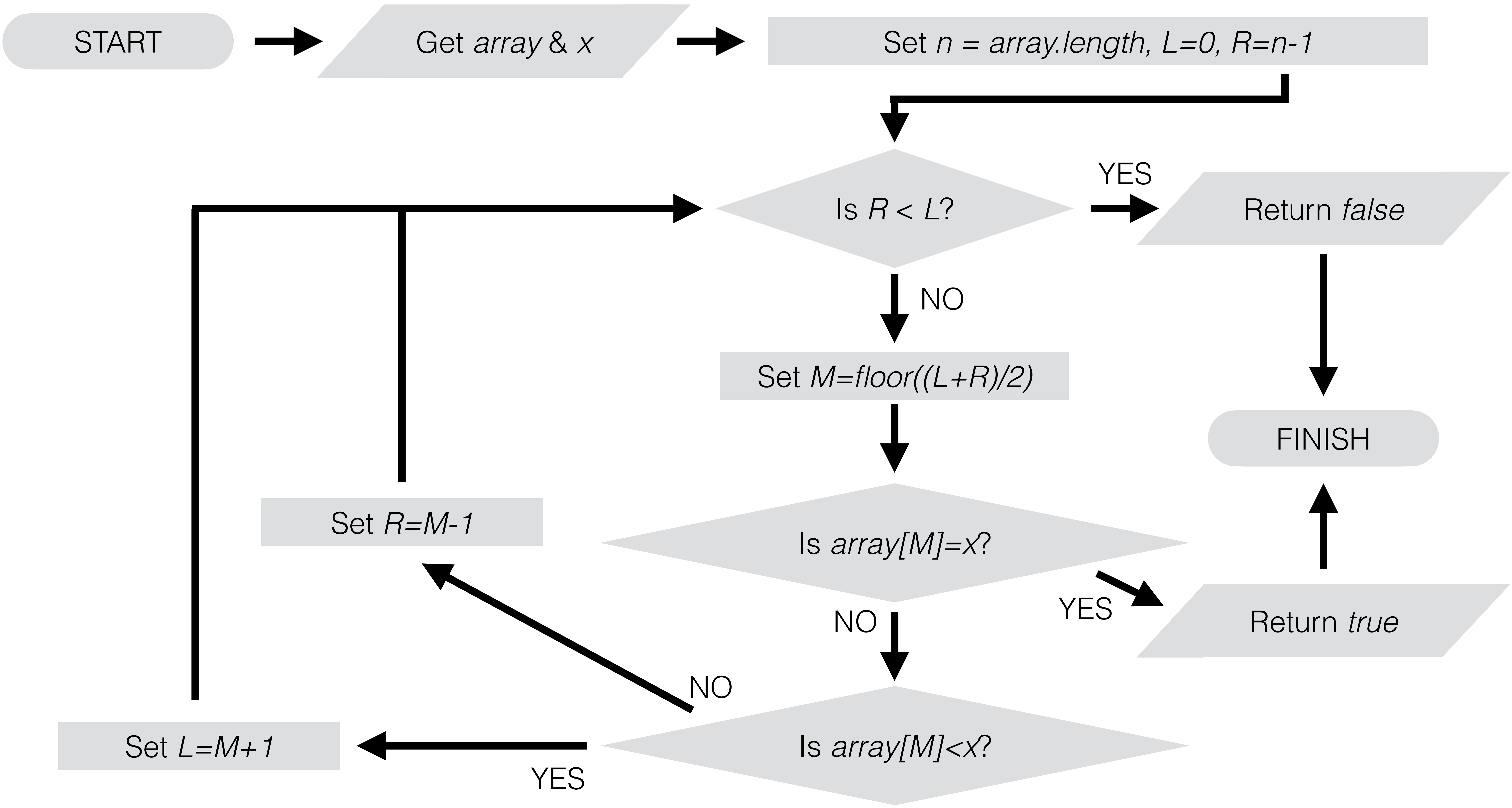
Admin

- Sixth quiz available
  - Fifth quiz deadline next Monday 15th March at 4pm

- Sudoku assignment
  - Cut-off date is **15th March 4pm**

- Primes assignment
  - Worksheet 6 available
  - Only involves programming tasks and submission of single js file
  - Help in next week's VCH with assignment
  - Deadline **15th March 4pm**
  - Cut-off date **29th March 4pm**

- New worksheet (not assessed) next Monday



Let’s implement this in JavaScript

```javascript
function binarySearch(array,x) {
    var n = array.length;
    var l = 0;
    var r = n - 1;
    var m;
    while (r >= l) {
        m = Math.floor((l+r)/2);
        if (array[m] === x) {
            return true;
        } else if (array[m] < x) {
            l = m + 1;
        } else {
            r = m - 1;
        }
    }
    return false;
}
```

Research task for Review Seminar

Until around 2006 nearly all Binary Search implementations in language libraries (e.g. Java) were broken

Find out why, and how you can have a better implementation of the algorithm

```javascript
function binarySearch(array,x) {
    var l = 0;
    var r = array.length - 1;
    if (r == 0) {
        return false;
    }

    while (r >= l) {
        var m = Math.floor((l + r) / 2);
        if (array[m] == x) {
            return true;
        } else if (array[m] < x) {
            l = m + 1;
        } else {
            r = m - 1;
        }
    }

    return false;

}
```

- Array indices are integers in Java
- For integer data type in Java, can only use 32 bits
- If l+r is 2^{32} (or greater) then number cannot be stored
- Get errors in Java and C
- JavaScript stores up to 2^{53}-1
- But we could still improve our code

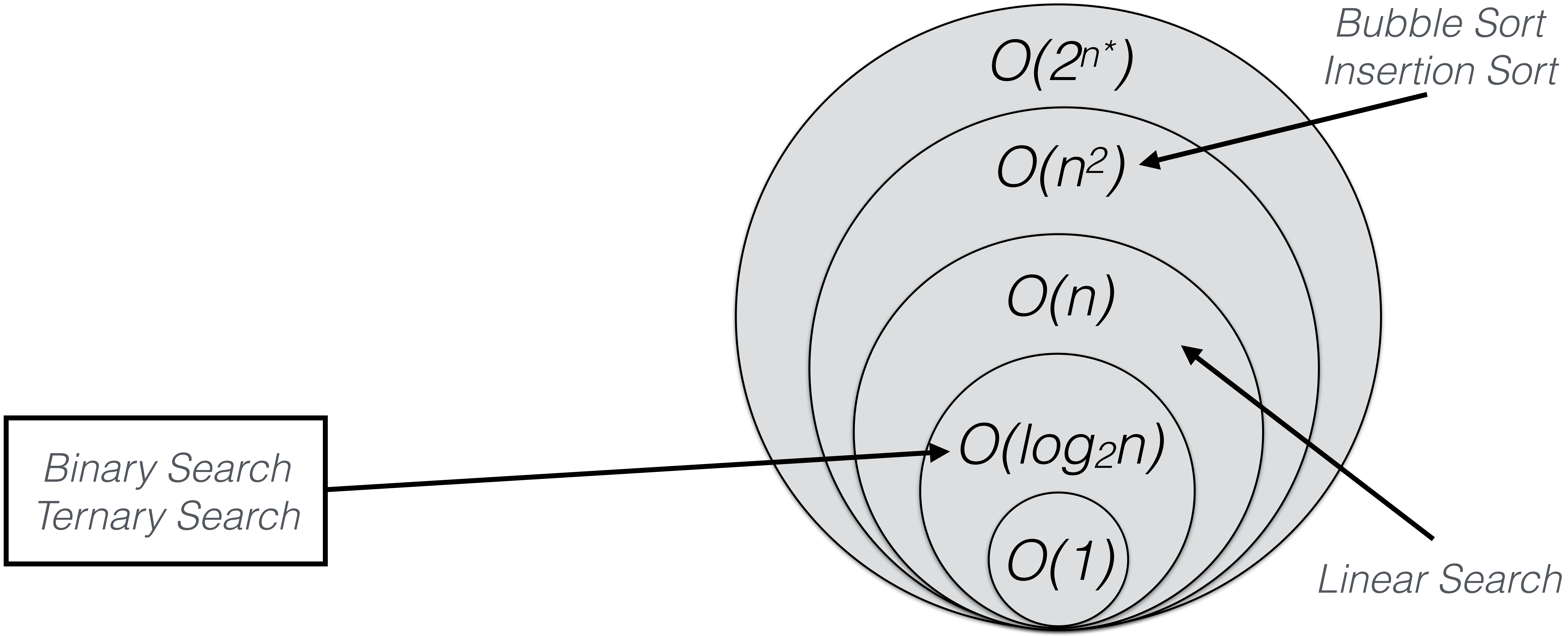
```
function binarySearch(array,x) {
    var l = 0;
    var r = array.length - 1;
    if (r == 0) {
        return false;
    }

    while (r >= l) {
        var m = Math.floor(l + ((r - l) / 2));
        if (array[m] == x) {
            return true;
        } else if (array[m] < x) {
            l = m + 1;
        } else {
            r = m - 1;
        }
    }

    return false;

}
```

- This does the same thing mathematically
- r - l/2 is less than r and l
- Arithmetic on smaller numbers



$O(2^{n^*})$
$O(n^2)$
$O(n)$
$O(\log_2 n)$
$O(1)$
Bubble Sort
Insertion Sort
Linear Search
Binary Search
Ternary Search

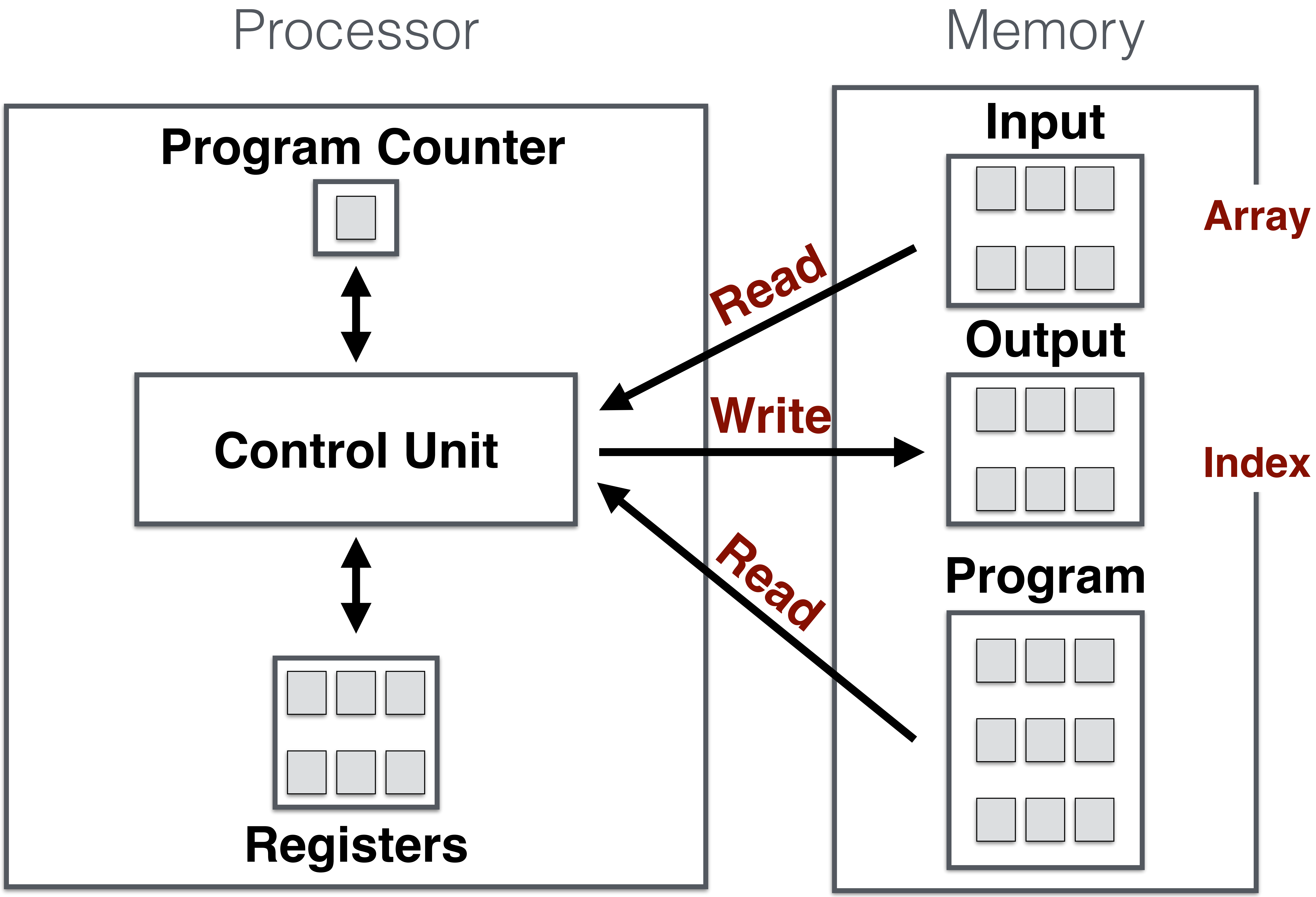
Binary Search

Can we do better than $O(\log n)$ worst-case time complexity?

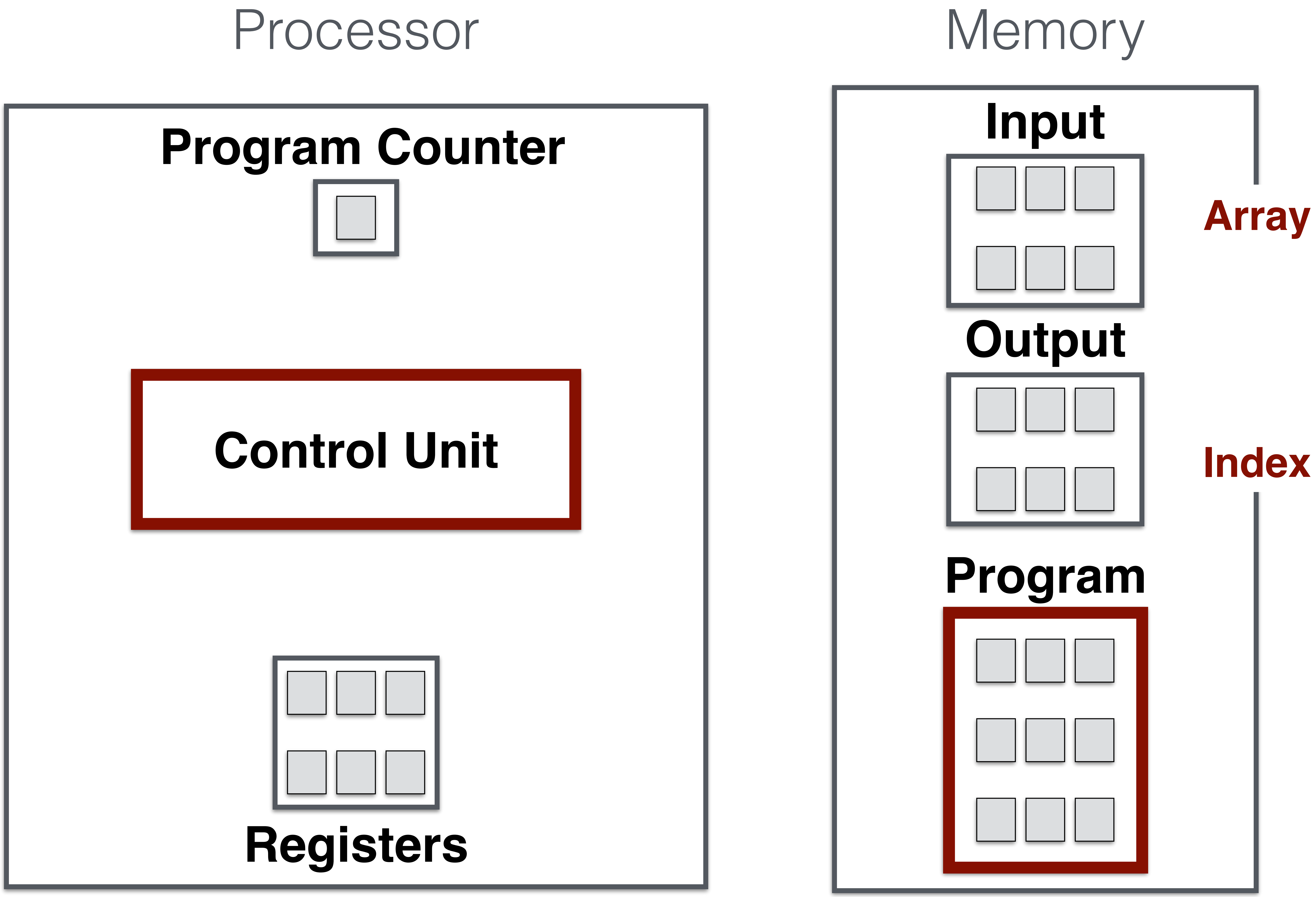
Assume we do not know values in elements

Consider version where we give index of found value as output

Random Access Machine



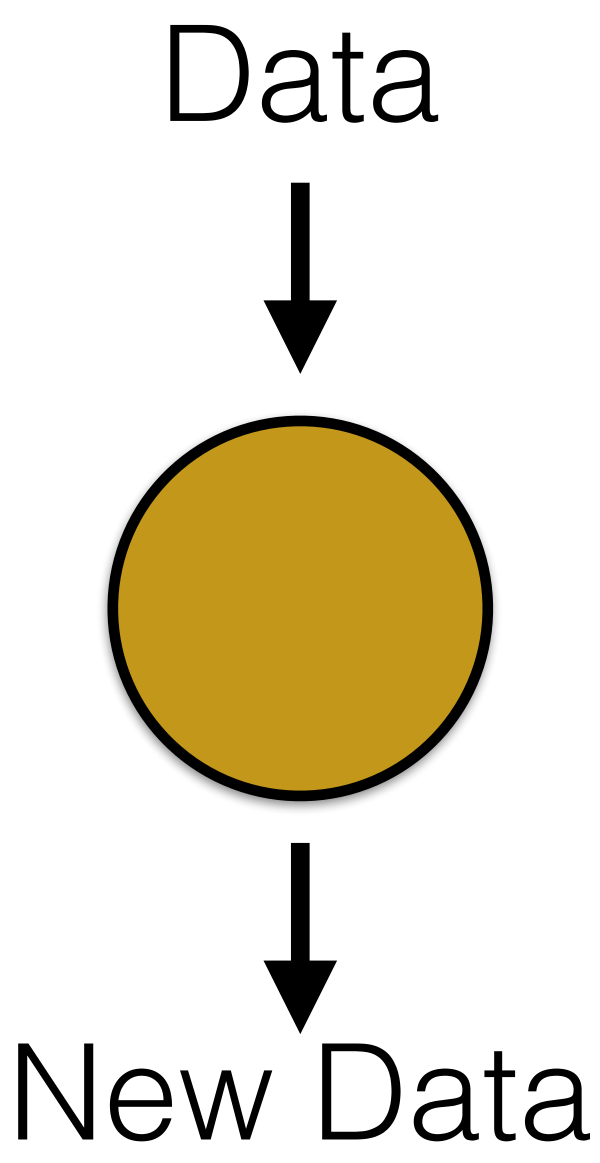
Random Access Machine

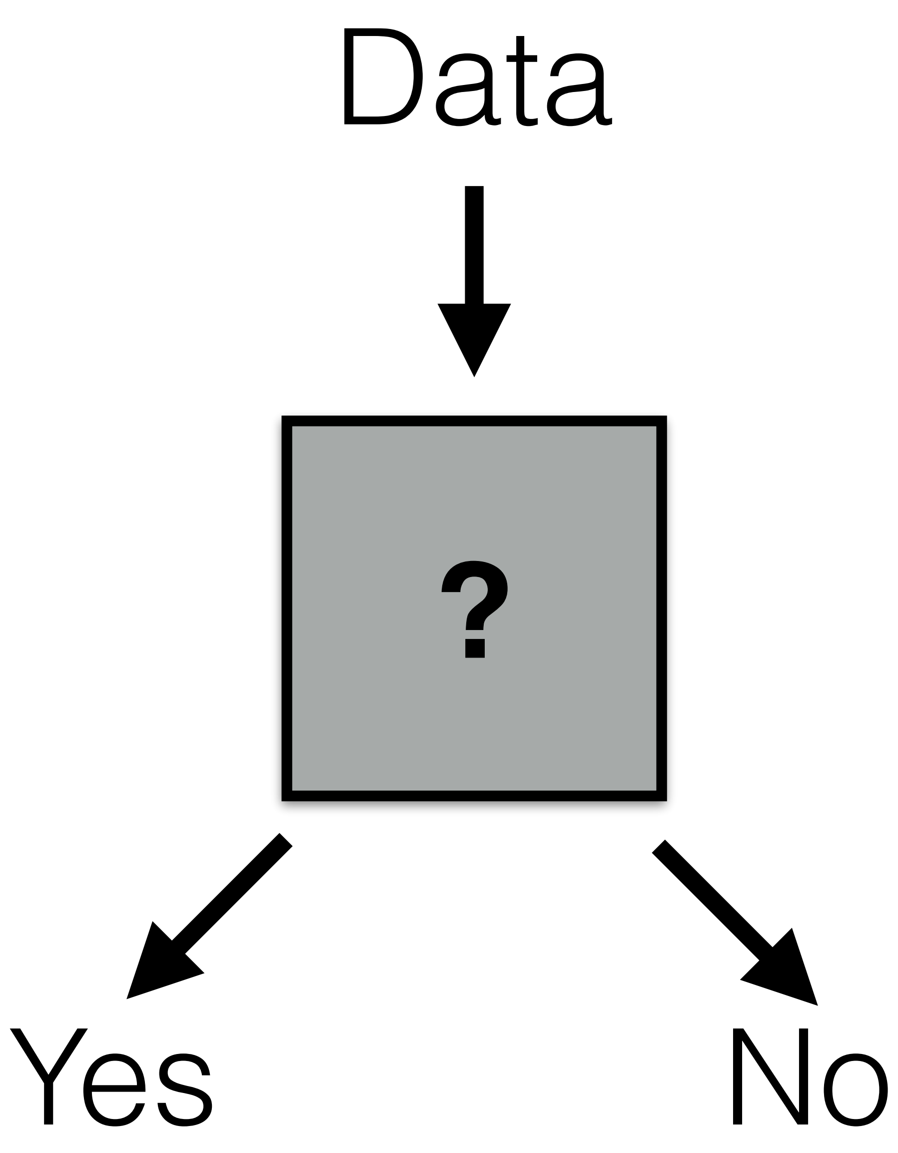


Algorithm encoded into the program will have decisions and basic actions

Basic actions have one outcome

Decisions can have two outcomes: yes and no

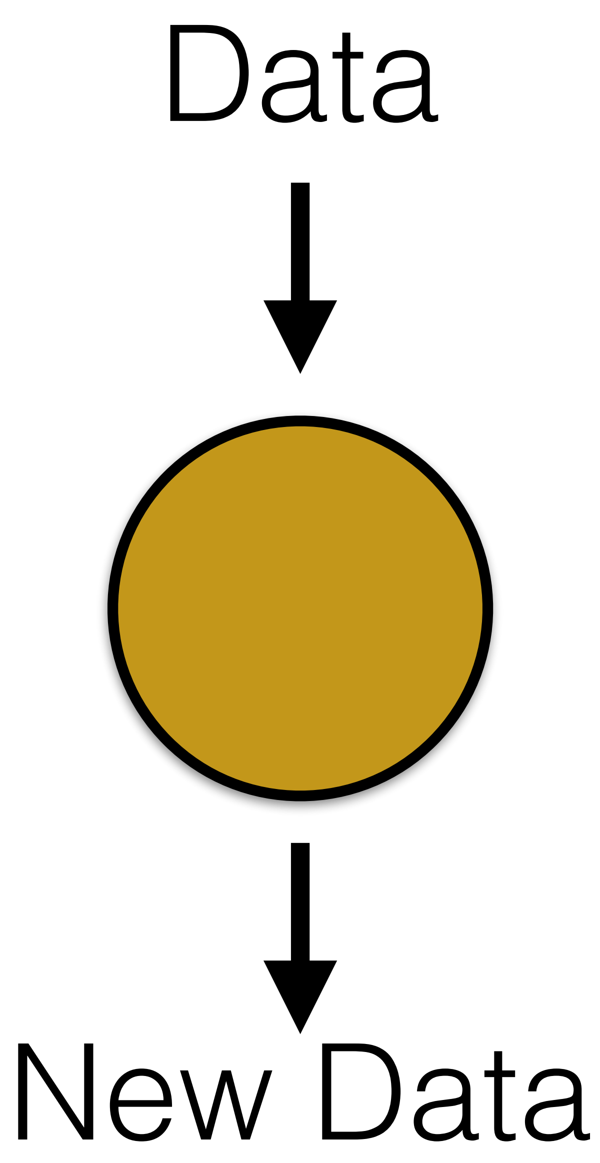


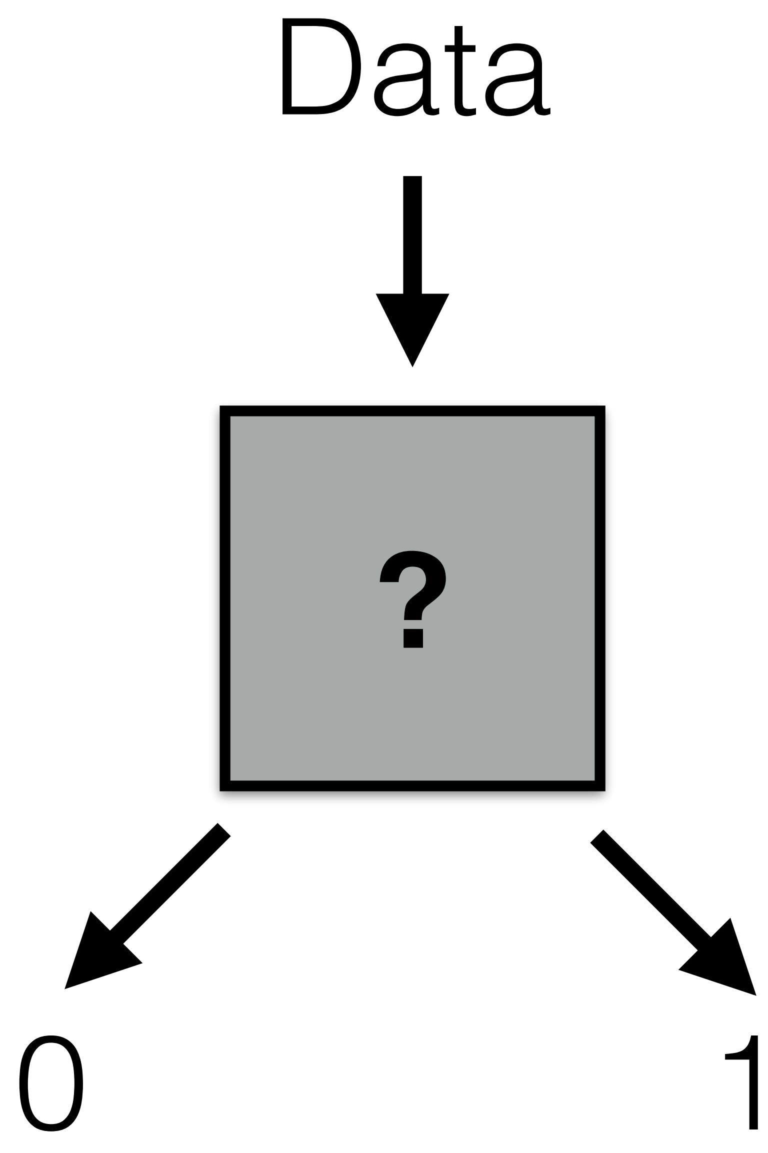


Algorithm encoded into the program will have decisions and basic actions

Basic actions have one outcome

Decisions can have two outcomes: yes and no

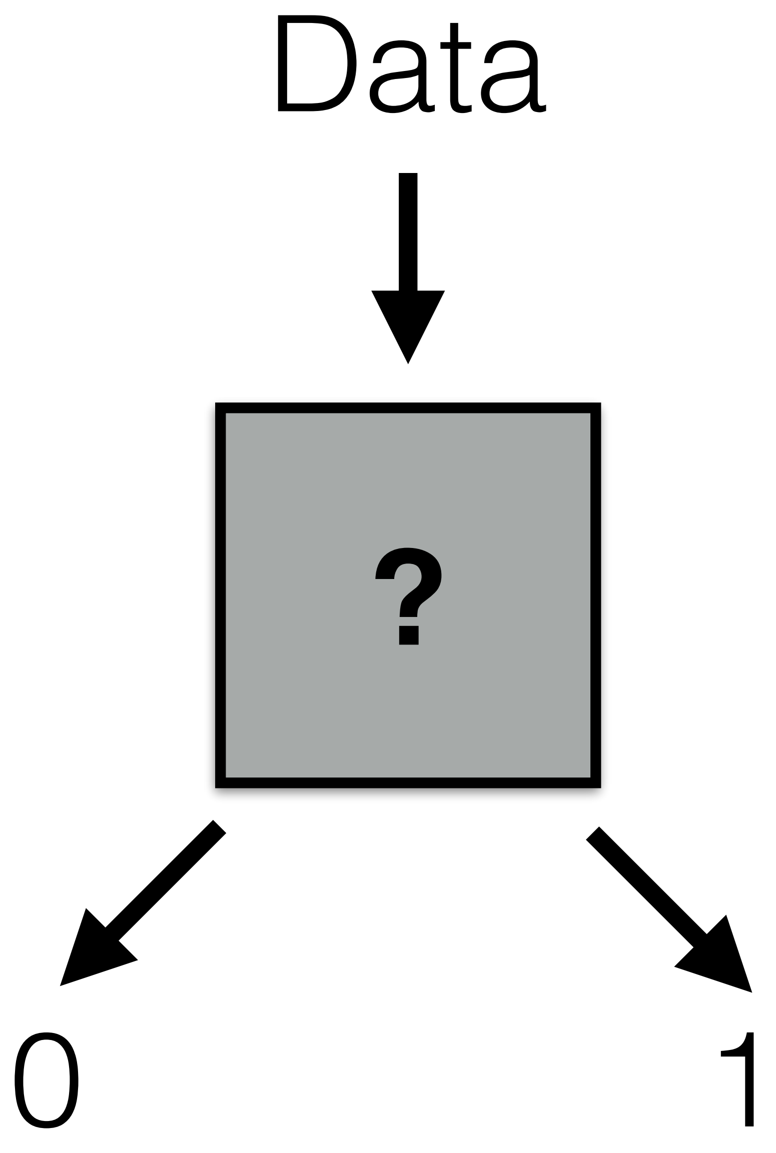




Turn them into bits

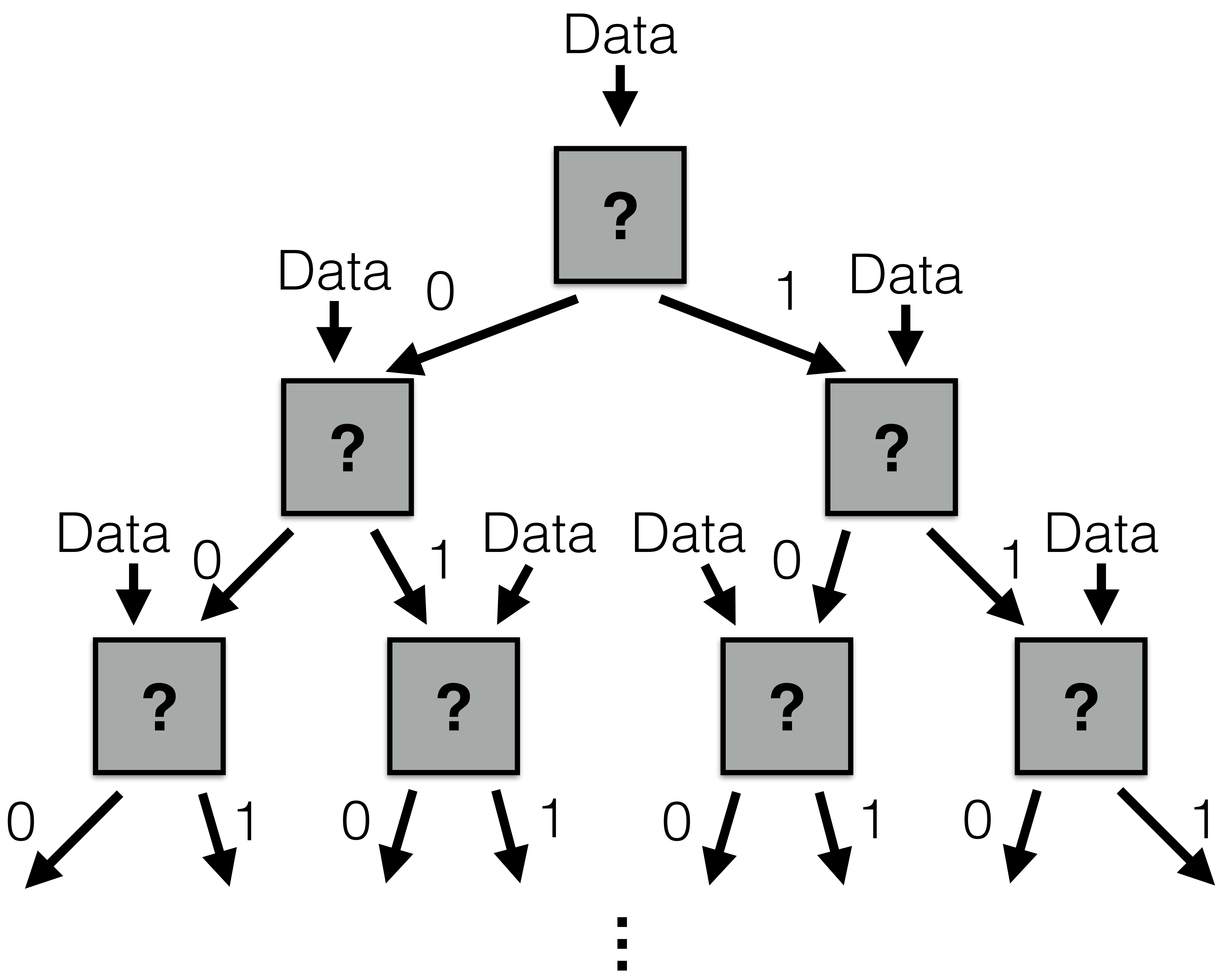
We want to know the **minimum number** of these operations required for the task

Let's start by ignoring the basic operations: will only add more operations

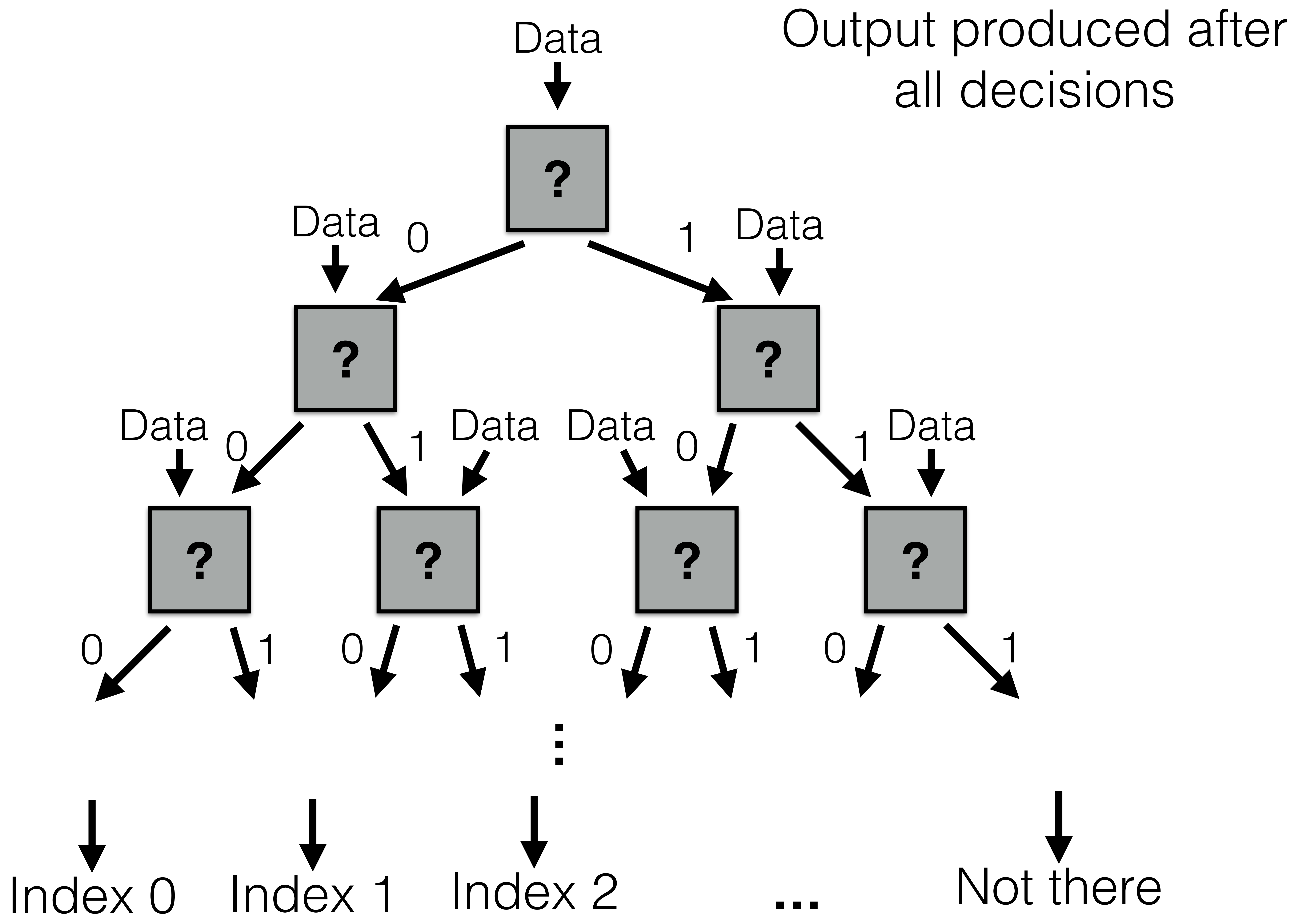


Any possible algorithm we can concoct has the following general structure

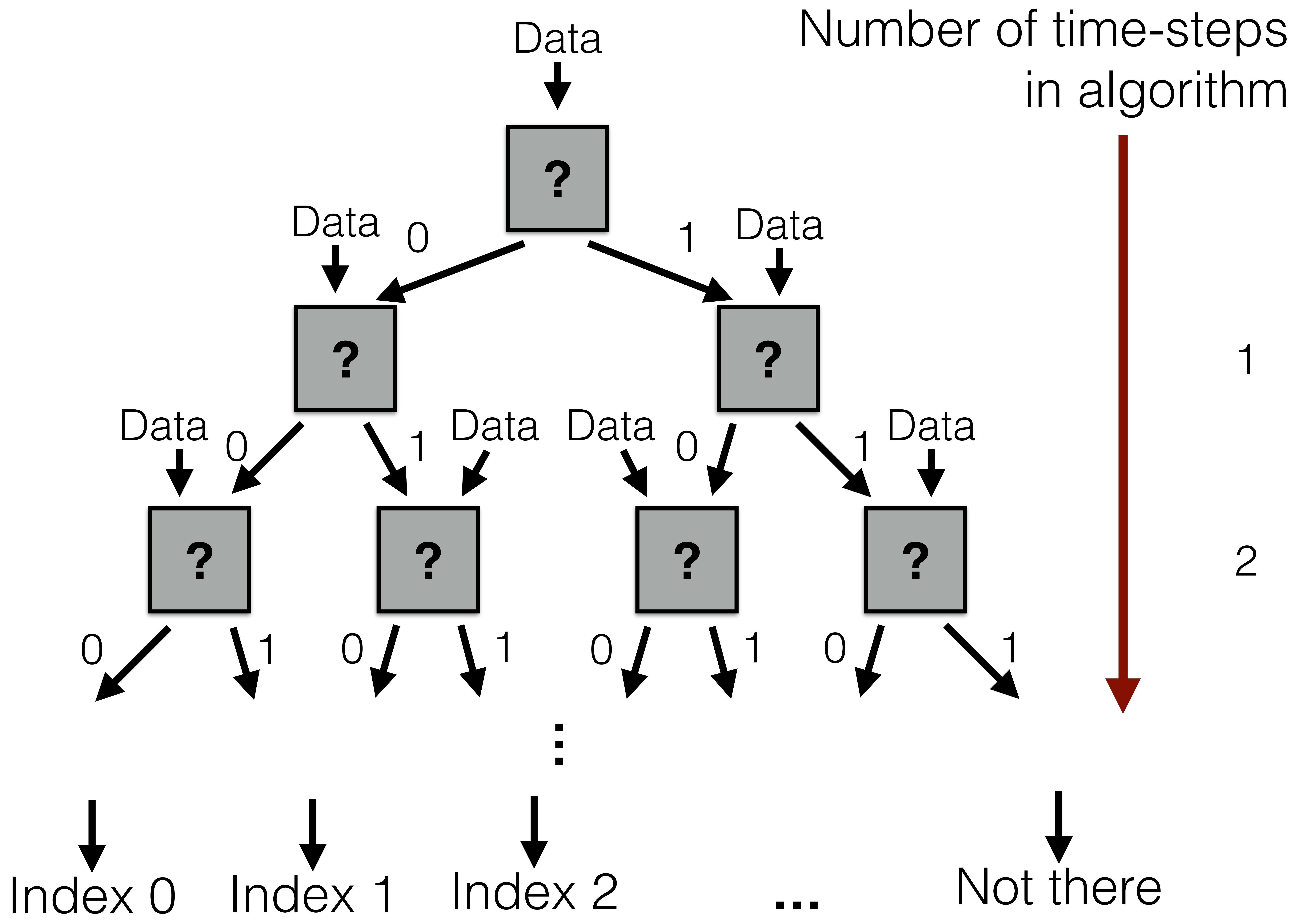
Implementation will look like this



Implementation will look like this

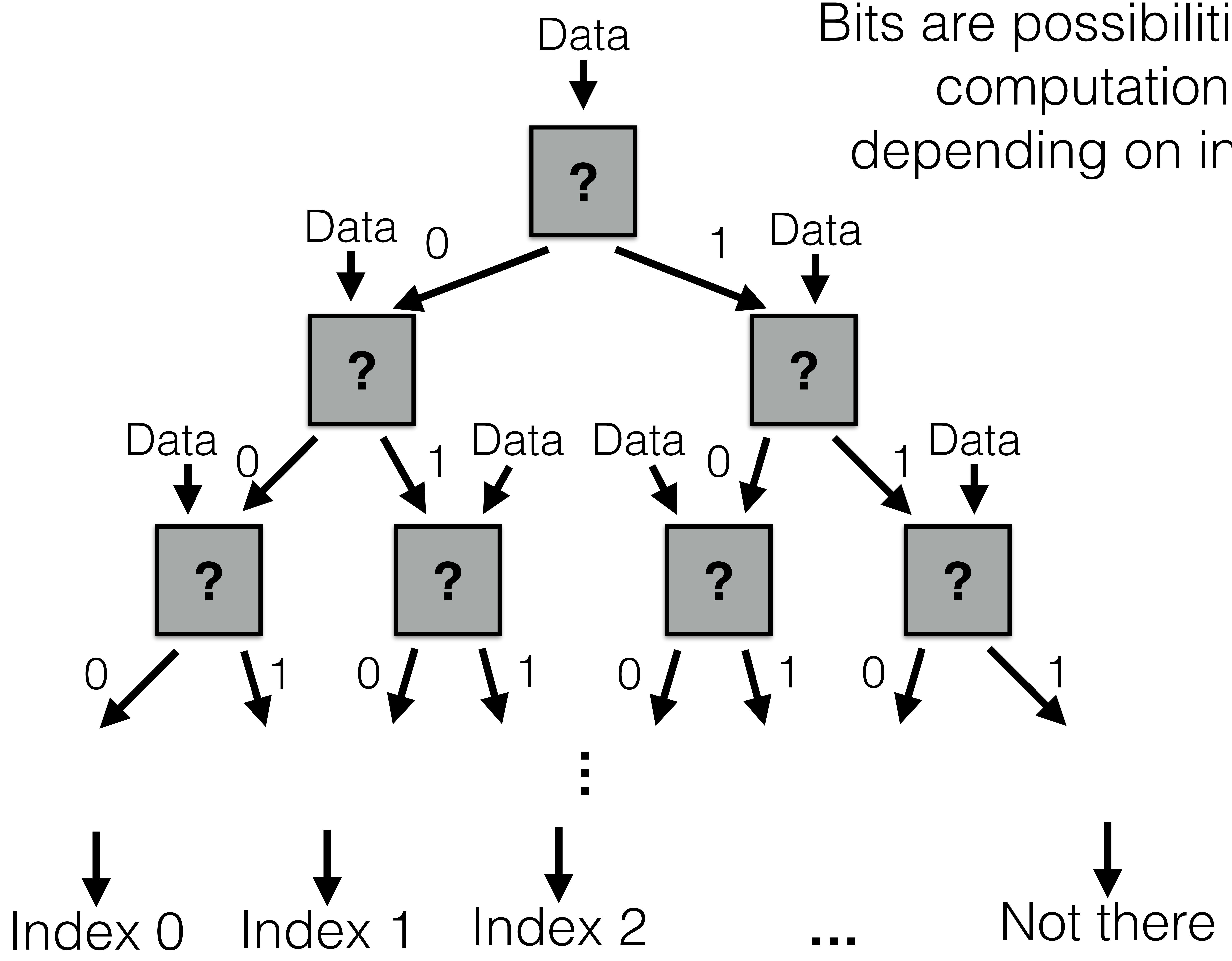


Implementation will look like this



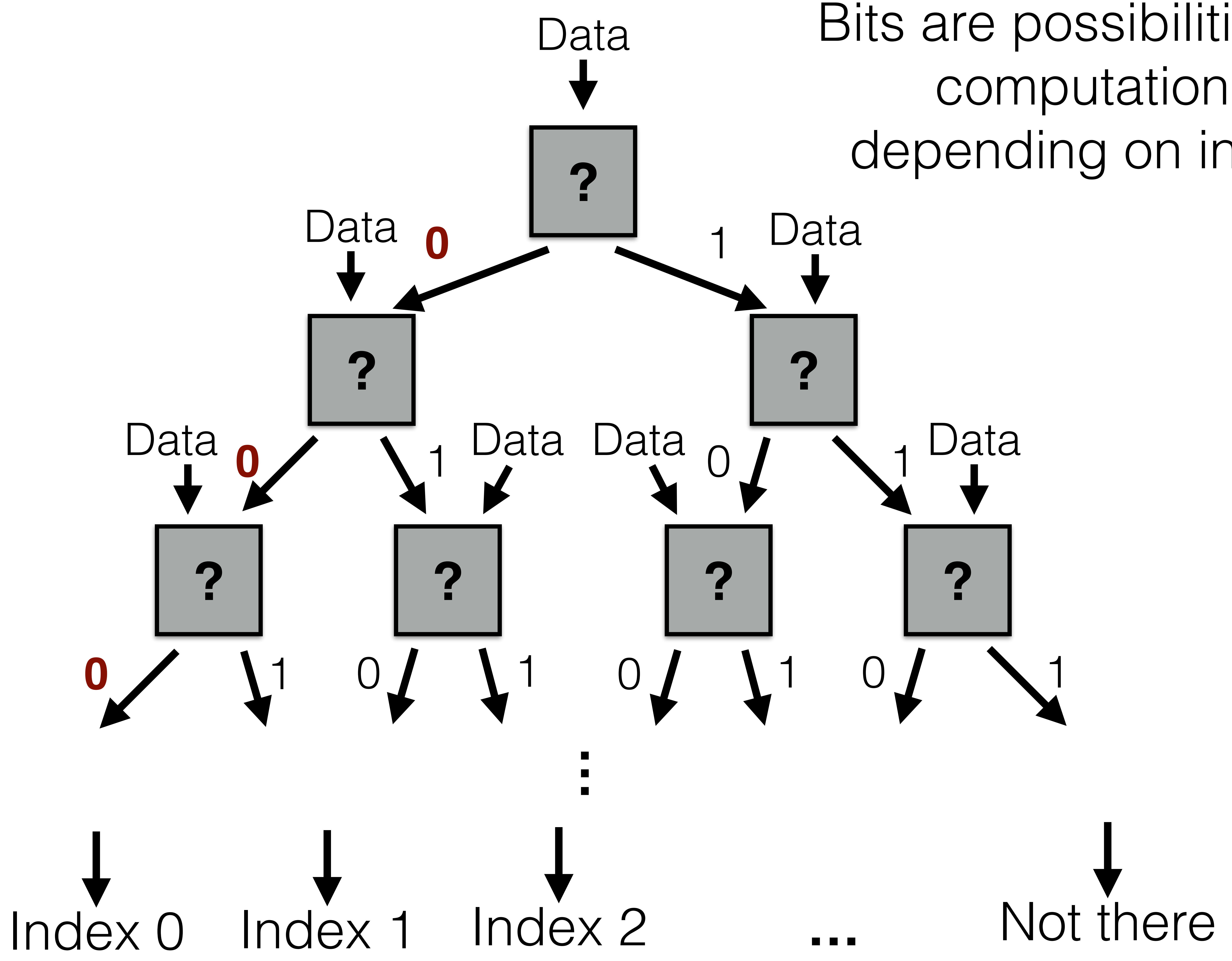
Implementation will look like this

Bits are possibilities of computation depending on input



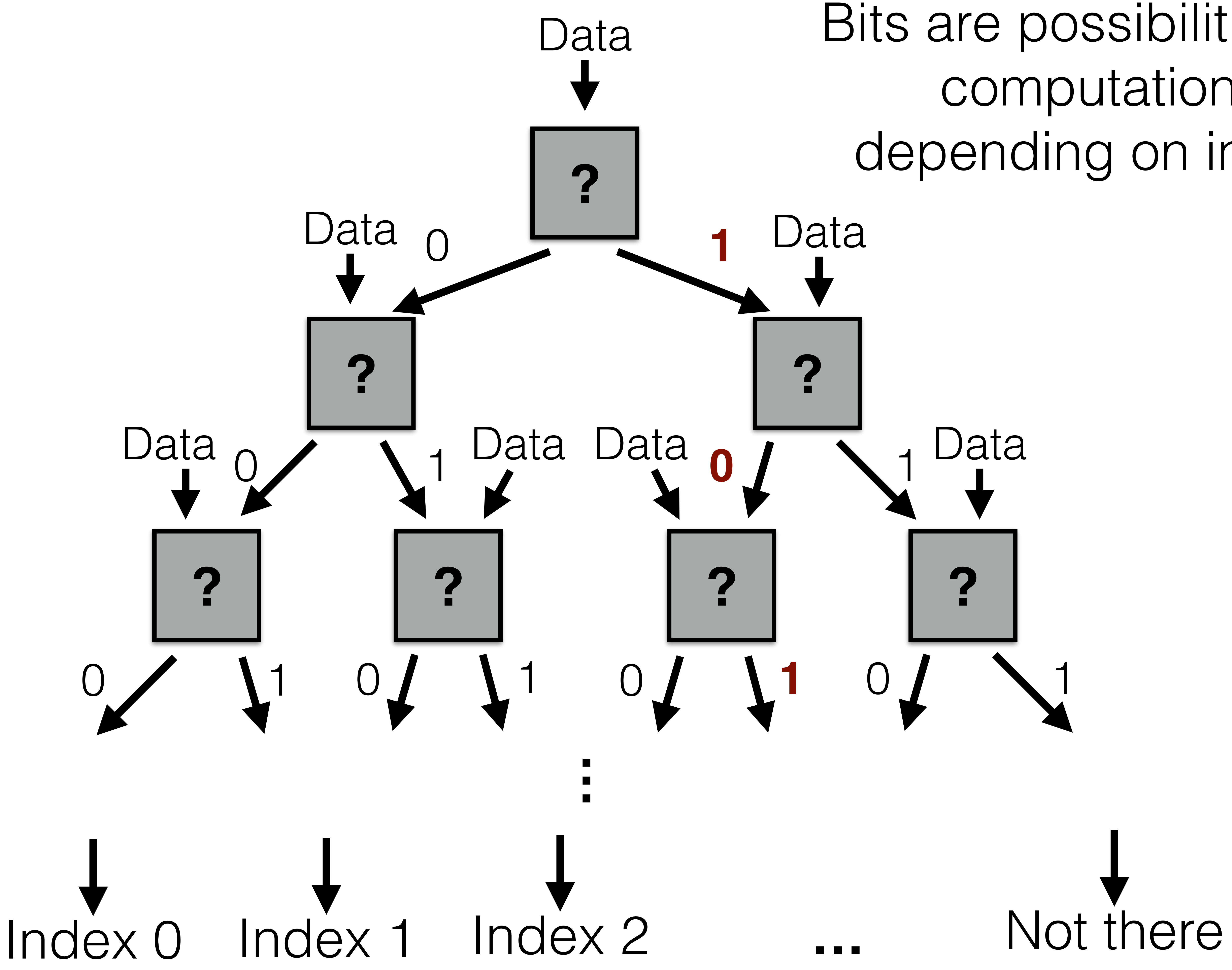
Implementation will look like this

Bits are possibilities of computation depending on input

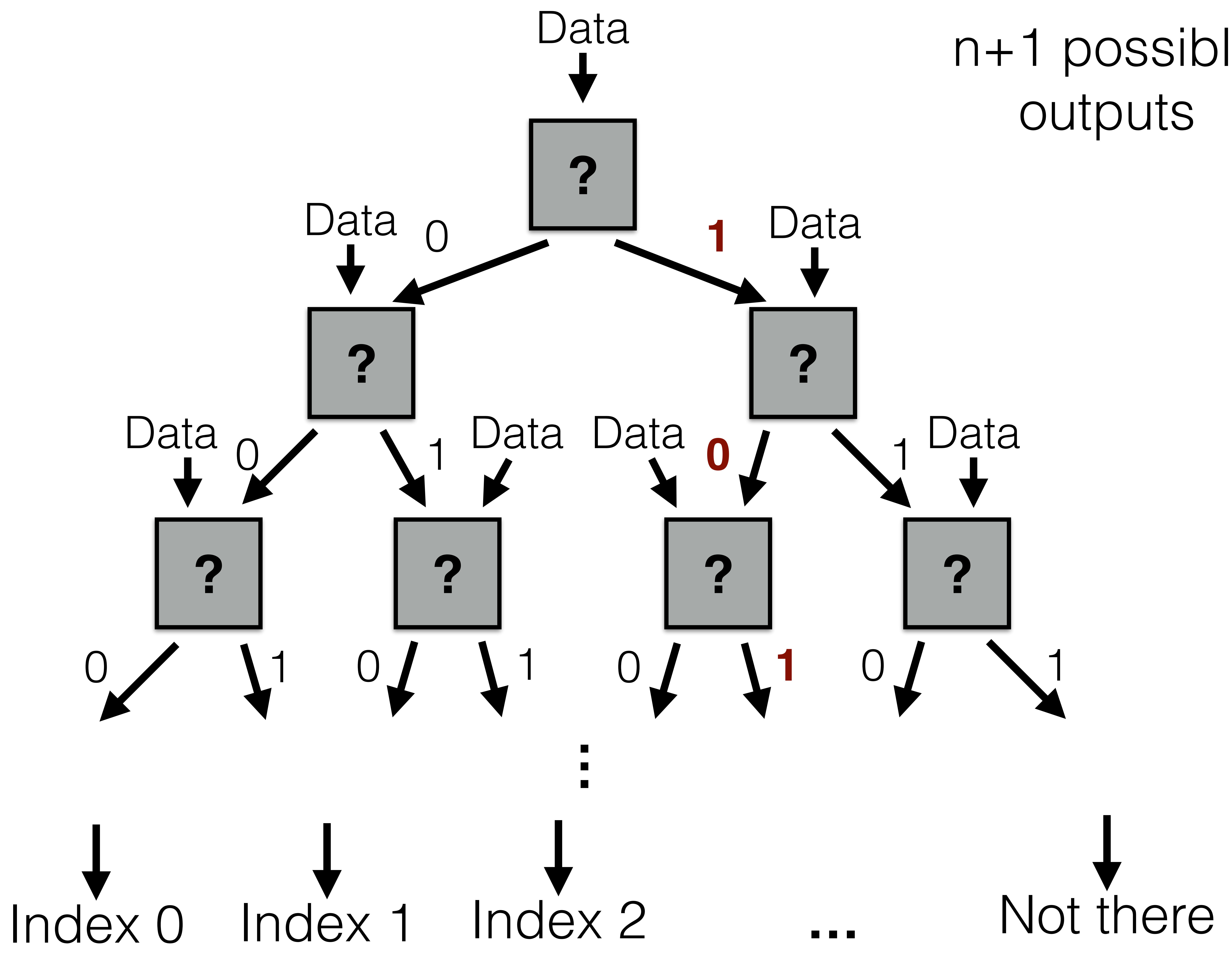


Implementation will look like this

Bits are possibilities of computation depending on input



Implementation will look like this



n+1 possible outputs

We have n+1 possible outputs

Number of time-steps is T

T is **at least** the length of bit-string describing steps in computation: 000… or 101…

We have *n+1* possible outputs

Number of time-steps is *T*

T is **at least** the length of bit-string describing steps in computation:
000… or 101…

2^T possible bit-strings	*n+1* outputs
00000…	index 1
10001…	index 2
00010…	index 3
01011…	index 4
11110…	not there

Two outputs cannot come from the same bit-string

Algorithm cannot tell the two outputs apart!

2^T possible bit-strings	$n+1$ outputs
00000…	index 1
10001…	index 2
00010…	index 3
01011…	index 4
11110…	not there

If *n+1* is greater than 2^T then two bit-strings must correspond to more than one output

Therefore 2^T is at least equal to $n+1$: $T = O(\log n)$

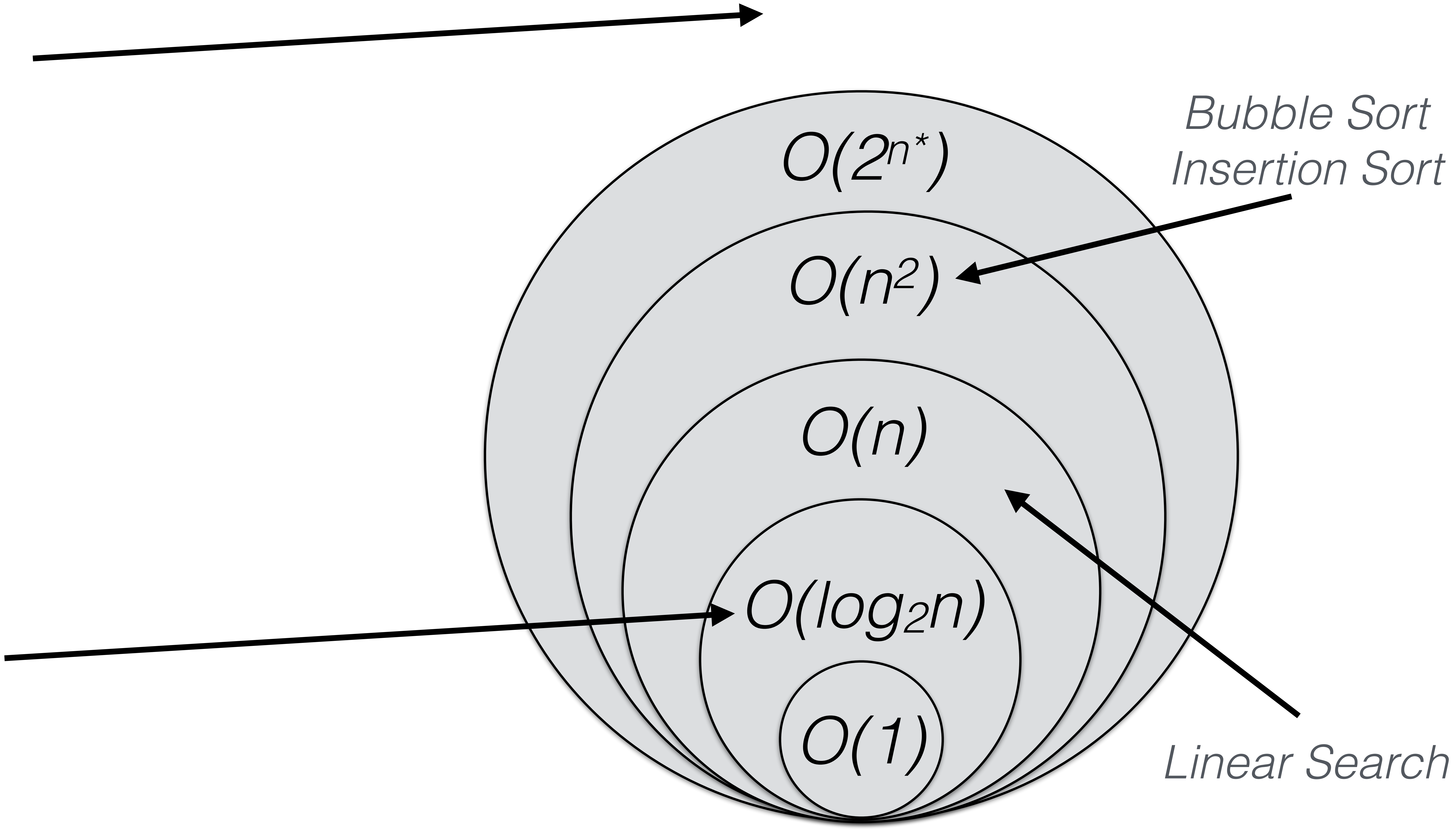
Achievable by Binary Search

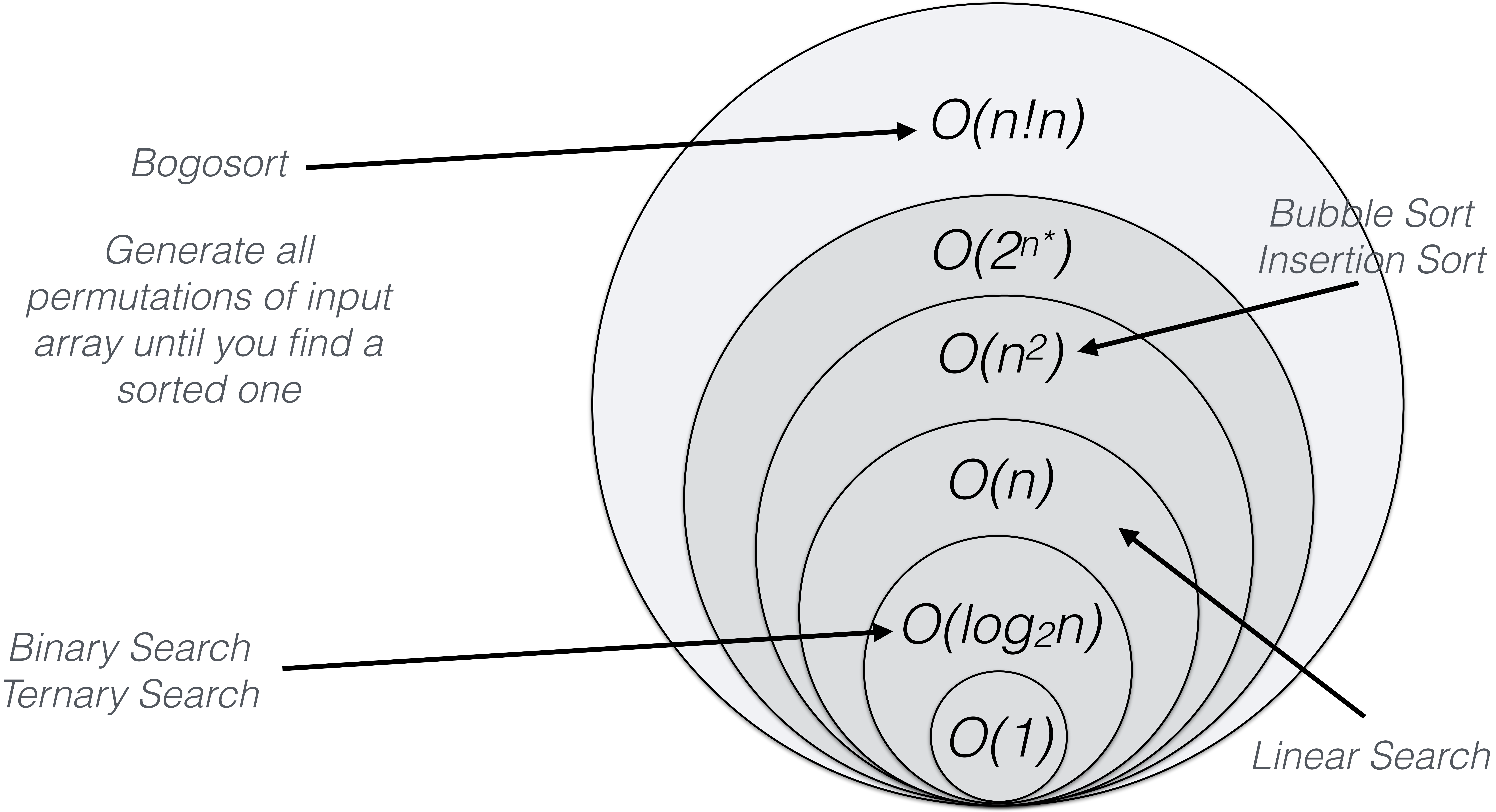
2^T possible bit-strings	$n+1$ outputs
00000…	index 1
10001…	index 2
00010…	index 3
01011…	index 4
11110…	not there

The argument more or less amounts to knowing what the structure of the **problem** is

We did not have to invent any new algorithms

Review Seminar





O(n!n)
Bogosort
Generate all permutations of input array until you find a sorted one
O(2^n*)
Bubble Sort
Insertion Sort
O(n^2)
O(n)
Linear Search
O(log2n)
Binary Search
Ternary Search
O(1)

Bogosort

```
function isSorted(array) {
    var n = array.length;
    for (var i = 0; i < n - 1; i++){
        if (array[i] > array[i + 1]) {
            return false;
        }
    }
    return true;
}

function swap(array,i,j) {
    var store = array[i];
    array[i] = array[j];
    array[j] = store;
    return array;
}
```

```
function permutationSort(array) {
    var n = array.length;
    if (isSorted(array)) {
        return array;
    }
    var p = [];
    for (var i = 0; i < n; i++) {
        p.push(0);
    }
    var i = 0;
    while (i < n){
        if (p[i] < i) {
            if (i % 2 == 0) {
                swap(array,0,i);
            } else {
                swap(array,p[i],i);
            }
            if (isSorted(array)) {
                return array;
            }
            p[i]++;
            i = 1;
        } else {
            p[i] = 0;
            i++;
        }
    }
}
```

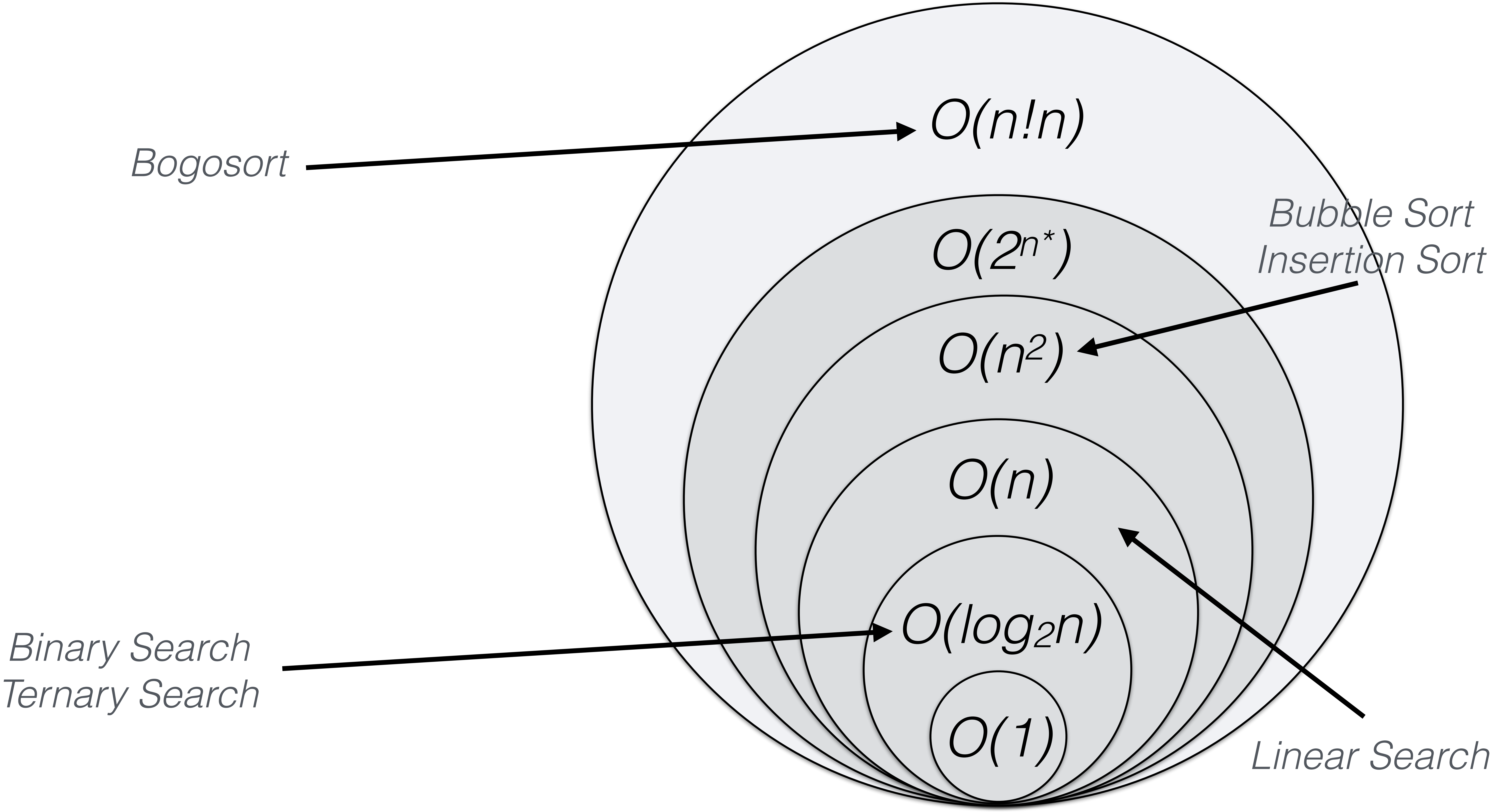
Generates all permutations of input array through swaps

If array is now sorted, return it

There are at most *O(n!)* iterations in loop for array of length *n*

In each iteration needs *O(n)* operations to check if sorted

O(n!n) worst-case time complexity



Bogosort
$O(n!n)$
Bubble Sort
Insertion Sort
$O(2^{n*})$
$O(n^2)$
$O(n)$
Binary Search
Ternary Search
$O(\log_2 n)$
$O(1)$
Linear Search