Notes - Algorithms & Data Structures - Binary Search

- Dr Nick Hayward

An initial consideration of binary search relative to algorithms and data structures.

Contents

Intro

Binary search algorithm is a common option for finding individual items in a larger dataset.

For example, we might use this algorithm to find a person in a directory or, perhaps, find a user in a broader network.

Instead of progressing from A to B to C &c. within a defined directory, we may start in the middle and then divide the data in half.

This type of division, however, is predicated on a sorted list of data for the binary search algorithm.

As binary search progresses through the dataset, it will return the index position for a matched result of `null` for no match.

This helps to eliminate possible results, and continually focus the dataset to find the search criteria.

Conceptual example

We may start with a simple example for guessing a given number from the ordered sequence 1 to 100.

search for a number

For example,

- first guess is `54`
  - this guess is too low
  - remove all numbers from `1 to 54`
  - number sequence is updated to `55 to 100`
- second guess is `75`
  - this guess is too high
  - remove all numbers from `100 to 75`
  - number sequence is updated to `55 to 74`

- third guess is `65`
    - this guess is too high
    - remove all numbers from `65 to 74`
    - number sequence is updated to `55 to 64`
- fourth guess is `60`
    - this guess is *correct*

By using binary search, we have shown a stark contrast with the algorithm for simple search.

For example, if we consider the above number search, we can see how the algorithm optimises performance.

- 100 -> 56 -> 20 -> 10 -> 0 - answer found...

Binary search has helped us find the correct number in four turns, instead of iterating through each number sequentially.

A key part of working with binary search is the need to start with an ordered list of data.

benefits of scale

A noted benefit of this type of algorithm is the potential to scale for larger datasets.

As the dataset grows exponentially, the search algorithm is able to keep pace for simple queries.

Working example

The conceptual design and use of a binary search algorithm may be implemented in many different programming languages.

For example, we might consider the following sample for a Python application.

Python binary search

The `binary_search` function takes a sorted array of items, and a single item. If the item is in the defined array, the search function will commonly return its position.

In effect, we can keep a record of where to find a given value.

So, we'll start by defining how to track the high and low values in a given data set. For example,

```
low = 0
high = len(list) -1
```

As the example searches for a value, we keep a record of where to search in the passed array for a given value.

We may also query the middle of the array. For example,

```
mid = (low + high) / 2
guess = list(mid)
```

We may then modify these values as we use binary search with the passed dataset. For example, if we guess a value for an item, it may be higher, lower, or a known value.

For a lower value, we simply check the current stored value of `low`. If the guess is too low, we nay update the current low value accordingly.

```
if (guess < item) {
    low = mid + 1
}
```

A practical example

An example of choosing between simple and binary search.

n.b. this may seem like an obvious choice, but there may be contexts where *linear time* may be acceptable.

In many examples, we need an algorithm that is both fast and correct.

e.g. Landing on Mars...

We need to quickly choose an algorithm, usually in 10 seconds or less, to allow a spaceshup to land on Mars.

For this test, binary search will be quicker for most tests. However, simple search is easier to write, and may reduce errors due to its inherent simplicity.

As we're performing mission critical tasks, we can't have any bugs.

So, we begin by running each algorithm 100 times. Each task may take 1 millisecond to execute. If we run initial tests, we get the following results

- simple search = 100 ms (100 x 1ms)
- binary search = 7 ms ($\log_2 100 = 7$)

100 ms vs 7 ms.

Whilst the real-world usage difference is minimal, the actual program will likely require a billion plus tasks and executions.

We may perform a quick initial scaling of timings, e.g.

- binary search = ~30 ms ($\log_2$ 1,000,000,000)

Binary search was initially ~15 times faster, so simple search will scale to `30 x 15`.

This seems reasonable, and is within tolerances for the program.

However, there's a major issue with this cursory calculation. It's based on an assumption that both search algorithms grow at the same rate.

Run times grow at different rates, thereby impacting performance relative to each dataset.

If we consider this specific example, Big O notation shows us that binary search is closer to 33 million times faster than simple search.

So, we cannot use simple search for our Mars lander.