search history

400,000 years ago

500 years ago

30 years ago

yesterday

© John Atkinson, Wrong Hands
Sequential search

- **sequential search**: Locates a target value in a list by examining each element from start to finish. Used in `index`.

- How many elements will it need to examine?

- Example: Searching the list below for the value **42**:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>
Sequential search

• How many elements will be checked?

```python
def index(value):
    for i in range(0, size):
        if my_list[i] == value:
            return i
    return -1  # not found
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

• On average how many elements will be checked?
**Binary search**

- **binary search**: Locates a target value in a *sorted* list by successively eliminating half of the list from consideration.

- How many elements will it need to examine?

- **Example: Searching the list below for the value 42:**

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

  - min
  - mid
  - max
Binary search runtime

• For an list of size $N$, it eliminates $\frac{1}{2}$ until 1 element remains.
  $N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \ldots, 4, 2, 1$
  • How many divisions does it take?

• Think of it from the other direction:
  • How many times do I have to multiply by 2 to reach $N$?
    $1, 2, 4, 8, \ldots, \frac{N}{4}, \frac{N}{2}, N$
  • Call this number of multiplications "$x$".
    $2^x = N$
    $x = \log_2 N$

• Binary search looks at a logarithmic number of elements
binary_search

Write the following two functions:

# searches an entire sorted list for a given value
# returns the index the value should be inserted at to maintain sorted order
# Precondition: list is sorted
binary_search(list, value)

# searches given portion of a sorted list for a given value
# examines min_index (inclusive) through max_index (exclusive)
# returns the index of the value or -(index it should be inserted at + 1)
# Precondition: list is sorted
binary_search(list, value, min_index, max_index)
Using `binary_search`

```
# index 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92}

index1 = binary_search(a, 42)
index2 = binary_search(a, 21)
index3 = binary_search(a, 17, 0, 16)
index2 = binary_search(a, 42, 0, 10)
```

• `binary_search` returns the index of the number
  or
- (index where the value should be inserted + 1)
Binary search code

# Returns the index of an occurrence of target in a,
# or a negative number if the target is not found.
# Precondition: elements of a are in sorted order

def binary_search(a, target, start, stop):
    min = start
    max = stop - 1

    while min <= max:
        mid = (min + max) // 2
        if a[mid] < target:
            min = mid + 1
        elif a[mid] > target:
            max = mid - 1
        else:
            return mid  # target found

    return -(min + 1)  # target not found