Search algorithms for vectors

Jordi Cortadella
Department of Computer Science
Search in a vector

• We want to design a function that searches for a value in a vector. The function must return the index of the location in which the value is found. It must return -1 if not found.

• If several locations contain the search value, it must return the index of one of them.

Specification:

```// Pre: A is vector.
// Returns i, such that A[i] == x, if x is in A.
// Returns -1 if x is not in A.
```
Invariant: x does not exist in A[0..i-1]
Search in a vector

// Returns i, such that A[i] == x, if x is in A.
// Returns -1 if x is not in A.

int search(int x, const vector<int>& A) {

    // Inv: x does not exist in A[0..i-1].
    for (int i = 0; i < A.size(); ++i) {
        if (A[i] == x) return i;
    }

    return -1;
}
Search with sentinel

• The previous code has a loop with two conditions:
  – $i < A\text{.size}()$: to detect the end of the vector
  – $A[i] == x$: to detect when the value is found

• The search is more efficient if the first condition is avoided (if we ensure that the value is always in the vector).

• To enforce this condition, a \textit{sentinel} may be added in the last (unused) location of the vector. When the sentinel is found, it indicates that the value was not anywhere else in the vector.
Search with sentinel

// Returns i, such that A[i] == x, if x is in A.  
// Returns -1 if x is not in A.  
// Post: the vector is temporarily modified, but the  
//       final contents remains unchanged.

int search(int x, vector<int>& A) {
    int n = A.size();
    A.push_back(x);    // Writes the sentinel

    int i = 0;
    // Inv: x does not exist in A[0..i-1]
    while (A[i] != x) ++i;
    A.pop_back();      // Removes the sentinel

    if (i == n) return -1;
    return i;
}
How would you search in a dictionary?

• Dictionaries contain a list of sorted words.

• To find a word in a dictionary of 50,000 words, you would never check the first word, then the second, then the third, etc.

• Instead, you would look somewhere in the middle and decide if you have to continue forwards or backwards, then you would look again around the middle of the selected part, go forwards/backwards, and so on and so forth ...
Is 4 in the vector?

-9 -7 -6 -6 -4 -1 0 1 3 4 5 5 7 8 8 9

4 is larger

Half of the elements have been discarded!

-9 -7 -6 -6 -4 -1 0 1 3 4 5 5 7 8 8 9

4 is smaller

-9 -7 -6 -6 -4 -1 0 1 3 4 5 5 7 8 8 9

Found!
Binary search

• How many iterations do we need in the worst case?

<table>
<thead>
<tr>
<th>iteration</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>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>elements</td>
<td>n</td>
<td>n/2</td>
<td>n/4</td>
<td>n/8</td>
<td>n/16</td>
<td>n/32</td>
<td>n/64</td>
<td>n/128</td>
<td>n/2^i</td>
</tr>
</tbody>
</table>

• The search will finish when only one element is left:

\[ 2^i = n \quad \Rightarrow \quad i = \log_2 n \]
## The power of $\log_2 n$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n/2$</th>
<th>$\log_2 n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>1,024</td>
<td>512</td>
<td>10</td>
</tr>
<tr>
<td>32,768</td>
<td>16,384</td>
<td>15</td>
</tr>
<tr>
<td>1,048,576</td>
<td>524,288</td>
<td>20</td>
</tr>
<tr>
<td>33,554,432</td>
<td>16,777,216</td>
<td>25</td>
</tr>
<tr>
<td>1,073,741,824</td>
<td>536,870,912</td>
<td>30</td>
</tr>
<tr>
<td>34,359,738,368</td>
<td>17,179,869,184</td>
<td>35</td>
</tr>
<tr>
<td>1,099,511,627,776</td>
<td>549,755,813,888</td>
<td>40</td>
</tr>
</tbody>
</table>

- **vector size**
- **linear search (avg.)**
- **binary search**

World population (2020) ≈ 7,800,000,000 $\approx 2^{32.86}$
**Invariant:**

If $x$ is in vector $A$, then it will be found in fragment $A[left...right]$

The search will be completed when the value has been found or the interval is empty ($left > right$)
Binary search

// Pre: A is sorted in ascending order,
// 0 <= left, right < A.size()
// Returns the position of x in A[left...right]
// (returns -1 if x is not in A[left...right])

int bin_search(int x, const vector<int>& A, int left, int right) {

    while (left <= right) {
        int i = (left + right)/2;
        if (x < A[i]) right = i - 1;
        else if (x > A[i]) left = i + 1;
        else return i; //Found
    }

    return -1;
}
// The initial call to bin_search should
// request a search in the whole array

... 

int i = bin_search(value, A, 0, A.size() - 1);

...
Binary search (recursive)

// Pre: A is sorted in ascending order,
// 0 <= left,right < A.size()
// Returns the position of x in A[left...right]
// (returns -1 if x is not in A[left...right])

int bin_search(int x, const vector<int>& A, int left, int right) {
    if (left > right) return -1;
    else {
        int i = (left + right)/2;
        if (x < A[i]) return bin_search(x,A,left,i-1);
        else if (x > A[i]) return bin_search(x,A,i+1,right);
        else return i; // found
    }
}
def bin_search(x, A, left, right):
    """A is a list sorted in ascending order.
    0 <= left <= right < len(A).
    Returns the position of x in the slice A[left:right+1].
    If x is not in A, it returns None.
    """

    if left > right:
        return None
    i = (left + right)//2  # integer division
    if x < A[i]:
        return bin_search(x, A, left, i-1)
    elif x > A[i]:
        return bin_search(x, A, i+1, right)
    else:
        return i

-9 -7 -6 -4 -1 0 1 3 4 5 5 7 8 8 9

left   i    right
Summary

• Searching is one of the most often executed operations in data structures.

• Efficient algorithms exist when data structures are sorted.

• The runtime difference between linear and logarithmic algorithms is enormous.