Search Algorithms

Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

Search algorithms on arrays

Search

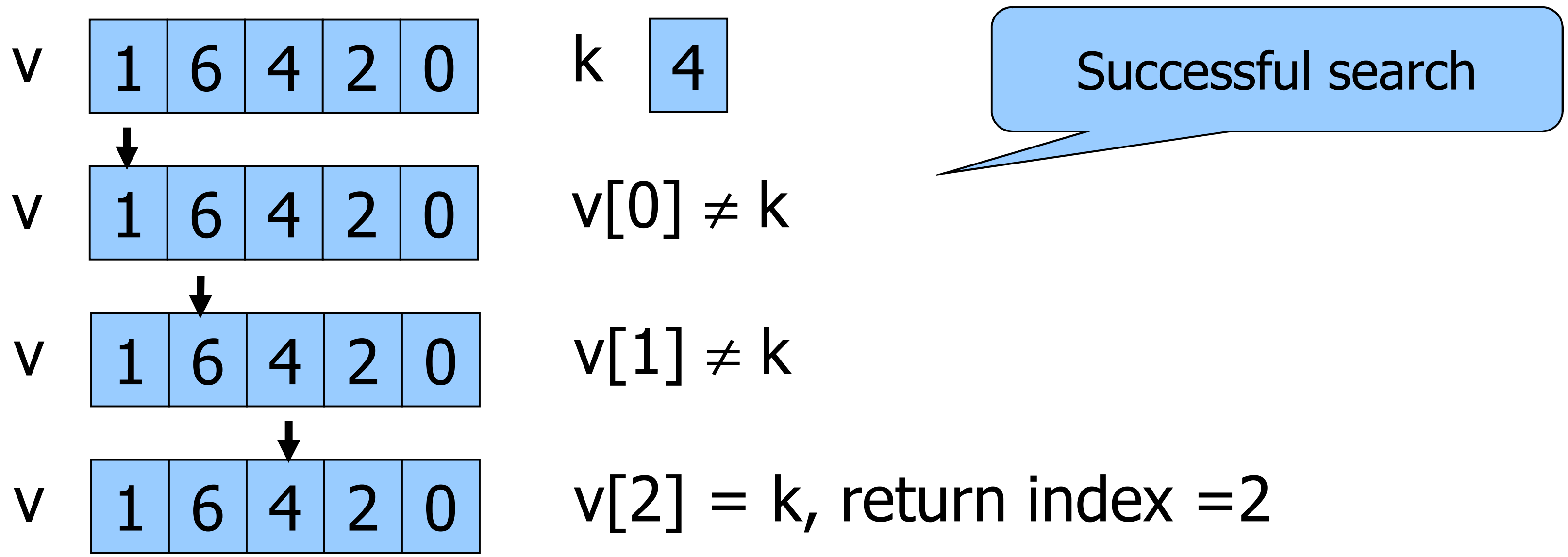
- Problem definizion
  - Is key k present in array v[N]?
  - Yes/No
- Input: v[N], k
- Output: Yes/No, if Yes, where in the array (index in the array)

Steps to developing a usable algorithm

- The scientific method
  - Model the problem
  - Find an algorithms to solve it
  - Fast enough? Fits in memory?
  - If not, figure out why
  - Find a way to address the problem
  - Iterate until satisfied

Algorithm 1: Sequential search

Sequential search: scan the array from the first element to potentially the last one, comparing key k and current value



Algorithm 1: Sequential search

v | 1 | 6 | 4 | 2 | 0 | k | 8 |

Unsuccessful search

v | 1 | 6 | 4 | 2 | 0 | $v[0] \neq k$

v | 1 | 6 | 4 | 2 | 0 | $v[1] \neq k$

v | 1 | 6 | 4 | 2 | 0 | $v[2] \neq k$

v | 1 | 6 | 4 | 2 | 0 | $v[3] \neq k$

v | 1 | 6 | 4 | 2 | 0 | $v[4] \neq k$, return index = -1

Algorithm 1: Sequential search

or size n

```
int LinearSearch (int v[], int l, int r, int k) {
  int i = l;
  int found = 0;

  while (i<=r && found==0) {
    if (k == v[i]) {
      found = 1;
    } else {
      i++;
    }
  }
  if (found==0)
    return -1;
  else
    return i;
}
```

Leftmost array index

Rightmost array index

Complexity Analysis

- Analytic analysis
  - Worst case = unsuccessful search
  - We assume unit cost for all operations

```
int i = l;                                1
int found = 0;                            1
while (i<=r && found==0) {
  if (k == v[i]) {                  r - l + 1 + 1
    found = 1;                        r - l + 1
  } else {                            r - l + 1
    i++;
  }}                                      1
if (found==0)
  return -1;                              1
else return i;
```

Complexity Analysis

1
1
r - l + 1 + 1
r - l + 1
r - l + 1
1
1

r – l +1 = n

T(n) =

= 1 + 1 + (r–l+1+1) + 2(r–l+1) + 1 + 1

= 1 + 1 + n + 1 + 2n + 1 + 1

= 3n + 5

= Θ (n)

T(n) grows linearly

Worst case
O(n) overall

Complexity Analysis

- Intuitive analysis
  - We consider n numbers for a search miss and in average n/2 for a search hit
  - T(n) grows linearly with n
    - $T(n) = \Theta(n)$

Algorithm 2: Binary search

Binary search in a **sorted** array

Problem definition

- Given a sorted array v[N]
- Is key k present in v[N]?
- Yes/No

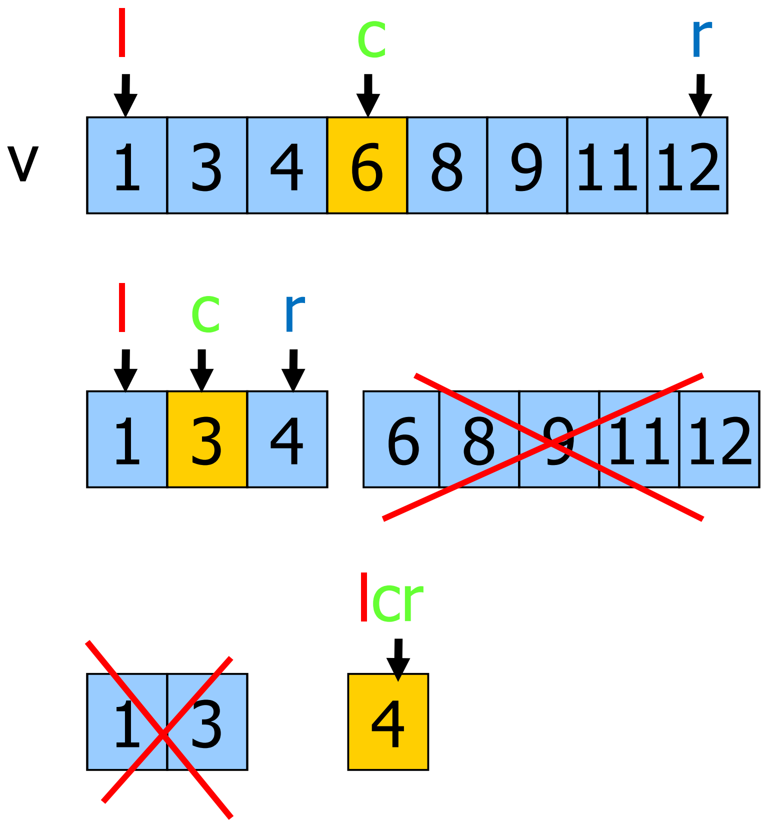
Approach

- At each step
  - compare k with middle element in the array
    - =: termination with success
    - <: search continues on left subarray
    - >: search continues on right subarray

First version: 1946
First bug-free version: 1962
Found bug in java
Arrays.binarySeaerch(): 2006

Algorithm 2: Binary search

k 4



y = middle element
l = leftmost array index
r = rightmost array index
c = index of middle element

Seach hit

v[2] = k, return index =2

Algorithm 2: Binary search

```
int BinSearch (int v[], int l, int r, int k) {
  int c;

  while (l<=r){
    c = (int) ((l+r) / 2);

    if(k == v[c]) {
      return(c);
    }
    if(k < v[c]) {
      r = c-1;
    } else {
      l = c+1;
    }
  }
  return(-1);
}
```

Binary Search: Complexity Analysis

- The array to be examined
  - At the beginning contains n numbers
  - At the 2nd iteration contains about n/2 numbers
  - …
  - At the i-th iteration contains about $n/2^i$ numbers
- Termination occurs when the array to be examined contains 1 number
  - thus $n/2^i = 1 \rightarrow n = 2^i \rightarrow i = \log_2(n)$
- T(n) grows logarithmically with n
  - $T(n) = \Theta(\log n)$