Algorithms and Complexity

Search Algorithms

Paolo Camurati and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Search on Unordered Arrays

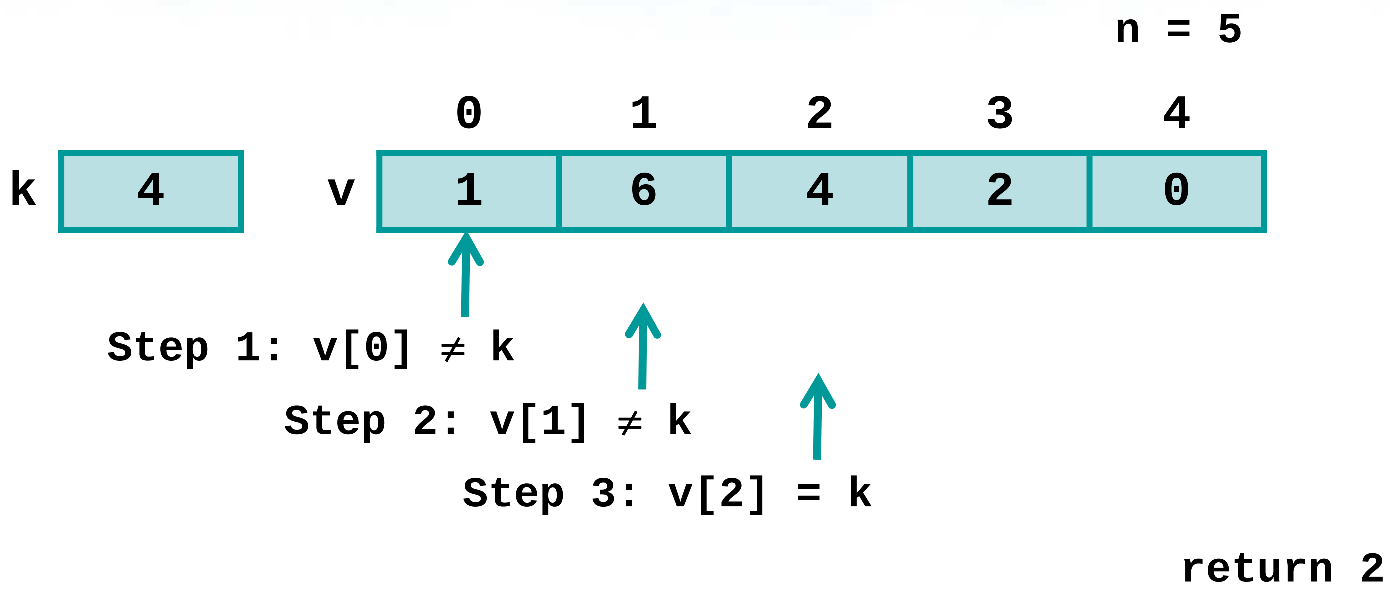
- Problem definition
  - Given an array storing **n** integer values
  - Given a specific value, i.e., a key **k**
  - Reply to the following question
    - Is key k present in the array v ?
- Input
  - v[n], n, k
- Output
  - Yes/No
  - If Yes in which position (index) k is in the array

Algorithm 1: Sequential Search

- Sequential search
  - Scan the array from the first element to potentially the last one
  - Compare key k and current value
    - Return index if the comparison is successful
  - When all comparisons have been unsuccessful, i.e., at the end of the scan, return -1

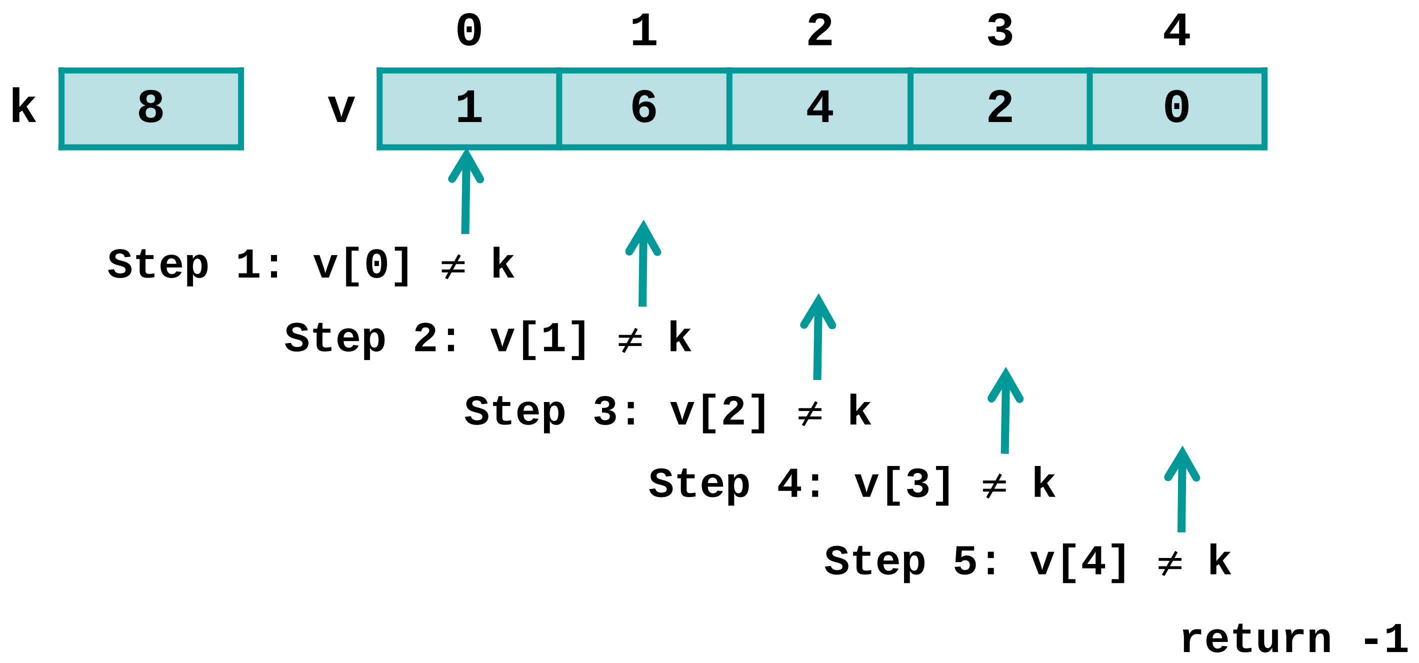
Algorithm 1: Sequential Search

- Successful search



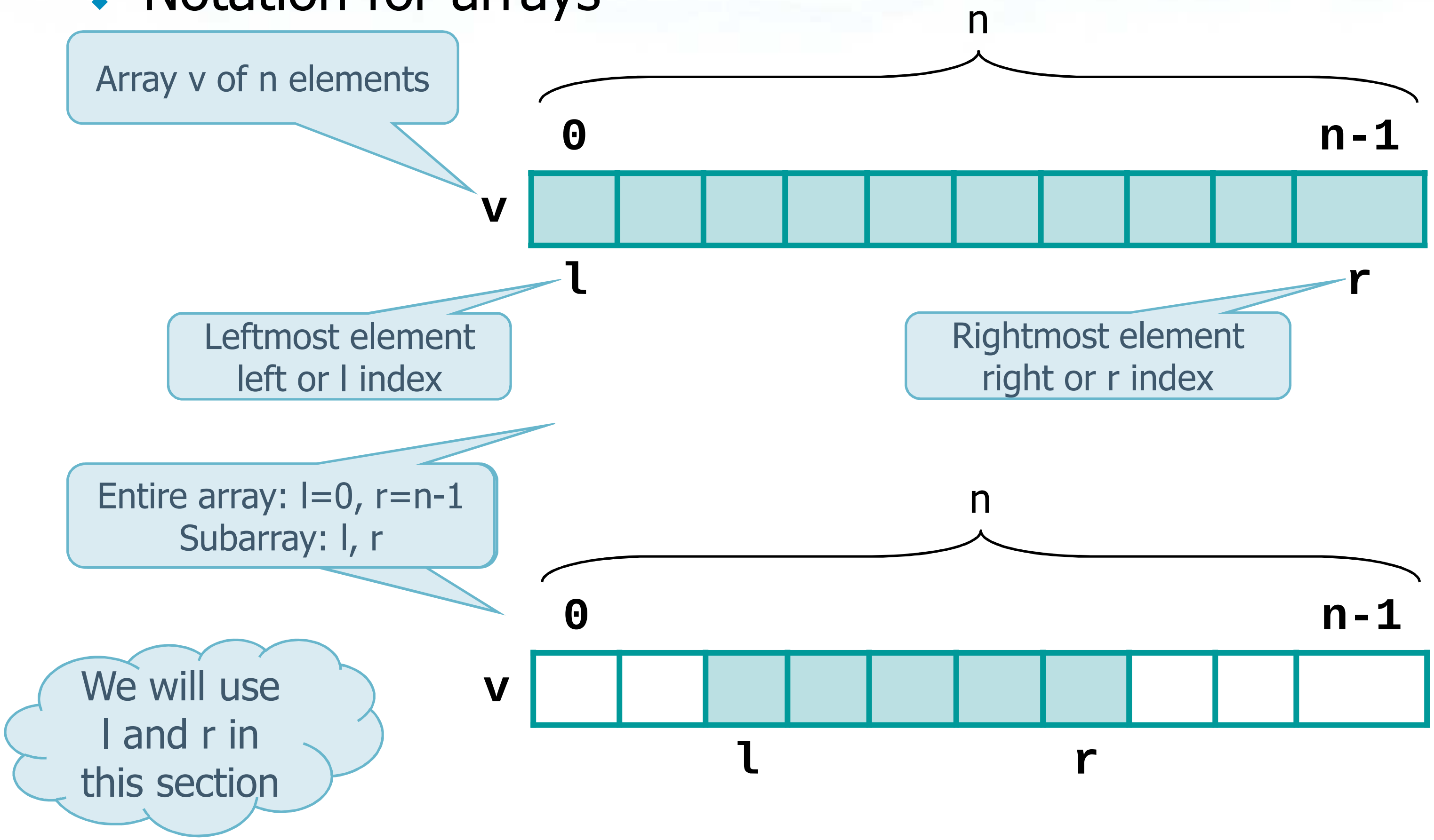
Algorithm 1: Sequential Search

- Unsuccessful search



Algorithm 1: Sequential Search

- Notation for arrays



Algorithm 1: Sequential Search

Leftmost array index

Rightmost array index

```
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;
}
```

Complexity Analysis

- Analytic analysis

Number of operations (with unit cost) for the worst case

```
int LinearSearch (...) {
  int i = l;                          1
  int found = 0;                      1

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

Complexity Analysis

$$T(n) = 1 + 1 + (r-l+1+1) + 2(r-l+1) + 1 + 1$$
$$= 1 + 1 + (n + 1) + 2n + 1 + 1$$
$$= 3n + 5$$
$$= \Theta(n)$$

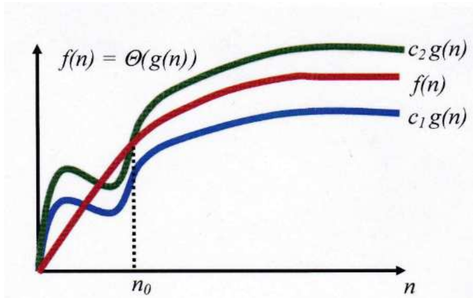
- T(n) grows linearly

Worst case
O(n) overall

```
1
1

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

r-l+1



1

1
```

Complexity Analysis

- Intuitive analysis
  - The worst case scenario is the one of an unsuccessful search
    - We have n steps for a search miss
    - In average n/2 steps for a search hit
  - T(n) grows linearly with n
    - $T(n) = \Theta(n)$



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

Search on Ordered Arrays

- Problem definition
  - Given an array storing **n** integer values **in ascending order**
  - Given a specific value, i.e., a key **k**
  - Reply to the following question
    - Is key k present in array v ?
- Input
  - v[n], n, k
- Output
  - Yes/No
  - If Yes in which position (index) k is in the array

Algorithm 2: Sequential Search

Leftmost array index

Rightmost array index

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

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

If k<v[i] it is not possibile to find k any more

Complexity Analysis

- Has complexity been improved?
- Intuitive analysis
  - The worst case scenario is the one of an unsuccessful search with an element larger than all array values
    - We have n steps
  - T(n) still grows linearly with n
    - $T(n) = \Theta(n)$
- Can we do better?

Algorithm 3: Binary Search

- Binary search in a sorted array
  - First version: 1946
  - First bug-free version: 1962

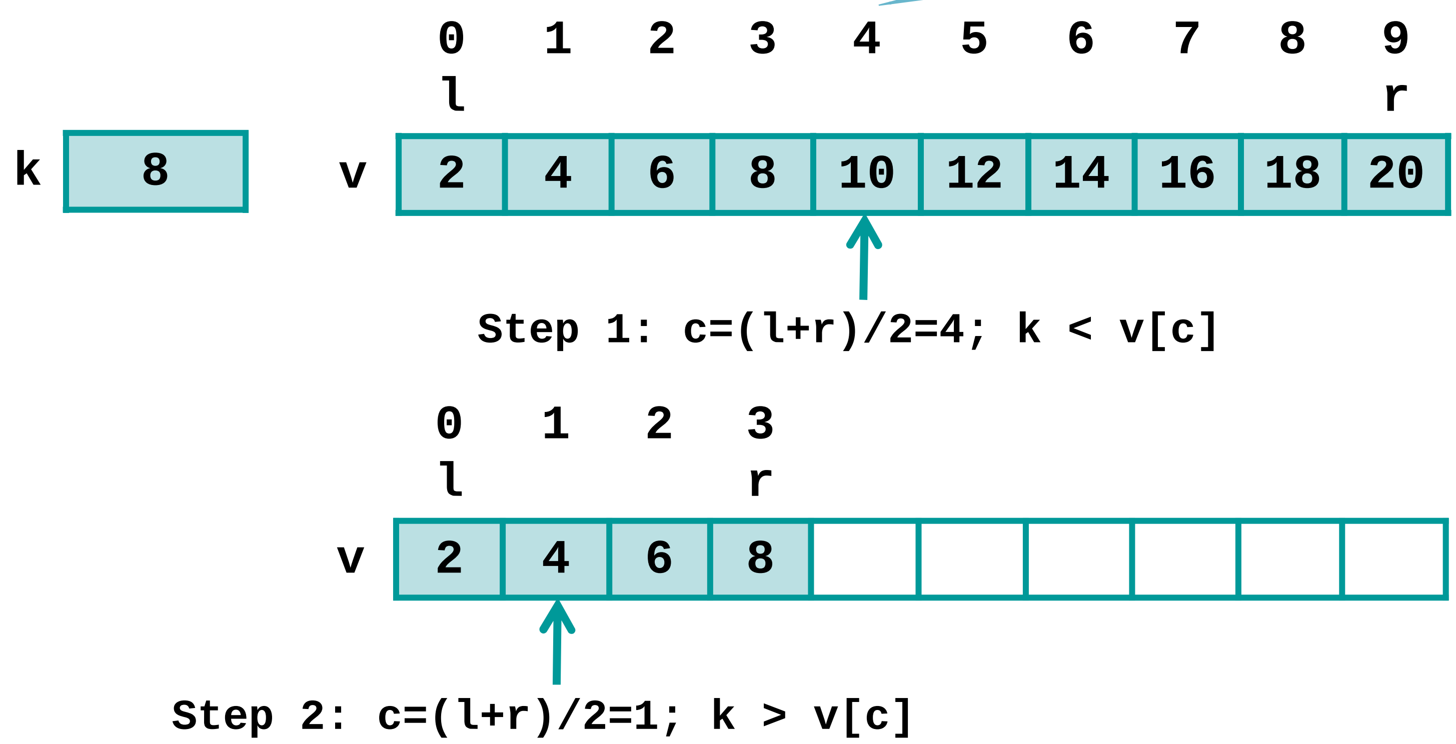
Found bug in java Arrays.binarySearch(): 2006

- Approach
  - Start with (sub-)array of extremes l and r
  - At each step
    - Find middle element c=(int)((l+r)/2)
    - Compare k with middle element in the array
      - =: termination with success
      - <: search continues on left subarray
      - >: search continues on right subarray

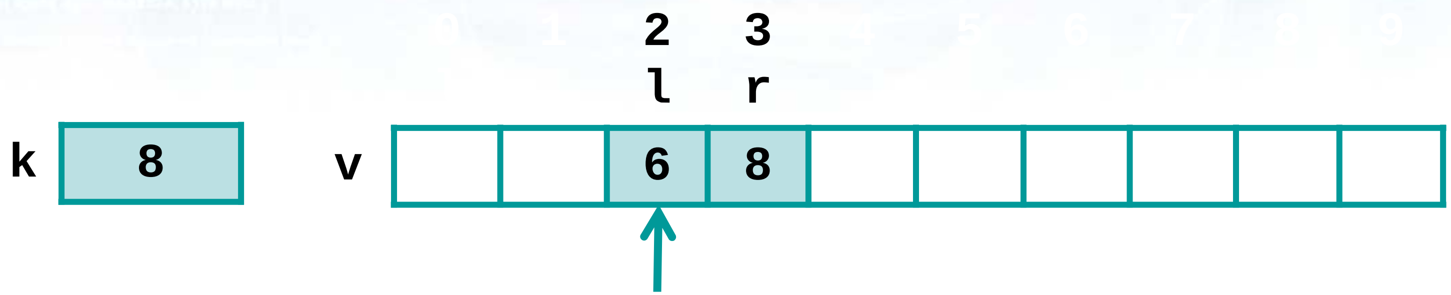
Algorithm 3: Binary Search

- Successful search

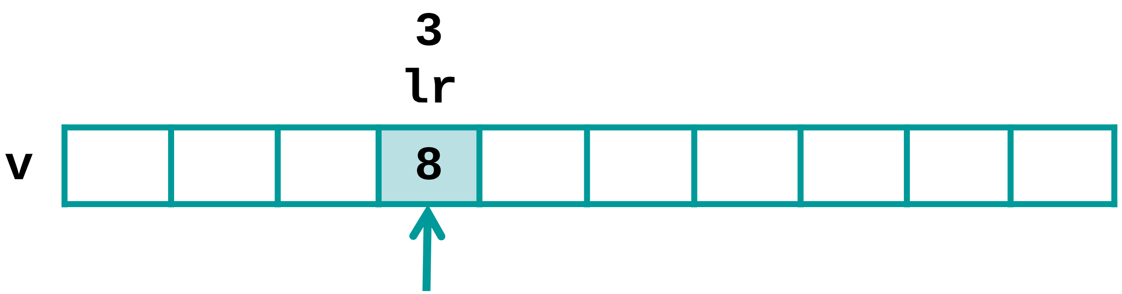
l = leftmost array index
r = rightmost array index
c = index of middle element



Algorithm 3: Binary Search



Step 3: c=(l+r)/2=2; k > v[c]



Step 4: c=(l+r)/2=3; k = v[c]

return 3

Algorithm 3: Binary Search

- Unsuccessful search

k = 8

	0	1	2	3	4	5	6	7	8	9
	l									r
v	2	4	6	7	10	12	14	16	18	20

```
Step 1: c=(l+r)/2=4; k < v[c]
```

… like previous case but last step …

				3						
				lr						
v				7						

```
Step 4: c=(l+r)/2=3; k ≠ v[c]    return -1
```

Algorithm 3: Binary Search

Leftmost array index

Rightmost array index

```
int BinarySearch (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);
}
```

Or just
c = (l+r)/2;
as c is an integer variable

Complexity Analysis

- Analytic analysis
  - The worst case scenario is the one of an unsuccessful search or a successful search with a hit at the final step
    - We have n elements
    - At each step, we divide n by 2
    - In i steps, we divide n by 2^i, i.e., $n/2^i$
    - We stop after i steps, when
      - $n/2^i = 1$
      - $n = 2^i$
      - $i = \log_2 n$
  - T(n) grows logaritmically with n
    - $T(n) = O(\log_2 n)$