Week 03 - Lecture 3 Slides

Dr. Yeganeh Bahoo

Created: 2023-09-25 Mon 10:39

Table of Contents

Lecture 3: Searching & Sorting (cont.)

Learning objectives: By the end of this lecture you should be able to

1. Explain and implement sequential search and binary search.
2. Explain and implement selection sort

Searching and sorting arrays

Now that we have learned about the array data structure, lets learn the two most prominent algorithms used with arrays:

- Searching

- Sorting

Sequential search

Searches over all elements of the vector starting from the first.

- Suppose you are doing a sequential search in the vector

  ```
  #(15 18 2 19 18 0 8 14 19 14)
  ```

  How many comparisons would you need to do in order to find the key 18? And to find the key 3?

- Suppose you are doing a sequential search on the following ordered vector

  ```
  #(3 5 6 8 11 12 14 15 17 18)
  ```

  How many comparisons would you need to do in order to find the key 13?

Sequential Search: complexity

List type	Case	Best Case	Worst Case
unordered	item is present	1	n
unordered	**item is not present**	**n**	**n**

List type	Case	Best Case	Worst Case
ordered	item is present	1	n
ordered	**item is not present**	**1**	**n**

Since the wors-case scenario provides an upper bound on the algorithm's performance, the time complexity of sequential search is $O(n)$

Binary Search

- If we know that our sequence is sorted, then we can do much better than $O(n)$
- We can start from the middle item, then use a **divide-and-conquer strategy** to search the list

- divide-and-conquer binary search:

```
if the number is smaller than the middle number
  search in the first half
if the number is greater that the middle number
  search in the other half
if the number is equal to the middle number
  found
else
  not found
```

For example: searching for number 6 in `#(1 2 3 4 5 6)`

Array slice	Position of first element in array slice wrt original array
`#(1 2 3 4 5 6)`	0
`#(4 5 6)`	3
`#(5 6)`	4
`#(6)`	5

Binary search: implementation

Classical divide-and-conquer binary search

```
(defun bin-search (val vec &optional (pos 0))
  "If val is in vec, return pos, i.e, the position of val in the original vector;
   otherwise, return NIL"
  (if (> (length vec) 1)
      (let* ((midpt (floor (length vec) 2))
             (midel (aref vec midpt)))
        (cond ((< midel val) (bin-search val (rtl:slice vec midpt) (+ pos midpt)))
              ((> midel val) (bin-search val (rtl:slice vec 0 midpt) pos))
              (t (+ pos midpt))))
      (when (= (aref vec 0) val)
        pos)))
```

Searching for 6 in `#(1 2 3 4 5 6)`

Array slice	Position of first element in array slice wrt original array
`#(1 2 3 4 5 6)`	0
`#(4 5 6)`	3
`#(5 6)`	4
`#(6)`	5

Binary search: complexity analysis

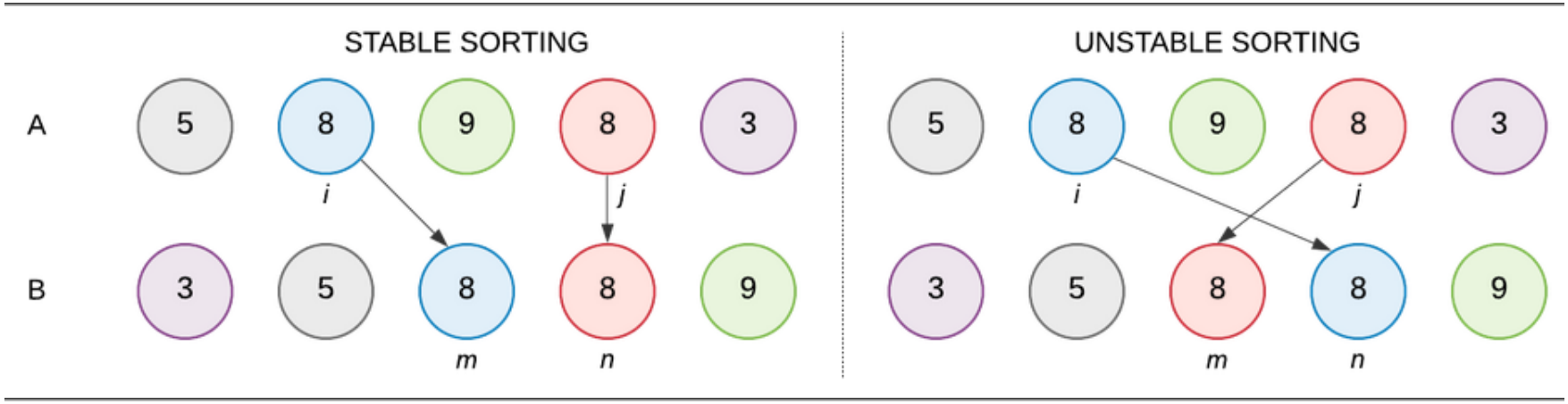
Comparisons	Approximate Number of Items Left
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
...	...
i	$\frac{n}{2^i}$

- when we split the array enough times we'll be left with one item, the one we are looking for or not.
- It will take i comparisons to reach that point
- and since there will be only one element left, $\frac{n}{2^i} = 1$, i.e., $i = \log_2 n$
- Therefore, binary search is $O(\log n)$

Binary search's caveat: you must sort the sequence before you search it.

Sorting

- **Objective**: place all elements of a sequence in a certain order determined by a *comparison predicate*
- Aspects that differentiate sorting algorithms:
  - **in-place**: done in the original data structure itself. The alternative is **copying-sort**
  - **stable**: whether two elements considered equal maintain their original relative positions



  - **online**: whether the algorithm has to see the whole sequence, or can work on a sorted prefix.

Selection sort

- It is an **in-place** sorting algorithm
- It moves left-to-right as it builds the sorted prefix to the left of the "*current element*"
- In a pass, it finds the "*best*" (smallest or largest) element to the right of the *current element*, then swaps these two elements

						Pass
$\#($	$\overset{cur}{3}$	1	$\overset{0}{best}$	7	8	$2)$	
$\#(0$	1	3	7	8	$2)$	1
$\#(0$	$\overset{cur}{\underset{best}{1}}$	3	7	8	$2)$	
$\#(0$	1	3	7	8	$2)$	2
$\#(0$	1	$\overset{cur}{3}$	8	7	$\overset{2}{best})$	
$\#(0$	1	2	7	8	$3)$	3
$\#(0$	1	2	$\overset{cur}{7}$	8	$\overset{3}{best})$	
$\#(0$	1	2	3	8	$7)$	4
$\#(0$	1	2	3	$\overset{cur}{8}$	$\overset{7}{best})$	
$\#(0$	1	2	3	7	$8)$	5
$\#(0$	1	2	3	$\overset{cur}{7}$	$8)$	

Selection sort algorithm

```
(defun selection-sort (vec comp)
  (dotimes (cur (1- (length vec)))
    (let ((best (aref vec cur))
          (idx cur))
      (do ((j (1+ cur) (1+ j))) ; j is initially cur+1
          ((> j (1- (length vec))))
        (when (funcall comp (aref vec j) best)
          (:= best (aref vec j)
              idx j)))
      (rotatef (aref vec cur) (aref vec idx)))) ; this is the Lisp swap operator
  vec)
```

Usage:

```
CL-USER> (defvar a (make-array 6 :initial-contents (list 3 1 0 7 8 2)))
A
CL-USER> a
#(3 1 0 7 8 2)
CL-USER> (selection-sort a #'<)
#(0 1 2 3 7 8)
```

```
CL-USER> (selection-sort a #'>)
#(8 7 3 2 1 0)
```

Analysis of selection sort

- Comparisons for each pass

Pass	Comparisons
1	*n*-1
2	*n*-2
3	*n*-3
...	...
n-1	1

- # of comparisons
  - Total number of comparisons $\sum_{i=1}^{n-1} i = \frac{1}{2}n^2 - \frac{1}{2}n$
  - Therefore $O(n^2)$ comparisons
- # of exchanges
  - best case: no exchanges if list is sorted
  - worst case: one exchange per pass, hence $O(n)$

Quiz

https://bit.ly/48vAmL9	bit.ly_48vAmL9.png