

**CS1010**

<http://www.comp.nus.edu.sg/~cs1010/>

*Programming Methodology*

## Unit 20

---

# Searching and Sorting



**NUS**  
National University  
of Singapore

School of  
Computing

# Unit 20: Searching and Sorting

## Objectives:

- Understand the basic searching algorithms and sorting algorithms
- Introduce the concept of complexity analysis (informally)
- Implement the searching and sorting algorithms using arrays

## Reference:

- Chapter 7 Array Pointers
  - Section 7.6 Searching and Sorting an Array

# Unit 20: Searching and Sorting (1/2)

1. Overall Introduction
2. Introduction to Searching
3. Linear Search
  - Demo #1
  - Performance
4. Binary Search



SEARCHING

# Unit 20: Searching and Sorting (2/2)

## 5. Introduction to Sorting

## 6. Selection Sort

- Demo #2
- Performance

## 7. Bubble Sort

- Demo #3
- Performance

## 8. More Sorting Algorithms

## 9. Animated Sorting Algorithms



**SORTING**

# 1. Overall Introduction

- You have accumulated quite a bit of basic programming experience by now.
- Today, we will study some simple yet useful classical algorithms which find their place in many CS applications
  - **Searching** for some data amid very large collection of data
  - **Sorting** very large collection of data according to some order
- We will begin with an algorithm (idea), and show how the algorithm is transformed into a C program (implementation).
- This brings back (reminds you) our very first lecture: the importance of beginning with an algorithm.

## 2. Introduction to Searching (1/2)



- Searching is a common task that we carry out without much thought everyday.
  - Searching for a location in a map
  - Searching for the contact of a particular person
  - Searching for a nice picture for your project report
  - Searching for a research paper required in your course
- In this lecture, you will learn how to search for an item (sometimes called a search key) in an array.



## 2. Introduction to Searching (2/2)

- Problem statement:

Given a list (collection of data) and a search key  $X$ , return the position of  $X$  in the list if it exists.

For simplicity, we shall assume there are no duplicate values in the list.

- We will count the number of comparisons the algorithms make to analyze their performance.
  - The ideal searching algorithm will make the least possible number of comparisons to locate the desired data.
  - We will introduce worst-case scenario.
  - (This topic is called **analysis of algorithms**, which will be formally introduced in CS2040. Here, we will give an informal introduction just for an appreciation.)

### 3. Linear Search (1/3)



- Also known as **Sequential Search**
- Idea: Search the list from one end to the other end in linear progression.
- Algorithm:

Example: Search for 24 in this list

```
// Search for key in list A with n items
linear_search(A, n, key)
{
    for i = 0 to n-1
        if Ai is key then report i
}
```

87	12	51	9	24	63
↑	↑	↑	↑	↑	
no	no	no	no	yes!	

Return 4

- Question: What to report if key is not found?
  - Aim for a clean design



### 3. Linear Search: Demo #1 (2/3)



- If the list is an array, how would you implement Linear Search algorithm?

```
// To search for key in arr using linear search
// Return index if found; otherwise return -1
int linearSearch(int arr[], int size, int key) {
    int i;

    for (i=0; i<size; i++)
        if (key == arr[i])
            return i;
    return -1;
}
```

See [linear\\_search.c](#) for full program

Useful and common technique

- Question: What if array contains duplicate values of the key?

Index of the first element found will be returned.

### 3. Linear Search: Performance (3/3)

- We use the **number of comparisons** here as a rough basis for measurement.
- Analysis is done for best case, average case, and worst case. We will focus on the **worst case**.
- For **an array with  $n$  elements**, in the worst case,
  - What is the number of comparisons in linear search algorithm?

$n$  comparisons  
This is called **linear time** algorithm:  $O(n)$
  - Under what circumstances does the worst case happen?

(a) Not found  
(b) Found at the last element

## 4. Binary Search (1/6)

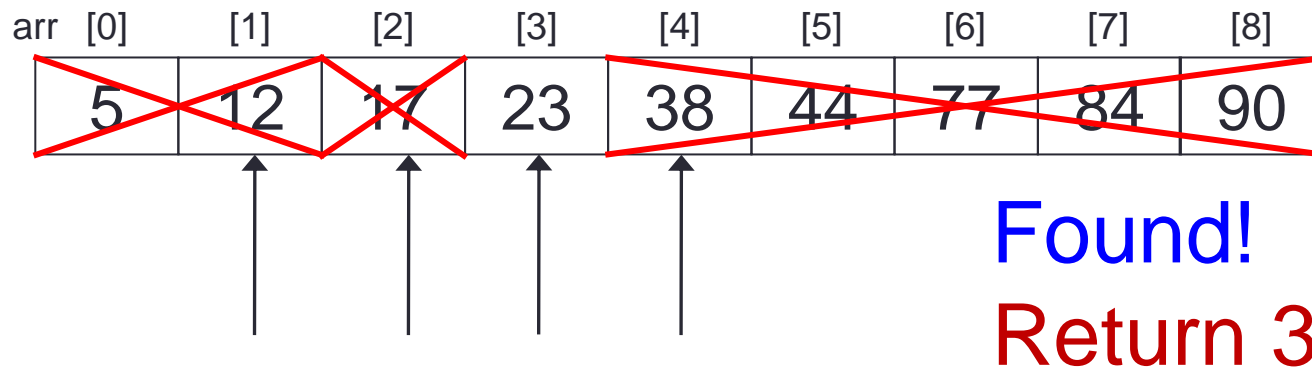
- You are going to witness a **radically different approach**, one that has become the basis of many well-known algorithms in Computer Science!
- The idea is simple and fantastic, but when applied on the searching problem, it has this pre-condition that the list must be **sorted before-hand**.
- How the data is organized (in this case, sorted) usually affects how we choose/design an algorithm to access them.
- In other words, sometimes (actually, very often) we **seek out new way to organize the data** so that we can process them more efficiency. → More of this in CS2040 Data Structures and Algorithms.

## 4. Binary Search (2/6)

- **The Binary Search algorithm**
  - Look for the key in the middle position of the list. Either of the following 2 cases happens:
    - **If the key is smaller than the middle element, “discard” the right half of the list and repeat the process.**
    - **If the key is greater than the middle element, “discard” the left half of the list and repeat the process.**
  - Terminating condition: when the key is found, or when all elements have been “discarded”.

## 4. Binary Search (3/6)

- Example: Search key = 23



1. low = 0, high = 8, mid =  $(0+8)/2 = 4$     `arr[4] = 38 > 23`
2. low = 0, high = 3, mid =  $(0+3)/2 = 1$     `arr[1] = 12 < 23`
3. low = 2, high = 3, mid =  $(2+3)/2 = 2$     `arr[2] = 17 < 23`
4. low = 3, high = 3, mid =  $(3+3)/2 = 3$     `arr[3] = 23 == 23`

## 4. Binary Search (4/6)

- In binary search, **each step eliminates the problem size (array size) by half!**
  - The problem size gets reduced to 1 very quickly! (see slide after next.)
- This is a simple yet powerful strategy, of **halving the solution space in each step**
  - This is a BIG DEAL in problem solving (remember the Santa Claus' dirty socks problem in your first discussion session?)
- Such strategy, a special case of divide-and-conquer paradigm, can be naturally implemented using **recursion**.
- But for now, we will stick to an iterative solution (loop). (We will see its implementation using recursion when we cover recursion.)

## 4. Binary Search (5/6)

`binary_search.c`

```
// To search for key in sorted arr using binary search
// Return index if found; otherwise return -1
int binarySearch(int arr[], int size, int key) {

    int low = 0, high = size - 1, mid = (low + high)/2;

    while ((low <= high) && (arr[mid] != key)) {
        if (key < arr[mid])
            high = mid - 1;
        else
            low = mid + 1;
        mid = (low + high)/2;
    }
    if (low > high) return -1;
    else return mid;
}
```

## 4. Binary Search (6/6)

- Worst-case analysis

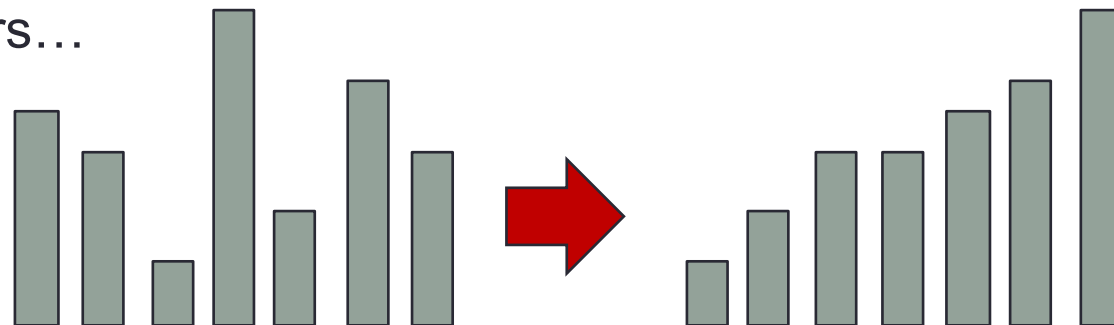
Array size $n$	Linear Search ( $n$ comparisons)	Binary search ( $\log_2 n$ comparisons)
100	100	$\approx 7$
1,000	1,000	$\approx 10$
10,000	10,000	$\approx 14$
100,000	100,000	$\approx 17$
1,000,000	1,000,000	$\approx 20$
$10^9$	$10^9$	$\approx 30$

This is called  **$\log_2 n$**  algorithm, or  $O(\log_2 n)$



## 5. Introduction to Sorting (1/2)

- **Sorting** is any process of arranging items in some sequence and/or in different sets – Wikipedia.
- Sorting is important because once a set of items is sorted, many problems (such as searching) become easy.
  - Searching can be speeded up. (From linear search to binary search)
  - Determining whether the items in a set are all unique.
  - Finding the median item in a list.
  - Many others...



## 5. Introduction to Sorting (2/2)

- Problem statement:

Given a list of  $n$  items, arrange the items into ascending order.
- We will implement the list as an integer array.
- We will introduce two basic sort algorithms.
- We will count the **number of comparisons** the algorithms make to analyze their performance.
  - The ideal sorting algorithm will make the least possible number of comparisons to arrange data in a designated order.
- We will compare the algorithms by analyzing their worst-case performance.

## 6. Selection Sort (1/6)

- **Selection Sort** algorithm

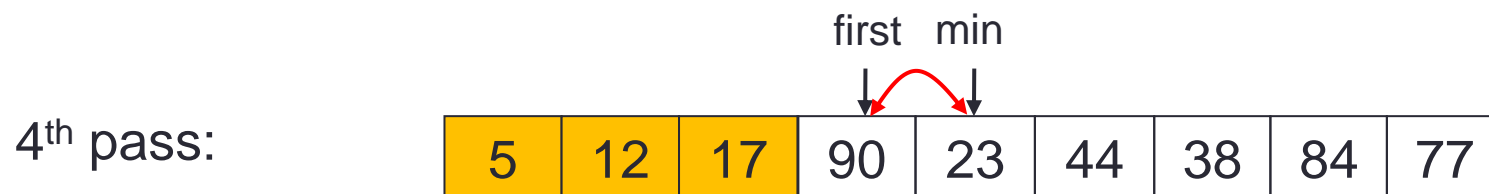
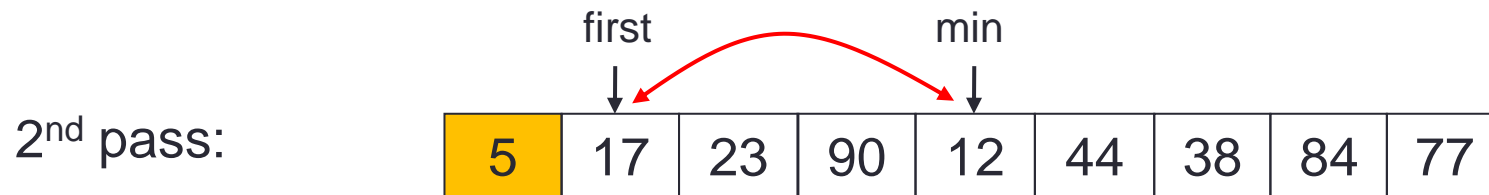
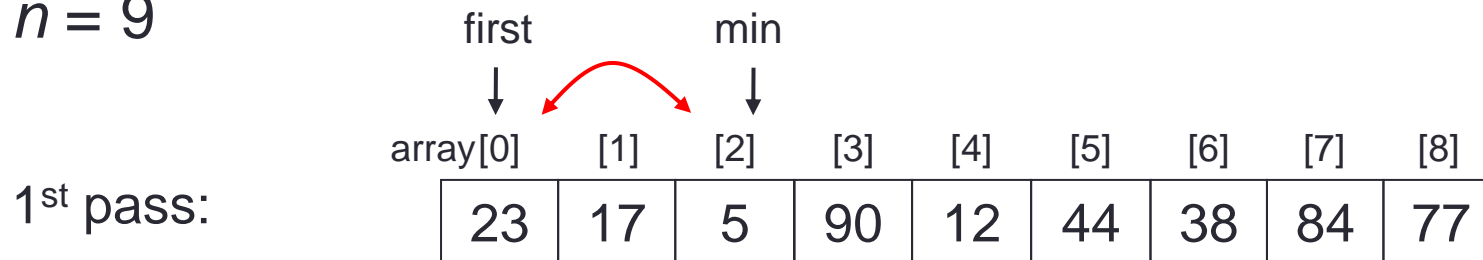
**Step 1:** Find the smallest element in the list (find\_min)

**Step 2:** Swap this smallest element with the element in the first position. (Now, the smallest element is in the right place.)

**Step 3:** Repeat steps 1 and 2 with the list having one fewer element (i.e. the smallest element just found and its place is “discarded” from further processing).

## 6. Selection Sort (2/6)

$n = 9$



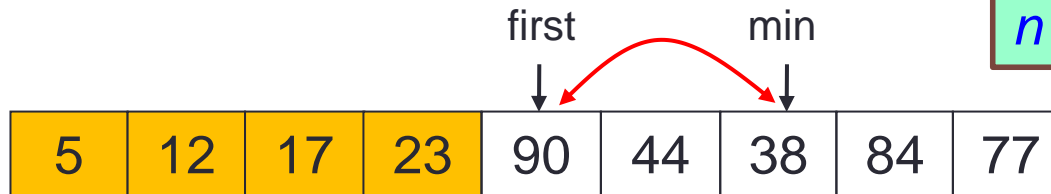
## 6. Selection Sort (3/6)

Q: How many passes for an array with  $n$  elements?

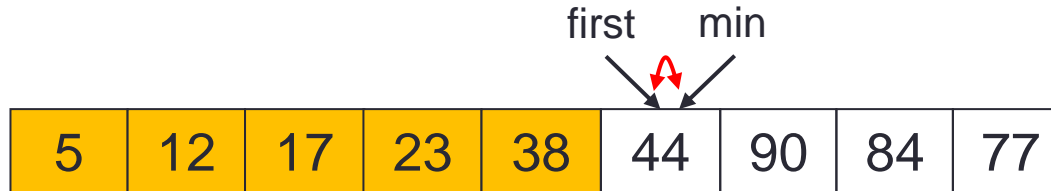
$n - 1$  passes

$n = 9$

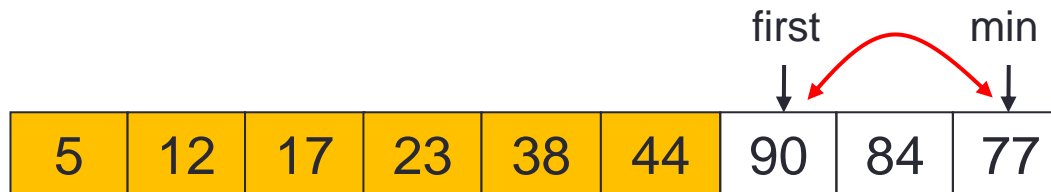
5<sup>th</sup> pass:



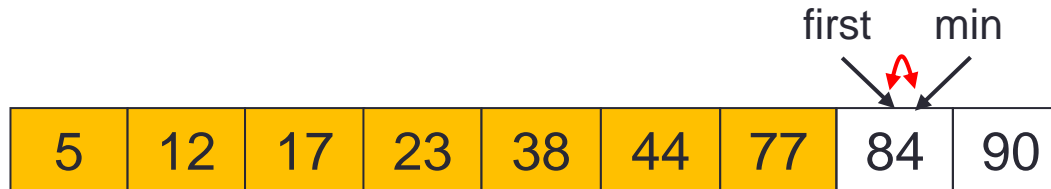
6<sup>th</sup> pass:



7<sup>th</sup> pass:



8<sup>th</sup> pass:



Final array:



## 6. Selection Sort: Demo #2 (4/6)

See [selection\\_sort.c](#) for full program

```
// To sort arr in increasing order
void selectionSort(int arr[], int size) {
    int i, start, min_index, temp;

    for (start = 0; start < size-1; start++) {
        // each iteration of the for loop is one pass

        // find the index of minimum element
        min_index = start;
        for (i = start+1; i < size; i++)
            if (arr[i] < arr[min_index])
                min_index = i;

        // swap minimum element with element at start index
        temp = arr[start];
        arr[start] = arr[min_index];
        arr[min_index] = temp;
    }
}
```

## 6. Selection Sort: Performance (5/6)

- We choose the **number of comparisons** as our basis of analysis.
- Comparisons of array elements occur in the inner loop, where the minimum element is determined.
- Assuming an array with  $n$  elements. Table below shows the number of comparisons for each pass.
- The total number of comparisons is calculated in the formula below.
- Such an algorithm is called an  **$O(n^2)$**  algorithm, or **quadratic algorithm**, in terms of running time complexity.

Pass	#comparisons
1	$n - 1$
2	$n - 2$
3	$n - 3$
...	...
$n - 1$	1

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} = \frac{n^2 - n}{2} \cong n^2$$

## 6. Selection Sort (6/6)

- Selection sort is classified under **exchange sort**, where elements are exchanged in the process.
- We could search for the minimum element as described earlier, or search for the maximum element and exchange it with the last element of the working array (assuming we sort in ascending order).



## 7. Bubble Sort (1/5)

- **Selection sort** makes one exchange at the end of each pass.
- What if we make more than one exchange during each pass?
- The key idea **Bubble sort** is to make pairwise comparisons and exchange the positions of the pair if they are in the wrong order.

# 7. Bubble Sort: One Pass of Bubble Sort (2/5)

0	1	2	3	4	5	6	7	8
23	17	5	90	12	44	38	84	77

↑ exchange

17	23	5	90	12	44	38	84	77
----	----	---	----	----	----	----	----	----

↑ exchange

17	5	23	90	12	44	38	84	77
----	---	----	----	----	----	----	----	----

↑ ok

17	5	23	12	90	44	38	84	77
----	---	----	----	----	----	----	----	----

↑ exchange

17	5	23	12	44	90	38	84	77
----	---	----	----	----	----	----	----	----

exchange ↑

17	5	23	12	44	38	90	84	77
----	---	----	----	----	----	----	----	----

exchange ↑

17	5	23	12	44	38	84	90	77
----	---	----	----	----	----	----	----	----

exchange ↑

17	5	23	12	44	38	84	77	90
----	---	----	----	----	----	----	----	----

Q: Is the array sorted?  
Q: What have we achieved?

Done!

## 7. Bubble Sort: Demo #3 (3/5)

See [bubble\\_sort.c](#) for full program

```
// To sort arr in increasing order
void bubbleSort(int arr[], int size) {
    int i, limit, temp;

    for (limit = size-2; limit >= 0; limit--) {
        // limit is where the inner loop variable i should end

        for (i=0; i<=limit; i++) {
            if (arr[i] > arr[i+1]) { // swap arr[i] with arr[i+1]
                temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
            }
        }
    }
}
```

## 7. Bubble Sort: Performance (4/5)

- Bubble sort, like selection sort, requires  $n - 1$  passes for an array with  $n$  elements.
- The comparisons occur in the inner loop. The number of comparisons in each pass is given in the table below.
- The total number of comparisons is calculated in the formula below.
- Like Selection sort, Bubble sort is also an  $O(n^2)$  algorithm, or quadratic algorithm, in terms of running time complexity.

Pass	#comparisons
1	$n - 1$
2	$n - 2$
3	$n - 3$
...	...
$n - 1$	1

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} = \frac{n^2 - n}{2} \cong n^2$$

## 7. Bubble Sort: Enhanced version (5/5)

- It is possible to enhance Bubble sort algorithm to **reduce the number of passes**.
- Suppose that in a certain pass, no swap is needed. This implies that the array is already sorted, and hence the algorithm may terminate without going on to the next pass.
- You will implement this enhanced version in your discussion session.

## 8. More Sorting Algorithms

- We have introduced 2 basic sort algorithms: **Selection Sort** and **Bubble Sort**. Together with **Insertion Sort** algorithm, these 3 are the simplest sorting algorithms.
- However, they are very slow, as their running time complexity is **quadratic**, or  $O(n^2)$ , where  $n$  is the array size.
- Faster sorting algorithms exist and are covered in more advanced modules.
- In CS2040, you will learn more advanced sorting algorithms with better running-time efficiency: **Quick Sort**, **Merge Sort**, **Radix Sort**, etc.

## 9. Animated Sorting Algorithms

- There are a number of animated sorting algorithms on the Internet.
- Here are just a few sites:
  - Visualgo (<http://visualgo.net>): Project under Dr Steven Halim
  - <http://www.sorting-algorithms.com/>
  - <http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html>
- There are also folk dances based on sorting!
  - Selection sort with Gypsy folk dance  
<http://www.youtube.com/watch?v=Ns4TPTC8whw>
  - Bubble sort with Hungarian folk dance  
<http://www.youtube.com/watch?v=lyZQPjUT5B4>



# Summary

- In this unit, you have learned about
  - 2 search algorithms: **Linear (sequential) Search** and **Binary Search**
  - 2 basic sort algorithms: **Selection Sort** and **Bubble Sort**.



End of File