

# CSC 211: Computer Programming

Sorting Algorithms, Binary Search, Big-O

Michael Conti

Department of Computer Science and Statistics  
University of Rhode Island

Spring 2025



## Administrative Announcements

- A03 due 04/01
- MC05 due 04/06
- Exam#02 ~ Tuesday, April 8th, 2025
  - ✓ Same time / place as lecture
  - ✓ One 11x8 notes sheet
  - ✓ No calculator
  - ✓ Weeks 6 - 10

2

# Basic Sorting Algorithms

## Sorting and Searching

- Two fundamental problems in CS
  - ✓ Sorting and Searching

4

## Sorting

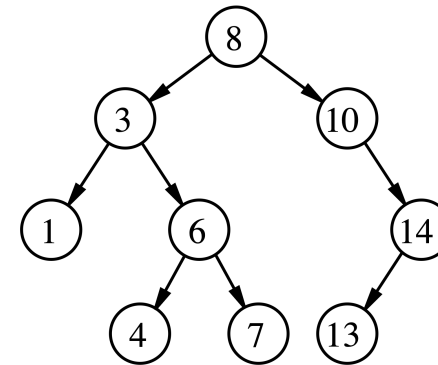
- Given an input sequence of **n** elements that can be compared to each other according to a **total order** relation
  - we want to rearrange them in non-increasing / non-decreasing order
- Example (sorting in non-decreasing order):
  - input**: array  $A = [k_0, k_1, \dots, k_{n-1}]$
  - output**: array  $B$  (permutation of  $A$ ), s.t.  $B[0] \leq \dots \leq B[n-1]$

Central problem in computer science

5

## Searching

- Given some data structure **S**, determine if some key **K** exists



6

## Efficiency

- We'll talk about efficiency in CSC 212. Why do we care?
- Computers are fast, but they can still take time to do complex actions.
- A major goal of computer scientists is not just to make algorithms that work, but algorithms that work efficiently.

7

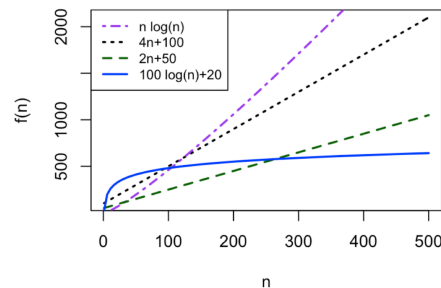
## Function Families

- When we count the actions taken by algorithms, we don't really care about one-off operations; **we care about actions that are related to the size of the input.**
- In math, a **function family** is a set of equations that all grow at the same rate as their inputs grow. (linearly v.s quadratically)
- When determining which equation family represents the actions taken by an algorithm, we say that  $n$  is the size of the input.

8

## Function Families

- As  $n$  grows, the two linear functions become larger than the  $\log(n)$  function, and then the  $n \log(n)$  function becomes larger than both linear functions, regardless of the constants.



9

## Big-O Notation

- When we determine an equation's function family, we **ignore constant factors and smaller terms**. All that matters is the dominant term (the highest power of  $n$ ). That is the idea of Big-O notation.

$f(n)$	Big-O
$n$	$O(n)$
$32n + 23$	$O(n)$
$5n^2 + 6n - 8$	$O(n^2)$
$18 \log(n)$	$O(\log n)$

10

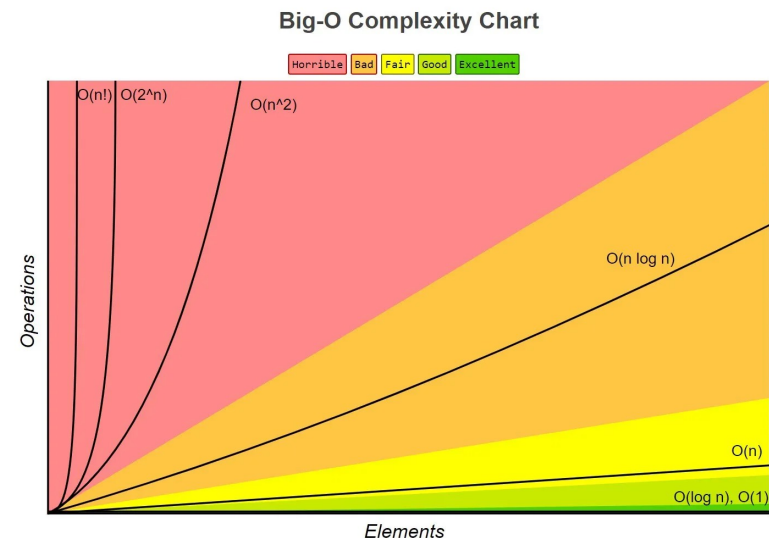
## Big-O Notation

- Unless specified otherwise, the Big-O of an algorithm refers to its worst case run time
- "Asymptotic Complexity" of algorithm

Complexity	Name	Family
$O(1)$	Constant	Sub-linear
$O(\log n)$	Logarithmic	Sub-linear
$O(n)$	Linear	Linear
$O(n \log n)$	Linearithmic	Sub-quadratic
$O(n^2)$	Quadratic	Polynomial
$O(n^3)$	Cubic	Polynomial
$O(2^n), O(n!)$	Exponential / Factorial	Super-polynomial

11

## Big-O Notation



12

## Bubble-Sort

- Basic sorting algorithm
  - yet too slow in practice
- Scan the input sequence from left-to-right
  - compare all adjacent elements and swap them if they are in the wrong order
- Repeat the scan until the list is sorted

After every pass (iteration), the smaller/larger element bubbles up to the end of the sequence

13

## Bubble Sort in 2

14

## Implement Bubble Algorithm

BUBBLESORT(*A*)

```
1 for i = 1 to A.length - 1
2   for j = A.length downto i + 1
3     if A[j] < A[j - 1]
4       exchange A[j] with A[j - 1]
```

```
void swap(int& v1, int& v2) {
    int temp = v1;
    v1 = v2;
    v2 = temp;
}
```

15

```
void swap(int& v1, int& v2) {
    int temp = v1;
    v1 = v2;
    v2 = temp;
}

void bubble(int A[], int n_elem) {
    bool sorted = false;
    while (!sorted) {
        sorted = true;
        for (int i = 0 ; i < (n_elem-1) ; i++) {
            if (A[i] > A[i+1]) {
                sorted = false;
                swap(A[i], A[i+1]);
            }
        }
    }
}

int main() {
    int array[] = {15, 12, 13, 24, 5};
    bubble(array, 5);
}
```

16

# Selection-Sort

- Basic sorting algorithm
  - yet too slow in practice
- Keep two parts: **left part** is **already sorted** and **right part** is **to be sorted**
  - initially, the sorted part is empty and the unsorted part is the input sequence
- At every iteration, find the smallest (or largest) element in the unsorted part and swap it with the leftmost unsorted element
  - then move the boundary between parts one element to the right

At every iteration we select the minimum/maximum

17

# Selection Sort in 3

18

## Implement Selection Algorithm

```
1: function SELECTION-SORT(A, n)
2:   for i = 1 to n-1 do
3:     min ← i
4:     for j = i + 1 to n do
5:       if A[j] < A[min] then
6:         min ← j
7:       end if
8:     end for
9:     swap A[i], A[min]
10:  end for
11: end function
```

19

```
void swap(int& v1, int& v2) {
    int temp = v1;
    v1 = v2;
    v2 = temp;
}

int find_min(int A[], int start, int last) {
    int min = start;
    for (int i = start + 1 ; i < last ; i++) {
        if (A[i] < A[min]) {
            min = i;
        }
    }
    return min;
}

void selection(int A[], int n_elem) {
    for (int i = 0, j ; i < (n_elem-1) ; i++) {
        j = find_min(A, i, n_elem);
        swap(A[i], A[j]);
    }
}

int main() {
    int array[] = {15, 12, 13, 24, 5};
    selection(array, 5);
}
```

20

## Insertion-Sort

- Basic sorting algorithm
  - slightly faster than bubble-sort and selection-sort
- Keep two parts: **left part** is **already sorted** and **right part** is **to be sorted**
  - initially, the sorted part contains the first element in the array and the unsorted part is the remaining elements
- At every iteration, the first element of the unsorted part is selected, and the algorithm finds the location it belongs within the sorted part, and inserts it there
  - then move the boundary between parts one element to the right
  - repeat until no elements remain in the unsorted part

21

## Insertion Sort in 2

22

## Implement Insertion Algorithm

```
ALGORITHM InsertionSort( $A[0..n-1]$ )
//Sorts a given array by insertion sort
//Input: An array  $A[0..n-1]$  of  $n$  orderable elements
//Output: Array  $A[0..n-1]$  sorted in nondecreasing order
for  $i \leftarrow 1$  to  $n-1$  do
     $v \leftarrow A[i]$ 
     $j \leftarrow i-1$ 
    while  $j \geq 0$  and  $A[j] > v$  do
         $A[j+1] \leftarrow A[j]$ 
         $j \leftarrow j-1$ 
     $A[j+1] \leftarrow v$ 
```

23

## Binary Search

## Linear search

- Two fundamental problems in CS
  - Sorting and Searching
- Linear (sequential) search** is a method for finding a **value** within a sequence
- A naive solution works by sequentially checking each element until a match is found
  - it also stops when there are no more elements to check
  - performs at most **n** comparisons for sequences of length **n**
  - considered **slow** for finding elements in collections of data

25

```
bool lin_search(int *A, int key, int len) {  
    for (int i = 0 ; i < len; i++){  
        if (A[i] == key){  
            return true;  
        }  
    }  
    return false;  
}
```

26

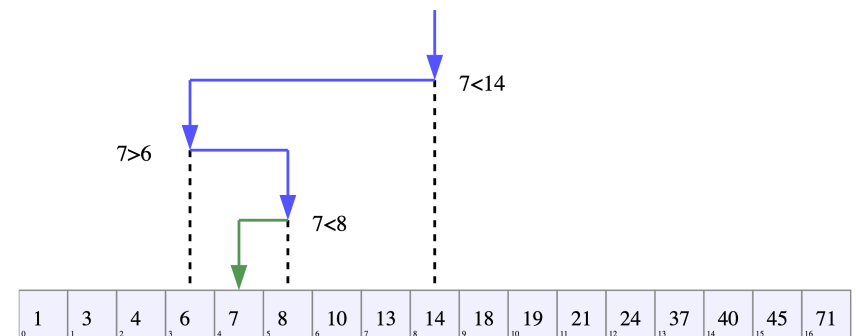
## Binary search

- Search algorithm to find a given value within a **sorted array**
- The algorithm compares the value to the middle element
  - if they are not equal, one of the half is eliminated and the search continues on the other half
  - repeat until value is found or no more elements are left (value is not in the array)
- Binary search is **faster than linear search**

27

## Binary search

k = 7



28

## Binary Search

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	5	10	15	20	22	30	35	40	43	48	51

low

high

k = 48?

29

## Binary Search

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	5	10	15	20	22	30	35	40	43	48	51

low

mid

high

k = 48?

30

## Binary Search

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	5	10	15	20	22	30	35	40	43	48	51

low

high

k = 48?

31

## Binary Search

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	5	10	15	20	22	30	35	40	43	48	51

low

mid

high

k = 48?

32



# Binary Search

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	5	10	15	20	22	30	35	40	43	48	51

low mid high



k = 48?

33

# Binary Search

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	5	10	15	20	22	30	35	40	43	48	51

low

high

k = 22?

k = 0?

k = 55?

34

```
// returns index of element k in A
// returns NOT_FOUND if element not in A
int bin_search(int *A, int lo, int hi, int k) {
    // base case
    if (hi < lo) {
        return NOT_FOUND;
    }
    // calculate midpoint index
    int mid = lo + ((hi-lo)/2);
    // key found?
    if (A[mid] == k) {
        return mid;
    }
    // key in upper subarray?
    if (A[mid] < k) {
        return bin_search(A, mid+1, hi, k);
    }
    // key is in lower subarray?
    return bin_search(A, lo, mid-1, k);
}
```

35

# Call stack

0	1	2	3	4	5	6	7	8	9
1	2	5	10	15	20	22	30	35	40

```
#define NOT_FOUND -1

int bsch(int *A, int lo, int hi, int k) {
    if (hi < lo) {
        return NOT_FOUND;
    }
    int mid = lo + ((hi-lo)/2);
    if (A[mid] == k) {
        return mid;
    }
    if (A[mid] < k) {
        return bsch(A, mid+1, hi, k);
    }
    return bsch(A, lo, mid-1, k);
}

int main() {
    int arr[] = {1,2,5,10,15,20,22,30,35,40};
    int idx = bsch(arr, 0, 9, 1);
}
```

<https://bit.ly/36cEQWK>

36



## Google Research Blog

<https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html>

The latest news from Research at Google

*"The version of binary search that I wrote for the JDK (java.util.Arrays) contained the same bug. It was reported to Sun recently when it broke someone's program, after lying in wait for nine years or so."*

### Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken

Friday, June 02, 2006

Posted by Joshua Bloch, Software Engineer

**overflow**

```
int mid = (low + high) / 2;
```

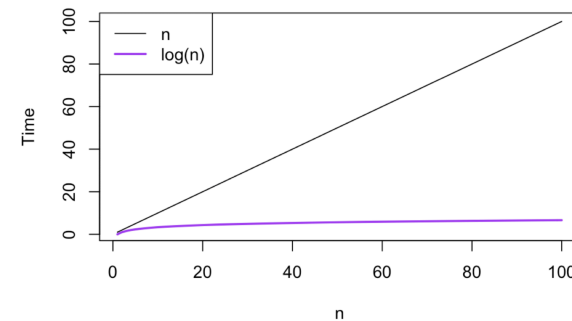
```
int mid = lo + ((hi-lo) / 2);
```



37

## Big-O of Linear Search v. Binary Search

- Because runtime for linear search is proportional to the length of the list in the worst case, it is  $O(n)$ . Every time we double the length of the list, binary search does just one more comparison; it is  $O(\log n)$ .



38

## Find peak in unimodal arrays

### Unimodal arrays

- An array is (**strongly**) **unimodal** if it can be split into an increasing part followed by a decreasing part

1	2	5	16	20	18	17	16	15	12	10	8	5
---	---	---	----	----	----	----	----	----	----	----	---	---

- An array is (**weakly**) **unimodal** if it can be split into a nondecreasing part followed by a nonincreasing part

1	2	5	5	15	20	22	22	35	38	13	8	5
---	---	---	---	----	----	----	----	----	----	----	---	---

40

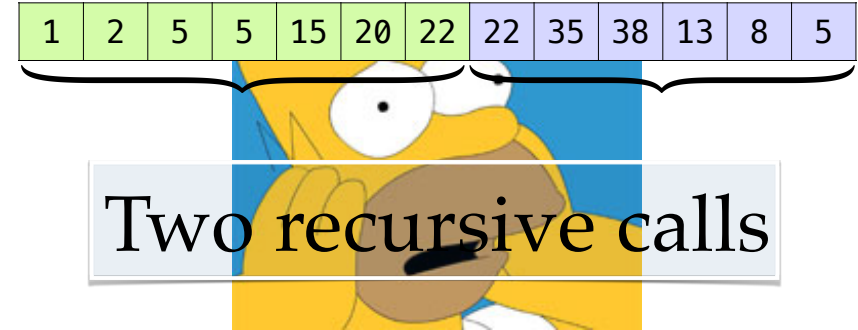
## Find the peak (strongly unimodal)

1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5

41

## Find the peak (weakly unimodal)

1	2	5	5	15	20	22	22	35	38	13	8	5
1	2	5	5	15	20	22	22	35	38	13	8	5
1	2	5	5	15	20	22	22	35	38	13	8	5



Two recursive calls

42