Introduction to Computer Science

Reichman University

Lecture 9-2

Efficiency, Searching, Sorting

Lecture plan

- Program efficiency
- Searching algorithms
- Sorting algorithms
- Graph algorithms (a taste)

Lecture plan

Program efficiency:

So far in the course we treated it informally and casually...

From now on we'll treat it formally and seriously.

Two kinds of program efficiency

- *Time efficiency*: How fast is the program?
- *Space efficiency*: How much memory does it consume?

Time efficiency

<u>Two approaches of studying time efficiency</u>

Timing the program

Analyzing the algorithm

Timing program's efficiency

Timer API (useful tool)

```
/** Represents a timer.
 *   Features methods for time measurements. */
public class Timer {

    /** Constructs a timer and sets it to the current time. */
    public Timer()

    /** Resets this timer. */
    public void reset()

    /** Returns how many milliseconds elapsed between
     *  the last reset of this timer and the current time. */
    public double elapsedTime()
}
```

Client code example:
(can appear in any class)

```
// Starts a timer
Timer timer = new Timer();

// Running some mission-critical method we wish to time
contains(arr, x)

// Checking how much time it took
System.out.println("Time to find out: " + timer.elapsedTime());
```

Timing program's efficiency

```
/** Times various operations on my computer. */
public class TimeOps{

    public static void main(String args[]) {
        int sum = 0;
        double d = 1.618;
        BankAccount bobAcct = new BankAccount("bob");

        // Starts a timer
        Timer timer = new Timer();

        // Runs an operation, a billion times
        for (int i=0; i<1000000000; i++) {
            ////  Uncomment the operation you wish to time:
            // 1. Do nothing
            // 2. sum = sum+i;
            // 3. d = 1.0/d;
            // 4. bobAcct.deposit(1000);
            // 5. BankAccount b = new BankAccount("Foo");
            // 6. Fraction f = new Fraction(i,i+1);
            // 7. d = Math.random();
            // 8. System.out.println(i);
        }

        // The loop's running time, in milliseconds
        System.out.println(timer.elapsedTime());
    }
}
```

	Shimon's old PC (2018)	Shimon's newer PC (2021)	Kfir' new Apple (2025)
1.	5 ms	2 ms	2 ms
2.	389	298	2
3.	4898	3811	1
4.	57	47	740
5.	1089	1088	3
6.	10500	4741	1897
7.	25310	21526	8177
8.	5510222	5443566	5443566

Timing program's efficiency

Limitations

Any one of the following factors impacts the timing results:

- Different hardware
- Different OS
- Different compiler
- And more.

	Shimon's old PC (2018)	Shimon's newer PC (2021)	Kfir' new Apple (2025)
1.	5 ms	2 ms	2 ms
2.	389	298	2
3.	4898	3811	1
4.	57	47	740
5.	1089	1088	3
6.	10500	4741	1897
7.	25310	21526	8177
8.	5510222	5443566	5443566

Timing program's efficiency

Limitations

Any one of the following factors impacts the timing results:

- Different hardware
- Different OS
- Different compiler
- And more.

Needed: A *platform-independent* way to measure time efficiency.

Time efficiency

<u>Two approaches of studying time efficiency</u>

Timing the program

Analyzing the algorithm

Running time analysis

Example 1: Searching an array of size N

```
// Checks if the given array contains the given value
boolean contains(int[] arr, int x) {

   int N = arr.length;

   for (int i=0; i<N; i++)

      if (arr[i]==x)

         return true;

   return false;
}
```

Running time analysis

Example 1: Searching an array of size N

```
// Checks if the given array contains the given value
boolean contains(int[] arr, int x) {
                 c0
   int N = arr.length;
               c1     c2    c3
   for (int i=0; i<N; i++)
                   c4
      if (arr[i]==x)
                c5
         return true;
        c6
   return false;
}
```

Identify the program's operations

<u>How many operations are actually executed during run-time?</u>

Depends on N = input size

Running time analysis

Example 1: Searching an array of size N

```
// Checks if the given array contains the given value
boolean contains(int[] arr, int x) {
         c0
   int N = arr.length;
               c1     c2     c3
   for (int i=0; i<N; i++)
                   c4
      if (arr[i]==x)
              c5
         return true;
      c6
   return false;
}
```

How many operations are actually executed during run-time?

Depends on N = input size

Running time (in this particular problem)

Best case: x is the first element in the array

Worst case: x is not in the array

Average case: x is somewhere in the array

Our approach to running time analysis

Focus on the *worst case*

When we say “running time”, we mean *running time in the worst case*

The most conservative, honest approach.

Running time analysis

Example 1: Searching an array of size N

```
// Checks if the given array contains the given value
boolean contains(int[] arr, int x) {
             c0
   int N = arr.length;
                c1    c2    c3
   for (int i=0; i<N; i++)
                   c4
      if (arr[i]==x)
                c5
         return true;
       c6
   return false;
}
```

<u>How many operations are actually executed during run-time?</u>

Depends on N = input size

$RunningTime(N)$ = (in the worst case)

$$c0 + c1 + (c2 + c3 + c4) \cdot N + c6 =$$

Assuming that each operation takes more or less the same time, c:

$$\mathbf{3c + 3c \cdot N}$$

(Note: the fact that 3 appears twice in this polynomial is just a coincidence)

Running time analysis

Example 1: Searching an array of size N

```
// Checks if the given array contains the given value
boolean contains(int[] arr, int x) {
              c0
   int N = arr.length;
                 c1    c2    c3
   for (int i=0; i<N; i++)
                    c4
      if (arr[i]==x)
                 c5
         return true;
         c6
   return false;
}
```

<u>How many operations are actually executed during run-time?</u>

Depends on N = input size

$RunningTime(N)$ = (in the worst case)

$$c0 + c1 + (c2 + c3 + c4) \cdot N + c6 =$$

Assuming that each operation takes more or less the same time, c:

$$\mathbf{3c + \underline{3c \cdot N}}$$

Observation: The term that grows fastest when N increases: $\mathbf{3c \cdot N}$

<u>Big *O*</u>

- Focus on the term that *grows fastest as the input size* ($\boldsymbol{N}$) *increases*
- Running time $\overset{def}{=}$ order of magnitude of this term (ignoring all constants): $\boldsymbol{O(N)}$
- **Big *O*:** a computer science standard for describing an algorithm's running time.

Running time analysis

Example 2: Searching a 2D array of size N by N

```
// Checks if the given array contains the given value
Boolean contains(int[][] arr, int x) {

   int N = arr.length;

   for (int i=0; i<N; i++)

      for (int j=0; j<N; j++)

         if (a[i][j]==x)

            return true;

   return false;
}
```

Running time analysis

Example 2: Searching a 2D array of size N by N

```
// Checks if the given array contains the given value
Boolean contains(int[][] arr, int x) {
              c0
   int N = arr.length;
                  c1     c2     c3
   for (int i=0; i<N; i++)
                    c4     c5     c6
      for (int j=0; j<N; j++)
                          c7
         if (a[i][j]==x)
                    c8
            return true;
       c9
   return false;
}
```

Assumption: Each operation takes more or less the same time, *c*

$RunningTime(N) =$

$c0 + c1 +$

$(c2 + c3) \cdot N + c4 \cdot N +$

$(c5 + c6) \cdot N^2 + c7 \cdot N^2 + c9 =$

Assuming that each operation takes more or less the same time, c:

$$\mathbf{3c + 3c \cdot n + \underline{3c \cdot N^2}}$$

<u>The term that grows fastest</u> when n increases: $\mathbf{3c \cdot n^2}$

<u>Running time</u>

Focus on the term that *grows fastest as **N** increases*

Running time $\overset{def}{=}$ order of magnitude of this term (ignoring all constants): $\boldsymbol{O(N^2)}$

Running time analysis

Example 3: Parity

```
// Checks the parity of the given number
Boolean isEven(int n) {
   return (n % 2) == 0;
}
```

Running time

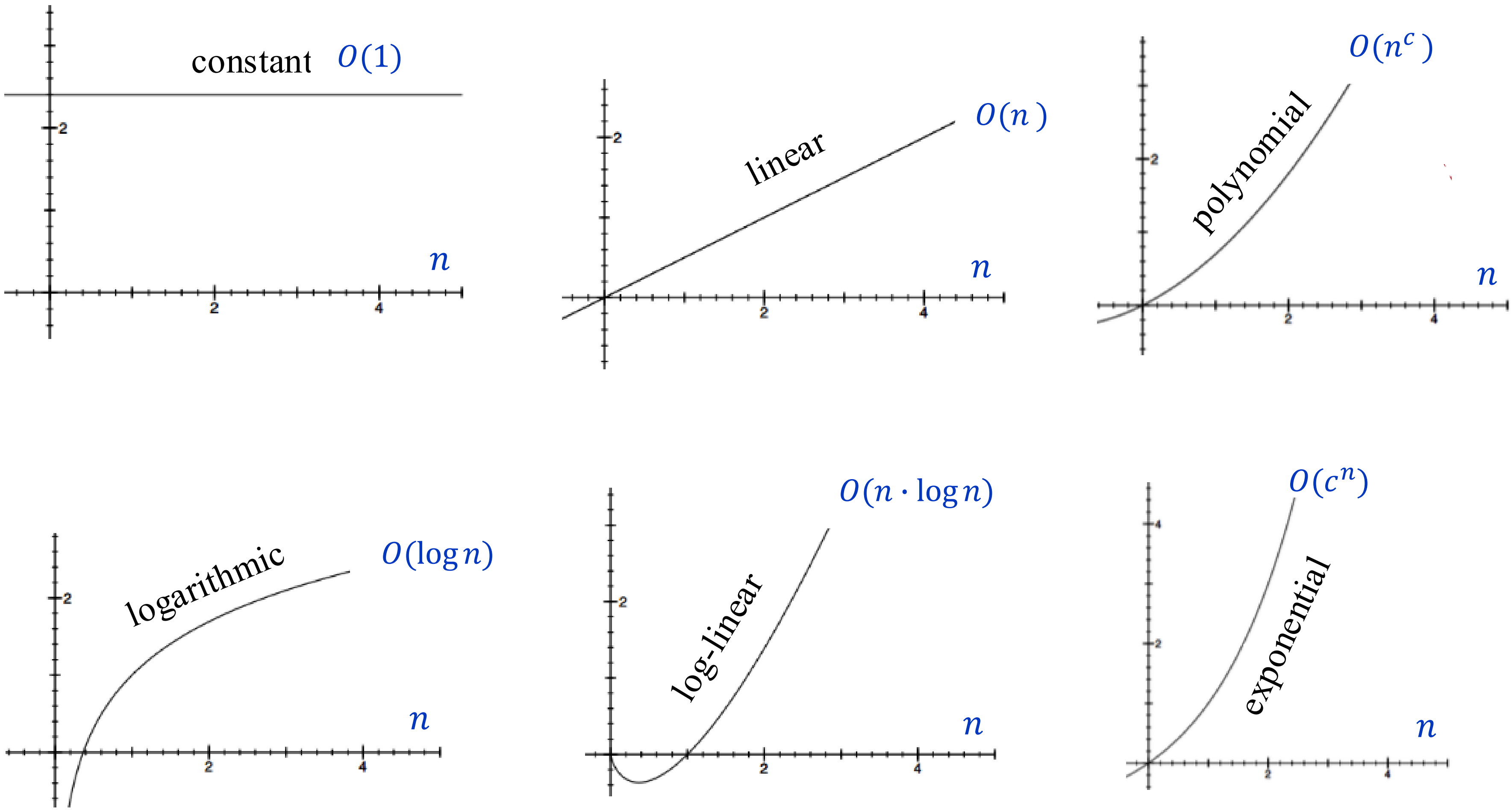
- Does not depend on the input size (n)
- $\text{RunningTime}(n) \overset{def}{=} \boldsymbol{O(1)}$

Common running time functions

<u>Examples</u>

$$
\left.
\begin{array}{ll}
27 & O(1) \\
519 \cdot n + 1734 & O(n) \\
16 \cdot n^2 + 5 \cdot n + 9 & O(n^2) \\
n^3 + 1000 \cdot n^2 & O(n^3) \\
200 \cdot \log(n) + 15 & O(\log n) \\
7 \cdot n \cdot \log(n) + 500 \cdot n & O(n \cdot \log n) \\
2 \cdot n^{10} + 2^n & O(2^n)
\end{array}
\right\} \; n = \textit{input size}
$$

Common running time functions



Linear VS constant running time

Removing element *i* from an ordered array with *size* = *n* elements:

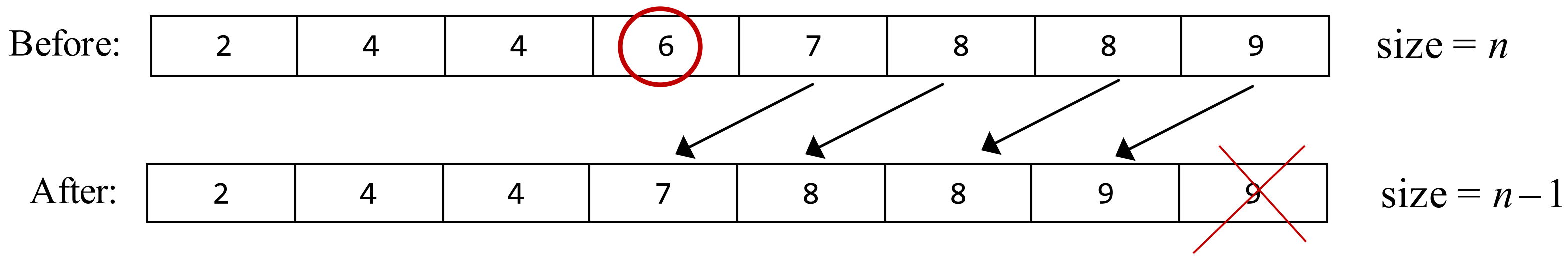
2	4	4	6	7	8	8	9	size = *8*

The algorithm's input is an object consisting of three fields: `arr`, the array itself, `maxSize`, the fixed length of the array, and `size`, the number of elements that are presently in the array. Thus the actual input size is *`size`*.

In the example above the array is presently full: `size = maxSize`

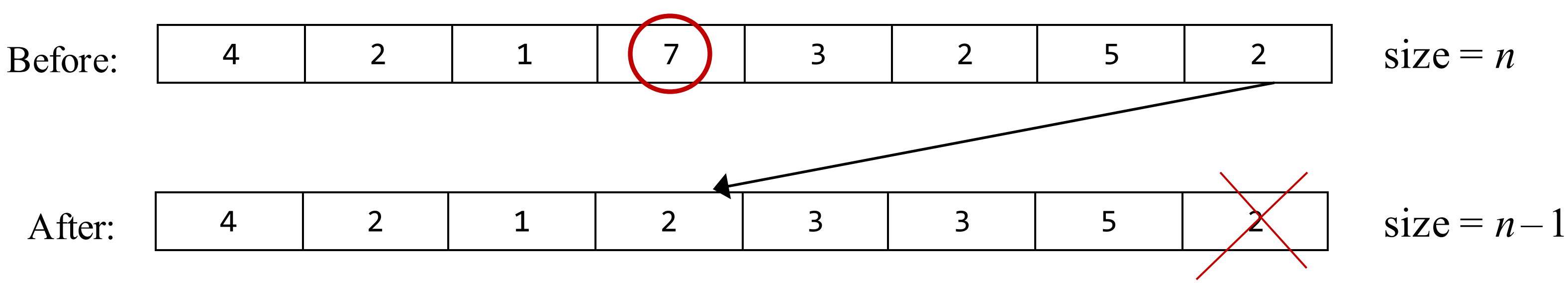
Linear VS constant running time

Removing element i from an ordered array with $size = n$ elements:



$RunningTime(n) = \boldsymbol{O(n)}$ **Can we do better?**

Removing element i from an **unordered** array with $size = n$ elements:



$RunningTime(n) = \boldsymbol{O(1)}$

Lecture plan

- Program efficiency
- Search algorithms
  - Sequential search
  - Binary search
- Sorting algorithms
- Graph algorithms

Context example: Search engine

mountain biking morocco | Search

About 403,000 results (0.15 seconds)

[**Mountain Biking**, Walking Tours and Tailor Made Holidays in **Morocco**]
Group **mountain biking** and walking tours in and around the Atlas Mountains **...**
www.epic**morocco**.co.uk/tours.html - Cached - Similar
Show more results from epicmorocco.co.uk

[**Mountain Biking** Holidays in Morzine, **Morocco**, Megavalanche, La **...**]
flowmtb: **Mountain Bike** Holidays. Catered Chalet holidays in Morzine Les Gets, and Alpe d'Huez France. Guided trips in **Morocco**. Downhill, Cross Country **...**
www.flowmtb.com/ - United Kingdom - Cached - Similar

[**Mountain Biking** Holidays in Morzine, **Morocco**, Megavalanche, La **...**]
flowmtb: **mountain biking** trips in **Morocco**. Ride the best of the Atlas Mountains.
www.flowmtb.com/**morocco**/unknown-**morocco**/ - United Kingdom - Cached - Similar
Show more results from flowmtb.com

[**Mountain biking** in **Morocco** with Wildcat]
The Anti Atlas Mountains in the South West of the country is the perfect location to enjoy your adventure **mountain biking** during the winter period. **...**
www.wildcat-bike-tours.co.uk/.../**Mountain-biking-morocco**/index.htm - Cached - Similar

[**Bike** Tours Since 1985. **Morocco** Road Tuareg Trail - Anti Atlas **...**]
Morocco Road and **Mountain bike** Tours - The Tuareg Trail. The Anti Atlas **...**
www.wildcat-bike-tours.co.uk/**morocco**-tuareg.html - Cached - Similar

[Bike Tours Since 1985. **Morocco Mountain Bike** Tour - The Tuareg Trail]
Morocco MTB Tour The Tuareg Trail - Anti Atlas Mountains. The Anti Atlas **...**
www.wildcat-bike-tours.co.uk/**morocco**-tuareg-mtb.htm - Cached

Context example: Search engine

The search engine's data

A list of words,
each associated with the URLs
of web pages that include it;

Stored on many servers and maintained
continuously by hard-working robots.

word	URLs
ninex	1955, 21
moize	2, 11
morgan	13, 100,
namibia	95
mormon	1 ,7
morning	4, 83
nancy	5, 17
morocco	12, 4
mortal	1, 7, 4, 5
Mortgage	17
nalini	5, 17
mountain	81, 9
mohican	11, 4, 5
nader	10, 3, 5, 4
name	5, 11, 12,
never	5, 17
nike	3, 51, 7,
nitro	1955, 21

… n words

Context example: Search engine

morocco	search

word	URLs
ninex	1955, 21
moize	2, 11
morgan	13, 100,
namibia	95
mormon	1 ,7
morning	4, 83
nancy	5, 17
morocco	12, 4
mortal	1, 7, 4, 5
Mortgage	17
nalini	5, 17
mountain	81, 9
mohican	11, 4, 5
nader	10, 3, 5, 4
name	5, 11, 12,
never	5, 17
nike	3, 51, 7,
nitro	1955, 21

… n words

Typical search scenario

1. User enters a word
2. The search engine searches the word in the list
3. Returns the URL's, sorted by PageRank

Critical success factors:

- ➡ Fast searching
- Fast sorting.

Sequential search

morocco	search

word	URLs
ninex	1955, 21
moize	2, 11
morgan	13, 100,
namibia	95
mormon	1 ,7
morning	4, 83
nancy	5, 17
morocco	12, 4
mortal	1, 7, 4, 5
Mortgage	17
nalini	5, 17
mountain	81, 9
mohican	11, 4, 5
nader	10, 3, 5, 4
name	5, 11, 12,
never	5, 17
nike	3, 51, 7,
nitro	1955, 21

… n words

Running time $= O(n)$

Sequential search

```
// Returns the index of x in arr, or -1 if not found
public static int indexOf(int x, int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] == x) {
            return i;
        }
    }
    // Value not found
    return -1;
}
```

0	1	1	3	4	5	6	7	8	9	
97	51	72	83	20	7	9	15	91	2	`arr`

Worst case: `x` is not in `arr`

(similar to the `contains` method we saw earlier)

Running time $= O\,(n)$

Can we do better?

Binary search

morocco | search

word	URLs
mohican	11, 4, 5
moize	2, 11
morgan	13, 100,
mormon	1 ,7
morning	4, 83
morocco	12, 4
mortal	1, 7, 4, 5
mortgage	17
mountain	81, 9
nader	10, 3, 5, 4
nalini	9
name	5, 11, 12,
namibia	95
nancy	17, 2, 8
never	5, 17
nike	3, 51, 7,
ninex	9, 1
nitro	1955, 21

… n words

The data is sorted

Binary search

morocco | search

word	URLs
mohican	11, 4, 5
moize	2, 11
morgan	13, 100,
mormon	1 ,7
morning	4, 83
morocco	12, 4
mortal	1, 7, 4, 5
mortgage	17
mountain	81, 9
nader	10, 3, 5, 4
nalini	9
name	5, 11, 12,
namibia	95
nancy	17, 2, 8
never	5, 17
nike	3, 51, 7,
ninex	9, 1
nitro	1955, 21

… n words

?
"morocco" <= "mountain"

1

The data is sorted

Binary search

morocco search

word	URLs
mohican	11, 4, 5
moize	2, 11
morgan	13, 100,
mormon	1 ,7
morning	4, 83
morocco	12, 4
mortal	1, 7, 4, 5
mortgage	17
mountain	81, 9
nader	10, 3, 5, 4
nalini	9
name	5, 11, 12,
namibia	95
nancy	17, 2, 8
never	5, 17
nike	3, 51, 7,
ninex	9, 1
nitro	1955, 21

… n words

2: "morocco" $\overset{?}{<=}$ "mormon"

1: "morocco" $\overset{?}{<=}$ "mountain"

The data is sorted

Binary search

morocco | search

word	URLs
mohican	11, 4, 5
moize	2, 11
morgan	13, 100,
mormon	1 ,7
morning	4, 83
morocco	12, 4
mortal	1, 7, 4, 5
mortgage	17
mountain	81, 9
nader	10, 3, 5, 4
nalini	9
name	5, 11, 12,
namibia	95
nancy	17, 2, 8
never	5, 17
nike	3, 51, 7,
ninex	9, 1
nitro	1955, 21

… n words

2: “morocco” $\overset{?}{<=}$ “mormon”

3: “morocco” $\overset{?}{<=}$ “morocco”

1: “morocco” $\overset{?}{<=}$ “mountain”

In each iteration we divide the size of the search space by 2

How many times can we divide n by 2?

Running time $= \boldsymbol{O(\log n)}$

The data is sorted

Binary search

```
// Returns the index of x in a sorted arr, or -1 if not found
// Precondition: arr must be sorted
public static int indexOf(int x, int[] arr) {
```

0	1	1	3	4	5	6	7	8	9	
2	7	9	15	20	51	72	83	91	97	arr

Running time = $O(\log n)$

Binary search

```
// Returns the index of x in a sorted arr, or -1 if not found
// Precondition: arr must be sorted
public static int indexOf(int x, int[] arr) {
    int low = 0;
    int high = arr.length - 1;
    while (low <= high) {
        int med = (low + high) / 2;
        if (arr[med] == x) {
            return med;
        }
        if (x < arr[med]) {
            high = med - 1;
        } else {
            low = med + 1;
        }
    }
    // Value not found
    return -1;
}
```

0	1	1	3	4	5	6	7	8	9	
2	7	9	15	20	51	72	83	91	97	`arr`

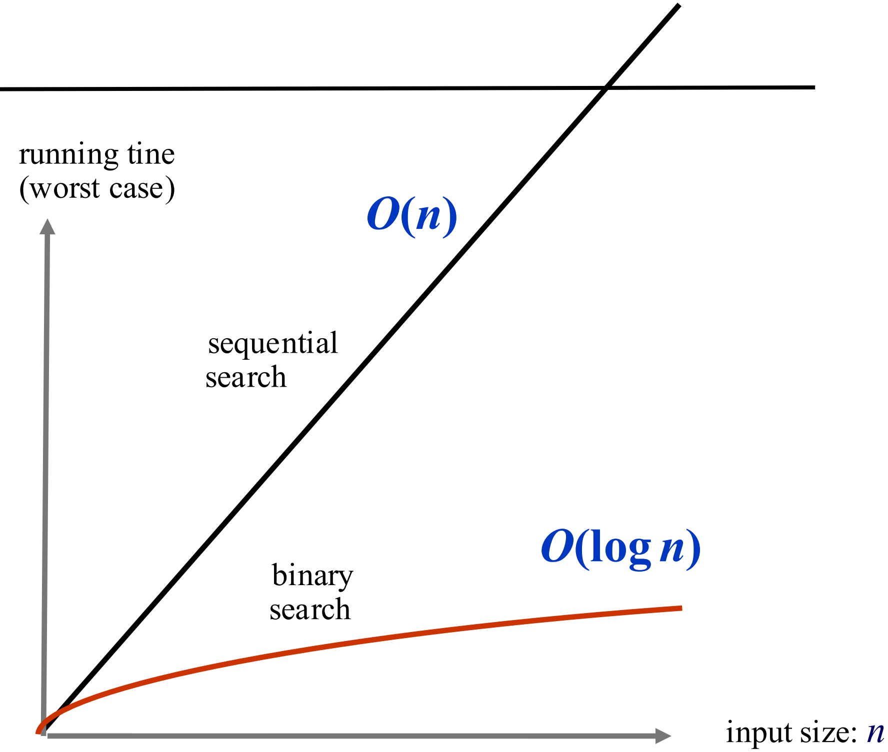
Simulating `indexOf(72, arr)`:

Iteration	`low`	`high`	`med`	`arr[med]`	test
0	0	9	4	20	72 > 20 ?
1	5	9	7	83	72 < 83 ?
2	5	6	5	51	72 > 51 ?
3	6	6	6	72	72 = 72

Running time $= \boldsymbol{O(\log n)}$

Binary vs sequential search

Input size: n	Seq. search RunningTime $O(n)$	Binary search $O(\log_2 n)$
8	8	3
16	16	4
32	32	5
64	64	6
100	100	7
1,000	1,000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30



Why is logarithmic running time attractive? Because $\log_2(2n) = \log_2 n + 1$

<u>Implications for search engines:</u>

As the size of the Internet **doubles**, each search requires **one more iteration.**

Lecture plan

- Program efficiency
- Search algorithms
  - Sequential search
  - Binary search
- Sorting algorithms
  - Selection sort
  - Insertion sort
  - Radix sort
- Graph algorithms

Sorting

Before:

13	10	7	3	15	1	4	11

After in-place sorting:

After:

1	3	4	7	10	11	13	15

There are many sorting algorithms, of which we'll discuss three.

Selection sort

```
// Sorts an array of length n
for i = 0 .. n - 1
   min = i
   for k = i + 1 .. N
      if (a[k] < a[min])
         min = k
   swap a[i], a[min]
```

(informal pseudo-code)

Analysis

step 0: we scan n elements

step 1: we scan $n - 1$ elements

step 2: we scan $n - 2$ elements

. . .

Total number of iterations =

$$n + (n - 1) + \ldots + 1 = \tfrac{1}{2}\, n\,(n - 1) = \tfrac{1}{2}\, n^2 - \tfrac{1}{2}\, n$$

Running time = $\boldsymbol{O(n^2)}$

step 0:	13	10	7	3	15	1	4	11
step 1:	1	10	7	3	15	13	4	11
step 2:	1	3	7	10	15	13	4	11
step 3:	1	3	4	10	15	13	7	11
. . .	1	3	4	7	15	13	10	11

Selection algorithm:

Find the minimum, swap with the next left-most element

Insertion sort

```
// Sorts array of length n
for j = 1 .. n – 1
   k = j – 1
   while (a[k] > a[j] and k >= 0)
      swap a[k + 1], a[k]
      k--
```

Analysis

How many swaps we have to do in the worst case?

$1 + 2 + 3 + \ldots + n = \frac{1}{2}\, n\,(n + 1)$

Running time = $O(n^2)$

step 0:	10	2	14	5	4	18	3	7
step 1:	2	10	14	5	4	18	3	7
step 2:	2	10	14	5	4	18	3	7
step 3:	2	5	10	14	4	18	3	7
. . .	2	4	5	10	14	18	3	7

Insertion algorithm:

Going from left to right:

For each element, swap with the element on the left, until the element reaches its place

Simple sort algorithms

- Selection sort
- Insertion sort
- Bubble sort
- . . .

There are many similar sorting algorithms, all based on pairwise comparisons

All $O(n^2)$ . . .

Can we do better?

Radix sort

Input: 230, 34, 76, 80, 735, 21, 4, 59

Set up: Create an array of 10 empty lists, one for each digit:

```
0:  ( )
1:  ( )
2:  ( )
3:  ( )
4:  ( )
5:  ( )
6:  ( )
7:  ( )
8:  ( )
9:  ( )
```

Radix sort

Input: 230, 34, 76, 80, 735, 21, 4, 59

Step 0: Add the values to the lists, according to their least significant (right-most) digit:

230, 34, 76, 80, 735, 21, 4, 59

```
0:  (230, 80)
1:  (21)
2:  ( )
3:  ( )
4:  (34, 4)
5:  (735)
6:  (76)
7:  ( )
8:  ( )
9:  (59)
```

Remove the values from the lists, preserving their order:

Output: 230, 80, 21, 34, 4, 735, 76, 59

Radix sort

Input: 230, 80, 21, 34, 4, 735, 76, 59 (Output of previous stage)

```
0:  ( )
1:  ( )
2:  ( )
3:  ( )
4:  ( )
5:  ( )
6:  ( )
7:  ( )
8:  ( )
9:  ( )
```

Radix sort

Input: 230, 80, 21, 34, 4, 735, 76, 59 (Output of previous stage)

Step 1: Add the values to the lists, according to their next right-most digit:

230, 080, 021, 034, 004, 735, 076, 059

```
0:  (4)
1:  ( )
2:  (21)
3:  (230, 34, 735)
4:  ( )
5:  (59)
6:  ( )
7:  (76)
8:  (80)
9:  ( )
```

Remove the values from the lists, preserving their order:

Output: 4, 21, 230, 34, 735, 59, 76, 80

Radix sort

Input: 4, 21, 230, 34, 735, 59, 76, 80 (Output of previous stage)

```
0:  ( )
1:  ( )
2:  ( )
3:  ( )
4:  ( )
5:  ( )
6:  ( )
7:  ( )
8:  ( )
9:  ( )
```

Radix sort

Input: 4, 21, 230, 34, 735, 59, 76, 80 (Output of previous stage)

Step 2: Add the values to the lists, according to their next right-most digit:

004, 021, 230, 034, 735, 059, 076, 080

```
0:  (4, 21, 34, 59, 76, 80)
1:  ( )
2:  (230)
3:  ( )
4:  ( )
5:  ( )
6:  ( )
7:  (735)
8:  ( )
9:  ( )
```

Time complexity:

$RunningTime = \boldsymbol{O(kN)}$

where k is the number of digits of the maximal value

Space complexity:

Storing the 10 lists

Remove the values from the lists, preserving their order:

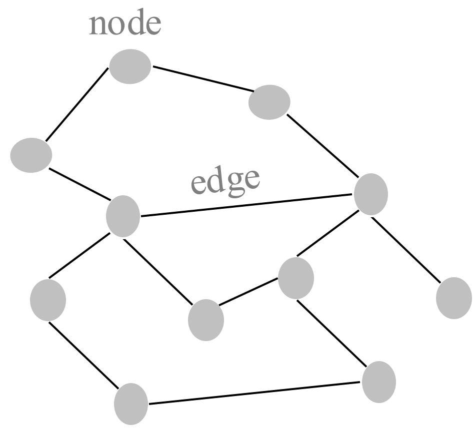
Output: 4, 21, 34, 59, 76, 80, 230, 735

sorted

Lecture plan

- Program efficiency
- Search algorithms
  - Sequential search
  - Binary search
- Sorting algorithms
  - Selection sort
  - Insertion sort
  - Radix sort
- Graph algorithms (a taste)

Graph algorithms



Input size: N nodes

Euler path: Is there a path that visits *every edge* exactly once?

There is a simple algorithm that answers this question in linear time: $O(N)$

Hamilton path: Is there a path that visits *every node* exactly once?

So far, the only algorithms that answer this question run in exponential time: $O(2^N)$

RUNI's **Algorithms course** focus: *Graph Theory*

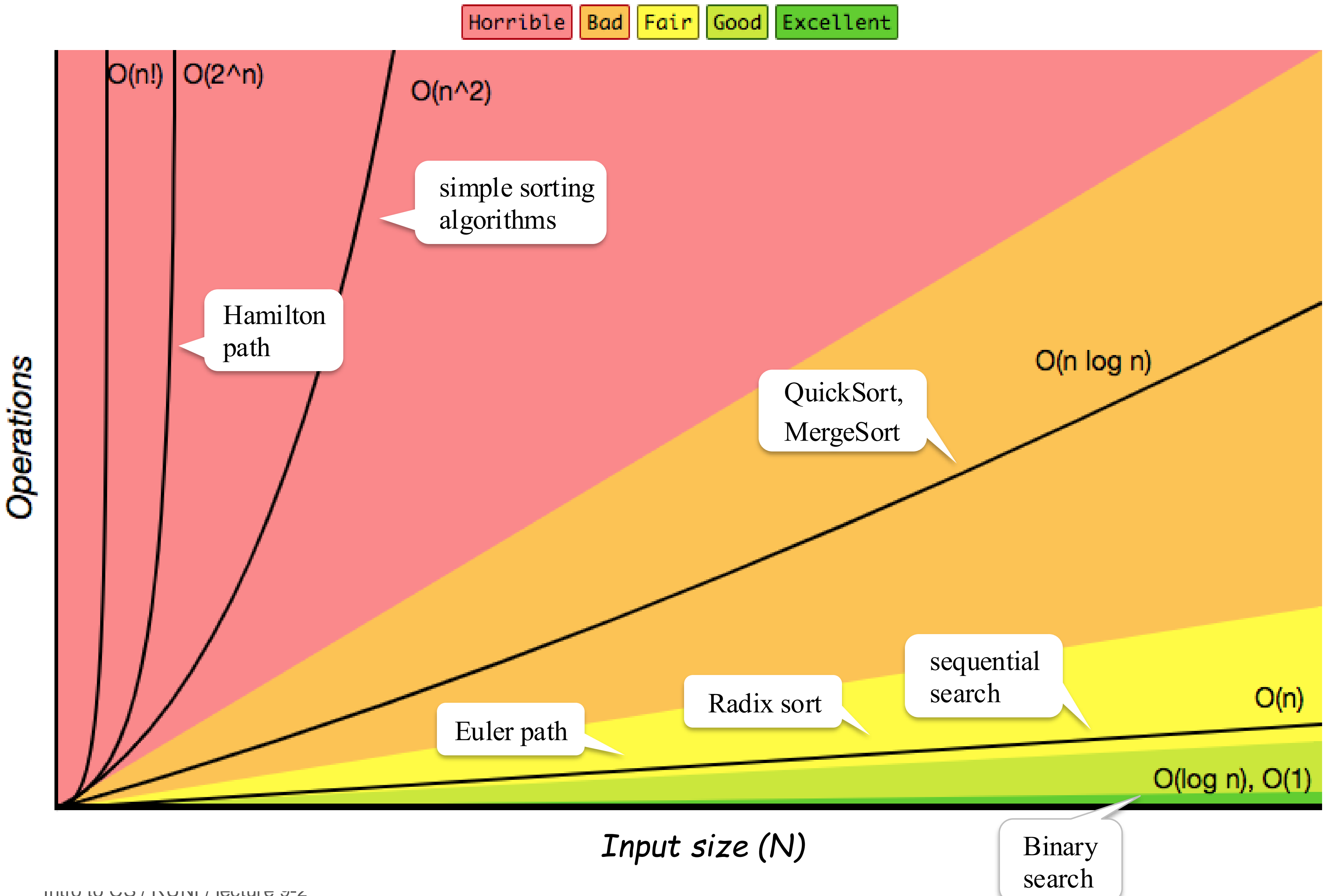
Recap

Algorithms we discussed:

- Removing an element from an array (ordered): $O(n)$
- Removing an element from an array (unordered): $O(1)$
- Sequential search $O(\mathrm{n})$
- Binary search $O(\log n)$
- Selection sort $O(n^2)$
- Insertion sort $O(n^2)$
- Radix sort $O(kn)$
- Euler path $O(\mathrm{n})$
- Hamilton path $O(2^N)$

And... we now have a formal language for classifying algorithms according to their performance.

Common running time functions



Appendix: The **Timer** class

```
/** Represents a timer.
 *   Features methods for time measurements. */
public class Timer {

    // The time at which this timer started running
    private long startTime;

    /** Constructs a timer and sets it to the current time. */
    public Timer() {
        reset();
    }

    /** Resets this timer. */
    public void reset() {
        startTime = System.currentTimeMillis();
    }

    /** Returns how many milliseconds elapsed between
     *  the last reset of this timer and the current time. */
    public double elapsedTime() {
        return (System.currentTimeMillis() - startTime);
    }
}
```

A Java function, returns the time difference, in milliseconds, between 00:00, 1/1/1970, and the current time.