

# **Lecture 21: Searching and Sorting**

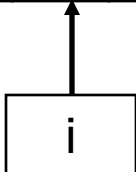
Building Java Programs: A Back to Basics Approach  
by Stuart Reges and Marty Stepp

Copyright (c) Pearson 2013.  
All rights reserved.

# Sequential search

- **sequential search:** Locates a target value in an array/list by examining each element from start to finish.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



- Notice that the array is sorted. Could we take advantage of this?

# Binary search (13.1)

- **binary search:** Locates a target value in a *sorted* array/list by successively eliminating half of the array from consideration.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

min

mid

max

# The Arrays class

- Class `Arrays` in `java.util` has many useful array methods:

Method name	Description
<code>binarySearch(array, value)</code>	returns the index of the given value in a <i>sorted</i> array (or $< 0$ if not found)
<code>copyOfRange(array, index1, index2)</code>	returns a new resized copy of an array starting with <code>index1</code> to <b><code>index2-1</code></b>
<code>equals(array1, array2)</code>	returns <code>true</code> if the two arrays contain same elements in the same order
<code>fill(array, value)</code>	sets every element to the given value
<code>sort(array)</code>	arranges the elements into sorted order
<code>toString(array)</code>	returns a string representing the array, such as <code>"[10, 30, -25, 17]"</code>

# Using `binarySearch`

```
// index    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
int[] a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92};

int index  = Arrays.binarySearch(a, 42);    // index1 is 10
int index2 = Arrays.binarySearch(a, 21);    // index2 is -7
```

- `binarySearch` returns the index where the value is found
- if the value is *not* found, `binarySearch` returns:
  - (`insertionPoint` + 1)
  - where `insertionPoint` is the index where the element *would* have been, if it had been in the array in sorted order.
  - To insert the value into the array, negate (`returnedValue` + 1)  
`int indexToInsert21 = -(index2 + 1); // 6`

# Recursive binary search (13.3)

- Write a recursive `binarySearch` method.
  - If the target value is not found, return its negative insertion point.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

```
int index  = binarySearch(data, 42);  // 10
int index2 = binarySearch(data, 66);  // -1
```

**Note:** Recursive version does not return  
– `(insertionPoint+1)`.

# Exercise solution

```
// Returns the index of an occurrence of the given value in  
// the given array, or a negative number if not found.
```

```
// Precondition: elements of a are in sorted order
```

```
public static int binarySearch(int[] a, int target) {  
    return binarySearch(a, target, 0, a.length - 1);  
}
```

```
// Recursive helper to implement search behavior.
```

```
private static int binarySearch(int[] a, int target,  
                                int min, int max) {  
    if (min > max) {  
        return -1;           // target not found  
    } else {  
        int mid = (min + max) / 2;  
        if (a[mid] < target) {           // too small; go right  
            return binarySearch(a, target, mid + 1, max);  
        } else if (a[mid] > target) {    // too large; go left  
            return binarySearch(a, target, min, mid - 1);  
        } else {  
            return mid;    // target found; a[mid] == target  
        }  
    }  
}
```

# For Each Loop

- The for-each loop is another traversing technique

```
for (type var : arr) {  
    ...  
}
```

where type is the type of objects in the array arr.

- The for-each loop cannot be used for replacing or removing elements as you traverse.
- The loop hides the index variable that is used with arrays.



# For Each Examples

```
int[] numbers = {1,2,3,4,5};  
ArrayList<String> list = new  
ArrayList<String>();  
list.add("Mike"); list.add("John");  
  
for(int x : numbers)  
    System.out.println(x);  
  
for(String name: list)  
    System.out.println(name);
```

**Note:** The syntax is the same for both arrays and arraylists.

# For Each 2D arrays

```
int[][] numbers = {{1,2},{3,4}};  
  
for(int[] row: numbers)  
    for(int entry: row)  
        System.out.println(entry);
```

# For each example

The for-each loop cannot be used for replacing or removing elements as you traverse.

```
int[] numbers = {1, 2, 3, 4};
```

```
for(int x : numbers)  
    x = 7;
```

```
System.out.println(Arrays.toString(numbers)) ;  
// {1,2,3,4}  
// changing local variable x DOES NOT change  
// array!
```

# Sorting

- **sorting**: Rearranging the values in an array or collection into a specific order (usually into their "natural ordering").
  - one of the fundamental problems in computer science
  - can be solved in many ways:
    - there are many sorting algorithms
    - some are faster/slower than others
    - some use more/less memory than others
    - some work better with specific kinds of data
    - some can utilize multiple computers / processors, ...
  - *comparison-based sorting* : determining order by comparing pairs of elements:
    - `<`, `>`, `compareTo`, ...

# Sorting algorithms

- **bubble sort:** swap adjacent pairs that are out of order
- **selection sort:** look for the smallest element, move to front
- **insertion sort:** build an increasingly large sorted front portion
- **merge sort:** recursively divide the array in half and sort it
- **heap sort:** place the values into a sorted tree structure
- **quick sort:** recursively partition array based on a middle value

# Sorting algorithms

- **bubble sort:** swap adjacent pairs that are out of order
- **selection sort:** look for the smallest element, move to front
- **insertion sort:** build an increasingly large sorted front portion
- **merge sort:** recursively divide the array in half and sort it
- **heap sort:** place the values into a sorted tree structure
- **quick sort:** recursively partition array based on a middle value

# Bubble sort

- **bubble sort:** For each pass through the array, look at adjacent elements and swap them if they are out of order. Repeat until the entire array is sorted.

What's the result at the end of the first pass?

The largest element is at the end of the array.

# Selection sort

- **selection sort:** Orders a list of values by repeatedly putting the smallest or largest unplaced value into its final position.

The algorithm:

- Look through the list to find the smallest value.
- Swap it so that it is at index 0.
- Look through the list to find the second-smallest value.
- Swap it so that it is at index 1.
- ...
- Repeat until all values are in their proper places.



# Selection sort example

- Initial array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

- After 1st, 2nd, and 3rd passes:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	18	12	22	27	30	36	50	7	68	91	56	2	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	12	22	27	30	36	50	7	68	91	56	18	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	22	27	30	36	50	12	68	91	56	18	85	42	98	25

# Insertion Sort

- **insertion sort:** Shift each element into a sorted sub-array
  - faster than selection sort (examines fewer values)

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	12	18	22	27	30	36	50	7	68	91	56	2	85	42	98	25

sorted sub-array (indexes 0-7)



# Insertion Sort Algorithm

- Check element.
- If larger than the previous element, leave it.
- If smaller than the previous element, shift previous larger elements down until you reach a smaller element (or beginning of array).
- Insert element.

# Insertion Sort Algorithm

- 64 54 18 87 35
  - 54 less than 64
  - Shift down and insert 54
- 54 64 18 87 35(1<sup>st</sup> pass)
  - 18 less than 64
  - 18 less than 54
  - Shift down and insert 18
- 18 54 64 87 35(2<sup>nd</sup> pass)
  - 87 greater than 64
  - Go to next element
- 18 54 64 87 35(3<sup>rd</sup> pass)
  - 35 less than 87
  - 35 less than 64
  - 35 less than 54
  - 35 greater than 18
  - Shift down and insert 35
- 18 35 54 64 87(4<sup>th</sup> pass)



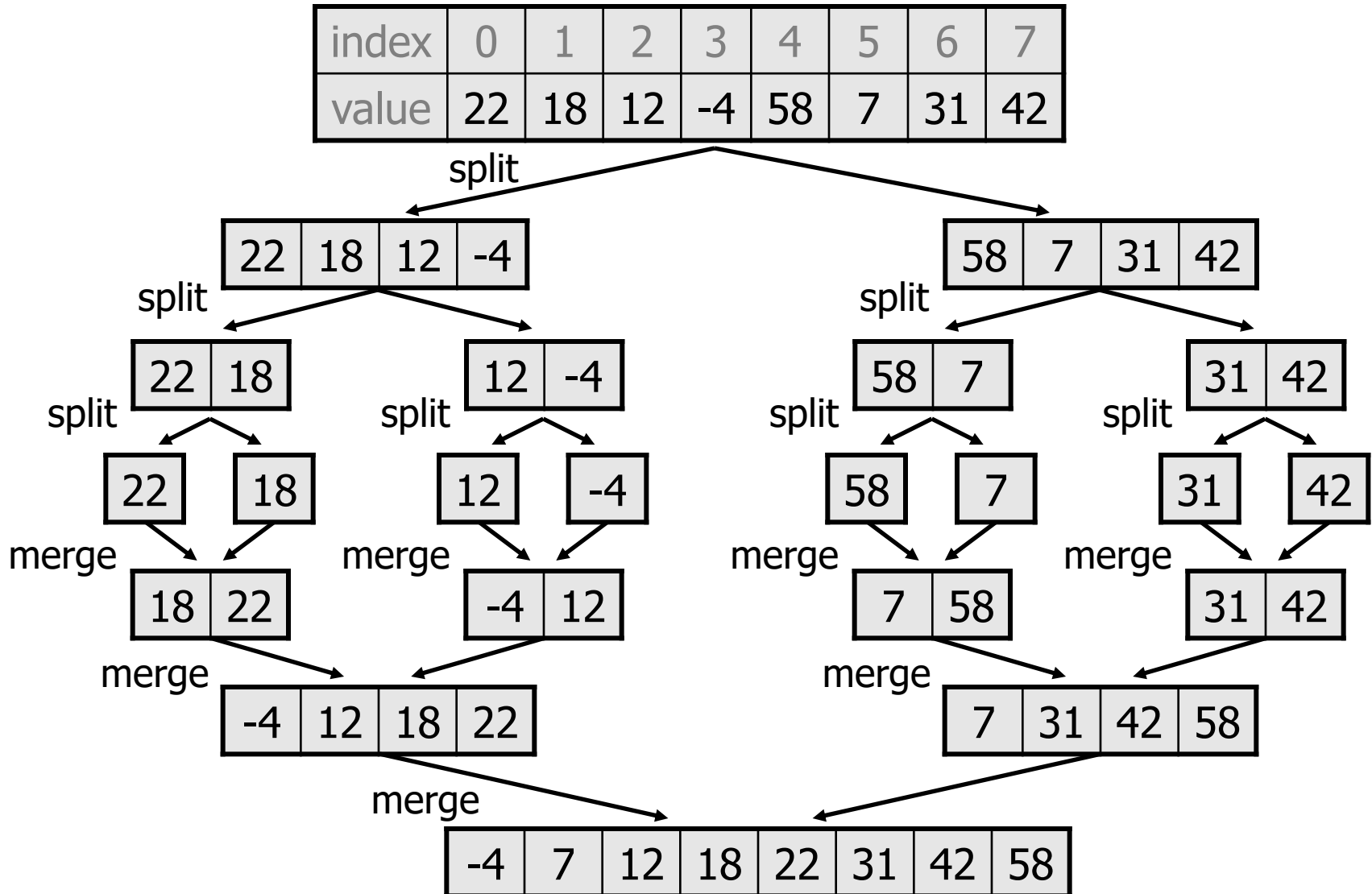
# Merge sort

- **merge sort:** Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

The algorithm:

- Divide the list into two roughly equal halves.
- Sort the left half.
- Sort the right half.
- Merge the two sorted halves into one sorted list.
  
- Often implemented recursively.
- An example of a "divide and conquer" algorithm.
  - Invented by John von Neumann in 1945

# Merge sort example



# Merging sorted halves

Subarrays				Next include				Merged array								
0	1	2	3	0	1	2	3		0	1	2	3	4	5	6	7
14	32	67	76	23	41	58	85	14 from left	14							
i1				i2					i							
14	32	67	76	23	41	58	85	23 from right	14	23						
i1				i2					i							
14	32	67	76	23	41	58	85	32 from left	14	23	32					
i1				i2					i							
14	32	67	76	23	41	58	85	41 from right	14	23	32	41				
i1				i2					i							
14	32	67	76	23	41	58	85	58 from right	14	23	32	41	58			
i1				i2					i							
14	32	67	76	23	41	58	85	67 from left	14	23	32	41	58	67		
i1				i2					i							
14	32	67	76	23	41	58	85	76 from left	14	23	32	41	58	67	76	
i1				i2					i							
14	32	67	76	23	41	58	85	85 from right	14	23	32	41	58	67	76	85
				i2					i							i

# Merge halves code

```
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
public static void merge(int[] result, int[] left,
                        int[] right) {
    int i1 = 0;    // index into left array
    int i2 = 0;    // index into right array

    for (int i = 0; i < result.length; i++) {
        if (i2 >= right.length ||
            (i1 < left.length && left[i1] <= right[i2])) {
            result[i] = left[i1];    // take from left
            i1++;
        } else {
            result[i] = right[i2];    // take from right
            i2++;
        }
    }
}
```



# Merge sort code

```
// Rearranges the elements of a into sorted order using
// the merge sort algorithm (recursive).
public static void mergeSort(int[] a) {
    if (a.length >= 2) {
        // split array into two halves
        int[] left  = Arrays.copyOfRange(a, 0, a.length/2);
        int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

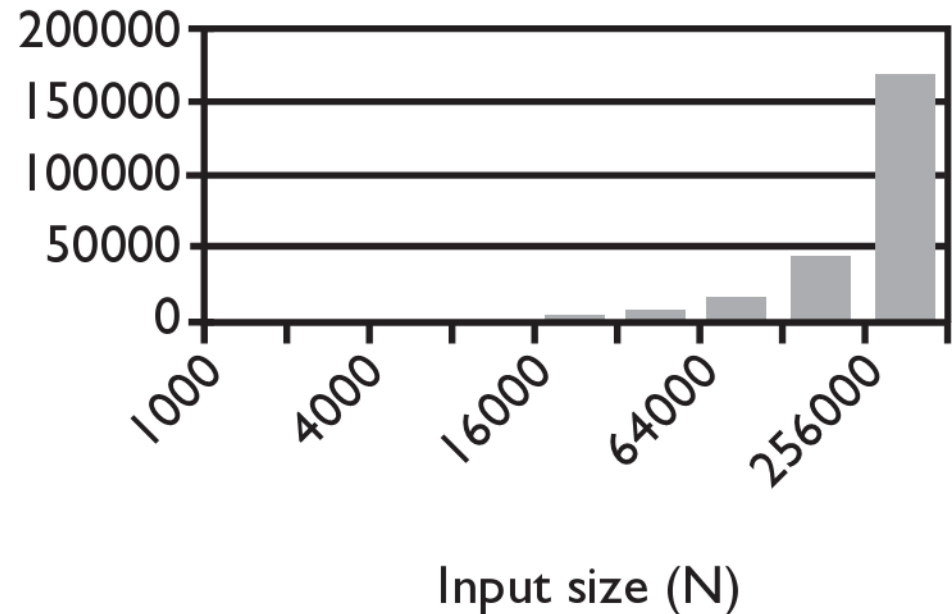
        // sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        merge(a, left, right);
    }
}
```

# Selection sort runtime (Fig. 13.6)

- What is the complexity class (Big-Oh) of selection sort?

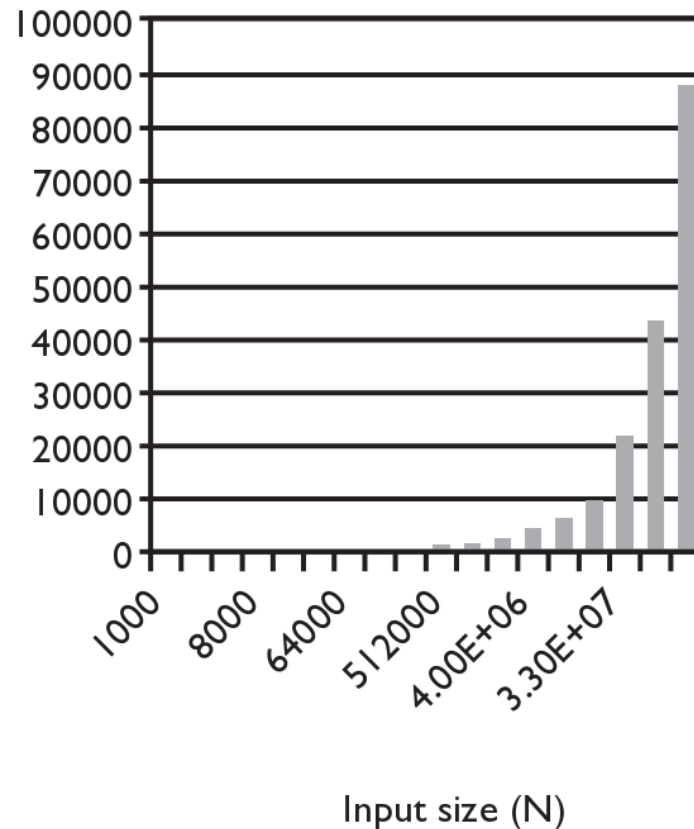
N	Runtime (ms)
1000	0
2000	16
4000	47
8000	234
16000	657
32000	2562
64000	10265
128000	41141
256000	164985



# Merge sort runtime

- What is the complexity class (Big-Oh) of merge sort?

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	15
64000	16
128000	47
256000	125
512000	250
1e6	532
2e6	1078
4e6	2265
8e6	4781
1.6e7	9828
3.3e7	20422
6.5e7	42406
1.3e8	88344



# Searching/Sorting in Java

- The `Arrays` and `Collections` classes in `java.util` have a static method `sort` that sorts the elements of an array/list
- `Arrays.sort()` is used for arrays.

```
String[] words = {"foo", "bar", "baz", "ball"};
```

```
Arrays.sort(words) ;
```

```
System.out.println(Arrays.toString(words)) ;
```

```
// [ball, bar, baz, foo]
```

```
int index = Arrays.binarySearch(words, "bar") ; //  
1
```

# Searching/Sorting in Java

- Collections.sort() is used for arraylists.

```
List<String> words2 = new ArrayList<String>();  
for (String word : words) {  
    words2.add(word);  
}
```

```
Collections.sort(words2);
```

```
System.out.println(words2);  
// [ball, bar, baz, foo]
```

# Computational Complexity

- Computational complexity of an algorithm is the amount of computational resources(e.g. comparison operations) needed to perform the algorithm.
  - This measure is a function of the input size(e.g. the size of the array.)
  - Can either be worst-case complexity or average-case complexity

# Computational Complexity for Searching

Suppose we have an array of size  $n$ .

1) What is the average number of comparisons needed to find the target using sequential search?

Answer: Approximately  $n/2$ . ( Big  $O(n)$  )

2) What is the overall all number of comparisons needed to find the target using binary search?

Answer: Approximately  $\log(n)$ . ( Big  $O(\log n)$  )

# Computational Complexity for Sorting

Suppose we have an array of size  $n$ .

1) What is the number of comparisons needed for bubble sort?

Answer: Efficiently can be done in  $n(n+1)/2$ . Approximately  $n^2$ . Big  $O(n^2)$

2) What is the number of comparisons needed for selection sort?

Answer: : Efficiently can be done in  $n(n+1)/2$ . Approximately  $n^2$ . Big  $O(n^2)$ .

2) What is the number of comparisons needed for mergesort?

Answer: Approximately  $n \log(n)$ .



# Some Examples

What's the total number of operations?

```
for(int i = 0; i < n; i++){  
    for(int j = 0; j < n; j++){  
        // do one simple operation.  
    }  
}
```

Answer:  $n^2$

```
for(int i = 0; i < n; i++){  
    for(int j = i; j < n; j++){  
        // do one simple operation.  
    }  
}
```

Answer:  $n(n+1)/2$  or Big  $O(n^2)$

# Exponential Complexity Problems

- Algorithms that can be implemented with a polynomial time complexity can be executed quickly on a modern processor. (Problems of this type belong to a class called P, Polynomial Time.)
- However, there exists important and practical problems for which there exists no known polynomial time algorithm.
  - E.g. given a set of integers, find a subset that sums to zero. A brute-force algorithm would try every possible subset. But there are  $2^n$  different subsets. This is an example of an exponential time algorithm. If  $n$  is large, even the fastest computers would take too long.

# Travelling Salesman(TSP)

- Given a set of cities and paths connecting them, find the shortest path that visit each of them exactly once.
- There is no known polynomial time algorithm that solves TSP. Solving TSP can, for example, lead to better transportation and bus routes.
- TSP belongs to a class of related problems called NP(Non-deterministic Polynomial Time). None of these problems has a polynomial time solution. And if one does, then so do the rest.

# Is $P=NP$ ?

- Is  $P=NP$ ? In other words, can Travelling salesman and the other NP-complete problems be solved in polynomial time? Mathematicians believe that  $P$  is not equal to  $NP$ . No proof is known.
- This is one of 7 Millenium Problems. The Clay Mathematics Institute has offered a million dollar prize for solving any of them.
- Grigori Perelman solved one of the Millenium Problems, the Poincare Conjecture. He declined the million dollar prize as well as the Fields Medal, the equivalent of the Nobel Prize for Mathematics.
- <https://medium.com/@phacks/how-grigori-perelman-solved-one-of-maths-greatest-mystery-89426275cb7>

# What if $P=NP$ ?

- What if  $P=NP$ ? Many very important problems in math, physics and engineering are currently intractable, i.e., solutions take exponential time. If  $P=NP$ , then there are efficient algorithms for solving them. This can lead to many advances in science.
- But  $P=NP$  can have negative consequences.
- For example, cryptography relies on certain problems being difficult. Public-key cryptography, a foundation for security applications including financial transactions over the internet, would be vulnerable if one can prove  $P=NP$  constructively.
- Most mathematicians believe that  $P$  is not equal to  $NP$ .

# Lab 1

Write the nonrecursive version of `binarySearch` method.

```
// Returns the index of an occurrence of target in the  
// array a, or a negative number(-(min+1)) if the target  
// is not found.
```

```
public static int binarySearch(int[] a, int target) {  
  
}
```

# Lab 2

Write the methods `selectionSort` and `swap`.

```
public static void selectionSort(int[] a)
{...}
```

```
public static void swap(int[] a, int i, int j)
{...}
```