# Lecture 15: Algorithms

## AP Computer Science Principles

# Algorithm

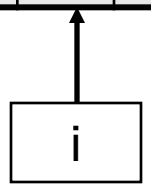
- **algorithm**: precise sequence of instructions to solve a computational problem.
  - Search for a name in a phone's contact list.
  - Sort emails by dates.
  - Given a credit card number, securely encrypt it before transmission.

# Searching Algorithms

# Sequential search

- **sequential search**: Locates a target value in an 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?

# Pseudocode

- **Pseudo-code**: sequence of instructions for an algorithm that is written in a language independent way.
  - Anyone should be able to understand pseudocode.
  - A programmer of any language should be able convert clear pseudocode into code.

Let's write the pseudo-code for sequential search.

# Pseudocode

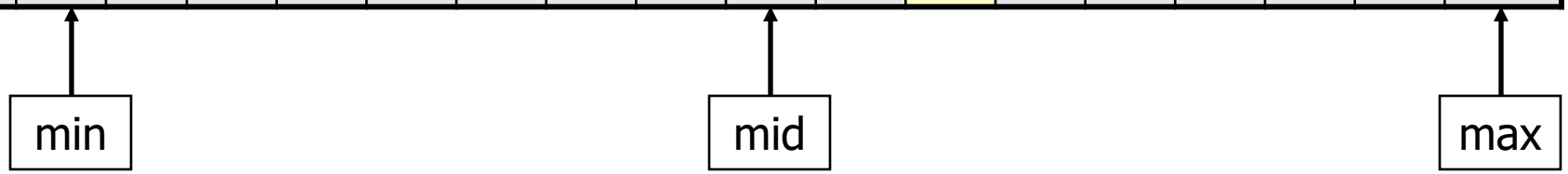
Pseudo-code for sequential search.

```
sequentialSearch(array, target):
     for i=0 to length of array-1
          if( array[i] = target)
               return i
     return -1
```

# Binary search (13.1)

- **binary search**: Locates a target value in a *sorted* 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 |



# The `Arrays` class

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

| Method name | Description |
|---|---|
| `binarySearch(`**array, value**`)` | returns the index of the given value in a *sorted* array (or < 0 if not found) |
| `sort(`**array**`)` | arranges the elements into sorted order |

Note: Must import java.util package to use this.

```
import java.util.*; //first line of program.
```

# Using `binarySearch`

```
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 < 0
```

- `binarySearch` returns the index where the value is found

# Pseudocode

Pseudo-code for binary search. **Returns -1 if not found.**

```
binarySearch(array, target):
        left = 0
        right = array's length -1
        while(left < right)
                mid = integer((left+right)/2)
                if(target > array[mid])
                        left = mid + 1
                else
                        right = mid - 1
        if(a[left] = target) return left
        else return -1
```

# Sorting Algorithms

# 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

# 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.

# Pseudocode

Very simplified pseudo-code for selection sort.

```
selectionSort(array):
      for i=0 to array's length - 1
            find the smallest element from i to end
            swap this element with array[i]
```

Notice that the bold code is another for loop. This is a nested loop!

# Selection sort example

| 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 |

| 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 |

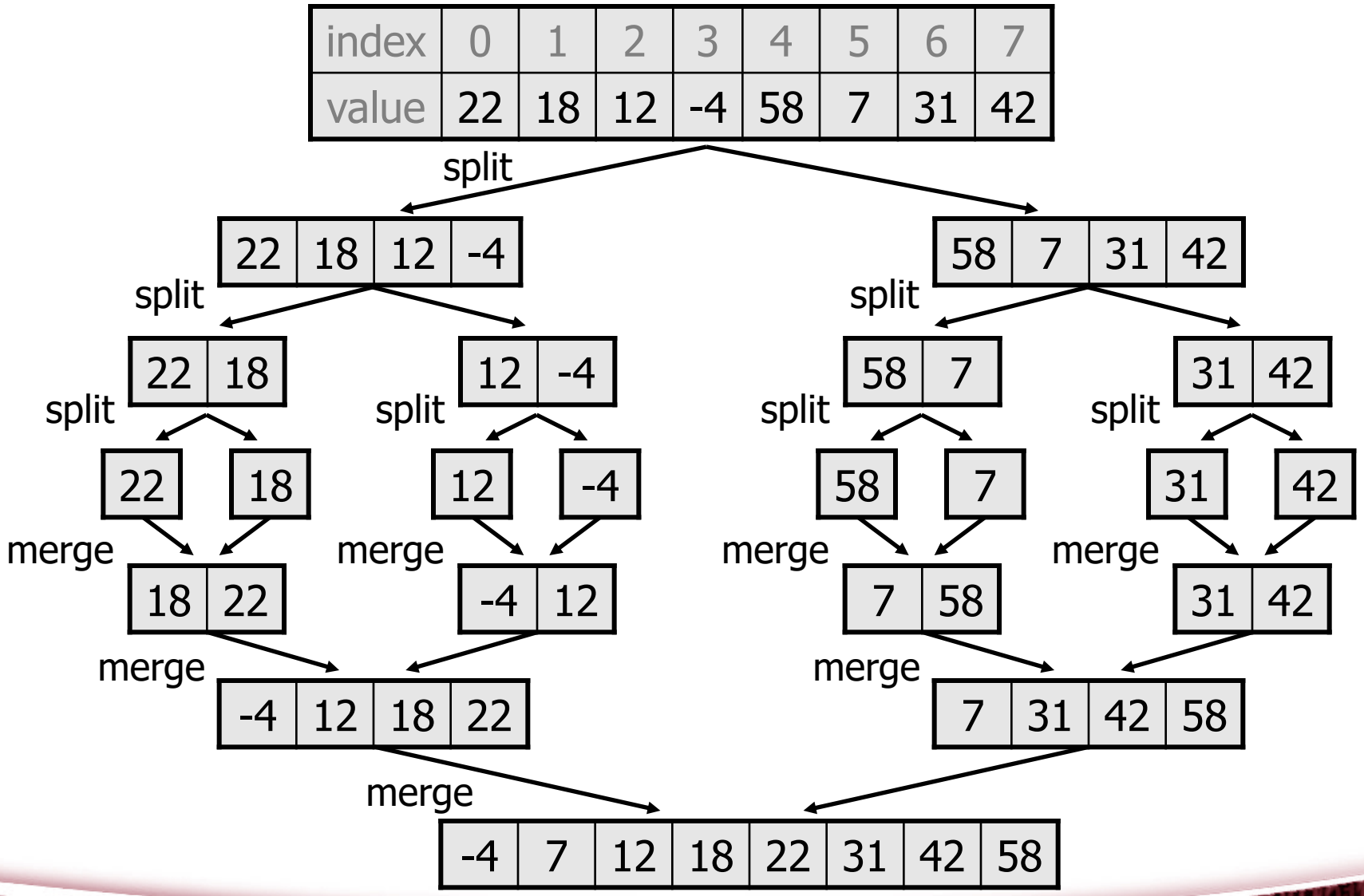
# Merge sort

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

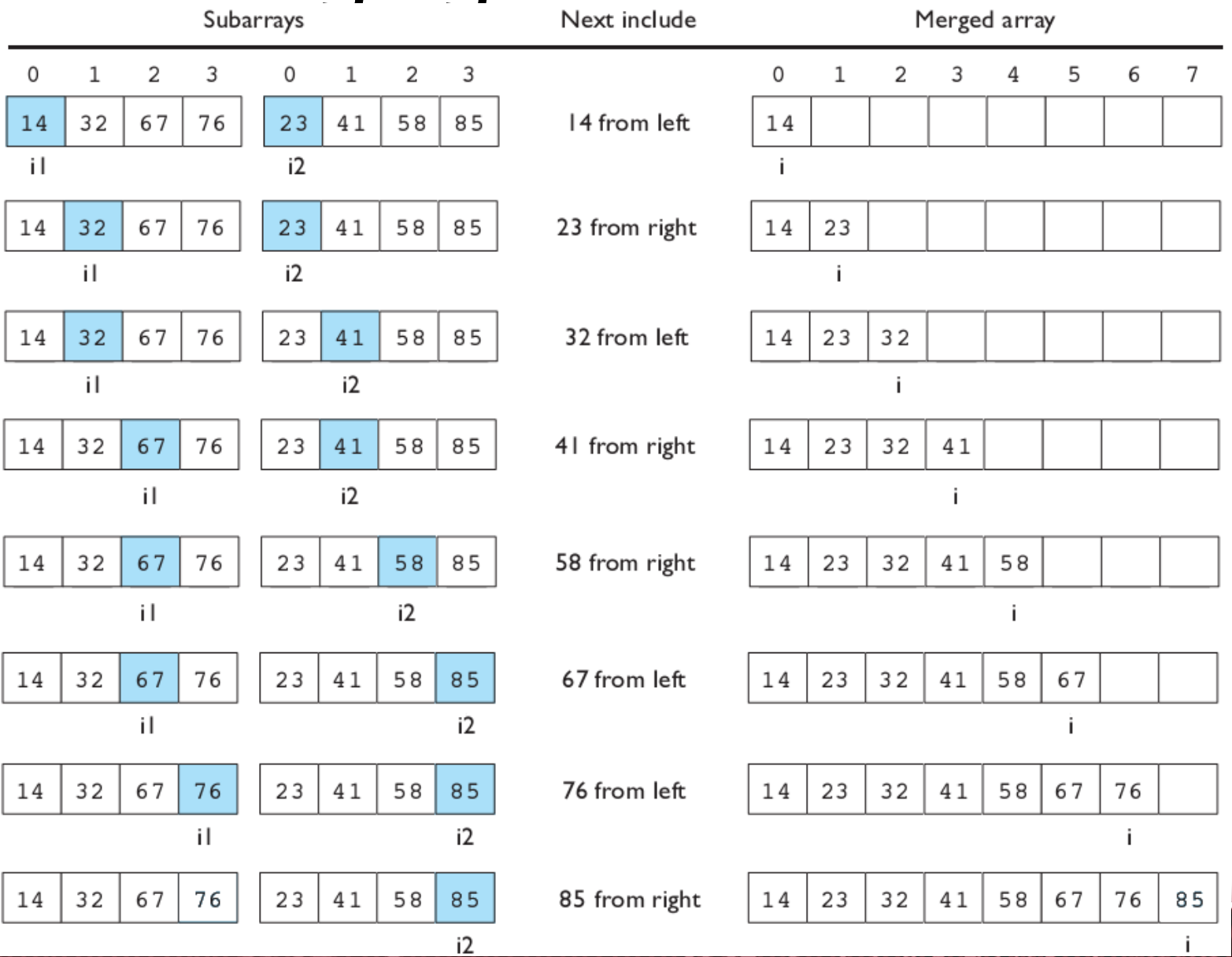
  The algorithm:
  - Divide the list into two roughly equal halves.
  - Sort the left half.
  - Sort the right half.
  - Merge the two sorted halves into one sorted list.

  - Often implemented recursively.
  - An example of a "divide and conquer" algorithm.
    - Invented by John von Neumann in 1945

# Merge sort example



# Merging sorted halves



# 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.

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

Answer: Approximately 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: Approximately n^2.

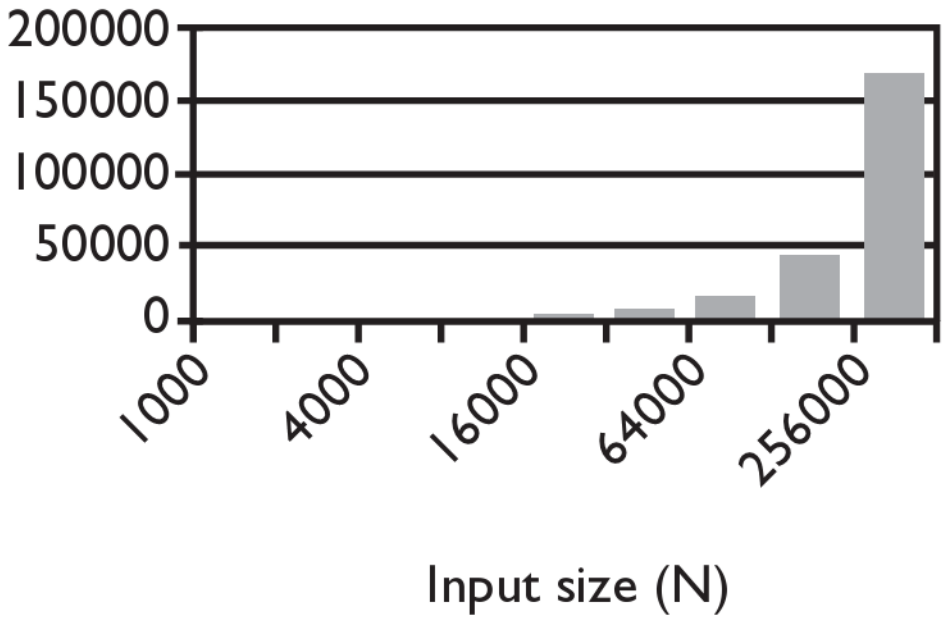
2) What is the number of comparisons needed for selection sort?
Answer: Approximately n^2.

2) What is the number of comparisons needed for mergesort?
Answer: Approximately nlog(n).

# Selection sort runtime (Fig. 13.6)

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

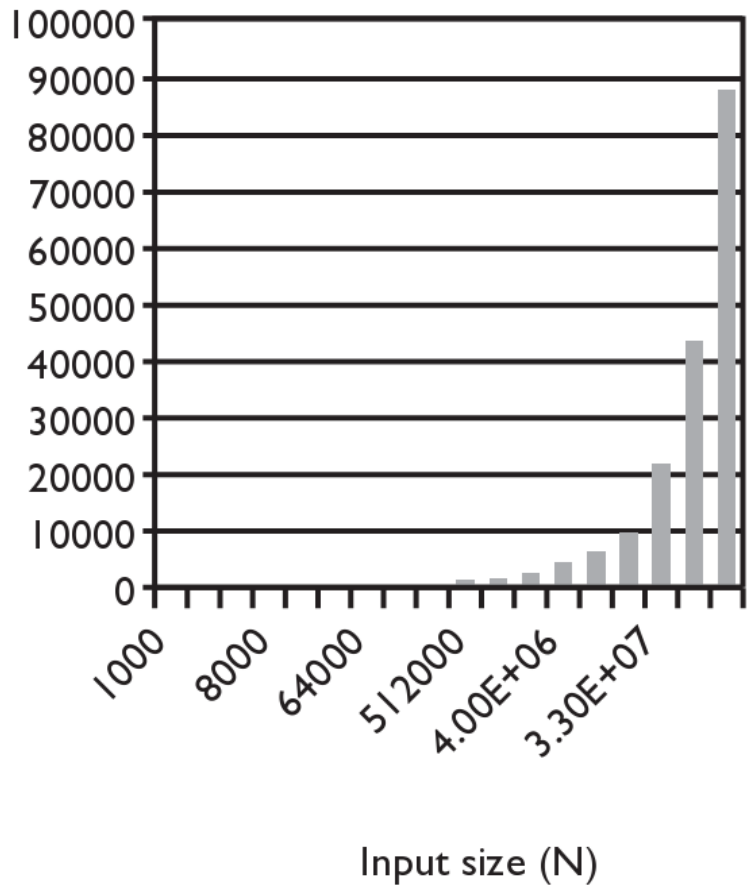
| N | Runtime (ms) |
|---|---|
| 1000 | 0 |
| 2000 | 16 |
| 4000 | 47 |
| 8000 | 234 |
| 16000 | 657 |
| 32000 | 2562 |
| 64000 | 10265 |
| 128000 | 41141 |
| 256000 | 164985 |



# 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 |



# 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 known 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. At the time, he was living with his mom and was unemployed!
- 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.

# Decidability

- A decidable problem is one in which an algorithm can be constructed to answer "yes" or "no" for all inputs.
  - E.g. Is the number even?

- An undecidable problem is one in which no algorithm can be constructed that always lead to a correct "yes" or "no" answer.

- Alan Turing, considered by many to be the father of computer science, proved that there exists undecidable problems. An example he posed is the Halting Problem.

# The Halting Problem

- Here's a problem: given a computer program and some input to that program, will the program go into an infinite loop when fed that input?
- In other words, can you write a program that takes the source code of another program and some input and returns whether the program will terminate with the given input?

```
int mystery(int n){
  if(n==1)
      return 1;
  else
      return n*mystery(n-1);
}
```

`Halt(mystery() source code, 10)` should return "yes".

# The Halting Problem

```
int mystery2(int n){
      if(n==1)
            return 1;
      else
            return n*mystery(n+1);
}
```

`Halt(mystery2() code, 10)` should return "no" .

Alan Turing proved that such a program does not exist. Please watch this video for homework.

https://www.youtube.com/watch?v=7TycxwFmdB0

# Lab

Read the pseudo-code for the sequential search, binary search and selection sort algorithms and convert it into actual code. Test your code in setup() or draw().

```
int sequentialSearch(int[] array, int target)
{}
void selectionSort(int[] array)
{}
int binarySearch(int[] array, int target)
{}
```

# Homework

1)Read and reread these lecture notes.

1)Complete the lab.

2)Read about Perelman and a friendly introduction to what he proved.

- https://medium.com/@phacks/how-grigori-perelman-solved-one-of-maths-greatest-mystery-89426275cb7

3)Watch this PBS video from the series "CrashCourse Computer Science" on Alan Turing.

- https://www.youtube.com/watch?v=7TycxwFmdB0

# References

Part of this lecture is taken from the following book.

1) Stuart Reges and Marty Stepp. Building Java Programs: A Back to Basics Approach. Pearson Education. 2008.