

# Analysis of Algorithms

---

Dr. Anirban Ghosh

**School of Computing**  
**University of North Florida**



# Introduction

## What is an algorithm?

An algorithm is a step-by-step well-defined procedure for performing some given task in a finite amount of time

## What do we mean by analyzing an algorithm?

Figuring out theoretically (using math) how much **time** and **memory** it is going to take when implemented using any programming language

## In this course

We will assume that an algorithm and a program written using it are equivalent from the analysis point of view

Why it is so important these days?



**Nobody likes slow and memory-hungry softwares!**

## Problem

Generate **really long** strings made up of English language characters

## Approach A: uses String

```
public static String generateLongStringA(int length) {  
    String password = "";  
  
    for(int i = 0; i < length; i++)  
        password += randomLetter();  
  
    return password;  
}
```

## Approach B: uses StringBuilder

```
public static String generateLongStringB(int length) {  
    StringBuilder password = new StringBuilder();  
  
    for(int i = 0; i < length; i++)  
        password.append( randomLetter() );  
  
    return password.toString();  
}
```

# Speed

```
long startA = System.currentTimeMillis();
generateLongStringA(200000);
long timeTakenA = System.currentTimeMillis() - startA;

System.out.println("Time taken by A: " + timeTakenA + " ms" );

long startB = System.currentTimeMillis();
generateLongStringB(200000);
long timeTakenB = System.currentTimeMillis() - startB;

System.out.println("Time taken by B: " + timeTakenB + " ms" );

System.out.println("Speedup: " + (double)timeTakenA/timeTakenB);
```

## Output in one run (result varies slightly every time)

```
Time taken by A: 6726 ms
Time taken by B: 33 ms
Speedup: 203.8181818181818
```



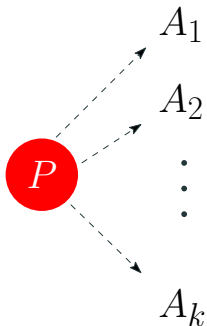
 **Tip:** for `toString` methods use `StringBuilder` instead of `String`

We will come back to the discussion of **StringBuilder** later this semester...

# Real-world challenge for coders handling Big Data

## The most challenging aspect of coding

Given a computational problem  $P$ , there may exist quite a few algorithms, say,  $A_1, A_2, \dots, A_k$ , for solving  $P$ . Now, which algorithm among these ones will be one of the fastest when implemented using a programming language?

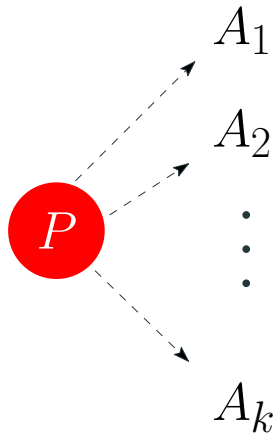


*A super-stressed coder!*



*"Gosh, which one am I going to use!"*

## Real-world challenge for coders handling Big Data



*A super-stressed coder!*



*"Gosh, which one am I going to use!"*

### An easy-peasy answer

I am going to implement all of them and find out the best ones



## The real world situation

- ☞ Nobody has this amount of time unless you are doing **Algorithm Engineering and Experiments** for one problem (may even take months)
- ☞ Even if you do, which datasets are you going to use to judge the implementations? For complex algorithms, finding appropriate datasets is a hassle
- ☞ Computational experiments are always hardware and software dependent

## Common concerns

- Why my program is taking so long?
- Why it is running out of memory?
- I am unsure if my program will run to completion within a reasonable time for every dataset! What should I do?

# A good solution to all these problems

## Theoretical approach

Use math to analyze algorithms/programs so that we can get away from time-consuming experiments

## Does it work?

Yes, it does in most cases and will work everywhere in this course

## What is it?

Analysis of algorithms (completely theoretical, no machines needed!)

# Exponents

Let  $a, b, n, m$  be real numbers. Then,

- $a^n \times a^m = a^{n+m}$

**Example:**  $2^3 \times 2^{10.5} = 2^{13.5}$

- $a^n / a^m = a^{n-m}$

**Example:**  $2^{30} / 2^{10} = 2^{20}$ ,  $6^3 / 6^{10} = 6^{-7}$

- $(a^n)^m = a^{nm}$

**Example:**  $(5^{30})^9 = 5^{270}$

- $a^0 = 1$ , when  $a \neq 0$

**Example:**  $(-27.18)^0 = 1$

# Logarithms

The **logarithm** of a positive real number  $x$  with respect to base  $b$  is the exponent by which  $b$  must be raised to yield  $x$ . The logarithm of  $x$  to base  $b$  is denoted by  $\log_b x$ .

$$\log_2 1024 = 10 \text{ since } 2^{10} = 1024$$

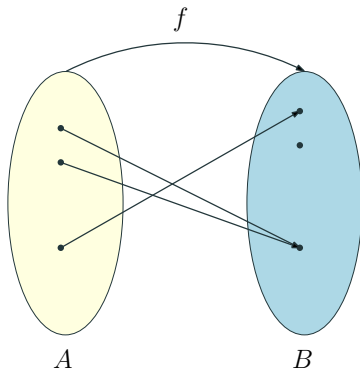
$$\log_{10} 100 = 2 \text{ since } 10^2 = 100$$

In this course, we will use **base-2** logarithms ( $b = 2$ ) mostly

# What is a function?

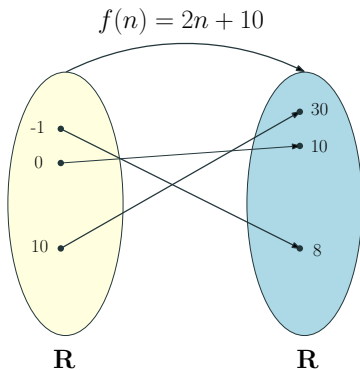
## Definiton

Given two sets  $A, B$ , a function  $f$  from  $A$  to  $B$  is a rule that associates every element of  $A$  to an element of  $B$



👉  $A, B$  can be infinite sets

## Example



The function  $f(n) = 2n + 10$  is shown pictorially

👉  $R$  is an infinite set

## Upper bounds

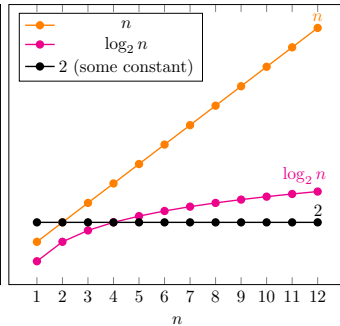
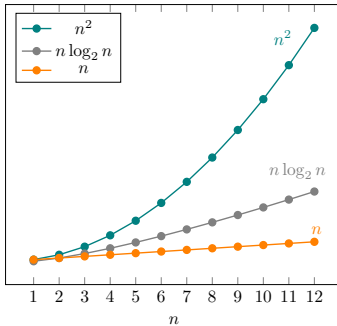
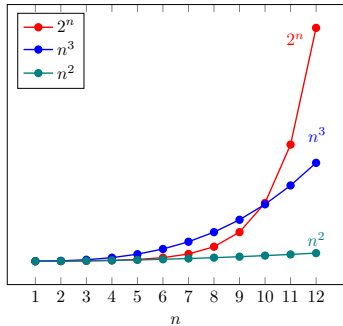
Since we are interested in worst-case time analysis of algorithms, we will use functions like  $T(n) \leq 2n + 10$ , etc. to derive upper bounds on algorithm runtimes



## The 7 types of functions we are interested in

- 1 **Constant.**  $T(n) \leq k$ , for some positive constant  $k$   
Examples:  $T(n) \leq 19.87$ ;  $T(n) \leq 10$ , etc.
- 2 **Logarithmic.**  $T(n) \leq k \log n$ , for some positive constant  $k$   
Examples:  $T(n) \leq 5 \log_2 n$ ;  $T(n) \leq 19 \log_{10} n$ , etc.
- 3 **Linear.**  $T(n) \leq kn$ , for some positive constant  $k$   
Examples:  $T(n) \leq 6n$ ;  $T(n) \leq 2^{100}n$ , etc.
- 4 **Linearithmic.**  $T(n) \leq kn \log n$ , for some positive constant  $k$   
Examples:  $T(n) \leq 100n \log_2 n$ ;  $T(n) \leq 16n \log_{10} n$ , etc.
- 5 **Quadratic.**  $T(n) \leq kn^2$ , for some positive constant  $k$   
Examples:  $T(n) \leq 100n^2$ ;  $T(n) \leq \sqrt{2}n^2$ , etc.
- 6 **Cubic.**  $T(n) \leq kn^3$ , for some positive constant  $k$   
Examples:  $T(n) \leq 8999n^3$ ;  $T(n) \leq \sqrt{99}n^3$ , etc.
- 7 **Exponential.**  $T(n) \leq k \cdot 2^n$ , for some positive constant  $k$   
Examples:  $T(n) \leq 10 \cdot 2^n$ ;  $T(n) \leq 9^{99} \cdot 2^n$ , etc.

# Growth rates comparisons of the 7 functions



## What are we trying to accomplish?

- Assume that we have an algorithm  $A$
- Denote the input size by  $n$
- Find out the function  $T(n)$ , such that  $A$  takes  $T(n)$  units of time when executed on any input of size  $n$
- $T(n)$  is a theoretical upper bound on the runtime; a.k.a. the **time complexity** of the algorithm  $A$

One can also do the same for the additional space needed by  $A$ . This is known as the **space complexity** of  $A$ .

## The two important aspects of an algorithm

- ☞ **Time complexity**. how much time as a function of the input size  $n$  does an algorithm takes to give the desired result?
- ☞ **Space complexity**. how much extra space (apart from the input and the expected output) as a function of the input size  $n$  does an algorithm need to get executed successfully?

*Note that both are important in this age of Big Data!*

# Primitive operations

## Definition

Basic computations performed by an algorithm which take constant amount of time on a fixed machine; such operations are largely independent from programming languages and hence can be used for theoretical analyses

## Some examples

- `a = b + c;`
- `i++;`
- `arr[k] = 0;`
- `arr[k] = s[i] + t[j];`
- `Counter wallet = new Counter();`
- `return arr;`
- `boolean b = (i > j);`
- `⋮`

## So what?

- The **runtime** of a program is proportional to the number of primitive operations it has; larger input means more such operations must be executed
- Clearly, depending on  $n$  (input size), the number of such operations will vary; larger  $n$  means higher runtime
- Let  $T(n)$  denote the theoretical runtime of a given algorithm (the total number of primitive operations executed by it)
- In algorithm analysis, we aim to obtain an upper-bound for  $T(n)$  by estimating the number of primitive operations executing by the algorithm under investigation
- Know that  $a \leq b$  is pronounced as ‘ $a$  is at most  $b$ ’

## Example

```
1 public static double findMax(double[] array) {  
2     int n = array.length;           // c_2 = 1  
3     double maxSoFar = array[0];     // c_3 = 1  
4  
5     for(int i = 1; i < n; i++)      // c_5 = n  
6         if( array[i] > maxSoFar )  // c_6 = n - 1  
7             maxSoFar = array[i];    // c_7 ≤ n - 1  
8  
9     return maxSoFar;                // c_9 = 1  
10 }
```

We assume that every primitive operation or a constant number of contiguous primitive operations takes 1 unit of time. Let the total cost of line  $i$  be  $c_i$ . In our case,  $c_i$  is the number of times line  $i$  is executed. Then,

$$\begin{aligned} T(n) &\leq c_2 + c_3 + c_5 + c_6 + c_7 + c_9 \\ &\leq 1 + 1 + n + (n - 1) + (n - 1) + 1 \\ &= 3 + 3n - 2 \\ &= 3n + 1 \leq 3n + n = 4n \end{aligned}$$

So, we conclude that  $T(n) \leq 4n$ .

## What does this mean?

☞ It means that the worst-case wall-clock runtime  $W(n)$  of this method is proportional to  $4n$ . In other words,

$$W(n) = m \cdot T(n) \leq m \cdot 4n,$$

for some positive constant  $m$ .

- If you are using a speedy machine,  $m$  is smaller than when you are using a slower machine
- But for a fixed machine  $m$  remains the same



## Another example

```
1 void doBubbleSort(int[] array) { // demo: https://visualgo.net/en/sorting
2   int n = array.length;          // c_2 = 1
3
4   for (int i = 0; i < n - 1; i++) { // c_4 = n
5     for (int j = 0; j < n - i - 1; j++) { // c_5 < n^2
6       if (array[j] > array[j + 1]) { // c_6 < n^2
7         int hold = array[j]; // c_7 < n^2
8         array[j] = array[j + 1]; // c_8 < n^2
9         array[j + 1] = hold; // c_9 < n^2
10      }
11    }
12  }
13 }
```

$$\begin{aligned} T(n) &\leq c_1 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 \\ &\leq 1 + n + n^2 + n^2 + n^2 + n^2 + n^2 \\ &= 5n^2 + n + 1 \\ &\leq 5n^2 + n^2 + n^2 \\ &\leq 7n^2 \end{aligned}$$

So, we conclude that  $T(n) \leq 7n^2$ .

## Growth rates and runtime

Class of $T(n)$	How does $T(n)$ look like?	Real-world impact
<b>Constant</b>	$T(n) \leq k$	Best possible speed!
<b>Logarithmic</b>	$T(n) \leq k \cdot \log n$	Crazy-fast
<b>Linear</b>	$T(n) \leq k \cdot n$	Super-fast
<b>Linearithmic</b>	$T(n) \leq k \cdot n \log n$	Very much acceptable in practice
<b>Quadratic</b>	$T(n) \leq k \cdot n^2$	Very slow in practice
<b>Cubic</b>	$T(n) \leq k \cdot n^3$	Do not expect to finish anytime soon
<b>Exponential</b>	$T(n) \leq k \cdot 2^n$	You will surely get fired!

Observations are made when  $n$  (size of the input) is large

## Deeper thoughts

- Say an algorithm  $A$  has runtime  $T(n) \leq k \cdot n^2$
- What happens when the input size is doubled?
- $\frac{k \cdot (2n)^2}{k \cdot n^2} = \frac{k \cdot 4n^2}{k \cdot n^2} = 4$
- This means runtime may quadruple on some inputs!
- But if  $T(n) \leq k \cdot n$ , then  $\frac{k \cdot 2n}{k \cdot n} = 2$  (quite expected, right?)
- What if  $T(n) \leq k \cdot 2^n$ ?
- $\frac{k \cdot 2^{2n}}{k \cdot 2^n} = 2^n$  (is not even bounded by a constant!)



## A real-world example

- Say you want to sort a million integers
- Quite common these days!
- Insertion sort takes roughly 70 hours on a 'slow' machine
- Merge sort takes roughly 40 seconds on the same machine
- Insertion sort:  $T(n) \leq k \cdot n^2$  (quadratic)
- Merge sort:  $T(n) \leq k \cdot n \log n$  (linearithmic)
- If the machine is 100x faster, then it is 40 minutes vs less than 0.5 seconds

*Moral of the story: theoretical analyses maybe hard but super-helpful in practice!*

## Facts about $T(n)$

- $T(n)$  can be simplified if we just consider the highest order term without its coefficient; we denote it by  $h$
- $h$  for a given  $T(n)$  is defined considering the growth rates of all the terms in  $T(n)$
- **Example:**  $T(n) \leq 10n^2 + 3n + 99$  (in this case,  $h = n^2$ )
- $T(n) \leq 10n^2 + 3n^2 + 99n^2 = 112n^2$
- **Another example:**  $T(n) \leq 22n \log n + 56$  (in this case,  $h = n \log n$ )
- $T(n) \leq 22n \log n + 56n \log n = 78n \log n$
- **Yet another example:**  $T(n) \leq 99 \cdot 2^n + 66n^3 + n + 100$  (in this case,  $h = 2^n$ )
- $T(n) \leq 99 \cdot 2^n + 66 \cdot 2^n + 2^n + 2^n = 167 \cdot 2^n$
- $T(n)$  can always be simplified to a form  $T(n) \leq c \cdot h$ , for some constant positive constant  $c$
- This brings us to the famous **Big-Oh** notation used by the programmers around the globe no matter what programming language they are using!

# Big-Oh

- The constant  $c$  does not matter since  $h$  is enough to determine the class of the algorithm we are analyzing; so just drop  $c$
- $T(n) \leq 112n^2$  is thus written as  $O(n^2)$
- $T(n) \leq 78n \log n$  is thus written as  $O(n \log n)$
- $T(n) \leq 167 \cdot 2^n$  is thus written as  $O(2^n)$
- **For a particular problem  $P$ , there may exist many algorithms for  $P$  which have the same runtime when expressed using Big Oh; in that case, they all are considered to be equivalent for solving  $P$  from efficiency perspective**
- An easy method for determining the Big-Oh of  $T(n)$ : *drop the lower-order terms and then drop the coefficient of the highest order term*
- Examples:  $T(n) \leq 5n^2 + 6n + 21 = O(n^2)$ ;  
 $T(n) \leq 99 \cdot 2^n + 66n^3 + n + 100 = O(2^n)$
- In other words, given  $T(n)$ , we can simply write  $T(n) = O(h)$  where the highest order term in  $T(n)$  without its coefficient

## Runtimes using Big-Oh

Class of $T(n)$	How it looks like?	Expressing $T(n)$ using Big-Oh
<b>Constant</b>	$T(n) \leq k$	$O(1)$
<b>Logarithmic</b>	$T(n) \leq k \cdot \log n$	$O(\log n)$
<b>Linear</b>	$T(n) \leq k \cdot n$	$O(n)$
<b>Linearithmic</b>	$T(n) \leq k \cdot n \log n$	$O(n \log n)$
<b>Quadratic</b>	$T(n) \leq k \cdot n^2$	$O(n^2)$
<b>Cubic</b>	$T(n) \leq k \cdot n^3$	$O(n^3)$
<b>Exponential</b>	$T(n) \leq k \cdot 2^n$	$O(2^n)$

## Order-of-growth classifications

- **CONSTANT.** The algorithm under consideration has a constant (can be terribly big) number of operations; runtime is not dependent on input size
  - Printing out the first integer in  $A$
  - Finding the maximum and minimum elements in a sorted array  $A$
  - Finding the Euclidean distance between two given points  $(x_1, y_1)$  and  $(x_2, y_2)$

We say that the algorithm runs in constant time or  $O(1)$  time

- **LOGARITHMIC.** The algorithm under consideration has a runtime of  $O(\log n)$  and is barely slower than a constant-time algorithm
  - Binary search on a sorted array of size  $n$

We say that the algorithm has a logarithmic runtime



## Order-of-growth classifications

- **LINEAR.** The algorithm under consideration spends a constant amount of time processing each piece of input data, or is based on a single loop and has a runtime of  $O(n)$ 
  - Finding the largest integer in an array  $A$  of length  $n$
  - Given a set  $A$  of  $n$  points, find the farthest point in  $A$  from a given point not in  $A$

We say that the algorithm has a linear runtime

- **LINEARITHMIC.** The running time of the algorithm under consideration has a runtime of  $O(n \log n)$ 
  - Merge sort an array of  $n$  comparable items

We say that the algorithm has a linearithmic runtime

## Order-of-growth classifications

- **QUADRATIC.** The running time of the algorithm under consideration has a runtime of  $O(n^2)$ ; usually has two nested loops each of which iterates for  $n$  times approximately
  - Bubble sort an array of  $n$  comparable items
  - Insertion sort an array of  $n$  comparable items
  - Selection sort an array of  $n$  comparable items

We say that the algorithm has a quadratic runtime

- **CUBIC.** The running time of the algorithm under consideration has a runtime of  $O(n^3)$ ; usually has three nested loops each of which iterates for  $n$  times approximately
  - Multiply two  $n \times n$  matrices in a naive way

We say that the algorithm has a cubic runtime

## Order-of-growth classifications

- **EXPONENTIAL.** The running time of the algorithm under consideration has a runtime of  $O(2^n)$  or even worse
  - Print out all possible subsets of an  $n$ -element set

We say that the algorithm has an exponential runtime

## What to find the runtime without doing math?

- 1 Find out the most costly statement/step; the one that is executed most number of times among all the statements/steps. Let the cost of that statement/step be  $t(n)$  and  $h$  be its highest order term without its constant coefficient
  - ☞ *Approximating  $t(n)$  is okay but over-approximation is certainly a bad idea!*
- 2 Runtime of the algorithm is  $O(h)$

# Figuring out runtime just by eyeballing

```
1 public static double findMax(double[] array) {  
2     int n = array.length;  
3     double maxSoFar = array[0];  
4  
5     for(int i = 1; i < n; i++)  
6         if( array[i] > maxSoFar )  
7             maxSoFar = array[i];  
8  
9     return maxSoFar;  
10 }
```

Runs in  $O(n)$  time

# Figuring out runtime just by eyeballing

```
1 void doBubbleSort(int[] array) {  
2     int n = array.length;  
3  
4     for (int i = 0; i < n-1; i++) {  
5         for (int j = 0; j < n-i-1; j++) {  
6             if (array[j] > array[j+1]) {  
7                 int hold = array[j];  
8                 array[j] = array[j+1];  
9                 array[j+1] = hold;  
10            }  
11        }  
12    }  
13 }
```

Runs in  $O(n^2)$  time

 Visualize sorting algorithms

<https://visualgo.net/en/sorting>

# Runtime?

```
1 void somemethod(int[] items) {  
2     int result = 0;  
3     for (int i = 0; i < items.length; i++)  
4         for (int j = 0; j < 100; j++)  
5             result += items[i];  
6  
7     System.out.println(result);  
8 }
```

## Now from problem to code

### Problem: used in finance to keep tracking of rising and falling profit averages

You are given an array  $X$  of  $n$  numbers. For every  $0 \leq i \leq n - 1$ , you need to find out:  $A[i] = \frac{X[0] + X[1] + \dots + X[i]}{i + 1}$ . These are called **prefix averages** of an array.

### Example

**Input:**  $X = \{10, 0, 1, 5, 99, 3\}$  (here  $n = 6$ )

$$A[0] = (10)/1 = 10$$

$$A[1] = (10 + 0)/2 = 5$$

$$A[2] = (10 + 0 + 1)/3 \approx 3.67$$

$$A[3] = (10 + 0 + 1 + 5)/4 = 4$$

$$A[4] = (10 + 0 + 1 + 5 + 99)/5 = 23$$

$$A[5] = (10 + 0 + 1 + 5 + 99 + 3)/6 \approx 19.67$$

**Output:**  $A = \{10, 5, 3.67, 4, 23, 19.67\}$



## Approach A

```
1 public static double[] findPrefixAveragesA(double[] x) {  
2     int n = x.length;  
3     double[] a = new double[n];  
4  
5     for(int i = 0; i < n; i++) {  
6         double total = 0;  
7  
8         for(int j = 0; j <= i; j++)  
9             total += x[j];  
10  
11         a[i] = total/(i+1);  
12     }  
13     return a;  
14 }  
15
```

Lines 8, 9 are the most expensive ones; they are executed approximately  $n^2$  times each; so overall runtime is  $O(n^2)$

## Approach B

```
1 public static double[] findPrefixAveragesB(double[] x) {  
2     int n = x.length;  
3     double[] a = new double[n];  
4  
5     double total = 0;  
6  
7     for(int i = 0; i < n; i++) {  
8         total += x[i];  
9         a[i] = total/(i+1);  
10    }  
11    return a;  
12 }
```

Lines 7, 8, 9 are the most expensive ones; they are executed approximately  $n$  times each; so overall runtime is  $O(n)$

# Experiment

```
Random generator = new Random();
int n = 100000;
double[] testArray = new double[n];
for( int i = 0; i < n; i++)
    testArray[i] = generator.nextInt(n);

long startA = System.currentTimeMillis();
findPrefixAveragesA(testArray);
long timeTakenA = System.currentTimeMillis() - startA;

System.out.println("Time taken by approach A (runs in O(n^2) time): " + timeTakenA + " ms" );

long startB = System.currentTimeMillis();
findPrefixAveragesB(testArray);
long timeTakenB = System.currentTimeMillis() - startB;

System.out.println("Time taken by B (runs in O(n) time): " + timeTakenB + " ms" );

System.out.println("Speedup: " + (double)timeTakenA/timeTakenB);
```

## Output in one run (result varies slightly every time)

```
Time taken by approach A (runs in O(n^2) time): 4454 ms
Time taken by approach B (runs in O(n) time): 3 ms
Speedup: 1484.6666666666667
```

## What happens in real life?

- ① You get a computational problem  $P$
  - ② You find multiple algorithms for solving  $P$
  - ③ Having the abstract ideas and/or pseudo-codes of these algorithms and using pencil + paper, you find out their theoretical runtimes
  - ④ Implement the algorithm that appears best (has the best theoretical runtime)
- 👉 In some cases, you may need to find out their **space complexities** too

# Sorting faster

## Problem

Sort an array of length  $n$ , where the array elements are taken from the set  $\{0, 1, 2, 3, 4\}$ .

## Question

What's the fastest algorithm you can design for this problem?



# Solution

## Straightforward solution: runs in $O(n \log n)$ time

```
Arrays.sort(arr1);  
System.out.println("Time taken by Arrays.sort(): " + (System.currentTimeMillis() - start) + " ms");
```

## A faster solution: runs in $O(n)$ time

```
// count the number of 0s, 1s, 2s, 3s, and 4s  
int[] count = new int[5];  
for(int i = 0; i < n; i++)  
    count[arr2[i]]++;  
  
// put 0s first, then 1s, and then 2s, 3s, and 4s  
int pos = 0;  
for(int i = 0; i < 5; i++)  
    for(int j = 0; j < count[i]; j++)  
        arr2[pos++] = i;
```

```
System.out.println("Time taken by our approach: " + (System.currentTimeMillis() - start) + " ms");
```

## Speed comparison, $n = 1,000,000$

Time taken by Arrays.sort(): 27 ms

Time taken by our approach: 11 ms

## **Chapter 4** from

<https://opensa-server.cs.vt.edu/OpenDSA/Books/CS3/html/index.html>