

# PG4200: Algorithms And Data Structures

## Lesson 03: Runtime Analysis and Sorting

Prof. Andrea Arcuri

# How Long?



- You want **fast** algorithms
- You could just run some “experiments”, and check how long your algorithm takes
- But what if algorithm will need to be run on a larger problem than I used in the experiments?
- If the problem is **twice as big**, will my algorithm take just **twice as long**???

```
public static int sum(int[] array) {  
  
    int sum = 0;  
    for(int i=0; i<array.length; i++) {  
        sum += array[i];  
    }  
    return sum;  
}
```

- “Cost” can be measured in number of executed statements
- Given size of array N, the loop will be taken N times
- There is some constant cost independent of N, eg creation of “*int sum*” variable
- If N doubles, would expect function will be *roughly* twice as slow

$$\text{instructions}(N) = 3N + 4$$

$$\text{instructions}(N=0)=4$$

```
1. int sum = 0;  
2. int i=0;  
3. i<array.length;  
4. return sum;
```

- Number of instructions depends on size N of the array, plus some constant cost
- Can be represented with a function, eg  $f(N)=3N+4$
- For large N, constants are not important

$$\text{instructions}(N=3)=13$$

```
1. int sum = 0;  
2. int i=0;  
3. i<array.length;  
4. sum += array[i];  
5. i++;  
6. i<array.length;  
7. sum += array[i];  
8. i++;  
9. i<array.length;  
10. sum += array[i];  
11. i++;  
12. i<array.length;  
13. return sum;
```

```

public static int pairs(int[] array) {
    int pairs = 0;

    for(int i=0; i<array.length; i++){
        for(int j=0; j<array.length; j++){
            if(i!=j && array[i] == array[j]){
                pairs++;
            }
        }
    }
    return pairs;
}

```

/\*

*On my machine, repeated 100 times:*

N=100	seconds=0.005
N=200	seconds=0.005
N=400	seconds=0.012
N=800	seconds=0.072
N=1600	seconds=0.211
N=3200	seconds=0.754
N=6400	seconds=2.829
N=12800	seconds=11.48

\*/

- Two nested loops
- Inner loop executed once per each element in array
- So,  $N * N = N^2$
- Twice as big is now  $2 * 2 = 4$  times as slow!!! (roughly)

# Scalability

- When analyzing algorithms, we will not look at the low level optimization details
- **N** as representation of the problem size (eg, length of array or number of elements in a container)
- How does the algorithm *scale* for larger sizes???
- Example: if my website works fine with a load of 100 users, what will happen with 2,000??? Will I just need 20 times the resources?

# Wheat/Rice and Chessboard Problem



- 1 rice grain on first square
- Double at each square
- How many grains on the board?
- 18,446,744,073,709,551,615
- ie, 18 Quintillions

# Analysis of Algorithms

- Mathematically define the cost as a function of the input size
- *Precise* functions can be impractical, so we need approximations
- Usually, we are interested in *upper* and *lower* bounds

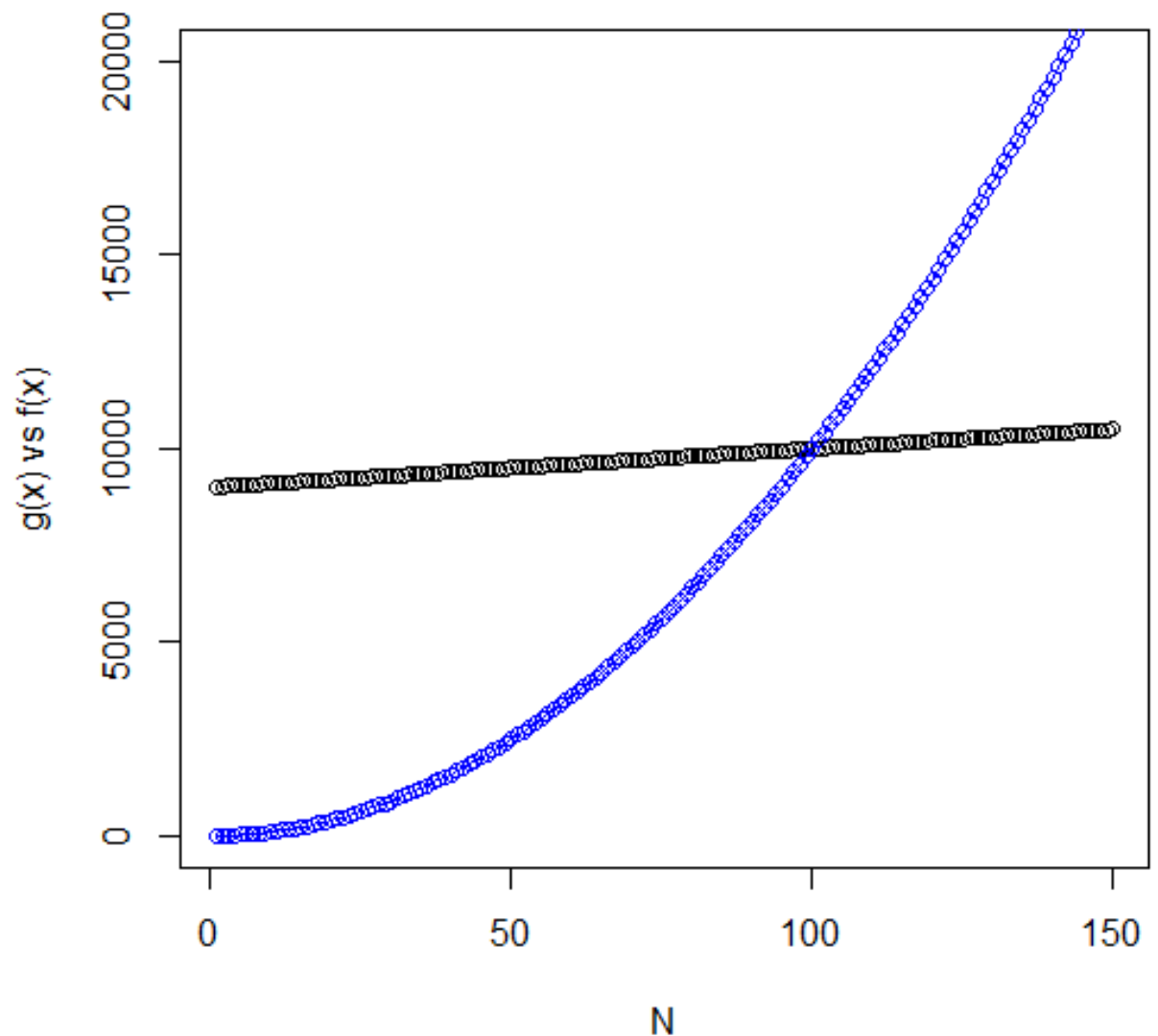


# Example

- $f(n) = a N^2 + b N + c$
- Given an algorithm whose performance is described by the polynomial  $f(n)$ , finding the actual values for  $a, b, c$  might be too difficult
- However, can we say something about the **scalability**?
- YES!!! Regardless of  $a = 5$  or  $a = 400$ , still doubling  $N$  would result in increase of at least 4 times (roughly...)

# Which Is Better?

- $f(n) = n^2$
- $g(n) = 10n + 9000$
- For small values  $f(n)$  is better, but it becomes worse from  $n > 100$
- We will look at *large*  $n$ , so for us  $g(n)$  is better



# Large N???

- How do we define *large*?
  - 10? 50? 10000000000000000000000???
- We can't really say... however, things grow so fast... what we think is *large* today, is likely going to be considered *tiny* in few years...
- Today I know how fast my algorithms are, because I run them. But I want to know how they will *scale* to the larger problem instances of tomorrow.
  - Eg, when my apps get more users

FPS... large increase in number of polygons to render...



Doom (1993)

Doom (2016)



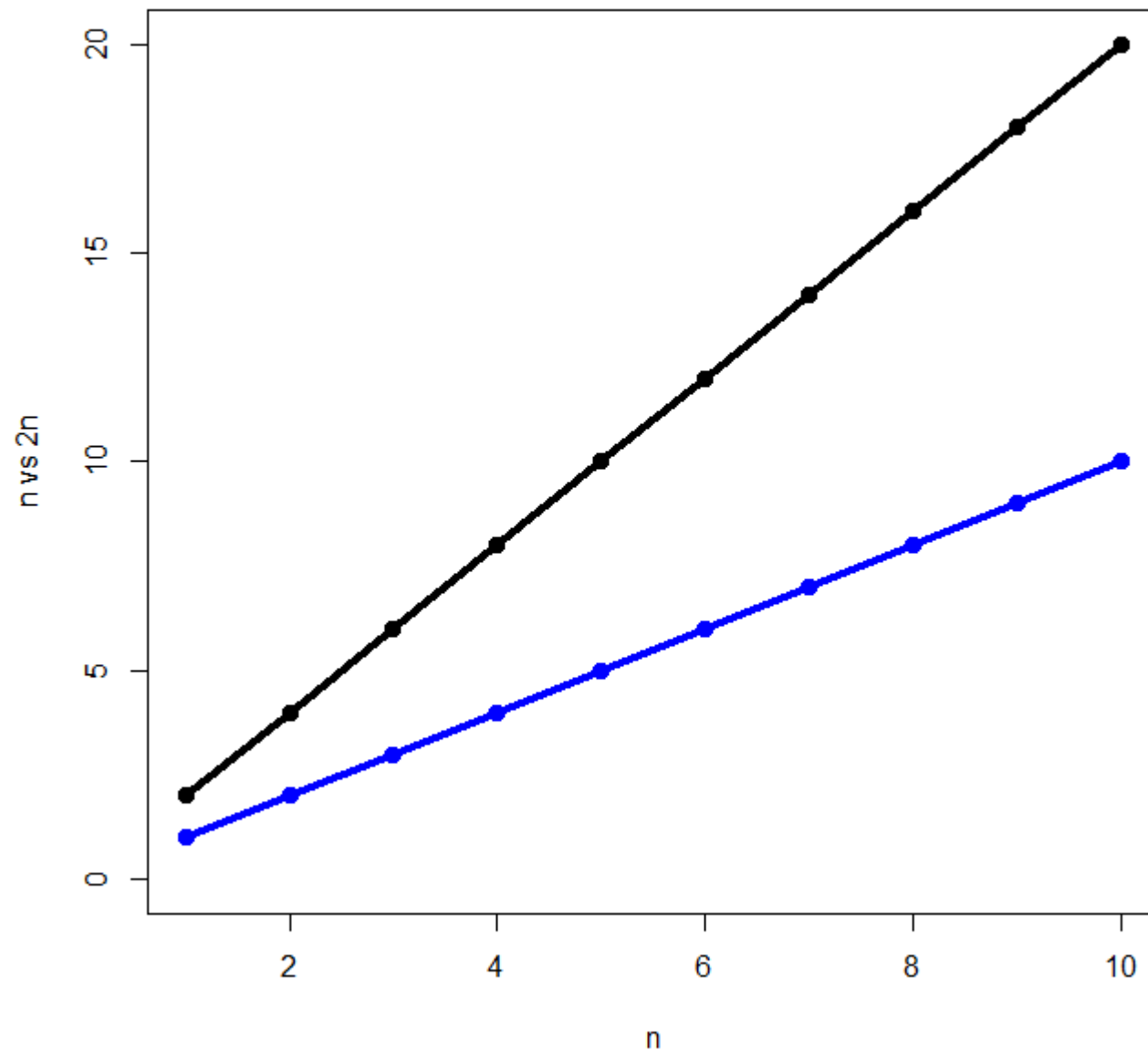
# Scalability

- $f(n) = 5n + 100$ 
  - If I am interested in scalability, the constants 5 and 100 are *irrelevant*
- $g(n) = 2n^2 + 10n + 7$ 
  - The constants 2, 10 and 7 are irrelevant. But what about the  $n$  compared with  $n^2$ ??? It is smaller, but maybe still important?

# Big O Upper Bound

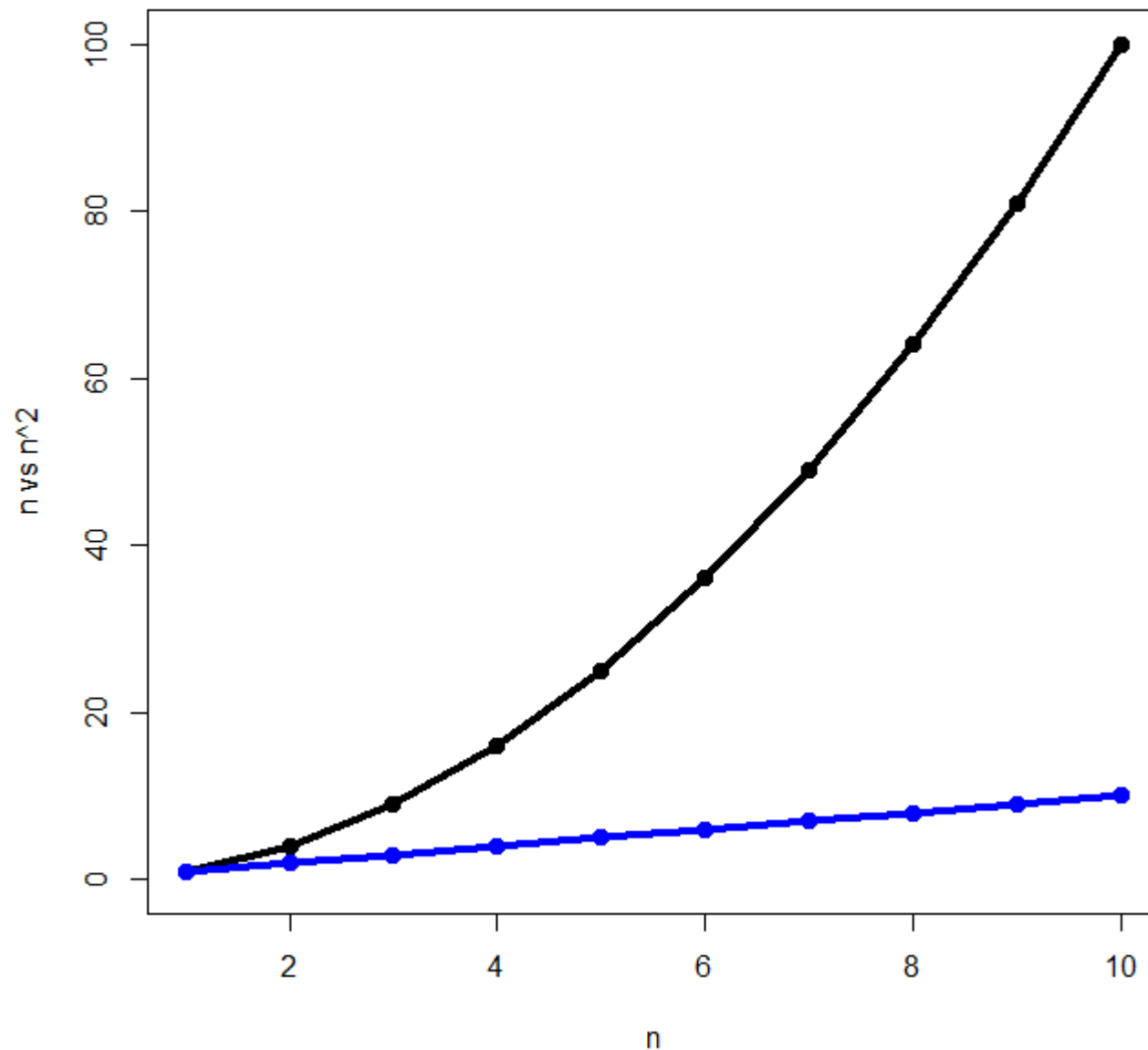
- $f(n) = O(g(n))$
- If there exists positive constants  $c$  and  $n'$ , such that  $0 \leq f(n) \leq c * g(n)$  for all  $n \geq n'$
- In other words,  $c * g(n)$  is an **upper bound** for  $f(n)$  for large values of  $n$
- Useful to consider *worst case scenarios*

$$n = O(n)$$



- $g(n) = n$
- Examples:  $c = 2, n' = 1$
- $n < 2n$  for  $n \geq 1$

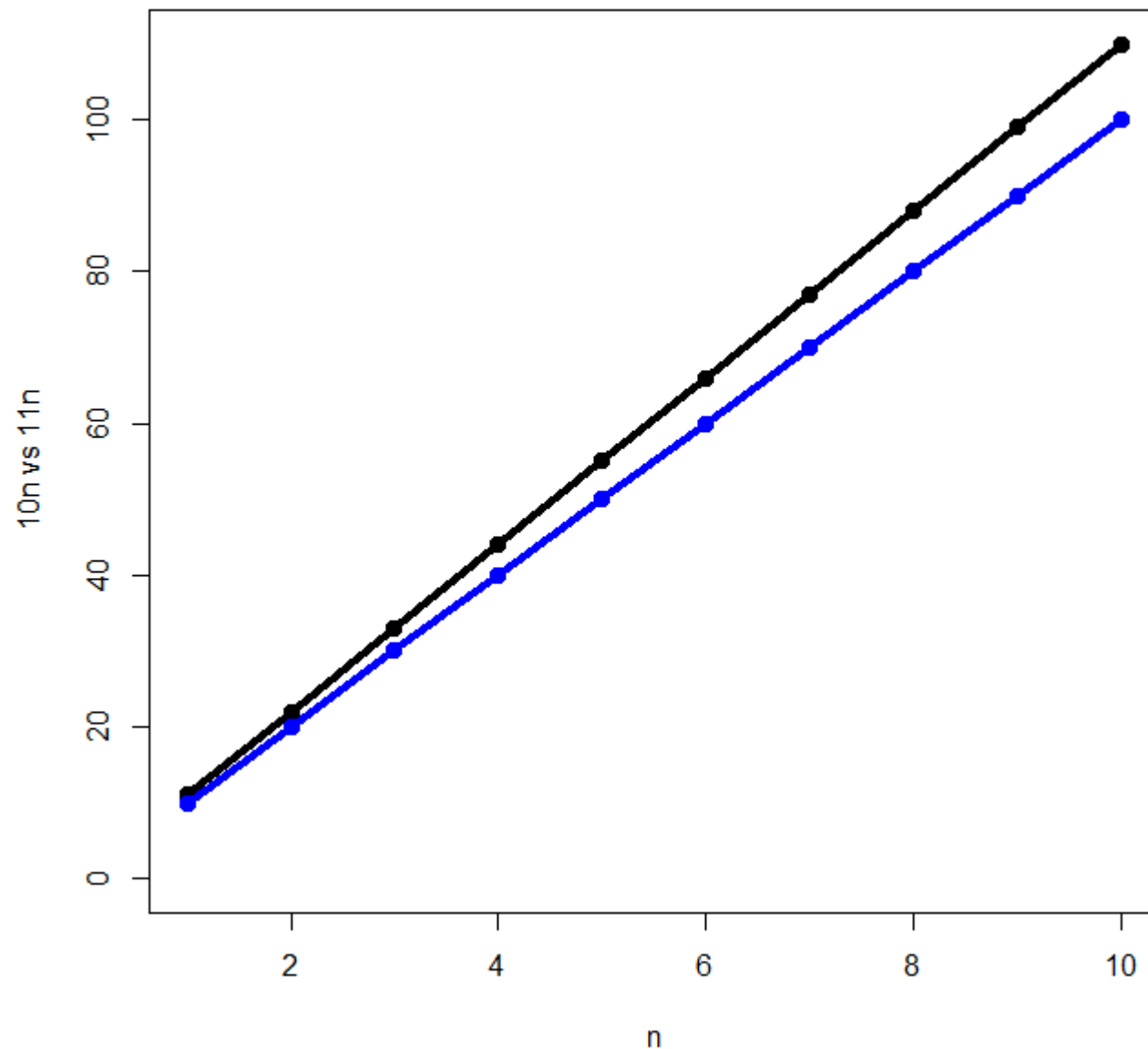
$$n = O(n^2)$$



- $g(n) = n^2$
- Examples:  $c = 1, n' = 2$
- $n < n^2$  for  $n \geq 2$

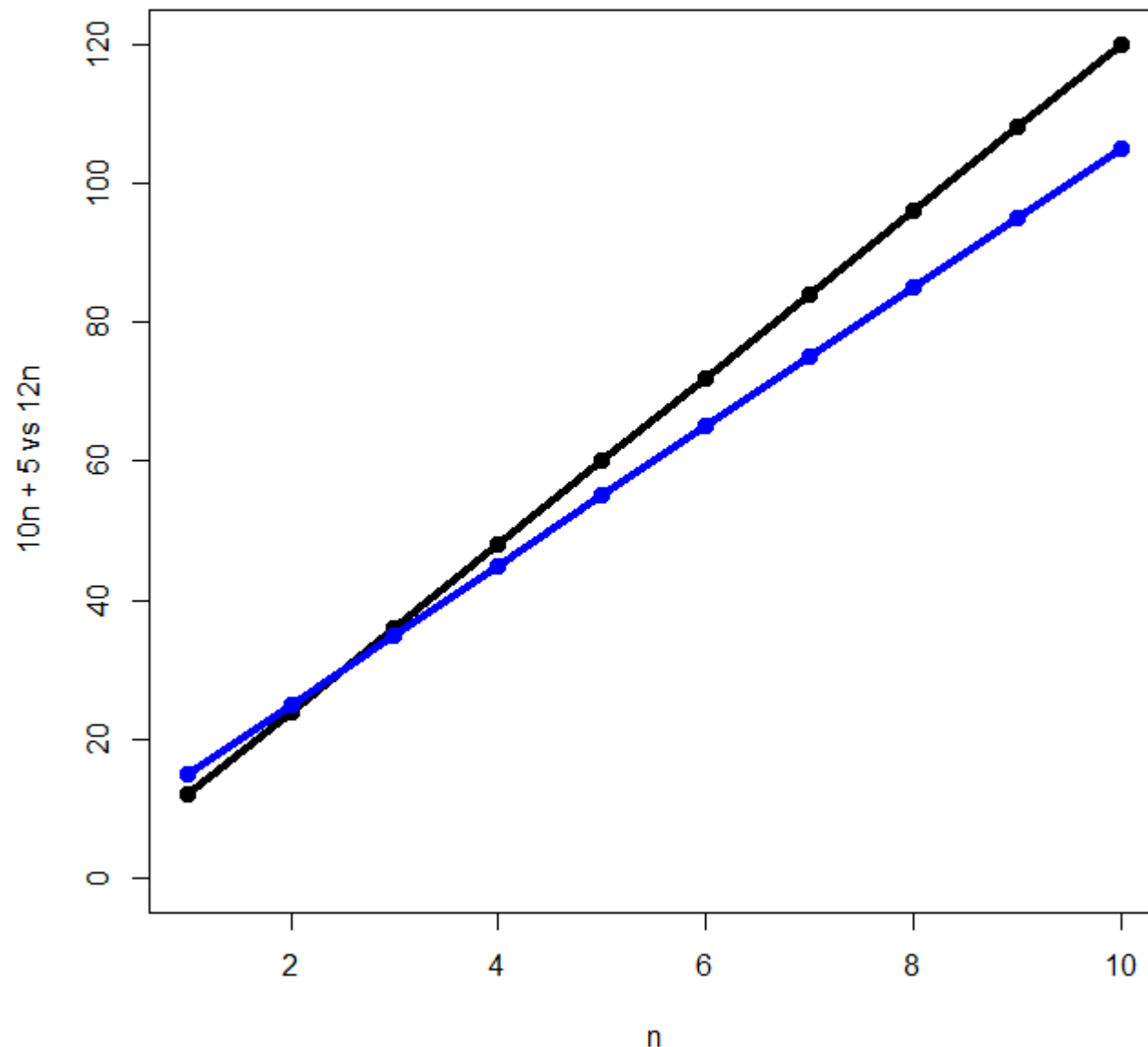


$$10n = O(n)$$



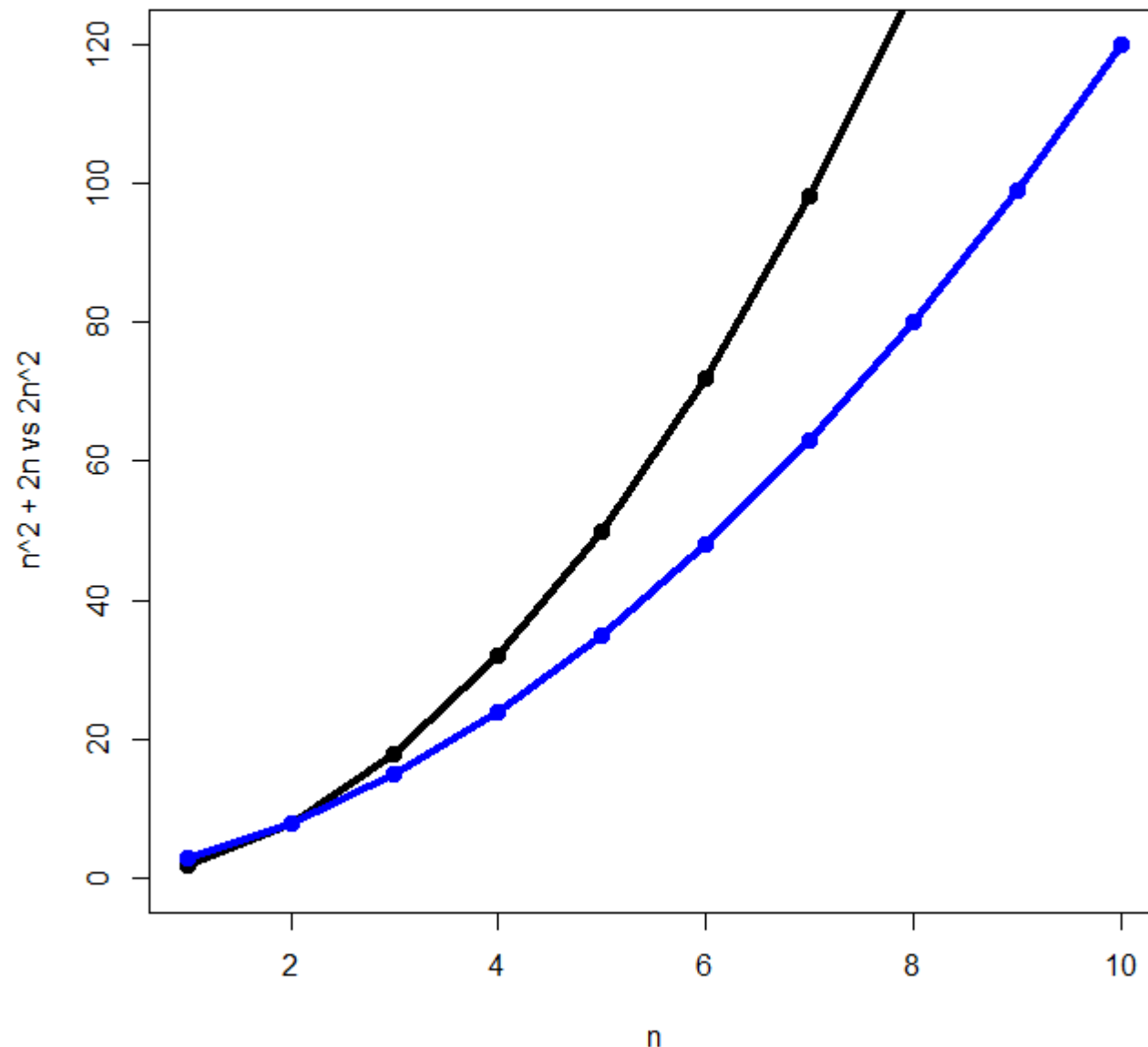
- $g(n) = n$
- Examples:  $c = 11, n' = 1$
- $10n < 11n$  for  $n \geq 1$

$$10n + 5 = O(n)$$



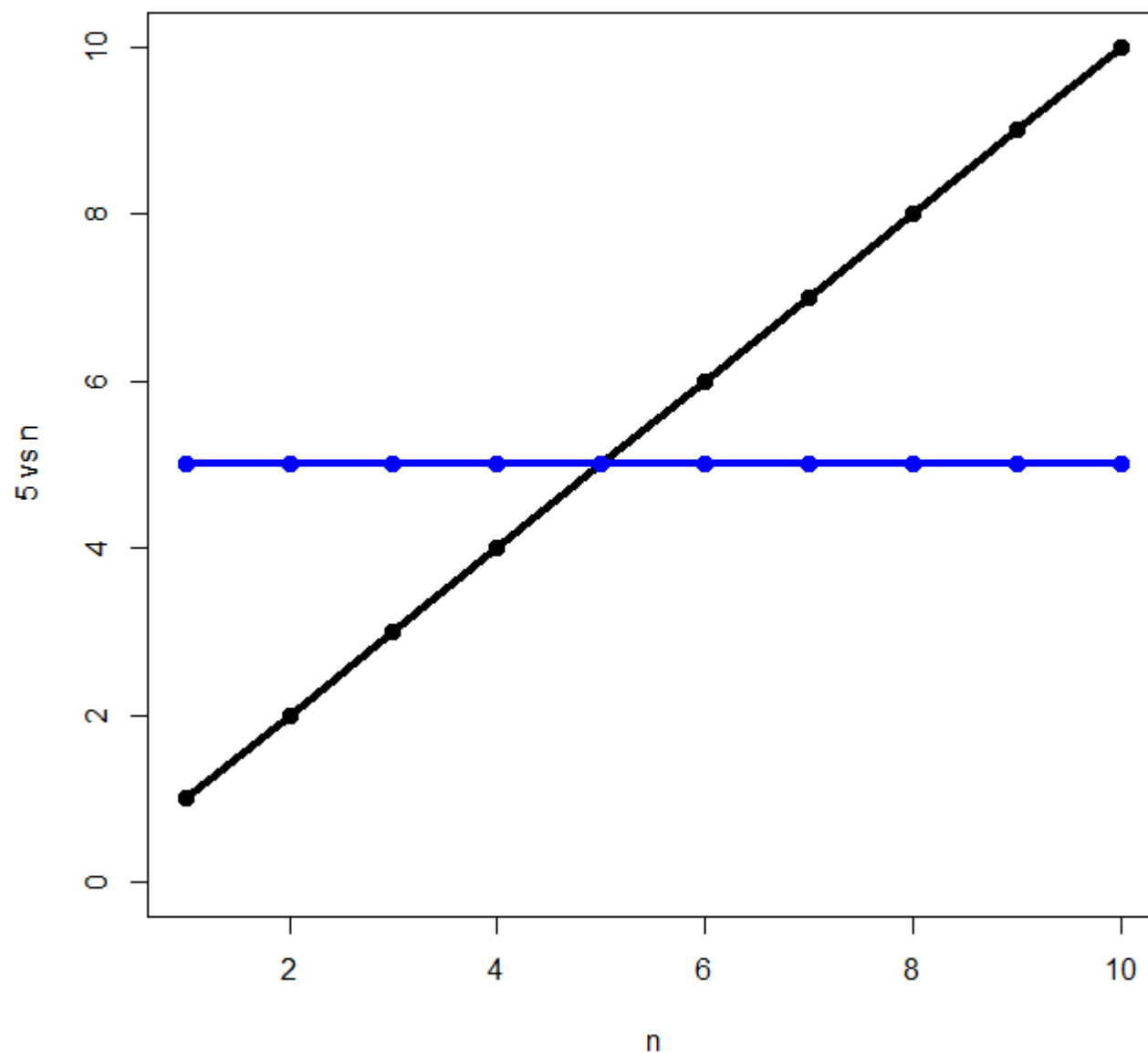
- $g(n) = n$
- Examples:  $c = 12, n' = 3$
- $10n + 5 < 12n$  for  $n \geq 3$
- Eg:  $n=3 \rightarrow f(n)=35, g(n)=36$
- Note: for  $n \leq 2$ ,  $f(n)$  is actually larger

$$n^2 + 2n = O(n^2)$$



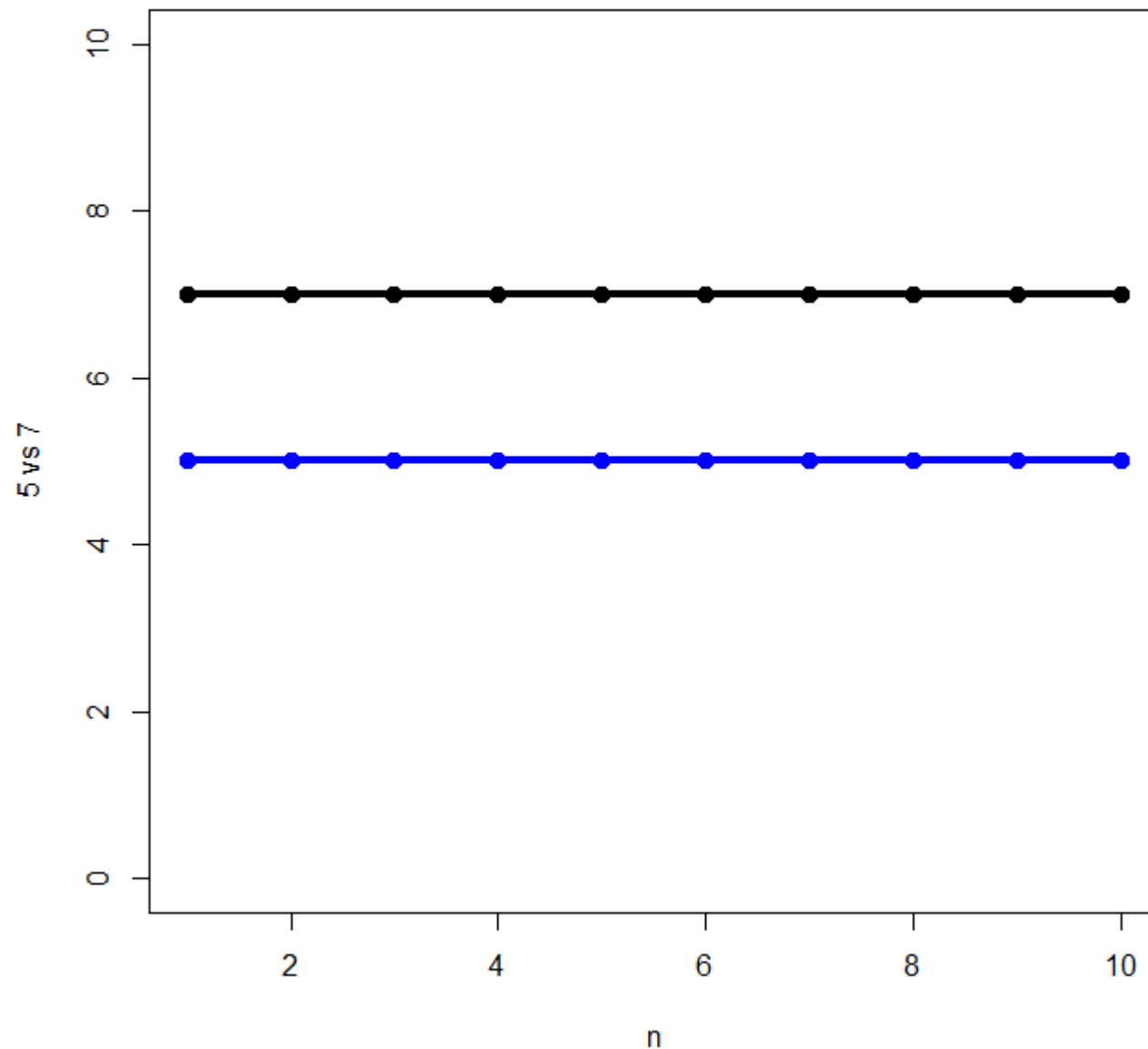
- $g(n) = n^2$
- Examples:  $c = 2, n' = 3$
- $n^2 + 2n < 2n^2$  for  $n \geq 3$
- Eg:  $n=3 \rightarrow f(n)=15, g(n)=18$

$$5 = O(n)$$



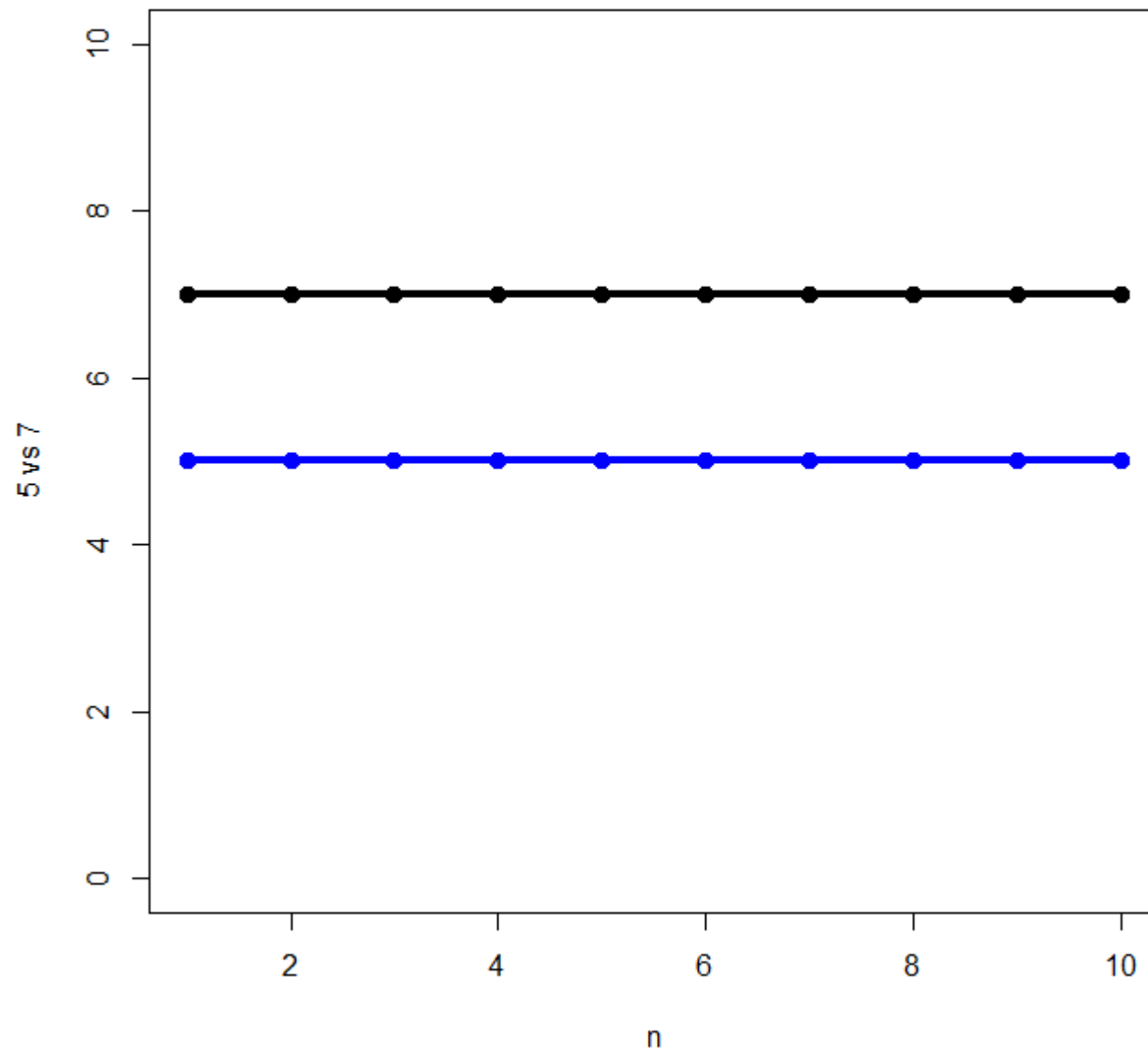
- $g(n) = n$
- Examples:  $c = 1, n' = 6$
- $5 < n$  for  $n \geq 6$

$$5 = O(7)$$



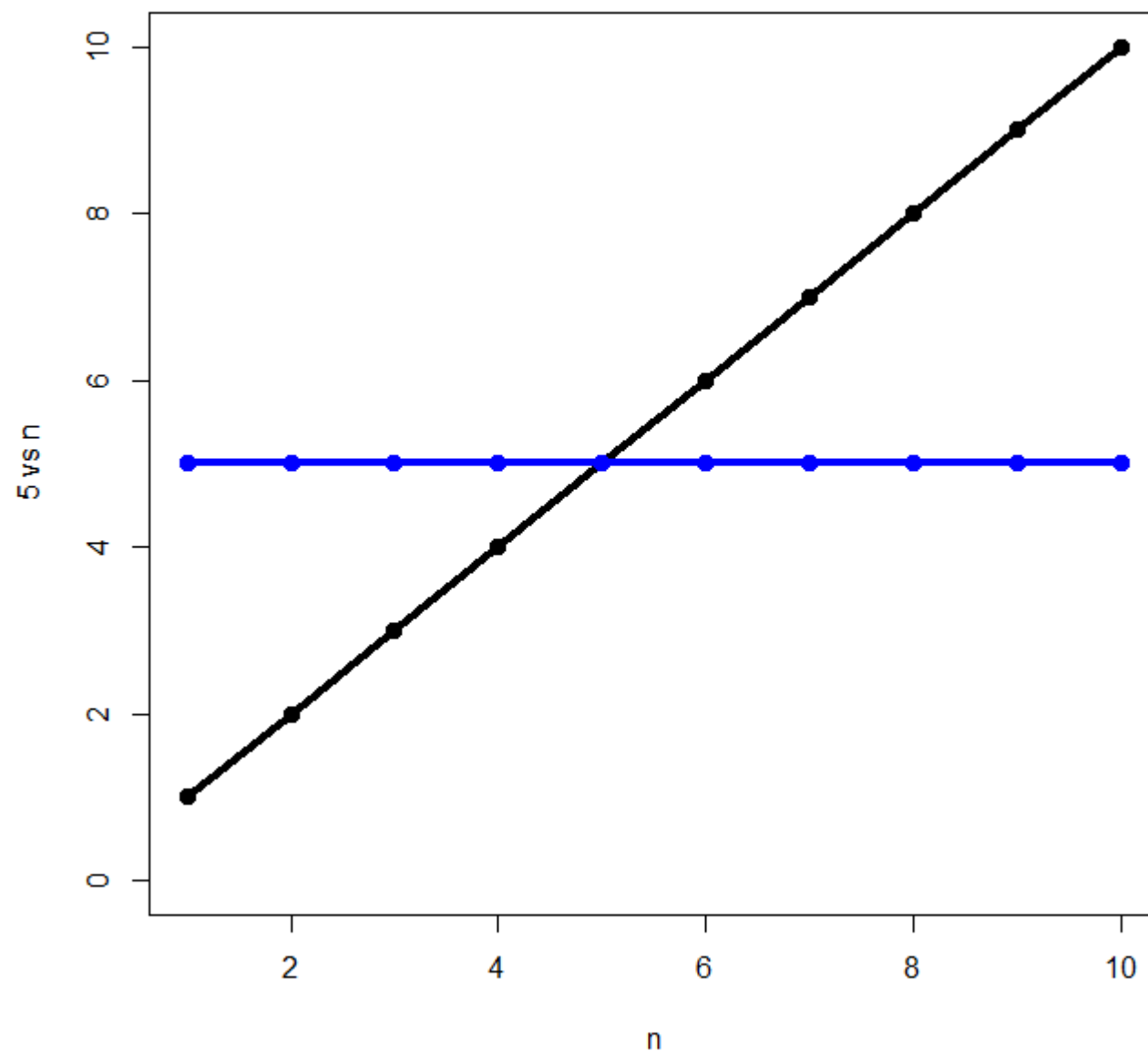
- $g(n) = 7$
- Examples:  $c = 1, n' = 1$
- $5 < 7$  for regardless of  $n$

$$5 = O(1)$$



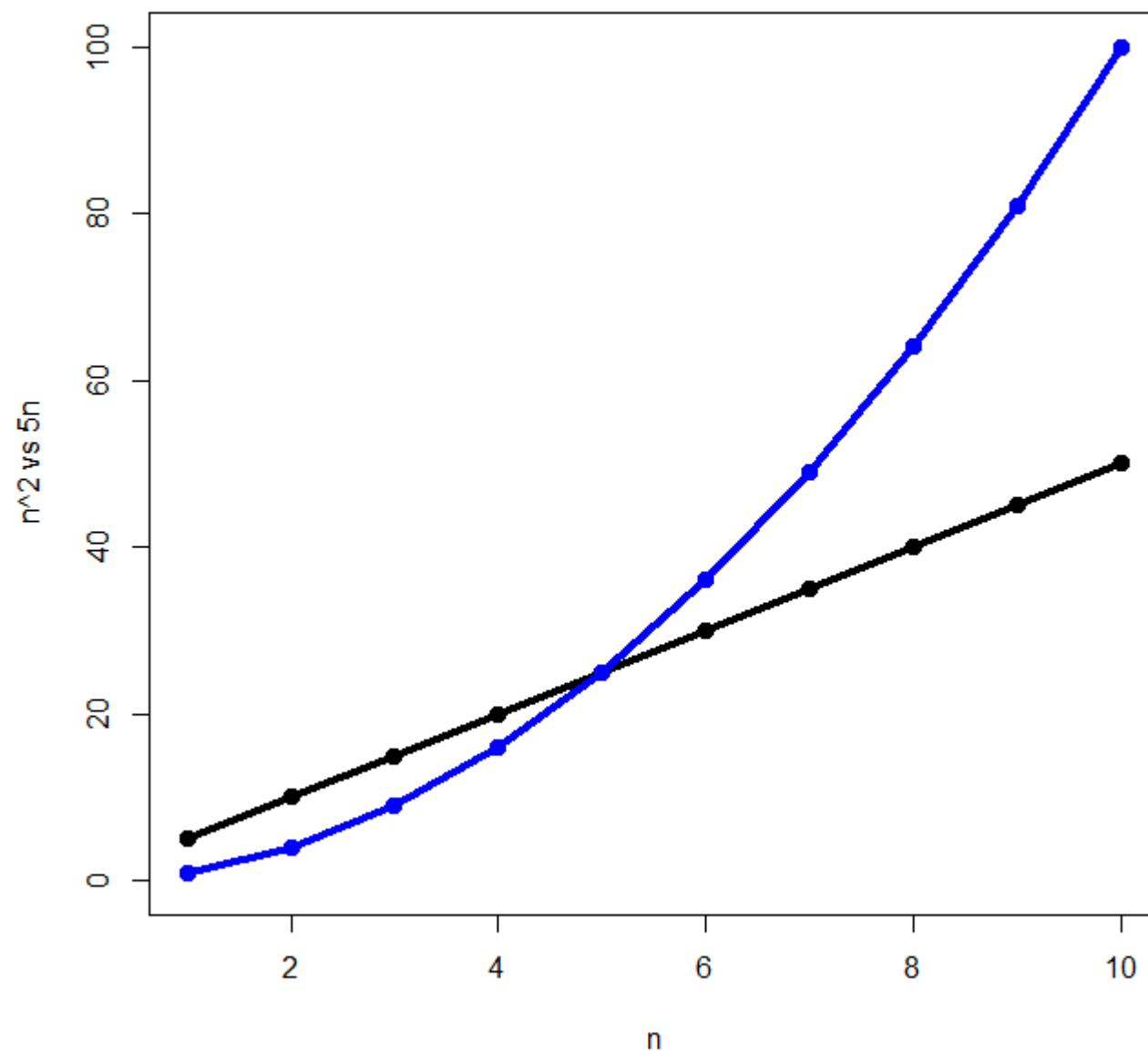
- $g(n) = 1$
- Examples:  $c = 7, n' = 1$
- $5 < 7$  for regardless of  $n$

$$n \neq O(1)$$



- $g(n) = 1$
- Whatever  $c$  I use (eg 5),  $f(n) < c$  will not hold for  $n' > c$ , eg  $n' = c + 1$

$$n^2 \neq O(n)$$



- $g(n) = n$
- Whatever  $c$  I use (eg 5),  $f(n) < cn$  will not hold for  $n' > c$ , eg  $n' = c + 1$



# Big O Examples

- $n = O(n)$
- $n = O(n^2)$
- $10n = O(n)$
- $10n + 5 = O(n)$
- $n^2 + 2n = O(n^2)$
- $5 = O(n)$
- $5 = O(7)$
- $5 = O(1)$
- $n \neq O(1)$
- $n^2 \neq O(n)$

# Big O Rules of Thumb

- When you have a polynomial, for upper bound just look at the highest exponent
- $an^3 + bn^2 + cn + d = O(n^3)$
- This means that, when analyzing an algorithm, you can ignore the parts with less impact
- When representing constants independent from the problem size, just use 1 by convention, eg  $O(1)$

# Notation

- $O(g(n)) = \{ f(n) : \text{there exists positive constants } c \text{ and } n', \text{ such that } 0 \leq f(n) \leq c * g(n) \text{ for all } n \geq n' \}$
- $O(g(n))$  is actually a *set* of functions
- $f(n) = O(g(n))$  is not fully correct as notation, as we use it to represent the fact that  $f(n)$  is one member of the set  $O(g(n))$
- $f(n) \in O(g(n))$  would be more precise, but often for simplicity you will see “=” instead of “ $\in$ ”

# Big $\Omega$ Lower Bound

- $f(n) = \Omega(g(n))$
- If there exists positive constants  $c$  and  $n'$ , such that  $0 \leq c * g(n) \leq f(n)$  for all  $n \geq n'$
- In other words,  $g(n)$  is a *lower bound*
- Useful to consider *how expensive* algorithm is even in the *best possible scenario*
- $\Omega(1)$  is a trivial lower bound valid for all functions

# Big $\Theta$ Tight Bound

- $f(n) = \Theta(g(n))$
- If there exists positive constants  $c, d$  and  $n'$ , such that  $0 \leq c * g(n) \leq f(n) \leq d * g(n)$  for all  $n \geq n'$
- In other words, this happens when the lower and upper bounds are asymptotically the same (and just differ by the constant)

```
public static int sum(int[] array) {  
    int sum = 0;  
    for(int i=0; i<array.length; i++) {  
        sum += array[i];  
    }  
    return sum;  
}
```

- In this case, the actual cost in number of instructions is  $f(n) = 3n + 4$
- Asymptotically, we can say that  $3n+4 = \Theta(n)$
- As the number of instructions does not depend on the content of the array, the *best case* and *worst case* for the runtime are the *same*

# Order Of Growth Classification

- 1: **constant** (best you can have)
- $\log(N)$ : **logarithmic** (very, very efficient)
- $N$ : **linear** (OK for most cases)
- $N \log(N)$ : **linearithmic** (OK for most cases)
- $N^2$ : **quadratic** (bearable, but things start to get expensive)
- $N^3$ : **cubic** (becoming painful)
- $2^N$ : **exponential** (*completely hopeless*, time to cry in a corner)

# Which Scales Best?

- $f(n) = O(n)$
- $g(n) = \Omega(n)$
- $t(n) = \Omega(n \log(n))$
- $k(n) = O(n^2)$
- $z(n) = \Theta(n)$

The **only** thing that I can say for sure is that  $f(n)$  is better than  $t(n)$ . Why???



	$1$	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$
$O(n)$							
$\Omega(n)$							
$\Omega(n \log(n))$							
$O(n^2)$							
$\Theta(n)$							

# In Practice

- Proving tight bounds is often infeasible
- Usually, from practical standpoint, the *worst case scenario* is what matters, so most discussions are about  $O(g(n))$
- Often, lower bound is not so interesting, as in happy-day scenario you get  $\Omega(1)$

# A Big Mistake

- **BIG MISTAKE:** assuming that an upper bound is tight, eg claiming  $f(n) = O(n)$  is necessarily better than  $g(n) = O(n^2)$ 
  - Although you might still want to prefer  $f(n)$  if you do not have any other information
- Even if a lower upper bound  $O$  exists, it might be too difficult to formally prove it, so  $O(n^2)$  might just be the best approximation we currently have
- However, it is also important to consider that worst case scenario is different from the average one, but proving averages is much more difficult

```

public static void doSomething(int[] array) {

    for(int i=0; i<array.length; i++) {
        for(int j=i; j<array.length; j++) {
            //... something
        }
    }
}

```

- Note the “int j=i”. What can we say about that function?
- Without digging into the math, we can say that, even in best case, first loop is taken at least once, so  $\Omega(n)$
- In worst case, not worse than assuming “int j=0”, so  $O(n^2)$ 
  - In other words, we can mentally consider a more expensive algorithm which does more iterations, but that is easier then to analyze
- Is the true complexity closer to the lower or to the upper bound? Maybe  $\Theta(n \log n)$  ???

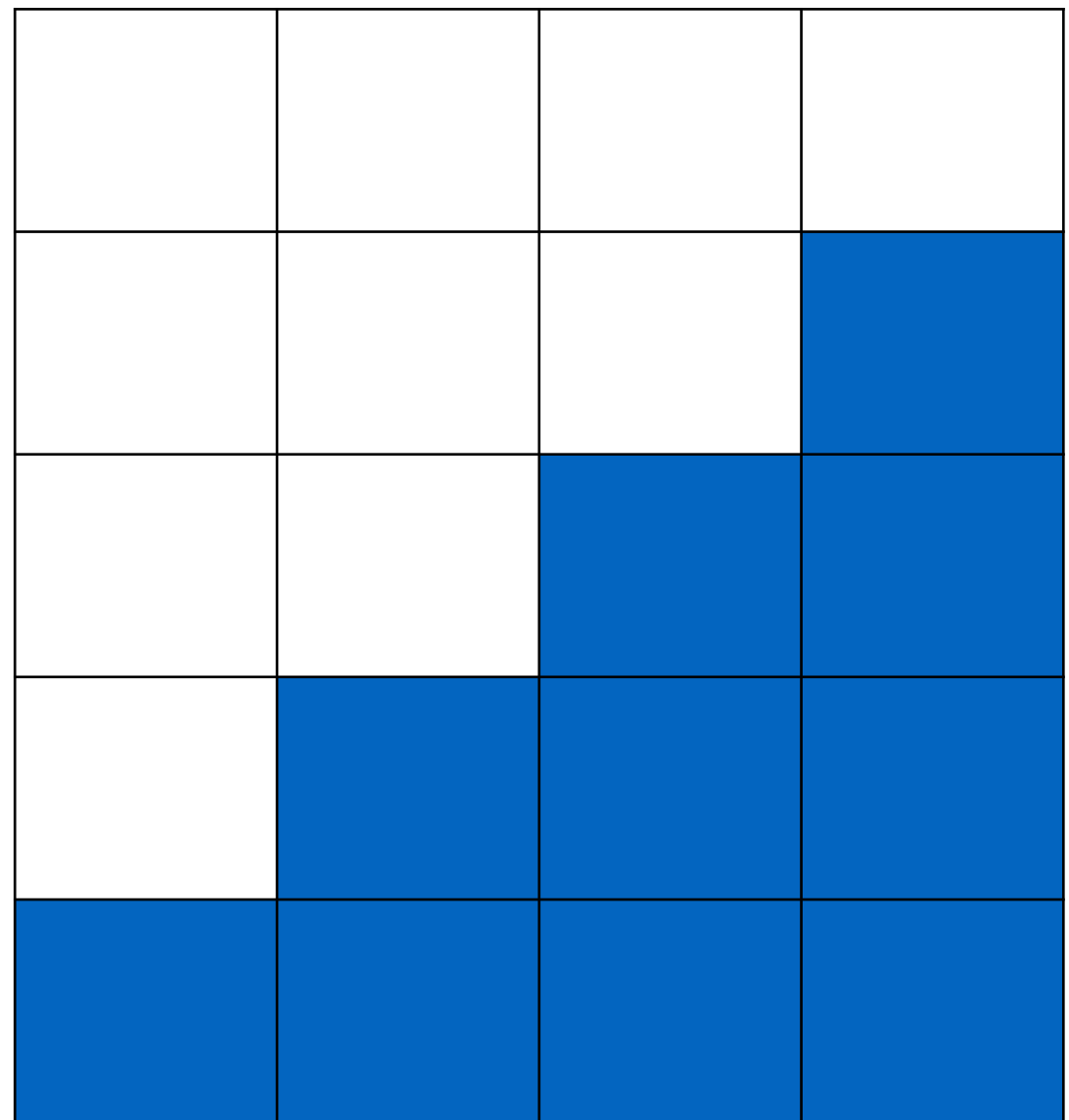
# Let's Dig Into the Math...

```
public static void doSomething(int[] array) {  
    for(int i=0; i<array.length; i++) {  
        for(int j=i; j<array.length; j++) {  
            //... something  
        }  
    }  
}
```

- Outer loop is taken N times
- Inner loop is shorter by 1 at each iteration
- $N + (N - 1) + (N - 2) + \dots + 1 = \sum_{i=0}^N i = \frac{1}{2}N(N + 1) = \frac{1}{2}(N^2 + N) = \Theta(N^2)$

$$\sum_{i=0}^N i = \frac{1}{2}N(N+1)$$

- Think about a rectangle with sides  $N$  and  $N+1$
- Its area is  $N * (N+1)$
- But we are interested only in the colored area, so divide by 2
- $1+2+3+4 = 4 * 5 / 2 = 10$



# Sorting

# Consider a Playlist

...

Browse

Radio

YOUR LIBRARY

Recently Played

Songs

Albums

Artists

Stations

Local Files

Podcasts

PLAYLISTS

While marking exams

< >

UPGRADE

Andrea

PLAYLIST

While marking exams

Created by Andrea • 6 songs, 25 min

PLAY

...

FOLLOWERS 0

Download ☐

	TITLE	ARTIST	ALBUM		
+	Paranoid - 2009 Remastered Version	Black Sabbath	Paranoid (2009 Remastered Version)	2 minutes ago	2:47
+	Iron Man - 2009 Remastered Version	Black Sabbath	Paranoid (2009 Remastered Version)	2 minutes ago	5:54
+	Square Hammer	Ghost	Meliora (Redux)	a minute ago	3:58
+	The Number Of The Beast - 1998 Remastered Version	Iron Maiden	The Number Of The Beast (1998 R...	2 minutes ago	4:51
+	Run to the Hills - 1998 Remastered Version	Iron Maiden	The Number Of The Beast (1998 R...	2 minutes ago	3:54
+	Toxicity	System Of A Down	Toxicity	a minute ago	3:39



# Sort by Title or Artist, Ascending or Descending

The image shows a Spotify interface with a playlist titled "While marking exams" created by Andrea. The playlist contains 6 songs and has a duration of 25 minutes. The interface is split into two panels. The left panel shows the playlist's sidebar with a search bar and a list of songs. The right panel shows the playlist's main view with a search bar and a list of songs. Two purple arrows point to the sorting options in the top right corner of the playlist view. One arrow points to the "ARTIST" sorting option, and the other points to the "TITLE" sorting option.

**PLAYLIST**  
**While marking exams**  
Created by Andrea • 6 songs, 25 min

**PLAY** **...**

**UPGRADE** **Andrea** **▼**

**Filter**

TITLE	ARTIST	ALBUM	Time	Duration
+ Paranoid - 2009 Remastered Version	Black Sabbath	Paranoid (2009 Remastered Version)	2 minutes ago	2:47
+ Iron Man - 2009 Remastered Version	Black Sabbath	Paranoid (2009 Remastered Version)	4 minutes ago	5:54
+ Square Hammer	Ghost	Meliora (Redux)	2 minutes ago	3:58
+ The Number Of The Beast - 1998 Remastered Version	Iron Maiden	The Number Of The Beast (1998 R...	3 minutes ago	3:54
+ Run to the Hills - 1998 Remastered Version	Iron Maiden	The Number Of The Beast (1998 R...	3 minutes ago	4:51
+ Toxicity	System Of A Down	Toxicity	3 minutes ago	3:39

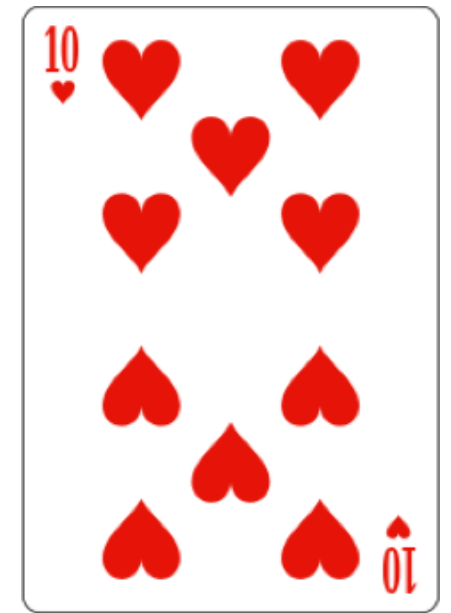
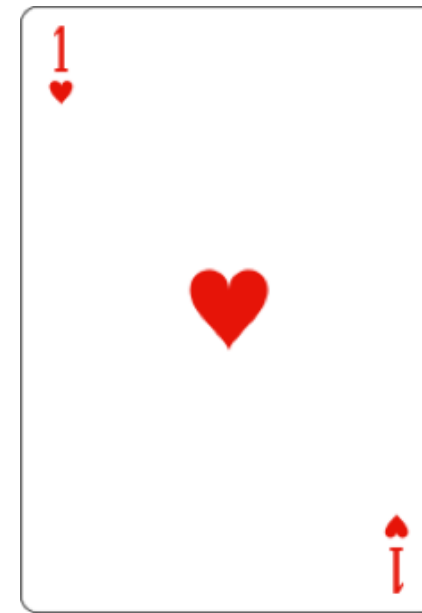
**PLAYLIST**  
**While marking exams**  
Created by Andrea • 6 songs, 25 min

**PLAY** **...**

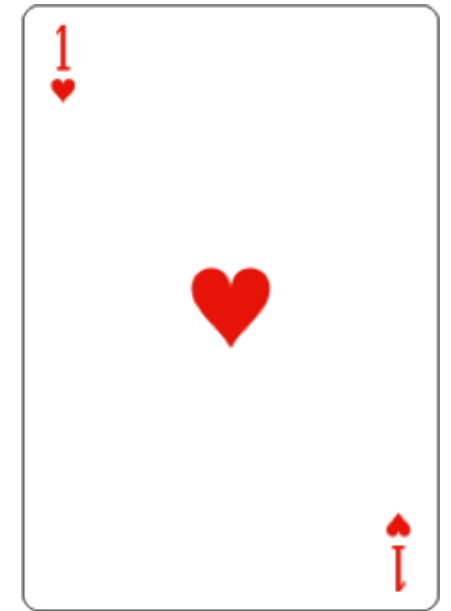
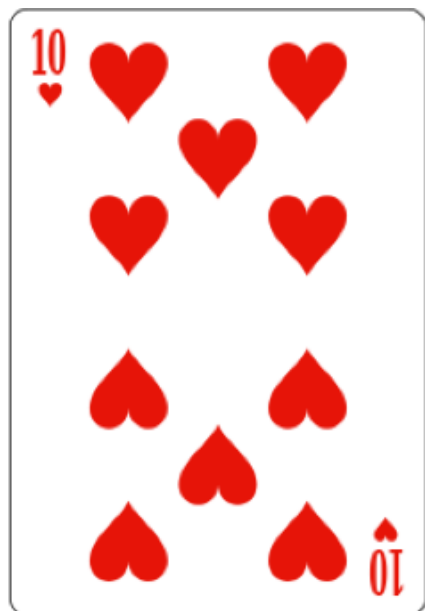
**UPGRADE** **Andrea** **▼**

**Filter**

TITLE	ARTIST	ALBUM	Time	Duration
+ Toxicity	System Of A Down	Toxicity	3 minutes ago	3:39
+ Square Hammer	Ghost	Meliora (Redux)	2 minutes ago	3:58
+ Run to the Hills - 1998 Remastered Version	Iron Maiden	The Number Of The Beast (1998 R...	3 minutes ago	3:54
+ Paranoid - 2009 Remastered Version	Black Sabbath	Paranoid (2009 Remastered Version)	4 minutes ago	2:47
+ The Number Of The Beast - 1998 Remastered Version	Iron Maiden	The Number Of The Beast (1998 R...	3 minutes ago	4:51
+ Iron Man - 2009 Remastered Version	Black Sabbath	Paranoid (2009 Remastered Version)	4 minutes ago	5:54



How do you sort when playing cards?



# Sorting Algorithms

- Many different sorting algorithms, with different properties
- Given two items  $A$  and  $B$ , just need a *comparator* that can state which one is greater or equal
  - easy to say that 5 greater than 2, but what does it mean that song  $A$  is greater than song  $B$ ? e.g., could look at alphabetic ordering of titles or artist names
- Most language APIs provides good defaults
  - Unless very large data, default will be fine 99% of the cases
- Sorting is very popular in programming
  - Important to understand how it works under the hood
- Tractable mathematically
  - So good example to show how to analyze algorithms

# Sorting Algorithms

- Bubble Sort
- Insertion Sort
- Merge Sort (next class)
- Quick Sort (next class)
- There are more, but those are the most famous that you need to know
- Good way to see a problem been solved in many different ways

# Bubble Sort

- *Easiest* sorting algorithms
- From left to right
- Look at adjacent cards, and swap them if not in order
- Repeat from left to right till no more swap



No swap, they are in order

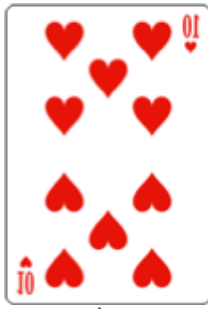
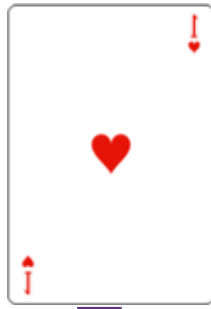


Swap

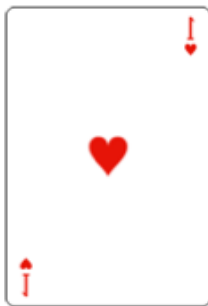


No swap, they are in order





Swap



- Restart from beginning.
- At each iteration, at least one card will be in right position, as it *bubbles up* to the top.

# Runtime of Bubble Sort

- To sort  $N$  cards, need *at most*  $N$  iterations, in which you check *at most*  $N-1$  pairs
- Even if already sorted, need to check each of  $N-1$  pairs at least once, to see if indeed sorted
- $\Omega(N)$  and  $O(N^2)$  pair comparisons



# Insertion Sort

- An array of size 0 or 1 is always considered sorted
- From left to right, till length  $N$
- $K$ -leftmost values are sorted
- Position  $K+1$  is not sorted, insert it in the first  $K$ 
  - by swapping adjacent elements, like in Bubble Sort

4	5	1	3	2	6
---	---	---	---	---	---

 $K=0$ 

4	5	1	3	2	6
---	---	---	---	---	---

 $K=1$ 

4	5	1	3	2	6
---	---	---	---	---	---

 $K=2$ 

4	5	1	3	2	6
---	---	---	---	---	---

 $K=3$ 

4	1	5	3	2	6
---	---	---	---	---	---

 swap

1	4	5	3	2	6
---	---	---	---	---	---

 swap

# Cont.

1	4	5	3	2	6
---	---	---	---	---	---

K=4

1	4	3	5	2	6
---	---	---	---	---	---

swap

1	3	4	5	2	6
---	---	---	---	---	---

swap

1	3	4	5	2	6
---	---	---	---	---	---

K=5

1	3	4	2	5	6
---	---	---	---	---	---

swap

1	3	2	4	5	6
---	---	---	---	---	---

swap

1	2	3	4	5	6
---	---	---	---	---	---

swap

1	2	3	4	5	6
---	---	---	---	---	---

K=6

- Best case: already sorted, e.g., 1-2-3-4-5-6, need to do  $N-1$  comparisons, so  $\Omega(N)$
- Worst case: opposite order, e.g., 6-5-4-3-2-1, each element needs to be compared and swapped with all previous  $K$  ones, so  $O(N^2)$

# Best vs. Worst Case

- “Best case” is not when  $N=0$
- Runtime might not only depend on size  $N$ , but also on the **content** of the input
  - an array already sorted will likely take less time to sort than a shuffled one
  - on the other hand, summing all integers in an array will likely have no difference between Best and Worst case, ie  $\Theta(N)$
- Considering all possible inputs in a mathematical formula would not be viable
  - that is why we talk about BEST and WORST cases in relation to  $N$

# Homework

- Study Book Chapter 1.4 and 2.1
  - Note: in the book, the authors use the  $\sim$  notation instead of the standard  $O$ ,  $\Omega$  and  $\Theta$ . They explain their reason in the Q&A for such weird choice, but I do disagree
- Study code in the *org.pg4200.les03* package
- Do exercises in *exercises/ex03*
- Extra: do exercises in the book