Data Structures and Algorithms

Lesson 1: *Know How Long A Program Takes*

TSP, Sorting, time complexity, Big-Theta, Big-O, Big-Omega

Outline

1. **TSP**
2. Sorting
3. Time Complexity, Big Theta, Big O
4. Measuring Complexity

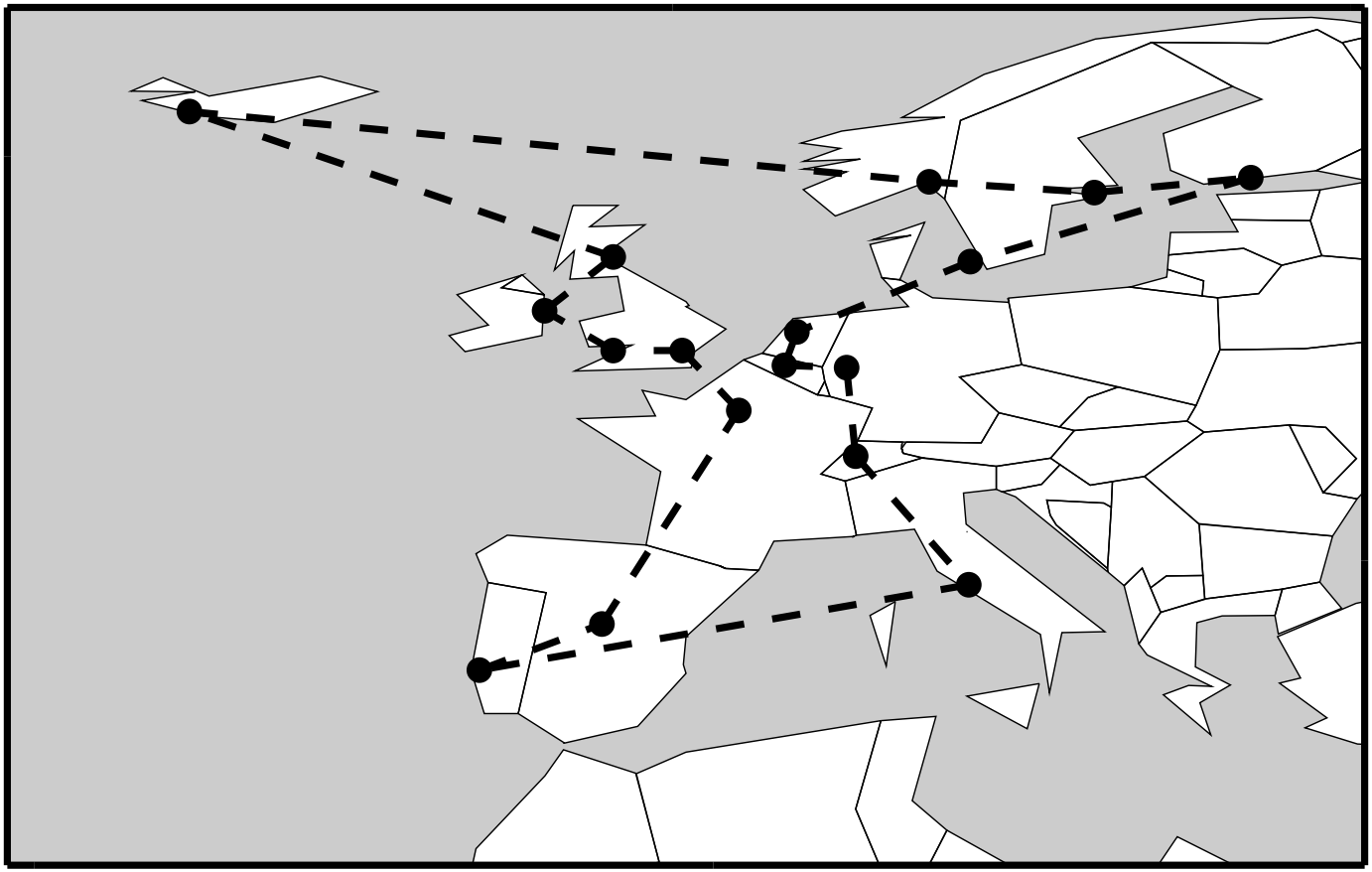
Travelling Salesperson Problem

- Given a set of cities
- A table of distances between cities
- Find the shortest tour which goes through each city exactly once and returns to the start

Example of Distance Table

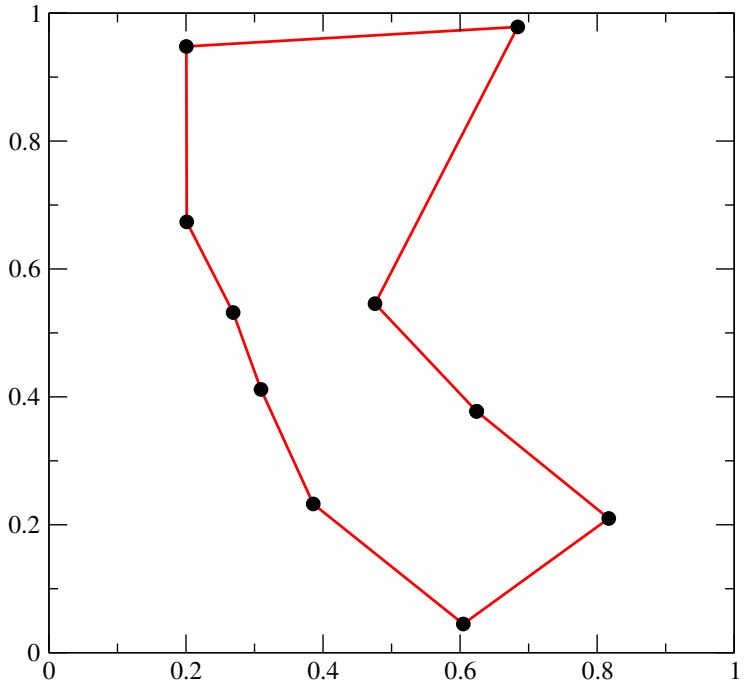
	Lon	Car	Dub	Edin	Reyk	Oslo	Sto	Hel	Cop	Amst	Bru	Bonn	Bern	Rome	Lisb	Madr	Par
London	0	223	470	538	1896	1151	1426	1816	950	349	312	503	743	1429	1587	1265	337
Cardiff	223	0	290	495	1777	1277	1589	1985	1139	564	533	725	927	1600	1492	1233	492
Dublin	470	290	0	350	1497	1267	1628	2026	1239	756	775	956	1207	1886	1638	1449	777
Edinburgh	538	495	350	0	1374	933	1314	1708	984	662	758	896	1243	1931	1964	1728	872
Reykjavik	1896	1777	1497	1374	0	1746	2134	2418	2104	2020	2130	2255	2617	3304	2949	2892	2232
Oslo	1151	1277	1267	933	1746	0	416	788	481	917	1088	1048	1459	2011	2739	2390	1343
Stockholm	1426	1589	1628	1314	2134	416	0	398	518	1126	1281	1181	1542	1978	2987	2593	1543
Helsinki	1816	1985	2026	1708	2418	788	398	0	881	1504	1650	1530	1856	2203	3360	2950	1910
Copenhagen	950	1139	1239	984	2104	481	518	881	0	625	769	662	1036	1538	2479	2076	1030
Amsterdam	349	564	756	662	2020	917	1126	1504	625	0	173	235	629	1296	1860	1480	428
Brussels	312	533	775	758	2130	1088	1281	1650	769	173	0	194	489	1174	1710	1315	262
Bonn	503	725	956	896	2255	1048	1181	1530	662	235	194	0	422	1067	1843	1420	400
Bern	743	927	1207	1243	2617	1459	1542	1856	1036	629	489	422	0	689	1630	1156	440
Rome	1429	1600	1886	1931	3304	2011	1978	2203	1538	1296	1174	1067	689	0	1862	1365	1109
Lisbon	1587	1492	1638	1964	2949	2739	2987	3360	2479	1860	1710	1843	1630	1862	0	500	1452
Madrid	1265	1233	1449	1728	2892	2390	2593	2950	2076	1480	1315	1420	1156	1365	500	0	1054
Paris	337	492	777	872	2232	1343	1543	1910	1030	428	262	400	440	1109	1452	1054	0

Example Tour



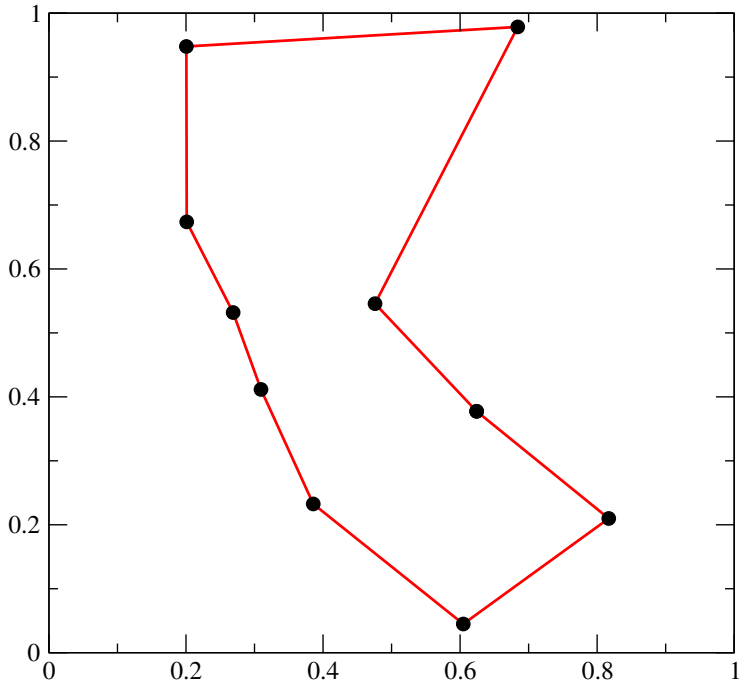
Brute Force

- I wrote a program to solve TSP by enumerating all paths and finding the shortest
- I checked that it worked on some problems with 10 cities
- It takes just under half a second to solve this problem
- I set the program running on a 100 city problem

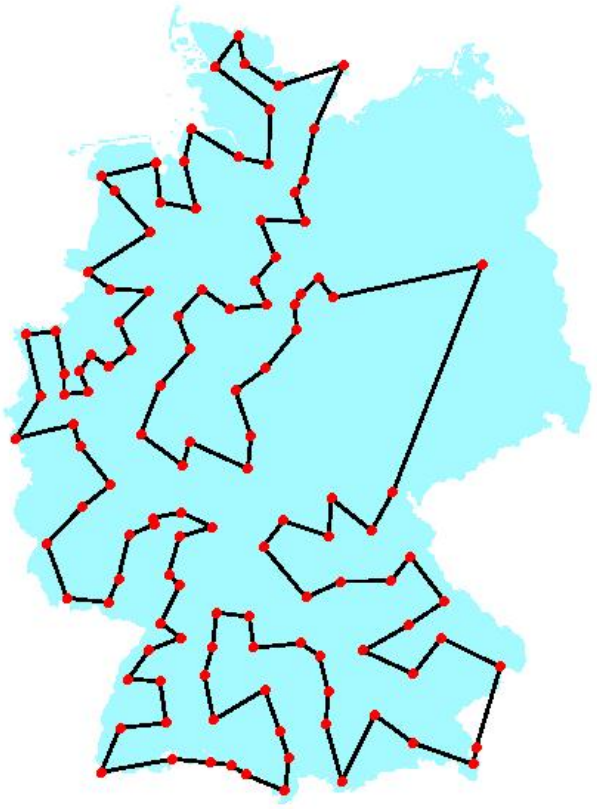


Brute Force

- I wrote a program to solve TSP by enumerating all paths and finding the shortest
- I checked that it worked on some problems with 10 cities
- It takes just under half a second to solve this problem
- I set the program running on a 100 city problem–**How long will it take to finish?**

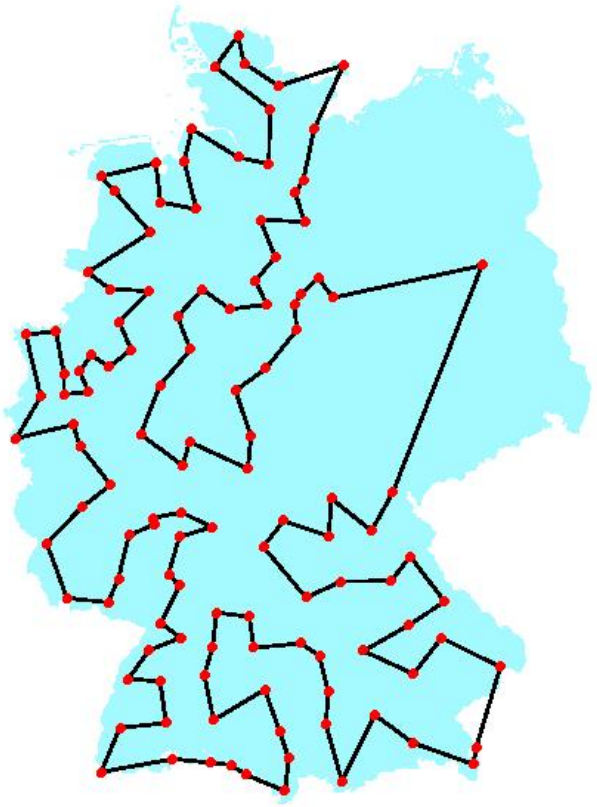


How Many Possible Tours Are There?



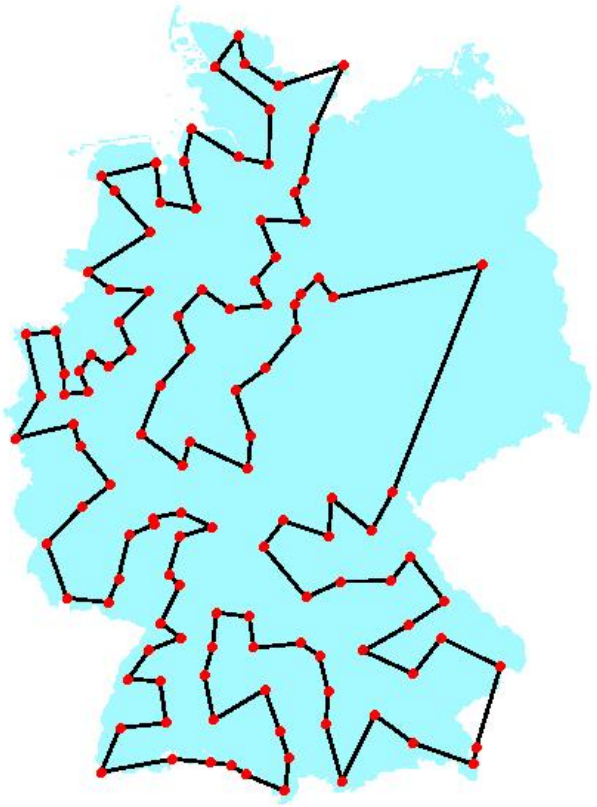
- For 100 cities how many possible tours are there?

How Many Possible Tours Are There?



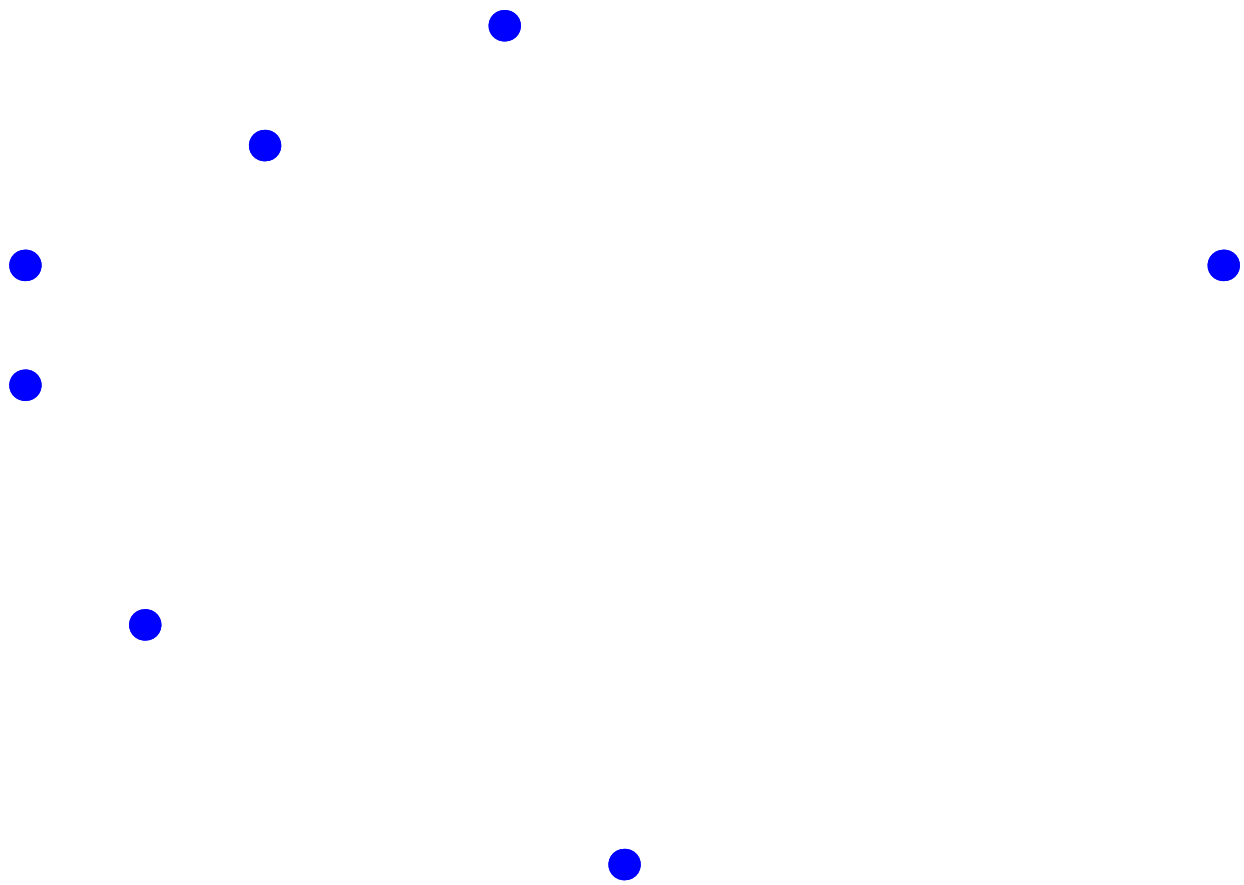
- For 100 cities how many possible tours are there?
- It doesn't matter where we start

How Many Possible Tours Are There?



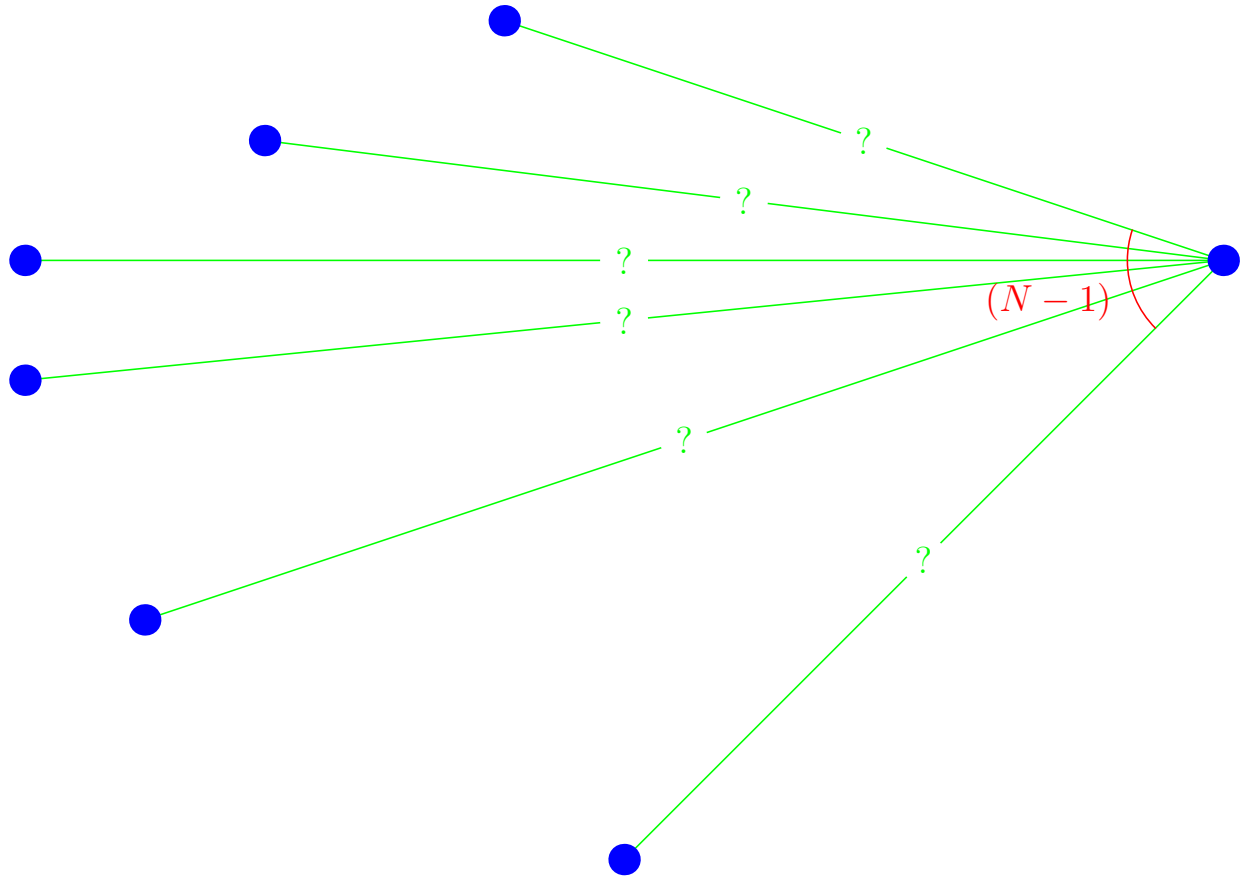
- For 100 cities how many possible tours are there?
- It doesn't matter where we start
- Starting from Berlin there are 99 cites we can try next

Counting Tours



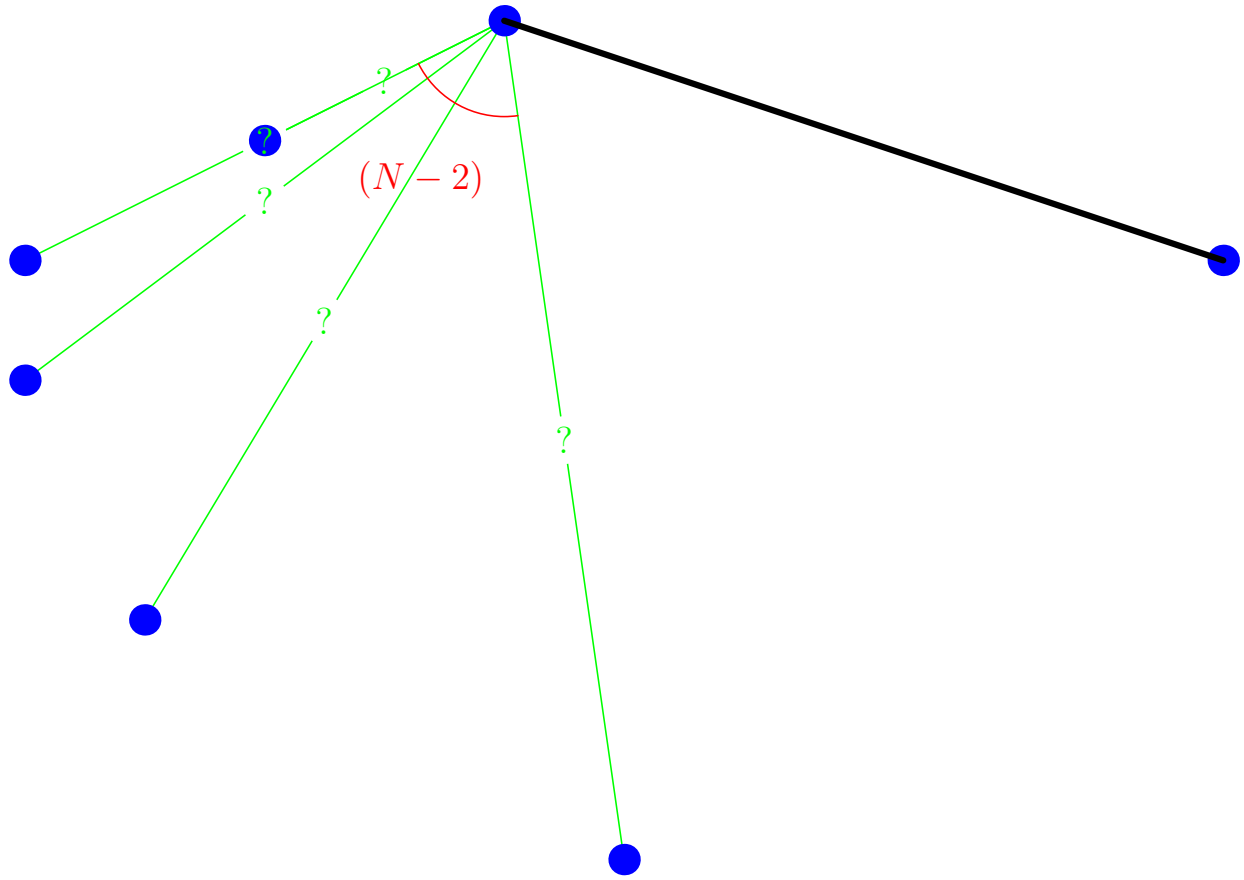
Number of tours =

Counting Tours



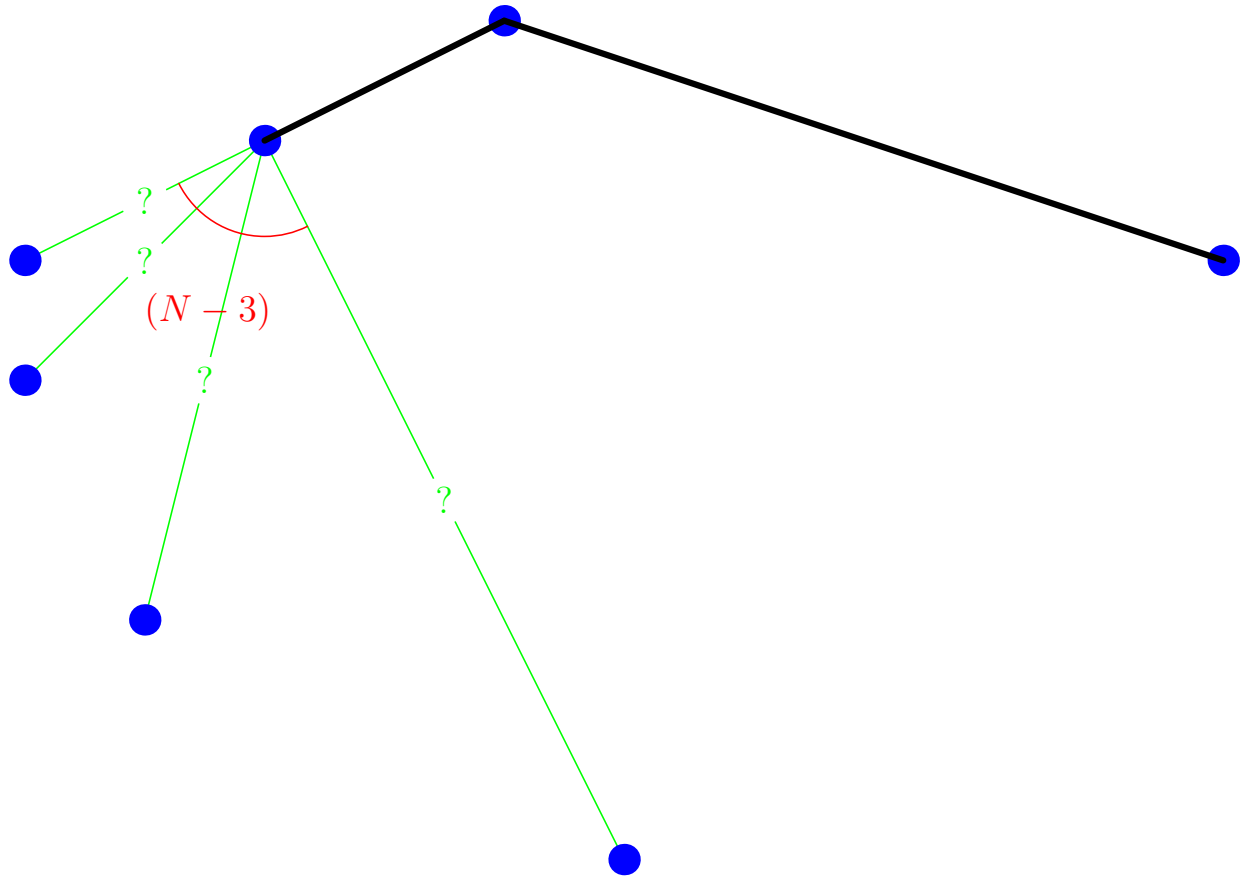
Number of tours = $(N-1)$

Counting Tours



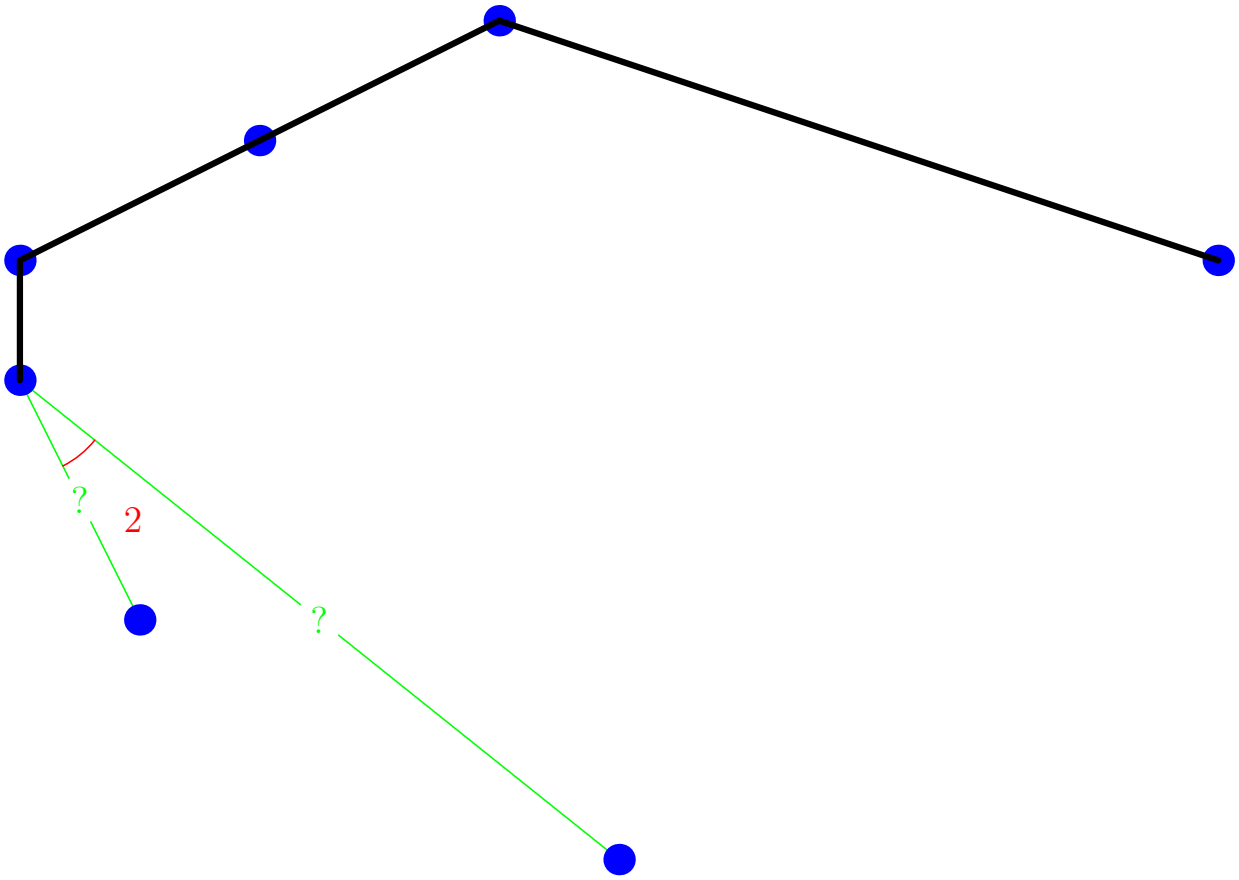
Number of tours = $(N-1) \times (N-2)$

Counting Tours



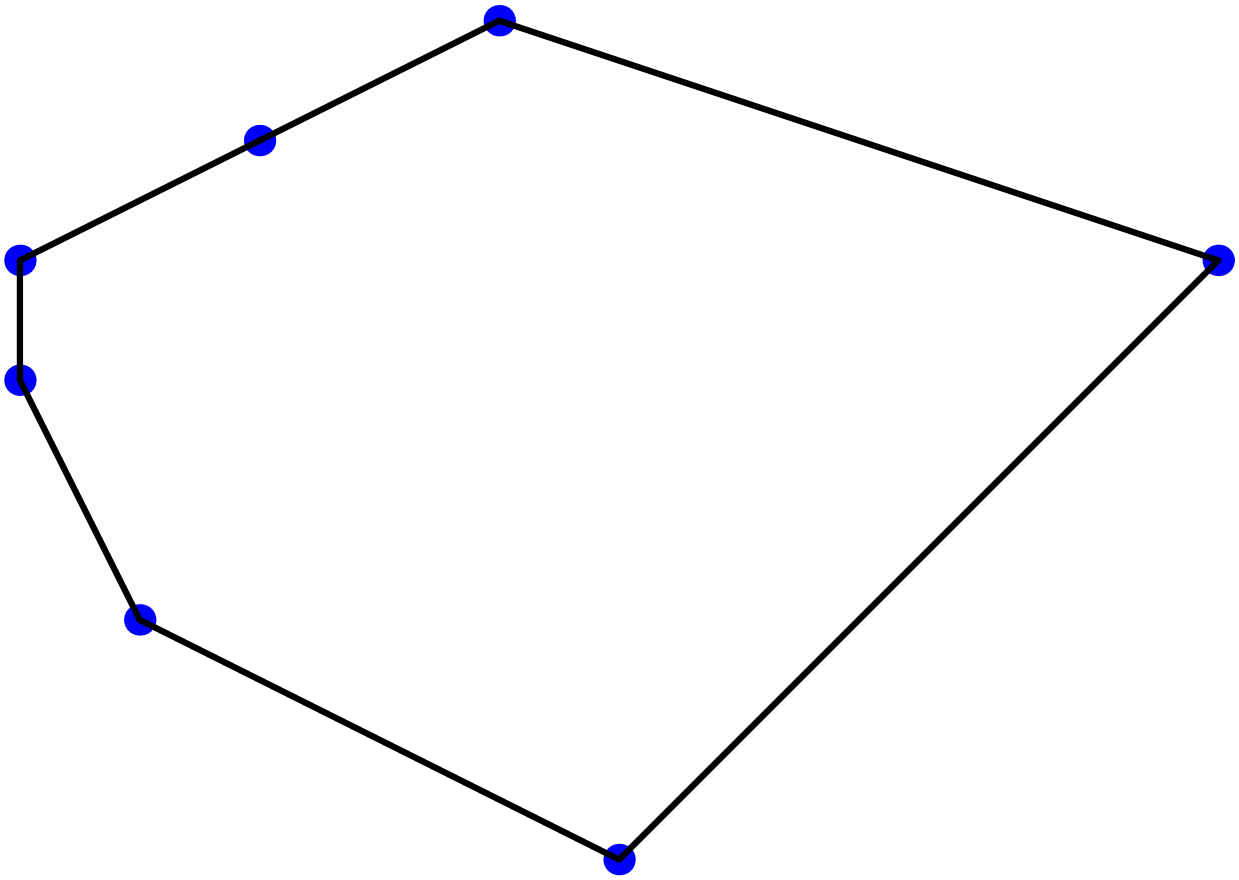
Number of tours = $(N-1) \times (N-2) \times (N-3)$

Counting Tours



Number of tours = $(N-1) \times (N-2) \times (N-3) \times \cdots 2$

Counting Tours



Number of tours = $(N-1) \times (N-2) \times (N-3) \times \cdots 2 \times 1 = (N-1)!$

How Long Does It Take?

- The direction we go in is irrelevant

How Long Does It Take?

- The direction we go in is irrelevant
- Total number of tours is $99!/2$

How Long Does It Take?

- The direction we go in is irrelevant
- Total number of tours is $99!/2$
- **Any more guesses how long it will take?**

How Big is 99 Factorial?

- $99! = 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1 = ?$

How Big is 99 Factorial?

- $99! = 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1 = ?$
- Upper bound

How Big is 99 Factorial?

- $99! = 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1 = ?$
- Upper bound

$$\begin{aligned} 99! &= 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1 \\ 99! &< 99 \cdot 99 \cdot 99 \cdots 99 \cdot 99 = 99^{99} \end{aligned}$$

How Big is 99 Factorial?

- $99! = 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1 = ?$
- Upper bound

$$\begin{aligned} 99! &= 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1 \\ 99! &< 99 \cdot 99 \cdot 99 \cdots 99 \cdot 99 = 99^{99} \end{aligned}$$

- Lower bound

How Big is 99 Factorial?

- $99! = 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1 = ?$
- Upper bound

$$\begin{aligned} 99! &= 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1 \\ 99! &< 99 \cdot 99 \cdot 99 \cdots 99 \cdot 99 = 99^{99} \end{aligned}$$

- Lower bound

$$\begin{aligned} 99! &= 99 \cdot 98 \cdot 97 \cdots 50 \cdot 49 \cdots 2 \cdot 1 \\ 99! &> 50 \cdot 50 \cdot 50 \cdots 50 \cdot 1 \cdots 1 \cdot 1 = 50^{50} \end{aligned}$$

How Long Does It Take?

- For $N > 1$

$$\left(\frac{N}{2}\right)^{N/2} < N! < N^N$$

How Long Does It Take?

- For $N > 1$

$$\left(\frac{N}{2}\right)^{N/2} < N! < N^N$$

- $99!/2 = 4.666 \times 10^{155}$

How Long Does It Take?

- For $N > 1$

$$\left(\frac{N}{2}\right)^{N/2} < N! < N^N$$

- $99!/2 = 4.666 \times 10^{155}$
- How long does it take to check all possible tours?

How Long Does It Take?

- For $N > 1$

$$\left(\frac{N}{2}\right)^{N/2} < N! < N^N$$

- $99!/2 = 4.666 \times 10^{155}$
- How long does it take to check all possible tours?
  - ⋆ We computed about $200\,000$ tours in half a second

How Long Does It Take?

- For $N > 1$

$$\left(\frac{N}{2}\right)^{N/2} < N! < N^N$$

- $99!/2 = 4.666 \times 10^{155}$

- How long does it take to check all possible tours?
  - ⋆ We computed about $200\,000$ tours in half a second
  - ⋆ 3.15×10^7sec $= 1$ year

How Long Does It Take?

- For $N > 1$

$$\left(\frac{N}{2}\right)^{N/2} < N! < N^N$$

- $99!/2 = 4.666 \times 10^{155}$

- How long does it take to check all possible tours?
  - ⋆ We computed about $200\,000$ tours in half a second
  - ⋆ 3.15×10^7sec $= 1$ year
  - ⋆ Age of Universe ≈ 15 billion years

Answer

Answer

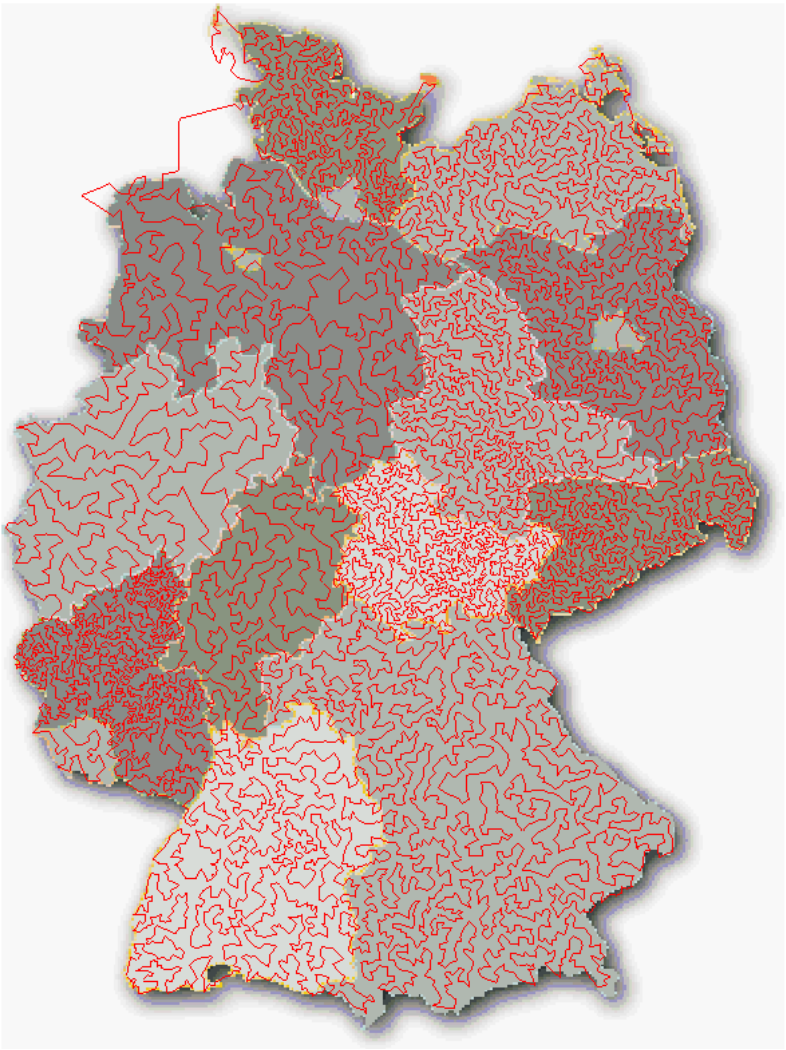
- 2.72×10^{132} ages of the universe!

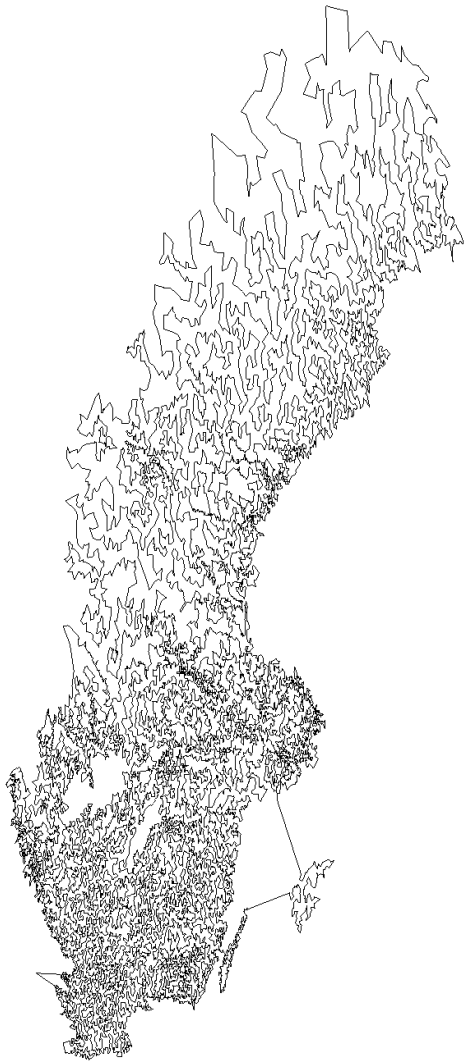
Answer

- 2.72×10^{132} ages of the universe!
- Incidental

$$
\begin{aligned}
99!/2 \;=\; & 46663107721972076340849619428133350 \\
& 24535798413219081073429648194760879 \\
& 99966149578044707319880782591431268 \\
& 48960413611879125592605458432000000 \\
& 0000000000000000
\end{aligned}
$$

Record TSP Solved – $15\,112$ and $24\,978$ Cities





In Case You're Curious

- Number of tours: $15111!/2 = 7.3 \times 10^{56592}$
- Current record $24\,978$ cities with 1.9×10^{98992} tours
- The algorithm for finding the optimum path does not look at every possible path

Lessons

- Even relatively small inputs can take you an astronomical time to deal with using simple algorithms

Lessons

- Even relatively small inputs can take you an astronomical time to deal with using simple algorithms
- As a professional programmer you need to have an estimate for how long an algorithm takes

Lessons

- Even relatively small inputs can take you an astronomical time to deal with using simple algorithms
- As a professional programmer you need to have an estimate for how long an algorithm takes
- For the 100 city problem, if
  - ⋆ I had 10^{87} cores, one for every particle in the Universe

Lessons

- Even relatively small inputs can take you an astronomical time to deal with using simple algorithms
- As a professional programmer you need to have an estimate for how long an algorithm takes
- For the 100 city problem, if
  - ⋆ I had 10^{87} cores, one for every particle in the Universe
  - ⋆ I could compute a tour distance in 3×10^{-24} seconds, the time it takes light to cross a proton

Lessons

- Even relatively small inputs can take you an astronomical time to deal with using simple algorithms
- As a professional programmer you need to have an estimate for how long an algorithm takes
- For the 100 city problem, if
  - ⋆ I had 10^{87} cores, one for every particle in the Universe
  - ⋆ I could compute a tour distance in 3×10^{-24} seconds, the time it takes light to cross a proton

  The brute force algorithm would still take $10^{39}\times$ the age of the universe!!!

Lessons

- Even relatively small inputs can take you an astronomical time to deal with using simple algorithms
- As a professional programmer you need to have an estimate for how long an algorithm takes
- For the 100 city problem, if
  - ⋆ I had 10^{87} cores, one for every particle in the Universe
  - ⋆ I could compute a tour distance in 3×10^{-24} seconds, the time it takes light to cross a proton

  The brute force algorithm would still take $10^{39}\times$ the age of the universe!!!
- **Smart algorithms can make a much bigger difference than fast computers!**

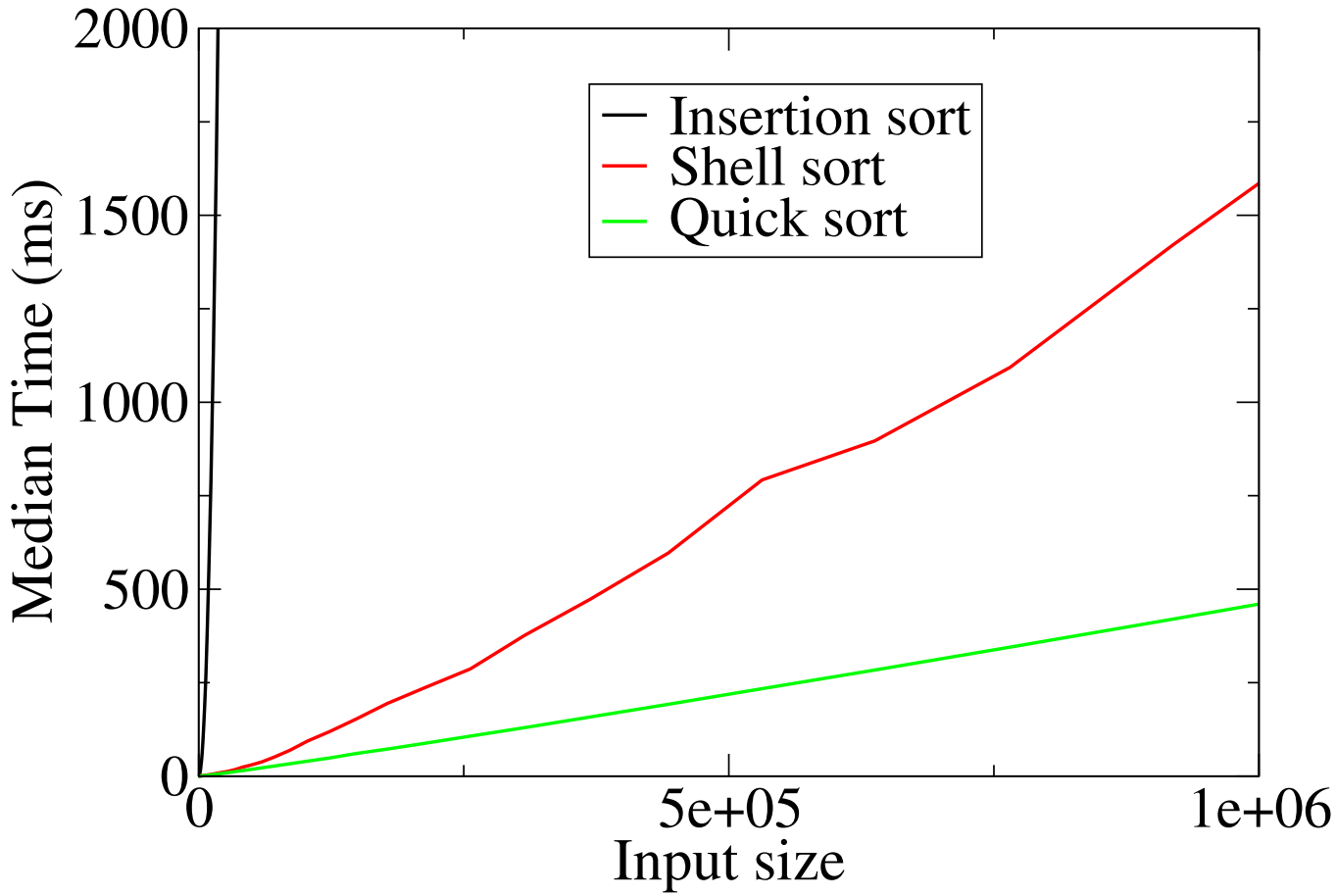
Outline

1. TSP
2. **Sorting**
3. Time Complexity, Big Theta, Big O
4. Measuring Complexity

Sort

- Comparison between common sort algorithms
  - ⋆ Insertion sort – an easy algorithm to code
  - ⋆ Shell sort – invented in 1959 by Donald Shell
  - ⋆ Quick sort – invented in 1961 by Tony Hoare
- These take an array of numbers and return a sorted array
- Sorting is a very common problem, so you care about how long it takes to solve it

Empirical Run Times



Lessons

- There is a right and wrong way to do easy problems

Lessons

- There is a right and wrong way to do easy problems
- You only really care when you are dealing with large inputs

Lessons

- There is a right and wrong way to do easy problems
- You only really care when you are dealing with large inputs
- Good algorithms are difficult to come up with

Lessons

- There is a right and wrong way to do easy problems
- You only really care when you are dealing with large inputs
- Good algorithms are difficult to come up with
- We would like to analyse/quantify the performance of an algorithm – how much better is quick sort than insertion sort?

Outline

1. TSP
2. Sorting
3. **Time Complexity, Big Theta, Big O**
4. Measuring Complexity

Estimating Run Times

- We would like to estimate the run times of algorithms
  - ⋆ This depends on the hardware (how fast your computer is)
  - ⋆ We want to abstract away differences in hardware!

Estimating Run Times

- We would like to estimate the run times of algorithms
  - ⋆ This depends on the hardware (how fast your computer is)
  - ⋆ We want to abstract away differences in hardware!
- We could count the number of **elementary operations** (e.g. addition, subtraction, comparison, . . .)
  - ⋆ An elementary operation takes a fixed amount of time to perform
  - ⋆ Thus counting elementary operations gives a good measure of the time taken by the algorithm, up to a constant factor

Time Complexity

- The **worst-case time complexity** of an algorithm is a function $T : \mathbb{N} \to \mathbb{N}$ where

Time Complexity

- The **worst-case time complexity** of an algorithm is a function $T : \mathbb{N} \to \mathbb{N}$ where
  - $T(n)$ is the maximum number of elementary operations the algorithm uses on inputs of size n

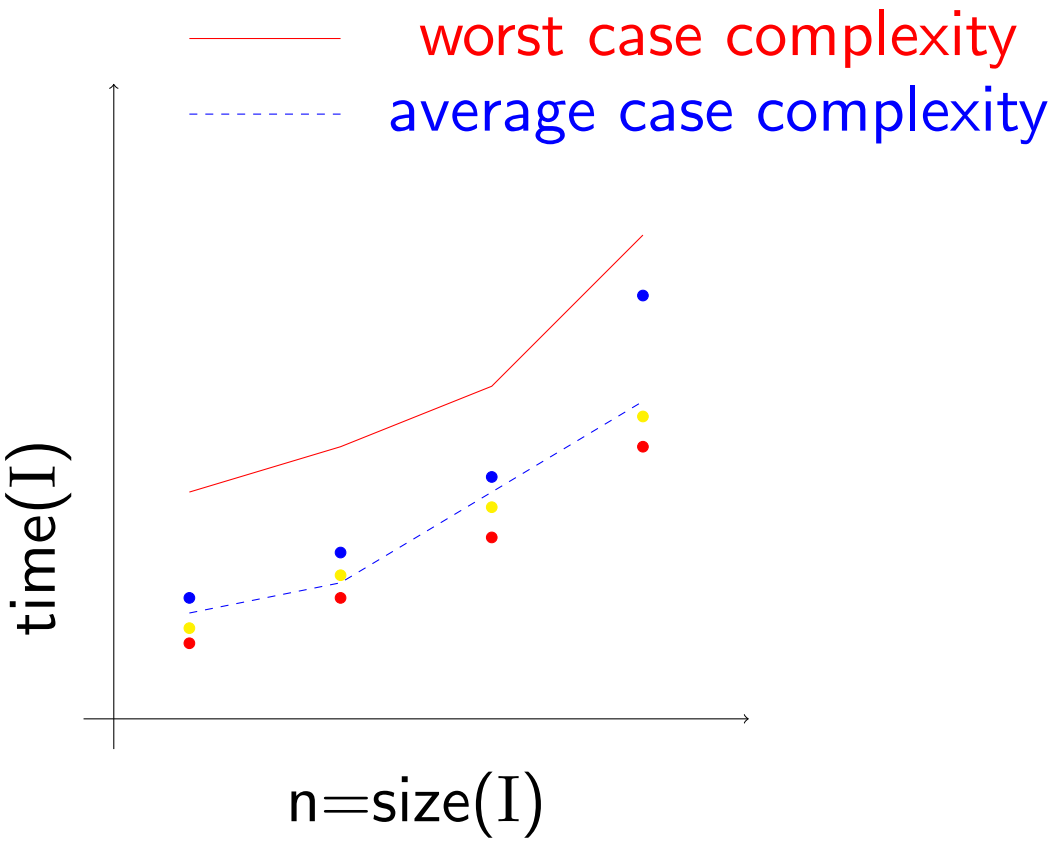
Time Complexity

- The **worst-case time complexity** of an algorithm is a function $T : \mathbb{N} \to \mathbb{N}$ where
  - $T(n)$ is the maximum number of elementary operations the algorithm uses on inputs of size n
  - the **input size** is problem-dependent
    - e.g. number of cities, length of an array, number of nodes/edges in a graph, . . .

Time Complexity

- The **worst-case time complexity** of an algorithm is a function $T : \mathbb{N} \to \mathbb{N}$ where
  - $T(n)$ is the maximum number of elementary operations the algorithm uses on inputs of size n
  - the **input size** is problem-dependent
    - e.g. number of cities, length of an array, number of nodes/edges in a graph, . . .
- The **average-case complexity** is similar: replace maximum by average in the above

Average versus Worst Case



Example: Finding the minimum element in an array

- a simple algorithm:

```
int m = a[0];
int i = 1;
while (i<n) {
  m = min(m,a[i]);
  i++;
}
```

- take as elementary operations assignments and comparisons (min)
- for each input array of size n, the algorithm takes

$$3 + 4 * (n - 1) = 4n - 1 \text{ elementary operations}$$

- hence both the average and the worst-case complexity is $4n - 1$!

Example: Finding the minimum element in an array

- a simple algorithm:

```
int m = a[0];
int i = 1;
while (i<n) {
  m = min(m,a[i]);
  i++;
}
```

- take as elementary operations assignments and comparisons (min)
- for each input array of size n, the algorithm takes

$$3 + 4 * (n - 1) = 4n - 1 \text{ elementary operations}$$

- hence both the average and the worst-case complexity is $4n - 1$!
- how does the complexity change if the assignment is replaced by

```
if (a[i]<m) m = a[i];      ?
```

Example: Finding the minimum element in an array

- a simple algorithm:

```
int m = a[0];
int i = 1;
while (i<n) {
  m = min(m,a[i]);
  i++;
}
```

- take as elementary operations assignments and comparisons (min)
- for each input array of size n, the algorithm takes

$$3 + 4 * (n - 1) = 4n - 1 \text{ elementary operations}$$

- hence both the average and the worst-case complexity is $4n - 1$!
- how does the complexity change if the assignment is replaced by

```
if (a[i]<m) m = a[i];      ?
```

- the number of elementary operations now depends on the input

Example: Finding the minimum element in an array

- a simple algorithm:

```
int m = a[0];
int i = 1;
while (i<n) {
  m = min(m,a[i]);
  i++;
}
```

- take as elementary operations assignments and comparisons (min)
- for each input array of size n, the algorithm takes

$$3 + 4 * (n - 1) = 4n - 1 \text{ elementary operations}$$

- hence both the average and the worst-case complexity is $4n - 1$!
- how does the complexity change if the assignment is replaced by

```
if (a[i]<m) m = a[i];      ?
```

- the number of elementary operations now depends on the input
- but on average it is still **roughly proportional** to n!

Rate of Growth of the Time Complexity

- Suppose we have an algorithm that takes $4n^2 + 12n + 199$ elementary operations in the worst case

Rate of Growth of the Time Complexity

- Suppose we have an algorithm that takes $4n^2 + 12n + 199$ elementary operations in the worst case
- Look at the **asymptotic leading functional behaviour**

Rate of Growth of the Time Complexity

- Suppose we have an algorithm that takes $4n^2 + 12n + 199$ elementary operations in the worst case
- Look at the **asymptotic leading functional behaviour**
  - ⋆ **asymptotic**: what happens when n becomes very large

Rate of Growth of the Time Complexity

- Suppose we have an algorithm that takes $4n^2 + 12n + 199$ elementary operations in the worst case
- Look at the **asymptotic leading functional behaviour**
  - ⋆ **asymptotic**: what happens when n becomes very large
  - ⋆ **leading**: ignore the $12n + 199$ part as it is dominated by $4n^2$ (i.e. for large enough n we have $4n^2 \gg 12n + 199$)

Rate of Growth of the Time Complexity

- Suppose we have an algorithm that takes $4n^2 + 12n + 199$ elementary operations in the worst case
- Look at the **asymptotic leading functional behaviour**
  - ⋆ **asymptotic**: what happens when n becomes very large
  - ⋆ **leading**: ignore the $12n + 199$ part as it is dominated by $4n^2$ (i.e. for large enough n we have $4n^2 \gg 12n + 199$)
  - ⋆ **functional behaviour**: ignore the constant 4

Rate of Growth of the Time Complexity

- Suppose we have an algorithm that takes $4n^2 + 12n + 199$ elementary operations in the worst case
- Look at the **asymptotic leading functional behaviour**
  - ⋆ **asymptotic**: what happens when n becomes very large
  - ⋆ **leading**: ignore the $12n + 199$ part as it is dominated by $4n^2$ (i.e. for large enough n we have $4n^2 \gg 12n + 199$)
  - ⋆ **functional behaviour**: ignore the constant 4
- We call this an order n^2, or quadratic time, algorithm

Rate of Growth of the Time Complexity

- Suppose we have an algorithm that takes $4n^2 + 12n + 199$ elementary operations in the worst case
- Look at the **asymptotic leading functional behaviour**
  - ⋆ **asymptotic**: what happens when n becomes very large
  - ⋆ **leading**: ignore the $12n + 199$ part as it is dominated by $4n^2$ (i.e. for large enough n we have $4n^2 \gg 12n + 199$)
  - ⋆ **functional behaviour**: ignore the constant 4
- We call this an order n^2, or quadratic time, algorithm
- We can write this in **Big-Theta** notation as $\Theta(n^2)$

Rate of Growth of the Time Complexity

- Suppose we have an algorithm that takes $4n^2 + 12n + 199$ elementary operations in the worst case
- Look at the **asymptotic leading functional behaviour**
  - ⋆ **asymptotic**: what happens when n becomes very large
  - ⋆ **leading**: ignore the $12n + 199$ part as it is dominated by $4n^2$ (i.e. for large enough n we have $4n^2 \gg 12n + 199$)
  - ⋆ **functional behaviour**: ignore the constant 4
- We call this an order n^2, or quadratic time, algorithm
- We can write this in **Big-Theta** notation as $\Theta(n^2)$
  - ⋆ "$4n^2 + 12n + 199$ is $\Theta(n^2)$" means that the function $4n^2 + 12n + 199$ grows at the same rate as the function n^2

Big-Theta Notation and its Advantages

- Measures the rate of growth of functions
  - ⋆ If $f(n)$ is $\Theta(g(n))$, then $f(n) \approx cg(n)$ for $n \gg 1$, for some constant $c > 0$
- We can use it to express the rate of growth of the time complexity of an algorithm

Big-Theta Notation and its Advantages

- Measures the rate of growth of functions
  - ⋆ If $f(n)$ is $\Theta(g(n))$, then $f(n) \approx cg(n)$ for $n \gg 1$, for some constant $c > 0$
- We can use it to express the rate of growth of the time complexity of an algorithm
- We can estimate algorithm run times on large inputs by measuring run time on a small input
  - ⋆ If I have a $\Theta(n^2)$ average-case algorithm
  - ⋆ It takes x seconds on average on input of size 100
  - ⋆ It will take about $\frac{x \times n^2}{100^2}$ seconds on average on an input of size n

Disadvantage of Big-Theta Notation

- Can not compare algorithms whose time complexities have the same rate of growth

Disadvantage of Big-Theta Notation

- Can not compare algorithms whose time complexities have the same rate of growth
- For small inputs Big-Theta can be misleading. E.g.
  - ⋆ algorithm A takes $n^3 + 2n^2 + 5$ operations
  - ⋆ algorithm B takes $20n^2 + 100$ operations
  - ⋆ algorithm A is $\Theta(n^3)$ and algorithm B is $\Theta(n^2)$
  - ⋆ algorithm A is faster than algorithm B for $n < 18$

Disadvantage of Big-Theta Notation

- Can not compare algorithms whose time complexities have the same rate of growth
- For small inputs Big-Theta can be misleading. E.g.
  - ⋆ algorithm A takes $n^3 + 2n^2 + 5$ operations
  - ⋆ algorithm B takes $20n^2 + 100$ operations
  - ⋆ algorithm A is $\Theta(n^3)$ and algorithm B is $\Theta(n^2)$
  - ⋆ algorithm A is faster than algorithm B for $n < 18$

  but who cares?

More on Ignoring the Constant

- Does an algorithm that uses $4n^2$ operations run faster than one that uses $7n^2$ operations?

More on Ignoring the Constant

- Does an algorithm that uses $4n^2$ operations run faster than one that uses $7n^2$ operations?
- Answer depends on the operations which might depend on the programming language or the machine architecture

More on Ignoring the Constant

- Does an algorithm that uses $4n^2$ operations run faster than one that uses $7n^2$ operations?
- Answer depends on the operations which might depend on the programming language or the machine architecture
- However asymptotically (i.e. for sufficiently large n) an order $n\log(n)$ algorithm will always run faster than an order n^2 algorithm

More on Ignoring the Constant

- Does an algorithm that uses $4n^2$ operations run faster than one that uses $7n^2$ operations?
- Answer depends on the operations which might depend on the programming language or the machine architecture
- However asymptotically (i.e. for sufficiently large n) an order $n\log(n)$ algorithm will always run faster than an order n^2 algorithm **even when they are run on different machines!**

More on Ignoring the Constant

- Does an algorithm that uses $4n^2$ operations run faster than one that uses $7n^2$ operations?
- Answer depends on the operations which might depend on the programming language or the machine architecture
- However asymptotically (i.e. for sufficiently large n) an order $n\log(n)$ algorithm will always run faster than an order n^2 algorithm **even when they are run on different machines!**
- The constant is important in practice (if there are two algorithms A and B that are both $n\log(n)$, but algorithm A runs twice as fast as algorithm B, which one should you use?)

More on Ignoring the Constant

- Does an algorithm that uses $4n^2$ operations run faster than one that uses $7n^2$ operations?
- Answer depends on the operations which might depend on the programming language or the machine architecture
- However asymptotically (i.e. for sufficiently large n) an order $n\log(n)$ algorithm will always run faster than an order n^2 algorithm **even when they are run on different machines!**
- The constant is important in practice (if there are two algorithms A and B that are both $n\log(n)$, but algorithm A runs twice as fast as algorithm B, which one should you use?)
- Nevertheless, ignoring the constant is often essential to make analysis of algorithms doable

Counting Operations

- The time complexity of an algorithm is often easy to calculate

Counting Operations

- The time complexity of an algorithm is often easy to calculate

- a $\Theta(n)$ algorithm

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
}
// clean up
```

Counting Operations

- The time complexity of an algorithm is often easy to calculate

- a $\Theta(n)$ algorithm

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
}
// clean up
```

- a $\Theta(n^2)$ algorithm

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  for (int j=0; j<n; j++) {
    // do other stuff
  }
}
// clean up
```

Counting Operations

- The time complexity of an algorithm is often easy to calculate

- a $\Theta(n)$ algorithm

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
}
// clean up
```

- a $\Theta(n^2)$ algorithm

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  for (int j=0; j<n; j++) {
    // do other stuff
  }
}
// clean up
```

- For the above algorithms, the average-case and worst-case time complexities coincide – **Why?**

Not So Sure

- In some cases the time complexity is harder to compute

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  if (/* some condition */) {
    for (int j=0; j<n; j++) {
      // do other stuff
    }
  }
}
// clean up
```

Not So Sure

- In some cases the time complexity is harder to compute

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  if (/* some condition */) {
    for (int j=0; j<n; j++) {
      // do other stuff
    }
  }
}
// clean up
```

- Average time complexity now depends on the `if` statement

Not So Sure

- In some cases the time complexity is harder to compute

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  if (/* some condition */) {
    for (int j=0; j<n; j++) {
      // do other stuff
    }
  }
}
// clean up
```

- Average time complexity now depends on the `if` statement
  - ⋆ If the condition holds almost always, the average-case complexity is $\Theta(n^2)$

Not So Sure

- In some cases the time complexity is harder to compute

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  if (/* some condition */) {
    for (int j=0; j<n; j++) {
      // do other stuff
    }
  }
}
// clean up
```

- Average time complexity now depends on the `if` statement
  - ⋆ If the condition holds almost always, the average-case complexity is $\Theta(n^2)$
  - ⋆ If the condition is rarely true, the average-case complexity is $\Theta(n)$

Not So Sure

- In some cases the time complexity is harder to compute

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  if (/* some condition */) {
    for (int j=0; j<n; j++) {
      // do other stuff
    }
  }
}
// clean up
```

- Average time complexity now depends on the `if` statement
  - ⋆ If the condition holds almost always, the average-case complexity is $\Theta(n^2)$
  - ⋆ If the condition is rarely true, the average-case complexity is $\Theta(n)$
- Worst-case complexity still $\Theta(n^2)$ if condition is not always false!

Not So Sure

- In some cases the time complexity is harder to compute

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  if (/* some condition */) {
    for (int j=0; j<n; j++) {
      // do other stuff
    }
  }
}
// clean up
```

- Average time complexity now depends on the `if` statement
  - ⋆ If the condition holds almost always, the average-case complexity is $\Theta(n^2)$
  - ⋆ If the condition is rarely true, the average-case complexity is $\Theta(n)$
- Worst-case complexity still $\Theta(n^2)$ if condition is not always false!
- In general, both complexities depend on the shape of the condition!

Bounds

- To avoid having to think really hard we can look for **upper** and **lower** bounds for the rate of growth of the time complexity

Bounds

- To avoid having to think really hard we can look for **upper** and **lower** bounds for the rate of growth of the time complexity
- We can use **big-O** notation to give an **upper bound** for the rate of growth of the time complexity
  - ⋆ The worst-case complexity of the previous algorithm is $O(n^2)$
  - ⋆ I.e. it runs in **no more than** order n^2 operations in the worst case
  - ⋆ The same holds for the average-case complexity

Bounds

- To avoid having to think really hard we can look for **upper** and **lower** bounds for the rate of growth of the time complexity
- We can use **big-O** notation to give an **upper bound** for the rate of growth of the time complexity
  - ⋆ The worst-case complexity of the previous algorithm is $O(n^2)$
  - ⋆ I.e. it runs in **no more than** order n^2 operations in the worst case
  - ⋆ The same holds for the average-case complexity
- We can use **big-Omega** notation to give a **lower bound** for the rate of growth:

Bounds

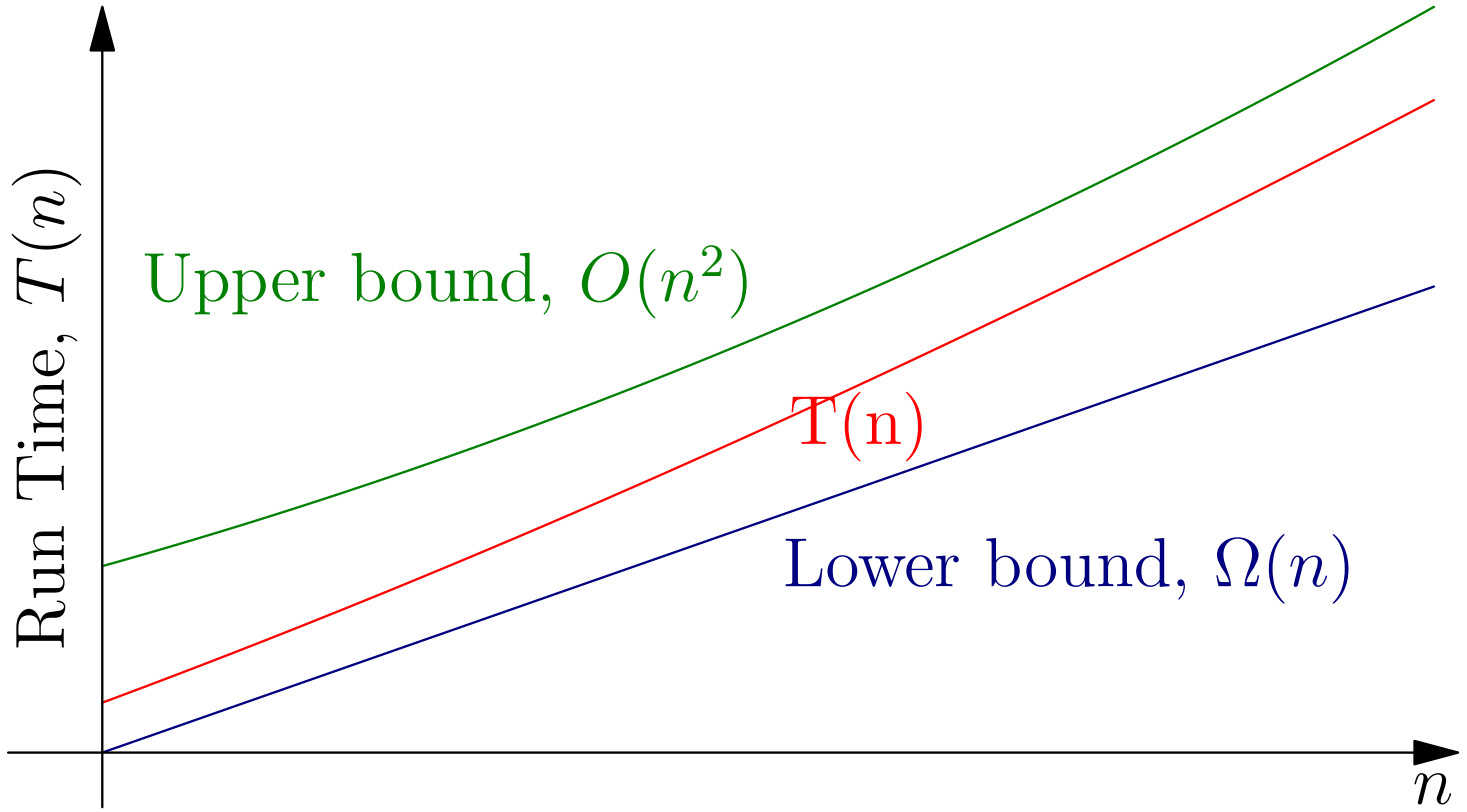
- To avoid having to think really hard we can look for **upper** and **lower** bounds for the rate of growth of the time complexity
- We can use **big-O** notation to give an **upper bound** for the rate of growth of the time complexity
  - ⋆ The worst-case complexity of the previous algorithm is $O(n^2)$
  - ⋆ I.e. it runs in **no more than** order n^2 operations in the worst case
  - ⋆ The same holds for the average-case complexity
- We can use **big-Omega** notation to give a **lower bound** for the rate of growth:
  - ⋆ The previous algorithm is a $\Omega(n)$ average-case algorithm
  - ⋆ I.e. it runs in **no less than** order n operations on average
  - ⋆ And the same holds for the worst case

Bounding Run Time Complexity

- When we are given an algorithm to analyse, we want to find the rate of growth of $T(n)$

Bounding Run Time Complexity

- When we are given an algorithm to analyse, we want to find the rate of growth of $T(n)$
- This may be difficult, however, it is often easy to find bounds



Example: Insertion Sort

- insertion sort algorithm:

```
int i=1;
while (i<n) {
  /* insert a[i] into sorted sequence a[0..i-1] */
  int e = a[i];
  int j = i-1;
  while (j >=0 && a[i]<a[j]) a[j+1] = a[j];
  a[j+1]=e;
}
```

Example: Insertion Sort

- insertion sort algorithm:

```
int i=1;
while (i<n) {
  /* insert a[i] into sorted sequence a[0..i-1] */
  int e = a[i];
  int j = i-1;
  while (j >=0 && a[i]<a[j]) a[j+1] = a[j];
  a[j+1]=e;
}
```

- its **worst-case complexity** is $\Theta(n^2)$ — why?

Example: Insertion Sort

- insertion sort algorithm:

```
int i=1;
while (i<n) {
  /* insert a[i] into sorted sequence a[0..i-1] */
  int e = a[i];
  int j = i-1;
  while (j >=0 && a[i]<a[j]) a[j+1] = a[j];
  a[j+1]=e;
}
```

- its **worst-case complexity** is $\Theta(n^2)$ — why?
- its **average-case complexity** is $O(n^2)$
  - ⋆ It is not so easy to find the exact rate of growth of the average case complexity – more on this later in the module.

Precise Definitions

- $f(n)$ **is** $O(g(n))$ if

  $f(n) \leq cg(n)$ for $n \geq N$, where $c > 0$ and $N \in \mathbb{N}$ are constants

Precise Definitions

- $f(n)$ **is** $O(g(n))$ if

  $f(n) \leq cg(n)$ for $n \geq N$, where $c > 0$ and $N \in \mathbb{N}$ are constants

- $f(n)$ **is** $\Omega(g(n))$ if

  $f(n) \geq cg(n)$ for $n \geq N$, where $c > 0$ and $N \in \mathbb{N}$ are constants

Precise Definitions

- $f(n)$ **is** $O(g(n))$ if

  $f(n) \leq cg(n)$ for $n \geq N$, where $c > 0$ and $N \in \mathbb{N}$ are constants

- $f(n)$ **is** $\Omega(g(n))$ if

  $f(n) \geq cg(n)$ for $n \geq N$, where $c > 0$ and $N \in \mathbb{N}$ are constants

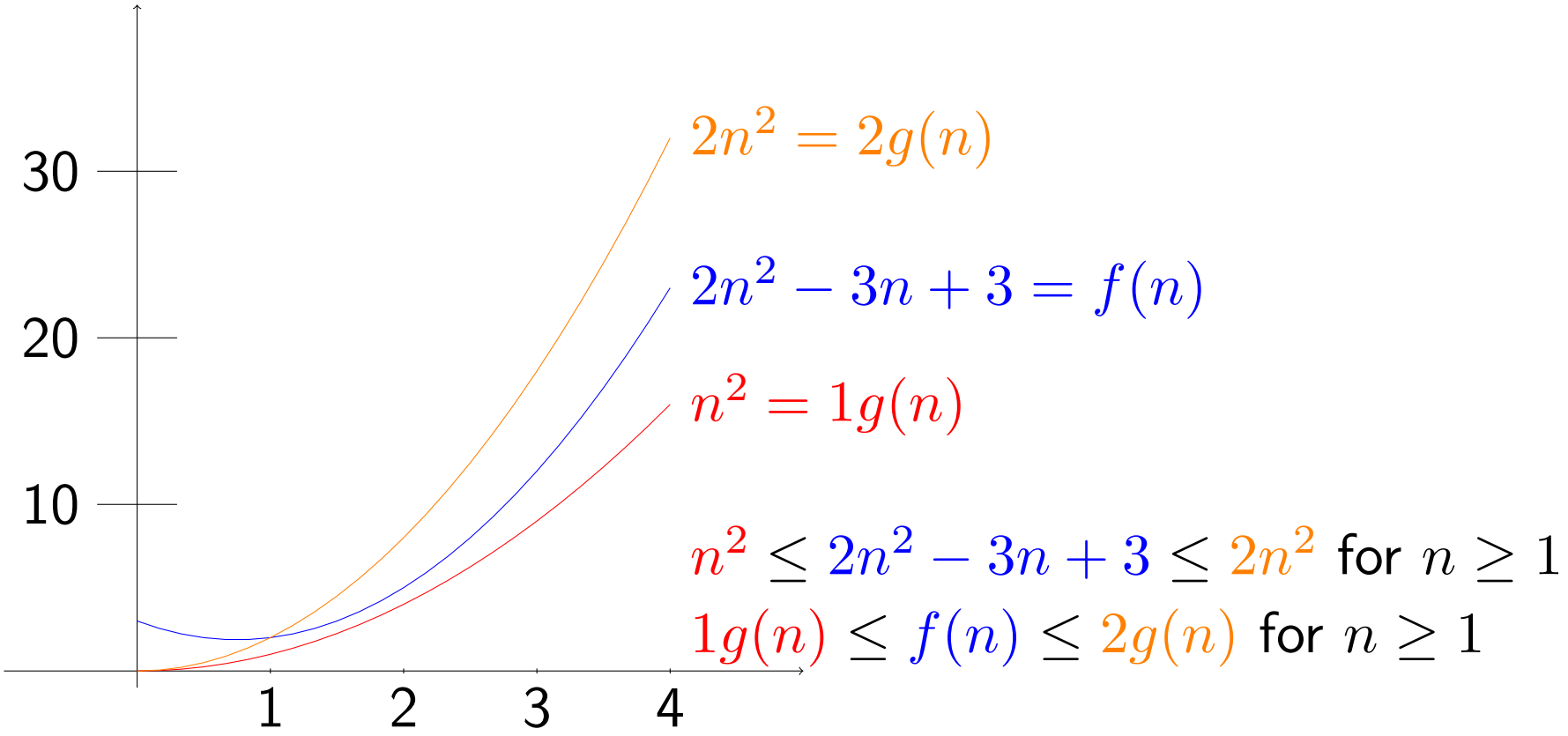
- $f(n)$ **is** $\Theta(g(n))$ if

$$f(n) = O(g(n)) \quad \text{and} \quad f(n) = \Omega(g(n))$$

  i.e. the lower bound is identical to the upper bound

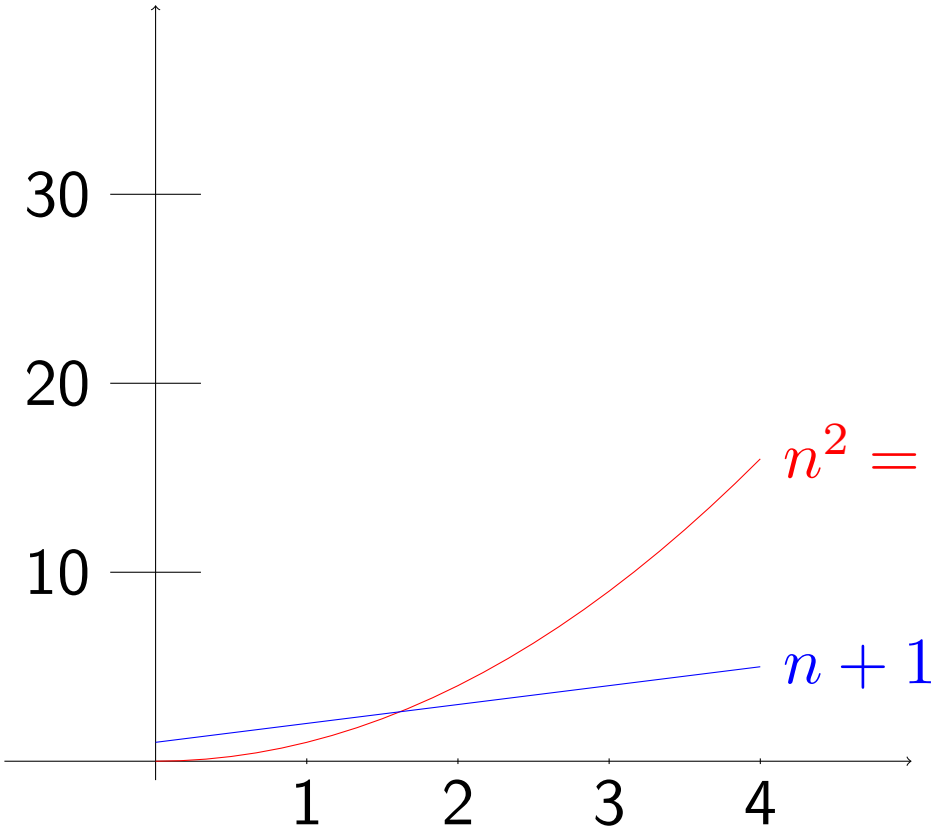
Big-Θ Notation

- $2n^2 - 3n + 3$ is $\Theta(n^2)$:



Big-O Notation

- $n + 1$ is $O(n^2)$:



$n + 1 \leq n^2$ for $n \geq 2$

$f(n) \leq 1g(n)$ for $n \geq 2$

Use and Misuse

- Note: big-O notation is most commonly used.
- Often people say they have a $O(n^2)$ when in fact they have a $\Theta(n^2)$ algorithm (a stronger statement).
- Note that an $O(n^2)$ algorithm is also a $O(n^3)$ algorithm
- A $O(n^2)$ algorithm **may not** be faster than a $O(n^3)$ algorithm when n becomes larger.
- A $\Theta(n^2)$ algorithm **will** be faster than a $\Theta(n^3)$ algorithm when n becomes larger.

Outline

1. TSP
2. Sorting
3. Time Complexity, Big Theta, Big O
4. **Measuring Complexity**

Empirical Estimates

- If $T(n) = a\,n^2 + b\,n + c$
  - ⋆ If $n \gg 1$ then $a\,n^2 \gg b\,n + c$ so $T(n) \approx a\,n^2$

Empirical Estimates

- If $T(n) = a\,n^2 + b\,n + c$
  - ⋆ If $n \gg 1$ then $a\,n^2 \gg b\,n + c$ so $T(n) \approx a\,n^2$
  - ⋆ $\log\big(T(n)\big) \approx 2\,\log(n) + \log(a)$

Empirical Estimates

- If $T(n) = a\,n^2 + b\,n + c$
  - ⋆ If $n \gg 1$ then $a\,n^2 \gg b\,n + c$ so $T(n) \approx a\,n^2$
  - ⋆ $\log\big(T(n)\big) \approx 2\,\log(n) + \log(a)$
  - ⋆ If we plot $\log\big(T(n)\big)$ versus $\log(n)$ we should get a graph with gradient 2 and intersection $\log(a)$

Empirical Estimates

- If $T(n) = a\,n^2 + b\,n + c$
  - ⋆ If $n \gg 1$ then $a\,n^2 \gg b\,n + c$ so $T(n) \approx a\,n^2$
  - ⋆ $\log\big(T(n)\big) \approx 2\,\log(n) + \log(a)$
  - ⋆ If we plot $\log\big(T(n)\big)$ versus $\log(n)$ we should get a graph with gradient 2 and intersection $\log(a)$
  - ⋆ We can use any logarithm ($\log_{10}$, $\ln$, $\log_2$)

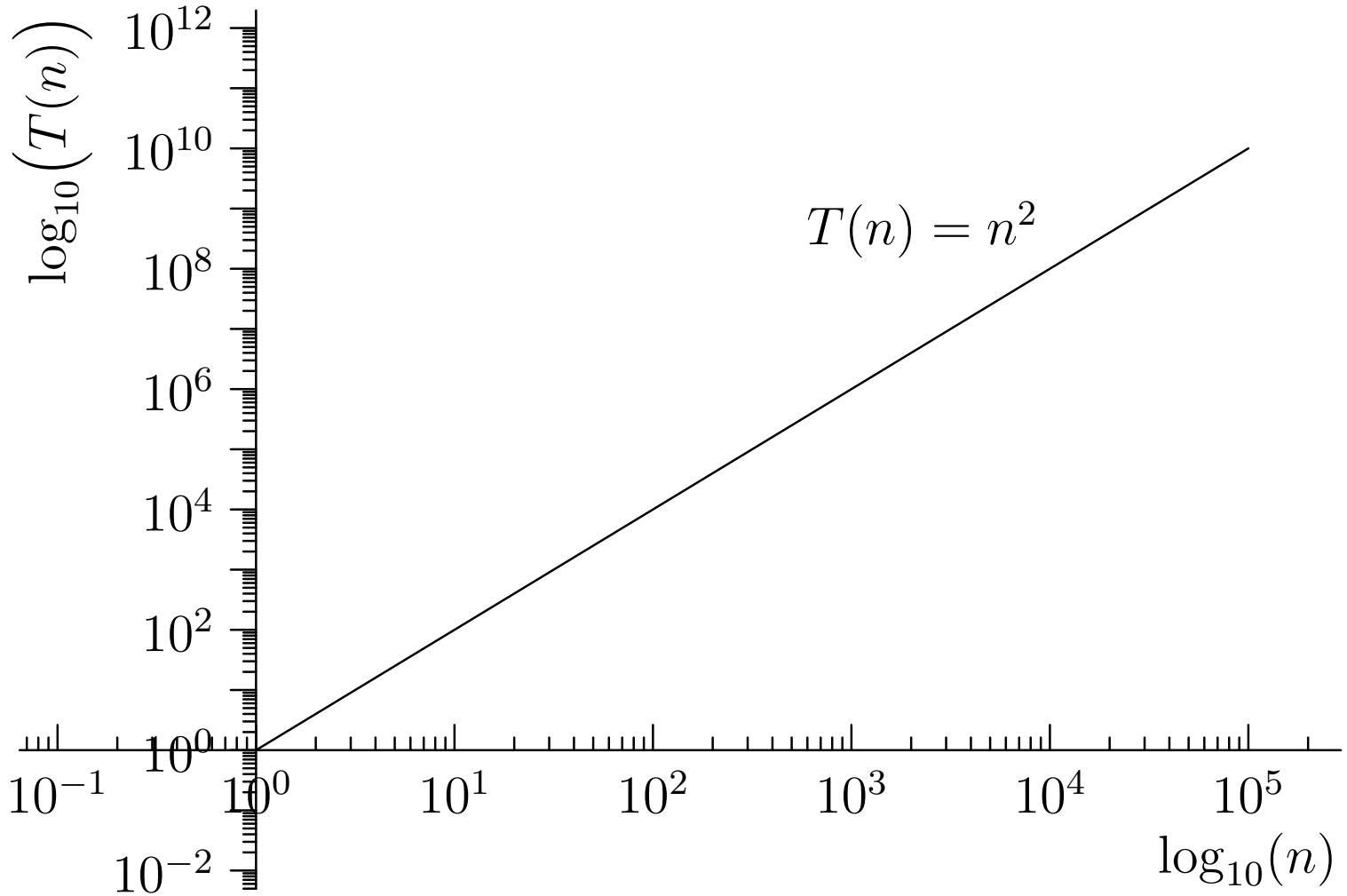
Empirical Estimates

- If $T(n) = a\,n^2 + b\,n + c$
  - ⋆ If $n \gg 1$ then $a\,n^2 \gg b\,n + c$ so $T(n) \approx a\,n^2$
  - ⋆ $\log\big(T(n)\big) \approx 2\,\log(n) + \log(a)$
  - ⋆ If we plot $\log\big(T(n)\big)$ versus $\log(n)$ we should get a graph with gradient 2 and intersection $\log(a)$
  - ⋆ We can use any logarithm ($\log_{10}$, $\ln$, $\log_2$)
- If $T(n) \approx a\,n^c$
  - ⋆ $\log\big(T(n)\big) \approx c\,\log(n) + \log(a)$

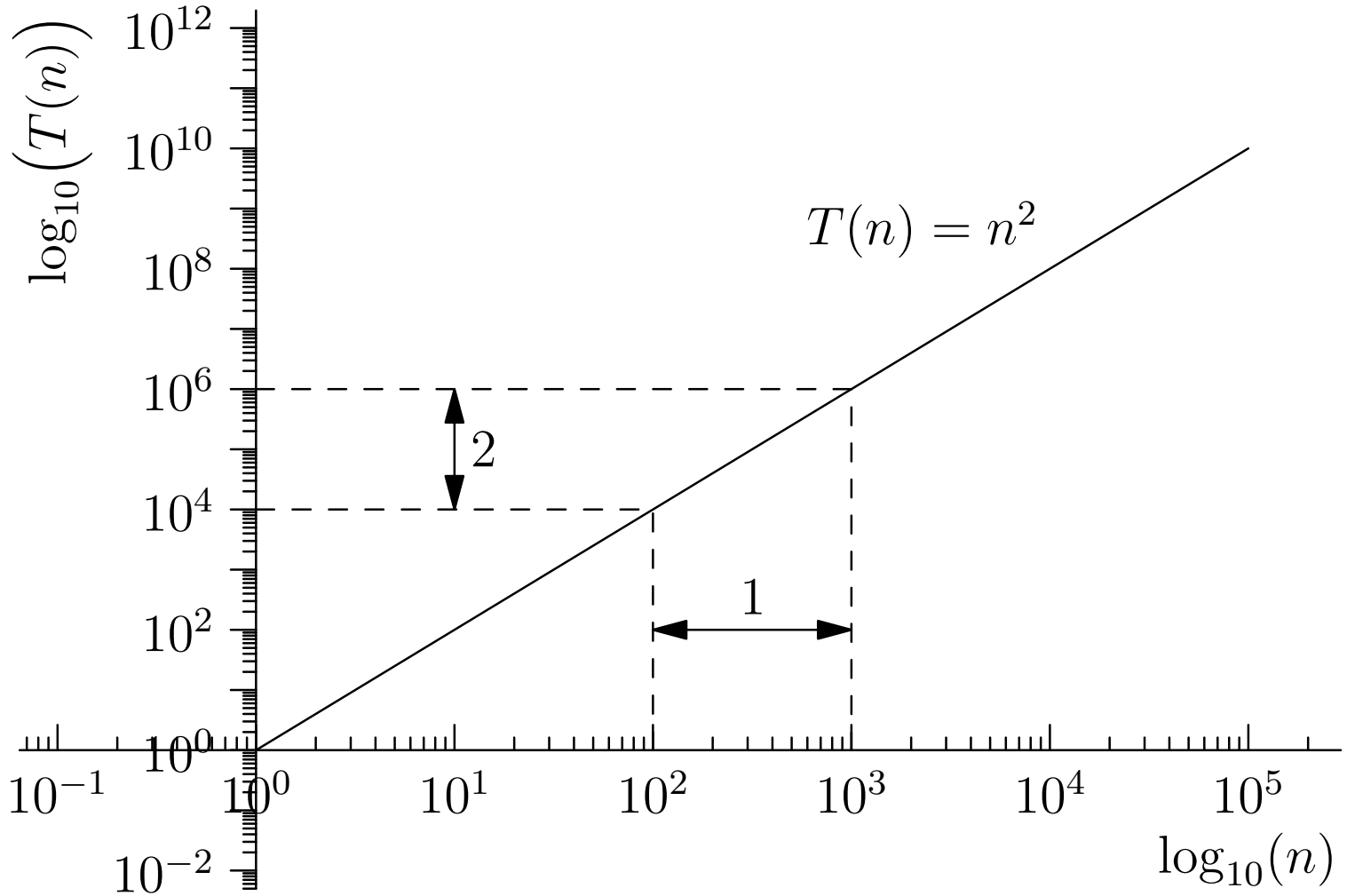
Empirical Estimates

- If $T(n) = a\,n^2 + b\,n + c$
  - ⋆ If $n \gg 1$ then $a\,n^2 \gg b\,n + c$ so $T(n) \approx a\,n^2$
  - ⋆ $\log\big(T(n)\big) \approx 2\,\log(n) + \log(a)$
  - ⋆ If we plot $\log\big(T(n)\big)$ versus $\log(n)$ we should get a graph with gradient 2 and intersection $\log(a)$
  - ⋆ We can use any logarithm ($\log_{10}$, $\ln$, $\log_2$)
- If $T(n) \approx a\,n^c$
  - ⋆ $\log\big(T(n)\big) \approx c\,\log(n) + \log(a)$
- The gradient tells us the rate of growth of the time complexity

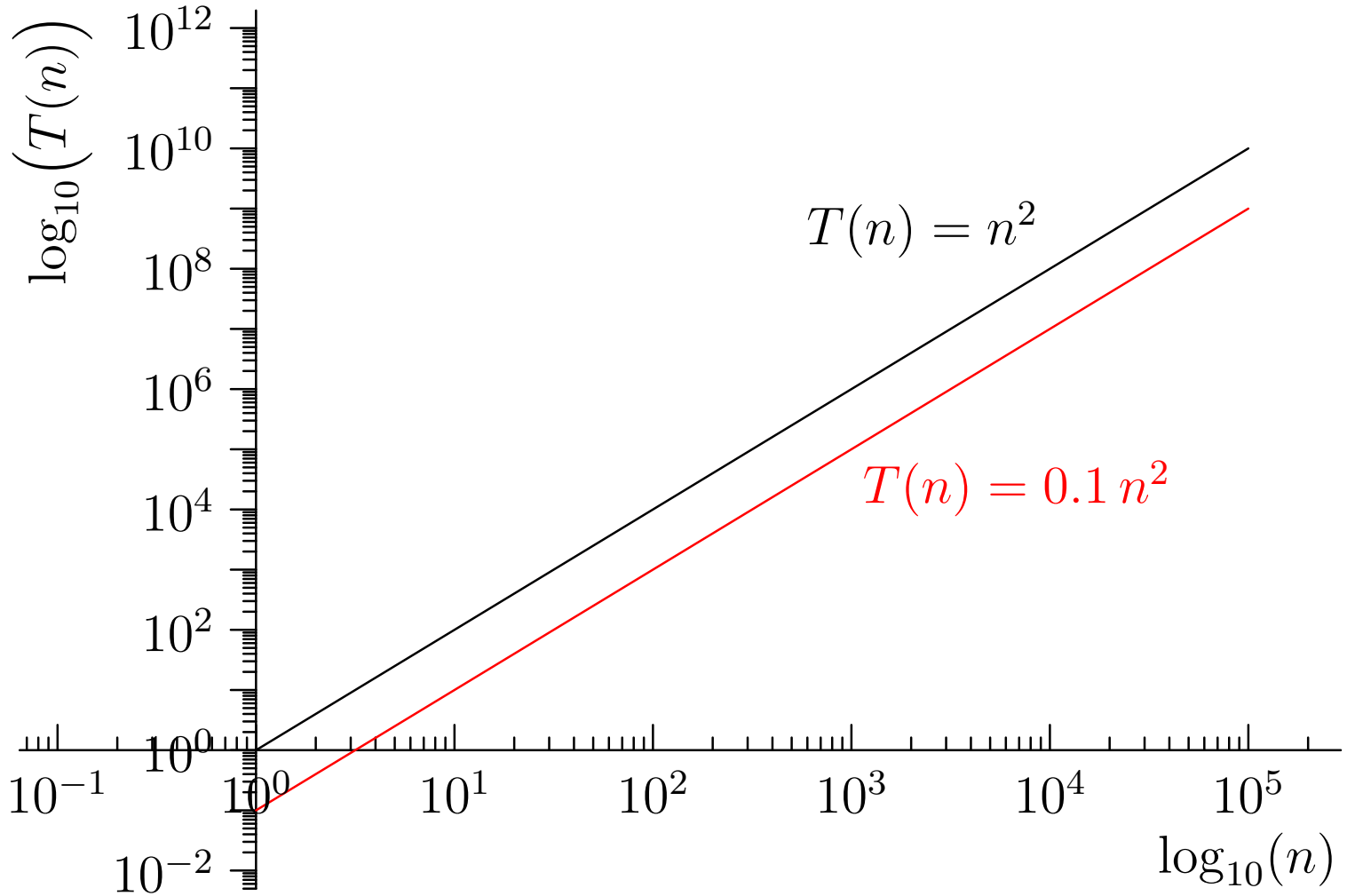
Log-Log Plots



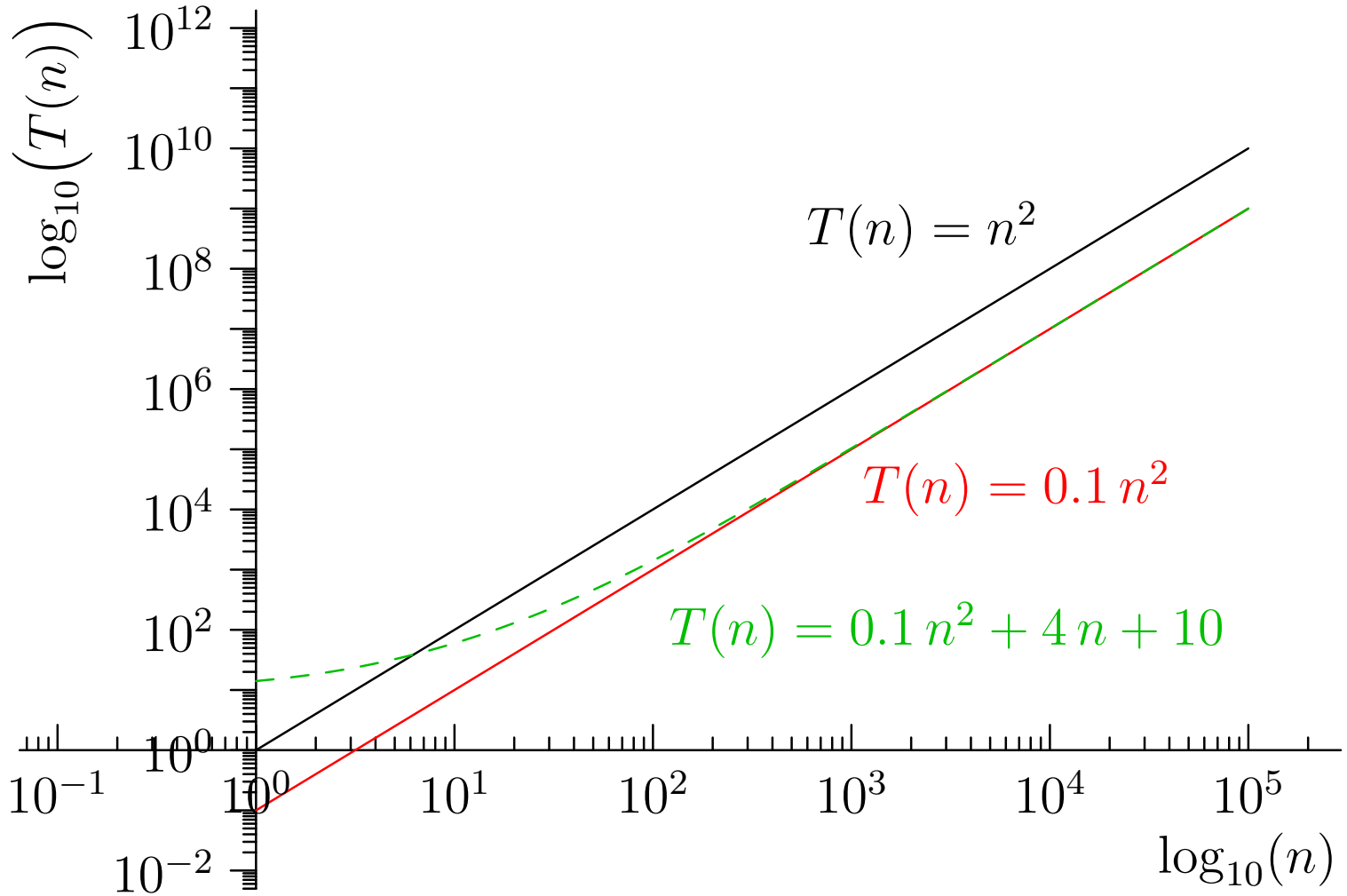
Log-Log Plots



Log-Log Plots



Log-Log Plots



Exponential Time Complexity

- The time complexity of some algorithms is larger than any polynomial

Exponential Time Complexity

- The time complexity of some algorithms is larger than any polynomial
- For example:

$$T(n) = a\,\mathrm{e}^{c\,n}$$

(This is an example of **exponential time** complexity.)

Exponential Time Complexity

- The time complexity of some algorithms is larger than any polynomial
- For example:

$$T(n) = a\,\mathrm{e}^{c\,n}$$

(This is an example of **exponential time** complexity.)

- Taking logarithms (with base e)

$$\log\big(T(n)\big) \approx c\,n + \log(a)$$

Exponential Time Complexity

- The time complexity of some algorithms is larger than any polynomial
- For example:

$$T(n) = a\,\mathrm{e}^{c\,n}$$

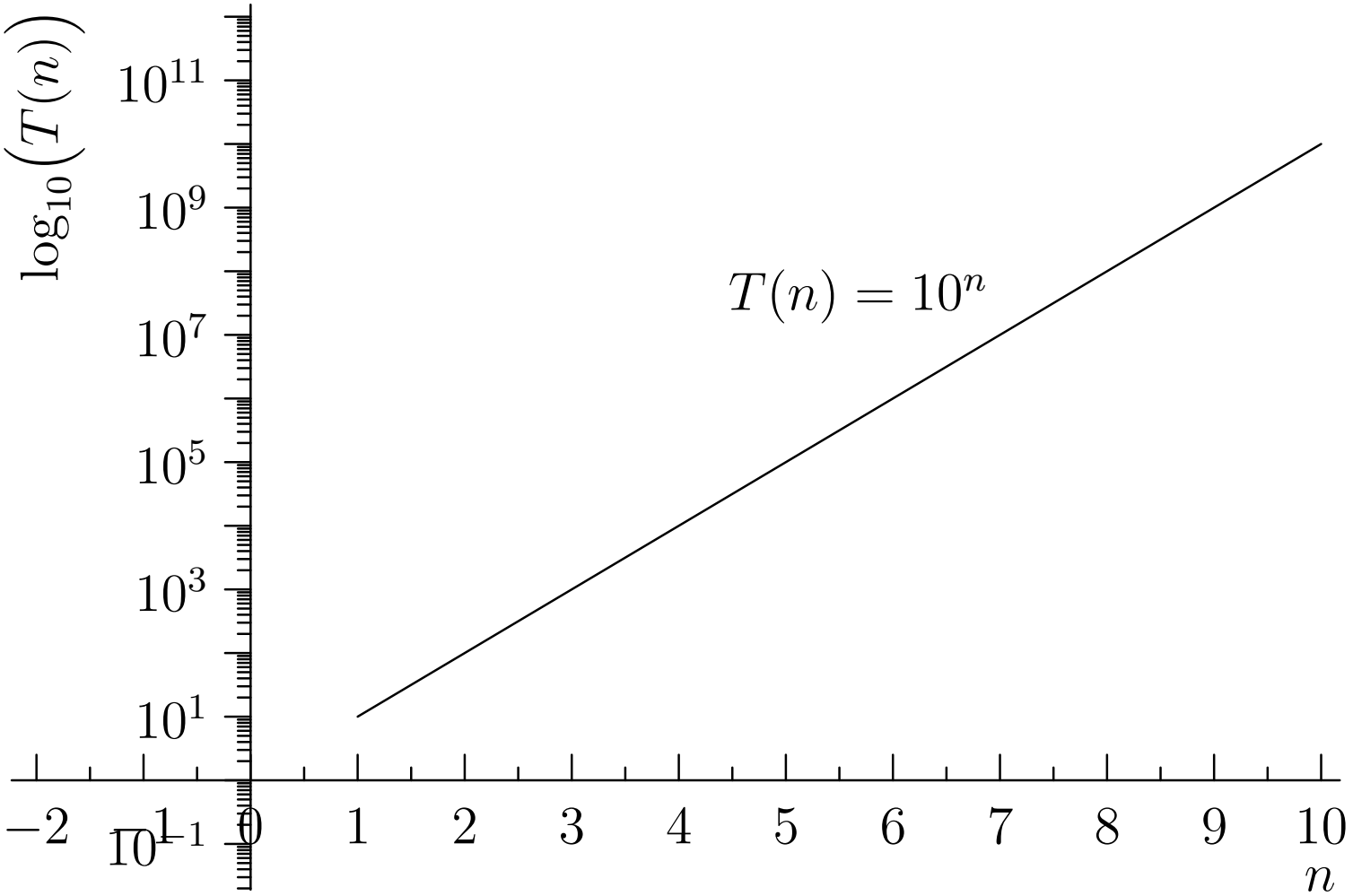
(This is an example of **exponential time** complexity.)

- Taking logarithms (with base e)

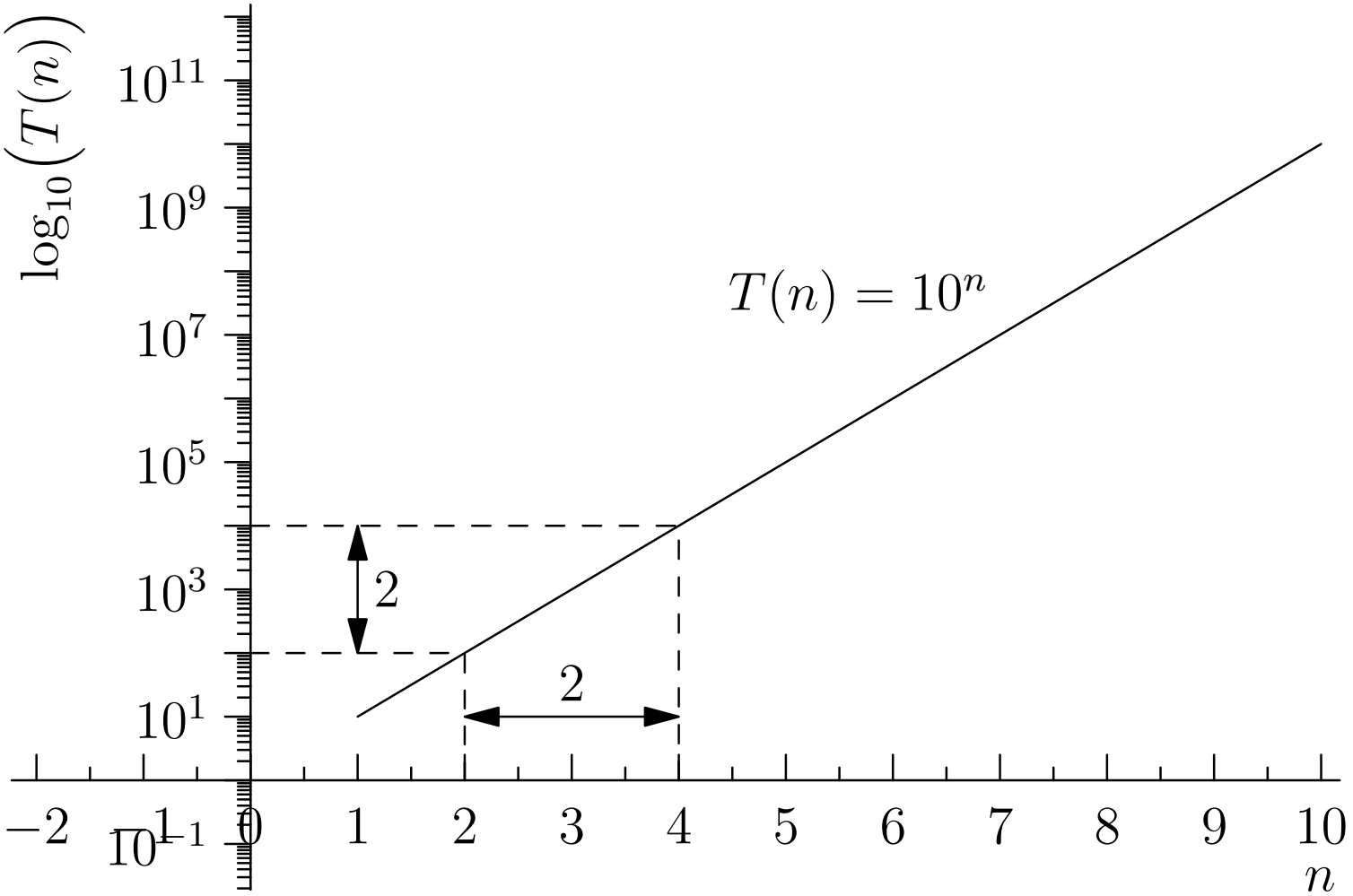
$$\log\big(T(n)\big) \approx c\,n + \log(a)$$

- Thus, plotting $\log\big(T(n)\big)$ versus n gives a line with gradient c and intersection $\log(a)$

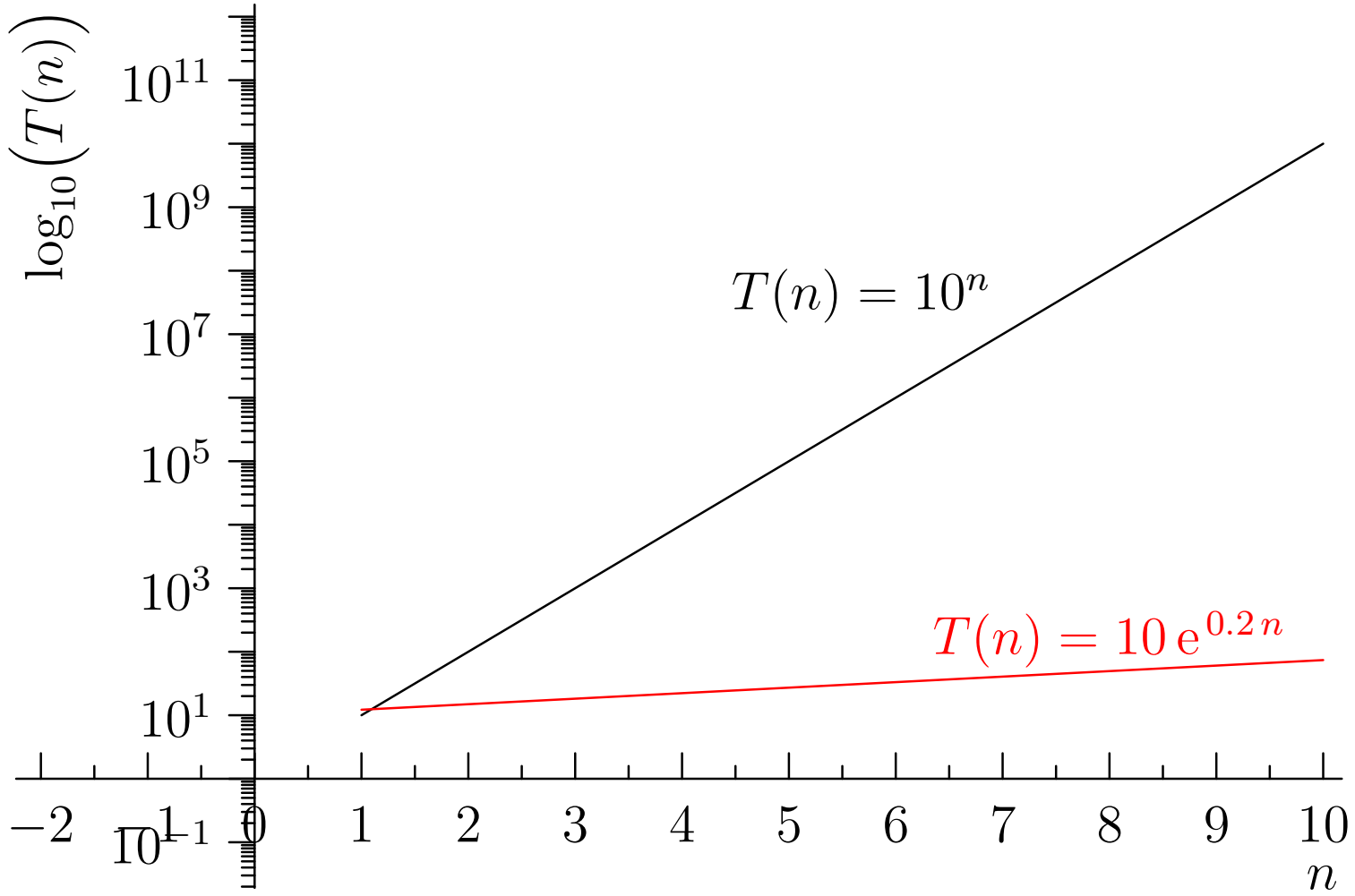
Log-Linear Plots



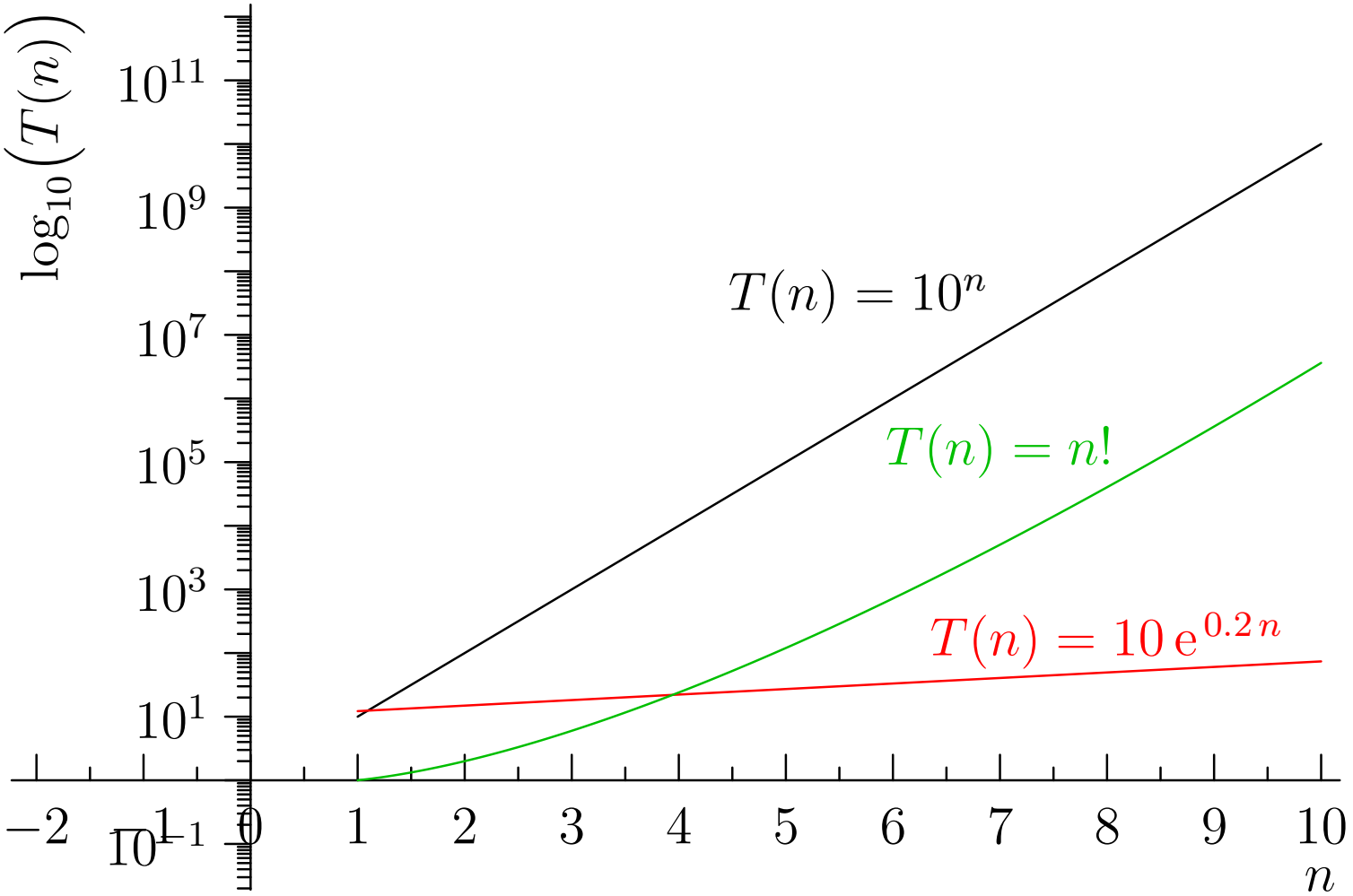
Log-Linear Plots



Log-Linear Plots



Log-Linear Plots



Lessons to Learn

- Run times / computational time complexities matter
- Choosing an algorithm with the best time complexity is important
- Understand the meaning of big-Theta, big-O and big-Omega
- Know how to estimate time complexity for simple algorithms (loop counting)
- Know how to empirically measure time complexity