Further Mathematics and Algorithms

Lesson 2: *Know How Long A Program Takes*

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

Outline

1. **TSP**

2. Sorting

3. Big O

Travelling Salesperson Problem
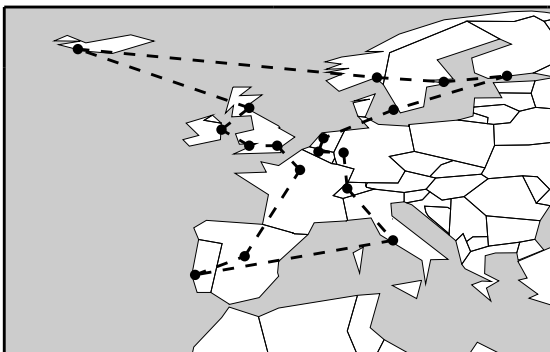
- Given a set of cities

- A table of distances between cities

- Find the shortest tour which goes through each city 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 every path 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?**
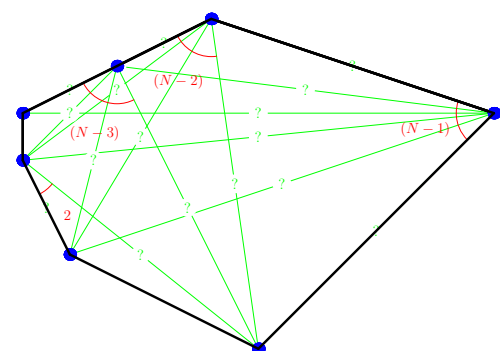

How Many Possible Tours Are There?

- For 100 cities how many possible tours are there?

- It doesn't matter where we start

- Starting from Berlin there are 99 cites we can try next
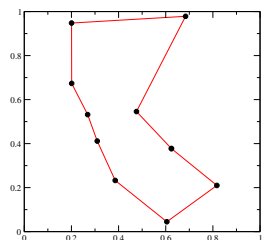
Counting Tours

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
- Total number of tours is $99!/2$
- **Any more guesses how long it will take?**

How Big is 99 Factorial?

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

$$99! = 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1$$
$$99! < 99 \cdot 99 \cdot 99 \cdots 99 \cdot 99 = 99^{99}$$

- Lower bound

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

How Long Does It Take?

- For $N > 1$

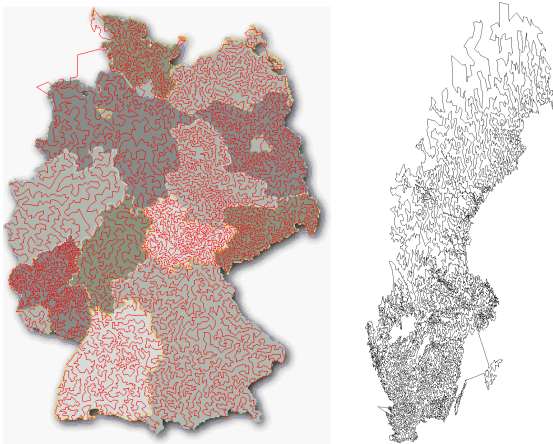$$\left(\frac{N}{2}\right)^{N/2} < N! < N^N$$

- $99!/2 = 4.666 \times 10^{155}$
- How long does it take to search all possible tours?
  - ⋆ We computed about $200\,000$ tours in half a second
  - ⋆ $3.15 \times 10^7 \text{sec} = 1$ year
  - ⋆ Age of Universe ≈ 15 billion years

Answer

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

$$\begin{aligned} 99!/2 = \ & 4666310772197207634084961942813335 0 \\ & 2453579841321908107342964819476087 9 \\ & 9996614957804470731988078259143126 8 \\ & 4896041361187912559260545843200000 0 \\ & 0000000000000000 \end{aligned}$$

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
- If your interested look for the TSP homepage on the web `http://www.math.uwaterloo.ca/tsp/`

Lessons

- Even relatively small problems can take you an astronomical time to solve using simple algorithms
- As a professional programmer you need to have an estimate for how long an algorithm takes—otherwise you can look silly
- 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
  - ⋆ It would still take $10^{39}\times$ the age of the universe
- Smart algorithms can make a much larger difference than fast computers!

Outline

1. TSP
2. **Sorting**
3. Big O

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 returns a sorted array

- Sort is very commonly used algorithm so you care about how long it takes

Empirical Run Times

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, but they exist

- We would like to quantify the performance of an algorithm—how much better is quick sort than insertion sort?
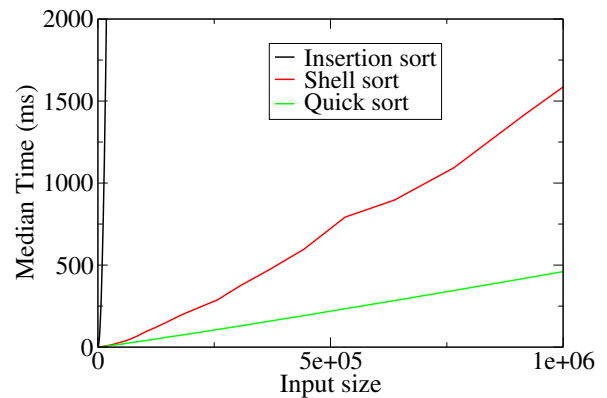
Outline

1. TSP

2. Sorting

3. **Big O**

Estimating Run Times

- We would like to estimate the run times of algorithms

- This depends on the hardware (how fast is your computer)

- We could count number of elementary operations but

  ⋆ different machines have different elementary operations
  ⋆ many algorithms use complex functions such as sqrt (matrix inversion using Cholesky decomposition) or sin and cos (FFT)
  ⋆ would need to count memory accesses which you shouldn't need to think about
  ⋆ code after compiling can be very different from code before compiling

Engineering Solution

- Compute the **asymptotic leading functional behaviour**
- Lets take that statement to pieces
- Suppose we have an algorithm that takes $4n^2 + 12n + 199$ operations (clock cycles)

  ⋆ **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 $\Theta(n^2)$
- This notion of 'run time' is known as **time complexity**

Advantages of Big-Theta Notation

- Doesn't depend on what computer we are running

- Don't need to know how many elementary operations are required for a non-elementary operation

- Can estimate run times by measuring run time on a small problem

  ⋆ If I have a $\Theta(n^2)$ algorithm
  ⋆ It takes x seconds on an input of 100
  ⋆ It will take about $\frac{x \times n^2}{100^2}$ seconds on a problem of size n
    ($T(100) \approx c\,100^2 = x$ therefore $c = x/100^2$
    thus $T(n) = c\,n^2 = x\,n^2/100^2$)

Counting Instructions

- Big-Theta run times are 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
```

Disadvantage with Big-Theta notation

- Can't compare algorithms with the same Big-Theta time complexity

- For small inputs Big-Theta time complexity 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?

- In some cases Big-Theta time complexity is hard to compute

Not So Sure

- Some algorithms are 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
```

- Time complexity now depends on the `if` statement

- If the condition is often satisfied we have a $\Theta(n^2)$ algorithm

- If the condition is true only rarely then we have a $\Theta(n)$ algorithm

Bounds

- To avoid having to think really hard we define upper and lower bounds

- The upper bound we write using **big-O** notation

  - ⋆ The above algorithm is an $O(n^2)$ algorithm
  - ⋆ I.e. it runs in no more than order n^2 operations

- The lower bound we write using **big-Omega** notation

  - ⋆ The above algorithm is a $\Omega(n)$ algorithm
  - ⋆ I.e. it runs in no less than order n operations

Precise Definitions of $O(n)$

- An algorithm that runs in $f(n)$ operations is $O(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c \qquad \text{where } c \text{ is a constant (could be zero)}$$

- E.g.. $f(n) = 3n^2 + 2n + 12$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{3n^2 + 2n + 12}{n^2} = 3 \quad \Rightarrow 3n^2 + 2n + 12 = O(n^2)$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{3n^2 + 2n + 12}{n^3} = 0 \quad \Rightarrow 3n^2 + 2n + 12 = O(n^3)$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{3n^2 + 2n + 12}{n} = \infty \quad \Rightarrow 3n^2 + 2n + 12 \neq O(n)$$

Lower Bound Definition

- An algorithm that runs in $f(n)$ operations is $\Omega(g(n))$ if

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = c \qquad \text{where } c \text{ is a constant (could be zero)}$$

- E.g. $f(n) = 3n^2 + 2n + 12$

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \lim_{n \to \infty} \frac{n^2}{3n^2 + 2n + 12} = \frac{1}{3} \quad \Rightarrow 3n^2 + 2n + 12 = \Omega(n^2)$$

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \lim_{n \to \infty} \frac{n^3}{3n^2 + 2n + 12} = \infty \Rightarrow 3n^2 + 2n + 12 \neq \Omega(n^3)$$

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \lim_{n \to \infty} \frac{n}{3n^2 + 2n + 12} = 0 \quad \Rightarrow 3n^2 + 2n + 12 = \Omega(n)$$

Big-Theta

- An algorithm that runs in $f(n)$ operations is $\Theta(g(n))$ if

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = c \qquad \text{where } c \text{ is a non-zero constant}$$

- That is, $f(n) = \Theta(g(n))$ 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

- Often the most straightforward way of obtaining big-Theta is to show the upper and lower bounds are the same

Use and Misuse

- Note: big-O notation is most commonly used

- often people say they have a $O(n^2)$ when in fact they mean they have a $\Theta(n^2)$ algorithm (a much stronger result)

- Note that an $O(n^2)$ algorithm is also a $O(n^3)$ algorithm

- Strictly 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

Lessons to Learn

- Run times (computational time complexity) matters

- 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)