The supremacy of quantum algorithms: Lesson 1 – Introduction to search algorithms

Programming version

This series of lessons considers the issue of how computers search for data. Search algorithms are some of the most important tools used in computing. In the world of classical computing, there are a number of well-known algorithms for performing searches. For many years, it was thought that the efficiency of these algorithms formed a hard limit on how quickly computers could search for data. The advent of quantum computers promises to change all of this, with new quantum algorithms offering the potential for radical speed-up in a number of areas.

This lesson will cover the basics of three types of three classical algorithm that are used to search for data. In the remainder of the course, we will revisit these algorithms to see how quantum computers can be used to improve upon the performance in each case.

Problem 1: Searches in unstructured data arrays

Imagine we have a list of eight names, but the list is not in order. You want to find where one particular name occurs in this list. Here is the list of names:

Adriano	Birte	Edouard	Elena	Florian	Inez	Niamh	Oliver

Unfortunately, a computer is not able to scan the whole array at the same time, it can only look at one element of the array at a time and tell you whether the entry for that element matches the one you are looking for or not.

Exercise 1

The following shows a Python listing which sets up and shuffles the names array shown above:

```
import random
namesArray = ["Adriano", "Birte", "Edouard", "Elena", "Florian", "Inez", "Niamh", "Oliver"]
random.shuffle(namesArray)
nameToFind="Elena"
```

Add to this code to write a search routine that looks for the name “Elena” in the shuffled array. What is your strategy for finding the correct element? Play the game 10 times. What is the maximum number of guesses it takes you and what is the average number?

Problem 2: Searches in structured data arrays

Exercise 2a: searching an ordered array

The following shows a Python listing which sets up the array to be searched:

```
namesArray = ["Adriano", "Birte", "Edouard", "Elena", "Florian", "Inez", "Niamh", "Oliver"]
nameToFind="Inez"
```

Add to this code to write a search routine that looks for the name “Inez” in the array. When you check an element this time, you can check whether it matches the name you are looking for, and also whether it is higher or lower in terms of alphabetical order. What is your strategy for finding the correct element? Play the game with each of the different names. What is the maximum number of guesses it takes you and what is the average number?

Exercise 2b: searching an array of elements with sets of characteristics

Imagine our characters now have the following facial characteristics:

The game is now to identify one character from the set by asking questions about their appearance. Each question, however, can only be answered “yes” or “no”.

What questions do you ask? How many goes does it take to find the character?

The following shows a Python listing which sets up the array to be searched:

```
namesArray = [["Adriano", "Blond", "Blue", "No Glasses"], ["Birte", "Blond", "Blue", "Glasses"],
              ["Edouard", "Blond", "Brown", "No Glasses"], ["Elena", "Blond", "Brown", "Glasses"],
              ["Florian", "Brown", "Blue", "No Glasses"], ["Inez", "Brown", "Blue", "Glasses"],
              ["Niamh", "Brown", "Brown", "No Glasses"], ["Oliver", "Brown", "Brown", "Glasses"]]
```

Add to this code to write a search routine that looks for a person based on their characteristics. How many goes does it take to identify the character?

Problem 3: Factoring

Similar techniques to those used in search algorithms are also required for many different mathematical problems, e.g. to find a square root iteratively. One particular problem is how to find the prime factors of a large number.

Exercise 3

The following shows a Python listing which sets up an array of prime numbers to test and a function for checking whether a number is a prime number in that array:

```
primesArray = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
               43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
numberToFactorise=8192
primeFactorArray=[]

totalCalcs=0

def inPrimesArray(x, primesArray):
    isPrime=False
    for i in range(0,len(primesArray)):
        if x==primesArray[i]:
            isPrime=True
    return isPrime
```

Add to this code to write a routine which finds all the prime factors of the following two numbers: (a) 8192; (b) 6557. What was your strategy for doing this? How many calculations did it take to find all the prime factors?

The key to quantum supremacy: superposition

Quantum computing offers a fundamentally different approach to the classical algorithms shown above. The quantum algorithms described in the rest of this course all depend on the use of Hadamard gates to create and unwind superposition states, so before you go further with this course, please check you understand the basics of how quantum gates work.

Superposition gives us a tool for probing all possible input states at once. It will not always be the case that this will provide the search result in one go. In some cases, we may not know the answer with 100% certainty, but the probabilistic advantage we will gain can provide sufficient information to be a massive improvement over the classical world. Quantum computing involves considerable creativity and ingenuity, and in the remainder of this course we will show you some of the remarkable and beautiful tricks that will fundamentally change how we are able to search for data in the future.

The first realisation of the potential for quantum supremacy in search problems was **Deutsch's Algorithm**, which was first proposed by David Deutsch in 1985. In **Lesson 2**, we will explore this algorithm, which forms the basis for all the subsequent algorithms in this course.

Commentary on Exercise 1

The strategy you probably followed was to start with the first element and test each in turn until you found the data item you were looking for. You might also have chosen elements at random (ensuring that you never picked the same one twice), but in fact this would be no better. In either case, the maximum number of guesses you would need was 8 and the (long-run) average number of guesses was 4.5.

The strategy of testing each element in an array until you find a match is known as **linear search**. In the worst-case scenario, the number of comparisons we need is the same as the size of the data array we are searching. We say that the order of complexity of the algorithm is **O(n)** (using big O notation), because the number of comparisons we have to perform on an array of n elements is *of the order of n.*

In classical computing there is no way of searching an unsorted array of data faster than this. However, using a quantum computer, we may be able to perform such a search in **O($\sqrt{n}$)** operations, meaning we can search an array with n elements by a number of comparisons of the order of $\sqrt{n}$. We will learn about this method, known as **Grover's Algorithm**, in **Lesson 4**.

Commentary on Exercise 2a

The best strategy here is to guess in the middle of the array. If you find a match, stop. If your name is higher in the alphabet than the element you checked, go to the mid-point of the upper half of the array and test again. If it is lower, go to the mid-point of the lower half of the array and test again. Keep doing this until you find the name.

Depending on how you chose to round your mid-points, you should have found "Inez" in either two or three guesses. The greatest number of guesses required is four and the average required over all the names (using the algorithm shown) is 2.625.

This algorithm is known as **binary search**. Knowledge of the structure of the array makes this algorithm much faster than a linear search. In the worst-case scenario, the number of comparisons we need is linked to the logarithm (in base 2) of the number of elements of the array. In the example above, there are 8 elements: $\log_2 8 = 3$. We say that the order of complexity of the algorithm is **O(log(n))** (using big O notation), because the number of comparisons we have to perform on an array of n elements is *of the order of log(n).*

Commentary on Exercise 2b

Hopefully, you will have seen that by guessing one of the two possibilities for each characteristic, you could rule out half of the candidates each time. As there were eight characters, it took three questions to uniquely identify the character each time (note that the characters were chosen to represent all unique combinations of the three binary characteristics, so there could be no ambiguity in the final selection).

The answer in all cases is three guesses. This is a variant of the binary search shown in part 2a and, again, the order of complexity of the algorithm is **O(log(n))** (though in this case it will take exactly $\log_2 n$ guesses for every search).

In quantum computing, this particular structure of problem has a remarkable solution: the answer can be found using **exactly one computation, however large n**. It is known as the **Bernstein-Vazirani problem**, and we shall see how the algorithm works in **Lesson 3**.

Commentary on Exercise 3

One possible algorithm is to test each prime in turn. If the number does not divide by the prime, try the next prime in the list. If it does divide by the prime, calculate the quotient and test whether this number divides by each prime in the list (starting from the beginning again).

This strategy is known as **trial division**. With a number like 8192, which has many low prime factors, we can break down a number into its prime factors quite quickly (in this case, with 12 calculations). However, where a number has only two similar-sized prime factors, the algorithm is much slower. In the case of 6557 (=79 x 83), it takes 22 calculations to find the factors. If we were to construct a number which was the product of two primes which were, for example, 600 digits in length, not even a supercomputer would be able to factorise it. The **RSA algorithm**, which is used worldwide in encryption for the banking system and the wider internet, relies on this key fact for its security. In the classical computing world, this encryption routine is thought to be unbreakable. However, quantum computing may offer a way to break it. In **Lessons 5 and 6**, you will learn about the **RSA encryption system** and how **Shor's algorithm** offers a potential method for a quantum computer to it.