Die Überlegenheit von Quantenalgorithmen: Lektion 1 – Einführung in Suchalgorithmen

Version mit Programmierung

Diese Unterrichtsreihe beschäftigt sich mit der Frage, wie Computer Daten durchsuchen. Suchalgorithmen gehören zu den wichtigsten Werkzeugen der Informatik und sind grundlegend dafür, wie das World Wide Web funktioniert. Bei klassischen Computern gibt es eine Reihe gut bekannter Algorithmen, mit denen Suchvorgänge durchgeführt werden können. Über viele Jahre ging man davon aus, dass die Effizienz dieser Algorithmen eine feste Grenze dafür setzt, wie schnell Computer Daten finden können. Mit dem Aufkommen von Quantencomputern könnte sich das ändern: Neue Quantenalgorithmen versprechen in mehreren Bereichen eine deutlich schnellere Suche.

In dieser Stunde behandeln wir die Grundlagen von drei klassischen Suchverfahren. Im weiteren Verlauf des Kurses greifen wir diese Algorithmen wieder auf und untersuchen, wie Quantencomputer ihre Leistungsfähigkeit in jedem dieser Fälle verbessern können.

Szenario 1: Suchen in unstrukturierten Datenfeldern

Stellen Sie sich vor, wir haben eine Liste mit acht Namen, aber die Liste ist nicht sortiert. Sie möchten herausfinden, an welcher Stelle ein bestimmter Name in dieser Liste vorkommt. Hier ist die Namensliste:

Adriano	Birte	Edouard	Elena	Florian	Inez	Niamh	Oliver

Leider kann ein Computer das gesamte Datenfeld nicht gleichzeitig durchsuchen. Er kann immer nur ein Element nach dem anderen betrachten und feststellen, ob der Eintrag an dieser Stelle mit dem gesuchten Namen übereinstimmt oder nicht.

Aufgabe 1

Der folgende Python-Code zeigt, wie das oben dargestellte Namens-Array erzeugt und zufällig gemischt wird:

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

Ergänzen Sie diesen Code um eine Suchroutine, die im gemischten Array nach dem Namen „Elena“ sucht. Welche Strategie verwenden Sie, um das richtige Element zu finden? Führen Sie das Programm zehnmal aus. Wie viele Versuche benötigen Sie maximal, und wie viele im Durchschnitt?

Kommentar zu Aufgabe 1

Vermutlich sind Sie so vorgegangen, dass Sie beim ersten Element begonnen und dann jedes Element der Reihe nach geprüft haben, bis Sie den gesuchten Eintrag gefunden haben. Sie hätten auch Elemente zufällig auswählen können, sofern Sie dabei keines zweimal wählen. Im Durchschnitt wäre das jedoch nicht besser.

TAKEN FROM QUANTUM COMPUTING IN STEM EDUCATION
A project by Science on Stage Germany

In beiden Fällen liegt die maximale Anzahl an Versuchen bei 8, und im Durchschnitt benötigt man über viele Durchgänge hinweg 4,5 Versuche.

Die Strategie, jedes Element eines Datenfeldes nacheinander zu prüfen, bis eine Übereinstimmung gefunden wird, nennt man **lineare Suche**. Im ungünstigsten Fall entspricht die Anzahl der Vergleiche der Größe des Datenfeldes, das durchsucht wird. Man sagt, die Komplexität des Algorithmus ist $O(n)$(Big-O-Notation), weil die Anzahl der nötigen Vergleiche bei einem Datenfeld mit n Elementen in der Größenordnung von n liegt.

Bei klassischen Computern gibt es keine Möglichkeit, ein unsortiertes Datenfeld schneller als mit dieser Vorgehensweise zu durchsuchen. Mit einem Quantencomputer kann man eine solche Suche in $O(\sqrt{n})$Schritten durchführen. Diese Methode, bekannt als **Grover-Algorithmus**, behandeln wir in Stunde 4.

Szenario 2: Suchen in strukturierten Datenfeldern

Aufgabe 2a: Suche in einem geordneten Feld

Der folgende Python-Code zeigt, wie das zu durchsuchende Array angelegt wird:

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

Ergänzen Sie den Code so, dass er im Array nach dem Namen „Inez“ sucht. Wenn Sie ein Element überprüfen, können Sie dieses Mal nicht nur feststellen, ob es mit dem gesuchten Namen übereinstimmt, sondern auch, ob es im Alphabet vor oder nach dem gesuchten Namen steht. Welche Strategie verwenden Sie, um den richtigen Eintrag zu finden? Testen Sie Ihr Vorgehen mit allen Namen aus dem Array. Wie viele Versuche brauchen Sie höchstens, und wie viele brauchen Sie im Durchschnitt?

Kommentar zu Aufgabe 2a

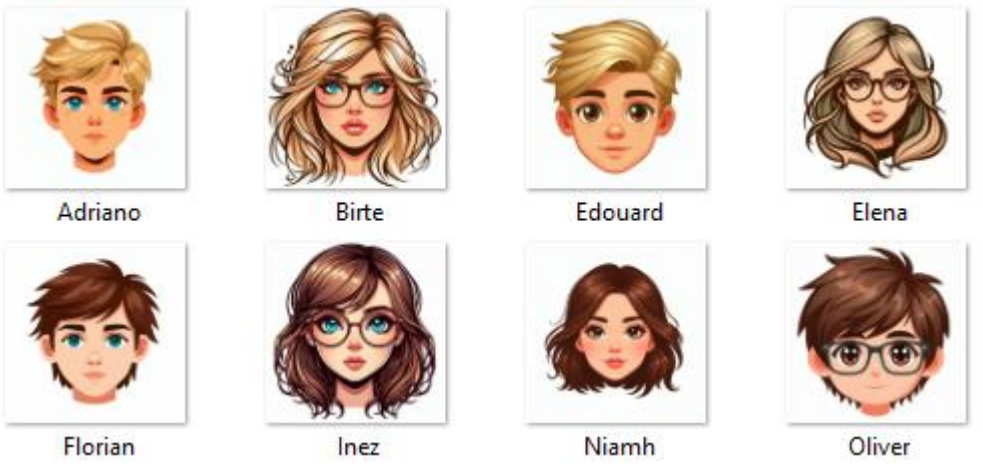
Die beste Strategie ist, zunächst das mittlere Element des Datenfeldes zu prüfen. Wenn Sie eine Übereinstimmung finden, sind Sie fertig. Ist der gesuchte Name im Alphabet weiter hinten als der geprüfte Name, gehen Sie zum Mittelpunkt der oberen Hälfte des Datenfeldes und prüfen erneut. Ist er im Alphabet weiter vorne, gehen Sie zum Mittelpunkt der unteren Hälfte und prüfen erneut. Wiederholen Sie dieses Vorgehen, bis Sie den Namen gefunden haben.

Je nachdem, wie Sie bei der Bestimmung der Mittelpunkte gerundet haben, sollten Sie „Inez“ nach zwei oder drei Versuchen gefunden haben. Die maximale Anzahl an Versuchen beträgt vier, und der Durchschnitt über alle Namen liegt bei Anwendung dieser Strategie bei 2,625.

Dieses Verfahren nennt man binäre Suche. Die Kenntnis der Struktur des Datenfeldes macht diese Suche deutlich schneller als eine lineare Suche. Im ungünstigsten Fall hängt die Anzahl der Vergleiche mit dem Logarithmus zur Basis 2 der Anzahl der Elemente zusammen. Im Beispiel oben gibt es 8 Elemente: $\log_2(8) = 3$. Wenn es 1024 Elemente gäbe, wären höchstens 11 Vergleiche nötig $\log_2(1024) = 10$. Man sagt, die Komplexität des Algorithmus ist $O(\log(n))$ (Big-O-Notation), weil die Anzahl der nötigen Vergleiche bei einem Datenfeld mit n Elementen in der Größenordnung von $\log(n)$ liegt.

Aufgabe 2b: Suche in einem Datenfeld mit Elementen, die durch Merkmale beschrieben sind

Stellen Sie sich vor, unsere Figuren haben nun die folgenden Gesichtseigenschaften:



Das Ziel ist nun, eine Figur aus der Menge zu bestimmen, indem Sie Fragen zu ihrem Aussehen stellen. Jede Frage darf jedoch nur mit **„Ja“** oder **„Nein“** beantwortet werden. Welche Fragen stellen Sie? Wie viele Versuche benötigen Sie, um die Figur zu finden?

Der folgende Python-Code zeigt, wie das zu durchsuchende Array angelegt wird:

```
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"]]
```

Ergänzen Sie den Code so, dass er anhand der Merkmale nach einer Person sucht. Wie viele Versuche sind nötig, um die Figur zu bestimmen?

Kommentar zu Aufgabe 2b

Vermutlich haben Sie erkannt, dass Sie durch das Abfragen einer von zwei Möglichkeiten pro Merkmal jedes Mal etwa die Hälfte der Kandidaten ausschließen konnten. Da es acht Figuren gab, reichten drei Fragen, um die Figur jedes Mal eindeutig zu bestimmen (die Figuren wurden so ausgewählt, dass sie alle unterschiedlichen Kombinationen der drei binären Merkmale darstellen, sodass am Ende keine Mehrdeutigkeit entstehen kann).

Das Ergebnis ist in allen Fällen: drei Versuche. Das ist eine Variante der binären Suche aus Aufgabe 2a. Entsprechend liegt die Komplexität des Algorithmus wieder bei $O(\log(n))$(wobei man in diesem Fall bei jeder Suche genau $\log_2(n)$ Fragen benötigt).

Im Quantencomputing hat genau diese Problemstruktur eine bemerkenswerte Lösung: Die gesuchte Antwort kann mit einem einzigen Durchlauf gefunden werden, unabhängig davon, wie groß n ist. Dieses Problem ist als Bernstein-Vazirani-Problem bekannt, und wir werden in Stunde 3 sehen, wie der zugehörige Algorithmus funktioniert.

Szenario 3: Mathematische Suchaufgaben

Suchalgorithmen werden für viele unterschiedliche mathematische Aufgaben benötigt, zum Beispiel um eine Quadratwurzel zu berechnen oder die Lösung einer Gleichung zu finden. Eine spezielle Aufgabe besteht darin, die **Primfaktoren** einer großen Zahl zu bestimmen.

Aufgabe 3

Der folgende Python-Code zeigt, wie ein Array mit Primzahlen zum Testen sowie eine Funktion zur Überprüfung angelegt wird, ob eine Zahl eine Primzahl aus dieser Liste ist:

```
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
```

Ergänzen Sie den Code so, dass er alle Primfaktoren der folgenden beiden Zahlen bestimmt: (a) 8192, (b) 6557. Welche Strategie verwenden Sie dabei? Wie viele Rechenschritte sind nötig, um alle Primfaktoren zu finden?

Kommentar zu Aufgabe 3

Die richtige Strategie besteht darin, jede Primzahl der Reihe nach zu testen. Wenn die Zahl nicht durch die Primzahl teilbar ist, nimmt man die nächste Primzahl in der Liste. Ist sie teilbar, berechnet man den Quotienten und prüft dann erneut, ob dieser Quotient durch die Primzahlen in der Liste teilbar ist (wieder beginnend von vorne).

Diese Strategie nennt man **Probeteilung** (*trial division*). Bei einer Zahl wie 8192, die viele kleine Primfaktoren besitzt, kann man die Primfaktorzerlegung relativ schnell durchführen (in diesem Fall mit 12 Rechenschritten). Wenn eine Zahl jedoch nur zwei Primfaktoren hat, die zudem ähnlich groß sind, ist das Verfahren deutlich langsamer. Im Fall von 6557 ($= 79 \cdot 83$) benötigt man 22 Rechenschritte, um die Faktoren zu finden.

Würde man eine Zahl konstruieren, die das Produkt aus zwei Primzahlen ist, die zum Beispiel jeweils 600 Stellen haben, könnte sie nicht einmal ein Supercomputer faktorisieren. Der **RSA-Algorithmus**, der weltweit zur Verschlüsselung im Bankwesen und im Internet eingesetzt wird, stützt seine Sicherheit genau auf diese Tatsache. Bei klassischen Computern gilt dieses Verschlüsselungsverfahren daher als praktisch nicht zu knacken.

Quantencomputing könnte jedoch Wege eröffnen, diese und andere Verfahren zu brechen, die bislang als unknackbar galten. In **Stunde 5 und 6** lernen Sie das **RSA-Verschlüsselungssystem** kennen und sehen, wie der **Shor-Algorithmus** einen möglichen Weg bietet, wie ein Quantencomputer RSA angreifen könnte.

Der Schlüssel zum Quanten-Vorteil: Superposition

Quantencomputing bietet einen grundlegend anderen Ansatz als die oben gezeigten klassischen „Trial-and-Error"-Verfahren. Die Quantenalgorithmen, die im weiteren Verlauf dieses Kurses behandelt werden, beruhen darauf, mithilfe von Hadamard-Gattern Superpositionszustände zu erzeugen und am Ende wieder zurückzuführen. Bevor Sie mit dem Kurs fortfahren, sollten Sie daher sicherstellen, dass Sie die Grundlagen dazu verstehen, wie Quantengatter funktionieren.

Superposition ist ein Werkzeug, mit dem man viele mögliche Eingabefälle gleichzeitig in einem Zustand darstellen kann. Das bedeutet jedoch nicht automatisch, dass man das Suchergebnis immer „in einem Schritt" erhält. In manchen Fällen zwingt uns die Quantenphysik zu einem Kompromiss: Je nachdem, was wir messen, verlieren wir Information über andere Aspekte des Zustands. In anderen Fällen liefert ein Quantenalgorithmus die Antwort nicht mit 100% Sicherheit, aber der Wahrscheinlichkeitsvorteil kann bereits ausreichen, um gegenüber klassischen Verfahren einen großen Gewinn zu erzielen. Quantencomputing erfordert viel Kreativität. Im weiteren Verlauf dieses Kurses lernen Sie einige bemerkenswerte und elegante Tricks kennen, die grundlegend verändern, wie wir in Zukunft nach Daten suchen können.

Der erste Hinweis auf einen möglichen Quanten-Vorteil bei Suchproblemen war der **Deutsch-Algorithmus**, der 1985 von David Deutsch vorgeschlagen wurde. In **Stunde 2** werden wir diesen Algorithmus behandeln. Er bildet die Grundlage für alle weiteren Algorithmen in diesem Kurs.