6.4 Weitere einfache Suchverfahren

6.4.1 Die Sprungsuche

Man springt im sortierten Array in festen Schritten vorwärts, bis der Block gefunden ist, in dem die Suchzahl liegen muss. Danach durchsucht man nur noch diesen Block linear.

1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39
0				4				8				12				16			

Vergleich 1 <= 31 = true → 4 Elemente weiter

Vergleich 9 <= 31 = true → 4 Elemente weiter

Vergleich 17 <= 31 = true → 4 Elemente weiter

Vergleich 25 <= 31 = true → 4 Elemente weiter

Vergleich 33 <= 31 = false → zurück zum vorherigen Block (Index 12+1)

Ab hier: Lineare Suche

Vergleich 27 <= 31 = false → 1 Element weiter

Vergleich 29 <= 31 = false → 1 Element weiter

Vergleich 31 <= 31 = false → 1 Element weiter

Zeitverhalten O(N), aber deutlich schneller als sequentielle Suche.

Optimale Sprungweite

Die optimale Sprungweite **m** berechnet sich nach **m = sqrt(n)**, wobei **n** die Zahl der zu durchsuchenden Elemente ist.

Bei einem Array der Größe 1000 wäre die optimale Sprungweite also **sqrt(1000) = 32** (gerundet).

Zwei-Ebenen-Sprungsuche

Bei einem Array der Größe 1.000.000 wäre die optimale Sprungweite **m = 1000**. Statt jetzt den gefundenen Zielabschnitt linear oder binär zu durchsuchen, bietet sich hierfür eine weitere Sprungsuche an. Man spricht dann von einer ***Zwei-Ebenen-Sprungsuche***.

Anwendung:

Die Sprungsuche eignet sich am besten dann, wenn die Zahlen bzw. zu suchenden Elemente weitgehend **gleichmäßig verteilt** sind.

6.4.2 Die Indexsuche

Man führt ein zusätzliches Index-Array, um schnell den passenden Wertebereich einzugrenzen. Anschließend sucht man nur noch im zugehörigen Teilbereich, meistens linear oder binär oder ebenfalls mit einer Indexsuche. Im letzteren Fall spricht man von einer ***mehrstufigen Indexsuche***.

Beispiel 1

Das Haupt-Array:

1	4	5	9	12	13	15	16	18	19	20	25	32	34	38	44	46	49	52	55	58	63	64	67	74	77
0				4						10		12			15			18			21			24	

Das Index-Array:

0	4	10	12	14	18	21	24
0	1	2	3	4	5	6	7
Start	erste Zahl >= 10	erste Zahl >= 20	erste Zahl >= 30	erste Zahl >= 40	erste Zahl >= 50	erste Zahl >= 60	erste Zahl >= 70

Beschreibung

Oben sehen wir ein Array aus 26 sortierten int-Zahlen im Bereich zwischen 1 und 77.

Darunter ist ein ***Index-Array*** gezeigt, dieses besteht aus 8 int-Zahlen. Die Werte dieses Index-Arrays verweisen auf bestimmte Indizes in dem Haupt-Array.

Im Element mit dem Index 1 steht beispielsweise die Zahl 4. Das ist die Position der ersten Zahl im ***Haupt-Array***, die größer oder gleich 10 ist.

An Position 2 im Index-Array steht die Zahl 10. An Index 10 im Haupt-Array befindet sich die erste Zahl >= 20. Entsprechend sind die Startwerte der nächsten Zehner-Intervalle an den Positionen 3 bis 7 des Index-Arrays gespeichert: 12, 14, 18, 21 und 24.

Suchbeispiel

Wir wollen nun die Zahl 44 suchen.

- Wir dividieren 44 durch 10 (ganzzahlig) und erhalten als Ergebnis 4.
- Dann schauen wir im Index-Array an Position 4 nach und sehen dort den Index 14.
- Wir springen in das Haupt-Array an Position 14 und sehen dort die 38. Das ist unser Startpunkt für die eigentliche Suche.
- Die nächste Zahl stimmt bereits mit der Suchzahl überein.

Beispiel 2

In dem zweiten Beispiel wollen wir die Indexsuche auf ein Array anwenden, das aus 10.000 Schlüsselwerten besteht, die im Bereich zwischen 1 und 20.000 liegen und auf die eigentlichen Daten verweisen.

In einem Hilfs-Array werden jetzt bestimmte Indizes des Haupt-Arrays gespeichert:

- hilf[0] = Index der ersten Zahl >= 2.000 ist 1.567
- hilf[1] = Index der ersten Zahl >= 4.000 ist 3.455
- hilf[2] = Index der ersten Zahl >= 6.000 ist 4.678
- hilf[3] = Index der ersten Zahl >= 8.000 ist 4.899
- hilf[4] = Index der ersten Zahl >= 10.000 ist 5.012
- hilf[5] = Index der ersten Zahl >= 12.000 ist 6.301
- hilf[6] = Index der ersten Zahl >= 14.000 ist 6.900
- hilf[7] = Index der ersten Zahl >= 16.000 ist 7.520
- hilf[8] = Index der ersten Zahl >= 18.000 ist 8.910

Angenommen, wir suchen jetzt den Schlüsselwert 12.345. Dann würden wir zunächst im Hilfs-Array nachschauen, bei welchem Index die Zahlen >= 12.000 beginnen. Das ist dann die Position 6.301.

Die eigentliche Suche im Hauptarray beginnt dann bei genau diesem Index. Der Abschnitt umfasst immer ca. 600 Zahlen, also könnte man hier vielleicht statt einer linearen Suche noch eine einfache binäre Suche einsetzen.

Zeitverhalten der Indexsuche

Das Zeitverhalten der Indexsuche ist O(N), wie bei der sequentiellen Suche.

Trotzdem ist die Indexsuche deutlich schneller als die sequentielle Suche. Solche Zeitvorteile werden in der O-Notation allerdings nicht berücksichtigt. Entscheidend ist nur der Unterschied: **logarithmisch < linear < quadratisch < exponentiell**.

6.4.3 Die Interpolationssuche

Wenn die Zahlen bzw. Elemente in dem Hauptarray einigermaßen gleichmäßig verteilt sind, benötigt man keinen Index-Array. Stattdessen kann man versuchen, die Einstiegsposition für die Suche zu *berechnen*, also zu interpolieren. Daher wird dieses Suchverfahren auch als ***Interpolationssuche*** bezeichnet.

1	3	5	8	11	13	15	17	18	23	25	27	28	29	30	**31**	33	38	39	41
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	**15**	16	17	18	19

Grundlegende Berechnungen

Spannweite

Zunächst einmal müssen wir die ***Spannweite*** der Werte in dem Array ermitteln. Dazu subtrahieren wir den kleinsten Wert vom größten Wert des Arrays. Bei einem sortierten Array ist das kein Problem, der kleinste Wert steht an Index `0`, der größte Wert an Index `length-1`.

Für unser Beispiel-Array beträgt die Spannweite also: 41 - 1 = 40.

Zahl der Intervalle

Die Zahl der Intervalle ist um 1 kleiner als die Zahl der Arrayelemente. Bei unserem Array haben wir es also mit 19 Intervallen zu tun.

Wertanstieg pro Index

Als Nächstes berechnen wir, wie stark die Werte des int-Arrays von Intervall zu Intervall ansteigen. Dazu dividieren wir die Spannweite durch die Zahl der Intervalle:

40 / 19 = 2,1

Pro Index steigt der Wert der Zahlen um ca. 2,1.

Interpolierter Index

Die Formel für die Berechnung des wahrscheinlichsten Einstiegspunkts lautet:

(Suchzahl - kleinster Wert) / Wertanstieg pro Index

Suchbeispiele

Suche nach der 31

Berechnung der möglichen Position der Suchzahl:

(31 - 1) / 2,1 = 14

Der kleinste Wert des Arrays ist 1, daher wird hier und in den beiden folgenden Beispielen die entsprechende Zahl eingesetzt.

Ab Index 14 beginnt dann die lineare Suche. Nach einem Vergleich wird die 31 gefunden.

Suche nach der 13:

Berechnung der möglichen Position der Suchzahl:

(13 - 1) / 2,1 = 5

Suchzahl wird sofort gefunden.

Suche nach der 25:

Berechnung der möglichen Position der Suchzahl:

(25 - 1) / 2,1 = 11

An Index 11 befindet sich die 27 - also wird rückwärts weiter gesucht.

Nach einem weiteren Vergleich wird die 25 gefunden.

Das wohl bekannteste Praxisbeispiel für eine Interpolationssuche ist die Suche in einem Telefonbuch. Wenn Sie in einem dicken Telefonbuch der Stadt Hamburg nach einem Namen wie "Wehemeier" suchen, fangen Sie bestimmt nicht in der Mitte des Buches an, sondern schlagen es gleich weiter hinten auf. Sie schätzen also den Einstieg in die Suche ab und ermitteln eine günstige Anfangsposition.

Auch wenn Sie in einem dicken Hochschul-Lehrbuch im Register nach einem Fachbegriff wie "Binäre Suche" suchen, werden Sie nicht bei "A" oder "Z" anfangen, sondern abschätzen, wo ungefähr die Stichwörter mit "Bi" stehen.

Das Register des berühmten Hollemann/Wiberg, Lehrbuch der Anorganischen Chemie, besteht beispielsweise aus 95 eng bedruckten Seiten. Wenn Sie dort nach "Bleioxid" suchen, werden sich ganz sicher eine Art Interpolationssuche anwenden.

6.5 Vergleich der nicht-linearen Suchverfahren

Wir haben nun

die binäre Suche

die Sprungsuche

die Indexsuche und

die Interpolationssuche

als nicht-lineare Suchverfahren kennengelernt. Alle vier Verfahren funktionieren nur, wenn die zu durchsuchende Sammlung von Elementen aufsteigend oder absteigend sortiert ist, wobei natürlich das Sortierkriterium mit dem Suchkriterium übereinstimmen muss. Die Suche nach einem Interpreten in einer nach Kaufdatum sortierten CD-Sammlung kann natürlich nur linear (sequentiell) erfolgen.

In diesem Abschnitt wollen wir nun das Zeitverhalten der vier nicht-linearen Suchalgorithmen experimentell erforschen

6.5.1 Die Klasse VergleicheSuchverfahren

Auf den oben verlinkten Webseiten finden Sie kurze Quelltexte zu den einzelnen Suchverfahren, die aber noch nicht lauffähig sind. Sie müssen jeweils noch in eine Java-Klasse eingebettet werden. Diese Aufgabe sollte für Sie aber kein Problem darstellen.

Ich habe dann die ChatGPT gebeten, aus den vier von mir implementierten Java-Methoden (auf den Webseiten vorhanden) eine große Klasse **VergleicheSuchverfahren** zu erstellen, mit der die vier Suchverfahren verglichen werden können.

Meine Vorgaben dabei waren:

- Es sollen aufsteigend sortierte int-Arrays mit 64, 128, 256, 512, 1024 und 2048 Elementen untersucht werden.
- Der Wertebereich der Arrayelemente soll zwischen 1 und 5000 liegen.
- Bei den vier Algorithmen soll nur die Zahl der erforderlichen Vergleiche mitgezählt werden. Eine Methode wie sprungsuche() liefert also nicht den Index der gefundenen Zahl (oder -1) zurück, sondern die Anzahl der erforderlichen Vergleiche.

Den kompletten und sehr langen Quelltext dieser von ChatGPT erstellten Klasse können Sie hier als Java-Datei herunterladen:

Java-Datei VergleicheSuchverfahren.java

Die Klasse wurde unter BlueJ und IntelliJ getestet, sie funktioniert dort einwandfrei.

6.5.2 Ergebnisse der Vergleiche

Wir wollen uns nun die Ergebnisse dieser Untersuchung anschauen und betrachten dazu die Konsolenausgabe des Programms:

```
Vergleich von vier Suchverfahren

Suchanfragen: alle Zahlen von 1 bis 5000
```

n	Sprungsuche	Indexsuche	Interpolation	Binaersuche
64	91032 / 18,21	83743 / 16,75	29934 / 5,99	59982 / 12,00
128	114770 / 22,95	115771 / 23,15	29870 / 5,97	69712 / 13,94
256	151441 / 30,29	179692 / 35,94	29742 / 5,95	79290 / 15,86
512	199152 / 39,83	307549 / 61,51	29486 / 5,90	88504 / 17,70
1024	271109 / 54,22	562964 / 112,59	28972 / 5,79	96960 / 19,39
2048	367891 / 73,58	1073763 / 214,75	27950 / 5,59	103886 / 20,78

Links sieht man jeweils die Gesamtzahl der Vergleiche für alle 5000 Suchzahlen, rechts daneben jeweils den Mittelwert für das Suchen nach einer Zahl.

Es stellt sich heraus, dass - wie bereits vermutet - die Interpolationssuche das schnellste Verfahren ist, gefolgt von der Binärsuche. Die Sprungsuche ist auf Platz 3, und am langsamsten ist die Indexsuche.

Auffällig ist insbesondere das Verhalten der ***Interpolationssuche*** bei wachsender Arraygröße n. Während bei den anderen Verfahren die Zahl der Vergleiche mit zunehmendem n mehr oder weniger deutlich ansteigt, bleibt sie bei der Interpolationssuche nahezu konstant. Das ist mathematisch gut erklärbar: Bei gleichmäßig verteilten Werten kann die Position der Suchzahl sehr genau abgeschätzt werden, sodass häufig schon der erste berechnete Zugriff direkt zum Ziel führt oder nur noch sehr wenige Nachvergleiche nötig sind.

Die ***Binärsuche*** zeigt ebenfalls ein recht günstiges Verhalten. Ihre Zahl der Vergleiche wächst nur logarithmisch mit n, also ungefähr proportional zu log2(n). Selbst bei einer Verdopplung der Arraygröße steigt der Aufwand daher nur um etwa einen zusätzlichen Vergleich.

Die ***Sprungsuche*** ist deutlich langsamer als die Binärsuche, aber immer noch schneller als die Indexsuche.

Bei der ***Indexsuche*** kann ein Hilfsarray den Einstieg in die Suche verbessern, doch entsteht ein zusätzlicher Aufwand durch das Suchen im Index und durch die anschließende Suche im betreffenden Intervall.