Data Oriented Design

1. Steckbrief Algorithmen

1.1 Bubble Sort

Beim Bubblesort Algorithmus wird ein Array paarweise von links nach rechts sortiert, es durchläuft sogenannte Bubble-Phasen. Es wird mit der ersten Zahl gestartet, diese wird mit dem Nachbar verglichen nach dem Sortierkriterium, falls beide Zahlen nicht in der richtigen Reihenfolge sind, werden sie getauscht. In-Place Prinzip.

Theoretische Komplexität (Time/Space)

Durchschnittliche Zeitkomplexität von $O(n^2)$, ziemlich langsam, kaum verwendet in der Praxis. Auch im Worst Case $O(n^2)$, da der Alogrithmus paarweise voranschreitet und damit entsprechend viele Paare vergleichen muss. Nur im Best Case erreicht er eine Laufzeit von O(n).

Beispiel Best Case Szenario

Ein Best Case Szenario wäre ein Array, welches bereits von Anfang an nach dem Sortierkriterium sortiert ist wie zum Beispiel:
[2] [3] [5] [9] [15] [20]
Laufzeit -> O(n)

Beispiel Worst Case Szenario

Ein Worst Case Beispiel wäre das genau Gegenteil, ein Array welches in die andere Richtung sortiert ist, die nicht dem Sortierkriterium entspricht.
[20] [15] [9] [5] [3] [2]
Laufzeit -> O(n^2)

1.2 Counting Sort

Beim Counting Sort handelt es sich um keinen vergleichsbasierten Sortieralgorithmus, er arbeitet **adressbasiert**. Out-of-place Prinzip. Das bedeutet, statt einzelne Werte zu vergleichen, vergleicht man die Häufigkeiten der Schlüsselwerte.

Theoretische Komplexität (Time/Space)

Die Laufzeit hängt von der Anzahl der Elemente in einer Liste n und der Größe des Zahlenintervalls k ab. Es handelt sich um einen linearen Zeitaufwand, durch die for-Schleife besitzt jeweils n- oder k- Durchläufe.-> **O(n+k**)

Außerdem weist der Algorithmus eine Speicherkomplexität von O(n+k) auf, da der Algorithmus eine temporäre Liste / Hilfsarray für die Zwischenspeicherung der Zahlenwerte benötigt, daher beträgt der Speicherplatz k-Elemente.

1.3 Selection Sort

Der Selection sort gehört zu den instabilen Sortieralgorithmen. Man kann den Selection Sort auf zwei Arten implementieren, entweder auf die min Basis oder max Basis. Das Vorgehen bleibt bei beiden Variationen gleich. Entweder sucht man sich den kleinsten Wert oder den größten Wert in der Liste und vergleicht ihn mit den anderen Werten.

Theoretische Komplexität (Time/Space)

Das Verfahren hat eine Zeitkomplexität von **$O(n^2)$**, egal für welchen Fall. Die Laufzeit bleibt beim **Worst Case** genau gleich wie beim **Best Case.** Denn die Liste benötigt bei jedem Durchlauf n (n-1)/2 Vergleiche.

2. Unser Programm

In unserem Array Creator erstellen wir die benötigten Arrays, und speichern sie in einer csv Datei ab, damit wir immer konstante Daten haben. Unsere CacheSize definierten wir mit 16 * 1024 * 1024 da wir 16MB Cache besitzen.
Unser FatArray repräsentiert die Array Size > Cache Size.
ThinArray repräsentiert die Array Size < Cache Size.
ThinStructArray repräsentiert ein StructArray aus int und strings.
ZeroOneArray besteht aus lauter 0 und 1en.
ZeroTenThousandArray besteht aus 0 – 10.000.

Nach erfolgreicher Erstellung der benötigten Arrays können wir mit den Benchmarks beginnen. Unser Programm ist als kleines Menü aufgebaut, indem man den gewünschten Algorithmus und Array auswählen kann. Danach lädt das Programm die gewünschte csv Datei und das Sortieren beginnt. Gleichzeitig lassen wir eine Stopwatch laufen, die die benötigte Zeit erfasst. Zum Schluss wird die benötigte Zeit in Stunden, Minuten, Sekunden und Millisekunden ausgegeben.

3. Erwartete Auswirkungen

Wir erwarten uns im Vorhinein, dass der Bubble Sort Algorithmus am schlechtesten Abschneiden wird, da dieser Algorithmus mit einer Laufzeit von $O(n^2)$ daherkommt da auch unsere Arrays nicht vorsortiert sind welche dem Bubble Sort Algorithmus einen Vorteil verschaffen könnte, da er dort eine Laufzeit von O(n) erreicht. Da das bei uns nicht so ist, glauben wir, dass der Selection Sort Algorithmus gegenüber dem Bubble Sort Algorithmus im Vorteil liegen könnte, da er Bubble Sort Algorithmus jeden benachbarten Wert vergleicht, der Selection Sort hingegen sucht sich gleich zu Beginn den kleinsten Wert (oder größten Wert) und setzt ihn an den Anfang (Ende) der Liste. Aus diesem Grund könnte Selection Sort einen Vorteil gegenüber dem Bubble Sort Algorithmus haben. Unser Favorit ist jedoch der Counting Sort Algorithmus da er adressbasiert arbeitet und eine Laufzeit von O(n+k) hat.

Array vs Cache:
Wenn das Array kleiner als die Cache Size gewählt wird, erhoffen wir uns bei allen Algorithmen eine schnellere Laufzeit, als wenn das Array größer als die Cache Size ist.

Struct vs int Array:
Beim Vergleich Struct vs int Array glauben wir, dass das int Array besser abschneiden wird, da die Größe des Arrays konstant ist. In Structs hingegen kann die Größe stark variieren, da darin int als auch strings stehen können und daher glauben wir, das die Sturcts eine längere Laufzeit haben.

Array 0 – 1 vs Array 0 – 10.000
Bei unserem letzten Vergleich erhoffen wir uns, dass bei einem Array mit den Werten von 0-1 eine kürzere Laufzeit auftritt als bei einem Array mit den Werten von 0 – 10.000. Denn wir vermuten, dass hier die Vergleiche schneller von statten gehen, da nicht so viele Umsortierungen getätigt werden müssen.

4. Benchmark Maschine
Prozessor: Intel® Core™ i7-10875H CPU, 2.30GHz mit 16MB Cache

6. Laufzeiten und Interpretation der 6 Ergebnisse

Variation	**Algorithmus**	**Laufzeit**
Array Size < Cache	Bubble Sort	11min 40sek 946ms
Array Size < Cache	Counting Sort	0,0043ms
Array Size < Cache	Selection Sort	03min 04sek 965ms
Array Size > Cache	Bubble Sort	>10h abgebrochen
Array Size > Cache	Counting Sort	0,43492ms
Array Size > Cache	Selection Sort	>15h abgebrochen
Array complex Struct	Bubble Sort	30min 38sek 652ms
Array complex Struct	Counting Sort	0,0108ms
Array complex Struct	Selection Sort	08min 17sek 376ms
Array besteht aus 0 und 1	Bubble Sort	10min 45sek 357ms
Array besteht aus 0 und 1	Counting Sort	0,00419ms
Array besteht aus 0 und 1	Selection Sort	03min 05sek 656ms
Array besteht aus 0 – 10.000	Bubble Sort	11min 41sek 714ms
Array besteht aus 0 – 10.000	Counting Sort	0,00458ms
Array besteht aus 0 – 10.000	Selection Sort	03min 03sek 536ms

7. Conclusio

Die absolute Speerspitze stellt der Counting Sort Algorithmus da. Die adressbasierten Zugriffe scheinen sehr effizient zu sein, da er nicht wie der Bubble Sort Algorithmus jeden benachbarten Wert einzeln vergleichen muss. Er überzeugt mit absoluter Performance und Schnelligkeit. Der Selection Sort Algorithmus arbeitet effizienter als der Bubble Sort Algoritmus, was wir schon im Vorhinein erwartet haben aber das der Unterschied so groß ist hätten wir uns nicht gedacht. Das kleinste Element als erstes zu Suchen und es an den Anfang der Liste zu setzen, scheint durchaus effizient zu sein. Der Bubble Sort Algorithmus ist äußerst langsam und belegt klarerweise den dritten Platz unserer Benchmark Übung.

Sehr interessant zu beobachten war, wenn die Array Größe größer als der Cache gewählt wird, verschlechtert sich die Laufzeit um ein Vielfaches. Egal bei welchem Algorithmus jeder schnitt schlechter ab. Wir brachen den BubbleSort Algorithmus sowie den SelectionSort Algorithmus jedoch nach 10 bzw. 15 Stunden ab. Des Weiteren stechen uns die Structs ins Auge. Auch hier stellten wir einen erheblichen Arbeitsaufwand fest, welcher sich in einer längeren Laufzeit auswirkte. Unsere Vermutung liegt darin, dass hier nicht nur Integer gespeichert sind sondern auch Strings und dadurch verschlechtert sich der Zugriff enorm. Ganz witzig zu beobachten war unser letzter Vergleich, welcher die Arrays von 0-1 und 0-10.000 vergleicht. Anscheinend muss man bei dem letzteren Array mehrere Vergleiche anstellen, wenn die Zahlen in einem großen Zahlen Pool liegen, als wenn sie nur zwischen 0 und 1 liegen.

Zusammenfassend können wir sagen, dass wir von der Performance des Counting Sort Algorithmus sehr begeistert sind und ihn auf jeden Fall für Zukünftige Projekte im Hinterkopf behalten werden, aufgrund seiner Performance.