# Benchmarking Cache Efficiency

## Algorithmen - Steckbriefe

### Bubble Sort

Der Bubble Sort Algorithmus sortiert Elemente vergleichsbasiert, stabil und in-place. Die Eingabe-Liste wird von links nach rechts durchlaufen und dabei in jedem Schritt das aktuelle Element mit dem rechten Nachbarn verglichen. Falls die beiden Elemente das Sortierkriterium verletzen, werden sie getauscht. Diese Phase wird meist als „Bubble-Phase“ bezeichnet und solange wiederholt, bis die Eingabeliste vollständig sortiert ist, wobei das jeweils letzte Element des vorherigen Durchlaufs nicht mehr betrachtet werden muss.

**Theoretische Komplexität (Time/Space):**

| Average Case | Worst Case | Best Case | Space |
|---|---|---|---|
| **$O(N^2)$** | **$O(N^2)$** | **O(N)** | **O(1)** |

**Beispiel eines Best-Case Szenarios:**
Der Best-Case für den Bubble Sort Algorithmus tritt ein, wenn die Elemente bereits sortiert sind. Hier wird der Algorithmus die Liste nur einmal durchgehen und somit feststellen, dass bereits alle richtig sortiert sind und niemals getauscht werden muss.

Der Bubble Sort funktioniert also vor allem dann gut, wenn nur wenige Elemente an der falschen Stelle sind. Wenn die Elemente einer Liste also mit einer hohen Wahrscheinlichkeit bereits sortiert sind, dann würde sich der Bubble Sort Algorithmus in diesem Fall gut eignen.

**Beispiel eines Worst Case Szenarios:**
Der Worst-Case für den Bubble Sort Algorithmus tritt ein, wenn die Elemente genau umgekehrt sortiert sind. Hier muss nämlich in jedem Schritt in jedem Durchlauf ein Tausch durchgeführt werden, was gesamt ½ * (n*(n-1)) Vertauschungen entspricht.

Hat man beispielsweise eine bereits sortierte Liste, die man genau andersrum sortieren möchte, dann sollte man auf keinen Fall den Bubble Sort Algorithmus dafür verwenden.

**Mögliche Einschränkungen bei der Anwendbarkeit:**
Aufgrund der auch im Average-Case schlechten Laufzeit wird der Bubble Sort eher selten in der Praxis eingesetzt, da andere verfahren im Vergleich deutlich schneller (im Durchschnitt) sind. Allerdings kann er für kleine Eingaben in Frage kommen und vor allem auch zu Demonstrations- und Lernzwecken Anwendung finden. Ein weiterer Anwendungsfall wäre die Verwendung von Bubble Sort innerhalb eines rekursiv arbeitenden Sortierverfahrens, um die Anzahl an Rekursionen zu verringern.

### Insertion Sort

Der Insertion Sort Algorithmus sortiert Elemente stabil und in-place. Dabei entnimmt er der unsortierten Eingabefolge ein beliebiges Element und fügt es an der richtigen Stelle in die Ausgabenfolge ein. Die eigentlich aufwendige Operation stellt das Verschieben der Elemente hinter dem neu eingefügten Element dar (wenn auf einem Array gearbeitet wird). Das Auffinden der richtigen Einfügeposition wird meist über eine binäre Suche implementiert.

**Theoretische Komplexität (Time/Space):**

| Average Case | Worst Case | Best Case | Space |
|---|---|---|---|
| **$O(N^2)$** | **$O(N^2)$** | **O(N)** | **O(1)** |

**Beispiel eines Best-Case Szenarios:**
Für den Insertion Sort gelten ähnliche Kriterien wie auch für den Bubble Sort. So tritt der Best-Case für den Insertion Sort Algorithmus ebenfalls dann ein, wenn die Elemente bereits sortiert sind.

Der Insertion Sort funktioniert also vor allem dann gut, wenn nur wenige Elemente an der falschen Stelle sind. Hat man beispielsweise eine Liste an bereits sortierten Elementen und nun kommt ein neues hinzu, das richtig einsortiert werden muss, dann würde sich der Insertion Sort Algorithmus in diesem Fall gut eignen.

**Beispiel eines Worst Case Szenarios:**
Auch der Worst-Case für den Insertion Sort ist ähnlich wie beim Bubble Sort, nämlich ein absteigend sortiertes Array, da jedes Element von seiner Ursprungsposition j bis auf die erste Arrayposition verschoben wird und dabei j − 1 Verschiebeoperationen nötig sind. Das entspricht somit im Worst-Case (wie auch bei Bubble Sort) ½ * (n*(n-1)) Operationen.

**Mögliche Einschränkungen bei der Anwendbarkeit:**
Auch die Einschränkungen bei der Anwendbarkeit des Insertion Sort Algorithmus decken sich ziemlich mit denen des Bubble Sort. Für beide Sortierverfahren gilt: Sie sind stabil und arbeiten in-place. Je nach Implementation kann der Insertion Sort jedoch geringere konstante Laufzeitfaktoren als der Bubble Sort haben.

## Quick Sort

Der Quick Sort Algorithmus sortiert Elemente schnell und rekursiv, ganz nach dem Prinzip „Divide & Conquer“, allerdings ist er nicht stabil. Zunächst wird ein sogenanntes „Pivotelement“ gewählt und die zu sortierende Liste in zwei Teillisten getrennt, wobei in die linke Teilliste alle Elemente kommen, die kleiner als das Pivotelement sind und in die rechte Teilliste alle Elemente die größer als das Pivotelement sind. Anschließend werden die Teillisten sortiert, indem das gleiche Prinzip wieder angewandt wird, das heißt, dass der Quick Sort Algorithmus auf beiden Teillisten ausgeführt wird (Rekursion). Wenn eine Teilliste der Länge eins oder null auftritt, so ist diese bereits sortiert und es erfolgt der Abbruch der Rekursion. Die Laufzeit (und auch der Speicherverbrauch) des Algorithmus hängt stark von der Wahl des Pivotelementes ab. Konkret hängt der Speicherverbrauch von der Rekursionstiefe ab.

**Theoretische Komplexität (Time/Space):**

| Average Case | Worst Case | Best Case | Space (Best + Average) | Space (Worst) |
|---|---|---|---|---|
| **O(N*log(N))** | **$O(N^2)$** | **O(N*log(N))** | **O(log(N))** | **O(N)** |

**Beispiel eines Best-Case Szenarios:**
Im Best-Case wird beim Quick Sort das Pivotelement so gewählt, dass die beiden entstehenden Teillisten etwa gleich groß sind. In der Praxis wird oft ein zufälliges Element als Pivotelement ausgewählt (randomisierter Quicksort), da die Wahrscheinlichkeit, dass der Worst-Case für den Quick Sort eintritt, somit relativ gering ist.

**Beispiel eines Worst Case Szenarios:**
Der Worst-Case beim Quick Sort tritt dann ein, wenn das Pivotelement so gewählt wird, dass es das größte oder das kleinste Element der Liste ist. Das ist beispielsweise dann der Fall, wenn als Pivotelement stets das Element am Ende der Liste gewählt wird und die zu sortierende Liste bereits sortiert vorliegt.

**Mögliche Einschränkungen bei der Anwendbarkeit:**
Obwohl es Algorithmen gibt, deren Laufzeiten auch im Worst Case durch O(N*log(N)) beschränkt sind (wie beispielsweise Heap Sort), wird in der Praxis oft Quick Sort eingesetzt, da der Worst Case bei Quick Sort nur sehr selten auftritt und der Algorithmus somit im mittleren Fall sogar schneller als Heap Sort ist, da die innerste Schleife von Quick Sort nur einige wenige, sehr einfache Operationen enthält.

Quick Sort steht ganz im Gegensatz zu Bubble Sort und Insertion Sort. Sind die Elemente beispielsweise mit einer hohen Wahrscheinlichkeit bereits sortiert, dann würde er sich in diesem konkreten Fall sogar weniger gut eignen als die anderen Varianten. In der Praxis ist das jedoch eher selten der Fall.

## Bucket Sort

Bucket Sort ist ein out-of-place Sortierverfahren, das für bestimmte Werte-Verteilungen eine Eingabe-Liste in linearer Zeit sortiert. Dabei werden die Elemente zunächst auf die Buckets aufgeteilt. Anschließend wird jeder Bucket mit einem weiteren Sortierverfahren (wie beispielsweise Merge Sort oder Insertion Sort) sortiert. Abschließend wird der Inhalt der sortierten Buckets konkateniert, also in anderen Worten zusammengeführt, ohne die Reihenfolge der Elemente zu verändern.

**Theoretische Komplexität (Time/Space):**

| Average Case | Worst Case | Best Case | Space |
|---|---|---|---|
| **O(N)** | **O($N^2$)** | **O(N)** | **O(N)** |

Anmerkung: Konkrete Laufzeit beträgt $\mathcal{O}(n) + \sum_{i=0}^{n-1} \mathcal{O}(l_i \log l_i)$ wobei wobei $l_i$ die Anzahl der Elemente im i-ten Bucket bezeichnet.

**Beispiel eines Best-Case Szenarios:**
Der Best-Case für den Bucket Sort Algorithmus tritt ein, wenn die Elemente gleichmäßig in den Buckets verteilt sind und nahezu die gleiche Anzahl von Elementen in jedem Eimer vorhanden ist. Die Komplexität wird noch besser, wenn die Elemente in den Eimern bereits sortiert sind. In diesem Fall würde sich zum Beispiel Insertion Sort sehr gut als Algorithmus zum Sortieren der Elemente in den Buckets eignen. Die Komplexität beträgt dann O(N) oder genauer gesagt O(N+k), wobei O(N) die Komplexität für das Erstellen der Buckets darstellt und O(k) die Komplexität für das Sortieren der Elemente in den Buckets.

**Beispiel eines Worst Case Szenarios:**
Elemente die nah aneinander sind, werden sehr wahrscheinlich im selben Bucket platziert. Das kann dazu führen, dass einige Buckets mehr Elemente als andere aufweisen. Dadurch hängt die Komplexität vom Sortieralgorithmus ab, der zum Sortieren der Elemente des Buckets verwendet wird. Die Komplexität erreicht beispielsweise dann im Worst-Case O($N^2$), wenn zum Sortieren der Elemente Insertion Sort verwendet wird und die Elemente in umgekehrter Reihenfolge vorliegen (siehe Worst Case von Insertion Sort).

**Mögliche Einschränkungen bei der Anwendbarkeit:**
Bucket Sort eignet sich vor allem dann gut, wenn die Eingabewerte gleichmäßig über einen Bereich

verteilt sind oder auch um Gleitkommazahlen (floating point) zu sortieren. Bei anderen Werte-Verteilungen kann die Laufzeit von Bucket Sort jedoch von der Laufzeit des Sortier-Algorithmus dominiert werden, der zur Sortierung der Elemente in den Buckets verwendet wird.

## Gewählte Variationen

1. **Variante 1 (Basisvariante)**
    a. Gesamtes Datenarray passt in den CPU-Cache
    b. Array Eintrag ist ein simpler Integer
    c. Array enthält direkt die konkreten Inhalte (Array of Stucts/Primitives)
2. **Variante 2**
    **a. Array ist zu groß für den Cache**
    b. Array Eintrag ist ein simpler Integer
    c. Array enthält direkt die konkreten Inhalte (Array of Stucts/Primitives)
3. **Variante 3**
    a. Gesamtes Datenarray passt in den CPU-Cache
    **b. Array Eintrag ist ein komplexes Struct mit mehreren Variablen (und einem Integer Key)**
    c. Array enthält direkt die konkreten Inhalte (Array of Stucts/Primitives)
4. **Variante 4**
    a. Gesamtes Datenarray passt in den CPU-Cache
    b. Array Eintrag ist ein simpler Integer
    **c. Jeder einzelne Inhalt wird separat erzeugt(allokiert). Das Array enthält nur Referenzen oder Pointer auf diese Inhalte.**
5. **Variante 5**
    **a. Array ist zu groß für den Cache**
    **b. Array Eintrag ist ein komplexes Struct mit mehreren Variablen (und einem Integer Key)**
    c. Array enthält direkt die konkreten Inhalte (Array of Stucts/Primitives)
6. **Variante 6**
    **a. Array ist zu groß für den Cache**
    b. Array Eintrag ist ein simpler Integer
    **c. Jeder einzelne Inhalt wird separat erzeugt(allokiert). Das Array enthält nur Referenzen oder Pointer auf diese Inhalte.**
7. **Variante 7**
    a. Gesamtes Datenarray passt in den CPU-Cache
    **b. Array Eintrag ist ein komplexes Struct mit mehreren Variablen (und einem Integer Key)**
    **c. Jeder einzelne Inhalt wird separat erzeugt(allokiert). Das Array enthält nur Referenzen oder Pointer auf diese Inhalte.**
8. **Variante 8**
    **a. Array ist zu groß für den Cache**
    **b. Array Eintrag ist ein komplexes Struct mit mehreren Variablen (und einem Integer Key)**
    **c. Jeder einzelne Inhalt wird separat erzeugt(allokiert). Das Array enthält nur Referenzen oder Pointer auf diese Inhalte.**

Alle Variationen wurden in der Programmiersprache C++ implementiert. Fett gedruckt sind jeweils die Unterschiede zur Basisvariante.

## Benchmark Maschine

**CPU: AMD Ryzen 5 5600x**

- Kerne: 6
- Threads: 12
- Basistakt: 3,7 GHz
- Turbotakt: 4,6 Ghz
- L1 Cache: 6x 64KB
- L2 Cache: 6x 512KB
- L3 Cache: 32MB

**RAM: G.Skill TridentZ RGB**

- Technologie: DDR4 SDRAM
- Kapazität: 32 GB
- Module: 2
- Modulgröße: 16GB
- Geschwindigkeit: 3600 MHz
- Latenzzeiten: CL16 (16-19-19-39)
- Datenintegrität: Non-ECC

## Verwendete Compiler und Compilereinstellungen

**Compiler:** g++

**Cpp Standard:** C++17

**Optimierung:** -O3

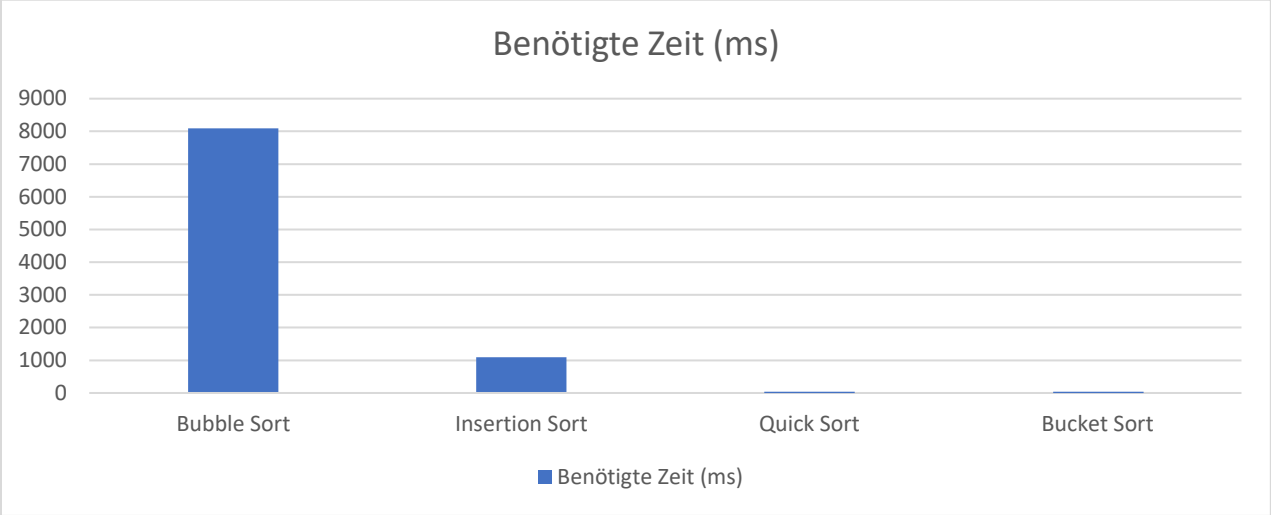
## Gemessene Laufzeiten im Vergleich

Gemessen wurden die folgenden Werte mit je **100.000** Datensätzen bei den Varianten, wo das Array klein genug sein soll, um in den Cache zu passen und mit je **10.000.000** Datensätzen bei den Varianten, wo das Array zu groß ist, um in den Cache zu passen. Insgesamt wurden je **50 Durchläufe** gemessen und dann der Durchschnitt der Ergebnisse berechnet.

| Algorithmus | Bubble Sort | Insertion Sort | Quick Sort | Bucket Sort |
|---|---|---|---|---|
| Variante 1 | 8s 95ms | 1s 91ms | 4ms | 6ms |
| Variante 2 | >22h* | >3h* | 589ms | 2s 531ms |
| Variante 3 | 13s 103ms | 15s 937ms | 5ms | 7ms |
| Variante 4 | 15s 222ms | 6s 176ms | 5ms | 7ms |
| Variante 5 | >36h* | >44h* | 791ms | 2s 671ms |
| Variante 6 | >42h* | >17h* | 1s 301ms | 2s 504ms |
| Variante 7 | 16s 597ms | 6s 396ms | 5ms | 7ms |
| Variante 8 | >46h* | >17h* | 1s 327ms | 2s 322ms |

*Abschätzung basierend auf der Performance, die beim Testen mit einem kleineren Array (100.000 Datensätzen statt 10.000.000) gemessen wurde

### Visualisierung der gemessenen Zeit (Variante 1):




## Interpretation der Ergebnisse

Über alle getesteten Varianten hinweg lässt sich klar der Trend erkennen, dass die gemessenen Laufzeiten meist dem Average Case der theoretischen Laufzeit entsprechen. So benötigen Bubble Sort und Insertion Sort in allen Fällen deutlich länger als der Quick Sort und der Bucket Sort, aufgrund des quadratischen Laufzeitverhaltens.

Allerdings ist der Bucket Sort nie schneller als der Quick Sort, was darauf zurückzuführen ist, dass die Eingabedaten völlig zufällig generiert (und damit nicht zwingend gleichverteilt) waren. Dadurch ist die Laufzeit des Bucket Sort von dem Laufzeitverhalten des Algorithmus abhängig, der zum Sortieren der Elemente in den Buckets verwendet wird. In unserem Fall haben wir std::sort gewählt, dieser hat eine Average Case Laufzeit von O(N*log(N)), was auch der Average Case Laufzeit von Quick Sort entspricht.

Es lässt sich auf jeden Fall klar erkennen, dass der Quick Sort Algorithmus cache-effizienter als der Bucket Sort Algorithmus ist, was mit der theoretischen Space-Complexity übereinstimmt. Theoretisch sollten der Bubble Sort und der Insertion Sort Algorithmus ebenfalls cache-effizient sein, allerdings ist dies in den gemessenen Zeiten nicht merkbar, da beide Algorithmen quadratisches Laufzeitverhalten im Average Case besitzen und somit die Laufzeit förmlich „explodiert“, wenn so viele Datensätze verwendet werden müssen, dass sie nicht mehr in den Cache passen. Daher konnten hier auch nur Abschätzungen basierend auf Messungen mit einer geringeren Anzahl an Datensätzen durchgeführt werden, da sie ansonsten viel zu viel Zeit benötigt hätten.

Eine weitere interessante Beobachtung ist zudem, dass alle Algorithmen zwar etwas länger brauchen (im Vergleich zur Basisvariante 1), wenn ein Array aus Structs, das mehrere Integer Werte enthält (Variante 3) sortiert werden muss, anstatt einem Array, das direkt die Integer Werte enthält, sowie wenn das Array nur Pointer zu den Objekten/Primitiven speichert (Variante 4), anstatt sie direkt abzuspeichern. Bei einer Kombination dieser beiden Variationen (Variante 7) benötigen die Algorithmen im Durchschnitt annähernd gleich lange, wie wenn nur jeweils eine der beiden Variationen verwendet wurde.

Ebenfalls interessant ist, dass der Insertion Sort durchschnittlich schneller als der Bubble Sort ist. Einzig wenn die Array-Einträge komplexe Structs anstatt simpler Integer Werte sind, scheint der Insertion Sort etwas langsamer zu laufen als der Bubble Sort, was vor allem in den Varianten 3 und 5 ersichtlich ist.