Sortieren von Daten - Sortieralgorithmen

Das grundlegende Problem der Verarbeitung großer Datenmengen ist deren Sortieren und damit Strukturieren. Es wird geschätzt, dass ca. 25% der kommerziellen Rechenzeit für Sortiervorgänge benötigt wird.

Aufgabe: Überlegen Sie sich gesellschaftlich, wirtschaftlich relevante Beispiele, bei denen das Sortieren großer Datenmengen und die Basis des Handelns darstellt und viel Zeit/Kosten bindet.

Vorbetrachtungen:

- Festlegung: Wir sortieren immer vom kleinsten zum größten Wert.

- Die Effizienz der Sortieralgorithmen ist in den vielem Fällen vom **Ausgangszustand** abhängig
Best Case - sie sind bereits komplett sortiert, **Worst Case -** sie sind komplett unsortiert, **Average Case** – der Normalfall, die Daten sind weder sortiert noch unsortiert.

- Es gibt unzählige Sortierverfahren: einfache und komplizierte; gute und schlechte; Verfahren, die nur in bestimmten Konstellationen oder bei kleiner Datenmenge gut arbeiten.

- Am häufigsten arbeiten Sortieralgorithmen **vergleichsbasiert**. Z.B. Bubble Sort, Insertion Sort, Selection Sort, Quicksort, Heapsort)

Das erfolgt vereinfacht immer so:

solange bis es nichts mehr zu tauschen gibt {durchlaufe die Liste mit der festgelegten Strategie, vergleiche und vertausche Werte. }

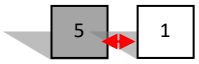
Eine Liste wird also mehrfach durchlaufen und dabei zunehmend besser sortiert. Oft wird diese dabei
- in einen bereits sortierten Teil, der mit jeder Sortierung länger wird und
- in einen noch unsortierten Teil, der bei jedem Durchlauf kürzer wird, zerlegt.

- Welches das jeweils effektivste Verfahren ist, hängt vom **Vor-Sortierungsgrad**, der **Anzahl der Elemente** sowie vom verwendeten **Sortieralgorithmus** ab.

Die Komplexität Θ eines Sortieralgorithmus ergibt sich aus der Anzahl der notwendigen **Vergleiche**, **Vertauschungen** und **Sortier-Durchläufe**, um eine Datenmenge vollständig zu sortieren. Dabei ist ein Vergleich rechentechnisch viel weniger aufwändig als ein Tauschvorgang.

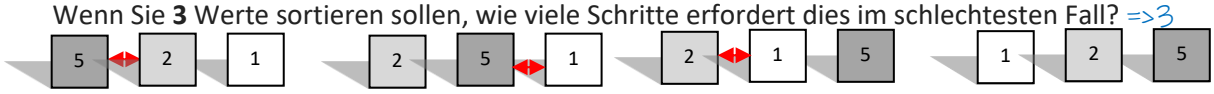
Wir unterscheiden **linearen**, **logarithmischen** (-> geringer) oder **quadratische.** (-> großen) Sortieraufwand.

Wenn Sie **2** Werte sortieren sollen, wie viele Schritte erfordert dies im schlechtesten Fall? Na logo. =>1.

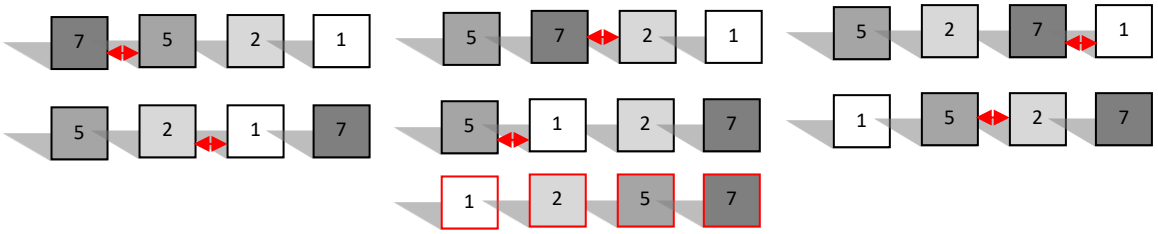


Tauschvorgang

Wenn Sie **3** Werte sortieren sollen, wie viele Schritte erfordert dies im schlechtesten Fall? =>3



Wenn Sie **4** Werte sortieren sollen, wie viele Schritte erfordert dies im schlechtesten Fall? =>6



Die Zahl der Tauschvorgänge nimmt mit der Anzahl der Elemente rasch zu und erfordert aus Zeitgründen vom „Programmierer“ den Sortieralgorithmus geschickt zu planen.

Ein Algorithmus kann auch nicht, wie ein Mensch eine bereits sortierte Menge erkennen, sondern sortiert trotzdem weiter, z.B. indem er weiter vergleicht und dann eben bestenfalls nichts tut.

Vergleichsbasierte Verfahren

1. Bad Algorithm → zufälliges Sortieren

Der Bad Algorithm ist die wohl schlechteste Art und Weise eine Liste zu sortieren. Die Eingabe-Liste wird bei diesem Verfahren immer wieder so lange zufällig gemischt, bis diese zufällig korrekt sortiert ist.

Im engsten Sinne ist dieses Verfahren gar kein Algorithmus, da nicht sicher ist, dass dieser am Ende eine sortierte Menge liefert. Werden 8 Elemente sortiert ist der Algorithmus normal nach weniger als 0,1 Sekunden abgeschlossen. Verdoppelt man die Anzahl der Elemente (also 16 Elemente) war nach 8 Stunden noch kein Ergebnis gefunden.

Komplexitätsbetrachtung: (=> Sortieraufwand):

Wie viele **Durchgänge** brauchen wir bei diesem Verfahren, um **n-Elemente** zu sortieren?

Es sind **zufällig viele Durchgänge** erforderlich, um die Liste zu sortieren.

Wie viele **Vergleiche** müssten stattfinden, um bei diesem Verfahren die Liste komplett zu sortieren?

pro Durchgang (n-1) → ca. $(n-1)^D$ **Vergleiche** sind notwendig

Wie viele Tauschvorgänge werden im **worst case** gebraucht?

unbestimmte Anzahl **Tauschvorgänge**

Wie viele Tauschvorgänge werden im **best case** gebraucht?

keine **Tauschvorgänge.**

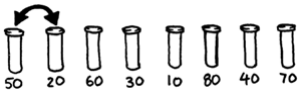
Wovon ist der Aufwand **abhängig**?

Anzahl der Werte, Zufall

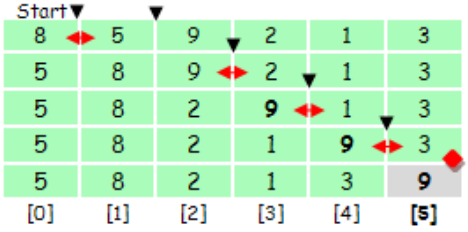
2. Bubblesort → Sortieren durch direktes Tauschen

Start: am Listenanfang -> durchlaufe die Liste:
- Ist Element größer als rechter Nachbar? -> vertausche die Elemente
- Listenende erreicht? *(-> letztes Element = größtes Element)* -> Liste ist wird um 1 verkürzt *(letztes Element wird weggelassen)*
- Solange die Liste länger als 1 ist -> beginne wieder bei Start.

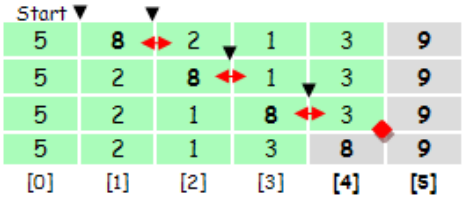
==> der größte Wert wandert an das Ende der Liste. *(deshalb „bubbles")*



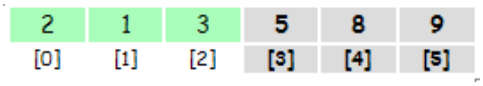
1. Durchlauf



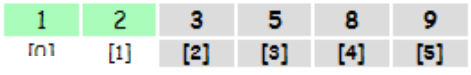
2. Durchlauf



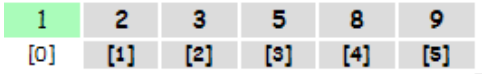
3. Durchlauf



4. Durchlauf



5. Durchlauf



Komplexitätsbetrachtung:

Wie viele **Durchgänge** brauchen wir bei diesem Verfahren, um n-Elemente zu sortieren?

Es sind **n-1 Durchgänge** erforderlich, um die Liste zu sortieren.

Wie viele Vergleiche müssten stattfinden, um bei diesem Verfahren die Liste komplett zu sortieren?

(n-1)+(n-2)+(n-3)+...2+1= ½ n(n-1) → ca. **½ n^2 Vergleiche** sind notwendig

$O(n^2)$

quadratisch

Wie viele Tauschvorgänge werden im **worst case** gebraucht?

Genauso viele wie Vergleiche → ca. **½ n² Tauschvorgänge** sind notwendig

Wie viele Tauschvorgänge werden im **best case** gebraucht?

Keine Tauschvorgänge.

Wovon ist der Aufwand **abhängig**?

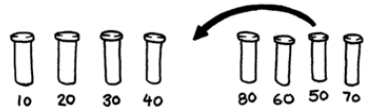
Von der Anzahl der Werte und der **Vorsortierung,** da dann weniger Tauschvorgänge durgeführt werden**.**

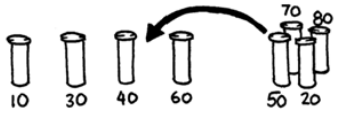
3. Selectionsort → Sortieren durch Auswahl des Minimums

Start: am Listenanfang -> durchlaufe die Liste:
- finde durch mehrfaches Vergleichen das Minimum der Liste
- tausche das Minimum mit dem ersten Element → Liste ist wird um 1 verkürzt *(erstes Element weglassen)*
- solange die Liste länger als 1 ist -> beginne bei Start

[0]	[1]	[2]	[3]	[4]	[5]	
8	5	1	2	9	3	1. Durchlauf
1	5	8	2	9	3	2. Durchlauf
1	2	8	5	9	3	3. Durchlauf
1	2	3	5	9	8	4. Durchlauf
1	2	3	5	9	8	5. Durchlauf
1	2	3	5	8	9	6. Durchlauf

bedeutet: Werte tauschen.





Komplexitätsbetrachtung:

Wie viele **Durchgänge** brauchen wir bei diesem Verfahren, um n-Elemente zu sortieren?

Es sind **n-1 Durchgänge** erforderlich, um die Liste zu sortieren.

Wie viele Vergleiche müssten stattfinden, um bei diesem Verfahren die Liste komplett zu sortieren?

(n-1)+(n-2)+(n-3)+...2+1= ½ n(n-1) →ca. **½ n² Vergleiche** sind notwendig

Wie viele Tauschvorgänge werden im **worst case** gebraucht?

→**(n-1) Tauschvorgänge** sind notwendig

Wie viele Tauschvorgänge werden im **best case** gebraucht?

Keine Tauschvorgänge.

Wovon ist der Aufwand **abhängig**?

Nur von der Anzahl der Werte, da eine evtl. Vorsortierung keine Rolle spielt.

$O(n^2)$

quadratisch

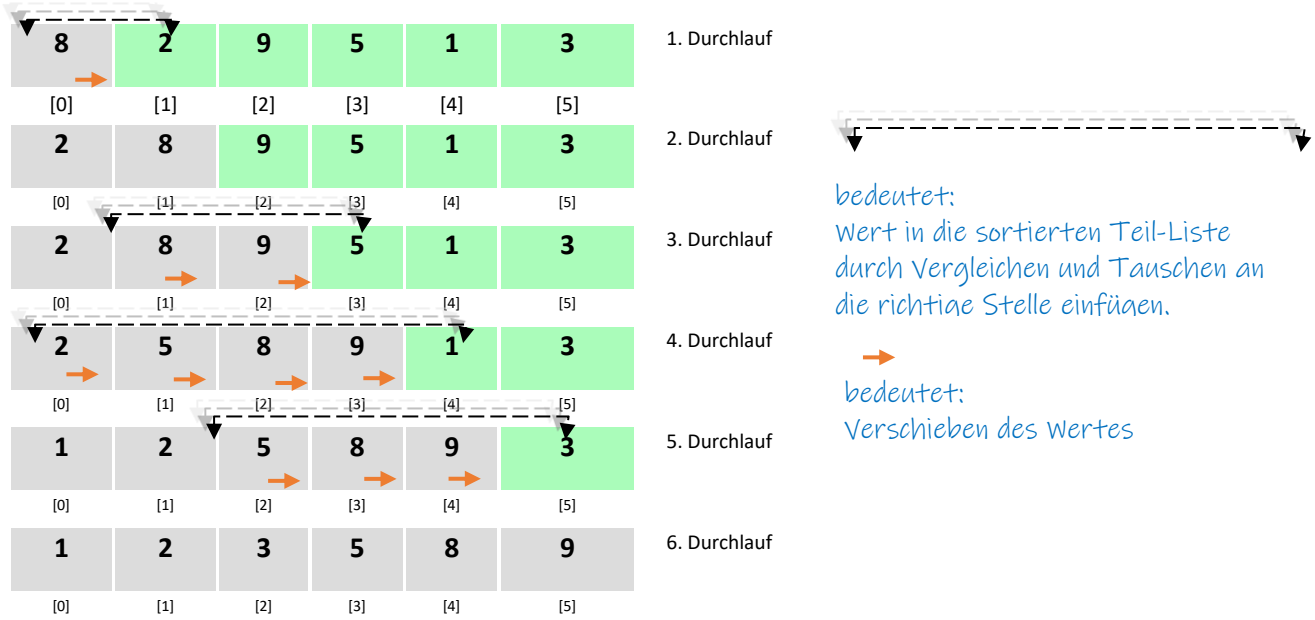
4. Insertionsort → Sortieren durch direktes Einfügen

Den Algorithmus “Sortieren durch Einfügen” benutzen die meisten Menschen intuitiv, wenn sie beim Kartenspielen ihre Karten in der Hand sortieren bzw. aufnehmen.

- Start: am Listenanfang -> index x=1, durchlaufe die Liste:
- Ist das x. Element kleiner als linker Nachbar bei x-1? → tausche Elemente,
- x => x+1; -> es entsteht eine sortierte linke Teil-Liste,
- Start: x. Element in die sortierte Teil-Liste an die richtige Stelle einfügen, d.h. solange die Zahl links von dir größer ist tausche die Elemente.
- solange x kleiner als die Listenlänge ist → beginne bei Start

Bei diesem Verfahren ist besonders aufwändig, dass nach dem Einfügen eines Wertes im sortierten Bereich alle darauffolgenden Elemente verschoben werden müssen.

Der sortierte Teil besteht zu Beginn aus dem ersten Element der Liste.



Komplexitätsbetrachtung:

Wie viele **Durchgänge** brauchen wir bei diesem Verfahren, um die Liste komplett zu sortieren?

Es sind **n-1 Durchgänge** erforderlich, um die Liste zu sortieren.

Wie viele Vergleiche müssten stattfinden, um bei diesem Verfahren die Liste komplett zu sortieren?

worst case (n-1)+(n-2)+(n-3)+...2+1= ½ n(n-1) →ca. ½ n^2 Vergleiche sind notwendig

best case (schon sortierte Reihe) nur (n-1) Vergleiche sind notwendig

average case (½ n^2+n)/2 → ca**. ¼ n^2 Vergleiche** sind im **Durchschnitt** notwendig.

$O(n^2)$

quadratisch

Wie viele Tauschvorgänge werden im **worst case** gebraucht?

(n-1)+(n-2)+(n-3)+...2+1= ½ n(n-1) →ca. **½ n^2 Tauschvorgänge** sind notwendig

Wie viele Tauschvorgänge werden im **best case** gebraucht?

keine Tauschvorgänge

Wovon ist der Sortieraufwand **abhängig**?

Der Aufwand hängt sehr **stark von** der Anordnung (**Vorsortierung**) ab. Je gründlicher der Datenbestand bereits sortiert ist, desto geringer wird der Aufwand, im besten Falle mit dem minimalsten Aufwand.

Nicht-vergleichsbasierte Sortierverfahren

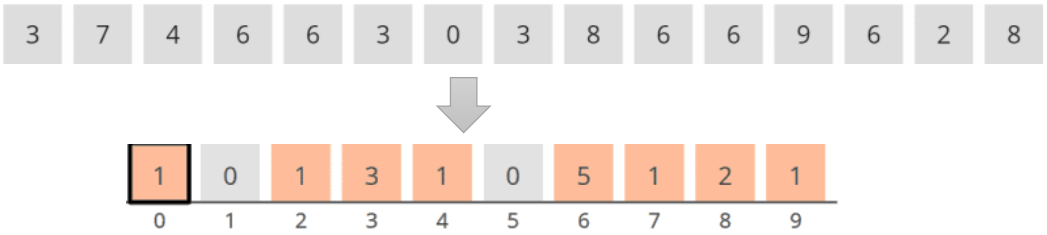
- basieren nicht auf direktem Vergleich zweier Elemente (Countingsort, Bucketsort, Radixsort)

- Unter Ausnutzung bestimmter Eigenschaften der zu sortierenden Daten lassen sich in einigen Fällen Sortierverfahren entwickeln, die schneller als optimale Sortierverfahren sind

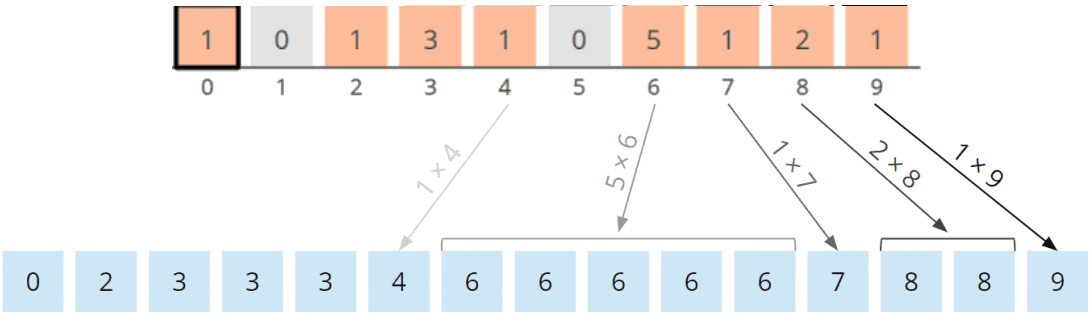
5. Counting Sort → Sortieren durch Zählen

Anstatt die Elemente zu vergleichen, wird bei Counting Sort gezählt, wie oft welche Elemente vorkommen. Eine vereinfachte Form von Counting Sort kann angewendet werden, wenn Zahlen (z. B. *int*) sortiert werden sollen

Phase 1: Durchlaufen der Reihe und die Anzahl der Elemente zählen und in einer indizierten Liste speichern.



Phase 2: Durchlaufen indizierten Liste und den Index so oft in eine leere Kopie der Originalreihe schreiben, wie der Listenwert angibt.



Komplexitätsbetrachtungen:

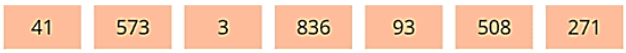
Die Anzahl der Operationen ist unabhängig von der Ausgangsreihenfolge der Elemente, aber natürlich abhängig von der Anzahl der Daten.

Der Aufwand entspricht der Zeitkomplexität **O(n + k)** steigt also linear mit der Anzahl der zu sortierenden Elemente n und der Größe k des Zahlenraums.

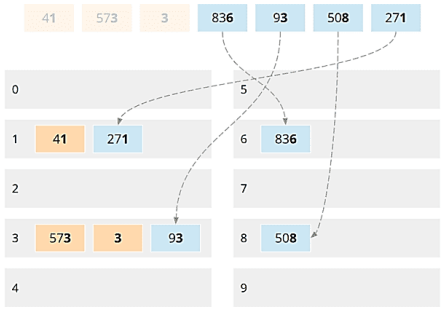
6. **Radix Sort → Sortieren durch Partitionieren**

Bei Radix Sort sortieren wir die Zahlen Ziffer für Ziffer – und nicht, indem wir zwei Zahlen miteinander vergleichen.

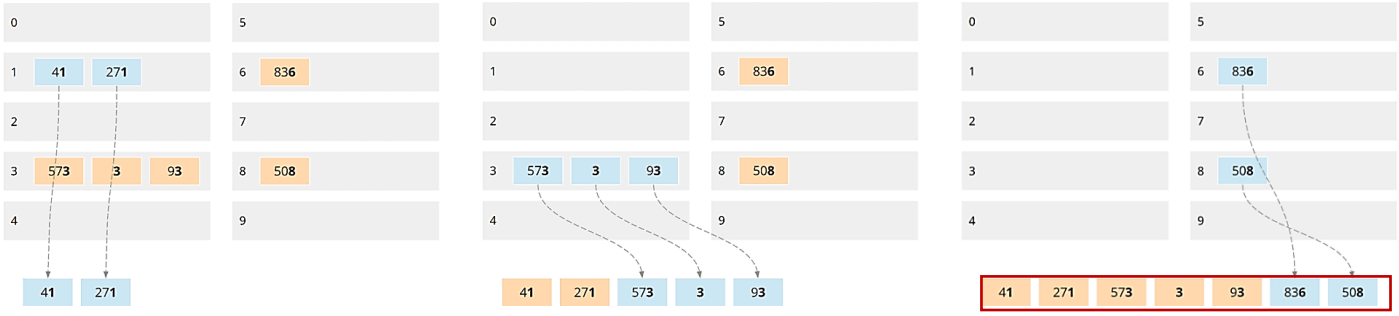
Wir betrachten zu Beginn ausschließlich die **Einerstelle** (es gibt auch Radix Sort-Varianten, bei denen man bei der *ersten* Ziffer beginnt, aber dass ist komplizierter).
Wir sortieren die Zahlen in zwei Phasen: einer Partitionierungsphase und einer Sammelphase.



Phase 1 → Partitionierungsphase **Einer**:
Wir legen zehn sogenannte "Buckets" an, bezeichnet mit "0" bis "9". Auf diese verteilen wir die Zahlen entsprechend ihrer jeweils letzten Ziffer.



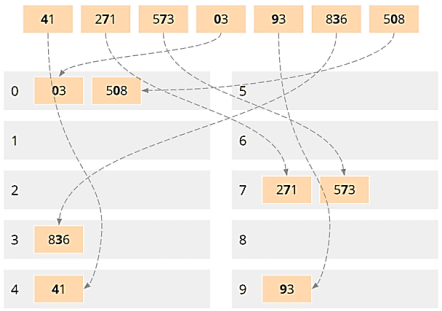
Phase 2 → Sammelphase **Einer**:
Wir sammeln die Zahlen, Bucket für Bucket, in aufsteigender Reihenfolge und innerhalb der Buckets von links nach rechts in eine neue Liste, bis alle Buckets wieder leer sind.



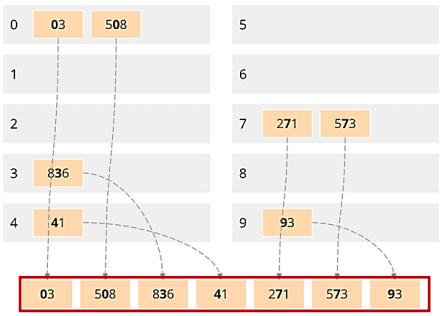
In dieser neuen Liste sind die Zahlen nach ihrer letzten Ziffer aufsteigend sortiert: 1, 1, 3, 3, 3, 6, 8

Wir wiederholen die Partitionierungs- und Sammelphase für die **Zehnerstelle**.

Phase 1 → Partitionierungsphase **Zehner**:

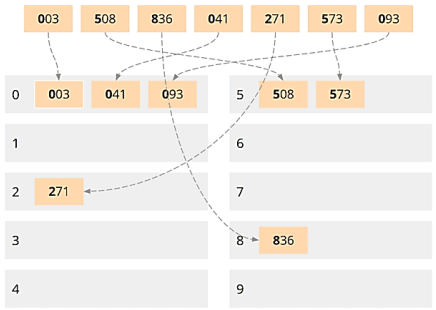


Phase 2 → Sammelphase **Zehner**:

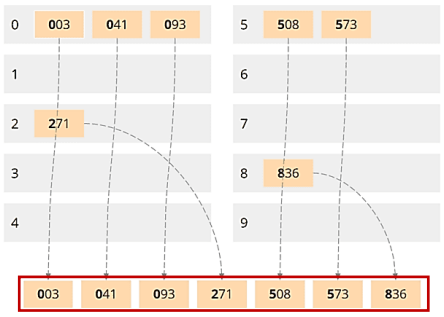


Wir wiederholen die Partitionierungs- und Sammelphase für die **Hunderter-Stelle**.

Phase 1 → Partitionierungsphase **Hunderter**:



Phase 2 → Sammelphase **Hunderter**:



Nach der dritten und letzten Sammelphase sind die Zahlen nun vollständig sortiert.

3 41 93 271 508 573 836

Komplexitätsbetrachtungen:

Wir verwenden die folgenden Variablen:
n = die Anzahl der zu sortierenden Elemente, *k* = Anzahl der Stellen der zu sortierenden Elemente,
Der Algorithmus iteriert über *k* Stellen; für jede Stelle betreibt er den folgenden Aufwand:

Er legt *10* Buckets an. Der Aufwand dafür ist jeweils konstant.
Er iteriert über alle *n* Elemente, um diese in die Buckets einzusortieren. Der Aufwand für die Berechnung der Bucket-Nummer und für das Einfügen in den Bucket ist konstant.
Er iteriert über *10* Buckets und entnimmt diesen wieder insgesamt *n* Elemente. Der Aufwand für jeden dieser Schritte ist wiederum konstant.

Konstante Aufwände vernachlässigen wir bei der Bestimmung der Zeitkomplexität. Somit ergibt sich:

Die Zeitkomplexität für Radix Sort ist: $O(k \cdot (10 + n)) = O(n)$ →linear

Der Aufwand ist unabhängig davon, wie die Eingabezahlen angeordnet sind. Ob diese zufällig verteilt oder bereits vorsortiert sind, macht keinen Unterschied für den Algorithmus. Best case, average case und worst case sind also identisch.

Zusammenfassung Zeit- und Platzkomplexitäten:

Algorithmus	Zeit best case	Zeit avg. case	Zeit worst case	Platz	Stabil
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Ja
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Nein
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Ja
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(n + k)$	Ja
Radix Sort	$O(k \cdot (10 + n))$	$O(k \cdot (10 + n))$	$O(k \cdot (10 + n))$	$O(10 + n)$	Ja