

Prof. Dr. Claudia Müller-Birn, Barry Linnert

Objektorientierte Programmierung, SoSe 17

Übung 06

TutorIn: Thierry Meurers

Tutorium 10

Stefaan Hessmann, Jaap Pedersen, Mark Niehues

13. Juni 2017

1 Aufgabe 1

- a) Die von Python bereitgestellte `sorted()`-Funktion nutzt einen hybriden Sortier-Algorithmus (Mix aus Mergesort und Insertionsort) namens Timsort.
- b) Timsort ist ein Hybrid, der auf Mergesort und Insertionsort basiert. Trotz der theoretisch günstigeren durchschnittlichen Laufzeit des Mergesort gibt es einige Fälle in denen Insertionsort praktisch schneller arbeitet und häufig Optimierungspotential für die zu sortierenden Daten, die ein effektiveres Sortieren zulassen.
- Da Mergesort erst ab einer bestimmten Listenlänge (in der Python Implementierung $N = 64$) schneller arbeitet als Insertionsort, wird bei einem $N < 64$ schlicht Insertionsort angewandt.
 - Ansonsten wird zunächst nach bereits nach sortierten oder umgekehrt sortierten Teilfolgen gesucht und diese zu sogenannten *runs* zusammengefasst.
 - Die *runs* werden während des Durchsuchens bereits gemerched, um die schnellen Zugriffszeiten auf Daten, die erst kürzlich gecached wurden auszunutzen. (In der Python Implementierung werden aus praktischen Erkenntnissen immer nur 3 *runs* im Cache behalten, die nach bestimmten Regeln zur Laufzeit gemerched werden, um Stabilität und eine Begrenzung des notwendigen Caches - also quasi In-Place - beim Mergen zu garantieren.)
 - Danach werden die *runs* mittels Insertionsort auf eine bestimmte Länge *minrun* aufgefüllt. *Minrun* wird so gewählt, dass die Gesamtlänge der Daten geteilt durch *minrun* einer Zweierpotenz entspricht, da mergen von derart balancierten vorsortierten Daten am effektivsten funktioniert.
 - Abschließend kommt es dann zum eigentlichen Mergeprozess, wobei als weitere Optimierung sgn. *Galloping* angewandt wird. Hierbei werden Listeneinträge übersprungen, falls beim Vergleich zweier Teillisten häufig dieselbe Seite "gewinnt".

Durch diese Optimierungen ist - neben praktischen performance Vorteilen - die best-case Laufzeit des Timsort die des Insertionsort, nämlich $\mathcal{O}(n)$.

- d)** Im Vergleich zu den anderen vergleichsbasierten Sortieralgorithmen, die in der Vorlesung besprochen wurden, erzielt Timsort immer die bestmögliche Laufzeit für best-, average- und worst-case. In der Tabelle befindet sich kein anderer Algorithmus der in den drei Fällen die beste Laufzeit liefert. Timsort ist außerdem ein stabiler Sortieralgorithmus, sodass Elemente mit gleichem Schlüssel nicht vertauscht werden.

Aufgrund seiner Laufzeit und Stabilität eignet sich Timsort gut als Standard-Sortieralgorithmus für die Python Umgebung, da die Sortierung maximal schnell abläuft und zusätzlich die Sortierung von gleichen Schlüsseln beibehalten wird. Diese Eigenschaft ist besonders beim Sortieren nach mehreren Schlüsseln (beispielsweise Tabellen) notwendig.

Algorithmus	best	average	worst	in-place	stabil
Selection Sort	n^2	n^2	n^2	Ja	Nein
Insertion Sort	n	n^2	n^2	Ja	Ja
Bubble Sort	n	n^2	n^2	Ja	Ja
Quick Sort	$n\log(n)$	$n\log(n)$	n^2	Nein	Nein
Merge Sort	$n\log(n)$	$n\log(n)$	$n\log(n)$	Nein	Ja
Heap Sort	$n\log(n)$	$n\log(n)$	$n\log(n)$	Ja	Nein
Timsort	n	$n\log(n)$	$n\log(n)$	Nein	Ja

Tabelle 1: Vergleichsbasierte Sortieralgorithmen