Freie Universität Berlin

Prof. Dr. Claudia Müller-Birn, Barry Linnert

Objektorientierte Programmierung, SoSe 17

Übung 06

TutorIn: Thierry Meurers
Tutorium 10

Stefaan Hessmann, Jaap Pedersen, Mark Niehues

14. 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 Hybird, 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 effektieveres 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 Lauftzeit gemerched werden, um Stabilität und eine Begrenzung des notwendingen Caches - wenn auch nicht In-Place - beim Mergen zu garantieren.)
- Danach werden die *runs* mittles Insertionsort auf eine bestimmte Länge *minrun* aufgefüllt. *Minrun* wird so gewählt, dass die Gesamtlänge der Daten geteilt durch *minrun* einer Zweierpotens entspricht, da mergen von derart balancierten vorsortierten Daten am effektievsten 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 diesselbe Seite "gewinnt".

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

c) Über die Python Documentation der sort() Funktion stößt man auf die folgende Erklärung: http://svn.python.org/projects/python/trunk/Objects/listsort.txt

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	$nlog(n)$	$nlog(n)$	n^2	Ja	Nein
Merge Sort	$nlog(n)$	$nlog(n)$	$nlog(n)$	Nein	Ja
Heap Sort	$nlog(n)$	$nlog(n)$	$nlog(n)$	Ja	Nein
Timsort	n	$nlog(n)$	$nlog(n)$	Nein	Ja

Tabelle 1: Vergleichsbasierte Sortieralgorithmen

2 Aufgabe 2

Listing 1: Iteratives Quicksortverfahren

```
r"""
Iterative Version of Quicksort

inspired by:
http://codexpi.com/quicksort-python-iterative-recursive-implementations/

"""
import random

def quicksortIterative(liste):
    #initialize left and right limiting indices to sort list
    left = 0
    right = len(liste)-1
    #holds limiting indeces of seperated sublists
    temp_stack = []
    temp_stack.append((left,right))

    # Main loop to pop and push items until stack is empty and list is sorted
    while temp_stack:
        pos = temp_stack.pop()
        right, left = pos[1], pos[0]
        #perform sorting of sublist
        piv_idx = partition(liste, left, right)
        # If there are items in the left of the pivot push them to the stack
        if piv_idx - 1 > left:
            temp_stack.append((left, piv_idx - 1))
        # If there are items in the right of the pivot push them to the stack
        if piv_idx + 1 < right:
            temp_stack.append((piv_idx + 1, right))


def partition(list_, left, right):
    """
    Partition method
    """
```

```
36      # use the left item of the list as pivot
37      piv = list_[left]
38      i = left + 1
39      j = right

41      #loop over elements and exchange bigger and smaller elements
42      while True:
43          while i <= j and list_[i] <= piv:
44              i += 1
45          while j >= i and list_[j] >= piv:
46              j -= 1
47          if j <= i:
48              break
49          # Exchange items
50          list_[i], list_[j] = list_[j], list_[i]
51      # Exchange pivot to the rightest position of the left sublist
52      list_[left], list_[j] = list_[j], list_[left]
53      #return index of the pivot element
54      return j

56  if __name__ == '__main__':
57      #create random list of integer
58      n = int(input("How many items shall be in your list?:\n"))

60      list = [random.randint(0,30) for i in range(0,n)]
61      print("your randomly created list:")
62      print(list)
63      quicksortIterative(list)
64      print("your sorted list:")
65      print(list)
```