Sorting

Data Structures and Algorithms for Computational Linguistics III
(ISCL-BA-07)

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2021/22

version: 58258b @2021-11-09

Why study sorting

- Sorting is one of the most studied (and common) problems in computing
- It is important to understand strengths and weaknesses of algorithms for sorting
- Many problems look like sorting. Learning sorting algorithms will help you solve others
- Available implementations are highly optimized (we are not just talking about asymptotic performance guarantees)
- In some (rare) cases, implementing your own sorting algorithm may be beneficial

Bubble sort

- We start with an ‘educational’ sorting algorithm
- Bubble sort is easy to understand, but performs bad – not used in practice
- We start from bubble sort, and see the improvements over it
- The idea is simple:
  - compare first two elements, swap if not in order
  - shift and compare the next two elements, again swap if needed
  - when you reach to the end, repeat the process from the beginning unless there were no swaps in the last iteration

Bubble sort

demonstration

89 | 67 | 88 | 12 | 72 | 76 | 93 | 57

```
swapped = True
n = len(seq)
while swapped:
  swapped = False
  for i in range(n - 1):
    if seq[i] > seq[i + 1]:
      seq[i], seq[i + 1]\
          = seq[i + 1], seq[i]
      swapped = True
```


Bubble sort

summary

- Worst case: $O(n^2)$
  $O(n^2)$ comparisons, $O(n^2)$ swaps
- Average case: $O(n^2)$
  $O(n^2)$ comparisons, $O(n^2)$ swaps
- Best case: $O(n)$
  $O(n)$ comparisons, $O(1)$ swaps
- Space complexity: $O(1)$
- There are more concerns than performance
  - Many swaps
  - Bubble sort is *in-place*
- The repetitive algorithm pattern is common

```
swapped = True
n = len(seq)
while swapped:
  swapped = False
  for i in range(n - 1):
    if seq[i] > seq[i + 1]:
      seq[i], seq[i + 1]\
          = seq[i + 1], seq[i]
      swapped = True
```

- Not practical – it is not used in practice

Insertion sort

- Insertion sort is one of the simpler sorting algorithm
- It is easy to understand, and reasonably fast for sorting short sequences
- On longer sequences, it performs worse than more advanced algorithms, like merge sort or quicksort (we will study those later)
- The general idea simple:
  - assume the elements arrive one by one, and we have a sorted sequence
  - insert the element to the right position:
    - shift all elements larger than the new one to the right
    - place the new element in its correct place

Insertion sort

demonstration 1

89 | 67 | 88 | 12 | 72 | 76 | 93 | 57

```
for i in range(1, len(seq)):
  cur = seq[i]
  j = i
  while seq[j - 1] > cur\
          and j in range(1,i+1):
    seq[j] = seq[j - 1]
    j -= 1
  seq[j] = cur
```


Insertion sort

demonstration 2

67

89 | | 88 | 12 | 72 | 76 | 93 | 57

```
for i in range(1, len(seq)):
  cur = seq[i]
  j = i
  while seq[j - 1] > cur\
          and j in range(1,i+1):
    seq[j] = seq[j - 1]
    j -= 1
  seq[j] = cur
```


Insertion sort

demonstration 3

67

89 | 88 | 12 | 72 | 76 | 93 | 57

```
for i in range(1, len(seq)):
  cur = seq[i]
  j = i
  while seq[j - 1] > cur\
          and j in range(1,i+1):
    seq[j] = seq[j - 1]
    j -= 1
  seq[j] = cur
```


Insertion sort

demonstration 4

67 | 89 | 88 | 12 | 72 | 76 | 93 | 57

```
for i in range(1, len(seq)):
  cur = seq[i]
  j = i
  while seq[j - 1] > cur\
          and j in range(1,i+1):
    seq[j] = seq[j - 1]
    j -= 1
  seq[j] = cur
```


Insertion sort

demonstration 5

88

67 | 89 | | 12 | 72 | 76 | 93 | 57

```
for i in range(1, len(seq)):
  cur = seq[i]
  j = i
  while seq[j - 1] > cur\
          and j in range(1,i+1):
    seq[j] = seq[j - 1]
    j -= 1
  seq[j] = cur
```


Insertion sort

demonstration 6

88

67 | | 89 | 12 | 72 | 76 | 93 | 57

```
for i in range(1, len(seq)):
  cur = seq[i]
  j = i
  while seq[j - 1] > cur\
          and j in range(1,i+1):
    seq[j] = seq[j - 1]
    j -= 1
  seq[j] = cur
```

Insertion sort
demonstration 7

67	88	89	12	72	76	93	57

```
for i in range(1, len(seq)):
  cur = seq[i]
  j = i
  while seq[j - 1] > cur\
          and j in range(1,i+1):
    seq[j] = seq[j - 1]
    j -= 1
  seq[j] = cur
```


Insertion sort
performance

- Worst case: $O(n^2)$
  $O(n^2)$ comparisons, $O(n^2)$ swaps
- Average case: $O(n^2)$
  $O(n^2)$ comparisons, $O(n^2)$ swaps
- Best case: $O(n)$
  $O(n)$ comparisons, $O(1)$ swaps
- Space complexity: $O(1)$
- In practice, insertion sort is faster than the bubble sort (and also selection sort)

```
for i in range(1, len(seq)):
  cur = seq[k]
  j = i
  while seq[j - 1] > cur\
          and j in range(1,i+1):
    seq[j] = seq[j - 1]
    j -= 1
  seq[j] = cur
```

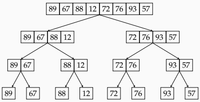

Insertion sort
summary

- Insertion sort is simple
- It is efficient for short sequences
- For long sequences it is much worse than more advanced algorithms like merge sort or quicksort (coming next)
- It is in-place
- It is *online*: it can sort items as they arrive
- It is *stable*: it does not swap elements with equal keys
- It is *adaptive*: faster if order of elements is closer to the sorted sequence

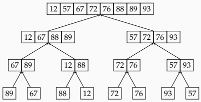
Merge sort
Introduction

- Merge sort is a divide-and-conquer algorithm for sorting
- It is relatively easy to understand (once you get your head around recursion)
- It has good asymptotic performance
- There are many practical cases where merge sort is used
- Basic idea is divide-and-conquer:
  - split the sequence
  - sort the subsequences
  - merge the sorted lists

Merge sort
demonstration – divide



Merge sort
demonstration – combine

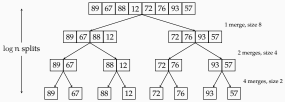


Merging sequences

```
# s1, s2: sequences to be merged
# s: target sequence
i, j = 0, 0
n = len(s1) + len(s2)
while i + j < n:
  if j == len(s2) or \
      i < len(s1) and s1[i] < s2[j]:
    s[i+j] = s1[i]
    i += 1
  else:
    s[i+j] = s2[j]
    j += 1
```

- Keep two indices on both sequences, starting from the beginning
- Pick the smallest, place it in the target sequence
- The algorithm requires $O(n)$ steps to complete

Complexity of the merge sort



$O(n) = n \log n$

Merge sort
the implementation

```
def merge_sort(s):
  n = len(s)
  if n <= 1: return
  s1, s2 = s[:n//2], s[n//2:]
  merge_sort(s1)
  merge_sort(s2)
  merge(s1, s2, s)
```

- Once we have `merge()`, the rest is trivial:
  - Split the array into two
  - Recursively sort both sides
  - Stop when the input is length 1

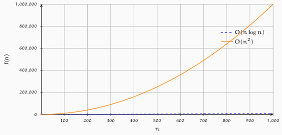
Merge sort: summary

- Straightforward application of divide-and-conquer
- Worst case $O(n \log n)$ complexity (best/average cases are the same)
- Merge sort is not in-place: requires $O(n)$ additional space
- It is particularly useful for settings with low random-access memory, or sequential access
- Merge sort is stable
- It is a well studied algorithm, there are many variants (in-place, non-recursive)

A short divergence to complexity
the difference between $O(n^2)$ and $n \log n$

n	$n \log n$	n^2
2	2	4
8	24	64
64	384	4096
1K	10 240	1 048 576
1M	20 971 520	1 099 511 627 776
1G	32 212 254 720	1 152 921 504 606 846 976

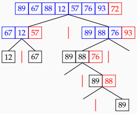
A short divergence to complexity
the difference between $O(n^2)$ and $n \log n$



Quicksort
introduction

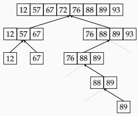
- Quicksort is another popular divide-and-conquer sorting algorithm
- The main difference from the merge sort is that big the part of the work is done before splitting
- Its worse time complexity is $O(n^2)$, but in practice it performs better than merge sort on average
- General idea: pick a pivot p, and divide the sequence into three parts as
  - L smaller than a particular element p
  - G larger than a particular element p
  - E equal to a particular element p
- sort L and G recursively
- combination is simple concatenation

Quicksort
demonstration – divide



At each divide step
- Pick a pivot
- Recursively call quicksort twice
  - L for items less than the pivot
  - G for items greater than the pivot
- $O(n)$ operations

Quicksort
demonstration – combine



At each combine step:
- Simply concatenate
  - L the sorted items less than p
  - E items equal to p
  - G the sorted items greater than p
- No need for $O(n)$ merging

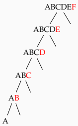
Quicksort
Python three-liner implementation

```
def qsort(seq):
  if len(seq <= 1): return seq
  return qsort([x for x in seq if x <  seq[-1]])\  # < p
        +      [x for x in seq if x == seq[-1]]\  # = p
        + qsort([x for x in seq if x >  seq[-1]])  # > p
```

- Practical implementations are not very different
- Common improvements include
  - in-place sorting
  - selecting the pivot more carefully

Quicksort
analysis

- Similar to the merge sort, quicksort performs $O(n)$ operations at each level in recursion
- The overall complexity is proportional to $n \times \ell$, where ℓ is depth of the tree
- The recursion tree of merge sort is balanced, so depth is $\log n$.
- For quicksort, we do not have a balanced-tree guarantee
- In the worst case, the depth of the tree can be n, resulting in $O(n^2)$ complexity



Quicksort
average-case complexity and preventing the worst case

- Worst case of the quicksort is when the input sequence is sorted
- If the input sequence is (approximately) random, the *expected* number of elements in each divide is $n/2$
- To reduce the probability of worst case, *randomized* quicksort picks the pivot randomly
- Best case happens if we pick the *median* of the sequence as the pivot, but finding median already requires $O(n \log n)$ (or $O(n)$, but not very practical)
- A common approach is picking three values (typically first, middle and last) from the sequence, and selecting the 'median of three' as the pivot

Quicksort
summary

- Complexity: $O(n \log n)$ average, $O(n^2)$ worst
- Despite its worst-case $O(n^2)$ complexity, quicksort is faster than merge sort on average (in practice)
- The algorithm can easily implemented in-place (in-place version is more common)
- Quicksort is not stable
- Quicksort is one of the most-studied algorithms: there are many variants, its properties are well known

Sorting algorithms so far, and the lower bound

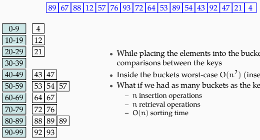
Algorithm	worst	average	best	memory	in-place	stable
Bubble sort	n^2	n^2	n	1	yes	yes
Insertion sort	n^2	n^2	n	1	yes	yes
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	no	yes
Quicksort	n^2	$n \log n$	$n \log n$	$\log n$	yes	no

- Can we do better than $O(n \log n)$?
- If our sorting algorithms requires comparing individual elements, the answer turns out to be 'no'
- Lower bound of worst-case sorting is $\Omega(n \log n)$
- In some special cases, linear-time complexity can be possible

Bucket sort
introduction

- Bucket sort puts elements of the input into a pre-defined number of ordered 'buckets'
- Elements in each bucket is sorted (typically using insertion sort)
- We can than retrieve the sorted elements by visiting each bucket
- The bucket sort *does not compare elements* to each other when deciding which bucket to place them
- In special cases, this results in $O(n)$ worst-case complexity

Bucket sort
demonstration



- While placing the elements into the buckets, no comparisons between the keys
- Inside the buckets worst-case $O(n^2)$ (insertion sort)
- What if we had as many buckets as the keys?
  - n insertion operations
  - n retrieval operations
  - $O(n)$ sorting time

Radix sort

- In a large number of cases, we want to sort object using multiple keys
- In such cases, we define the order of key pairs as $(k_1, l_1) < (k_2, l_2)$ if $k_1 < k_2$, or $k_1 = k_2$ and $l_1 < l_2$
- This definition can be generalized to key tuples of any length
- This ordering is known as *lexicographic* or dictionary order
- Radix sort is the name for the technique that uses multiple stable bucket sorts for this purpose

Summary

- Sorting is an important and well-studied computational problem
- Most sorting algorithms/applications used in practice are highly optimized, often based on multiple basic algorithms
- Naive sorting algorithms run in $O(n^2)$ time
- Lower bound on worst-case sorting time is $\Omega(n \log n)$, divide-and-conquer algorithms achieve this
- Reading: Goodrich, Tamassia, and Goldwasser (2013, chapter 12)
- And a fun way to see sorting in action: `https://www.youtube.com/user/AlgoRythmics`

Next:
- Trees
- Reading: Goodrich, Tamassia, and Goldwasser (2013, chapter 8)

Acknowledgments, credits, references

- Some of the slides are based on the previous year's course by Corina Dima.

Goodrich, Michael T., Roberto Tamassia, and Michael H. Goldwasser (2013). *Data Structures and Algorithms in Python*. John Wiley & Sons, Incorporated. ISBN: 9781118476734.

blank

blank

blank

blank