Sorting

Advanced programming

Artur Kulmizev

2020-02-09

What is sorting

- Place the objects in an array in a certain (ascending or descending) order
- We will focus on sorting in an array
- The most important operations that can be used are:
  - compare ($==$, $<$, $>$) – compare two values in the array
  - swap – swap two values in the array
- sometimes shift operations can be more efficient

Swap

Naive swap:

```
a = b
b = a
```

Swap

Naive swap: not working!

```
a = b
b = a
```

Swapping in most programming languages:

```
temp = a
a = b
b = temp
```

Alternative swapping in Python:

```
a,b = b,a
# or
(a,b) = (b,a)
```

How to sort!

- Try to come up with a way to sort the items in an array
- Available operations:
  - Compare $(<, >, ==)$
  - Swap

6	1	9	3	5	2	3	8	2	9	7	1

●	●	●	●	●	●	●	●	●	●	●	●

Selection sort

- Intuition:
  1. Split the array into a sorted part (empty from the beginning) and an unsorted part (the full array from the beginning)
  2. Find the smallest value in the unsorted part
  3. Swap the first item in the unsorted part, with the smallest item. This item now belongs to the sorted part
  4. Repeat steps 2+3 until the whole array is sorted

Selection sort – code

```
def selectionSort(alist):
   for slot in range(0, len(alist)-1):
       minPos=slot
       for location in range(slot+1,len(alist)):
           if alist[location] < alist[minPos]:
               minPos = location

       alist[slot], alist[minPos] = alist[minPos], alist[slot]
```

Selection sort – code (PS)

Variant from PS book, moving largest value to the end, rather than smallest to the beginning

```
def selectionSort(alist):
   for fillslot in range(len(alist)-1,0,-1):
       positionOfMax=0
       for location in range(1,fillslot+1):
           if alist[location]>alist[positionOfMax]:
               positionOfMax = location

       temp = alist[fillslot]
       alist[fillslot] = alist[positionOfMax]
       alist[positionOfMax] = temp
```

Selection sort – code (lab package 2)

Variant from lab 2, using a predefined API

```
def selectionSort(sa):
    for i in range(0, sa.getSize()-1):
        min = i
        for j in range(i+1, sa.getSize()):
            if sa.cmp(j,min) < 0:
                min = j
        sa.swap(i,min)
```

Lab package 2, part 3

- Sorting of integers with a given API
  - Implement two sorting algorithms
  - Compare three sorting algorithms – connect to theory

Lab package 2, part 3

- Sorting of integers with a given API
  - Implement two sorting algorithms
  - Compare three sorting algorithms – connect to theory
- The code contains a class, SortArray that encapsulates the data, and counts the operations performed
- Provides the following methods for sorting:
  - cmp(i,j) – compares the elements at position i and j < 0 if element i $<$ element j, 0 if element i $==$ element j, > 0 if element i $>$ element j,
  - swap(i,j) – swaps the elements at position i and j
  - getSize() – how many elements are there in the array

Lab package 2, part 3

- Sorting of integers with a given API
  - Implement two sorting algorithms
  - Compare three sorting algorithms – connect to theory
- The code contains a class, SortArray that encapsulates the data, and counts the operations performed
- Provides the following methods for sorting:
  - cmp(i,j) – compares the elements at position i and j
    < 0 if element i $<$ element j, 0 if element i $==$ element j, > 0 if element i $>$ element j,
  - swap(i,j) – swaps the elements at position i and j
  - getSize() – how many elements are there in the array
- Also methods for printing array and statistics

Lab package 2, part 3

- Sorting of integers with a given API
  - Implement two sorting algorithms
  - Compare three sorting algorithms – connect to theory
- The code contains a class, SortArray that encapsulates the data, and counts the operations performed
- Provides the following methods for sorting:
  - cmp(i,j) – compares the elements at position i and j
    < 0 if element i $<$ element j, 0 if element i $==$ element j, > 0 if element i $>$ element j,
  - swap(i,j) – swaps the elements at position i and j
  - getSize() – how many elements are there in the array
- Also methods for printing array and statistics
- Lists can be preordered in different ways – scrambled, nearly sorted, and reversed

Aside – encapsulation

- Information hiding
- Prohibiting others from using attributes/methods in a class
- Often considered more safe, that the class takes care about its own variables
- Most object-oriented programming languages have **access control**, defining who can use an attribute/method
  - Private: can only be used within the class
  - Protected: can be used in the class and in subclasses
  - Public: can be used everywhere

Aside – encapsulation

- Information hiding
- Prohibiting others from using attributes/methods in a class
- Often considered more safe, that the class takes care about its own variables
- Most object-oriented programming languages have **access control**, defining who can use an attribute/method
  - Private: can only be used within the class
  - Protected: can be used in the class and in subclasses
  - Public: can be used everywhere
- Python does not have this, but it can be signaled by naming conventions:
  - _variableName: is meant to be treated as private/protected
  - __variableName: is meant to be treated as private, has name mangling (replaced with _classname__variableName outside the class)
- The public API of a class are the methods that are supposed to be used outside the class

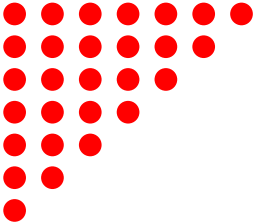
Encapsulation in lab package 2

- All attributes (instance variables) in class SortArray are meant to be treated as private.
- You may not access them directly
- You may only use the methods defined in the class
- Reasons for this:
  - Exercise in using a predefined API
  - An example of encapsulation
  - To force you to think about sorting algorithms, in order to be able to re-write them using this API
  - To provide you with measurements of how many operations a sorting algorithm needs, in a clean way (abstracted away)

Selection sort – complexity

```
def selectionSort(sa):
    for i in range(0, sa.getSize()-1):     # n-1
        min = i                             # constant
        for j in range(i+1, sa.getSize()): # n-1 + n-2 + ... + 1
            if sa.cmp(j,min) < 0:           # constant
                min = j                     # constant
        sa.swap(i,min)                      # constant
```

Selection sort – complexity



- $(n-1)+(n-2)+...+2+1=\frac{(n^2+n)}{2} \in O(n^2)$
- e.g. $f(8)=28$
- So-called triangular numbers
- Quadratic complexity – the n^2 term dominates

Bubble sort

- Intuition:
    1. Split the array in a sorted part (empty in the beginning) and an unsorted part (everything in the beginning)
    2. For each pair of values from left to right in the unsorted part:
        - Swap if the right value is smaller than the left value
    3. This means that the smallest value is “bubbled up” to the first position in the unsorted part, and can be moved to the sorted part
    4. Repeat until the full array is sorted

Bubble sort – code (PS)

Variant from PS book, moving largest value to the end, rather than smallest to the beginning

```
def bubbleSort(alist):

    passnum = len(alist)-1
    while passnum > 0:

       for i in range(passnum):
           if alist[i]>alist[i+1]:

               temp = alist[i]
               alist[i] = alist[i+1]
               alist[i+1] = temp
       passnum = passnum-1
```

Bubble sort – improved version

- Depending on the configuration of numbers in the array, the array might actually become sorted before we are done with the full algorithm
- In this case, there will be only comparisons, no swaps
- We can notice that if we do a full outer loop without a single swap, the array must necessarily be sorted
- To take advantage of this, record if some swap is performed, and stop the algorithm if no swaps are done

Improved bubble sort – code (PS)

Variant from PS book, moving largest value to the end, rather than smallest to the beginning

```
def impBubbleSort(alist):
    exchanges = True
    passnum = len(alist)-1
    while passnum > 0 and exchanges:
       exchanges = False
       for i in range(passnum):
           if alist[i]>alist[i+1]:
               exchanges = True
               temp = alist[i]
               alist[i] = alist[i+1]
               alist[i+1] = temp
       passnum = passnum-1
```

Insertion sort

- Intuition:
  1. Split the array into a sorted part (containing the left-most element from the beginning) and an unsorted part (containing the rest of the array)
  2. Move the first value in the unsorted part to the correct position in the sorted part
  3. Repeat until the whole array is sorted

Insertion sort – code

Variant from PS book, uses a shift operation, to move items in the array

```
def insertionSort(alist):
   for index in range(1,len(alist)):

     currentvalue = alist[index]
     position = index

     while position>0 and alist[position-1]>currentvalue:
         alist[position]=alist[position-1]
         position = position-1

     alist[position]=currentvalue
```

Insertion sort – code

Variant from PS book, uses a shift operation, to move items in the array

```
def insertionSort(alist):
   for index in range(1,len(alist)):

     currentvalue = alist[index]
     position = index

     while position>0 and alist[position-1]>currentvalue:
         alist[position]=alist[position-1]
         position = position-1

     alist[position]=currentvalue
```

Note that in the lab, you will have to use swap instead (because of the API), which is slightly less efficient, but which serves the purpose of illustrating the strengths of the algorithm

Complexity – bubble sort and insertion sort

- $O(n^2)$
- Use the same reasoning as for selection sort, triangular numbers

Sorting algorithms – properties

- Time Complexity – how fast is it?
- Memory Complexity – how much memory does it need?
- Stable – are equal values guaranteed to be in the same order as before the search?
- Adaptive – can it take advantage of the existing order of elements, so that it is faster for sequences with less disorder, e.g. if applied to an already sorted, or nearly sorted array?

Sorting algorithms – properties

- All algorithms we have discussed so far
    - Time: $O(n^2)$
    - Memory: $O(1)$ extra memory
    - Stable
- Selection sort and standard bubble sort: non-adaptive
- Insertion sort and improved bubble sort: adaptive (best case time complexity: O(1))

Sorting algorithms – properties

- All algorithms we have discussed so far
  - Time: $O(n^2)$
  - Memory: $O(1)$ extra memory
  - Stable
- Selection sort and standard bubble sort: non-adaptive
- Insertion sort and improved bubble sort: adaptive (best case time complexity: O(1))
- There are more advanced algorithms which can have a worst case complexity of $O(n \log n)$
- That's the limit of how fast a sorting algorithm based on comparisons can be
- These algorithms might not be stable, and might require extra memory

Merge sort

- Intuition:
    - Split the array into two halves
    - Keep splitting until the size is 1 - i.e. the partition is sorted
    - Iteratively merge already sorted partitions, until the full array is sorted

Merge sort

- Intuition:
  - Split the array into two halves
  - Keep splitting until the size is 1 - i.e. the partition is sorted
  - Iteratively merge already sorted partitions, until the full array is sorted
- Time complexity (worst):
- Stable:
- Needs extra memory:
- Adaptive:

Merge sort

- Intuition:
  - Split the array into two halves
  - Keep splitting until the size is 1 - i.e. the partition is sorted
  - Iteratively merge already sorted partitions, until the full array is sorted
- Time complexity (worst): $O(n \log n)$
- Stable: Yes
- Needs extra memory: $O(n)$ extra memory
- Adaptive: No (unless modified)

Quick sort

- Intuition:
  - Pick a pivot
  - Split the array in two halves, one with elements smaller than the pivot, one with elements larger than the pivot
  - Recursively sort each partition using quick sort
  - Concatenate: the small part, the pivot, the large part

Quick sort

- Intuition:
  - Pick a pivot
  - Split the array in two halves, one with elements smaller than the pivot, one with elements larger than the pivot
  - Recursively sort each partition using quick sort
  - Concatenate: the small part, the pivot, the large part
- Time complexity (average):
- Time complexity (worst):
- Stable:
- Needs extra memory:
- Adaptive:

Quick sort

- Intuition:
  - Pick a pivot
  - Split the array in two halves, one with elements smaller than the pivot, one with elements larger than the pivot
  - Recursively sort each partition using quick sort
  - Concatenate: the small part, the pivot, the large part
- Time complexity (average): $O(n \log n)$
- Time complexity (worst): $O(n^2)$
- Stable: No (unless modified)
- Needs extra memory: No (depending on implementation)
- Adaptive: No

Shell sort

- Intuition:
  - Run insertion sort, but on sequences with different gaps
  - Start with far apart elements
  - Reduce the gaps until they are one (when this is ordinary insertion sort)
- Behavior depends on the gap sequence chosen

Shell sort

- Intuition:
  - Run insertion sort, but on sequences with different gaps
  - Start with far apart elements
  - Reduce the gaps until they are one (when this is ordinary insertion sort)
- Behavior depends on the gap sequence chosen
  - Time complexity (worst):
  - Stable:
  - Needs extra memory:
  - Adaptive:

Shell sort

- Intuition:
  - Run insertion sort, but on sequences with different gaps
  - Start with far apart elements
  - Reduce the gaps until they are one (when this is ordinary insertion sort)
- Behavior depends on the gap sequence chosen
  - Time complexity (worst): Hard to analyze, between $O(n^2)$ and $O(n \log^2 n)$ depending on gap sequence. $O(n^{3/2})$ is also common
  - Stable: No
  - Needs extra memory: No
  - Adaptive: Yes

Sorting in lists

- All sorting algorithms we have discussed today can only be efficiently applied to arrays, since they require "random access"
- Very inefficient for linked structures
- Some of the algorithms can be modified for linked structures
    - Insertion sort $O(n^2)$
    - Merge sort $O(n \log n)$

Sorting in lists

- All sorting algorithms we have discussed today can only be efficiently applied to arrays, since they require ”random access”
- Very inefficient for linked structures
- Some of the algorithms can be modified for linked structures
  - Insertion sort $O(n^2)$
  - Merge sort $O(n \log n)$
- Alternative: insert the values sorted in a linked list (as you will do in the linked list part of lab 2) (Time complexity: $O(n^2)$ for inserting n values – boils down to insertion sort)
- But in that case, there are better options, for instance binary search trees

Timsort

- The default sort in Python
- Very optimized and fast
- Based on a combination of binary insertion sort and merge sort
- Takes advantage of natural runs of sorted data
- Worst case: O(n log n)
- Best case: O(n) – adaptive
- Stable
- Requires some extra memory (worst case O(n) - but typically less)

More about sorting algorithms

- We have not looked at code for all algorithms today
- Study it on your own, and implement some algorithms in lab 2
- You should be able to describe all algorithms that we talked about today
- On the exam:
  - Show how each algorithm works, by using an example on an array
  - Be able to write code for the simple algorithms
  - Discuss the time complexity and other features of all algorithms
- The PS book has code and good descriptions and animations for all algorithms
- Wikipedia has very good material about sorting algorithms

Coming up

- Lab tomorrow and Thursday
- Soft lab deadline: Feb. 19
- Next week: web scraping and text processing