Sorting

Data Structures
Lecturer: Caio Fonseca

Introduction

- This section looks at a variety of sorting algorithms for arranging elements of a list/collection in a specified order.
  - For instance, ascending or descending order.
  - Sorts can use one or more data attributes for sorting. For instance, sort films/movies by descending year and then by ascending title.
- Different sorting algorithms have different advantages and disadvantages.
  - Simplicity, efficiency, speed, memory usage, etc.
  - Algorithmic analysis is considered in more detail in the follow-on module next semester.
- We will also see how sorting can help avoid inefficient linear/sequential search by enabling "binary search".

Continued…

- Note that for the purposes of introducing the sort algorithms, we will generally assume we are sorting an array of integers in ascending/natural order in the first instance. However, the same sorting principles apply for lists of objects.

Selection Sort

- Selection sort is a simple algorithm that iterates over the list starting at the first element and "selects" the next (e.g. lowest) element for that position from the remainder of the list.
- Elements become sorted "one-by-one".
- Elements before the current "selection point" are known to be sorted, so only remaining elements after the selection point need to be considered.
- A nested loop is the standard idiom for selection sort.
- Advantages of the selection sort include its simplicity of implementation, and that it doesn't need auxiliary memory to work.

Selection Sort Example

```
def selectionSort(a):
  for sp in range(len(a)-1):
        smallestIndex=sp

        i = sp+1
        for i in range(i, len(a)):
            if a[i]<a[smallestIndex]:
                smallestIndex=i

        If smallestIndex != sp:
            swap=a[sp]
            a[sp]=a[smallestIndex]
            a[smallestIndex]=swap
```

Assume selection point element is smallest to start with (it could be!)

Select smallest element in remainder of list

Found a smaller one so use that instead

If the selection point is not the smallest element already then swap them

Insertion Sort

- Insertion sort is similar to selection sort in that it sorts lists one element at a time.
- Insertion sort takes each element in turn and inserts it into its sorted position in the list. The element at the insertion point and all elements above it have to be shifted up to make room for the insertion.
- Insertion sort's advantage over selection sort is that it only scans as many already sorted elements as needed to determine the correct position of the current element, whereas selection sort must scan all remaining unsorted elements to find the smallest one.

Insertion Sort Example

```
def insertionSort(a):
  for e in range(len(a)):
        i = e
        for i in range(0, i):
           if a[e]<a[i]:
              tmp = a[e]
              a[e] = a[i]
              a[i] = tmp

        return a
```

Bubble Sort

- Another simple sorting algorithm, bubble sort compares pairs of adjacent elements and swaps them as required (i.e. if in the wrong order) over multiple passes/iterations.
- When bubble sorting in ascending order, the larger values essentially "bubble" towards the end of the list and the smaller values "bubble" towards the start of the list.
- After the first pass, the largest value will have bubbled to the end of the list (and therefore does not need to be considered again). The next largest item will similarly bubble into its final place on the next iteration, and so on.
  - All other values will not necessarily be in place after a pass, but will at least have moved (one position) closer to their sort order positions.
- If no swaps are performed in a given pass then the list is already sorted (if you count swaps).

Bubble Sort Example

```
def bubbleSort(a):
   for b in range(1, len(a):
      for i in range(0, len(a)-b):

         If a[i]>a[i+1]:
            swap=a[i]
            a[i]=a[i+1]
            a[i+1]=swap

   return a
```

Rabbits and Turtles

- During bubble sorting, elements move into their correct positions at different speeds.
- Elements that move into position quickly are known as "rabbits" and elements that move into position slowly are known as "turtles".
  - For instance, the largest element will move quickly to the end of the list (a rabbit) in a single bubble sort pass as it will win every swap/comparison, but a small element towards the end of the list can only move one position on each pass (a turtle).
- Turtles are a problem as they slow down the sort.
- Variations on the regular bubble sort that attempt to deal with turtles include:
  - Comb sort. This compares/swaps elements that are N apart (looks like a comb!) and reduces N towards 1 after every pass (the teeth of the comb get progressively closer together). N usually starts out as the length of the list and reduces by a shrink factor (usually 1.3) on each pass.
  - Cocktail sort. This is a bidirectional bubble sort that reverses itself from beginning to end and then back again (like shaking a cocktail up/down or left/right). Turtles essentially become rabbits as the sort reverses direction.

Sorting by "Divide and Conquer"

- A number of sorting techniques take a "divide and conquer" approach by using recursion to break down the overall sorting task to progressively smaller sorting tasks before combining the efforts to form the overall result.
- Recursion is an approach whereby something is "defined in terms of itself".
  - A recursive method, for instance, calls itself (directly or indirectly).
  - Key elements to recursion are (1) the "base case" where the problem is so simple/small that it can be solved directly, and (2) the "recursive step" which uses a recursive call with a simpler/smaller version of the current problem to help solve the overall problem.

Quicksort

- Quicksort is an example of a recursive divide-and-conquer sorting technique.
- Quicksort sorts a list by:
  1. Choosing a "pivot" element.
     - The pivot can be arbitrary or random, but using the median of the first, last and middle elements can improve performance.
  2. "Partition" the list so elements smaller than the pivot are reordered to come before it, and elements greater than it reordered to come after it (equal values can go either side).
  3. These two partitions are now "recursively" sorted (repeat these steps all over for each partition!).
  4. The results from the recursive sorts are used "end-to-end" to form the overall result.

Quicksort Example

```
1   import random
2
3   def partition(seq, start, stop):
4       # pivotIndex comes from the start location in the list.
5       pivotIndex = start
6       pivot = seq[pivotIndex]
7       i = start+1
8       j = stop-1
9
10      while i <= j:
11          #while i <= j and seq[i] <= pivot:
12          while i <= j and not pivot < seq[i]:
13              i+=1
14          #while i <= j and seq[j] > pivot:
15          while i <= j and pivot < seq[j]:
16              j-=1
17
18          if i < j:
19              tmp = seq[i]
20              seq[i] = seq[j]
21              seq[j] = tmp
22              i+=1
23              j-=1
24
25      seq[pivotIndex] = seq[j]
26      seq[j] = pivot
27
28      return j
29
30  def quicksortRecursively(seq, start, stop):
31      if start >= stop-1:
32          return
33
34      # pivotIndex ends up in between the two halves
35      # where the pivot value is in its final location.
36      pivotIndex = partition(seq, start, stop)
37
38      quicksortRecursively(seq, start, pivotIndex)
39      quicksortRecursively(seq, pivotIndex+1, stop)
40
41  def quicksort(seq):
42      # randomize the sequence first
43      for i in range(len(seq)):
44          j = random.randint(0,len(seq)-1)
45          tmp = seq[i]
46          seq[i] = seq[j]
47          seq[j] = tmp
48
49      quicksortRecursively(seq, 0, len(seq))
```

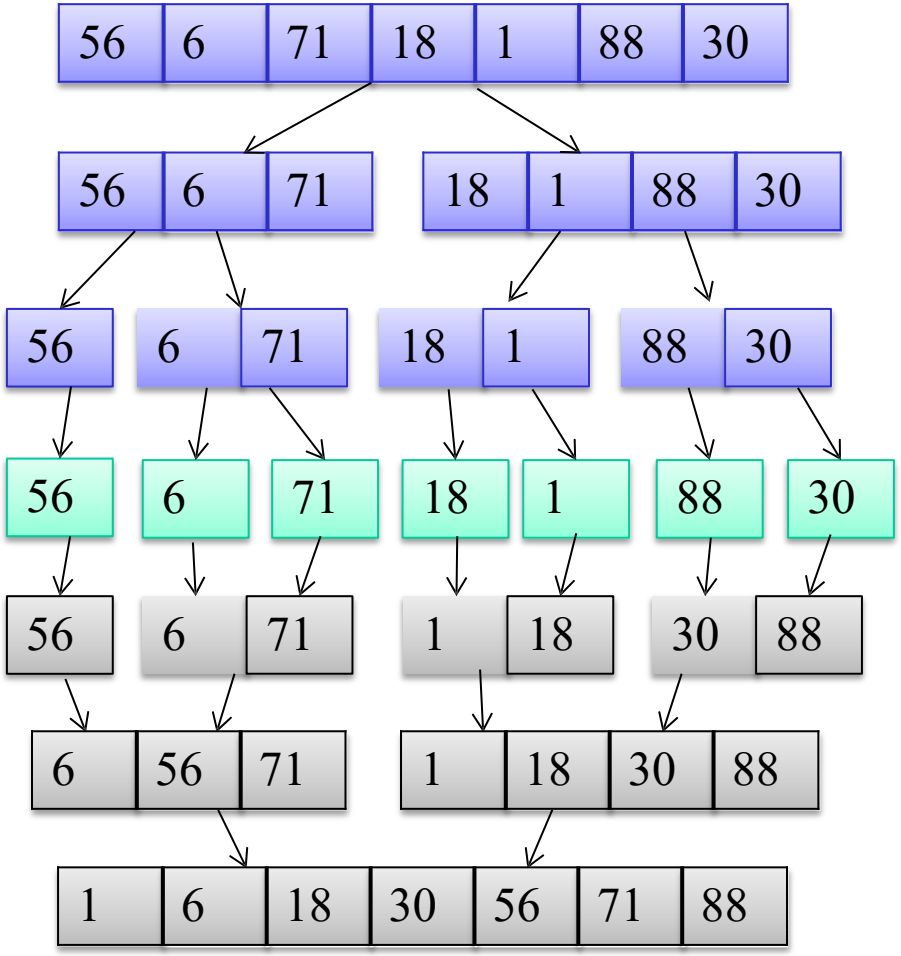
Example from the reference book: Data Structures and Algorithms with Python – Kent Lee & Steve Hubbard

Mergesort

- Mergesort is another divide-and-conquer recursive sorting technique.
- Mergesort works by (top-down approach!):
  - If the input list is only 1 element long then simply return it (base case; it is already sorted!)
  - Otherwise (recursive step) break the list into 2 sub-lists of equal size (±1 element for odd sized lists) and recursively call the mergesort method on these 2 sub-lists (which repeats these steps again!)
  - Once the 2 recursive calls return the 2 sorted sub-lists then simply merge them on an item-by-item basis.

Continued…

- Mergesort essentially repeatedly breaks down lists until only 1-element sub-lists remain. These lists are then merged / combined as the recursive chain unwinds to assemble larger sorted sub-lists and so on until the original list has been sorted.



Mergesort Example

```
1   def merge(seq, start, mid, stop):
2       lst = []
3       i = start
4       j = mid
5
6       # Merge the two lists while each has more elements
7       while i < mid and j < stop:
8           if seq[i] < seq[j]:
9               lst.append(seq[i])
10              i+=1
11          else:
12              lst.append(seq[j])
13              j+=1
14
15      # Copy in the rest of the start to mid sequence
16      while i < mid:
17          lst.append(seq[i])
18          i+=1
19      # Many merge sort implementations copy the rest
20      # of the sequence from j to stop at this point.
21      # This is not necessary since in the next part
22      # of the code the same part of the sequence would
23      # be copied right back to the same place.
24      # while j < stop:
25      #     lst.append(seq[j])
26      #     j+=1
27      # Copy the elements back to the original sequence
28      for i in range(len(lst)):
29          seq[start+i]=lst[i]
30
31  def mergeSortRecursively(seq, start, stop):
32      # We must use >= here only when the sequence we are sorting
33      # is empty. Otherwise start == stop-1 in the base case.
34      if start >= stop-1:
35          return
36
37      mid = (start + stop) // 2
38
39      mergeSortRecursively(seq, start, mid)
40      mergeSortRecursively(seq, mid, stop)
41      merge(seq, start, mid, stop)
42
43  def mergeSort(seq):
44      mergeSortRecursively(seq, 0, len(seq))
```

Example from the reference book: Data Structures and Algorithms with Python – Kent Lee & Steve Hubbard

Binary Search

- When a list is sorted (using any sorting algorithm) we can search/retrieve an element from it using a "binary search".
- A binary search is much more efficient than a linear search and is entirely unaffected by list size.
- A binary search works by considering the middle element of a sorted list in order to determine if the sought element would come before or after the middle. It essentially splits the list into two, thus halving the number of remaining elements to consider in a single step. The binary search continues in the same fashion (halving the input list on each iteration) using whichever half of the current list the sought element is in until the element is found or determined not to exist.
- It is easy to implement binary search both recursively and iteratively.

Summary

- In this section we have looked at:
  - Various sorting algorithms.
    - Selection, insertion, bubble (and comb/cocktail variants), shell, quick, and merge sorts.
  - Recursion for “divide and conquer”.
  - Binary search.

End of Notes