Sorting methods

Tom Smedsaas

Experiments with some sort functions.

The aim is to show how to verify theory with experiments and to point out when recursion is good and when it is bad.

Insertion sort againt

```
def ins_sort_rec(lst):
    return _ins_sort_rec(lst, len(lst))


def _ins_sort_rec(lst, n):
    if n > 1:
        _ins_sort_rec(lst, n-1)
        x = lst[n-1]
        i = n - 2
        while i >= 0  and lst[i] > x:
            lst[i+1] = lst[i]
            i -= 1
        lst[i+1] = x
    return lst
```

Merge sort again

```
def merge_sort(lst):
    if len(lst) <= 1: return lst
    else:
        n = len(lst)//2
        l1 = lst[:n]
        l2 = lst[n:]
        l1 = merge_sort(l1)
        l2 = merge_sort(l2)
        return merge(l1, l2)

def merge(l1, l2):
    if len(l1) == 0:
        return l2
    elif len(l2) == 0:
        return l1
    elif l1[0] <= l2[0]:
        return [l1[0]] + merge(l1[1:], l2)
    else:
        return [l2[0]] + merge(l1, l2[1:])
```

How will time grow for these methods?

We will look at how the times change when the size is doubled.

The inserton sort is a $\Theta(n^2)$ – method.

If the size is doubled, the time will grow with a factor $\frac{c \cdot (2n)^2}{c \cdot n^2} = 4$

Merge sort is a $\Theta(n \log n)$ – method.

If the size is doubled, the time will grow with a factor

$$\frac{c \cdot 2n \log(2n)}{c \cdot n \log n} = 2 \cdot \frac{\log 2 + \log n}{\log n} = 2 \cdot (1 + \frac{1}{\log_2 n})$$

If $n > 1000$ the factor is less than 2.1.

Test code

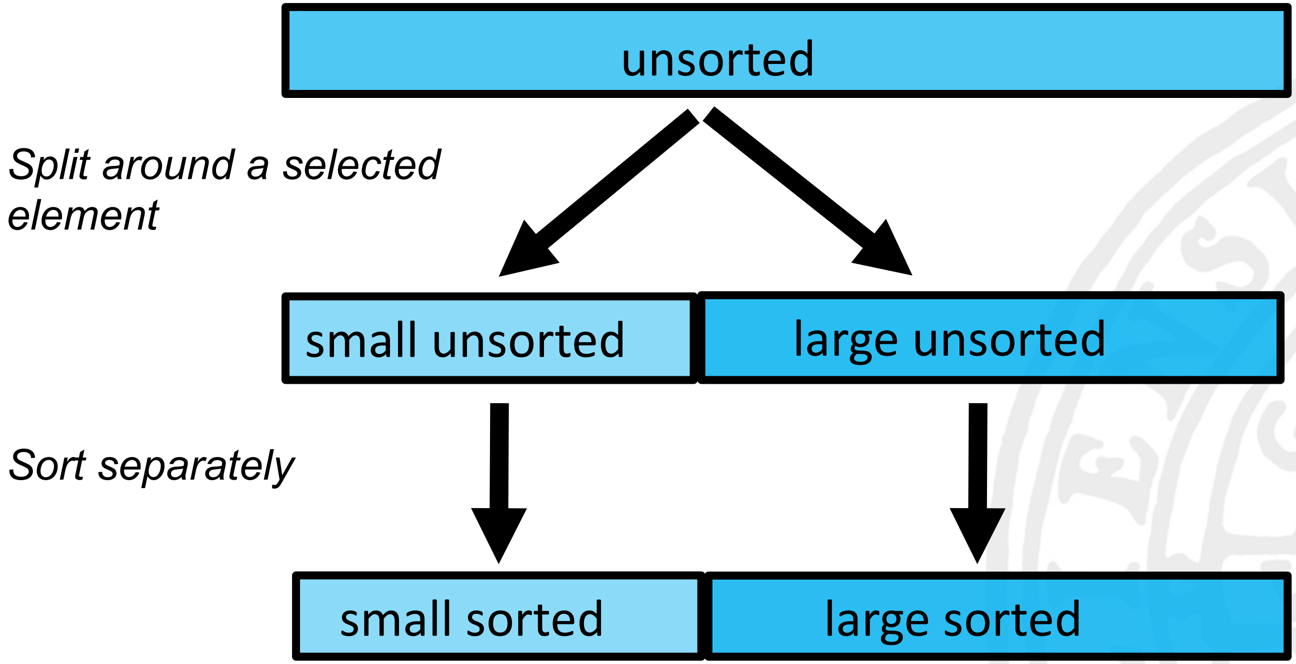
```
sort_functions = [ins_sort_rec, merge_sort]


for sort in sort_functions:
    print('\n', sort.__name__)
    for n in [1000, 2000, 4000, 8000]:
        lst = []
        for i in range(n):
            lst.append(random.random())
        tstart = time.perf_counter()
        lst = sort(lst)
        tstop = time.perf_counter()
        print(f" Time for {n}\t : {tstop - tstart:4.2f}")
```

Test run

Another sorting method

Idea: Rearrange the elements so that the small is on the left part (unsorted) and the big in the right part (also unsorted).



Partition sort

```
def psort(lst):
    _psort(lst, 0, len(lst)-1)
    return lst

def _psort(lst, n, m):
    if m > n:
        ip = partition(lst, n, m)
        _psort(lst, n, ip-1)
        _psort(lst, ip+1, m)
```

Similarities and differences with merge sort?

The partition

```
def partition(lst, n, m):
    if m>n:
        p = lst[n]
        i = n
        j = m
        while j > i:
            while j>i and lst[j] > p:
                j -= 1
            lst[i] = lst[j]
            while i < j and lst[i] < p:
                i += 1
            lst[j] = lst[i]
            j -= 1
        lst[i] = p
        return i
```

Note: The method is often called *quicksort.*

Test run

Summary

- Time measurements can be used to verify theoretical results.
- The theoretical results for insertion sort and merge sort agree very well in practice.
- $\Theta(n \log n)$ is much better than $\Theta(n^2)$!
- Good to balance the algorithms.
- Do not use recursion over long lists!.
- No problems with the recursion depth in mergesort and quicksort.

The end

.