Merge and Quick Sort

1. Divide and Conquer
   - applied to sorting
   - recursive and iterative selection sort
   - cost considerations
2. Merge Sort
   - split, sort, and merge
   - a recursive sort function
   - an iterative version
3. Quick Sort
   - partition and sort
   - cost considerations
   - timing Python code

MCS 275 Lecture 16
Programming Tools and File Management
Jan Verschelde, 15 February 2017

Merge and Quick Sort

1. Divide and Conquer
   - applied to sorting
   - recursive and iterative selection sort
   - cost considerations
2. Merge Sort
   - split, sort, and merge
   - a recursive sort function
   - an iterative version
3. Quick Sort
   - partition and sort
   - cost considerations
   - timing Python code

Divide and Conquer

applied to sorting

To solve a problem, divide it into smaller problems.

Solve the smaller problems and use the solutions of the smaller problems to solve the original problem.

Recursion is natural, base case is trivial problem,
apply mathematical induction for general case.

Sorting is a classical, important problem.

Consider various issues:

- data in memory or on file
- procedural or functional solution
- recursive or iterative algorithm

Merge and Quick Sort

1. Divide and Conquer
   - applied to sorting
   - recursive and iterative selection sort
   - cost considerations
2. Merge Sort
   - split, sort, and merge
   - a recursive sort function
   - an iterative version
3. Quick Sort
   - partition and sort
   - cost considerations
   - timing Python code

Selection Sort

E.g.: sort `[51, 26, 30, 51, 53, 43, 26]`

```
L = [51, 26, 30, 49, 43, 27]     S = []
   min(L) == 26;   L.index(26) == 1
L = [51, 30, 49, 43, 27]     S = [26]
   min(L) == 27;   L.index(27) == 4
L = [71, 51, 30, 49, 43]     S = [26, 27]
   min(L) == 30;   L.index(30) == 2
L = [71, 51, 49, 43]     S = [26, 27, 30]
   min(L) == 43;   L.index(43) == 3
L = [51, 49]     S = [26, 27, 30, 43]
   min(L) == 49;   L.index(49) == 1
L = [51]     S = [26, 27, 30, 43, 49]
   min(L) == 51;   L.index(51) == 0
L = []     S = [26, 27, 30, 43, 49, 51]
```

Sorting Recursively and Iteratively

Selection Sort done recursively:

- base case: length of list ≤ 1
- let `m` be the minimum of the list,
  denote by `rest` the list minus `m`,
  return `[m]` + sorted(`rest`)

Selection Sort done iteratively:

- let `S` be the sorted list, `S = []`
- as long as the unsorted list is not empty:
  1. select the minimum from the unsorted list
  2. append the minimum to the sorted list `S`
  3. remove the minimum from the unsorted list

Procedural version does not return a sorted list,
but sorts the list given on input.

the function `recursive_select`

```
def recursive_select(data):
    """
    Returns a list with the data
    sorted in increasing order.
    """
    if len(data) <= 1:
        return data
    else:
        mindata = min(data)
        data.pop(data.index(mindata))
        return [mindata] + recursive_select(data)
```

an iterative version

```
def iterative_select(data):
    """
    Returns a list with the data
    sorted in increasing order.
    """
    result = []
    while len(data) > 0:
        mindata = min(data)
        result.append(mindata)
        data.pop(data.index(mindata))
    return result
```

Merge and Quick Sort

1. Divide and Conquer
   - applied to sorting
   - recursive and iterative selection sort
   - cost considerations
2. Merge Sort
   - split, sort, and merge
   - a recursive sort function
   - an iterative version
3. Quick Sort
   - partition and sort
   - cost considerations
   - timing Python code

Cost Considerations

Cost factors in a sorting algorithms:

1. the number of comparisons
2. the number of moves
   $\rightarrow$ *memory access is often the bottleneck!*

Analyzing Selection Sort to sort n elements:

1. there are n steps in the algorithm
2. every step requires to look for a minimum
   in a list of k elements, $k = n, n-1, \ldots, 1$

$$\#\text{comparisons}: \quad n-1+n-2+\cdots 2+1 = \frac{n(n-1)}{2}$$

Selection Sort performs $O(n^2)$ comparisons.
If already sorted, then no moves needed,
but on average also $O(n^2)$ moves to sort a list.

Merge and Quick Sort

1. Divide and Conquer
   - applied to sorting
   - recursive and iterative selection sort
   - cost considerations
2. Merge Sort
   - split, sort, and merge
   - a recursive sort function
   - an iterative version
3. Quick Sort
   - partition and sort
   - cost considerations
   - timing Python code

split, sort, and merge

E.g.: sort `[23, 86, 41, 24, 69, 85, 30, 31]`.

`[23, 86, 41, 24, 69, 85, 30, 31]`

split

`[23, 86, 41, 24]` `[69, 85, 30, 31]`

split

`[23, 86]` `[41, 24]` `[69, 85]` `[30, 31]`

sort

`[23, 86]` `[24, 41]` `[69, 85]` `[30, 31]`

merge

`[23, 24, 41, 86]` `[30, 31, 69, 85]`

merge

`[23, 24, 30, 31, 41, 69, 85, 86]`

merging sorted lists

```
def merge(one, two):
    """
    Returns the merge of lists one and two,
    adding each time min(one[0], two[0]).
    """
    result = []
    while len(one) > 0 and len(two) > 0:
        if one[0] <= two[0]:
            result.append(one.pop(0))
        else:
            result.append(two.pop(0))
    return result + (one if len(one) > 0 else two)
```

Merge and Quick Sort

1. Divide and Conquer
   - applied to sorting
   - recursive and iterative selection sort
   - cost considerations
2. Merge Sort
   - split, sort, and merge
   - a recursive sort function
   - an iterative version
3. Quick Sort
   - partition and sort
   - cost considerations
   - timing Python code

the function `recursive_merge`

```
def recursive_merge(data, verbose=True):
    """
    Returns a list with the data
    sorted in increasing order.
    """
    if verbose:
        print('data =', data)
    if len(data) <= 1:
        return data
    else:
        middle = len(data)//2
        left = recursive_merge(data[:middle])
        right = recursive_merge(data[middle:])
        return merge(left, right)
```

Merge and Quick Sort

1. Divide and Conquer
   - applied to sorting
   - recursive and iterative selection sort
   - cost considerations
2. Merge Sort
   - split, sort, and merge
   - a recursive sort function
   - an iterative version
3. Quick Sort
   - partition and sort
   - cost considerations
   - timing Python code

merge sort done iteratively

E.g.: sort `[23, 86, 41, 24, 69, 85, 30, 31]`.

view list as lists of singletons:

```
[[23], [86], [41], [24], [69], [85], [30], [31]]
```

merge lists of length 1:

```
[[23, 86], [24, 41], [69, 85], [30, 31]]
```

merge lists of length 2:

```
[[23, 24, 41, 86], [30, 31, 69, 85]]
```

merge lists of length 4:

```
[[23, 24, 30, 31, 41, 69, 85, 86]]
```

The iterative loop splits, for as long as length of sublists is less than the length of the original list.

Assume the length of the list to sort is power of 2.

an iterative merge sort

```
def iterative_merge(data):
    """
    Returns a list with the data
    sorted in increasing order.
    Assumes len(data) is a power of 2.
    """
    ind = 1
    hop = 2
    while hop <= len(data):
        k = 0
        while k+hop <= len(data):
            data[k:k+hop] = merge(data[k:k+ind], \
                data[k+ind:k+hop])
            k = k + hop
        ind = hop
        hop = 2*hop
    return data
```

Merge and Quick Sort

1. Divide and Conquer
   - applied to sorting
   - recursive and iterative selection sort
   - cost considerations
2. Merge Sort
   - split, sort, and merge
   - a recursive sort function
   - an iterative version
3. Quick Sort
   - partition and sort
   - cost considerations
   - timing Python code

Partition and Sort

E.g.: sort `[28, 14, 12, 31, 19, 20, 65, 41]`

`[28, 14, 12, 31, 19, 20, 65, 41]`

≤ 28: `[14, 12, 19, 20]` > 28: `[31, 65, 41]`

≤ 14: `[12]` > 14: `[19, 20]` ≤ 31: `[]` > 31: `[65,41]`

`[12] + [14] + [19, 20]` `[] + [31] + [41, 65]`

`[12, 14, 19, 20] + [28] + [31, 41, 65]`

`[12, 14, 19, 20, 28, 31, 41, 65]`

the function quick_sort

```
def quick_sort(data, verbose=True):
    """
    Returns a list with the data
    sorted in increasing order.
    """
    if verbose:
        print('data =', data)
    if len(data) <= 1:
        return data
    else:
        (left, right) = ([], [])
        for k in range(1, len(data)):
            if data[k] < data[0]:
                left.append(data[k])
            else:
                right.append(data[k])
        result = quick_sort(left)
        result.append(data[0])
        return result + quick_sort(right)
```

Merge and Quick Sort

1. Divide and Conquer
   - applied to sorting
   - recursive and iterative selection sort
   - cost considerations
2. Merge Sort
   - split, sort, and merge
   - a recursive sort function
   - an iterative version
3. Quick Sort
   - partition and sort
   - cost considerations
   - timing Python code

Cost Considerations

Merge Sort and Quick Sort cut lists in half.

Cost for Merge Sort:

- Best case: if sorted, then no moves needed.
- Worst case: if in reverse order, then many swaps.
- always needs $2 \times \log_2(n)$ stages.

Cost for Quick Sort

- in best case only $\log_2(n)$ stages
- if unbalanced partition, then $O(n^2)$

Merge and Quick Sort

1. Divide and Conquer
   - applied to sorting
   - recursive and iterative selection sort
   - cost considerations
2. Merge Sort
   - split, sort, and merge
   - a recursive sort function
   - an iterative version
3. Quick Sort
   - partition and sort
   - cost considerations
   - timing Python code

timing Python code using `time.clock`

```
import time
starttime = time.clock()
< code to be timed >
stoptime = time.clock()
elapsed = stoptime - starttime
print 'elapsed time: %.2f seconds' % elapsed
```

Replace `< code to be timed >` by a function call.

timing with `os.times()`

The total elapsed time consists of

- user cpu time: time spent on user process,
- system time: time spent by operating system.

Interesting to separate user cpu from system time.

```
import os
a = os.times()
< code to be timed >
b = os.times()
print 'user cpu time : %.4f' % (b[0] - a[0])
print '  system time : %.4f' % (b[1] - a[1])
print ' elapsed time : %.4f' % (b[4] - a[4])
```

Counting Operations

Timing programs depends on hardware.

As a byproduct of the computations,
we could report the number of operations.

For sorting, we distinguish between

1. the comparisons of data,
2. the assignments, movements of data.

Altering the code by making it more general:

1. replace $<$, $=$, $>$ by a `compare` function,
2. replace $:=$ by a `copy` function.

The function `compare` and `copy` maintain tallies.
Similar to memoization for efficient recursive functions.

Exercises

1. Compare the recursive select sort with the iterative version for longer lists. Which is most efficient? Justify.
2. Modify the function select sort so that it takes on input an array instead of a list. Instead of returning the sorted array, the array on input should be sorted.
3. Compare running times for the iterative versions of select sort and merge sort. How long must the list be before merge sort beats select sort?
4. Assume the data to sort is in an array. Change both versions of merge sort so that it does not return a sorted array but sorts the array on input.
5. Adjust the implementation of the iterative merge sort so it can also sort lists whose length is not a power of 2.
6. Describe a linear cost algorithm to sort less than 100 natural numbers, all different from each other.