Mergesort
Computer Science 1321
Spring Semester 2012

0. Introduction:

Mergesort is another divide and conquer algorithm similar to quicksort, and in fact is competitive with quicksort in many sorting situations. Both algorithms are O(n*ln(n)) algorithms in the average case. Mergesort tends to be a bit more stable than quicksort in the sense that it is not bothered by anomalities in the data like quicksort. By this I mean that quicksort, assuming the first element is chosen as pivot, would degenerate to essentially a selection sort if the data were sorted or nearly sorted, or in reverse order. Mergesort does not suffer from this same deficiency. Mergesort has performance which is consistent independent of the original order of the data. You may find that quicksort will beat mergesort on random data. Which is chosen is in some cases a simple matter of preference. How hard the programmer has to work is also a factor.

1. Merge:

1.1. Non-recursive Merge:

Mergesort is predicated upon the assumption that if we have two sorted lists, merging them into a single sorted list is an easy task. This function is fundamental to the mergesort algorithm.

```
def Merge(lsa,lsb):
        #assume that lsa and lsb are sorted lists. The function is to return a single sorted list
        #as its return value
```

Suppose we wish to merge the lists

```
lsa = [1,4,5,6,9,10]
lsb = [2,3,5,7,8,10,11,12]
```

We initialize a list to return:

```
lsc = [].
```

We also initialize three pointers,

```
i,j,k = 0,0,0
```

Compare the ith element of lsa with the jth element of lsb. If lsa[i] <= lsb[j] then write the element lsa[i] to position k of lsc and increment i and k. Otherwise write the element lsb[j] to position k of lsc and increment j and k. Continue until one of the lists lsa or lsb is exhausted. Then simply dump the rest of the elements of the remaining list to lsc and return lsc as the merged list. This two-way merge is a well known algorithm and is easy to write in Python.

1.2. Recursive Merge:

The algorithm as described in the above paragraph is iterative. We simply compare the first elements of each of the lists, and copy the smaller to the returned list. It is also instructive to note that there is a purely recursive and functional way to envision the merge algorithm. Study the following functional version of merge to make sure you understand what it does:

```
def merge(a,b):
     if len(a) == 0:
          return b
     if len(b) == 0:
          return a
     if a[0] < b[0]:
          return [a[0]] + merge(a[1:],b)
     else:
          return [b[0]] + merge(a, b[1:])
```

You will be asked to make comparisons between the various ways of implementing merge, as well as the various ways there are to implement the mergesort algorithm. It is interesting to learn what works best. Certainly the above is easy to program. But is it as efficient? What of the ability of Python to perform recursion? Does Python slice lists efficiently? These questions and more will be posed below.

```
>>> a = range(10)
>>> b = range(15)
>>> c = merge(a,b)
>>> print a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print b
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> print c
[0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 11,
12, 13, 14]
>>>
```

Note that the merge algorithm as presented does not alter either of the orginal lists. They are not mutated.

2. Mergesort:

2.1. Recursive Mergesort:

Mergesort is commonly written as a recursive algorithm. Very simply, we divide the list to be sorted in half, and recursively sort each half. We then merge the resulting lists into a single sorted list, and the algorithm terminates. The base case of the recursion is a list of size zero or one elements, which are clearly sorted lists. Of course, in python, it is very easy to divide the list in half. We calculate:

lenlst = len(lst)
m = lenlst/2

m is then the index (integer division) of the middle of the list. The two halves of the list are then lst[:m] and lst[m:].

These halves may be recursively sorted, and then merged using the merge algorithm.

2.2. Non-recursive Mergesort:

There is another way to think of the mergesort algorithm. In a sense, the description above envisions the mergesort algorithm as a top down recursive algorithm. But we may also envision the algorithm as a bottom up iterative algorithm.

To illustrate this idea, suppose we wish to sort the list

ls = [1,9,8,2,3,7,6,4,5,10].

We apply a function which will produce a list of lists of size one:

ls = [[1],[9],[8],[2],[3],[7],[6],[4],[5],[10]]

We next apply a function which will merge each pair of lists, since a list of size one is a sorted list:

ls = [[1,9], [2,8], [3,7], [4,6], [5,10]]

Each of these two element lists is a sorted list, provided by the Merge function in repeated applications. We repeat again with each two element list, producing a list of four element lists:

ls = [[1,2,8,9], [3,4,6,7], [5,10]]

There is no one to merge the last pair with, so we simply keep it around for future merge passes through the list. Another merge pass produces:

ls = [[1,2,3,4,6,7,8,9], [5,10]]

The final merge pass produces a list of lists of size one:

ls = [[1,2,3,4,5,6,7,8,9,10]].

At this point the algorithm can terminate, returning ls[0] as its return value. This is the desired sorted list. As you can see, this is simply repeated applications of the merge algorithm. Since there is no recursive overhead for this algorithm it should be competitive with the other versions of mergesort that may be produced.

3. Student Experiments:

Note that with files of reasonable size, recursive stack space is at a premium. It is possible to allocate more space to recursion with a function in the module sys. To reset the recursion limit, you must

import sys

Then, in your main function, include a call to the function setrecursionlimit, similar to the following:

sys.setrecursionlimit(1000000).

The machines we are experimenting with have lots of ram, so allocating a large amount of space to handle the recursion should be no problem.

Using the data created for the earlier experiments, we will write four different versions of the mergesort algorithm and compare their efficiency. The four versions are

Task One: iterative merge with recursive mergesort
Task Two: iterative merge with non-recursive (bottom up) mergesort
Task Three: recursive merge with recursive mergesort
Task Four: recursive merge with non-recursive (bottom up) mergesort

Task Five: Write a summary report with your findings. We wish to know several things: How does python handle recursion? How does the mergesort compare to quicksort? How does mergesort compare with heapsort? How does mergesort compare with the treesort we have done earlier? Use a spreadsheet and summarize your timings.

Make a folder and put everything you produced for this exercise in it. Name the folder

youruserid.mergesort

Copy into my public directory:

```
cp -r youruserid.mergesort /users/meggen/Public
```