[S17 HW9] sorting

Due April 4, 2017, 9 a.m.

MATH 121 SPRING 2017 HOMEWORK 9

Sorting

For all the problems below you will submit a file containing the code that you are asked to write. For the first three problems, your code will be tested by the VRFY system but the score that is reported will not be your final grade. Instead, it will simply indicate whether our tests concluded that the code works as expected. We will look at your submissions later to see if your algorithms are correct.

The last problem is not auto-graded. Instead, you will also hand in on paper a short report on the timing experiments you ran using the code that you submit for Problem 4 on CSHW below.

[S17 HW9 P1]

Exercise: selection sort

Selection sort is a method for sorting a list that works by placing the 0-th item, then the 1st item, then the 2nd item, and so on until the list is sorted. It works *in place*, meaning that it swaps pairs of values until the list is sorted. It works by making *n-1* passes for a list of length *n*. Let's number these passes from 0 to *n-2*.

For the 0-th pass, we scan through the whole list to find the minimum element in the list. We then swap that element with the element at position 0. For the 1st pass, we scan through elements 1 through *n-1* to find the minimum of those elements. We then swap that element with the element at position 1. We continue in this way. In the *i*-th pass, we scan through elements *i* through *n-1* to find the next minimum value, then perform a swap to place it at position *i*.

Once we complete the pass numbered *n-2*, the whole list will be sorted.

Write a function `def selectionsort(aList)` that sorts the `aList` it is given using selection sort. Note that your code should not return anything. Instead, it should modify the list it is given, rearranging the items so that they are put in sorted order.

Answer the following question: What is the running time of this sorting algorithm? Include a comment in your code describing the running time of `selectionsort` .

[S17 HW9 P2]

Exercise: count sort

There are situations in coding where the list of integers that we need to sort are drawn from a limited range. For example, the list

```
[1, 2, 0, 0, 1, 3, 2, 1, 0, 1, 3, 3, 3, 1, 0, 1, 0]
```

only has elements with values between 0 and 3. When that is the case, it is common to use a special sort called *count sort*.

Here is how it works. Assume that the numbers in our list are integers from 0 up to *m-1*. There are two phases. In the first phase we scan through the list and count the number of occurrences of each integer in that list. This is done by making a list of counts `count` , of length *m*, where `count[i]` is the number of times that the value `i` occurs in the list we are sorting. It makes one pass through the list in order to figure out all these counts.

For the list above, the `count` list is of length 4 and has the contents

```
[5, 6, 2, 4]
```

because 0 appears five times, 1 appears six times, 2 appears twice, and 3 appears four times.

In the second phase, we "rewrite" the list we are sorting to have that data in order. That is, we do the work to place the *m* values in order into the list according to their counts. For the example list, that ultimately places five 0s, six 1s, two 2s, and four 3s to obtain the list contents

```
[0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 3, 3, 3, 3]
```

Write a function `def countsort(aList,m)` that is given a list and the value of `m` (indicating that the list values are from 0 to `m-1`) and sorts the contents of `aList` using count sort. Note that your code should not return anything. Instead, it should modify the list it is given so that the data is put in sorted order.

Answer the following question: What is the running time of your `countsort` code? Include a comment in your code describing the running time of your function.

[S17 HW9 P3]

Exercise: finding the i-th element

Let's now consider a different problem that is like sorting, but turns out to not require the full work of sorting. Suppose you are given a list but just want to find the value of the element that would be in the *i*-th position were the list sorted. For example, looking at the list

```
[8, 6, 1, 3, 11, 7]
```

we see that 1 would be in the 0-th sorted position because it is the smallest, 11 would be in the 5th sorted position because it is the largest, 3 would be in the 1st sorted position, and so on. This, in general, is called finding the *i*-th order statistic of the list. It is the value that has *i* values smaller than it in the list.

For this problem, to keep the description simple, we'll assume that the values in the list are distinct (no values appear more than once).

We can use a method like quicksort to find the *i*-th order statistic for any given *i*. The method restructures the list but does not sort it completely. First, we pick a "pivot" value from the list (just like in quicksort) and reorder the list, partitioning it into the values that are less the pivot---call this the left partition---and the values that are greater than the pivot---call this the right partition. Having done that, we have a count of the number of values that are less than the pivot (call this the *left count*) and a count of the number of values that are greater than the pivot (call this the *right count*).

Now, if the left count is equal to *i*, that means that the pivot is the *i*-th order statistic. The algorithm is done. It has found the *i*-th order statistic for the list.

Otherwise, if the left count is greater than *i*, that means that the *i*-th order statistic element is sitting in the left partition. If instead the left count is less than *i*, then the order statistic we want is sitting in the right partition. In either case, we can repeat this work on either the left or the right partition (ignoring the other partition) to narrow down the list contents to find the *i*-th order statistic for the original list.

Write the code for a function `def statistic(i,aList)` that does the work of the algorithm suggested above and returns the value that is the `i` -th order statistic of `aList` . You can assume that the list of values has no value appear more than once. You can also assume that `i` is non-negative and less than the length of `aList` .

Incidentally, for your reference, below is the code for `quicksort` from Adam's lecture slides:

```
def partition(ls, left, right):
    less = left + 1
    greater = right
    while less <= greater:
        if ls[less] < ls[left]:
            less = less + 1
        else:
            ls[less],ls[greater]= ls[greater],ls[less]
            greater = greater - 1
    ls[left], ls[less - 1] = ls[less - 1], ls[left]
    return less - 1

def qshelp(ls, first, last):
    if first < last:
        pivot = partition(ls, first, last)
        qshelp(ls, first, pivot-1)
        qshelp(ls, pivot+1, last)

def quicksort(ls):
    qshelp(ls, 0, len(ls)-1)
```

[S17 HW9 P4]

Exercise: timing

The goal in this exercise is to measure in real time the efficiency of various sorting algorithms and to compare them. You will write some code and time how quickly it runs. You will be asked to graph the measurements you take. You should write this code in a single file and submit it here. On paper, you should turn in the graphs, explanations, and any other data that you produce as requested below.

To do these measurements you will need to write several code components.

First, write a function that takes as input an integer *n* and returns a list of the integers 1 through *n* in random order. Make sure that it is truly random-—-all orders of the integers should be equally likely.

Next, **you should implement** bubble sort, insertion sort, merge sort, and quicksort. It is okay to use code that was given in lecture, though you should check it for bugs.

Now, **you should write code that times** how long it takes each algorithm to sort lists of various sizes. You should create a graph that shows a line representing each sort's time as a function of the input length. **Write a short note** explaining how they compare at various input sizes and whether the lines you are getting are consistent with our theoretical expectations. You will find the `time` module useful for this.

I recommend timing the algorithm sorting a number of lists of the same length and then averaging to find the typical time it takes the algorithm to sort a list of a certain length. That will give a more accurate estimate of the time it takes on average. Make sure you don't include generating the randomly-ordered lists when you do the timing---you want to time only how long it takes to sort them. Repeat this timing and averaging procedure for a number of different list sizes. Choose enough different input sizes to get a good idea of the shape of the graph.

Attempt [S17 HW9 P1] selection sort

Attempt [S17 HW9 P2] count sort

Attempt [S17 HW9 P3] statistic

Attempt [S17 HW9 P4] timing