Quantitative Reasoning 20
Final Exam
May 14, 2009

Suggested answers: These are not the only answers accepted, but should give a clear idea of what we had in mind. Also, there is a lot of explanatory verbiage and alternative answers given for each problem which you were NOT required to give.

1. Two good ways of going about this are to: sort the list into increasing order and then return the value in the middle position of the sorted list OR to repeatedly trim the list by slicing out the largest and smallest values until only 1 or 2 items remain. For the former, we could use a simple selection sort:

```python
def median(values):
    s=sort(values)
    return s[len(s)/2]
def sort(x):  # here is a simple selection sort
    # in class we wrote this out nicely with functions to switch values in 2 positions
    # in a list and to find the position of the smallest value in a list.
    # here I wrote it all in one
    for top in range(len(x)-1):
        pos=top  # guess position of smallest
        for i in range(top+1,len(x)):
            if x[i]<x[pos]:  # found smaller one?
                pos= i  # improve the guess
        temp=x[top]  # put the smallest on top
        x[top]=x[pos]
        x[pos]=temp  # by switching top and pos
    return x
```

OR

```python
def median(values):
    while len(values)>2:
        values=trim(values)  # trim will remove largest and smallest
    return values[0]
def trim(x):
    # slice out largest and smallest values in x
    pos=0  # guess position of smallest
    pob=0  # and biggest
    for i in range(1,len(x)):
        if x[i] < x[pos]:  # found a smaller
            pos=i
        if x[i] > x[pob]:  # found a bigger
            pob=i
    # slice out pos and pob
    x=x[:pos]+x[pos+1:]
    # and then pob unless it equals pos in which case we don’t want to slice again
    if pos!=pob:
        x=x[:pob]+x[pob+1:]
    return x
```
1b. If you took the latter approach (repeated trimming of the list), your functions must make a complete pass through the list with an if to improve the guess at the smallest and an if to improve the guess at the biggest. There are n items, actually on average n/2 since the list gets smaller every time we trim it, so that is 2 * n/2 executions for each trim. We reduce the length of the list by 2 (NOTE we SUBTRACT 2 not DIVIDE by 2!). That means we must make roughly n/2 repetitions to trim it to 1 item. So, we do (n/2) * (2*n/2) = n^2/2 executions of an if. It is traditional to say that this is an n^2 algorithm, ignoring the constant multipliers. I can’t think of any more efficient way to do this, in fact, there is none.

If you took the former approach (sorting first and then returning the middle position), we can fall back on what we know about sorts. The selection sort, that I used as it is the simplest to write, in my opinion, is an n^2 algorithm. I know that there are more efficient sorts. Either the insertion sort using a binary search to find the insert position or the merge sort are n * log n algorithms. Using one of them would have been more efficient. If I had used one of them for part 1a, I could say that there isn’t a more efficient way to do the median since they are, within a constant or so, the most efficient sorts.

2. a,c and f print
5
4
3
2
1

b prints  e prints  d prints
0        0        5
-1       1        4
-2       2        3
-3       3        2
-4       4        1

3a. The class is straightforward:

class candidate:
    def count(self):
        self.votes=self.votes+1
        return
    def __init__(self,name):
        self.name=name
        self.votes=0
        self.button=Button(root,text="vote",command=self.count)
        return
3b. This is just the datain we have used several times in class and homeworks with simple modification:

def datain():
    f=open("candidates.txt","r")
    data=[]
    while True:  #repeat the read loop until a empty line breaks out
        namein=f.readline()
        if len(namein)==0:
            break  #we are done
        namein=namein[:-1]  #slice off the new line character
        data=data+[candidate(namein)]  #add new candidate object
    f.close()  #always good to do explicitly, but not needed
    return data

3c. The script with parts 3a, 3b and 3d to be inserted:

    from Tkinter import *
    root=Tk()

    #the class definition from 3a goes here
    #the datain function from 3b goes here

    #read the name from the file and create the list of candidate instance objects
    candidates=datain()

    #grid the names and buttons
    namelabel=[]  #you could have made the name label in the class definition if you wanted
    for i in range(len(candidates)):
        namelabel=namelabel+[Label(root,text=candidates[i].name)]  #make up a label for the
        candidates[i].name
        namelabel[i].grid(row=i,column=0)  #and grid it for display
        candidates[i].button.grid(row=i,column=1)  #and grid candidates[i] button

    #the event bind and its callback from 3d go here
    root.mainloop()  #and turn it loose

3d. The event bind and its callback. It goes before the root.mainloop()

    #bind a Button-1 click to the first candidate's name label to show vote totals
    #first, its callback function
    def showvotes(event):
        for i in range(len(candidates)):
            votelabel=Label(root,text=str(candidates[i].votes))  #we don't need to make a list of these
            votelabel.grid(row=i,column=2)  #since once we grid them we don't refer to them again
        return

    #now bind it
    namelabel[0].bind("<Button-1>",showvotes)
4. The points of this are to realize that 0-255 can give only 2 hex digits, that the left digit is the number of times 16 can divide the given decimal integer and the right digit is the remainder, and that one can use the dividend and remainder as an index to select the corresponding hex symbol from a list.

def dectohex(n):
    #n is assumed to be from 0 to 255 so it can be only 2 hex digits
    sixteensplace=int(n/16)  #the number in the left digit
    onesplace=n%16  #the remainder is in the right digit
    #we must convert these to the hex symbols
    hexdigits="0123456789abcdef"
    #note that each symbol's position is its numerical value
    return hexdigits[sixteensplace]+hexdigits[onesplace]