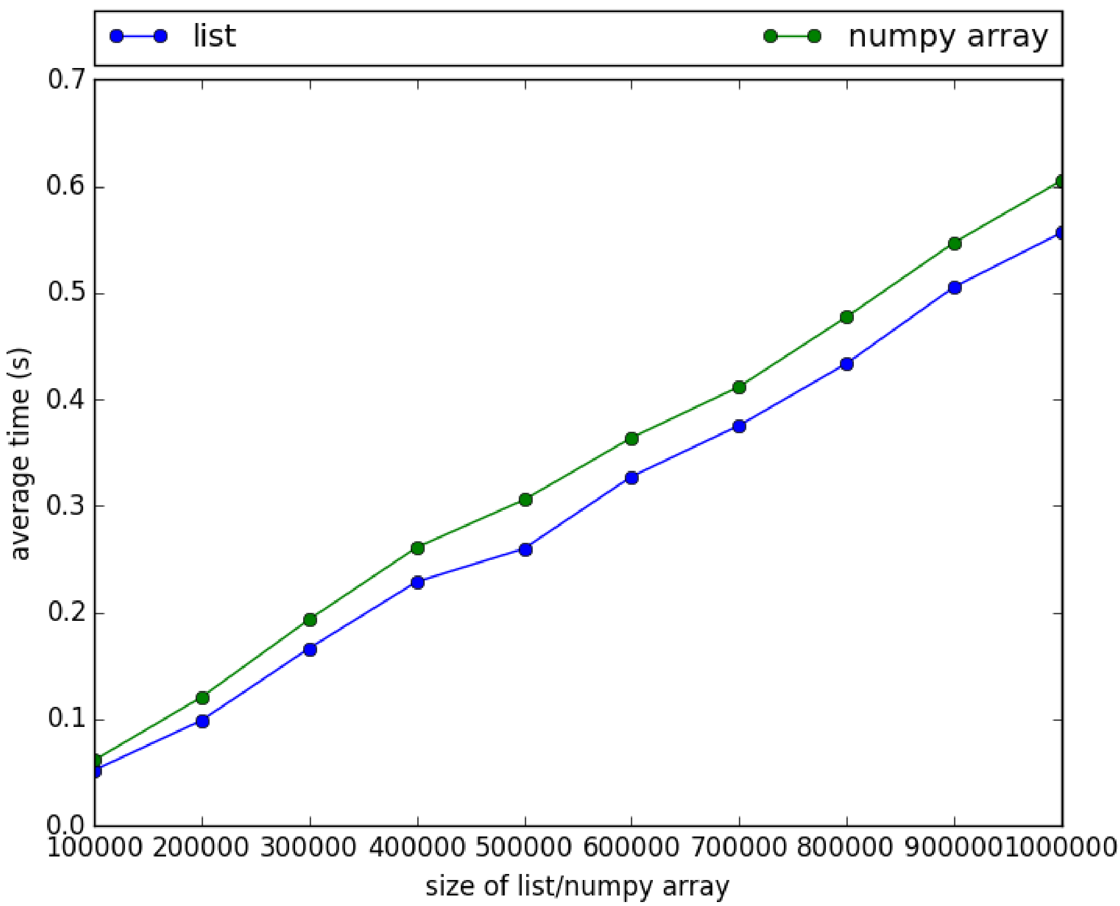
Question 1

- See code for function

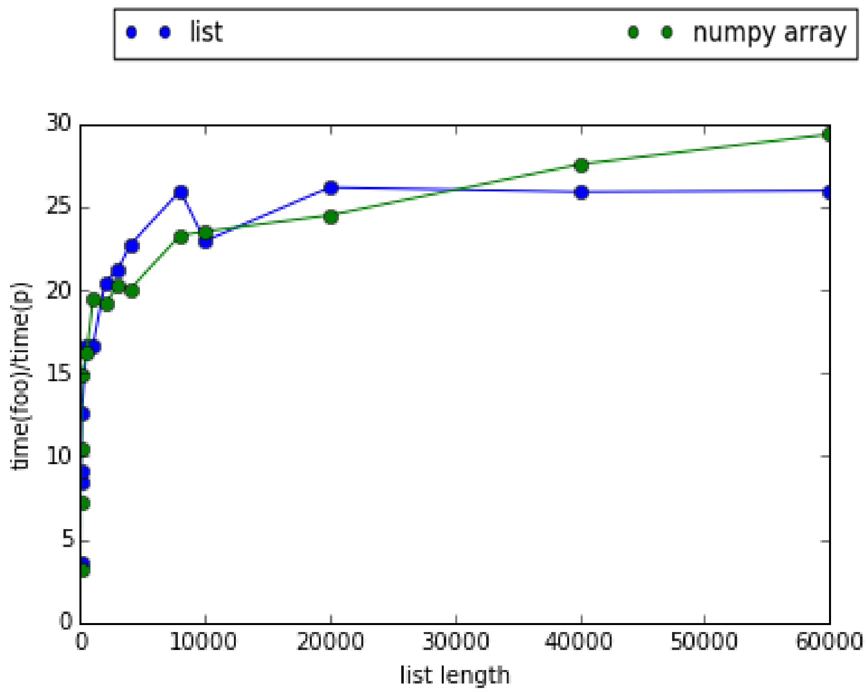
Question 2

- Complexity of the p function : O(n) (i.e. linear)
- Looks like it takes slightly longer for numpy arrays vs. python lists
- Plot:



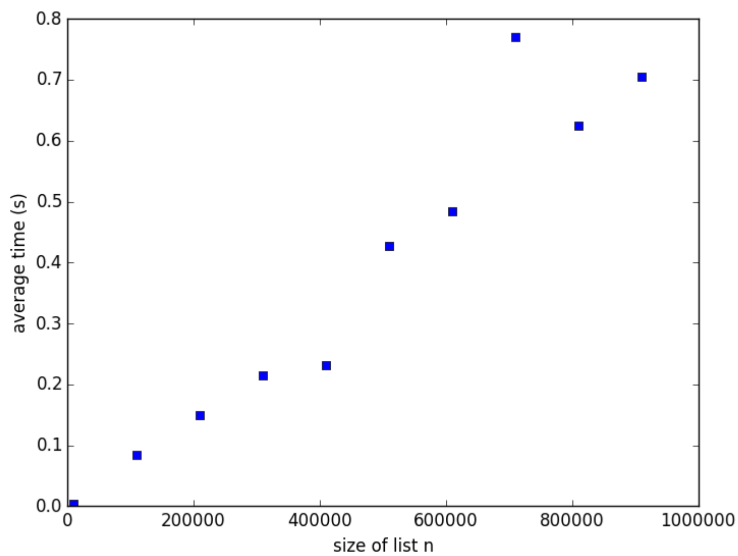
Question 3

- foo sorts the array “a” in increasing order
- Average time complexity = O(n*log(n))
- Plot time(foo)/time(p) vs. size (looks like log(n) since time(p) is O(n) and time(foo) is O(n*log(n)))

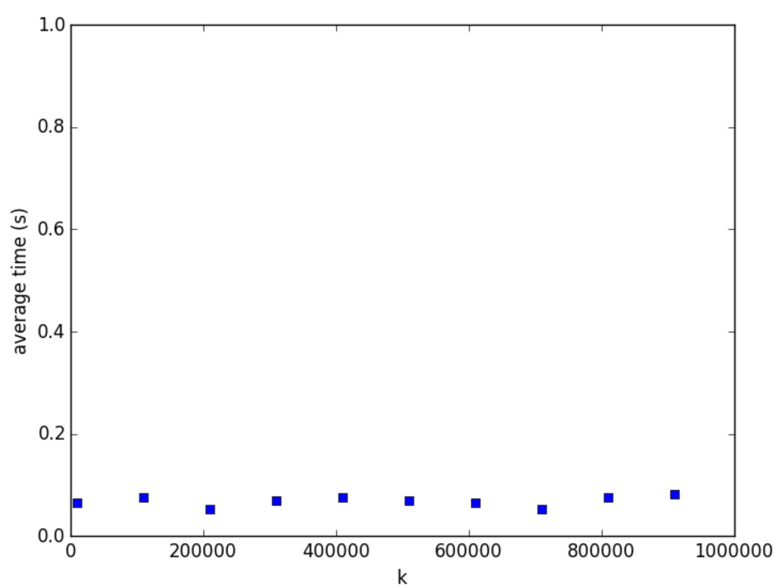


Question 4

- *Note: a mistake in the function in the problem statement was fixed*
- bar finds the kth smallest element in array "a" (i.e. k = 0 is the first smallest, k=1 is the second smallest, etc)
- Average time complexity = O(n) (i.e. linear) as graph shows
- Time complexity does not depend on k as graph shows (only depends on "n" or len(a))
- Plot time(bar) vs. n (for fixed k)



- Plot time(bar) vs. k (for fixed n)



Code

```
######## Exercise 1 #############################################

'''
Write a function p having three arguments.
The first one, l, is a mutable indexable object (e.g. a list or
numpy array).The second and third ones are integers, i and j.
Function p will rearrange in place the elements of l in the range [i:j]
(remember in python ranges i is included, and j is not included),
into two nonempty ranges [i:q] and [q:j] such that each element in
l[i:q] is less than or equal to each element l[q:j].
The index q is returned by this function.
'''

def p(l,i,j):

    import random
    import numpy
    # partition value
    pivot = l[random.randint(i,j-1)]
    # other option try median

    # define pointers moving from start (left) and end (right)
    left = i
    right = j-1

    # iterate until left and right pointers cross
    while left <= right:

        # advance left pointer until value greater or equal than pivot
        while l[left] < pivot:
            left += 1

        # advance right pointer until value less or equal than pivot
        while l[right]> pivot:
            right -= 1

        # swap if in wrong side of pivot
        if left <= right:
            temp = l[left]
            l[left] = l[right]
            l[right]= temp
            left +=1
            right-=1

    return left

######## Exercise 2 #############################################

# Complexity of the function p is O(n)

def findTimes(f,size,nreps,npy):
    import random
    import time # I prefer using time.perf_counter rather than timeit as
```

```
shown below
    import numpy as np

    avg_time = []
    #size = list(range(minsize,maxsize,stepsize))

    # iterate for different size of array
    for n in size:

        # generate list of size n or  numpy array
        if npy:
           l = np.array(range(n))
        else:
           l = list(range(n))

        totalTime = 0

        #  repetitions random shuffle
        for rep in range(0,nreps):
            random.shuffle(l) # randomize the list
            timeStamp = time.process_time() # get the current cpu time
            f(l, 0, n) # run p function
            timeLapse = time.process_time() - timeStamp
            totalTime += timeLapse

        # store average time
        avg_time.append(totalTime/nreps)

        print('p time: n[{0}]={1}'.format(n, totalTime/nreps))

    return [size,avg_time]


minsize = 100000
maxsize = 1000000+minsize
stepsize = minsize
size = list(range(minsize,maxsize,stepsize))
nreps = 15
npy = False
f = p

import random
random.seed(12345)
times_list =findTimes(f,size,nreps,npy)

random.seed(12345)
npy = True
times_npy =findTimes(f,size,nreps,npy)

# plot avg time vs size of list
import matplotlib.pyplot as plt

plt.plot(times_list[0],times_list[1],'-bo', label="list")
plt.plot(times_npy[0],times_npy[1],'-go', label="numpy array")
# Place a legend above this legend, expanding itself to
# fully use the given bounding box.
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
```

```python
            ncol=2, mode="expand", borderaxespad=0.)

plt.xlabel('size of list/numpy array')
plt.ylabel('average time (s)')

plt.show()


######## Exercise 3 ###############################################

def foo(a, i, j):
    if j-i>1:
        q = p(a,i,j)
        foo(a,i,q)
        foo(a,q,j)


import random
import numpy as np
# run for function p
#size = np.linspace(10000,1000000,20).astype(int)
size = [500, 1000, 2000,3000,4000,8000, 10000,20000,40000, 60000]

random.seed(12345)
npy = False
nreps = 50
f = p
times_list =findTimes(f,size,nreps,npy)

# run for function foo
random.seed(12345)
nreps = 5
f = foo
times_list_foo =findTimes(f,size,nreps,npy)

# repeat for numpy
random.seed(12345)
npy = True
nreps = 50
f = p
times_npy =findTimes(f,size,nreps,npy)

random.seed(12345)
nreps = 5
f = foo
times_npy_foo = findTimes(f,size,nreps,npy)

# take ratio foo/p
import numpy as np
ratio = np.array(times_list_foo[1])/np.array(times_list[1])
ratio_npy = np.array(times_npy_foo[1])/np.array(times_npy[1])

plt.plot(times_list[0],ratio,'-bo', label="list")
plt.plot(times_npy[0],ratio_npy,'-go', label="numpy array")
# Place a legend above this legend, expanding itself to
# fully use the given bounding box.
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
```

```
           ncol=2, mode="expand", borderaxespad=0.)

plt.xlabel('size of list/numpy array')
plt.ylabel('average time (s)')

plt.show()


######## Exercise 4 #############################################

def bar(a,i,j,k):
    if j-i==1:
        return a[i]
    q = p(a,i,j);
    if k<q:
        return bar(a,i,q,k)
    else:
        return bar(a,i,q,k)

# plot bar vs n
f=bar
minsize = 10000
maxsize = 1000000
stepsize = minsize*10
nreps = 10
k = 100

import random
import time # I prefer using time.perf_counter rather than timeit as shown
below
import numpy as np

avg_time = []
size = list(range(minsize,maxsize,stepsize))

random.seed(12345)
# iterate for different size of array
for n in range(minsize,maxsize,stepsize):

    # generate list of size n or  numpy array
    if npy:
       l = np.array(range(n))
    else:
       l = list(range(n))

    totalTime = 0

    #  repetitions random shuffle
    for rep in range(0,nreps):
        random.shuffle(l) # randomize the list
        timeStamp = time.process_time() # get the current cpu time
        f(l, 0, n,k) # run p function
        timeLapse = time.process_time() - timeStamp
        totalTime += timeLapse

    # store average time
    avg_time.append(totalTime/nreps)
```

```python
    print('p time: n[{0}]={1}'.format(n, totalTime/nreps))


# plot avg time vs size of list
import matplotlib.pyplot as plt

plt.plot(size,avg_time,'-bo', label="list")

plt.xlabel('size of list n')
plt.ylabel('average time (s)')

plt.show()

# plot bar vs k
f=bar
minsize = 10000
maxsize = 1000000
stepsize = minsize*10
nreps = 10
n = 100000

import random
import time # I prefer using time.perf_counter rather than timeit as shown
below
import numpy as np

avg_time = []
size = list(range(minsize,maxsize,stepsize))

random.seed(12345)
# iterate for different k
for k in range(minsize,maxsize,stepsize):

    # generate list of size n or  numpy array
    if npy:
       l = np.array(range(n))
    else:
       l = list(range(n))

    totalTime = 0

    #  repetitions random shuffle
    for rep in range(0,nreps):
        random.shuffle(l) # randomize the list
        timeStamp = time.process_time() # get the current cpu time
        f(l, 0, n,k) # run p function
        timeLapse = time.process_time() - timeStamp
        totalTime += timeLapse

    # store average time
    avg_time.append(totalTime/nreps)

    print('p time: n[{0}]={1}'.format(n, totalTime/nreps))

# plot avg time vs size of list
import matplotlib.pyplot as plt
```

```
plt.plot(size,avg_time,'-bo', label="list")

plt.ylim([0,1])
plt.xlabel('k')
plt.ylabel('average time (s)')

plt.show()
```