1. Find the Peak Entry in the unimodal list.

* Algorithm Description

Using a derivation of the binary search algorithm. My algorithm finds the Peak Entry in the Unimodal list in an O(log(n)) time complexity.

* Pseudocode

Given a unimodal list A

min = 1

max = n # assuming n is the length of the list

while min < max: # pointers have not crossed

mid = floor((min + max) / 2) # get middle index

if A[mid] < A[mid+1]: # if the mid point is less then the next point

min = mid + 1 # move the min pointer and keep looking

elif A[mid] > A[mid+1]: # if the mid point is greater then the next point

max = mid # move the max pointer and keep looking

return A[min] # once min and max are pointing at the same thing we know we have found peak

* Correctness

The search for the peak number only stops when the left and right have met. If they have not met their points are shifted closer and closer to the peak number.

* TIme Analysis

Because my algorithm is basically a binary search it can be assumed that the time complexity is the same. O(log(n))

1. SELECTION algorithm

In the median of medians algorithm the choice of grouping items in the list by 5 is arbitrary. As long as you group items in groups of 3 or more it should work fine, Only the constant values in the time complexity would change. However if you choice a group size that is close to the size of your original array or is the same size as your original array then you loose the time complexity of O(n) and go to O(n\*log(n)).

1. Oil company

* Algorithm Description

In Stat 3000 we learned about how to find the Line of Best Fit with the Least Squares method. My algorithm does exactly that algorithm because it's the equation to find the line to fit all the points.

* Pseudocode

Given an array of all the points (where the wells are)

x\_mean = 0

y\_mean = 0

for point in array:

# assuming that p[0] is the x value and p[1] is the y value

x\_mean += p[0]

y\_mean += p[1]

x\_mean /= len(array)

y\_mean /= len(array)

temp1 = 0

temp2 = 0

for p in array:

temp1 += (p[0] - x\_mean) \* (p[1] - y\_mean)

temp2 += (p[0] - x\_mean) ^ 2

m = temp1 / temp2

b = y\_mean - (m \* x\_mean)

return f’y = {m}x + {b}’

* Correctness

I used the algorithm I got from my stat3000 class so I am assuming the equations are correct. I have programmed my algorithm in python and used a module to plot the points and my fit line. The number of points above and below the line are either the same or they are off by 1 (meaning there is an odd amount of points).

* TIme Analysis

My algorithm has two for loops that are not nested. So I have an O(n+n) -> O(2n) -> O(n).

1. Multiple Selection
2. O(n\*log(n))

* Algorithm Description

Sort the array A and look up the m values.

* Pseudocode

# if A’s first index starts at 0

A.sort()

for k in m:

print(A[k-1])

* Correctness

This algorithm would be able to find all the m values in array A

* TIme Analysis

Because we sort array A then do a look up the time complexity would be O(n\*log(n)+m) -> O(n\*log(n))

1. O(nm)

* Algorithm Description

For every k in m, call the normal selection algorithm.

* Pseudocode

s = []

for k in m:

s.append(normal\_selection(A, k))

return s

* Correctness

If the normal\_selection algorithm works by using the median of medians strategy to get a pivot then the time complexity would be O(n) for every k in m

* TIme Analysis

Thus the time complexity of the algorithm is O(nm)

1. O(n\*log(m))

* Algorithm Description

To get O(n\*log(m)) the algorithm will need to cut down the amount of numbers to check at each level. First we get the pivot point and get our left and right lists. Then for every k in m check to see if the median is in the right position, or if the correct position should be in the right or left lists. Then call the normal selection algorithm to find the kth item for that k.

* Pseudocode

function get\_m\_ks(A, m, n, result): # n is the number of k’s in m at the start

get the pivot using the median of medians algorithm

split A into left and right lists based on the pivot

leftM = []

rightM = []

for k in m:

if the position of pivot is at the right place for k:

result.append(pivot)

continue

else if the position of pivot is to small for k:

k = k - (len(A) + 1)

rightM.append(k)

else:

leftM.append(k)

if len(result) == n:

return result

if len(rightM) > 0:

return get\_m\_ks(right, rightM, n, result)

if len(leftM) > 0:

return get\_m\_ks(left, leftM, n, result)

* Correctness

I have implemented this code in python and it does indeed find the right numbers for every k in m. It also makes sense that it would work based on the fact that I am just limiting what part of A should be searched for every k.

* TIme Analysis

The time complexity for finding the pivot is O(n)

The time complexity for splitting A into left and right is O(n)

The time complexity for checking every k in m is O(log(m))

Thus the overall time complexity is O(n\*log(m))

1. Selection bonus

def get\_Kth(A, k):

if k > len(A):

return None

if len(A) % 2 == 0:

median = get\_median(A, len(A) // 2)

else:

median = get\_median(A, (len(A) // 2) + 1)

left, right = split\_A(A, median)

if k == len(left) + 1:

return median

elif k > len(left) + 1:

return get\_Kth(right, k - (len(left) + 1))

else:

return get\_Kth(left, k)

def get\_median(A, k):

groups = get\_groups(A)

median = get\_median\_of\_medians(groups)

left, right = split\_A(A, median)

if k == len(left) + 1:

return median

elif k > len(left) + 1:

return get\_median(right, k - (len(left) + 1))

else:

return get\_median(left, k)

def split\_A(A, median):

left = []

right = []

for a in A:

if a > median:

right.append(a)

else:

left.append(a)

left.remove(median)

return left, right

def get\_median\_of\_medians(groups):

medians = []

for group in groups:

group.sort()

medians.append(group[len(group) // 2])

medians.sort()

return medians[len(medians) // 2]

def get\_groups(A):

groups = [[]]

count = 0

for a in A:

if len(groups[count]) < 5:

groups[count].append(a)

else:

count += 1

groups.append([])

groups[count].append(a)

return groups