**Algorithm Description**

* Starting at newSelect(), the variable input\_file is assigned to and opens input.txt to read its contents.
* Then, input\_args is assigned to reading the contents of input\_file.
* num\_files is assigned the first value, input\_args[0].
* arr\_length is assigned the second value, input\_args[1].
* kth\_smallest is assigned the third value, input\_args[2].
* The array variable indexes[] is then created. This variable holds the minimum and maximum location or range of values that the algorithm is using to determine the kth element.
* With a for-loop, the length of each array in the .dat files is stored in the array variable indexes[]. The amount of elements in indexes[] is equal to the amount .dat files and each element has the same value in the beginning.
* read\_file is then called, initially sending it file number zero and half the value of arr\_length.
* Inside read\_file filename is assigned to file number zero plus one, with .dat joined to the of the string.
* The first iteration, filename 1.dat is read in binary mode.
* Variable v is assigned to index (index = arr\_length at the call to read\_file).
* size\_ is assigned the value of 4, which is required to index the binary numbers in the .dat files.
* Until variable v \* size\_ is located (while True), the open 1.dat file will be parsed through.
* If no exception occurs, the value of the binary number at v \* size\_ is read, converted from binary to decimal and returned to newSelect and assigned to the variable mid\_value.
* If an exception occurs, -1 is returned and assigned to the variable mid\_value in newSelect.
* Back in newSelect, the function split is called with the array variable indexes, and int variables num\_files, mid\_value, 0, 0, and kth\_smallest.
* In split, a for-loop is started from zero to m (m = amount of arrays), getLength is called and is getting passed the variable indexes. Inside the function getLength the minimum index is subtracted from maximum and then 1 is added to it to adjust for element position, and this value is returned to split.
* If that value is greater than 1, then split breaks.
* Else if, the index counter of the for loop is equal to m (m = amount of arrays) and getLength returns a value less than or equal to 1, this means that all arrays are size 1 and are needing to be merged, sorted, and then the kth element is printed to output.txt.
* A for-loop from zero to the amount of arrays calls binarySearch, which returns the mid and that value is held in the variable index. Each index is appended into the lower\_indeces array. This is storing the index of where the mid is found in each array.
* The variable less is incremented by the index value subtracted by the indexes[i][0] value. This is the distance of the lowest value from the mid, still of interest. The variable greater is incremented by one plus the subtraction of index from indexes[i][1]. This is the distance of the highest value from the mid, still of interest.
* If the kth value is less than or equal to the variable less, then remove the upper half of the array. Recursively call split until the kth element is found.
* If the kth is greater than the variable less, the remove the lower half of the array and decrement k. recursively call split until the kth element is found.

**Run Time Analysis**

O(1)

O(1)

O(1)

O(1)

O(1)

O(1)

O(m)

O

O

def newSelect():

    input\_file = open("input.txt", "r")

    input\_args = input\_file.read().split(',')

    num\_files = int(input\_args[0])

    arr\_length = int(input\_args[1])

    kth\_smallest = int(input\_args[2])

    indexes = []

    for i in range(0, num\_files):

        indexes.append([0, arr\_length-1])

    mid\_value = read\_file(0, int((arr\_length-1)/2))

    split(indexes, num\_files, mid\_value, 0, 0, kth\_smallest)

def read\_file(file\_number, index):

O(1)

O(1)

    filename = ''.join([str(file\_number+1), ".dat"])

    with open(filename, mode='rb') as file:

        v = index

        size\_ = 4

        while True:

            file.seek(v \* size\_)

            try:

                return struct.unpack('>I', file.read(size\_))[0]

            except:

                return -1

def split(indexes, m, mid, less, greater, k):

O(n)

O(m)

O(log n)

O(m)

O(1)

O(1)

O(1)

O(m)

O(m)

O(1)

O(1)

O(1)

O(m)

for i in range(0, m):

if getLength(indexes[i]) > 1: # size > 1

break

elif i == m-1 and getLength(indexes[i]) <= 1: # all arrays are size 1

# merge all arrays and find kth elements

merged = []

for i in range(0, m):

merged.append(read\_file(i, indexes[i][0]))

f = open("output.txt", "w+")

f.write(str(mergesort(merged)[k-1]))

exit(0)

lower\_indeces = []

for i in range(0, m):

index = binSearch(i, mid, indexes[i][0], indexes[i][1]) + 1

lower\_indeces.append(index - 1)

less += index - indexes[i][0]

greater += (indexes[i][1] - index) + 1

if k <= less: # chop off upper half of array

for i in range(0, m):

if getLength(indexes[i]) >= 1:

indexes[i][1] = lower\_indeces[i]

elif getLength(indexes[i]) == 1:

if read\_file(i, indexes[i][0]) <= mid:

k += 1

split(indexes, m, getMid(indexes), 0, 0, k)

if k > less: # now k > L so chop lower half of array and reduce k

k -= less

for i in range(0, m):

if getLength(indexes[i]) >= 1:

indexes[i][0] = lower\_indeces[i] + 1

elif getLength(indexes[i]) == 1:

if read\_file(i, indexes[i][0]) <= mid:

k += 0

split(indexes, m, getMid(indexes), 0, 0, k)

Each iteration through the function, we go through each array and binSearch for the middle element of the largest array. Ideally, we hope to cute each array in half here and disregard all the elements greater than mid (if the sum of all elements less than mid is smaller than k). Although, it is possible that we only are able to remove one element from one array, in the worst case. So we are left with the fact that each iteration through our algorithm we

double the number of arrays => 2 \* m = 2m

and remove one element from those arrays:

So we are left with a recursive step with a new size of:

And although this is not an ideal reduction to the problem size, it is smaller than the previous iteration.

So, we binSearch with is log(n) time for each array:

Then call the recursive step so:

Which will happen for m \* n times in the worst case:

Leaving us with worst case runtime of: