Purpose of the Program:
Sorting arrays and comparing the efficiency of each type of sorting algorithm. Difference can be seen in the amount of moves and compares in number of elements provided.

Insertion Sort:

How does it work?

1. Work from left to right
2. Examine each item and compare it to items on its left
3. Insert the item in the correct position in the array

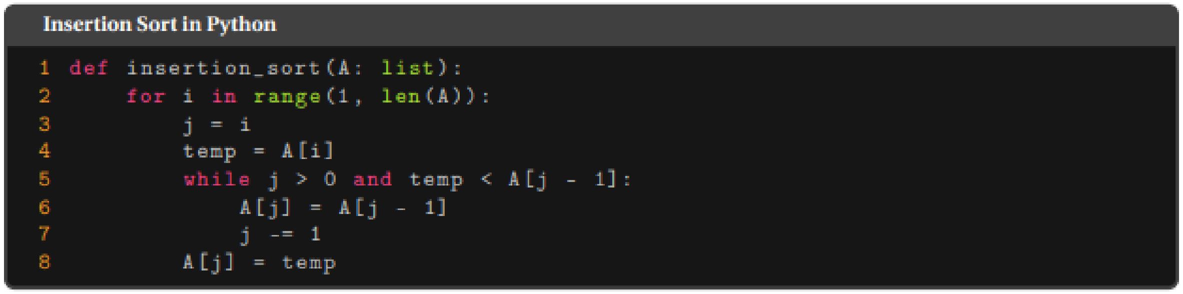
For the element being checked, you'll want to compare it to the previous element and continue doing that until the element is in the right position.

Also, there are 2 important rules to keep track of

Assume an array of size *n*, for each k in increasing value from 1<=k<= n. Check if A[k] is in the correct order by comparing the element A[k-1]

1. A[k] **is in the right place. Means that A[k] is greater or equal to A[k-1], then proceed to the next element**
2. **A[k] is in the wrong place. A[k-1] is shifted up to A[k], and the original value of A[k] is further compared to A[k-2].**

Pseudo-code:


Insertion Sort in Python

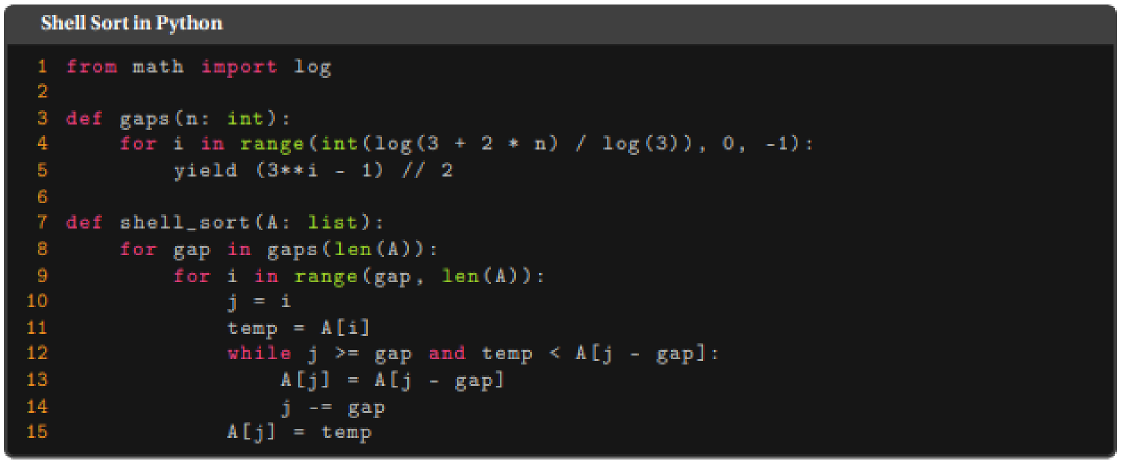
```python
def insertion_sort(A: list):
    for i in range(1, len(A)):
        j = i
        temp = A[i]
        while j > 0 and temp < A[j - 1]:
            A[j] = A[j - 1]
            j -= 1
        A[j] = temp
```

```
insertion(A_list)
    For i = 1,  i < A_list, i += 1
        store_i_counter = i
        element_of_current_i = A[i]
        While store_i_counter >0 and element_of_current_i < A[store_i_counter - 1]
    #previous element
            A[store_i_counter] = A[store_i_counter - 1]
```

```
                    Store_i_counter -= 1
                A[store_i_counter] = temp
```

A list will be passed through the insertion function. Use a for loop to keep track of the current element that we have to check and while loop to check the previous element and how many times we have to move it down

Shell sort:



Shell Sort in Python

```python
from math import log

def gaps(n: int):
    for i in range(int(log(3 + 2 * n) / log(3)), 0, -1):
        yield (3**i - 1) // 2

def shell_sort(A: list):
    for gap in gaps(len(A)):
        for i in range(gap, len(A)):
            j = i
            temp = A[i]
            while j >= gap and temp < A[j - gap]:
                A[j] = A[j - gap]
                j -= gap
            A[j] = temp
```

```
gaps(n element){
 For i = log(2n+3) / log(3), i > 1 , i -= 1
        Return (3^i - 1.0) / 2
}

shell_sort(A_array){
        For gap = 0, gap < gaps[gap], gap += 1
                For  i  = 0, i < gap and i < array length
                        j = i
                        Temp = A[i]
                        While j >=gap and temp < A[j - gap]
                                A{j] = A[j - gap]
                                J -= gap
                        A[j] = temp
}
```

From the gaps we can see that the largest number we go for is the log equation and we return a value

The generator can be replicated if we just create a for loop inside the function and call the return

For the sorting array itself I can kind of see how we're just matching the least with the greatest and moving them based on that.

Heap Sort:
-binary tree

2 kinds of heaps

1. Max heap: parent node must have a value that is greater than or equal to the value of its children
2. Min heap: any parent node must have a value that is less than or equal to the value of its children

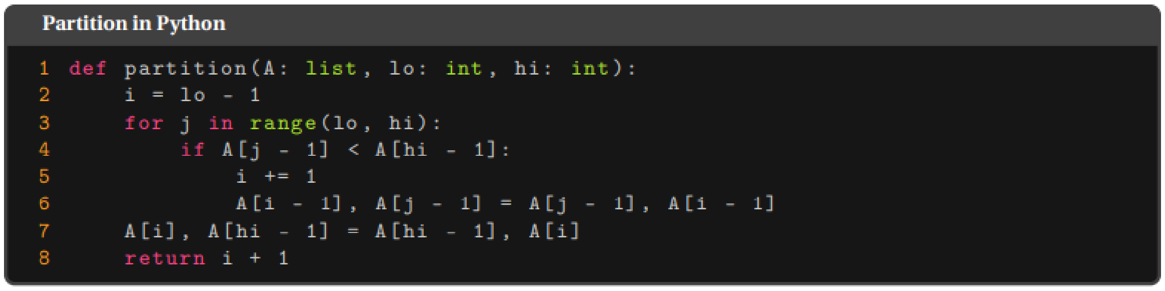
2 routines:

1. Building a heap: Taking the array to sort and building a heap from it. Constructed heap will be a max heap. This means that the largest element, the root of the heap, is the first element of the array from which the heap is built
2. Fixing a heap: The second routine is needed as we sort the array. The gist of heapsort is that the largest array elements are repeatedly removed from the top of the heap and placed at the end of the sorted array, if the array is sorted in increasing order. If you remove the largest element from the heap, fix the heap.

```
Import bool

max_child(A_array, first_int, last_int)
        Left = 2 * first
        Right = left + 1
        If right <= last && A[right -1] >A [left-1]
                Return right
        Return left

fix_heap((A_array, first_int, last_int)
        Found = false
        Mother = first
        Great = max_child(Array, mother, last)

        While mother <= last/2 %% not found (or found = true)
                If A[mother -1] < A[great-1]
                        A[mother-1], A[great-1] =A[great-1],A[mother-1]
                        Mother = great
                        Great = max_child(A,mother,last)
                Else:
                        Found = true
```

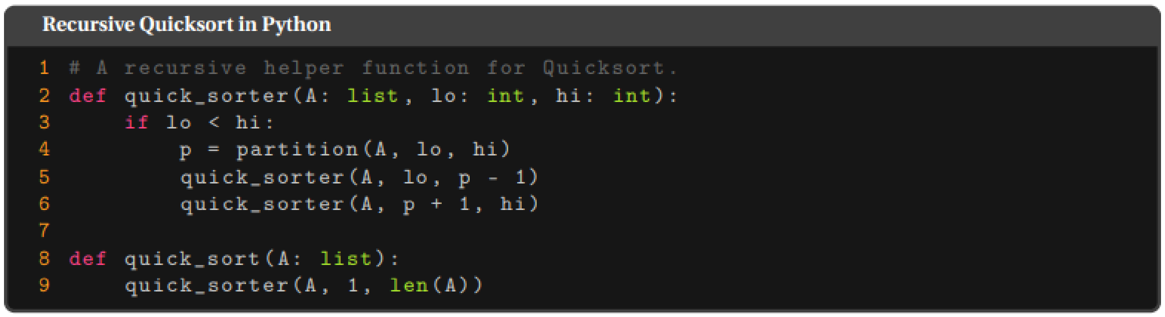
These functions are required to actually be used in the main heap_sort function

Quick sort:



Partition in Python

```
def partition(A: list, lo: int, hi: int):
    i = lo - 1
    for j in range(lo, hi):
        if A[j - 1] < A[hi - 1]:
            i += 1
            A[i - 1], A[j - 1] = A[j - 1], A[i - 1]
    A[i], A[hi - 1] = A[hi - 1], A[i]
    return i + 1
```



Recursive Quicksort in Python

```
# A recursive helper function for Quicksort.
def quick_sorter(A: list, lo: int, hi: int):
    if lo < hi:
        p = partition(A, lo, hi)
        quick_sorter(A, lo, p - 1)
        quick_sorter(A, p + 1, hi)

def quick_sort(A: list):
    quick_sorter(A, 1, len(A))
```

And for sorting.c should be pretty similar to the previous sorts but more research needed for taking in number inputs from the command prompt.

Files in this lab:

Heap.c
Heap.h
Insert.c
Insert.h
Quick.c
Quick.h
Set.h
Shell.c
Shell.h
Stats.c
stats.h

For each of the sorting algorithms here, we are going to be **NEEDING** stats.h and stats.c to keep track of our moves and comparisons. Simply call the function inside the sorting function when a comparison or move has happened. Be aware during conditions, most likely to use compares and moves and you are moving the element from one index to another.

Another addition to this lab, sorting.c is a little different from the previous lab because now we will have to implement set.h instead of using the boolean. We have to add the command prompt that is for sorting algorithms into the set. Next, we can use it to check by seeing if it’s a member of the set then do the sorting algorithm.