General Idea

- ★ We are to implement shell sort, bubble sort, quick sort, and heap sort algorithms in C using the information from the assignment and the Python pseudocode given to us.
- ★ We want to return an array (modify the given array).
- ★ Allocate memory dynamically using the C Standard Library functions `malloc()`, `calloc()`, and `free()`. These functions are defined in the `<stdlib.h>` header file.
- ★ Each sort function takes a pointer to a Stats struct as the first parameter, the array of uint32_ts to sort as the second parameter, and the length of the array as the third parameter.
- ★ For shell sort, it sorts pairs of elements which are far apart from each other.
    - The distance between these pairs of elements is called a gap. The "gap sequence" determines the initial size of the gap and how it is made smaller with each iteration.
    - Each iteration will decrease the gap until a gap of 1 is used.
    - Use a for-loop function that computes the next game from the current gap.
- ★ For bubble sort, we look at two values that are right next to each other. If the one after the first one is smaller, swap them.
    - The largest value will be at the end of this list.
        - This means we will only need to consider *n*-1 items. This pattern will continue into *n*-2, *n*-3, etc.
    - If we don't need to change anything after passing through the entire array, the array is sorted and nothing needs to be done.
        - Use a variable that is set to `true` before each pass and is set to `false` when swapping occurs.
    - We require an inner loop and an outer loop.
        - The inner loop will determine the O(n) comparisons.
- ★ For quick sort, it partitions arrays into three sub-arrays by selecting an element from the array and designating it as a pivot.
    - Shell sort will be used when the input sub-array becomes sufficiently small.
    - It finds a pivot, an approximation of the index of the middle of the array.
    - Every element less than the pivot should be near the start of the array, and every element greater should be near the end.
    - Needs to be in-place, do not allocate memory.
    - Probably the hardest to do.
- ★ For heap sort, we have to keep in mind building the heap and fixing the heap.
    - The smallest element will be at the root of the heap, at the very top.
    - Take the array and build a heap from it.
    - For each vertex, its children must be larger than it.
    - Children can be equal to each other.
- ★ Create a test harness that will create an array of pseudo random elements using `mtrand_rand64()`. This will be in our file `sorting.c`.
- ★ We will also gather statistics about each sort.
    - size of the array
    - number of moves required
    - number of comparisons required
- ★ Our test harness must support of the following command-line options (we must use a set for this):

- -a : Employs all sorting algorithms.
- • -s : Enables Shell Sort.
- -b : Enables Bubble Sort.
- -q : Enables Quick Sort.
- -h : Enables Heap Sort
- -r seed : Set the random seed to seed. The default seed should be 13371453.
- -n size : Set the number of elements in the array to size. The default size should be 100. The range of valid input is $1 \leq n \leq 250,000,000$ where n is the size specified.
- -p elements : Print out `elements` number of elements from the array. The default number of elements to print out should be 100. If the size of the array is less than the specified number of elements to print, print out the entire array and nothing more.
- -H : Prints out program usage. See reference program for example of what to print.
- Our test harness must print in the exact format the pdf tells us to.
- Each of your sorts for a given program run should sort the same pseudo random array. Make use of `mtrand_rand64()`.
  - The numbers generated should be bit-masked to fit in 30 bits. Use bitwise AND.

★ We must use a small statistics module to gather data.

<u>Pseudocode</u>

★ Shell sort
- Create a function for the gaps.
  - Create a while loop that will only terminate once n is greater than 1.
    - If n is less than or equal to 2, n will equal 1.
    - Else, 5 will be multiplied to n, then divided by 11.
    - Create a variable to hold n.
      - This will be used again to create the next gap.
- Create a function that will do the shell sorting. It will take in an array.
  - Use a for-loop that will use j as the stepping and starting condition. The termination will be once it has gone through the array. We will be calling the function gap and putting in the size of the array.
    - Create another for-loop that will use h as a starting and stepping condition. It will terminate when the range between j and the size of the array is gone through.
      - Create a variable, f, to equal h.
      - Create a temporary variable called temp to equal array[h].
      - Create a while-loop that's condition is that f is greater than or equal to j AND temp is less than array[f - j].
        - array[f] will equal array[f-j].
        - f will be a new value after f is subtracted from j.
      - array[f] will now equal the temporary variable.

★ Bubble sort
- Create a function for bubble that takes in a, an array.

- Create a temporary variable that will take in the first variable that needs to be swapped, so that the second variable can go into the first one, then the temporary variable's value will go into the second.
- Create a variable that will decrease the amount of time the loops need to search through the lengths.
    - Create a for-loop that uses i as the variable in the starting and stepping, the termination will be if i is equal to the size of the array minus a variable.
        - The variable swapped will be placed as false.
        - Create a for-loop that uses j as the variable in the starting and stepping, the conditional will be if i is equal to the size of the array minus a variable.
            - Using j as an index, create an if-statement to see if a[j] is less than a[j-1].
                - If it is, then place a[j] in the temporary variable, place a[j-1] into a[j], then place the temporary variable in a[j-1].
                - The variable swapped will equal true.
        - Add 1 to the variable that will decrease the lengths.
        - If swapped is false still, break out of the loop.

★ Quick sort

- Import our shell sort.
- Define small as an integer of 8.
- Create a function called quicksort that takes in an array, a.
    - If the size of a is less than small.
        - Return after doing shell sort for a.
    - To find the pivot variable, use the first value of the array.
    - Use a variable to hold the left mark for the array, which will be the first, a variable, + 1.
    - Use a variable to hold the right mark for the array, which will be the last value in the array.
    - Set a boolean variable to false to indicate that we are not done yet.
    - Create a while-loop that will only terminate if the done variable is true.
        - Create a while-loop that will check if the left mark is less than or equal to the right mark AND the value at array index of left mark is greater than or equal to the pivot value.
            - While it is, the left mark will be added with a 1. This will be the new left mark.
        - Create a while-loop that will check if the right mark is greater than or equal to the pivot value AND the value at array index of right mark is less than or equal to than the left mark.
            - While it is, subtract 1 from the right mark and set that as the new right mark.
        - Create an if-statement that checks if the right mark is less than the left mark.

            - done will be set to true.
        - If not, we will swap the value at index left mark and index right mark of the array.
    - Swap the value at array index of the variable first and value at array index right mark.
    - Return the right mark.
- Since we can't change the function that is given to us, we will create a new function that can take in the functions we want. It will take in our array, the amount of elements in it, the first variable, and the last variable.
    - If the amount of elements is less than 8, just go ahead and do the shell sort for the array.
    - If the first variable is less than the last variable…
        - Create a variable called split point that will call the partition function that we've created earlier.
        - Recursively call the function we're currently in and give it the array, the amount of elements, the first variable, and split point variable minus 1.
        - Recursively call the function we're currently in and give it the array, the amount of elements, the split pointer variable plus 1, and the last variable.
- The main quick sort function goes here.
    - Initialize first to 0 and last to equal the amount of elements minus 1.
    - Call the recursive function.

★ Heap sort

- Create functions for the left and right child that take in n. Make the left one return 2 * n + 1. Make the right one return 2 * n + 2.
- Create a function for the parent that takes in n. Make it return (n - 1) divided by 2.
- Create a function for values to go up the heap that takes in (a, n).
    - Create a while-loop that detects if n is above zero and that a[n] is less than a[parent(n)].
        - Create a temporary variable that will hold a[n] so that a[parent(n)] will go into a[n]. The temporary variable will be put into a[parent(n)].
        - Then n will equal parent(n).
- Create a function that will go down the heap, it takes in a and the heap's size.
    - n will equal 0.
    - While the left child is less than the heap size…
        - If the right child is equal to the heap size, then the bigger variable will be given to the left child.
        - Else, if a[l_child(n)] is less than a[r_child(n)] , then bigger will equal the left child.
            - Else, the right child will be equaled.
        - If a[n] is less than a[bigger], break out of the loop.
        - Create a temporary variable that will hold a[n]. Place a[bigger] into a[n], then place the temporary variable's value into a[bigger].
        - n will then equal bigger.

- Create a function to build the heap.
  - The variable, heap, will equal the length of the array by finding the size of the array. Allocate memory for this.
  - Create a for-loop that will have i as the starting and stepping condition. The loop will terminate when i has gone through the size of the array.
    - heap[n] will equal a[n].
    - We will call the function that goes up the heap and give it heap and n.
    - Return heap.
- Create a function to sort the heap. It will take in a, an array.
  - The variable, heap, will equal what is returned when we call the build heap function.
  - Create a variable for the sorted array. We will get the length of it by finding the size of the array. Allocate memory for this.
  - Create a for-loop that will use k as the starting and stepping condition. The loop will terminate once it has gone through all the values in the array.
    - Create a variable that will hold heap[0]. Place sorted_list[n] into heap[0] and place the temporary variable's value into heap[size of the array minus 1].
    - Call the function that goes down the heap and give it heap and the size of a minus n.
  - Return the sorted array.

★ Test harness

- getopt() will read the command-line options given to by the user. It will take in these values and do the corresponding action:
  - -a : Employs all sorting algorithms.
    - We will set the bits so that it can read to the program that we want to use all the sorting algorithms.
  - -s : Enables Shell Sort.
    - We will set the bits so that it can read to the program that we want to use the shell sorting algorithm.
  - -b : Enables Bubble Sort.
    - We will set the bits so that it can read to the program that we want to use the bubble sorting algorithm.
  - -q : Enables Quick Sort.
    - We will set the bits so that it can read to the program that we want to use the quick sorting algorithm.
  - -h : Enables Heap Sort
    - We will set the bits so that it can read to the program that we want to use the heap sorting algorithm.
  - -r seed : Set the random seed to seed. The default seed should be 13371453.
    - We will put in the value to the seed variable if the user gives us one. If not, the variable will be set to 13371453 by default when we initialize it.

        - -n size : Set the number of elements in the array to size. The default size should be 100. The range of valid input is $1 \leq n \leq 250,000,000$ where n is the size specified.
            - Use boolean logic to figure out if the user's input is between 1 and 250,000,000 inclusive. If not, put it to 100.
        - -p elements : Print out elements number of elements from the array. The default number of elements to print out should be 100. If the size of the array is less than the specified number of elements to print, print out the entire array and nothing more.
            - Use a while-loop to print out the amount that the user wants. The termination of the while-loop is when it reaches the size of the array. We would also need to subtract 1 from the user's input, or the default, since we need to access the first element in the array.
        - -H : Prints out program usage.
            - Write down what each option does and what values can be optionally placed after some of them. If this option is pressed with any other option (or by itself) or an invalid option has been pressed, this statement will be printed out. We will use a boolean to check if any of the situations has happened.
    - We will be using mtrand_rand64() to get some of the values for testing. We will need to mask this value to not get negative values.
    - To see which algorithms we need to do, check the sets to see what will be true.
    - Set memory for the array, using our default or what the user has given to us for the array.
- ★ Gathering stats
    - Using stats.c, we will place these variables and functions in the appropriate places to gather things such as the size of the array, the number of moves required, and the number of comparisons. These will be placed in for-loops and while-loops to get accurate results.
    - Use the reset function to reset the moves and comparisons to 0.