In this assignment we are implementing different types of sorting orders.

There are four types of sorts to implement: Insertion sort, Heap sort, Quick sort, and shell sort.

1. Insertion sort

Insertion sort splits the array into two parts, sorted and unsorted parts. The values from the unsorted part are placed in the correct position in the sorted part.

We can follow the following algorithm:

1. Assume the first element is already sorted
2. Take the second element and store it to ptr
3. Compare ptr with all elements in the sorted array
4. If the element in the sorted array is smaller than the current element, move on to the next element. Otherwise, shift greater elements in the array towards the right
5. Append the value
6. Repeat these steps until the array is fully sorted

We can use the pseudo code below:

```
for i = 1 to length(array)
        insert = A[i]
        position = i;
        while position > 0 and A[position -1 ] > insert do:
                A[position] = A[position -1]
                position = position - 1;
        A[position] = insert
```

2. Shell sort

Shell sort is a variation of insertion sort. It sorts pairs that are far away from each other. The distance between the elements is called a gap and we use the highest value of a gap initially and decrease the gap until it is 1. We use the gap to swap the necessary elements and sort them.

1. Find the gap size by computing 3**k - 1 with the largest k being log (2n + 3) / log(3), n = length
2. Divide the array into subarrays of equal gap size
3. Apply insertion sort of the subarrays
4. Keep iterating until the gap size reaches 0

We can follow the pseudocode below:

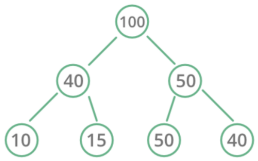
```
gap size()
        for each element of the list that is gap apart starting from log (2n + 3) / log(3)
                return gap size of 3 ** i - 1  / 2

shell sort()
        size = 0
        gap = gaps(n, size)
        for each element of the list to the size
                for each element in the list to the length of array
                        t  = l;
                        ptr = A[l]
                        while t is greater than gap[r] and ptr < A[t - gap[t]]
                                A[t] = A[t-gap[r]]
                                t-=gap[r]

                        A[t] = ptr
```

3. Heap Sort

A heap is tree structures used for sorting and priority queues. They are complete binary trees if they fulfill having a maximum of 2 node children, every level is filled except the leaf nodes, and every child is left of his parents. Following is an example of a max heap.



Max Heap

There are two critics in order to implement heap sort:

Building a heap

- We will be building a max heap which means that the parent node is always greater than both the child nodes.

1. Create a new node at the root of the heap
2. Assign a value to it
3. Compare the value of the child to the parent node

4. Swap those nodes if the value of the parent is less than either of the child
5. Repeat these steps until the toor parent nodes is the largest element

Fixing a heap

- We need to fix the heap in order to sort the array. The largest element towards the parents is placed into the sorted array. We can do this by simply setting our nodes to be implemented into an array. Once the largest element from the heap is removed, the heap needs to be fixed.

We can follow the provided pseudocode of heap sort:

```
max child()
        left = 2 * beg
        right  = left + 1
        if right <= end and A[right - 1] > A[left -1]
                swap A[right-1] and A[left -1]
                repeat swap
                return right
        return left

fix heap()
        set boolean variable called disc to false
        first = beg
        high = max_child(array, first, end)
        while first <= end / 2 and not disc
                if A[first - 1] < A[high - 1]
                        swap A[first - 1] A[high-1]
                        first = high
                        high = max_child(array, first, end)
                else
                        disc = true


build heap(array, beg, end)
        for i = end / 2 decrementing by one until i > beg - 1
                fix_heap(array, i end)


void heap_sort()
        beg = 1
        la  = n
```

```
build_heap(array, beg, la)
for la = n decrement by 1 until la > beg
        swap A[beg-1]A[la-1]
        fix_heap(array, beg, la - 1)
```

4. Quick Sort

Quick sort is a divide and conquer algorithm where it creates two empty arrays. One to hold elements less than the pivot. The other for elements greater than the pivot while recursively sorting the sub arrays. There are two key components in the algorithm which are swapping the elements and partitioning the array.

1. Assign a pivot to any element in the array
2. Partition the array on the basis of the pivot
3. Partition the left recursively using quick sort
4. Partition the right recursively using quick sort

We can follow the pseudocode below:

```
partition(array, min, n)
        i = min - 1
        for j = min until j < n and incrementing j
                if A[j-1] > A[n-1]
                        increment i
                        swapo A[i-1] and A[j-1]
        swap[A[i] and A[n-1]
        return i + 1

quick_sorter(array, n)
        if (min < n)
                pi = partition(array, min, n)
                quick_sorter(array, min, pi - 1)
                quick_sorter(array, pi + 1, n)

Quick_sort (array, n)
        quick_sorter(array, 1, n)
```

We want to implement all the sorts above given the header files of heap.h quick.h insert.h and shell.h. We will also be gathering statistics of each sort which should find the size of the array and calculate the number of moves and comparisons required.

In our main function of sorting.c

We want to support the following command line options read by getopt()

-a : Employs all sorting algorithms.

-e : Enables Heap Sort.

-i : Enables Insertion Sort.

-s : Enables Shell Sort.

-q : Enables Quicksort.

-r seed : Set the random seed to seed. The *default* seed should be 13371453.

-n size : Set the array size to size. The *default* size should be 100.

-p elements : Print out the 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.

Sets

We will be using sets to keep record of which command line options are specified when running the program. The code for sets is given in set.h. We will set or clear bits in the set from the command line.

We will use the following sets in our program:

set empty_set()
    This function returns an empty set where all the bits are equal to 0.

bool member_set()
    This function checks whether the given value x is present in the set s.

set insert_set()
    This function inserts x into s.

We will randomize each number in the array while applying a bit masked to fit in 30 bits.

The output should print the sort name, the number of elements in the array, and the moves and compare values. It should also print out the sorted array elements within 5 columns and a width of 13 for spacing each column.

Stats

We are given a statistics module called stats.h and stats.c which includes these functions to compare, swap, and move elements. When using compare, swap, or move it is automatically incrementing each time.

The cmp(stats, x, y) compares x and y and returns the increments of the comparisons field in stats. I plan to use the cmp statement comparing two elements most likely in an if statement.

The move(stats, x) statement is used when elements are stored in a temporary variable. I plan to mainly use move in shell and insertion sort where I do not want to swap but instead move elements. This also increments the move field in stats.

The swap(stats, x, y) statement swaps two elements using pointers. I will use this whenever I want to swap two elements.

Sorting.c

1. Create a typedef enum initializing all sorts
2. Set default size, seed, elements
3. Create while loop using getopt()
4. Switch case in the while loop to support each command line option. Insert the sort using insert_set when calling that sort. For example case ‘i’ would be s = insert_set(INSERTION, s);
5. Break out of the while loop and create a for loop that starts from the first element in the typedef enum array looping to the last element.
   For example: for x = heap x <= heap increment x by 1
6. Initialize the moves and compares to 0
7. Allocate memory using calloc
8. Randomize each number in the array applying a bit mask of 1073741823
9. Check if each sort is in the member set by doing if (member_set(x,s))
10. If the x value of the for loop equals one of the indices of the array then call that sort. For example: if (x == 0) then call heap_sort(). Repeat for all sorts.
11. Print the size, moves, compares, and sorted elements.
12. Make sure to free up space.

Create a makefile