CSE13s Fall 2021
Assignment 3 :- Sorting: Putting your affairs in order
By: Ruhin Gharai

The assignment:-

This assignment is about making different sorting algorithms which are insert, heap, quick and shell sort. These programs will be used to sort a random number of arrays and then using graphs will be compared which is more accurate and efficient.

- *insert.c = Insertion Sort :-*

Insertion sort performs two operations: it scans through the list, comparing each pair of elements, and it swaps elements if they are out of order.

Insertion Sort is a sorting algorithm that considers elements one at a time, placing them in their correct, ordered position. Assume an array of size n. For each k in increasing value from $1 \leq k \leq n$ (using 1-based indexing), Insertion Sort compares the k-th element with each of the preceding elements in descending order until its position is found.

Pseudo code:-

```
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
```

- *heap.c = Heap Sort:-*

First find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

The heap data structure is typically implemented as a specialized binary tree. There are two kinds of heaps: max heaps and min heaps. In a max heap, any parent node must have a

value that is greater than or equal to the values of its children. For a min heap, any parent node must have a value that is less than or equal to the values of its children.

The heap is typically represented as an array, in which for any index k, the index of its left child is 2k and the index of its right child is 2k +1. It's easy to see then that the parent index of any index k should be b k 2 c.

Pseudo code:-

```
Def max_child (A: list , first : int , last : int ):
    left = 2 * first
     right = left + 1
    if right <= last and A[ right - 1] > A[ left - 1]:
        return right
    Return left

def fix_heap (A: list , first : int , last : int ) :
    found = False
     mother = first
    great = max_child (A, mother , last )

while mother <= last // 2 and not found :
    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
```

- *quick.c = recursive Quicksort:-*

Quicksort is a divide-and-conquer algorithm. It partitions arrays into two sub-arrays by selecting an element from the array and designating it as a pivot. Elements in the array that are less than the pivot go to the left sub-array, and elements in the array that are greater than or equal to the pivot go to the right sub-array

Pseudo code:-

```
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

# 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) )
```

• stats.c implements the statistics module.

Pseudo code:-

int cmp(Stats *stats, uint32_t x, uint32_t y)
Compares x and y and increments the comparisons field in stats. Returns -1 if x is less than y, 0 if x is equal to y, and 1 if x is greater than y.

uint32_t move(Stats *stats, uint32_t x)
“Moves” x by incrementing the moves field in stats and returning x.
This is intended for use in Insertion Sort and Shell Sort, where array elements aren’t swapped, but instead moved and stored in a temporary variable.

void swap(Stats *stats, uint32_t *x, uint32_t *y)
Swaps the elements pointed to by pointers x and y, incrementing the moves field in stats by 3 to reflect a swap using a temporary variable.

void reset(Stats *stats)
Resets stats, setting the moves field and comparisons field to 0. It is possible that you don’t end up using this specific function, depending on your usage of the Stats struct.

- *shell.c implements Shell Sort.*

The following is the pseudocode for Shell Sort using Knuth's gap sequence.
From the definition of the gap sequence generator, you should see that gaps are computed as b 3 k−1 2 c and the largest k is b log(2n+3) log(3) c, where n is the length of the array to sort. You are iterating over the gap sequence starting with the largest k first, making your way down to a gap size of 1.

Pseudo code:-

```
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
```

- *sorting.c contains main() and may contain any other functions necessary to complete the assignment.*

• insert.c implements Insertion Sort.

• insert.h specifies the interface to insert.c.

- heap.c implements Heap Sort.
- heap.h specifies the interface to heap.c.
- quick.c implements recursive Quicksort.
- quick.h specifies the interface to quick.c.
- set.h implements and specifies the interface for the set ADT.
- stats.c implements the statistics module.
- stats.h specifies the interface to the statistics module.
- shell.c implements Shell Sort.
- shell.h specifies the interface to shell.c.

Design Process:- Trying to convert pseudo code to C .
Shell was difficult because I had gap errors because I forgot to convert int to uint32_t . Then quick was the easiest but sorting.c was not running as it was difficult to create an array to store the elements. Then to print out the random numbers in the array to then sort. During all of this my code got deleted because i forgot to :wq which means i had to write the code all over again. But I figured out my small errors were through not using some libraries and mistakes in defining.

- Describes how comparisons and moves are tracked between all of the sorts

Using the stats.h to call the stats function in my sorting algorithms . The Cmp compares if the element is less than or greater than the previous element. The move function is moving an element one step. The swap function swaps element position used for in heap sort. The reset function is to reset the position of the elements. This insert function compares the elements and then the move function changes the position. In the heap function there four functions use stats , cmp and swap the positions in heap_sort.Quicksort uses stats , comparing and swapping to swap positions of the element quickly after comparing which one is greater then putting it in front of the smaller element. Shell sort uses stats to see the position of the element and uses move function to order the elements from lowest to highest.