Nayeel Imtiaz
CSE 13S
10/14/21

Assignment 3 Design Document

Summary: The purpose of this lab is to implement different sorting algorithms from the command line. There are four different sorting algorithms to be implemented for this lab which are shell sort, insert sort, quick sort, and heap sort. The final executable will be only sorting a dynamically sized array of unsigned integers using the algorithms specified by the user from the command line. This program will also be able to keep track of statistics such as the number of moves, comparisons, and elements in the array.

Pseudocode:

Sorting.c - {This is the main executable to be run}

{The below function randomizes the array with pseudorandom numbers}
Randomizer function(array, size of array, seed)
    Assign seed number to srand.
    For element in array:
        Assign current element to the bitmask of random number and 0x3FFFFFFF

{The below function prints elements of the array in the proper format}
Print_array(array, size of array, number of elements to print)
    For element in array:
        Print element
        If on multiple of 5 element:
            Print new line
        (Break if the entire array was printed or the number of elements to be printed was reached.)

{main function below}
main(number of arguments, array of strings)
    Make an empty set called “flags”.
    Set default seed number to 13371453
    Size of array = 100
    Number of elements to print = 100
    Parse through the command line arguments
        For each command line argument, add to the set.
    Make a dynamically sized array.
    Make a stats structure and set moves and compares to 0.
    If “a” in set then run all sorting tests
    If “e” in set then run heap test
    If “i” in set then insertion test.
    If “s” in set then run shell test.
    If “q” in set then run quick sort test.

If “r” in set then make seed number r.
If “n” in set then make size of array n.
If “p” in set then make elements equal to p.
If “h” in set then give instructions on how to use program.

* For all the sorting algorithms, I just followed the python code examples provided on the instructions document.

Insertion_sort -
{Move increments = yellow}, {Compares increments = green}
For i in range(1, length of array):
    J = i
    Assign Temp to A[i]
    While j > 0 and Temp < A[j - 1]:
        A[j] = A[j - 1]
        Minus J by 1
    Assign A[j] to Temp

Shell_sort -
{Move increments = yellow}, {Compares increments = green}
For object in objects(ln(3+2n)/ln(3), 0, -1):
    For i in range(3^(object) - 1, length of array):
        J = i
        Temp = A[i]
        While j >= object and temp < A[j - gap]:
            A[j] = A[j - gap]
            J = J - object
        A[j] = temp

Heap_sort -
{Move increments = yellow}, {Compares increments = green}
max_child(list, first, last):
    Left = 2 * first
    Right = left + 1
If right <= last and A[right - 1] > A[left - 1]:
    Return right
Else:
    Return left

fix _heap(list, first, last):
    Set found to 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

Build_heap(A, first, last)
    For object in range(last // 2, first - 1, -1):
        fix_heap(A, object, last)

heap_sort(A)
    First = 1
    Last = len(A)
    build_heap(A, first, last)
    For leaf in range(last, first, -1)
        Swap A[first - 1] and A[leaf - 1]
        fix_heap (A, first, leaf - 1)
```

Quick sort -

{Move increments = yellow}, {Compares increments = green}

```
Partition(list, lo, high):
    I = low - 1
    For j in range(low, high):
        If A[j - 1] < A[high - 1]:
            I += 1
            A[i - 1] , A[j - 1] = A[j - 1], A[i - 1]
        A[i], A[high - 1] = A[high - 1], A[i]
        Return i + 1

Quick_sorter(list, lo, high)
    If lo < high
        P = Partition(A, lo, high)
        Quick_sorter(A, lo, p-1)
        Quick_sorter(A, p+1, hi)

Quick_sort(list)
    quick_sorter(A, 1, len(A))
```