Liam Murray
Lijamurr
CSE13s
October 18, 2022

All Sorts Of C Code Design Document

General Idea

The overarching plan of this project is to implement several very common sorting algorithms into C code. The algorithms to be implemented are Bubble Sort, Shell Sort, Quick Sort, and Heap Sort. These algorithms will then be run through a test harness with random values created using the Mersenne Twister random number generator, in order to gauge their speed and efficacy. In general, the way the code will function will be to use arrays to categorize the data, and then use the specified sorting method to organize the data by size, in ascending order.

Bubble Sort

Bubble sort functions by iterating through all the elements in the array, and comparing them as pairs to see which element is larger. The larger element is then moved into the rightmost position, ensuring that with each pass the largest element "bubbles" to the top of the array.

Pseudocode:

Define function, accepting one parameter, "arr"

    Create a for loop with var i, iterating as many times as the array has indices

        Set a checker to 0, to test after each pass if any elements were swapped

        Create a for loop with var j, starting at the number of elements, ending at i, decrementing one with each pass

            If arr[j] < arr[j-1]

                Swap arr[j] and arr[j-1]

                Set checker to 1

            If checker == 0;

                Break

            Return the array

Shell Sort:

The idea of shell sort is to first pair the elements in an array with elements that are far away from each other. Then, compare the two elements, and swap them if the left is larger than the right. Then, lowering the gap between the elements, shell sort creates a new subarray of multiple elements, and sorts it. This continues until the gap is zero. The idea is to create partially sorted arrays that make a different sorting method, in this case insertion sort, much more efficient.

Pseudocode:

```
Gap calculation (generator in c)
Define gap_calc(n)
    If N == 1;
        Return 0
    Else if If N <= 2
        N = 1
    Else:
        N =(N * 5) // 11
    Return N;
Def Shell_Sort(array);
    Start = (length(array) *5) // 11
    Gap = start
    For Loop, until gap <=0, incrementing gap_calc(gap) each pass
        I = gap
        For loop, iterating until i == len(array) -1, incrementing 1 each pass
            J = I
            Temp = arr[i]
            While j >= gap and temp < arr[j-gap]
                Arr[j] = arr[j-gap]
                J = j-gap
            Arr[j] = temp
```

Quicksort:

The idea of quick sort is pick a pivot point at some point in the array, then sort the data into sections based on whether they are greater than or less than the pivot. Then, the function

performs this again on the smaller arrays, splitting them and sorting them. This happens until the elements are split down into arrays that are sufficiently small, which are then sorted by shell sort.
Pseudocode:

```
SMALL = 6 (arbitrary number of where we stop sorting)
Define Quicksort (pointer, length of array)
    If length of array = 0
        Stop
    Length = length of array
    array = pointer to where the array is
    If length < SMALL;
        shellsort(array, length)
        Return 0;
    Pivot = array[(length/2)]
Counter = length
While(counter > 0)
    Decrement counter
    Pivot address = length /2
    Lswap_addr = length/2
    L_change = 0
    Rswap_addr= length/2
    R_change = 0
    Swapped = false
    For L in range(array[0], Pivot address)
        If array[L] >= pivot
            L_swap = L
            L_change = 1
            break
    For R in range(length, pivot address, -1)
        If array[R] < pivot
            R_swap = R
            r_change= 1
```

```
                    Break
        If l_swap != r_swap
            Swap array[l] and array[r]
            Set swapped = true
            If l_change = 0 and r_change =1
                Pivot address = r-swap
            If l_change = 1 and r_change =0
                Pivot address = l-swap
        If swapped = false;
            break
    Quicksort((pointer to array[0]), pivot)
    Quicksort((pointer to pivot, pivot)
    Returns 0
```

Heapsort:

The idea of heapsort is to store data in a structure known as a “heap”, which follows basic rules that ensure that the data is an understandable pattern and is sorted. For a min heap, in which the smallest element is at the top, the rules are as follows; Every element must be smaller than its two children. The smallest of any two children should be in the leftmost position. To add elements to the array, add them on the bottom right. If the added element is smaller than its parent, swap the two. Continue until the min heap rules are obtained. To remove an element (in a sorted fashion), remove the top element, replacing it with the smaller of its two children. Continue this down the heap until the heap rules are obtained. Once every element has been removed, a sorted array will have been produced.

Pseudocode;

```
    //address of left child
    l_child(addr)
        Return (addr * 2) +1
    //address of right child
    r_child(addr)
```

```
        Return (addr * 2) +2
    //address of parent(addr)
    parent(addr)
        Return (n-1) // 2
    //function to move element up the heap
    //moving something up the heap
    Up_heap(addr, array)
        While addr > 0 and array[addr] < array[parent(addr)]
            Temp = array[addr]
            Array[addr] = array[parent(addr)]
            array[parent(addr)] = temp
            Addr = parent(addr)
    //moving an element down the heap
    Down_heap (array, heap_size):
        Addr = 0
        While l_child(addr) > heap_size;
            If  l_child(addr) == heap_size //if there's no left right child, the left is
smaller
                Smaller  = r_child(addr)
            Else:
                If l_child(addr) < r_child(addr)
                    Smaller = l_child(addr)
                Else:
                    Smaller = r_child(addr)
            If array[addr] < array[smaller]
                Break
            Temp = array[addr]
            array[addr] = array[smaller]
            array[smaller] = temp
            Addr = smaller
    //building the heap
```

```
Build_heap(array, length of array)
        For addr in range(0, length -1)
                copy[addr] = array[addr] //copy array to the heap
                up_heap(addr, heap)
        Return

//sort the aforementioned heap
sort_heap(array, length)
        Copy = Allocate space of size length
        For loop, “c” for as many elements in the array
                Copy[c] = array[c]
        Build_heap with copy and length
        For arr in range(0, length)
                array[arr] =  copy[0]
                copy[0] = copy[length - arr - 1]
                down_heap(copy, length - arr)
        Free the memory
```

Test Harness Code

The test harness is designed to test and catalog the performance of the sorting algorithms, as well as to display the efficacy of all the algorithms when pitted against one another.

Specify the possible command line inputs: “asbqhr:n:p:H”

Define main function, with arguments (int argc, char **argv)

```
//Set default values
A_arg = 0
S_arg = 0
B_arg = 0
Q_arg = 0
h_arg = 0
R_arg = 13371453
N_arg = 100
P_arg = 100
```

```
H_arg = 0
Set = empty_set() //Make a set using empty_set()
//parse arguments
For loop, with arg = 0, until arg = (argc-1), incrementing 1
        insert_set(argv[arg], set) //put argument from argv into the set
Another loop, looping until counter  = argc
        If get opt = a;
                A_arg = 1
                delete_set(a, set)
        If get opt = s;
                s_arg = 1
                delete_set(s, set)
        If get opt = b;
                b_arg = 1
                delete_set(b, set)
        If get opt = q;
                q_arg = 1
                delete_set(q, set)
        If get opt = h;
                h_arg = 1
                delete_set(h, set)
        If get opt = r;
                r_arg = strtoul(optarg,NULL,10)
                delete_set(r, set)
        If get opt = n;
                n_arg = atoi(optarg)
                delete_set(n, set)
        If get opt = p;
                p_arg = atoi(optarg)
                delete_set(p, set)
        If get opt = h;
```

```
                    H_arg = 1
                    delete_set(H, set)
        //execution of code
        mtrand_seed(s_arg)
        allocate_space(n_arg)
        Set pointer ptr to allocated space
        For loop, counter = 0, until counter = n_arg-1
            *(ptr + counter) = mtrand_rand64()
        If a_arg = 1:
            print(bubble(prt, n_arg))
            print(shell(prt, n_arg)))
            print(quick_sort(prt, n_arg)))
            print(heap_sort(prt, n_arg)))
        If b_arg = 1:
            print(bubble(prt, n_arg))
        If s_arg = 1:
            print(shell(prt, n_arg)))
        If q_arg = 1:
            print(quick_sort(prt, n_arg)))
        If h_arg = 1:
            print(heap_sort(prt, n_arg)))
Return 0;
```