Sean Brandon
spbrando@ucsc.edu
11/11/2020

CSE13s Fall 2020
Assignment 5: Sorting
Design Document

Pre-lab Questions:

Part 1:

1. It takes exactly 10 rounds of swaps.
2. The worst case scenario for bubble sort takes exactly n^2 comparisons to finish, n being the number of elements in the array.

Part 2:

1. The worst case time complexity for Shell sort depends on the size of the gap because too few gaps slows down the speed at which the values are swapped but too many gaps produces an overhead. There are a number of other gap sequences that mathematicians have produced, I would try using one of those to improve the time complexity of this sort. I found information about other gap sequences here: https://en.wikipedia.org/wiki/Shellsort#Gap_sequences
2. If I knew the size of the unsorted array before hand I would use a predetermined gap sequence like the one published by Sedgewick in 1986 (found in the reference above). This would not be changing the gap size with every time it is run and would simply eliminate the need to compute a new gap size with each iteration.

Part 3:

1. Quick sort is not doomed by its worst case scenario because it is often much much faster than its worst case, in fact its average case is O(nlogn). Also, its worst case can be avoided most of the time by choosing a good pivot. I learned this in lecture but also used this reference: https://www.geeksforgeeks.org/quicksort-better-mergesort/

Part 4:

1. The binary search algorithm cuts down on time complexity when it is combined with the insertion sort algorithm because it is able to find the right spot for a given value much faster than the regular insertion sort because binary insertion sort halves the search space each time there is a comparison done.

Part 5:

1. I plan on keeping track of the number of moves and comparisons of each sort by having the functions themselves return an array value of 2 ints; the first int being the number of comparisons and the second int being the number of moves.

Introduction:

This document describes how to implement various sorting functions and how they work. In this lab the sorting functions implemented are bubble sort, shell sort, quick sort, and binary

insertion sort. I also have implemented a testing harness so that I would be able to efficiently investigate how each of these sorting algorithms compare to each other. More specifically, I gather statistics like number of moves and number of comparisons required to sort an array of a certain size. All of which has been written in C. The sorting functions sort an array of integers from lowest to highest value. The testing harness I made creates a random array of numbers up to 2^{30}. The user of this program has the following optional arguments:

-A This will print out the results of all sorting methods. This is the default argument if no other sorting methods are specified.
-b This will print out the results of bubble sort.
-s This will print out the results of shell sort.
-q This will print out the results of quick sort.
-i This will print out the results of binary insertion sort.
-p This, followed by an integer value, specifies the first number of elements to be printed from the sorted array. Defaults to 100.
-r This, followed by an integer value, specifies the seed for the random number generator that creates the unsorted array. Defaults to 822202.
-n This, followed by an integer value, specifies the length of the array to be sorted. Defaults to 100.

Top Level:

Explanations of the code are written in red
Citations are marked with yellow highlighter

sorting.c pseudo-code

Includes
Prototype print_sorted
Prototype reset

```
main(argc, **argv) {
        Bool All, bubble, shell, quick, insertion = false
        Int print_len, length = 100
        Int seed = 8222022
        Average getopt loop to take in user inputs, nothing special.
        Getopt loop {
                switch
                        Sets bools to true depending on getopt arguments
                        Sets ints depending on optarg arguments
                        Prints error statement and ends program if invalid arguments
        }

        Defaults to all sorts being printed
        If no sorts are specified
                then set All boolean to true
```

Making sure we don't print numbers that aren't in our array
If number of elements is greater than number of elements in array
    Print error message
    Return

I have two arrays so that I can sort one and then reset it to its original or "base" unsorted state if more than one sort is used back to back
Calloc workable_array
Calloc base_array

Fill arrays with random numbers
srand(seed)
for i in range(length) {
    Mask to limit values to 2^{30} as specified in the assignment
    Num = random number masked with 0x3FFFFFFF
    Set both arrays to the same values so workable can be reset
    Workable[i], base[i] = num
}

Sorts array with each sort and prints outputs for each
if (All) {
    Bubble sort
    Sort array using bubble_sort
    Print sorted array and results returned by function with print_sorted()

    Shellsort
    Reset workable array with reset()
    Sort array using shell_sort
    Print sorted array and results returned by function with print_sorted()

    Quick sort
    Reset workable array with reset()
    Sort array using quick_sort()
    Print sorted array and results returned by function with print_sorted()

    Binary insertion sort
    Reset workable array with reset()
    Sort array using binary_insertion_sort()
    Print sorted array and results returned by function with print_sorted()

    Free workable array and base array
    return 0;   End function here to avoid repeating outputs
}

```
        Sorts array with bubble and prints results
        If (bubble) {
                Sort array using bubble_sort
                Print sorted array and results returned by function with print_sorted()
        }
        Sorts array with shell and prints results
        If (shell) {
                Reset workable array with reset()
                Sort array using shell_sort
                Print sorted array and results returned by function with print_sorted()
        }
        Sorts array with quick and prints results
        If (quick) {
                Reset workable array with reset()
                Sort array using quick_sort
                Print sorted array and results returned by function with print_sorted()
        }
        Sorts array with binary_insertion and prints results
        If (binary) {
                Reset workable array with reset()
                Sort array using binary_insertion_sort
                Print sorted array and results returned by function with print_sorted()
        }

        Free workable array and base array
        Return 0;
}

Prints statistics gathered and the desired amount of lowest values of the sorted array
void print_sorted(arr[], array_length, compares, moves, print_length) {
        print( elements, moves, compares)
        For i in range(length) {
                print(array[i])
        }
        print(newline)
        return
}

Resets sorted array elements back to their original unsorted (AKA base) positions
void reset(workable[], base[], length) {
        For i in range(length) {
                Workable[i] = base[i]
        }
        return
```

}

bubble.c pseudo-code

Pseudocode given in the lab document but I made some modifications

Static int info[] = {0, 0} to keep track of swaps and compares

```
Def Bubble_Sort(arr):
    For i in range(len(arr) - 1):      Loops through length number of times
        j = len(arr) - 1               j is the end of the array
        While j > i:                   Loop through unsorted array
            If arr[j] < arr[j - 1]:    If the element were looking at is smaller than the
                                       element before it...
                arr[j], arr[j - 1] = arr[j -1], arr[j]        Swap them
                Info[1] += 3           Increment move count by three because a temp
                                       variable is necessary
            j -= 1                     j is decremented after each loop because at the end of
                                       each loop we know the right side is sorted
            info[0]++                  increment compare count

    return info      end function and return move and compare count in an array
```

The bubble sorting algorithm compares neighboring elements and if they are out of order it swaps them. It does this for the length of the array which moves the largest value to the far right. This far right becomes a sorted subsection of the entire array, and the values to the left of it remain unsorted. So the loop repeats through the unsorted subarray, moving the greatest value of it to the far right and into the sorted array by comparing each neighboring array and swapping when necessary. This grows the sorted subarray and shrinks the unsorted subarray until the entire array is sorted. The time complexity for bubble sort has a worst case and average case of $O(n^2)$. This is because the function would have to loop through the array comparing each element with its neighboring element until it is completely sorted. This means as the unsorted array increases in size, the amount of time it will take to sort increases exponentially which is definitely not ideal for large sets of data. The best case for bubble sort is $O(n)$.

Shell.c pseudo-code

Pseudocode given in the lab document but I made some modifications

Static int n Keeps track of the value that was yielded in original pseudo-code

Static int info[] = {0, 0} to keep track of swaps and compares

Returns the size of the gap to be used in shell_sort. Gets smaller and smaller

```
int gap(num) {
    n = num
    while n > 1 {
```

```
            If (n <= 2) {     If n is 2 or less, n = 1 and the gap function will be done being
                              called at that point
                n = 1
            }
            Else {            N becomes 5/11 times smaller (to the floor int)
                n = 5 * n // 11
            }
            return n
        }
        return n
}

Def shell_sort(arr[], array_length) {
        n = length        n starts as length of array but will be modified to smaller and smaller
                          values each time gap is called until it reaches 1
        do {
            step = gap(n)             Set step to gap size
            Loop for the distance between step and the end of the array (which, in the case
            of multiple steps, is also the distance between each step)
            For i in range(step, length) {
                Loop j from i to step - 1, decrementing by step
                For j in range(i, step - 1, -step) {
                    If the values j in each step section, separated by a step, are out of
                    order...
                    If arr[j] < arr [j - step] {
                        arr[j], arr[j - step]  = arr[j - step], arr[j]          Swap them
                        Info[1] += 3              Increment move count by 3
                    }
                    Info[0]++         increment compare count by 1
                }
            }
        } while (step > 1)        do until step is 1 then return move and compare counts in an array
        return info               end function and return move and compare count in an array
}
```

The shell sorting algorithm is similar to insertion sort but this algorithm sorts values that are a gap distance away from each other as it loops through the unsorted array. It reduces this gap distance with each loop through the array until the gap distance is comparing neighboring elements. The time complexity of shell sort is, for best case, O(n). For other cases though, it depends on the gap size, though its worst case is less than $O(n^2)$.

quick.c pseudo-code

Pseudocode given in the lab document but I made some modifications

Static int info[] = {0, 0} to keep track of swaps and compares

```
def partition(arr[], left, right) {
        pivot = arr[left]           Item to be pivoted around
        Lo = left + 1               Far left side of array that's being worked with
        Hi = right                  Far right side of array that's being worked with
        Loop indefinitely
        While true {
                While the low and high indexes haven't passed by each other AND while the high
                element is larger than or equal to the pivot element
                While (lo <= hi && arr[hi] >= pivot) {
                        hi -= 1             Move the high index down
                        Info[0] += 1        Increment compare count
                }
                While the low and high elements haven't passed by each other AND while the
                high element is less than or equal to the pivot element
                while (lo <= hi && arr[lo] <= pivot) {
                        Lo += 1             Move the low index up
                        Info[0] += 1        Increment compare count
                }

                If the current low element is less than the current high element then swap
                If (lo <= hi) {
                        arr[lo], arr[hi] = arr[hi], arr[lo]
                        Info[0]++           Increment compares by 1
                        Info[1] += 3        Increment moves by 3
                }
                Otherwise, the partition has been sorted except for the pivot so break loop and…
                Else {
                        break
                }
        }
        Move the pivot into the correct place
        Arr[left], arr[hi] = arr[hi], arr[left]
        Return high index as the partition for the next recursive loop
        Return hi
}

Quick sort function (recursive)
def quick_sort(arr[], left, right) {
        If the array is not yet fully sorted
        AKA if the subarrays on either side of the partition is not yet fully sorted
        AKA if the partition is more than one element
        If (left < right) {
                Info[0]++           increment compare count
```

```
            Index = partition(arr, left, right)
            Left of partition
            quick_sort(arr, left, index - 1)
            Right of partition
            quick_sort(arr, index + 1, right)
        }
        Return info     end function and return move and compare count in an array
}
```

The quick sorting algorithm is recursive and much of the work is done in a helper function called partition. This algorithm divides the unsorted array into two subarrays by the partition value and moves any value greater than the pivot value to the right subarray and any value less than the pivot to the left subarray and finally the pivot value is put into the correct position. These subarrays then have quick sort called on them, which is what makes it recursive, until the whole array is sorted. Quick sorts time complexity is as follows: Worst = $O(n^2)$, Best and average = $O(nlogn)$. While its worst case of $O(n^2)$ does seem troublesome, this can mostly be avoided by choosing a partition correctly.

binary.c pseudo-code

Pseudocode given in the lab document but I made some modifications

```
Static int info[2] = {0, 0}

def binary_insertion_sort(arr[], length) {
        Loop through each element in the array starting after the first. Through each loop the
        sorted subarray grows and the unsorted subarray shrinks
        For i in range(1, length) {
            Value = arr[i]          Element to be compared
            Left = 0
            Right = i
            Binary search
            While the left index hasn't passed right index
            While (left < right) {
                Find middle of left and right
                Mid = left + ((right - left) / 2)
                If value to be compared is greater than or equal to middle value…
                if (value >= arr[mid]) {
                    Move left index past middle index
                    Left = mid + 1
                }
                Else (the value to be compared is less than the middle value)
                Else {
                    Move right index to middle index
                    right = mid
                }
```

```
                Increment compare count
                info[0]++
            }
            While the amount of elements compared is greater than the amount of values
            greater than the middle index, decrementing the amount of elements compared
            we...
            For j in range(i, left, -1) {
                Swap values
                arr[j -1], arr[j] = arr[j], arr[j - 1]
                Info[1] += 3            increment move count by 3
            }
        }
        Return info     end function and return move and compare count in an array
}
```

The binary insertion sort is, as the name implies, similar to insertion sort. The difference being that binary insertion sort finds the correct place to insert the value by doing a binary search rather than a possibly large number of comparisons and swaps as we would have to in regular insertion sort. Binary insertion sort's time complexity is as follows: best case = O(nlogn), worst and average case $O(n^2)$.

Design Process:

The design process for this assignment was not very complex because the design for each of the sorting algorithms was already provided for us at the beginning. However I did have a couple of revisions I had to do with my testing harness implementation and with shell sort

- Originally I was going to keep track of the statistics (moves and compares) by dynamically allocating memory with a malloc inside my sort function then return the address of said memory and free that memory in my main file after I had finished with it but I realized that this was overly complicated and used malloc for no reason. In the end I returned a static array containing my moves and compares.
- Originally I used only for loops for my shell_sort function but this seemed overly confusing and it seemed to me to make the code much simpler to read if I switched the first for loop with a do while loop instead.

All in all this lab was a bit of a challenge to implement someone else's pseudo-code but it forced me to learn in depth about how each sorting algorithm works. In particular, implementing shell sort was the biggest challenge for me, (not because the gap function though despite the yield in that not being present in C) it was the first for loop that really had me confused but eventually I figured it out and even reworked it to an implementation that looks cleaner in my opinion. If I could change something about this lab I would probably just change the pseudocode of shell sort to be more understandable.