Aleesha Nageer
anageer@ucsc.edu
11/15/2020

CSE 13S Fall 2020

Assignment 5: Sorting

Design Document

Sorting is a common tool used in programming as it compares various numbers to place them in a particular order. In this assignment we are required to collect a random array of numbers using a seed and sort these numbers in ascending order using different sorting algorithms.

- This lab parses arguments to run different sorting methods as well as provide a seed, array size, and printing range:
  - -A: runs all sorting algorithms
  - -b: runs the Bubble Sort algorithm
  - -s: runs the Shell Sort algorithm
  - -q: runs the Quick Sort algorithm
  - -i: runs the Binary Insertion Sort algorithm
  - -p #: sets the range of sorted numbers to print
  - -r #: sets the seed number
  - -n #: sets the array size

Pre-Lab

Part 1

1. It will take 10 rounds of swapping to sort these numbers in ascending order using Bubble Sort.
2. The number of comparisons that we can expect in the worst-case scenario is n(n-1)/2, n being the total number of items being sorted. This is when the numbers are in the opposite order of which it intends to be. To clarify, this is when the numbers are initially in descending order and the goal is to put them in ascending order.

Part 2

1. The worst time complexity for Shell Sort is when the initial gap is larger rather than smaller. This is due to the fact that there will be more iterations required to decrease the size of the gap to ensure each value is properly sorted. Essentially, this algorithm is checking two items in an array, which are separated by the difference of the gap. For example, if the gap size was 5, then the function will loop through each position in the array, starting at zero, and check whether the value at index zero is greater than the value at index four. This loop continues until the gap size is one as the current index compares the values adjacent to it. Therefore, starting with a smaller gap size will allow the loop to run less times and give a better time complexity. Most examples I found online as well as our book suggested to set the gap equal to the length of the array divided by two and continually divide this gap by two until its size equals one. In the provided pseudocode, we multiply our array size by five and divide that number by 11, making the gaps smaller in most cases.

Sources:
https://www.youtube.com/watch?v=408TQi6MWmI
https://www.codingeek.com/algorithms/shell-sort-algorithm-explanation-implementation-and-complexity/#:~:text=Shell%20Sort%20is%20a%20comparison,n*%20log2n).&text=There%20are%20various%20increment%20sequences,and%20O(n2).

2. I would improve the runtime of this sort, without changing the gap size, by presorting the list. The fasted time complexity for Shell Sort is O(nlogn), which is when the list is already sorted. Therefore, if you presort the list, the runtime will improve.

Part 3

1. Although Quick Sort's worst time complexity is O(n^2), this sorting method isn't doomed by its worst-case scenario because the pivot can be chosen so that the worst case can be avoided. This worst-case is when the partition function picks either the greatest or smallest elements for the pivot, therefore, being able to choose the pivot, gives the programmer flexibility to avoid a slower time complexity.

   Sources
   https://www.geeksforgeeks.org/quick-sort/#:~:text=Analysis%20of%20QuickSort&text=k%20is%20the%20number%20of,or%20smallest%20element%20as%20pivot.

Part 4

1. By incorporating the binary search algorithm in the insertion sort algorithm, the best-case time complexity improves from O(n) to O(nlogn). This is because binary insertion sort reduces the number of comparisons being made and finds the proper location to place the value within the array. It can cut the comparisons in half since binary insertion sort since it only works if the beginning portion of the array is already sorted.

   Sources
   https://en.wikipedia.org/wiki/Insertion_sort
   https://www.geeksforgeeks.org/binary-insertion-sort/
   https://stackoverflow.com/questions/18022192/insertion-sort-with-binary-search
   https://book.huihoo.com/data-structures-and-algorithms-with-object-oriented-design-patterns-in-c++/html/page491.html

Part 5

1. In order to keep track of the number of moves and comparisons for each sort, I plan on building multiple arrays for each sorting algorithm. These arrays will be formed from a struct that contains values like the array size, the seed, and the number of moves/comparisons. By doing this, I will be able to differentiate the different number of moves and comparisons for each sorting algorithm.

TOP-LEVEL

Main function

```
Initialize Flags for Getopt
Initialize Seed, Print Range, Seed, & Array Size

Run Getopt & Parse Arguments →"Absqip:r:n:"
    Switch
    Case(A)
        Set Flag for Every Sorting Algorithm to 1
    Case(b)
        Set Flag for Bubble Sort Algorithm to 1
    Case(s)
        Set Flag for Shell Sort Algorithm to 1
    Case(q)
        Set Flag for Quick Sort Algorithm to 1
    Case(i)
        Set Flag for Binary Insertion Sort Algorithm to 1
    Case(p)
        Print range = (number user entered)
    Case(r)
        Seed = (number user entered)
    Case(n)
        Array size = (number user entered)

Build Structs with elements needed:
    Seed, Array Size, Array Contents, Moves, Comparisons

If flag = 1, run corresponding functions
```

Initialize Vector (Masking to remain 30-bits)

```
    Get random seed
    For i = 0; less than size of the array; increment i
        Vector = random number and-ed with 0x3FFFFFFF
```

Printing Statistics

```
    Print array size, moves, & comparisons
```

Printing Array Values

```
For i = 0; less than print range; increment i
    If i = 0 or if remainder of i/7 doesn't equal 0
        Print a tab and the array value at i
    Else
        Print a newline, a tab, and the array value at i
Print newline once out of for loop
```

DESIGN PROCESS

Throughout this lab, I changed my pseudocode and overall design of my sorting functions:

- Building my struct took some time, but I eventually did not need to change anything. I just added more variables, such as the moves and comparisons variables.
- For each of my sorting algorithms, I translated the Python pseudo code provided almost exactly, which ended up not working out. I had to eventually change my array sizes and implement more functions within some of my sorting files to account for passing in my struct.
- After this I had to adjust my printing algorithm several times as I couldn't properly get the sorted arrays to print with 7 items per row. I included a formatting label to fix my issues as well as a condition statement to account for when the next row of numbers should begin.

Overall, this design was challenging but I learned more about passing struct elements into different functions and how to convert Python code into C!