Roleen Ferrer

Professor Maxwell Dunne

CSE 13S - Computer Systems and C Programming

Assignment 5 - Sorting

In this lab assignment, I will be implementing sorting algorithms in order to sort an array of integers from 0 to 2^30 - 1. The sorting algorithm that I will implement are:

- Bubble Sort
- Shell Sort
- Quick Sort
- Binary Insertion Sort

I will also be collecting data from each sorting algorithm, such as the array size, the number of moves required, and the number of comparisons required.

In order to iterate through each sorting algorithm, I will be using getopt() to collect commands from the user. These commands are as follows:

- **-A**, which will run **all** sorting algorithms
- **-b**, which will run **Bubble Sort**
- **-s**, which will run **Shell Sort**
- **-q**, which will run **Quick Sort**
- **-i**, which will run **Binary Insertion Sort**
- **-p n**, which will print out the first **n** elements of the array
- **-r s**, which will set the random seed to **s**
- **-n c**, which will set the array t to some **c**

Pre-Lab Questions

Part I

1. I believe that the number of swaps to sort 8, 22, 7, 9, 31, 5, 13 in ascending order using bubble sort is 10 swaps.
2. We should expect to see at least 21 comparisons for the worst case when using Bubble Sort for this list of numbers.

Part II

1. The worst time complexity for Shell sort depends on the gap size, as if the gap size was much wider, there would have to be more comparison between each number. I would improve the runtime of this sort by decreasing each step in each gap, thus using a different algorithm to find different and more efficient steps. (https://www.programiz.com/dsa/shell-sort)
2. I would improve the runtime of this sort by decreasing the amount of comparisons in the algorithms that checks into each step.

Part III

1. The only way for the worst case time complexity to occur for Quick sort is when the pivot is either the "extreme", the smallest or largest element. When this happens, it generally means that the pivot selected when it is at its extreme will generate the worst case. Though this is extremely unlikely when using this sorting algorithm, as the chances of selecting the pivot as an extreme is very slim, but can still happen. (https://www.geeksforgeeks.org/can-quicksort-implemented-onlogn-worst-case-time-complexity/)

Part IV

1. Implementing sorting algorithms that incorporate both the binary search algorithm and the insertion sort algorithm may yield a better time complexity, as there would be less comparisons when parsing through each algorithm. (https://en.wikipedia.org/wiki/Insertion_sort)

Implementation

Main (sorting.c)

This source file is tasked with accepting tasks from the user getting getopt(). These commands will be translated into their respective functions and print each sorting algorithm accordingly.

Functions

1. int main(int argc, char **argv)
    a. This main function takes in the arguments from the user (see above for list of commands)
    b. Set the default values to some variables
        i. n = 100
        ii. p = 100

        iii. seed = 822022

    c. Create boolean variables that will represent when a certain sorting algorithm is called upon
    d. Run a while loop that will collect commands from the user
        i. while((opt = getopt(argc, argv, “Absqip:r:n:”)) != -1)
    e. For each command that is *true*, create an array to hold each values of random numbers
        i. uint32_t *array = calloc(n, sizeof(uint32_t))
    f. Fill the array with random numbers using rand() and a for loop
        i. Mask rand with a bit mask to restrict domain to 2^30 - 1
            1. rand() & ~(0xC0000000)
2. void printArray(uint32_t *array, int n, int p)
    a. This function will print an array. This is primarily used after each array is sorted from their respective sorting algorithm.
    b. Parameters:
        i. *array - an array of integers
        ii. n - the amount of integers in the array
        iii. p - the number of integers to print
    c. Declare a column variable that will keep track of the format for 7 columns of numbers
    d. Create a for loop that will print indexes of the array up to a certain p
        i. for (int i = 0; i <p; i++)
        ii. Make the print of integers right justified by 13
            1. printf(“%13d”, array[i])
        iii. Create a new line if columns == 7

(Bubble Sort) bubble.c

This source file contains the implementation of the bubble sorting algorithm. The function bubbleSort() will be called to sort an array using the bubble sort algorithm. The algorithm works by checking if the adjacent pairs of elements. The second element is smaller than the first, it would then be swapped, and eventually, the largest element will be at the bottom of the array.

Since it is at the bottom of the array, it is not considered after the next pass of the sorting algorithm. This will run until there are no pairs that are out of order.

Functions

1. void bubbleSort(uint32_t *array, int n)
    a. This function sorts the array using bubble sort
    b. Parameters:
        i. *array - an array of integers
        ii. n - the amount of integers in the array
    c. The following Pseudocode was provided in the lab document
    d. Create variables to keep track of the previous iteration of the array
        i. int temp
    e. Create variables to keep track of the amount of moves and comparisons
        i. int moves = 0, comparisons = 0
    f. Create a for loop that iterates through i through the range of the length of the array - 1
        i. for i in range(len(arr) - 1)
    g. Set j equal to the length of the array - 1
    h. Create a while loop that will iterate in the for loop such that j > i
        i. Add 1 to comparisons to simulate 1 comparison
        ii. In the while loop, if array[j] < array[j - 1], set array[j] to array[j - 1] and array[j - 1] to array[j]
            1. Use the temp variable to store the array[j] before moving.
            2. Add 3 to the moves variable to simulate 3 moves
        iii. Subtract j by 1
    i. After the for loop, print out the name of the algorithm and the amount of elements of integers in the array. Also print out the number of moves and the number of comparisons

(Shell Sort) shell.c

This source file contains the implementation of the shell sorting algorithm. The function shellSort() will be called to sort an array using the shell sort algorithm. The algorithm works by taking an interval between compared elements and will be continuously reduced throughout the sort.

Functions

1. int gap(int n)
    a. This function will calculate the intervales (gaps) that will be used in the Shell sorting algorithm
    b. Parameters:
        i. n - the amount of integers, which will change when gap() is called
    c. Create a conditional to check if n <= 2
        i. if it is, set n equal to 1
        ii. else, set n = 5 * n / 11
    d. Return n
2. void shellSort(uint32_t *array, int n)
    a. This function will sort the array using Shell sort
    b. Parameters:
        i. *array - an array of integers
        ii. n - the amount of integers in the array
    c. The following Pseudocode was provided in the lab document
    d. Create variables that will store the step (n) and temp to hold the a value for sorting
    e. Create a while loop that will end when step > 1
    f. Change the step variable to gap(step)
    g. Create a for loop that will iterate through step (i), such that step is less than n, incremented by 1.
    h. Create another for loop such that j is equal to i and will run until j > step - 1, decremented by step
        i. Add 1 to comparison
    i. In the second for loop, create a condition that will check if array[j] < array[j - step]
        i. Set temp to array[j]
        ii. Set array[j] to array[j - step]
        iii. Set[j - step] equal to temp

        iv. Add 3 to moves

(Quick Sort) quick.c

This source file contains the implementation of the quick sorting algorithm. The function quickSort() will be called to sort an array using the quick sort algorithm. The algorithm works by creating two subarrays by selecting an element from the array and setting it as the pivot. Elements that are less than the pivot will go to the left sub-array and larger elements will go to the right-sub array. The Quick Sort algorithm will then be run recursively through partition.

Functions

1. int partition(uint32_t *array, uint32_t left, uint32_32 right)
    a. This function will partition the array into 2 different sub-array, to which certain integers are placed in each sub-array
    b. Parameters:
        i. *array - an array of integers
        ii. left - the leftmost index of the array (0)
        iii. right - the rightmost index of the array (n-1)
    c. **The following Pseudocode was provided in the lab document**
    d. Create variables:
        i. pivot = array[left]
        ii. lo = left + 1
        iii. hi = right
    e. Create a variable temp that will store a move
    f. Create a while loop that will run until it reaches a break statement
        i. while(true)
    g. Check if lo <= hi
        i. If it is, add 1 to comparisons
        ii. Check if array[hi] >= pivot
            1. If it is, create a while loop that runs for lo <= hi && array[hi] >= pivot
                a. Subtract hi by 1
                b. Check again if lo <= hi
                    i. Add 1 to comparisons if it is

```
        h. Again, check if lo <= hi
            i. If it is, add 1 to comparisons
            ii. Check if array[lo] <= pivot
                1. If it is, create a while loop that runs for lo <= hi && array[hi] <=
                   pivot
                    a. Subtract lo by 1
                    b. Check again if lo <= hi
                        i. Add 1 to comparisons if it is
        i. Check if (lo <= hi)
            i. if it is, set temp to array[lo]
            ii. Set array[lo] to array[hi]
            iii. Set array[hi] to temp
            iv. Add 3 to moves
            v. else, break the while loop
        j. At the end of the while loop:
            i. Set temp to array[left]
            ii. Set array[left] to array[hi]
            iii. Set array[hi] to temp
            iv. Add 3 moves
        k. Return hi
    2. void quickSort(uint32_t *array, int left, int right)
        a. This function will sort the array using Quick sort. It will be called recursively
           until the array is sorted.
        b. Parameters:
            i. *array - an array of integers
            ii. left - the leftmost index of the array
            iii. right - the rightmost index of the array
        c. Create an index variable that will keep track of the index
        d. Create a conditional that will check if left < right
            i. if it is, set index to partition(array, left, right)
            ii. Call quickSort(array, left, index - 1)
            iii.  Call quickSort(array, index + 1, right)
```

(Binary Insertion Sort) binary.c

This source file contains the implementation of the binary insertion sorting algorithm. The function binarySort() will be called to sort an array using the binary insertion sort algorithm. The algorithm works by finding the correct position of the array for the element. While doing so, the search and sorting is done though half of the array using a midpoint. This reduces the number of comparisons between each of the array elements that would ordinarily be done.

Functions

```
1. void binarySort(uint32_t *array, int n)
    a. This function will sort the array using Binary Insertion sort.
    b. Parameters:
        i.   *array - an array of integers
        ii.  n - the amount of integers in the array
    c. Create variables value, left , right, mid, and temp
    d. Create a for loop that is in the range from [1, n)
        i.   for (int i =1; i < n, i++)
    e. Set value to array[i]
    f. Set left to 0
    g. Set right to i
    h. Create a while loop that will run for left < right
        i.   Set mid to left + ((right - left) / 2)
        ii.  Create a conditional that will check if value >= array[mid]
            1. if it is, set left to mid + 1
            2. Add 1 to comparisons
        iii. else:
            1. Set right to mid
            2. Add 1 to comparisons
    i. Create a for loop that will run from [i, left)
        i.   for (uint32_t j = i; j > left; j--)
            1. Set temp to array[j - 1]
            2. Set array[j - 1] to array[j]
```

3. Set array[j] to temp
4. Add 3 to moves