Rian Delarosa
rjdelaro
11/12/20

Assignment 5 Design

The C file sorting.c includes functions declared in bubble.h, shell.h, quick.h and binary.h and functions written in bubble.c, shell.c, quick.c and binary.c. The main function uses getopt() to receive user commands from the console and can print (-p #) number of elements of a sorted array with (-n #) of elements, and random values set by (-r #) random seed. The array can be sorted in bubble (-b), shell (-s), quick (-q), or binary insertion (-i) methods. All these sorting methods can be printed at once as well (-A).

Pre-Lab Questions

1. How many rounds of swapping do you think you will need to sort the numbers 8, 22, 7, 9, 31, 5, 13 in ascending order using Bubble Sort?

I believe that to sort the list of those specific numbers, 10 rounds of swaps would be needed in order to sort them in ascending order using the Bubble Sort method.

2. How many comparisons can we expect to see in the worse case scenario for Bubble Sort? Hint: make a list of numbers and attempt to sort them using Bubble Sort.

For each element in the array, bubble sort does n - 1 comparisons. In big O notation, bubble sort performs O(n) comparisons. So if we were sorting 8 elements the worst case scenario would be 56 because there would be 8 - 1 comparisons for every element (7 * 8 = 56).

1. The worst time complexity for Shell sort depends on the size of the gap. Investigate why this is the case. How can you improve the time complexity of this sort by changing the gap size? Cite any sources you used.

The worst time complexity is based on the size of gap because the entire outer function runs for the amount of gap value, so the smaller the gap size the fewer gaps and less time spent sorting. Having fewer gaps will improve the time complexity by decreasing the number of compares and exchanges needed in the inner loops of the function.

2. How would you improve the runtime of this sort without changing the gap size?

You can tremendously improve the runtime of Shell sorting by having some elements in the array already sorted, meaning fewer iterations of the inner loops would need to be used.

1. Quicksort, with a worse case time complexity of $O(n^2)$, doesn't seem to live up to its name. Investigate and explain why Quicksort isn't doomed by its worst case scenario. Make sure to cite any sources you use.

Quicksort is not doomed by its worst case scenario because the pivot value can be altered, meaning the largest or smallest element are not always chosen. This increases the likelihood of the best case scenario and decreases the chances for the worst case.

1. Can you figure out what effect the binary search algorithm has on the complexity when it is combined with the insertion sort algorithm?

Since the binary search algorithm locates and compares two elements closest to the middle of the array, the complexity increases while the time complexity decreases. Half of the array is sectioned to be searched based on if the middle element closer to the left is greater than the element closer to the right. This process will continue until there is one element remaining. This becomes important in relation to the insertion sort algorithm because it will swap elements if the left is larger than the right element. In combination, these algorithms will improve the time complexity generously since the arrays begin in the center and work their way outward sorting through insertions until there is a single element.

1. Explain how you plan on keeping track of the number of moves and comparisons since each sort will reside within its own file.

I plan to keep track of the number of moves and comparisons in each respective C file by using counting variables when such an action occurs and having each function return an array with the number of moves stored at index 0 and the number of comparisons at index 1.

Similar to the previous assignments I used the same boolean flag structure in order to keep track of the options entered by the user into the terminal while running the sorting executable. The variable c is used for the getopt() functions, there are several default values initialized such as print, seed, size, moves, comparisons, and even the array that will be sorted.

```
int main(args)
  c = 0
  uint32_t print = 100
  uint32_t seed = 8222022
  uint32_t size = 100
  uint32_t moves = 0
  uint32_t comparisons = 0
  uint32_t* array
  bool op_A, op_b, op_s, op_q, op_i, op_p, op_r, op_n
  op_A = op_b = op_s = op_q = op_i = op_p = op_r = op_n = false
  while ((c = getopt(args) != -1)
    switch (c)
      case 'A':
        op_A = true
        break
      case 'b':
        op_b = true
        break
      case 's':
        op_s = true
```

```
        break
      case 'q':
        op_q = true
        break
      case 'i':
        op_i = true
        break
      case 'p':
        op_p = true
        break
      case 'r':
        op_r = true
        break
      case 'n':
        op_n = true
        break
```

Because there are multiple options with numeric inputs expected afterward I used a similar design for making sure options n, p, and r would run correctly. If the option was entered I would locate the index in which the respective option was located and change the specific value to the value at the index plus one. For this example, seed was set to the value at index + 1, but for option "-n" size would be changed and for option "-p" print would be changed. In order to ensure the random number at the specific seed would be less than 2^{30} - 1, I used the bitwise operation "&" and the hex value for the maximum value to mask the random value. If the specific option was not entered by the user the seed, size, or print, would be set to their respective default value.

```
if (op_r)
    for (int i = 0, i < argc, i++)
      if (argv[i] == "-r")
        seed = argv[i + 1]
        srand (seed)
        for (i = 0, i < size, i++)
          array[i] = rand() & 0x3FFFFFFF
else
    srand (seed)
    for (i = 0, i < size, i++)
      array[i] = rand() & 0x3FFFFFFF
```

I used similar function access and print structure for all sorting methods and copied them into one option for option “-A”. I begin each option by making sure “-A” was not also inputted because we don’t want the same tests to print multiple times. I begin each structure by using a for loop to set all elements in the array to 0 to clear from any previous sorting functions. I use a second for loop to set the array needed to be sorted to the original random values generated form option “-r”. Because each of my functions return an array with the corresponding number of moves and comparisons from each sorting function, I set an array to that return value and use the correct index positions to store the values of moves and comparisons. I then print the header for the specific sorting method, and use a for loop to print each element in the array. There is a specific format needed to be followed so I use a print_count variable to ensure a new line is printed every seven elements and at the last element printed.

```
if (op_b AND !op_A)
    for (j = 0; j <= size; j++)
      sorted_array[j] = 0
    for (j = 0; j < size; j++)
      sorted_array[j] = array[j]
    move_compare = bubble_sort(sorted_array, size)
    moves = move_compare[0]
    comparisons = move_compare[1]
    printf("Bubble Sort\n%d elements, %d moves, %d compares\n")
    for (i = 0; i < print; i++)
      printf("%d”, sorted_array[i])
      print_count++;
      if (print_count == 7 OR i + 1 == print)
        printf("\n")
        print_count = 0
```

The following pseudo code was given to the class via asgn5.pdf

Bubble sort works by examining adjacent pairs of items. If the second item is smaller than the first, swap them. As a result, the largest element falls to the bottom of the array in a single pass. Since it is in fact the largest, we do not need to consider it again. So in the next pass, we only need to consider n −1 pairs of items. The first pass requires n pairs to be examined; the second pass, n − 1 pairs; the third pass n − 2 pairs, and so forth. If you can pass over the entire array and no pairs are out of order, then the array is sorted.

```
def Bubble_Sort(arr):
  for i in range(len(arr) - 1):
    j = len(arr) - 1
    while j > i:
      if arr[j] < arr[j - 1]:
        arr[j], arr[j - 1] = arr[j - 1], arr[j]
      j -= 1
  return
```

The following is the pseudocode for Shell Sort. Given the length of array n , the function gap (n) produces an array of gaps. The rules are that if $n \leq 2$, $n = 1$, else $n = 5 * n /11$. The array will be ranked from large to small. In the Shell_Sort(n), for each step in the array of gaps, it compares all the pairs that are away from each other by step in index and switches the elements in the pair if they are not sorted.

```
def gap(n):
  while n > 1:
    n = 1 if n <= 2 else 5 * n // 11
    yield n
```

```
def Shell_Sort(arr):
  for step in gap(len(arr)):
    for i in range(step, len(arr)):
      for j in range(i, step - 1, -step):
        if arr[j] < arr[j - step]:
          arr[j], arr[j - step] = arr[j - step], arr[j]
  return
```

Quicksort is a divide-and-conquer algorithm, utilizing a subroutine that partitions arrays into two subarrays by selecting an element from the array and designating it as a pivot. Elements in the array that are less than the pivot go to the left subarray, and elements in the array that are greater than or equal to the pivot go to the right subarray and returns the index that indicates the division between the partitioned parts of the array. Quicksort is then run recursively on the partitioned parts of the array, thereby sorting each array partition containing at least one element.

```
def Partition(arr, left, right):
  pivot = arr[left]
  lo = left + 1
  hi = right

  while True:
    while lo <= hi and arr[hi] >= pivot:
      hi -= 1

    while lo <= hi and arr[lo] <= pivot:
      lo += 1

    if lo <= hi:
      arr[lo], arr[hi] = arr[hi], arr[lo]
    else:
      break

  arr[left], arr[hi] = arr[hi], arr[left]
  return hi

def Quick_Sort(arr, left, right):
  if left < right:
    index = Partition(arr, left, right)
    Quick_Sort(arr, left, index - 1)
    Quick_Sort(arr, index + 1, right)
  return
```

Binary Insertion Sort is a special type of insertion sort which uses the binary search algorithm to find the correct position of an inserted element in an array. Insertion sort works by finding the correct position of the element in the array and then inserting it into its correct position. Searching for an element using binary search starts by identifying the element at the midpoint between either end of the array. If there is an element needed before the current element check the left side, after then check the right. Thus, it's clear that we are halving the search space each time we do a comparison, hence the name, binary search. Binary Insertion Sort uses binary search in order to determine where each element should go, reducing the number of comparisons between array elements we would ordinarily need for Insertion sort. For each

element in the array, simply run a binary search through the elements to the left of the current element in order to find the index in which it should go.

```
def Binary_Insertion_Sort(arr):
  for i in range(1, len(arr)):
    value = arr[i]
    left = 0
    right = i

    while left < right:
      mid = left + ((right - left) // 2)

      if value >= arr[mid]:
        left = mid + 1
      else:
        right = mid

    for j in range(i, left, -1):
      arr[j - 1], arr[j] = arr[j], arr[j - 1]

  return
```

These pseudocode sections are essential to my code for Assignment 5.