Assignment 5 Design - Jack Vento

Pre Lab

Part 1

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?

It should require 6 rounds of swapping. It performs swaps for 5 rounds and then requires one last final round of no swaps before being able to officially declare the array as sorted.

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

Since each pass only manages to put the largest element in its proper position, the worst case secnario for bubble sort is when the smallest element is at the end of the array: requiring the max amount of swaps and comps. In this case, I found it takes `n(n-1)` comparisons and half that many swaps. For example, an array of size 10 would take 90 comparisons and 45 swaps.

Part 2

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 gap determines how thoroughly the array will be sorted by the time we move to an insertion sort at the end (gap = 1). The better the gap, the less swaps will have to be performed at this final stage, which is great because comparisons are significantly less expensive than swaps.

By changing to a gap size of $4^k + 3 * 2^{k-1} + 1$ (Sedgewick, 1982), the complexity of Shellsort improves to $O(n^{4/3})$. As of now, there is no specific 1-to-1 relationship found between gap size and runtime complexity. Many of the best gap sequences were found through experimentation, rather than a specific methodology. There is not much information on *how* to find better gap sizes currently in the field.

Sources: GeeksForGeeks and HBFS.

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

By shifting the if statement proceding the swap in the innermost loop into the loop condition, there would be significantly less comparisons while still maintaining proper output. Additionally, saving the length of the array instead of calculating it in the loop conditions twice would be more efficient.

Part 3

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

On average, quicksort has a complexity of O(nLogn) despite having a worst case complexity of $O(n^2)$. This is actually not a dealbreaker as that worst case complexity only happens when an extreme pivot is chosen (greatest or smallest element). Over the years, there have been methods developed to ensure the pivot is almost always approximately the median (I don't particularly know how but my past professor said they exist), which leads to a consistent complexity of O(nLogn). By ensuring a good pivot is chosen, Quicksort is consistently fast.

Sources: GeeksForGeeks and Professor Ernsberger at OCC for my Data Structures class.

Part 4

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

It reduces the worst case comparison complexity from O(n) to O(Logn) since a binary search inside the loop helps find the correct insertion position logarithmically. Since this only affects the amount of comparisons performed, insertion sort still has a complexity of $O(n^2)$ due to the amount of swaps, but the c value is significantly lower. Although the Big O complexity is the same, binary insertion sort is much faster than a normal insertion sort

Part 5

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

Instead of implementing the same swap/comp repeatedly and trying to track statistics pointers in each file, I opted to define two counts and helpers for swapping/comparing in sorting.c and helper declarations in sorting.h for better encapsulation. Swap() uses a temporary variable to facilitate the swap and increments the moves counter three times per iteration. Compare() increments the comps count once and essentially works as an overloaded < operator. I hoped to get away with inline functions and forward declarations, but that did not work out. See the `sorting` section for further detail on current design and Design Changes for my process getting there.

Bubble Sort

Design:

Bubble sort is a classic yet highly inefficient algorithm that works by iterating through an array in pairs. With each iteration, the algorithm will swap the two elements in a pair if they are out of order until hitting our array bounds. At the end of each pass, the largest unsorted element will find its correct position, leading to one less pair being considered on the next iteration. This pattern continues until there are no more pairs to swap.

Pseudocode: Not noted in the following pseudocode, I use my own comparison and swap helpers defined in sorting.c. See `sorting` for further detail.

```
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
```

Bubble Sort (pseudocode)

Source: Lab Manual

Shell Sort

Design:

Insertion sort struggles with arrays that have their lexiographically adjacent items far apart, leading to many expensive swaps. Shell Sort helps alleviate this problem by working in gaps: swapping far apart elements in progressively smaller gaps. By the time the gap = 1 and we perform a classic insertion sort, the array should be significantly more sorted, leading to far less swaps.

See PreLab Pt2 for potential improvements and analysis of gap sizes.

Pseudocode: Not noted in the following pseudocode, I use my own comparison and swap helpers defined in sorting.c. See `sorting` for further detail.

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

gap (pseudocode)

```
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
```

Shell Sort (pseudocode)

Source: Lab Manual

Quicksort

Design:

Quicksort divides and conquers an array using pivots and partitions. After a pivot is chosen, we use a helper method (partition) to place the pivot in its correct position, shift all elements greater than the pivot into the right subarray, and shift all smaller elements into the left subarray. Those subarrays are then recursively sorted until we reach our base case (right >= left).

It is an incredibly quick algorithm with logarithmic performance that unfortunately can degrade to quadratic performance. See Prelab Pt3 for an explanation of this.

Pseudocode: Not noted in the following pseudocode, I use my own comparison and swap helpers defined in sorting.c. See `sorting` for further detail.

```
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
```

Quicksort (pseudocode)

Source: Lab Manual

Binary Insertion Sort

Design:

In normal insertion sort, the algorithm sorts by progressively growing a sorted section of the array and shrinking unsorted section. The first element is automatically marked as "sorted" at the start. Then, a new element is added to the sorted section with each iteration and shifted to the proper location in that segment. The algorithm ends when every element has been added to the sorted section and placed in the correct spot.

Binary insertion sort improves on this algorithm by using a binary search (one of the fastest searches possible) to find the correct position for each new element added to the sorted section. This unfortunately does not help solve the expensive swapping insertion sort does, but it does significantly reduce the amount of comparisons and speed up the algorithm as a whole.

Pseudocode: Not noted in the following pseudocode, I use my own comparison and swap helpers defined in sorting.c. See sorting for further detail.

```
1  def Binary_Insertion_Sort(arr):
2    for i in range(1, len(arr)):
3      value = arr[i]
4      left = 0
5      right = i
6
7      while left < right:
8        mid = left + ((right - left) // 2)
9
10       if value >= arr[mid]:
11         left = mid + 1
12       else:
13         right = mid
14
15     for j in range(i, left, -1):
```



```
16       arr[j - 1], arr[j] = arr[j], arr[j - 1]
17
18   return
```

Binary Insertion Sort (pseudocode)

Source: Lab Manual

Sorting

Design:

It primarily consists of a simple getopt loop that'll update the contents of a bool array (cmd) depending on what sorts are called, and optionally set array size, seed, and print amount if supplied.

I decided to use an array of bools (cmd) to track the command line arguments. cmd is indexed as follows: bubble(0), shell(1), quick(2), binary insertion(3). This allows for far easier argument management, readability, and any number of input combinations.

I will use two public helper functions: compare() and swap(). These allow for far easier statistic tracking, debugging, and avoiding implementation errors. The sorting algorithms will slightly suffer performance degradation having to call these functions so many times but by whatever incurred loss is significantly offset by the simplification provided by them.

I have one additional static helper: fill_array(). This helper uses the supplied seed to either fill the array for the first time or reset the array to its initial values. The implementation is the same, but the context of the latter case is when the function is called after a previous sorting algorithm already ran.

Pseudocode:

Main

- Define constant CMD_SIZE = 4
- Check if there aren't any command-line arguments supplied or no run preference. Print an error and return if so
- Initialize moves = comps = 0
- Initialize print_amt, seed, and array_size to defaults.
- Initialize all of cmd array to false
- while there are options to parse:
    - switch option
        - A:
            - Set all values in cmd to true
        - b, s, q, i:
            - Set respective bool status in cmd to true
        - p, r, n (if supplied):
            - Update print_amt, seed, array_size from defaults
- Allocate and default initialize array[array_size] on the heap (calloc)
- if (array = NULL)
    - Print error and return
- for i in range [0, CMD_SIZE):
    - fill_array(array, array_size, seed)
    - switch i:
        - Call respective sorting algorithm and print header
    - Print size, statistics (moves, comps), and array elements
    - Set statistics to 0

- Free array

fill_array(arr, size, seed)

- Supply seed to srand
- for i in range [0, array_size):
    - elem = rand()
    - elem = elem & 0x3FFFFFFF (mask to 30 bits max)
    - array[i] = elem

compare(elem1, elem2)

- Increment comps count
- Return elem1 < elem2

swap(*elem1, *elem2)

- Increment moves count by 3
- Swap elem1 and elem2 using a temp variable

Design Changes

Sorting Helpers:

I originally opted for inline functions to lessen the burden provided by calling compare() and swap() thousands of times in each sort. However, I ran into significant issues when it came to linking. I originally did not have a sorting header file and created it to help link the helpers to the rest of the sorts after realizing forward declarations were not a sufficient solution (without a header, each sort has no idea when or if those helpers are defined). I then found that even with a header, inline functions cannot access external variables due to their linkage. Since I needed both comps and moves to increment with each function call and inline functions make accessing those variables illegal, I decided to do away with inline functions. The performance gain from them would have been nice, but they are incompatible with the design I wanted: store my counts exclusively in sorting.c and let them be incremented through templated helpers called in each individual sort.