Michael Nguyen
mnguy181@ucsc.edu
2//2021

CSE13S Winter 2021
Assignment 5: Sorting: Putting your affairs in order
Design Document

PRE-LAB QUESTIONS

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?

In summary, Bubble Sort examines adjacent pairs of elements, swapping the two elements if the second element is smaller than the first element. In this example, let's conduct each pass of Bubble Sort.

First Pass: n pairs are examined, where n = 6 in this example

Pair 1: {8, 22} Since 8 is less than 22, no swap
Pair 2: {22, 7} Since 22 is greater than 7, swap (Array is now [8, 7, 22, 9, 31, 5, 13])
Pair 3: {22, 9} Since 22 is greater than 9, swap (Array is now [8, 7, 9, 22, 31, 5, 13])
Pair 4: {22, 31} Since 22 is less than 31, no swap
Pair 5: {31, 5} Since 31 is greater than 5, swap (Array is now [8, 7, 9, 22, 5, 31, 13])
Pair 6: {31, 13} Since 31 is greater than 13, swap (Array is now [8, 7, 9, 22, 5, 13, 31])

After the first pass of Bubble Sort, we have made in total 4 swaps done, but the numbers are still not sorted completely in ascending order.

Second Pass: n - 1 pairs are examined (since largest element fell to bottom of the array)

Pair 1: {8, 7} Since 8 is greater than 7, swap (Array is now [7, 8, 9, 22, 5, 13, 31])
Pair 2: {8, 9} Since 8 is less than 9, no swap
Pair 3: {9, 22} Since 9 is less than 22, no swap
Pair 4: {22, 5} Since 22 is greater than 5, swap (Array is now [7, 8, 9, 5, 22, 13, 31])
Pair 5: {22, 13} Since 22 is greater than 13, swap (Array is now [7, 8, 9, 5, 13, 22, 31])

After the second pass of Bubble Sort, we have made in total 7 swaps completed, but the numbers are still out of order.

Third Pass: n - 2 pairs are examined

Pair 1: {7, 8} Since 7 is less than 8, no swap
Pair 2: {8, 9} Since 8 is less than 9, no swap
Pair 3: {9, 5} Since 9 is greater than 5, swap (Array is now [7, 8, 5, 9, 13, 22, 31])
Pair 4: {9, 13} Since 9 is less than 13, no swap

After the third pass of Bubble Sort, we have made a total of 8 swaps.

Fourth Pass: n - 3 pairs are examined

Pair 1: {7, 8} Since 7 is less than 8, no swap
Pair 2: {8, 5} Since 8 is greater than 5, swap (Array is now [7, 5, 8, 9, 13, 22, 31])

Pair 3: {8, 9} Since 8 is less than 9, no swap

After the fourth pass of Bubble Sort, we have made a total of 9 swaps.

Fifth Pass: n - 4 pairs are examined

Pair 1: {7, 5} Since 7 is greater than 5, swap (Array is now [5, 7, 8, 9, 13, 22, 31])

Pair 2: {7, 8} Since 7 is less than 8, no swap

After the fifth pass of Bubble Sort, we have made a total of 10 swaps.

Sixth Pass: n - 5 pairs are examined

Pair 1: {5, 7} Since 5 is less than 7, no swap (Array is considered to be in order)

Thus, we can see that (excluding the sixth pass which served as a final check for the sorting of the numbers in ascending order), five passes/rounds of swapping were required to sort the numbers in ascending order using Bubble Sort. In total, to sort the numbers, 10 swaps were required.

2) How many comparisons can we expect to see in the worst case scenario for Bubble Sort?

Based on intuition, I would assume that the absolute worst case scenario for Bubble Sort would be if the numbers were put in reverse order (descending order), and we wanted Bubble Sort to sort in ascending order. In this case, given a list of numbers with n pairs, the required number of comparisons that we can expect to see in the worst case scenario would be:

n + (n - 1) + (n - 2) + … + 1 = (n[n-1])/2 comparisons

Example:

Numbers List = [5, 4, 3, 2, 1]

We want to use Bubble Sort to sort this list in ascending order. The number of pairs is 4.

First Pass:

{5, 4} Swap, List is now [4, 5, 3, 2, 1]

{5, 3} Swap, List is now [4, 3, 5, 2, 1]

{5, 2} Swap, List is now [4, 3, 2, 5, 1]

{5, 1} Swap, List is now [4, 3, 2, 1, 5]

Second Pass:

{4, 3} Swap, List is now [3, 4, 2, 1, 5]

{4, 2} Swap, List is now [3, 2, 4, 1, 5]

{4, 1} Swap, List is now [3, 2, 1, 4, 5]

Third Pass Result: List is now [2, 1, 3, 4, 5]

Fourth Pass Result: List is now [1, 2, 3, 4, 5]

Fifth Pass Result: List is deemed sorted as [1, 2, 3, 4, 5]

3) How would you revise the algorithm so the smallest element floats to the top instead?

The lab document provides some python code for how to implement Bubble Sort in Python. After looking at the code, we can see that the Bubble Sort code conducts a comparison that checks if the element at the current index of the array is less than the previous element. If it is, then swap the two elements. In this case, Bubble Sort sorts the array in ascending order (largest element floats to the top). To change this algorithm to have the array sorted in descending order (smallest element at the top), we can simply change the < operator to a > operator. This way, when the algorithm conducts a comparison, if the current element is greater

than the previous element, swap the elements. This would have the smallest element float to the top. An alternative route would be to have int i in the for loop begin at 0, and conduct comparisons between the element at index i and the next element at index i + 1. If the current element is less than the next element, swap, ending the iteration when i is equal to n-1.

Pre-lab Part 2:

1) The worst time complexity for Shell Sort depends on the sequence of gaps. Investigate why this is the case. How can you improve the time complexity of this sort by changing the gap size?

Because Shell Sort operates (in simple terms) as an improved Bubble Sort, its ability to efficiently sort is highly dependent on the gap size as well as the gap sequence. Thus, it is considered to be a less stable way of sorting than Bubble Sort (which is expected to always compare 2 neighboring elements in a pair). For example, if the gap sequence began with a gap size that was too big, it could potentially cause Shell Sort to make inaccurate swaps and more iterations than necessary. For example, given a List: [5, 10, 13, 2, 4, 1, 9, 3] where the initial gap size is 7. The first swap would be 3 and 5. However, if we were to choose a smaller gap size as the initial gap size, for example, 3, then {5, 2} and many others would be swapped in the first iteration. Thus, I think that finding the most optimal gap size (basically finding a middle ground between too big of a gap size and too small of a gap size) given the size of an array with a formula would definitely improve how efficient the sort is.

On another note, we also want to make sure that the gap sequence does not fall to a gap size of 1 too quickly since this would just be implementing Bubble Sort. The idea is to get elements into their relative ordered positions with Shell Sort.

Pre-lab Part 3:

1) Quicksort, with a worst 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.

Quick Sort operates by selecting a pivot, comparing the rest of the values in the array to that pivot, and sorting them to the right of the pivot (if greater than) and left (if less than). In doing so, Quick Sort is usually the most efficient sorting algorithm that uses comparisons. Its efficiency to sort is highly dependent on how "good" the pivot selected is. After doing some research, I found that there are 2 worst case scenarios for Quick Sort:

1) When the array is already sorted (ascending or descending) & the pivots chosen are the extremities (minimum or maximum in a list)
2) When the array contains a list of the same elements

Because these scenarios are quite rare, I would say that Quick Sort, in general, is the most efficient way of sorting with the use of comparisons, and thus, it is not doomed by its worst case scenarios even with a worst case time complexity of O(n^2).

Citations: https://stackoverflow.com/questions/4019528/quick-sort-worst-case

Pre-lab Part 4:

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 think that within each of the respective files that contains an implementation of each type of sort, I will include 2 global variables (ex: count_moves and count_comparisons) that

along with 2 helper functions swap() and compare(). Each time these functions are called, the respective count will be incremented by 1. In doing so, in sorting.c which contains my main() and includes the header files, we can simply call on whichever variables we need to display the number of moves and comparisons. While this answer might not be the most optimal way of keeping track of the number of moves and comparisons, this is the way I plan on doing it.

PURPOSE

The purpose of this program will be to implement and test each type of sort specified: Bubble Sort, Shell Sort, Quicksort, and Heapsort based upon the Python pseudocode that was provided on the lab document. The header files for each of the sorts bubble.h, shell.h, quick.h, and heap.h have been provided, and I will not be modifying these files.

The idea will be to successfully implement each sort, create a test harness for these implemented algorithms in sorting.c (containing my main()) that tests each of the sorts with an array of random elements, and finally showcase statistics about each sort and its performance. Specifically, the program will be gathering statistics about the size of the array provided, the number of moves/swaps required, and the number of comparisons required for each sort, and print these statistics to stdout. The program will also accept command-line options: one for enabling all the sorting algorithms, one for enabling each respective sort, one for setting the random seed, one for printing out the elements, and another command-line option to specify the size of the array. By default, the seed will be 7092016, and the size of the array will be 100. The default number of elements will also be 100, but if the size of the array specified is less than the number of specified elements, then the program will print out the entire array and nothing more.

To support these command-line options, the program will implement a set that utilizes bits to track which command-line options are specified when the program is run. The set ADT will be implemented in the file set.c. Also, Quicksort will also need a stack ADT since the program will implement the iterative version (not the recursive version). The stack ADT will be implemented in another file stack.c.

BUBBLE SORT DESCRIPTION

Bubble Sort operated by comparing neighboring elements in pairs. If the second item is less than the first, swap. In doing so, the largest element will go to the top of the array with a single pass. In the second pass, we only need to consider n - 1 pairs, and so on. Please do note that pseudocode in Python on how to implement Bubble Sort in bubble.c was provided in the lab document. As a result, I will instead be describing how I will conduct comparisons in Bubble Sort, and include an example. The sort will accept a pointer to the array, and will be given the length of the array to conduct Bubble Sort. In addition, the header file, bubble.h is also provided, and I will not be modifying this file.

The following is an example of how Bubble Sort compares neighboring elements. This example was taken from the pre-lab question.

Example: Let’s consider an array = [8, 22, 7, 9, 31, 5, 13] where n is the length of the array = 7
Note: I will be describing how each iteration of the for loop works, the comparisons, and note the state of the array before each iteration of the for loop() in the pseudocode.

Starting Array: Array = [8, 22, 7, 9, 31, 5, 13]
First Iteration: 6 pairs are examined, index starting at 1, going to index n = 7

Pair 1: {8, 22} Since 8 is less than 22, no swap
Pair 2: {22, 7} Since 22 is greater than 7, swap (Array is now [8, 7, 22, 9, 31, 5, 13])
Pair 3: {22, 9} Since 22 is greater than 9, swap (Array is now [8, 7, 9, 22, 31, 5, 13])
Pair 4: {22, 31} Since 22 is less than 31, no swap
Pair 5: {31, 5} Since 31 is greater than 5, swap (Array is now [8, 7, 9, 22, 5, 31, 13])
Pair 6: {31, 13} Since 31 is greater than 13, swap (Array is now [8, 7, 9, 22, 5, 13, 31])

After the first iteration of Bubble Sort, we have made in total 4 swaps done, but the numbers are still not sorted completely in ascending order. Bubble Sort then decrements n.

Starting Array: Array = [8, 7, 9, 22, 5, 13, 31]
Second Iteration: 5 pairs are examined, index starting at 1, going to index n = 6
Pair 1: {8, 7} Since 8 is greater than 7, swap (Array is now [7, 8, 9, 22, 5, 13, 31])
Pair 2: {8, 9} Since 8 is less than 9, no swap
Pair 3: {9, 22} Since 9 is less than 22, no swap
Pair 4: {22, 5} Since 22 is greater than 5, swap (Array is now [7, 8, 9, 5, 22, 13, 31])
Pair 5: {22, 13} Since 22 is greater than 13, swap (Array is now [7, 8, 9, 5, 13, 22, 31])

After the second iteration of Bubble Sort, we have made in total 7 swaps completed, but the numbers are still out of order. Bubble Sort once again decrements n.

Starting Array: Array = [7, 8, 9, 5, 13, 22, 31]
Third Iteration: 4 pairs are examined, index starting at 1, going to index n = 5
Pair 1: {7, 8} Since 7 is less than 8, no swap
Pair 2: {8, 9} Since 8 is less than 9, no swap
Pair 3: {9, 5} Since 9 is greater than 5, swap (Array is now [7, 8, 5, 9, 13, 22, 31])
Pair 4: {9, 13} Since 9 is less than 13, no swap

After the third iteration of Bubble Sort, we have made a total of 8 swaps. Bubble Sort decrements n.

Starting Array: Array = [7, 8, 5, 9, 13, 22, 31]
Fourth Iteration: 3 pairs are examined, index starting at 1, going to index n = 4
Pair 1: {7, 8} Since 7 is less than 8, no swap
Pair 2: {8, 5} Since 8 is greater than 5, swap (Array is now [7, 5, 8, 9, 13, 22, 31])
Pair 3: {8, 9} Since 8 is less than 9, no swap

After the fourth iteration of Bubble Sort, we have made a total of 9 swaps. Bubble Sort decrements n.

Starting Array: Array = [7, 5, 8, 9, 13, 22, 31]
Fifth Iteration: 2 pairs are examined, index starting at 1, going to index n = 3
Pair 1: {7, 5} Since 7 is greater than 5, swap (Array is now [5, 7, 8, 9, 13, 22, 31])
Pair 2: {7, 8} Since 7 is less than 8, no swap

After the fifth iteration of Bubble Sort, we have made a total of 10 swaps. Bubble Sort decrements n.

Starting Array: Array = [5, 7, 8, 9, 13, 22, 31]
Sixth Iteration: 1 pair is examined, index starting at 1, going to index n = 2

Pair 1: {5, 7} Since 5 is less than 7, no swap (Array is considered to be in order)
Bubble Sort has finished iterating.
In total, to sort the numbers, 10 swaps were required. 21 comparisons were needed to finish sorting.

SHELL SORT DESCRIPTION

Shell Sort serves as another variation of insertion sort that utilizes comparisons on elements a specific distance (gap) apart to swap elements more efficiently into their relative ordered positions. Once again, because the pseudo code for Shell Sort has been provided on the lab document, I will not be copying the pseudocode in Python down on this design document, but rather showcasing an example of how comparisons based on a specific gap size will be conducted under Shell Sort (basically doing a rundown of how I expect the code to sort an example array under Shell Sort). Again, the header file, shell.h will be provided.
Example: Let's consider an array = [8, 22, 7, 9, 31, 5, 13] where n is the length of the array = 7
Let's say that the initial gap size is 6.
First Iteration:

{8, 13} are compared. Since 8 < 13, no swap occurred

The gap size then reduces to 4.
Second Iteration:

{8, 31} are compared. Since 8 < 31, no swap occurred
{22, 5} are compared. Since 22 > 5, swap
{7, 13} are compared. Since 7 < 13, no swap

Thus, the array after second iteration = [8, 5, 7, 9, 31, 22, 13]
The gap size then reduces to 2.
Third Iteration:

{8, 7} are compared. Since 8 > 7, swap
{8, 31} are compared. Since 8 < 31, no swap
{31, 13} are compared. Since 31 > 13, swap
Current State of Array = [7, 5, 8, 9, 13, 22, 31]
{5, 9} are compared. Since 5 < 9, no swap
{9, 22} are compared. Since 9 < 22, no swap

Thus, the array after the third iteration = [7, 5, 8, 9, 13, 22, 31]
The gap size then reduces to 1.
Fourth Iteration:

{7, 5} are compared. Since 7 > 5, swap
{7, 8} are compared. Since 7 < 8, no swap
{8, 9} are compared. No swap
{9, 13} are compared. No swap
{13, 22} are compared. No swap
{22, 31} are compared. No swap

Thus, the array is deemed sorted as [5, 7, 8, 9, 13, 22, 31].
In this case, the number of comparisons required to sort this array were 15 comparisons, and only 4 swaps were needed to sort this array in ascending order under Shell Sort. This is an example of how I would anticipate Shell Sort to function when implemented.

QUICKSORT DESCRIPTION

Quicksort on average, is the most efficient sorting algorithm that partitions arrays into subarrays by selecting an element from the array to serve as a pivot. The elements are then compared to the pivot: elements less than the pivot go to the left of the pivot while elements greater than the pivot go into the right subarray. Pseudocode for how to implement functions partition() and quicksort() have already been provided in the lab document. Please do note that Quicksort will need to be implemented with a stack ADT since it is not the recursive version but rather the iterative Quicksort. Top level design and pseudocode for the stack ADT will be provided further below. For this section, I will be describing an example of how I expect Quicksort to sort using comparisons and pivots. The header file, quick.h, has also been provided, and I will not be allowed to modify this file.

Example: Let's consider an array = [8, 22, 7, 9, 31, 5, 13] where n is the length of the array = 7
The pivot calculated is going to be (lo + [hi - lo])/2, where lo will be the index of the leftmost element and hi will be the rightmost element's index. In this case: the pivot selected will be

(0 + [6 - 0]) / 2 = 3

Thus, the pivot will be selected as the element at array[3] which is 9.
First partition:

{8, 9} are compared. Since 8 < 9, no swap
{9, 13} are compared. Since 13 > 9, no swap
{22, 9} are compared. Since 22 > 9, hold off on swapping
{5, 9} are compared. Since 5 < 9, {22, 5} are swapped
Current state: array = [8, 5, 7, 9, 31, 22, 13]
{7, 9} are compared. Since 7 < 9, no swap
{9, 31} are compared. Since 9 < 31, no swap

Array = {{8, 5, 7}, 9, {31, 22, 13}}
New Subarrays: {8, 5, 7} and {31, 22, 13}
For the first subarray: pivot is equal to 1

{8, 5} are compared. Since 8 > 5, swap
{8, 7} are compared. Since 8 > 7, swap
{5, 7, 8} sorted occurs

For the second subarray: pivot is 5

{31, 22} are compared. Since 31 > 22, hold off on swapping
{22, 13} are compared. Since 22 > 13, swap {31, 13}

Thus, after two partitions, the array has been sorted in order [5, 7, 8, 9, 13, 22, 31].

HEAPSORT DESCRIPTION

Heapsort essentially starts off by creating a heap by taking an array and fixing it so that the largest element is at the front, and its left and right childs will be located at 2k and 2k+1 respectively. The second step is to remove the largest element and place it at the end of the sorted array. The first element of the array will always be the smallest element. Each time the largest element is removed, we need to fix the heap once again.

Example: Let's consider an array = [8, 22, 7, 9, 31, 5, 13] where n is the length of the array = 7
To fix this array into a max heap, we need to fix the array. In essence, the max heap would look like:
[31, 13, 22, 5, 7, 8, 9]

We then “remove” 31 and place it at the end of the sorted array.
Now, array = [13, 22, 5, 7, 8, 9, 31]
We then fix the heap.
Array = [22, 13, 9, 5, 7, 8, 31]
Sort the array once again and fix the heap.
Array = [13, 9, 8, 5, 7, 22, 31]
Sort the array by placing the largest value at the end & fix the heap.
Array = [9, 8, 5, 7, 13, 22, 31]
Keep sorting until we reach:
[5, 7, 8, 9, 13, 22, 31]

TOP LEVEL FOR STACK ADT

The top level design of my code for stack.c is given by the following pseudocode. Please note that this is just a brief general outline of how I plan to approach the programming assignment in regards to implementing the stack ADT. The stack ADT will be needed for the iterative Quicksort.

```
Define minimum capacity of the stack as 16
Struct Stack
        Declare fields: u32 top, u32 capacity, u64 pointer to items
stack_create(void)
        Memory allocation for the stack
        Set top to 0, capacity to minimum capacity
        Dynamically allocate memory for the items
        Return a pointer to the stack
stack_delete(stack **s)
        Free all items
        Free the pointer of the stack
        Set the pointer to null
stack_empty(stack *s)
        Return true if top is pointing to 0, false otherwise
stack_push(stack *s, u64 x)
        Check if top is pointing to the stack’s capacity
                Reallocate memory for the stack if so using realloc()
                Return false if realloc() returns a NULL pointer/failed
        Set stack’s items at the top = x
        Increment top
        Return true
stack_pop()
        Check if the stack is empty first
        Decrement top
        Set x equal to the items[top]
stack_print()
        For loop() that iterates through the stack, ending when equal to top
                pop()
                printf() the item returned when popping
```

TOP LEVEL FOR SET ADT

The top level design of my code for set.c is given by the following pseudocode. Please note that this is just a brief general outline of how I plan to approach the programming assignment in regards to implementing the set for my sorting.c test harness.

```
Typedef u32 Set
Set set_empty(void)
    Return 0
set_member()
    Mask is 1 left shifted by x
    AND the set and mask
    Right shift the result by x
set_insert()
    Mask is 1 left shifted by x
    OR the set and mask
    Return the set
set_remove()
    Mask is ~(1 left shifted by x)
    AND the set and mask
    Return the set
set_intersect()
    AND the two sets
    Return the result
set_union()
    OR the two sets
    Return the result
set_complement()
    Return ~(set)
set_difference()
    Second set = ~(second set)
    Return first set AND second set
```

TOP LEVEL FOR SORTING & TEST HARNESS

The top level design of my code for sorting.c is given by the following pseudocode. Note that this is the implementation of the main() function that ties everything in this assignment together. Its purpose is to test each sorting method while supporting command line arguments, and print out statistics related to each of the sorting methods.

```
Enum sorts(
    Bubble
    Shell
    Heap
    Quick
Int main():
    Set default seed to 7092016
    Set default size to 100
```

Set default elements to 100
Use srandom() to fill array
While loop using getopt() ending at -1
    Case ‘a’
        Loop that uses insert_set()
    Case ’b’
        insert_set(bubble)
    Case ‘s’
        insert_set(shell)
    Case ‘q’
        insert_set(quicksort)
    Case ‘h’
        insert_set(heap)
    Case ‘r’
        Set the random seed to the specified seed
    Case ‘n’
        Set the array size to the specified size
    Case ‘p’
        Set the number of printed elements to the specified number of elements
Loop that goes through each set_member
    Conduct the sorting method
    Print the statistics related to that sorting method,
    Print out the elements loop()
    starting from 0 to # of specified elements to the printed