Jessie Zhu jelzhu@ucsc.edu CSE 13s, Spring 2021 Wednesday, April 21, 2021

Assignment 3: Sorting, Putting your affairs in order

Design Document

For this assignment, we will be implementing sorting algorithms Bubble Sort, Shell Sort, and two Quicksorts. These sorts will keep track the number comparisons and moves made. A user can choose how many and how random items are to be sorted.

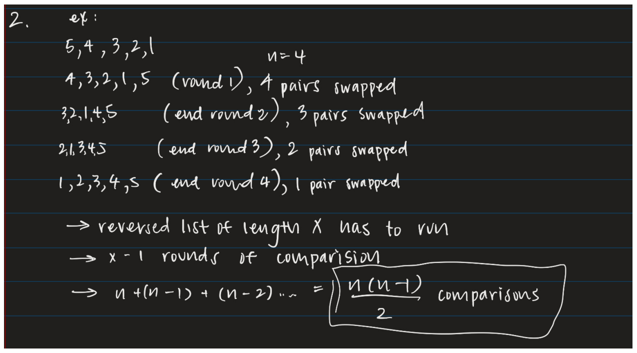
The purpose of this assignment is to become familiar with different types of sorting algorithms and understand how complexity works with each type of sorting. By implementing them, we will have a better understanding of how the complexity of each sort is determined and any other patterns to be aware of.

Prelab:

Part 1:

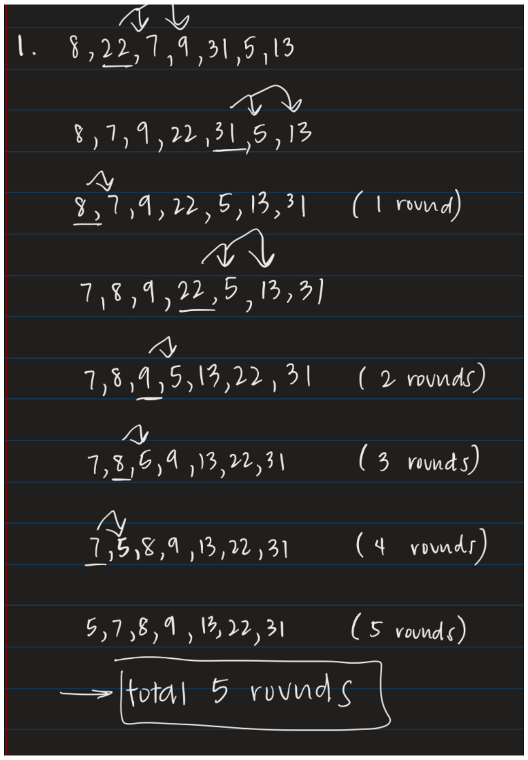
1. There will be a total of 5 rounds of swaps.

2. In the worst case scenario for bubble sort, we would need to swap every comparison. In the first round, there will be n pairs of comparisons and swaps. In the next round there will be n-1, then n-2… for a total of n(n+1)/2 comparisons in a worst case scenario.



2. ex:
5,4,3,2,1
n=4
4,3,2,1,5 (round 1), 4 pairs swapped
3,2,1,4,5 (end round 2), 3 pairs swapped
2,1,3,4,5 (end round 3), 2 pairs swapped
1,2,3,4,5 (end round 4), 1 pair swapped

$\rightarrow$ reversed list of length X has to run
$\rightarrow$ x - 1 rounds of comparision
$\rightarrow$ $n + (n-1) + (n-2) \ldots = \boxed{\frac{n(n-1)}{2} \text{ comparisons}}$



1. 8, 22, 7, 9, 31, 5, 13

8, 7, 9, 22, 31, 5, 13

8, 7, 9, 22, 5, 13, 31 (1 round)

7, 8, 9, 22, 5, 13, 31

7, 8, 9, 5, 13, 22, 31 (2 rounds)

7, 8, 5, 9, 13, 22, 31 (3 rounds)

7, 5, 8, 9, 13, 22, 31 (4 rounds)

5, 7, 8, 9, 13, 22, 31 (5 rounds)

$\rightarrow$ $\boxed{\text{total 5 rounds}}$

(Drawn on my iPad)

Part 2:

1. Time complexity depends on the gaps because the comparisons being made during the sort depend on that number. For gap sizes, technically any sequence of number will sort as long as the smallest number is 1, ignoring the efficiency. There are many common sequences that are used for shell sorting: Shell (O(n^2)), Knuth ((O)n^3/2), Pratt (nlog^2(n)) and of these common ones, Pratt sequence is has the

best worst time complexity. The way to improve the time complexity of a shell sort, would be to use a different sequence that outperforms our chosen Pratt's sequence. Ciura's sequence, an experimentally derived sequence of number, outperforms Pratt sequence in terms of quantity of comparisons.

source: https://en.wikipedia.org/wiki/Shellsort#Gap_sequences

Part 3:

1. Although Quicksort has a worst case time complexity of O(n^2), it is faster than other sorting methods. In most cases, the pivot can be chosen to avoid the worse case, which makes it relatively fast. The loop inside can usually be optimized so overall this is still an efficient sorting method despite the bad worst time case complexity.

source: https://www.geeksforgeeks.org/quick-sort/

Part 4:

1. I plan on using global variables to keep track of moves and comparisons. This way all files that need to access the variable can via the header file that contains the extern variables. I will create my own header file called moves.h to hold the extern variables and anywhere I need the extern variables, I will declare the extern variables in the file.

Pseudocode

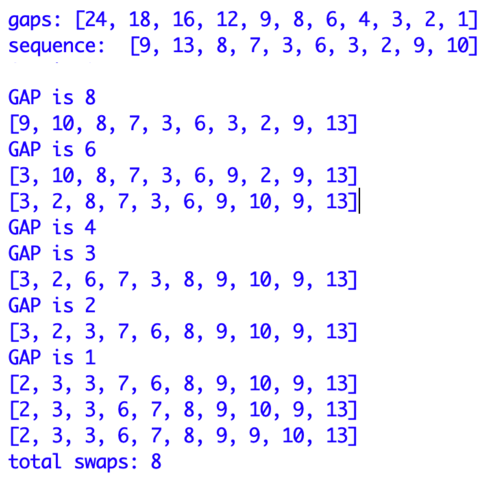
Bubble Sort:

Bubble Sort will compare every adjacent pair of objects in a line and swap if they are out of order. This process repeats until there are no swaps made in a round of comparison. In our code that would look like a while loop, running only if there was a swap made in the previous run of the loop. Then inside the while loop, there would be a for loop for all the adjacent pair of comparisons in one round of comparison. The worst case time complexity is O(n^2)

Bubble sort:

```
create boolean for status of whether a swap was made
while a swap has been made:
    set swap status to false
    for each pair in list:
        if out of order, swap pair and set pair status to true
```

Shell Sort:



```
gaps: [24, 18, 16, 12, 9, 8, 6, 4, 3, 2, 1]
sequence:  [9, 13, 8, 7, 3, 6, 3, 2, 9, 10]

GAP is 8
[9, 10, 8, 7, 3, 6, 3, 2, 9, 13]
GAP is 6
[3, 10, 8, 7, 3, 6, 9, 2, 9, 13]
[3, 2, 8, 7, 3, 6, 9, 10, 9, 13]
GAP is 4
GAP is 3
[3, 2, 6, 7, 3, 8, 9, 10, 9, 13]
GAP is 2
[3, 2, 3, 7, 6, 8, 9, 10, 9, 13]
GAP is 1
[2, 3, 3, 7, 6, 8, 9, 10, 9, 13]
[2, 3, 3, 6, 7, 8, 9, 10, 9, 13]
[2, 3, 3, 6, 7, 8, 9, 9, 10, 13]
total swaps: 8
```

(Example Shell Sort, made using IDLE)

Shell Sort uses a sequence of gaps to determine how numbers are compared.

We starts with the biggest gap that is within the values of the indices of the list. For every gap value, it will sort every item gap values apart until all items will have been compared. A for loop will loop through all the gaps until the last gap, which is 1. Inside will be a series of for loops and while loops that will sort every set of items that are gaps apart (which is repeated for all gaps).

```
Shell Sort:

    for each gap:

        check every set of items of gap values apart and sort only those items
        at a time.

        (the above will be repeated for every valid gap value)
```

<u>Quick Sort:</u>

Quick Sort sort using a divide and conquer method. Partition is used to split up the array during the quick sort. A pivot is picked, then objects are then divided by whether or not they are bigger than the pivot. Each side will keep dividing using the same method until everything has been sorted. For the assignment we will create an implementation using the stack and another implementation using the queue.

Partition works by using a pivot to determine where the divide is. Then two index counters are used to keep track of sorting. A while loop will handle the comparing and inside there will be two more while loops checking the positions of index i and j which then will move the index accordingly. Index j will be returned at the end.

Quick Sort:

```
Partition(array, lo, hi):
    get pivot
    i = index counter: lo - 1
    j = index counter: hi + 1
    while I is less than j:
        increment i
        while item at position i is less than pivot:
            increment i
        decrement j
        while item at position j is less than the pivot:
            decrement j
        if I is less than j, swap items at i and j
    return j
```

Quicksort using stack versus queue is very similar. The only difference is in how each structure stores an item. Quicksort begins by setting lo and hi values. Then both are put into stack/queue. Then a while loop repeats the next part until the stack/queue are empty. Hi and lo values are popped/dequeued. Then partition is used to split and a partition index is returns to show where the division occurred. If lo is less than the partition index, lo and p are put into stack/queue. If hi is greater than partition index + 1, then add p+1 and hi to stack.

```
quicksort_stack(array):
    set lo, hi
    create empty stack
    put lo and hi in stack
    while stack is not empty:
        hi, lo = pops from stack (top) (ORDER MATTERS)
        partition
        if lo is less than the partition, add lo to stack, add p to stack
        if hi is greater than partitions+1, add to stack p+1, add hi to stack
```

```
quick_sort_queue(array):
    set lo, hi
    create empty queue
    queue lo and hi
    while queue is not empty
        dequeue items to lo and hi (ORDER MATTERS)
        partition
        if lo is less than p, add queue lo and p
        if hi is greater than p+1, queue p+1 and hi
```

Jessie Zhu jelzhu@ucsc.edu CSE 13s, Spring 2021 Wednesday, April 21, 2021

Stack and Queue:

- Stack.c will have the following methods: (define the struct)
  - stack_create: creates the stack with define attributes of top, capacity, items and allocated space
  - stack_delete: deletes the created stack so prevent memory leaks when done with stack
  - stack_full: checks if stack is full, compare if top equals capacity
  - stack_size: returns number of items, which is basically top
  - stack_push: adds item in parameter to the stack, and move top to next available space
  - stack_pop: points pointer in parameter to the popped item, decrements the stack pointer
  - stack_print: prints all of stack
- Queue.c will have the following methods: (define the struct)
  - queue_create: creates the stack with defined attributes of head, tail, size, capacity, items and allocated space.
  - queue_delete: deletes the created queue to prevent memory leaks when done with queue
  - queue_full: checks if queue is full, compare if size equals capacity
  - queue_size: returns number of items, return queue.size
  - enqueue: adds item in parameter to the queue, and move head to next available space
  - dequeue: points pointer in parameter to the dequeued item, increments the tail
  - stack_queue: prints all of queue

<u>Design Process:</u>

- The majority of the lab wasn't hard to figure out, but there were a lot of small errors in my code were due to stupid mistakes or pointer issues (I am not as familiar with pointers). Because stack and queue processes are very similar, it wasn't hard to figure out the corresponding functions that weren't provided to us.

- The python code was very easy to understand although there were some things in python that had to be implemented slightly differently
- There were no redesigns in structure of my pseudocode, but there were minute details in the specific c code that were changed. Specifically the python function pop is different than what c can do. I tried to catch the return value, thinking it was the popped value, rather than passing a pointer to catch the value.
- I had issues placing the places to increments the comparison counter. I think this was due to the while loop having condition1&&condition2 as the while condition. The comparison counter for that was very confusing so mine is slightly different, but still within the margin of error.
- My code had memory leaks when I checked it, but that was because I forgot to use the stack/queue_delete. After adding that in, my memory leak issue was fixed.
- I think overall the lab wasn't extremely challenging but because I had started very late, it was a lot of stuff to do.