Assignment 3 Design Document

Joshua Valladares
CSE 13S
Winter 2021
UC Santa Cruz

Pre-lab Part 1

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

Round 1: [8, 7, 9, 22, 5, 13, 31]
Round 2: [7, 8, 9, 5, 13, 22, 31]
Round 3: [7, 8, 5, 9, 13, 22, 31]
Round 4: [7, 5, 8, 9, 13, 22, 31]
Round 5: [5, 7, 8, 9, 13, 22, 31]
Success!
This took 5 rounds.

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.

The worse case scenario, for n items, will require us to compare elements over and over until we need to compare the first element with the second element, with 1 comparison in the final round. Only then are we absolutely guaranteed that the list is sorted. This will result in:
`n + (n-1) + (n-2) + ... + 1 = n * (n + 1) / 2`
comparisons, as is given in the PDF. This is the the number of comparisons required in the worst case.

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? Cite any sources you used.

It seems to me that the key component that makes Shell Sort faster than Insertion Sort is the fact that by using gaps, you are able to move elements that are very far away from where they need to be to only a few swaps away. This means that if you use a “smart” sequence of gaps, you will be able to minimize the number of swaps you need to make as well as the number of gaps to to iterate over. Due to this, the time complexity of Shell Sort seems dependent on how you determine you gap sequence.

Sources used:

- tutorialspoint.com

Pre-lab 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.

After sufficient research, I have come to the conclusion that the time complexity of Quicksort is overall dependent on how the pivot point is chosen. Generally, it seems that we want to be able to pick a pivot point such that for an arbitrary list, we are putting half of the elements on each side of the pivot. This allows us to minimize the number of times we need to call partition thus decreasing the time complexity.

In the worst case, we are picking our pivot points so poorly so that each call to partition splits the list into *N - 1* and *1* elements for an *N* element list. To reach this level of inefficiency would almost require the programmer to intentionally pick the worst possible pivot points. Therefore, Quicksort is certainly not doomed by its worst case. In fact, it can be avoided quite easily on average by simply ensuring that a competent pivot selection is used.

Sources used:

- baeldung.com

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.

> To tackle this problem, what I plan on doing is defining two *extern* variables to keep track of both moves and comparisons. In each sorts respective file, I will link this extern variable so that it can be modified in these files. I will begin each sort by setting both of these to 0 and increment them as the sort runs. Finally, the sort printing behavior will use this variables and the process will repeat for each sort.

Lab Design

In this section I will be providing the pseudocode for all the different methods that I will be implementing.

Pseudocode for Set

I will be using the code provided in the repository by Eugene Chou for Set.

Pseudocode for swaping two elements of an array

```
swap( arr[], index_one, index_two)
    temp = arr[index_one]
    arr[index_one] = arr[index_two]
    arr[index_two] = temp
```

Pseudocode for Stack

Code for stack_create and stack_delete are provided in the assignment PDF and I will use these.

```
boolean stack_empty(Stack s)
    return size(s) == 0

boolean stack_full(Stack s)
    return size(s) == capacity(s)

Integer stack_size(Stack s)
    return s.top

boolean stack_push(Stack s, Integer num)
    if stack_full(s)
        return false
    s.items[top] = num
    top = top + 1
    return true

boolean stack_pop(Stack s, Integer result)
    if stack_empty(s)
        return false
    result = s.items[top - 1]
    top = top - 1
    return true

void stack_print(Stack s)
    for(Integer i in s)
        print(i)
    return
```

Pseudocode for Queue

I will be using the same constructor and destructor methods that are given for stack in the PDF, except I will also be sure to initialize head and tail to 0 as well in the constructor.

```
boolean queue_empty(Queue q)
    return q.tail == q.head

boolean queue_full(Queue q)
    return ((q.tail + 1) mod q.capacity) == q.head

Integer queue_size(Queue q)
    return (q.tail - q.head + q.capacity) mod q.capacity

boolean enqueue(Queue q, Integer x)
    if queue_full(q)
```

```
        return false
    q.items[tail] = x
    q.tail = (q.tail + 1) mod q.capacity
    return true

boolean dequeue(Queue q, Integer x)
    if queue_empty(q)
        return false
    x = q.items[head]
    q.head = (q.head + 1) mod q.capacity
    return true

void print_queue(Queue q)
    for i starting at head and ending at tail, looping around if necessary
        print(q.items[i])
    return
```

Pseudocode for Command-line Options

```
cmd_set = []
seed = 1337453 ? no_input : input_seed
size = 100 ? no_input : input_size
elements = 100 ? no_input : input_elements
for(options given by user)
    if(option = 'a')
        cmd_set = universal_set
        break
    cmd_set.add(option)
```