Ruchit Patel

CSE13S, Spring 2021

Prof. Darrell Long

18 April 2021

Assignment 3: Sorting: Putting your affairs in order

Brief Description:

- In this assignment, three sorting algorithms are implemented: Bubble sort, Shell sort, and Quick sort.
- Bubble sort works on an integer array passed as a parameter.
- Shell sort also works on an integer array with the gap sequence 3-smooth or 2^p3^q.
- Quick sort is implemented using two abstract data types: a stack and a queue.
- Arguments: -a (run all three sorting algorithms),
  -b (run bubble sort),
  -s (run shell sort),
  -q (run quick sort using a stack),
  -Q (run quick sort using a queue),
  -r (sets the random seed),
  -n (sets the array size),
  -p (prints out *n* elements of an array).

Pre-Lab Questions:

1. Part I
    a. How many rounds of swapping for [8, 22, 7, 9, 31, 5, 13] to be in ascending order? (Total Rounds: 6. Total Comparisons: 21)
        i. Round 1:
            1. Array = [8, 22, …]. Comparisons: 1. Swaps: 0.
            2. Array = [8, 7, 22…]. Comparisons: 2. Swaps: 1.
            3. Array = [8, 7, 9, 22…]. Comparisons: 3. Swaps: 2.

4. Array = [8, 7, 9, 22, 31, …]. Comparisons: 4. Swaps: 2.
5. Array = [8, 7, 9, 22, 5, 31,…]. Comparisons: 5. Swaps: 3.
6. Array = [8, 7, 9, 22, 5, 13, 31]. Comparisons: 6. Swaps: 4.

ii. Round 2:

1. Array = [7, 8, ...]. Comparisons: 7. Swaps: 5.
2. Array = [7, 8, 9, …]. Comparisons: 8. Swaps: 5.
3. Array = [7, 8, 9, 22 …]. Comparisons: 9. Swaps: 5.
4. Array = [7, 8, 9, 5, 22 …]. Comparisons: 10. Swaps: 6.
5. Array = [7, 8, 9, 5, 13, 22, 31]. Comparisons: 11. Swaps: 7.

iii. Round 3:

1. Array = [7, 8, ...]. Comparisons: 12. Swaps: 7.
2. Array = [7, 8, 9, …]. Comparisons: 13. Swaps: 7.
3. Array = [7, 8, 5, 9 …]. Comparisons: 14. Swaps: 8.
4. Array = [7, 8, 5, 9, 13, 22, 31]. Comparisons: 15. Swaps: 8.

iv. Round 4:

1. Array = [7, 8, ...]. Comparisons: 16. Swaps: 8.
2. Array = [7, 5, 8, ...]. Comparisons: 17. Swaps: 9.
3. Array = [7, 5, 8, 9, 13, 22, 31]. Comparisons: 18. Swaps: 9.

v. Round 5:

1. Array = [5, 7, ...]. Comparisons: 19. Swaps: 10.
2. Array = [5, 7, 8, 9, 13, 22, 31]. Comparisons: 20. Swaps: 10.

vi. Round 6:

1. Array = [5, 7, 8, 9, 13, 22, 31]. Comparisons: 21. Swaps: 10.

- Thus, it would take 6 rounds and 21 comparisons for the above array to be sorted in ascending order using bubble sort.

b. How many comparisons for bubble sort in the worst case scenario?

i. The number of comparisons would remain the same no matter the order of the elements in the array. Number of swaps, on the other hand, varies depending on the order of the array.

ii. From the previous example, it could be seen that after every round the number of comparisons decrease by one (since the upper limit is decreased by one each round). Thus, total number of comparisons would be: (n-1) + (n-2) + (n-3) + … + 1 + 0 which is equal to n (n+1) - (1+2+...+n-1). Thus, the number of comparisons can be determined by the following formula:

$$n(n + 1) - (0.5n(n + 1))$$

$$= \frac{n(n+1)}{2}$$

$$= 0.5n^2 + 0.5n \qquad O(n^2)$$

iii. The number of times we swap is totally dependent on the list order. For example, if we had to swap every time (the previous element is always larger), then the swap amount would be the same as the comparison one, namely $0.5n^2 + 0.5n$. Thus, the worst case scenario would be $O(n^2)$. This could happen when the whole array is in descending order and it needs to be in ascending order and vice versa.

2. Part II
   a. Why does shell sort's time complexity depend on the gap sequence? How can it be improved?
      i. Since the shell sort compares elements at a certain gap, the farther the comparison, the lower the swaps as the gap keeps getting smaller. For example, if the least element of the array was at the end of the array, it would be swapped in the first round (if the gap is large enough). However, if it is not swapped then at the later rounds, it would have to be compared more times and swapped more times to get it to the front.
      ii. The gap sequence can be improved if there is some sort of knowledge about the data itself. For example, if the array is in ascending order, then cutting the gap in half every time seems like a plausible solution. Also, for each round, more swapping would be favorable. Thus, making the gap

sequence as large as possible might not be the best solution but maximum swapping after first comparison seems like an efficient solution.

iii. Reference: https://en.wikipedia.org/wiki/Shellsort

3. Part III
    a. Why is quicksort not doomed by its worst case complexity O(n^2)?
        i. The time complexity for quicksort completely depends on the pivot point. If the pivot point happens to be the maximum element in the list, then the sublist is only reduced by one element. However, the probability of it happening each time decreases as the size of the sublists decreases. Thus, each time the list to be sorted gets reduced by half and we get the average-case complexity O(n logn).
        ii. Reference: https://en.wikipedia.org/wiki/Quicksort

Descriptions/Implementation:

Argument Parsing

- Arguments are parsed using getopt GNU utility.
- If a valid argument is encountered, an integer is added to an abstract data type Set (a bit mask).
- For example, if -a is encountered, inside the case for -a, an integer value in an enum is added to the Set.
- After the while loop for parsing the arguments, it is checked whether any sorting algorithm was selected. If not then an error is given and the program exits.
- After that, the flags are checked (whether in set or not). Each algorithm is ran based on the arguments.
- If the program has insufficient or incorrect arguments (either encountered during or after parsing) the usage message is printed and the program returns with -1.

Pseudocode:

```
define flags
while (getopt returns valid)
    switch to the argument case if valid
        add the flag to Set
    if invalid print usage message and return
if -a flag
    print and run all sorts  and return
else check other flags (in the Set or not)
    run respective print function if flag encountered
```

Sorting Algorithms

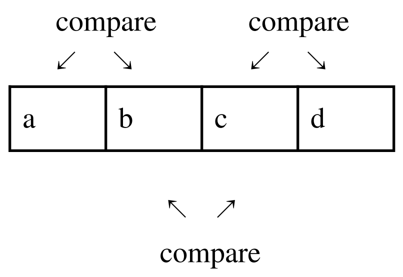
[NOTE: All credit for the pseudo code goes to the documentation uploaded to GitLab]

1. Bubble Sort
   - Bubble sort compares the value next to it and swaps based on the condition. This is done in a for loop until the array is sorted.
   - Calculating compares and moves: Incrementing compares and moves has a trivial solution. Each time before comparing with the previous element, increment by one (since the next line is always going to be executed no matter what) and if the "if" statement is true then add 3 to moves (swap takes 3 moves).

   Pseudocode: [CREDITS: Professor's Document]

```
bool = true
while bool:
    bool = false
    for every element in range (1, upper = length of arr) in array:
        if (curr element < previous element):
                swap them
                bool = true
        decrement upper
```

2. Shell Sort
    - Shell sort compares the value of the current element of the array with an element (current + gap) in the array.
    - If the element at current + gap position is less than the current element, shell sort finds a place before the current element for that current + gap element (by finding position, I mean until an element before the current one is lower than the current + gap element).
    - NOTE: The gap decreases on every loop and the array for gap elements is provided in the documentation.
    - Compares and moves:
        - Since in this implementation, we start from n = gap (till n < length of the array), we know that there are at least array length - gap comparisons are going to be made. For example, if current gap is 1 and total array length is 4, at least 3 comparisons would be made.



        - Inside the while loop, however, we know that a comparison would take place only if j >= gap value (because of short circuit analysis). Thus, if after j -= gap, the above mentioned condition is satisfied, we know that the second while loop condition (temp < A[j-gap]) will be executed , and thus we increment compare.
        - Moves would be incremented by 3 inside the while loop (swap takes 3 moves), and each time we have to backtrack with j.

Pseudocode: [CREDITS: Professor's Document]

```
for every gap in gap array:
    for every element starting from gap to end of the array:
        compare element with element at current-gap position
        swap if current is less
        keep comparing current-gap with element at gap position before
        that if possible (position of current-gap >= 0)
```

3. Quick Sort
    - Quick sort is a recursive algorithm that chooses a start and end point in an array and calls the helper partition function. It also chooses a pivot point.
    - The pivot point is returned by the partition function. The partition function chooses a pivot point and puts all the elements lower than the value at the pivot point at the left and all the values higher than the pivot point element on its right. The return value would be wherever the original pivot point ended up.
    - After getting the pivot point, quicksort pushes the starting and the ending points onto either stack or a queue for the next loop's partition call. (The values for the partition function are popped off the stack or the queue).
    - Compares and Moves:
        - This is also straightforward as bubble sort. The partition subroutine is the only function that performs array element comparisons and swapping and thus compares and moves are only updated there.
        - Before each while loop, compare is incremented (since next instruction would be always run), and inside each while loop, we increment compare (since if the condition is true, the while loop will run again).
        - Moves are incremented by 3 after the swap inside the last if statement (swap takes 3 moves).

Pseudocode: [CREDITS: Professor's Document on GitLab]

```
partition function:
    choose a pivot point
    while starting point < ending point:
        compare starting point value (low value) with pivot point value:
            if low val < pivot (no need to swap):
                increment starting point
        if high val > pivot (no need to swap):
                decrement ending point
        if those above conditions are not satisfied:
            swap the low and high values (to get everything less than
    pivot on
            its left and everything greater on its right)
    return new pivot point = current ending point

Quick sort:
    push starting and end points on the stack (initial 0 and last index of array)
    continue this process until stack is empty:
        call partition function with starting and end points (popped from
        stack or queue)
        if pivot position is greater than the low index:
            keep the previous starting point (push)
            make end point = pivot point (push on stack or queue)
        if pivot position is less than the high index:
            make start point = pivot point + 1 index (push)
            keep the previous ending point (push)
```

Abstract Data Types (ADTs):

1. Set
   - A set in this assignment is just a value (bit mask) that is used to keep track of which flags are passed.
   - Each bit corresponds to a flag. Upon encountering a flag, that bit is set to one to indicate the flag has been passed.
   - The element of the set structure is a mask (8 bit integer).
   - Set methods implementation:
     a. set_create()
        i. Allocate memory (dynamically) for set struct.
        ii. Create a pointer to set structure.
        iii. Initialize mask to 0.
        iv. Return pointer to newly created set (if possible).

     b. set_delete(pointer to set pointer)
        i. Free the memory of set structure (if present/possible).
        ii. Set the set pointer to null.

     c. is_member(set pointer, element)
        i. Return bitwise AND of the mask element of set struct with hex 1 left shifted by element (if possible).

     d. add_member(set pointer, element)
        i. Bitwise OR the mask element of the set struct with hex 1 left shifted by element (if possible).

2. Stack

- The stack ADT is implemented using a structure whose elements are: capacity, top, and items array.

- The top element is the index of the next empty element of the array. The capacity indicates how many elements the stack can hold. Last but not least, the items array is a dynamically allocated array for storing the stack elements.
- Stack methods implementation:
    a. stack_create(capacity)
        i. Allocate memory (dynamically) for stack struct.
        ii. Create a pointer to stack structure.
        iii. Initialize top element to 0, capacity to capacity.
        iv. Allocate memory (dynamically) for items array of size capacity.
        v. Return pointer to newly created stack.

    b. stack_delete(pointer to stack pointer)
        i. Free the memory of items array (if present/possible).
        ii. Free the memory of stack struct (if present/possible).
        iii. Make the pointer pointing to stack struct null.

    c. stack_empty(stack pointer)
        i. Return true if the top element is zero, otherwise false.

    d. stack_full(stack pointer)
        i. Return true if the top element is greater than the capacity of stack.

    e. stack_size(stack pointer)
        i. If the stack is full then return top element value - 1, else return top.

    f. stack_push(stack pointer, integer)
        i. If the stack is not full, assign the element in the items array at index top x’s value.

    g. stack_pop(stack pointer, integer pointer)

        i. If the stack is not empty, decrement the value of top and assign x (after dereferencing) value of element in the array at index top.

    h. stack_print(stack pointer)
        i. Start from the head, make a temporary variable to store top's value, and print items array till temporary >= 0 (decrementing temporary each time).

3. Queue

- The queue ADT is implemented in a similar fashion as the stack ADT. The elements of the queue structure are head, tail, size, capacity, and items array.
- The head element is the index of the head of the queue. The tail element is the index of the last used element of the array. The size element indicates total items in the queue. The capacity indicates how many elements the stack can hold. Last but not least, the items array is a dynamically allocated array for storing the queue elements.
- Queue methods implementation:
    a. queue_create(capacity)
        i. Allocate memory (dynamically) for queue struct.
        ii. Create a pointer to queue structure.
        iii. Initialize head, tail, and size element to 0, capacity to capacity.
        iv. Allocate memory (dynamically) for items array of size capacity.
        v. Return pointer to newly created queue (if possible).

    b. queue_delete(pointer to queue pointer)
        i. Free the memory of items array (if present/possible).
        ii. Free the memory of queue struct (if present/possible).
        iii. Make the pointer pointing to the queue struct null.

    c. queue_empty(queue pointer)
        i. Return true if the size element is zero, otherwise false.

d. queue_full(queue pointer)
    i. Return true if the size element is greater than the capacity of the queue.

e. queue_size(queue pointer)
    i. Return the size element.

f. enqueue(queue pointer, integer)
    i. If the queue is not full, assign the element in the items array at index tail integer's value.
    ii. Increment the tail (wrap around if necessary) and size elements.

g. dequeue(queue pointer, integer pointer)
    i. If the queue is not empty, assign x (after dereferencing) the value of the element in the array at index head.
    ii. Increment the head (wrap around if necessary) and size elements.

h. queue_print(queue pointer)
    i. Loop from head to tail and print each element of the items array in between.