Derick Pan
Dpan7

DESIGN.pdf

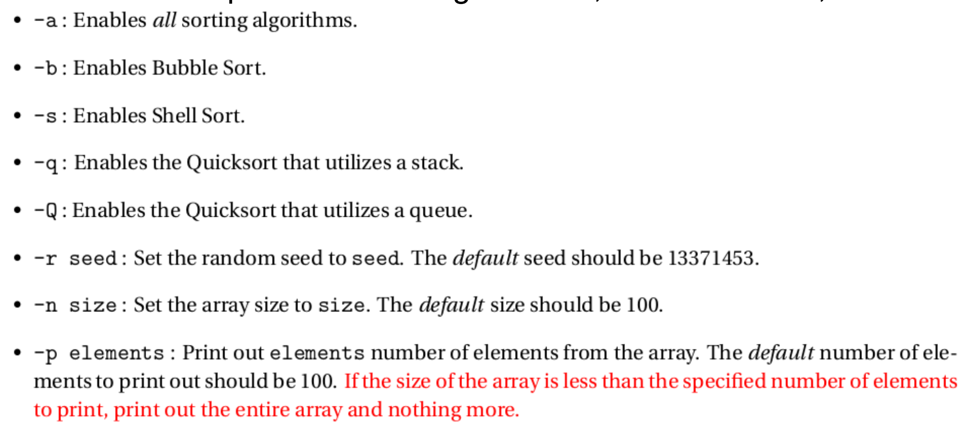
Prelab

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?
    a. Will need 6 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.
    a. The most amount of comparisons we could see in bubble sort is n(n+1)/2. So the worst time complexity possible would be O(n^2).

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

3. 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.
    a. To improve the time complexity we could change the Increment Sequences. Currently we are using gaps, predetermined gaps. Alternatively, we could use sequences like Hilbbard's or Knuth's. The time complexity for shell sort depends on the sequence of gaps because if we choose poor gaps then we might just get unlucky enough to completely miss the gap where all the swaps are. Sources I used for this answer are from https://www.codingeek.com/algorithms/shell-sort-algorithm-explanation-implementation-and-complexity/
4. Quicksort, with a worst case time complexity of O(n2), 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.
    a. Quicksort seems to be named quicksort for the less number of iterations and tests required to sort compared to other methods like bubble sort.
5. Explain how you plan on keeping track of the number of moves and comparisons since each sort will reside within its own file.
    a. To keep track of the moves and comparisons I thought of two methods. Either create a global variable in sorting.c, and reset the value to 0 before each sort method. OR Keep track of the moves within each of the sorts and print the moves and comparisons within the sorting methods.

In this lab, the summary of the problem is putting items into a sorted order without using the existing libraries. So I implement a testing harness, similar to lab 2, I will be looking for user arguments for what I'm looking for.



- `-a` : Enables *all* sorting algorithms.
- `-b` : Enables Bubble Sort.
- `-s` : Enables Shell Sort.
- `-q` : Enables the Quicksort that utilizes a stack.
- `-Q` : Enables the Quicksort that utilizes a queue.
- `-r seed` : Set the random seed to `seed`. The *default* seed should be 13371453.
- `-n size` : Set the array size to `size`. The *default* size should be 100.
- `-p elements` : Print out `elements` number of elements from the array. The *default* number of elements to print out should be 100. If the size of the array is less than the specified number of elements to print, print out the entire array and nothing more.

On the right there's 8 options for user input, and three of those options could take in a number for specifics.

Similar to Asgn2, I will be implementing a getopt() function. That will look something like the following

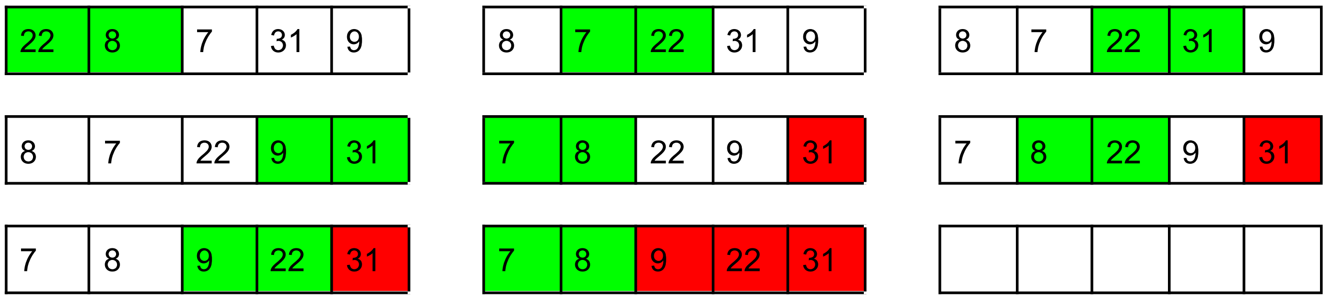
PSEUDOCODE TESTHARNESS

- Choice = getopt()
    - If choice is wrong then print program usage
    - If the choice is valid then set a corresponding flag to true
        - These flags will be called flagA, flagB, flagS, and so on…
        - Alternatively we create a set function that stores changes bits in order to signify what flags were flipped.
- Corresponding to which flags, call the sort function and print out the sorted array using a for loop.
    - Within those for loops we print the array in this manner printf ("%13" PRIu32);
    - So we could have organized printing.

Bubble Sort

Bubble sort works by examining adjacent pairs of terms. If j < i then swap them.

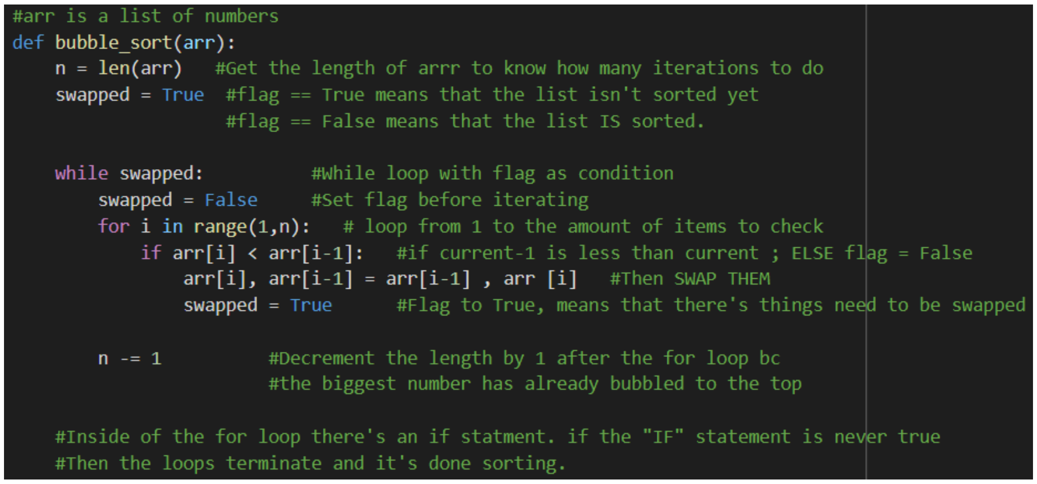
Example.



Now we know that all of these are in order, This is an example of how bubble sort works. It only took 7 Iterations to make this list in order. To check if the list is in order, just see if you made any swaps in the latest iteration.

To check how many passes need to be made, use this formula $n+(n-1)+(n-2)+\ldots+1=\frac{n(n+1)}{2}$

PseudoCode provided from the asgn3 doc. The comments show my understanding of the code but I will provide a flow chart



```
#arr is a list of numbers
def bubble_sort(arr):
    n = len(arr)   #Get the length of arrr to know how many iterations to do
    swapped = True  #flag == True means that the list isn't sorted yet
                    #flag == False means that the list IS sorted.

    while swapped:          #While loop with flag as condition
        swapped = False     #Set flag before iterating
        for i in range(1,n):   # loop from 1 to the amount of items to check
            if arr[i] < arr[i-1]:   #if current-1 is less than current ; ELSE flag = False
                arr[i], arr[i-1] = arr[i-1] , arr [i]   #Then SWAP THEM
                swapped = True      #Flag to True, means that there's things need to be swapped

        n -= 1          #Decrement the length by 1 after the for loop bc
                        #the biggest number has already bubbled to the top

    #Inside of the for loop there's an if statment. if the "IF" statement is never true
    #Then the loops terminate and it's done sorting.
```

Code provided by Professor Long
Comments Provided by me

My PsuedoCode

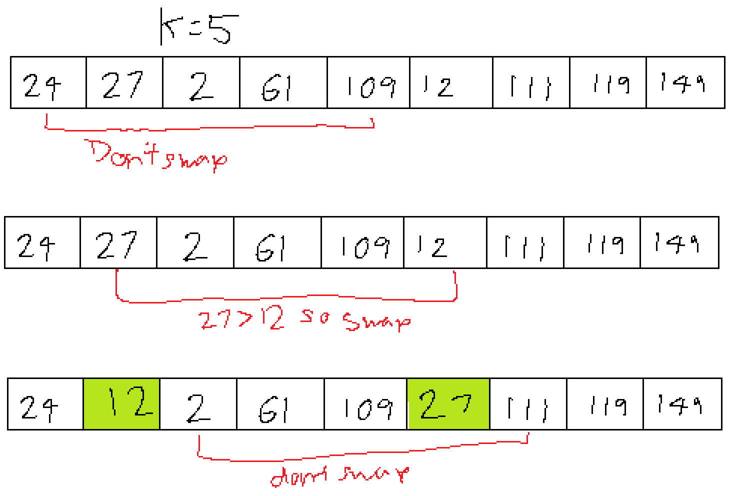
Imagine these bullet-points as functions and each smaller bullet-point are what's within the bigger bullet point

- Declare two variables and initialize them to 0. One for # of moves, one for # of compares
- While loop with condition of swapped:
    - Swapped will be set to false immediately, allowing this while loop to end.
    - Iterate from 1 to the length of the array (index)
        - If array(index-1) > array(index)
            - Swap them with each other.
            - Swapped = True //Swapped is now true, and the while loop will continue

- ○ Subtract 1 from the length of the array because we don't need to test the last value anymore as it's already in order.

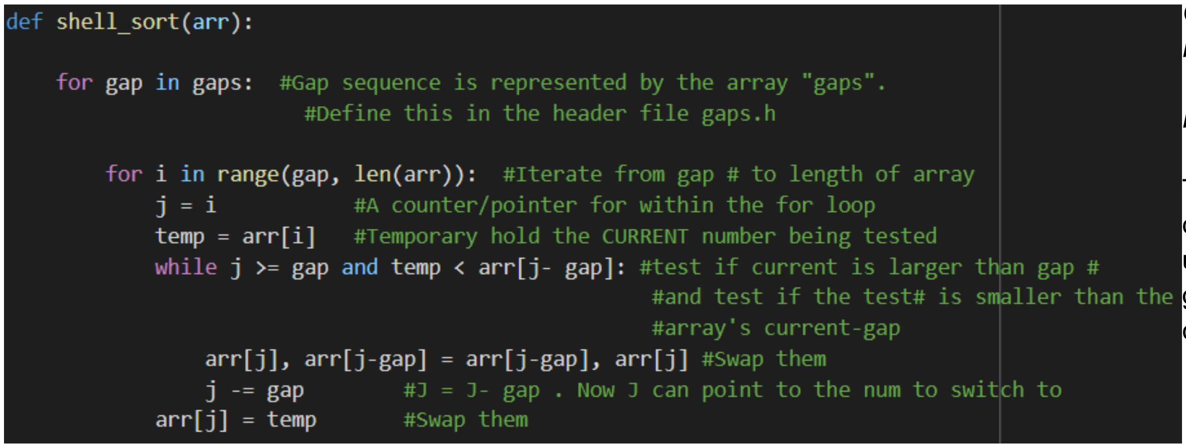
Shell Sort

Shell Sort works by sorting pairs of elements with a gap in between them.



Example.K = 5 because there are 9 elements, and 9/2 = 4.5, but we can't have a gap of 4.5, so i'm using 5 instead.

First you go one direction as seen in the example, then iterate to the front. Then divide your current k by half and do it again.

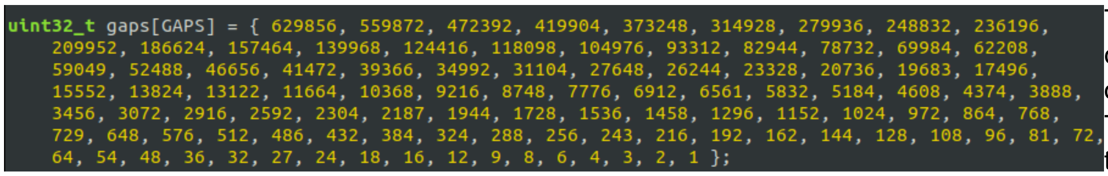


```
def shell_sort(arr):

    for gap in gaps:  #Gap sequence is represented by the array "gaps".
                       #Define this in the header file gaps.h

        for i in range(gap, len(arr)):  #Iterate from gap # to length of array
            j = i           #A counter/pointer for within the for loop
            temp = arr[i]   #Temporary hold the CURRENT number being tested
            while j >= gap and temp < arr[j- gap]: #test if current is larger than gap #
                                                    #and test if the test# is smaller than the
                                                    #array's current-gap
                arr[j], arr[j-gap] = arr[j-gap], arr[j] #Swap them
                j -= gap         #J = J- gap . Now J can point to the num to switch to
            arr[j] = temp        #Swap them
```

Code provided by Professor Long
Comments Provided by me.

The comments in this code show I understand what's going on but I will create my own

Gaps is an array of numbers that looks like this



```
uint32_t gaps[GAPS] = { 629856, 559872, 472392, 419904, 373248, 314928, 279936, 248832, 236196,
    209952, 186624, 157464, 139968, 124416, 118098, 104976, 93312, 82944, 78732, 69984, 62208,
    59049, 52488, 46656, 41472, 39366, 34992, 31104, 27648, 26244, 23328, 20736, 19683, 17496,
    15552, 13824, 13122, 11664, 10368, 9216, 8748, 7776, 6912, 6561, 5832, 5184, 4608, 4374, 3888,
    3456, 3072, 2916, 2592, 2304, 2187, 1944, 1728, 1536, 1458, 1296, 1152, 1024, 972, 864, 768,
    729, 648, 576, 512, 486, 432, 384, 324, 288, 256, 243, 216, 192, 162, 144, 128, 108, 96, 81, 72,
    64, 54, 48, 36, 32, 27, 24, 18, 16, 12, 9, 8, 6, 4, 3, 2, 1 };
```

The lists we have to sort could be 10 elements long, or 10,000 elements long. This list covers most to all of the bases we could sort with only uint32_t.

So why are we iterating over all of these numbers even though my list is only 10 digits long? Well, we're iterating over all of these numbers to narrow down which gap is right for me. Before any of these numbers are even attempted as a gap for my array. I need to test whether or not the gap could be used for my purpose. So, it's no big deal to use this array of preset gaps.

My Pseudocode

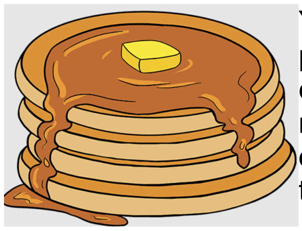
```
● Declare two variables and initialize them to 0. One for # of moves, one for # of compares.
● For loop that iterates through ALL of the gaps
    ○ For loop that starts from gap value and keeps running until the gap value is >= the length of the array we're sorting. Increment is gap val +=1. The for loop only runs when gap val is < length of array, which also doubles as the way to test gaps validity.
        ■ Tempval1 = gap;
        ■ Tempval2 = arr[gap];
        ■ Compares counter +1
        ■ While Loop: the temp val is greater than the gap and tempval2 < arr[tempval1-gap]. These two things are conditions for the while loop because both of those conditions need to be true to validate swapping items.
            ● Moves counter +3
            ● Swap A[tempval1] with A[tempval1-tempval2]
            ● Swap A[tempval1-tempval2] with A[tempval1] (this shouldn't equal to tempval1.
        ■ A[tempval1] = tempval2 //This is to update the current value after swapping.
```

Notes*

Shell sort makes more sense being used on bigger Arrays and lists rather than shorter ones, because the gaps are larger, and would overall just be easier. Though for the sake of assignment. Even short lists can be ordered using Shell sort.

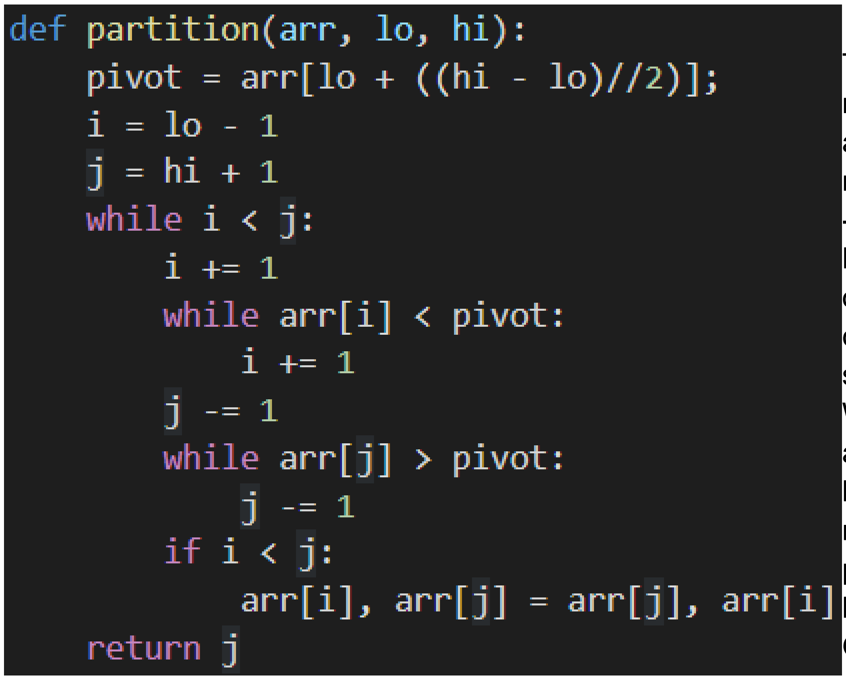
Quicksort

Quicksort for this assignment will be used in two ways. One that utilizes a stack and one that utilizes a queue. I already know what stack and queue means as should everyone else but here's a quick summary for what it is.

Stack:	Queue:
 You can add and take a pancake from the top. You can't take one out from the middle. Like a list, you can only add/remove an item from the top.	Imagine a line of people waiting in line to go to the store. If you want to add a person, they go to the END. If you want to remove a person, you remove the FRONT.

To implement Stack and Queue, I will be creating a function called partition that chooses a pivot (the middle num) then moving elements smaller than that to the left, bigger to the right.
Both quicksort functions will require an ADT or abstract data type, which means there will be manipulator functions.

Partition:



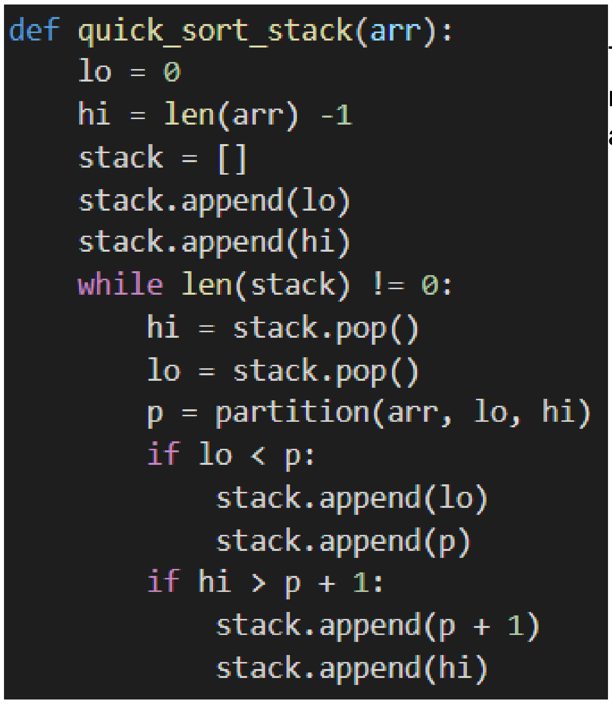
```
def partition(arr, lo, hi):
    pivot = arr[lo + ((hi - lo)//2)];
    i = lo - 1
    j = hi + 1
    while i < j:
        i += 1
        while arr[i] < pivot:
            i += 1
        j -= 1
        while arr[j] > pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    return j
```

This is the partition function that gets the pivot number then moves elements smaller to left, and bigger to the right. This function is necessary and useful for sorting lists.

PSUEDOCODE: This function partition is the only way for the stack and queue functions to communicate with the array we're needing to sort.
What partition receives as arguments are the array, and two elements being a smaller and bigger number. Then iterates over all the numbers in the array testing them against a pivot number. The pivot number is array[lo + hi-lo/2].
Once we have the pivot number, we test it against numbers smaller and bigger than it. During the testing, if i <j, we swap them. (This already puts two numbers in place.) Every pivot number we get, we test it against all other numbers which organizes the list incredibly efficiently.

Code provided by Professor Long
Comments and pseudo Provided by me

Stack:

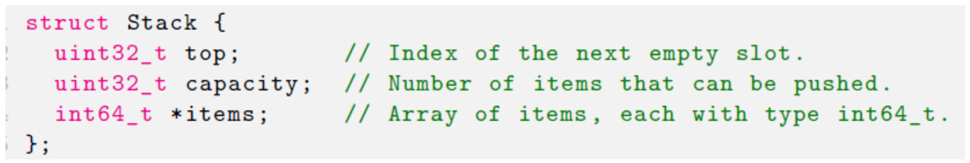


```python
def quick_sort_stack(arr):
    lo = 0
    hi = len(arr) -1
    stack = []
    stack.append(lo)
    stack.append(hi)
    while len(stack) != 0:
        hi = stack.pop()
        lo = stack.pop()
        p = partition(arr, lo, hi)
        if lo < p:
            stack.append(lo)
            stack.append(p)
        if hi > p + 1:
            stack.append(p + 1)
            stack.append(hi)
```

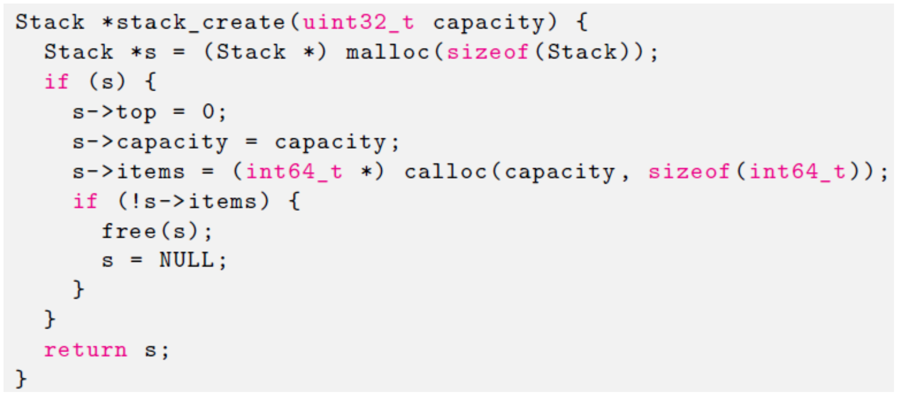
Code provided by Professor Long
Comments Provided by me

This stack function gets the top and bottom of the stack. Then we partition the array to get the lower and higher numbers. Then we compare the partitions to lower and higher and sort the numbers accordingly.

The stack function will include several helper functions: All of the python code as following is provided by Professor Long

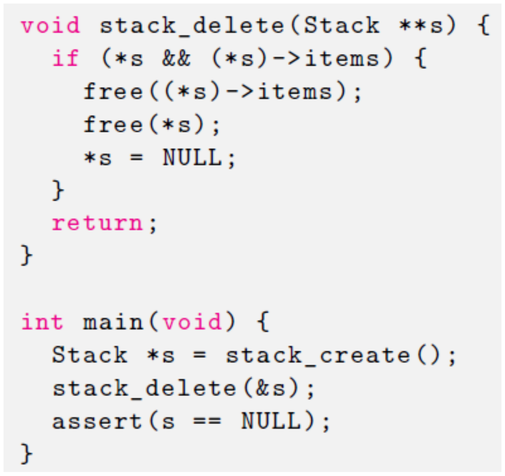
- Struct
    - This is for the following functions implementations
    - 

      ```c
      struct Stack {
        uint32_t top;        // Index of the next empty slot.
        uint32_t capacity;   // Number of items that can be pushed.
        int64_t *items;      // Array of items, each with type int64_t.
      };
      ```
    - Top represents the index of the next available position to add a value. Stacks work by adding and removing values from ONLY THE TOP of the list, and top keeps track of it.
    - Capacity is just a number that represents the MAXIMUM amount of values we could add into the stack. Capacity is important because there is no adding values bigger than the capacity because of segmentation fault.
    - *items, is the struct's way of holding all of the items. The reason for it being int64_t is because int64_t is capable of storing more variables. Somewhere up to the range of 5bill.
- Stack *stack_create(uint32_t capacity)
    - This is a constructor function. That says how much memory to allocate for the number of items.

- 

```c
Stack *stack_create(uint32_t capacity) {
  Stack *s = (Stack *) malloc(sizeof(Stack));
  if (s) {
    s->top = 0;
    s->capacity = capacity;
    s->items = (int64_t *) calloc(capacity, sizeof(int64_t));
    if (!s->items) {
      free(s);
      s = NULL;
    }
  }
  return s;
}
```

  Pseudo code from Professor Long
- Stack create does what the function name implies, it creates a stack that dynamically reallocates space within it. All positions in the stack without values are set to NULL so we won't accidentally reference them.
- The malloc, and calloc, are functions that allocate memory.

- void stack_delete(Stack **s)
  - This is a destructor function that frees memory no longer needed. This function is necessary because we don't want any memory leaks.
  - 

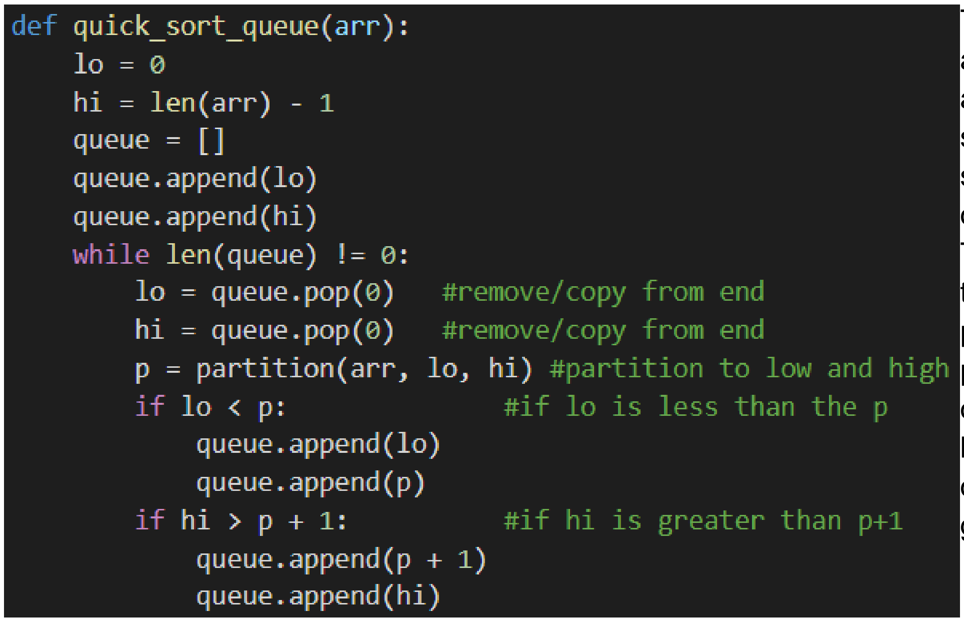
```c
void stack_delete(Stack **s) {
  if (*s && (*s)->items) {
    free((*s)->items);
    free(*s);
    *s = NULL;
  }
  return;
}

int main(void) {
  Stack *s = stack_create();
  stack_delete(&s);
  assert(s == NULL);
}
```

    Code from Professor Long,

  - Because we created an array with memory using the malloc and calloc functions, we need to free up the memory. If we don't free up the memory then crashes and memory leaks are prone to happen.
  - So stack delete frees up memory space and dereferencing pointers pointer.
- bool stack_empty(Stack *s)
  - This is an accessor Function. Return True if stack is empty, else false
  - We find this out by returning true if s->top ==0
    - Reason for this being, s->top represents the next available space, if it's zero then there's no value inside of the stack.
- bool stack_full(Stack *s)
  - Return True if stack full, else False

    - If s->top == s->capacity then return true, else false.
        - Reason for this being the top is the next available spot for data, and if the next spot for data is the max cap then we can't add anything.
- uint32_t stack_size(Stack *s)
    - Return # of items in stack
    - Return s->top
        - Reason for this being s->top is the next available space, and also the amount of things in the stack
- bool stack_push(Stack *s, int64_t x)
    - This is a manipulator function. This pushes item to the top of stack
    - s->items[s->top] =x;
    - s->top +=1
    - Return true
        - The reason for all of these being that s->items are the array of items, s->top is index.
        - s->top +=1 for next empty spot.
- bool stack_pop(Stack *s, int64_t *x)
    - Pop item from stack
    -
      ```
      // Dereferencing x to change the value it points to.
      *x = s->items[s->top];
      ```
    - That and s->top -=1
        - Bc we remove item
- void stack_print(Stack *s)
    - Debug function to print the array
    - We print the array.

Queue:

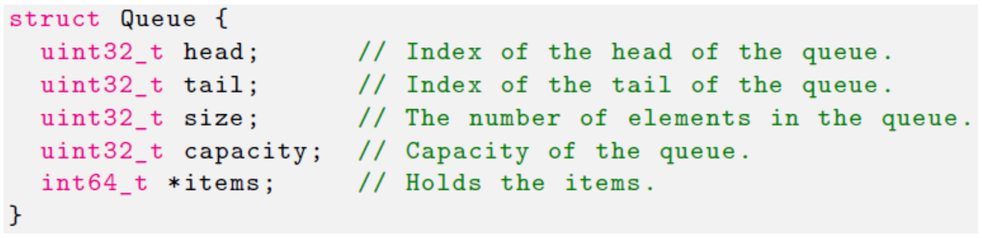


```python
def quick_sort_queue(arr):
    lo = 0
    hi = len(arr) - 1
    queue = []
    queue.append(lo)
    queue.append(hi)
    while len(queue) != 0:
        lo = queue.pop(0)   #remove/copy from end
        hi = queue.pop(0)   #remove/copy from end
        p = partition(arr, lo, hi) #partition to low and high
        if lo < p:              #if lo is less than the p
            queue.append(lo)
            queue.append(p)
        if hi > p + 1:          #if hi is greater than p+1
            queue.append(p + 1)
            queue.append(hi)
```

Code provided by Professor Long
Comments Provided by me

The queue for this problem works in a very specific way, that won't wrap around to the front. This queue is a straight line. The Queue function is similar to the Stack function in terms of variables, and method of testing. The significant difference between the two is the way queue takes in the popped variables. In queue, we will pop the lower val first, and higher one later. (Opposite of Stack.) This is because the queue is the polar opposite of the stack as shown in graphics above.

Necessary helper functions as follow: All of the python code as following is provided by Professor Long

- Struct
    - 

      ```c
      struct Queue {
        uint32_t head;      // Index of the head of the queue.
        uint32_t tail;      // Index of the tail of the queue.
        uint32_t size;      // The number of elements in the queue.
        uint32_t capacity;  // Capacity of the queue.
        int64_t *items;     // Holds the items.
      }
      ```
    - Tail represents the index of the next available position to add a value. Queue work by adding variables at the tail, and removing values from ONLY the head of the list, Tail represents the
    - Size represents the number of items in the queue. Kept track by adding and subtracting when enqueuing and dequeuing
    - Capacity is just a number that represents the MAXIMUM amount of values we could add into the queue. Capacity is important because there is no adding values bigger than the capacity because of segmentation fault.
    - *items, is the struct's way of holding all of the items. The reason for it being int64_t is because int64_t is capable of storing more variables. Somewhere up to the range of 5bill.
- Queue *queue_create(uint32_t capacity)
    - Queue create does what the function name implies, it creates a Queue that dynamically reallocates space within it. All positions in the Queue without values are set to NULL so we won't accidentally reference them.
    - The malloc, and calloc, are functions that allocate memory.
    - **SIMILAR TO STACK**
- void queue_delete(Queue **q)

    - Because we created an array with memory using the malloc and calloc functions, we need to free up the memory. If we don't free up the memory then crashes and memory leaks are prone to happen.
    - So queue_delete frees up memory space and dereferencing pointers pointer.
    - **SIMILAR TO STACK**
- bool queue_empty(Queue *q)
    - If q->size is 0 return true, else false
- bool queue_full(Queue *q)
    - If q->tail ==q->capacity+1 return true, else false
        - This is because the capacity is the max, and tail is the next spot to add something.
- uint32_t queue_size(Queue *q)
    - Return q->size
        - Yeah we've been keeping track.
- bool enqueue(Queue *q, int64_t x)
    - q->items[q->tail] = x
        - Set that val to x
    - q->tail +=1, q-> size +=1, return true
- bool dequeue(Queue *q, int64_t *x)
    - *x = q->items[q->head]
    - q->head +=1, q->size -=1
        - This is because the head goes up a value as we just removed the value, and size gets smaller. It dequeues because we dereference the pointer.
- void queue_print(Queue *q)
    - Print the array