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.

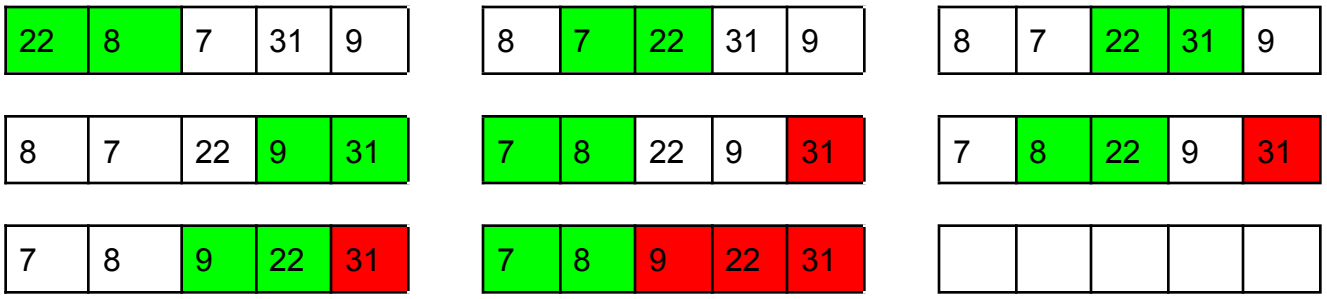
$$n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2}$$

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.
4. Quicksort, with a worse 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.
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.

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

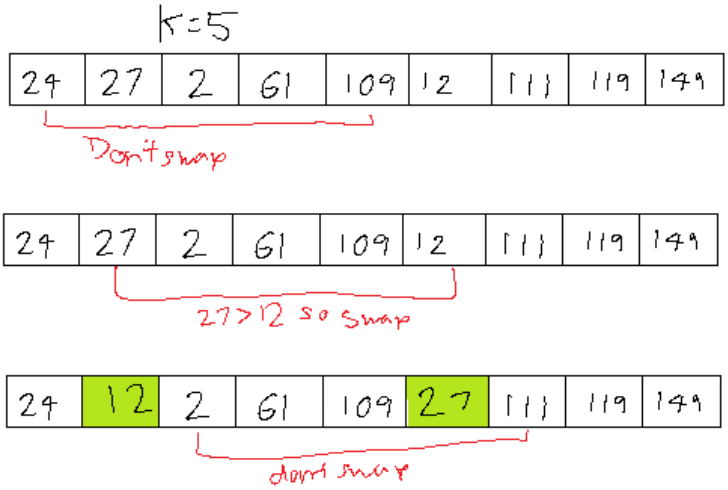
PseudoCode

```
#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

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.

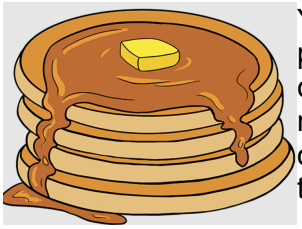
Psuedocode

```
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

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.

Code provided by Professor Long
Comments Provided by me

Stack:

```
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

```
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.
};
```

- Stack *stack_create(uint32_t capacity)
  - This is a constructor function. That says how much memory to allocate for the number of items.

```
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;
}
```

- 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.

```
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);
}
```

- bool stack_empty(Stack *s)
  - This is an accessor Function. Return True if stack is empty, else false
- bool stack_full(Stack *s)
  - Return True if stack full, else False
- uint32_t stack_size(Stack *s)
  - Return # of items in stack
- bool stack_push(Stack *s, int64_t x)
  - This is a manipulator function. This pushes item to the top of stack
- 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];
```

- void stack_print(Stack *s)
  - Debug function to 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)
```

The queue

Code provided by Professor Long
Comments Provided by me

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.
      }
      ```
- Queue *queue_create(uint32_t capacity)
- void queue_delete(Queue **q)
- bool queue_empty(Queue *q)
- bool queue_full(Queue *q)
- uint32_t queue_size(Queue *q)
- bool enqueue(Queue *q, int64_t x)
- bool dequeue(Queue *q, int64_t *x)
- void queue_print(Queue *q)
      ```