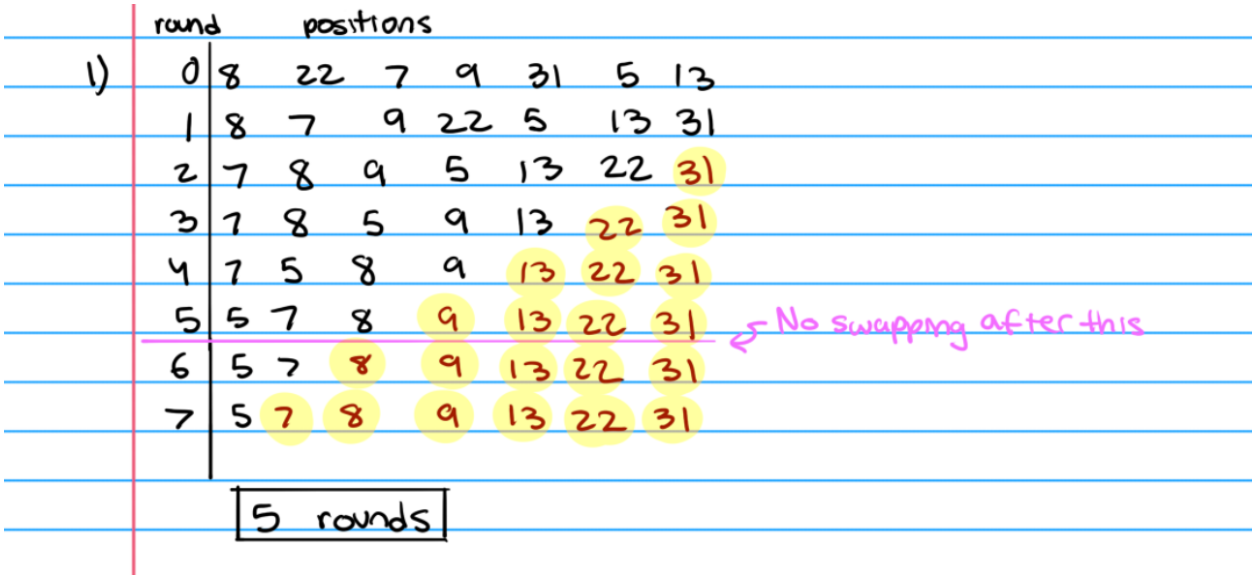
Pre-lab Part 1

1. How many rounds of swapping do you think you will need to sort the numbers 8, 22, 7, 9, 31, 5, 13 in ascending order using Bubble Sort?
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.
3. How would you revise the algorithm so the smallest element floats to the top instead?

1.



2. Worst case is if all the values are in reverse order. I was able to see this while swapping the last array as the '5' took very long to get to its position from one end to the other.
3. To reverse this, I would flip the '<' sign on line 7 to be '>'

Bubble Sort in Python

```
1  def bubble_sort(arr):
2      n = len(arr)
3      swapped = True
4      while swapped:
5          swapped = False
6          for i in range(1, n):
7              if arr[i] < arr[i - 1]:
8                  arr[i], arr[i - 1] = arr[i - 1], arr[i]
9                  swapped = True
10         n -= 1
```

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.
2. How would you improve the runtime of this sort without changing the gap sequence?

1. Gap sequences are important bc they determine the time complexity of sorting an array with shell sort. The thing about gap sequences is that they can change the complexity of an array very drastically. The best case for shell sort is going through an already sorted array with a gap of 1.[1] However, if you apply that same gap on a very randomly sorted array, it'll take way longer to sort everything.
   The best gap sequence I found was from "The online Encyclopedia of Integer Sequences"[2]. The gap sequence is "1, 2, 3, 4, 6, 8, 9, 12, 16, 18,..." and they used smooth numbers to create it
2. #2 was removed from the assignment

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.

1. According to geeks for geeks[3], "This happens when input array is sorted or reverse sorted and either first or last element is picked as pivot." Because these are very unlikely scenarios, quick sort should work well most of the time.

Pre-lab Part 4

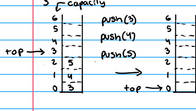
1. Explain how you plan on keeping track of the number of moves and comparisons since each sort will reside within its own file.

1. I will create a separate file to keep these values inside of. I'm thinking of making a c source file with accessible global variables.

[1] https://www2.cs.duke.edu/courses/fall01/cps100/notes/sorting_cheat.txt
[2] https://oeis.org/A003586
[3] https://www.geeksforgeeks.org/can-quicksort-implemented-onlogn-worst-case-time-complexity/#:~:text=The%20worst%20case%20time%20complexity,element%20is%20picked%20as%20pivot.

Stack

```
struct Stack {
    u32 top
    u32 capacity
    int64 * items
}
```




```
Stack *Stack-create(void)
    Stack*s = (Stack *)calloc(1,sizeof(Stack))
    s->top =0
    s->capacity = MIN_CAPACITY //16

    s-> items = (items *) calloc(s->capacity, sizeof(int64))
}

bool stack-empty(Stack *s):
    return s->top ==0; <- true if stack is empty

bool stack-push(Stack *s,
    if(s->top == s->capacity) {
        s->capacity *= 2;
        s->items = realloc(s->items,s->capacity * sizeof(int64_t))
    }
    s->items[s->top] =x <- puts a val at the top slot in the array
    s->top+=1 <- increments top
}

bool Stack_pop(S ,*x) {
    s->top -= 1
    * x = s->s->items[s->top]
}
```

Size in bytes to reallocate
ptr to original memory
remember to return false if realloc fails
8 bytes
remember to check if stack is empty

Sets

U8 → 8 items

```
typedef uint8_t  set;

enum sorts{
    bubble =0
    shell =1
    quick =2
    heap =3
}
```

empty set = MSB 0000 0000 LSB
add (bubble) 7654 3210
set with bubble = MSB 0000 0001 LSB
add(shell) 7654 3210
set w/bubble & shell = MSB 0000 0011 LSB

```
add (bubble)
    set s = MSB 0000 0000 LSB
    mask = 1 <<bubble
    s = s|mask
```

Inside getopt()

```
while (getopt()...)
    switch(opt)
        case 'b'
            sorts =set_insert(bubble)
        case 'h'
            sorts =set_insert(heap)

    for(sorts s = bubble
```

Marian Zlateva
mzlateva@ucsc.edu
2/13/2021

CSE13s Fall 2020
Assignment 5: Sorting

Description:

In this lab, I am implementing different sorting algorithms and investigating their complexities and how they work.

Flags:

-a:	Employs all sorting algorithms.
-b:	Enables Bubble Sort.
-s	Enables Shell Sort.
-q	Enables Quicksort.
-h	Enables Heapsort.
-r	seed : Set the random seed to seed. The default seed should be 7092016.
-n	size : Set the array size to size. The default size should be 100.
-p	number of elements to print from the array

Files:

- bubble.h
  - h specifies the interface to bubble.c.
- bubble.c
  - implements Bubble Sort.
- gaps.h
  - contains the Pratt gap sequence for Shell Sort
- shell.h
  - specifies the interface to shell.c.
- shell.c
  - implements Shell Sort.
- quick.h
  - specifies the interface to quick.c
- quick.c
  - implements Quicksort.
- stack.h
  - specifies the interface to the stack ADT
- stack.c
  - implements the stack ADT.
- heap.h
  - specifies the interface to heap.c
- heap.c
  - implements Heap Sort.
- set.h
  - specifies the interface to the set ADT
- set.c
  - implements the set ADT
- sorting.c
  - contains main()
  - implements the set

Functions:

sorting.c

int main()

- ➢

- ➔ parses command line options using getopt()
  - ➞ uses the SET adt to specify if a sort is activated
- ➔ initializes the array and stores it in the heap where it has more space
- ➔ make sure there is enough space for the array to fit in memory
- ➔ fixes print_count to not be more than the array size
- ➔ iterate through sort types
  - ➞ check if current sort is activated
    - ➝ resets array so that it is not sorted
    - ➝ resets the counters
    - ➝ performs specified sort
    - ➝ prints sorted array

void print_array(uint32_t *A, int print_count)

- ➢ Prints the array

- ➔ iterates through the array
  - ➞ prints the spot
  - ➞ if the current column is the size of PRINT_COLS or the printing is done, then break line

void fill_rand(uint32_t *A,int size, uint64_t seed)

- ➢ Fills array with pseudo random numbers

- ➔ reinitializes the random seed to that we generate the same array each time
  - ➞ iterates through the array
    - ➝ bitmasks the random number to 30 bits and sets it to the spot

bubble.c

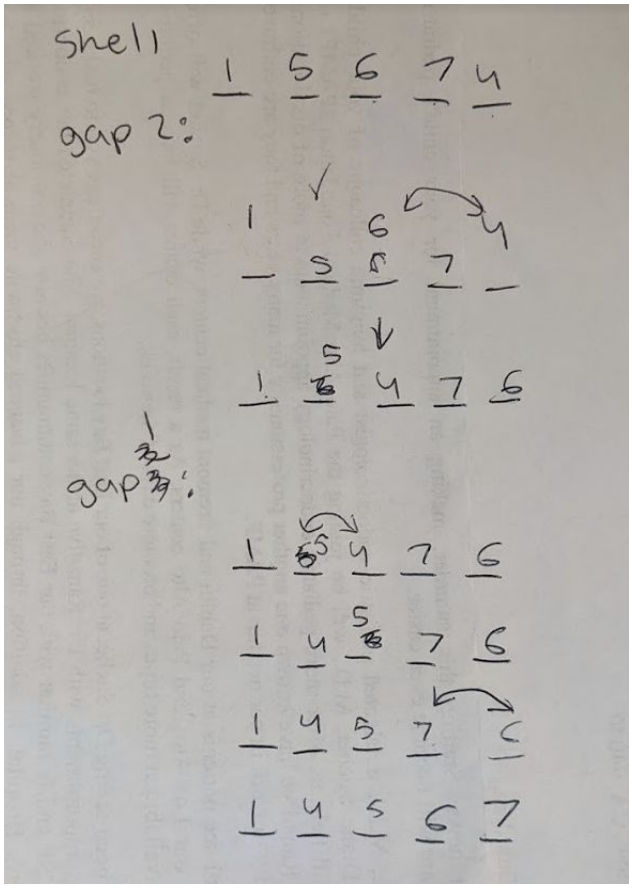
bubble sort example is on first page of the doc

void bubble_sort(uint32_t *A, uint32_t n)

- ➢ performs bubble sort

- ➔ go through the array until there is an iteration with no swapping
  - ➞ go through the array n times
    - ➟ if the current value is smaller than the val behind it, swap them
  - ➞ n = n -1

shell.c



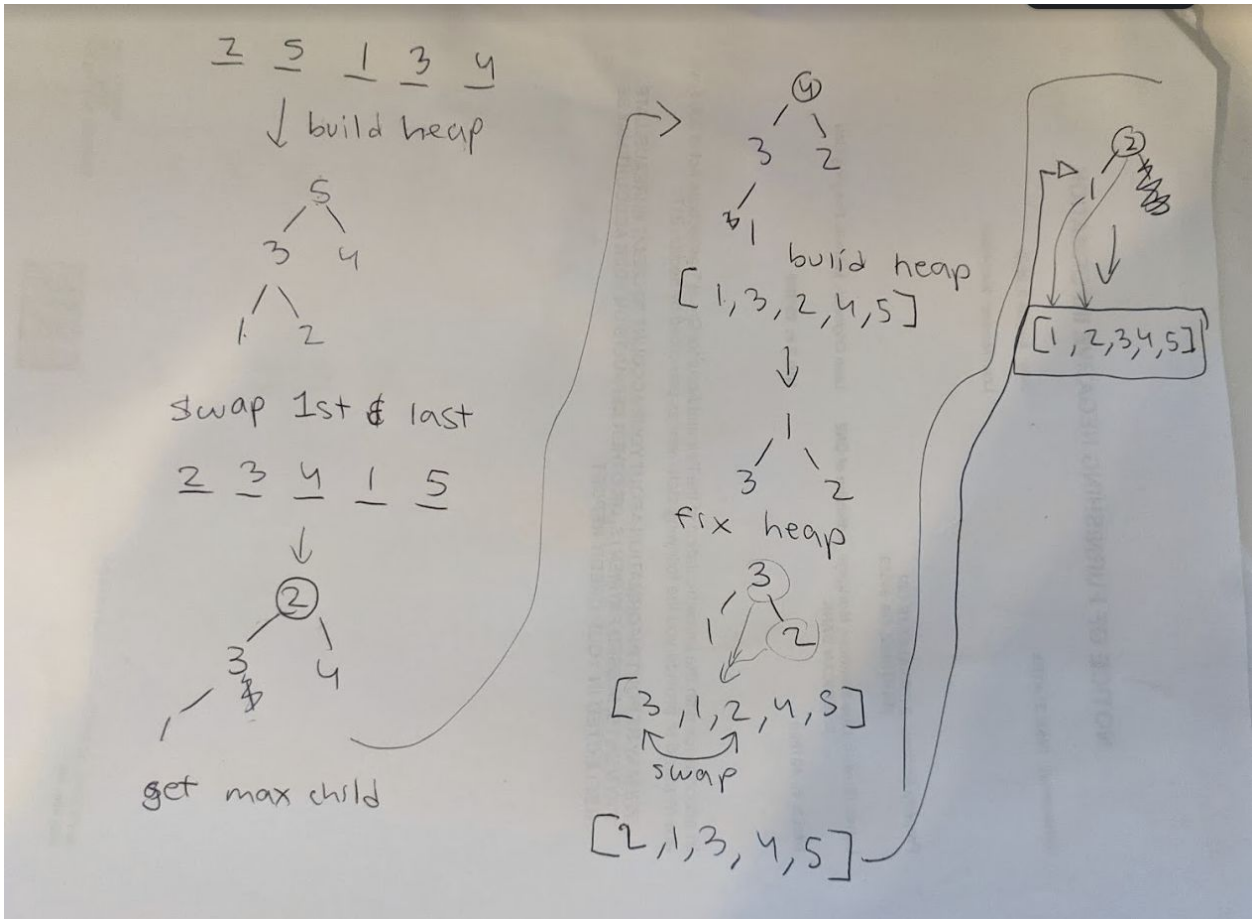
void bubble_sort(uint32_t *A, uint32_t n)

- ➢ performs bubble sort

- ➔ iterates through the gap sizes
  - ➞ iterates through the array
    - ➟ keep swapping the indexes with temp as the base until temp > than the val behind it
      - ➞ temp is the base number we are using to compare

- ➞ swaps temp with val behind it
- ➞ temp is now in index j-gap, so we need to make the base index into j-gap

heap.c



void heap_sort(uint32_t *A, uint32_t n)

➢ performs heap sort

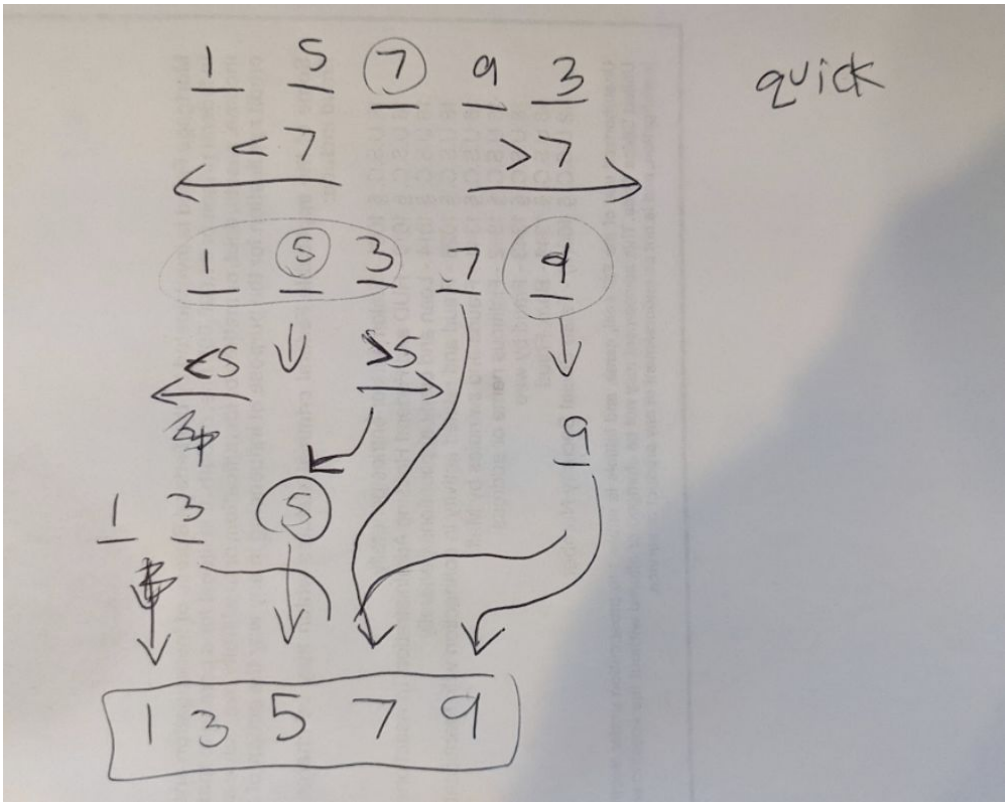
- ➔ calls build_heap() to fix array arrangment
- ➔ sorts the array by extracting elements from the heap
- ➔ for(int leaf=last; leaf>first; leaf--)
  - ➞ move root A[0] to the correct spot in the array A[leaf-1]
  - ➞ fix the heap so that it is arranged correctly

int max_child(uint32_t *A,int first, int last)

➢ gets the bigger child

- ➔ if the right child is larger than the other child, return the right child as the max_child

quick.c



void quick_sort(uint32_t *A, uint32_t n)

- performs quick sort

- pushes lowest and highest indexes in array to stack
- until there are no more partitions to split, create more and sort them
  - take the upper and lower bounds for the partition out of the array
  - sort array inside range and get a new partition
  - if the range above the partition is greater than 1, add it to the stack to that we can sort it later
  - if the range below the partition is greater than 1, add it to the stack to that we can sort it later

int64_t partition(uint32_t *A,uint32_t lo,uint32_t hi)

- creates a partition

- select midpoint of lo and hi as pivot
- variables l and r will be fixed to equal lo and hi inside the loop
- must increment l & r each iteration in case A[l] == A[r] == pivot
- find the first value from the left that's greater than the pivot

- ➔ find the first value from the right that's less than the pivot
- ➔ flip the values if they are on the wrong sides of the pivot
  - ➞ if all values to the right of the pivot are greater than the pivot, r <p
  - ➞ if all values to the left of the pivot are less than the pivot, l >p so r<l if nothing needs swapping
- ➔ return the index of where the numbers greater than or equal to the partition start
- ➔ everything might not all be sorted around the index of the pivot but we return r so that we get a division between numbers > or < the partition
- ➔ return r

set.c

Set set_empty(void)

- ➢ return an empty set (0)

- ➔ return 0

bool set_member(Set s, uint8_t x)

- ➢ determines if a spot in the Set is a 0 or 1

- ➔ gets the intersection of the set and an int with a 1 shifted x spots in any number that isn't 0 will return true

Set set_insert(Set s, uint8_t x)

- ➢ sets the value of a spot in the Set to 1

- ➔ gets the union of the set and an int with a 1 shifted x spots in

Set set_remove(Set s, uint8_t x)

- ➢ sets the value of a spot in the Set to 0

- ➔ gets the intersection of a set with all 1s except at spot x and the Set

Set set_intersect(Set s, Set t)

- ➢ gets a set containing the elements that are common to both sets (the intersection)

- ➔ return s & t;

Set set_union(Set s, Set t)

- gets a set containing all the elements in both sets (the union)

- return s | t;

Set set_complement(Set s)

- gets the complement of a set

- return ~(s)

Set set_difference(Set s, Set t)

- gets the elements of set s which are not in set t (the difference)

- return s & (~t)

set.c

Stack *stack_create(void){

- creates a stack pointer for an empty stack

- set top to 0
- set capacity to default (16)
- allocates space for 16 64-bit integers

void stack_delete(Stack **s)

- free's the stack and sets the pointer to NULL

- delete everything in s and set s to null

bool stack_push(Stack *s, int64_t x)

- pushes a value to the top of the stack

- checks if stack is full
  - doubles the capacity
  - reallocates memory
- puts a val in the top slot in the array
- makes top point at next avalible position

bool stack_pop(Stack *s, int64_t *x)

- ➢ pops a value off the stack

- ➔ if stack is empty, return false
- ➔ makes top point to the highest filled slot
- ➔ sets the value x pts to, to the thing in the top slot

void stack_print(Stack *s)

- ➢ prints the stack

- ➔ iterate through the stack and print

sorting_tools.c

st_swap(uint32_t *A, int index1, int index2)

- ➢ swappes 2 indexes in an array

- ➔ temp = A[index1]
- ➔ A[index1] = A[index2]
- ➔ A[index2] = temp

bool st_is_smaller(int A, int B)

- ➢ return true if A<B

- ➔ return - true if A<B