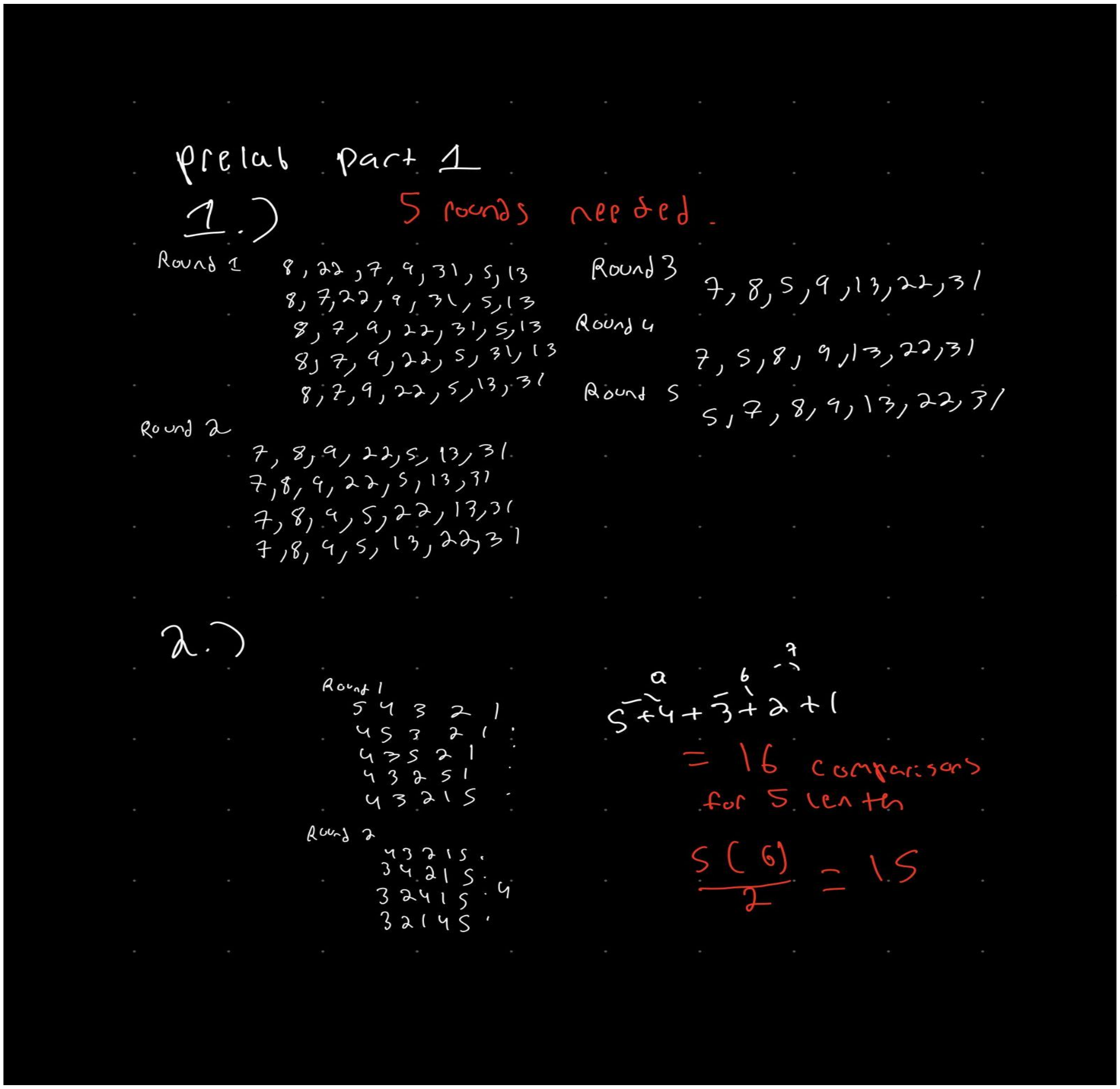
Mattiwos Belachew
mbelache@ucsc.edu
4/25/2021

CSE 13s Spring 2021
Assignment 3:
Sorting: Putting your affairs in order
Design Document

Prelab:

Prelab Part 1:



prelab part 1

1.) 5 rounds needed.

Round 1
8, 22, 7, 9, 31, 5, 13
8, 7, 22, 9, 31, 5, 13
8, 7, 9, 22, 31, 5, 13
8, 7, 9, 22, 5, 31, 13
8, 7, 9, 22, 5, 13, 31

Round 2
7, 8, 9, 22, 5, 13, 31
7, 8, 9, 22, 5, 13, 31
7, 8, 9, 5, 22, 13, 31
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

2.)

Round 1
5 4 3 2 1
4 5 3 2 1
4 3 5 2 1
4 3 2 5 1
4 3 2 1 5

Round 2
4 3 2 1 5
3 4 2 1 5
3 2 4 1 5 4
3 2 1 4 5

a 6 7
$5+4+3+2+1$

= 16 comparisons for 5 lenth

$\frac{5(6)}{2} = 15$

Prelab Part 2:

1. It depends on the sequence of gaps because the gaps are used in order to quickly move out-of-place elements into easier positions which is faster than a normal exchange with the nearest neighbor like Bubble sort. By getting rid or out-of-place values you are able to spend less time sorting the list in the future since you don't need to make as many comparisons or moves in order to get it to its correct position. (https://en.wikipedia.org/wiki/Shellsort)

Prelab Part 3:

1. The reason quicksort isn’t doomed is because it's very efficient on avg, cache-friendly, and it's inplace so it doesn’t use extra memory except the constant amount. In addition, quicksort has smaller constant factors than other algorithms. (https://cs.stackexchange.com/questions/3/why-is-quicksort-better-than-other-sorting-algorithms-in-practice)

Prelab Part 4:

1. In order to keep track of all the moves and comparisons I plan on using static variables which I will be able to manipulate in the sort functions. After, I will set it to 0 and run the next sorting algorithm.

Description:

_____I will be making many sorting algorithms such as Shell Sort, Bubble Sort, and Quick sort using stack and queues in order to implement the algorithm. In addition, I will be creating a stack and a queue used for quick sort. The reason why is because sorting items into a sorted order is one of the most common tasks in Computer Science and its important to understand how its done in order to make wise choices choosing algorithms for future programs.

Files:

- Stats.c
  - Contains function to count number of moves and compares
- Stats.h
  - Contains prototype for stats.c.
- Set.c
  - Contains functions for sets
- Set.h
  - Specifies the interface for set.c
- Queue.c
  - Implements the queue ADT
- Queue.h
  - Specifies the interface to the queue ADT
- Stack.c

    - Implements the stack ADT.
- Stack.h
    - Specifies the interface to the stack ADT
- Quick.c
    - Implements both stacks and queues quick sorts
- Quick.h
    - Specifies the interface to quick.c
- Shell.c
    - Implements shell sorts
- Shell.h
    - Contains the specifications for the interface for shell.c
- Gaps.h
    - Contains the Pratt gap sequence for Shell Sort.
- Bubble.c
    - Contains function to run bubble sort algorithm on an Array of n length.
- Bubble.h
    - Prototype function for bubble sort function required to implement it.
- Sorting.c
    - Contains a test harness to test sorting algorithms and compare moves and look at a number of comparisons.
- WRITEUP.pdf
    - It contains a discussion of the results for my tests. Specifically, analysis of the differences in the output of my implementations vs. math.h library.
- Makefile
    - Builds, cleans and debugs the program, sorting.c.
- DESIGN.pdf
    - The design document describes purpose, covers the layout, clear description of program parts, and pseudocode.
- README.md
    - Instruction page for building and running the program.

TOP LEVEL:

Main:

```
Main:
        Opt = 0
        Options = "absqQr:n:p:"
        Seed = 13371453
        Elements = 100 //default;
```

```
Size = 100 //num printed
S = {}
while  ((opt = getopt(argc, argv, OPTIONS)) != -1) { //program argument parser.
        Switch (opt) {
        case 'r':
           seed = argument;
           break;
        case 'p':
           size = argument;
           break;
        case 'n':
           elements = argument;
           break;
        case 'a':
           s = set_insert(s, BUBBLE_SORT);
           s = set_insert(s, SHELL_SORT);
           s = set_insert(s, QUICK_SORT_STACK);
           s = set_insert(s, QUICK_SORT_QUEUE);
           break;
        case 'b':
           s = set_insert(s, BUBBLE_SORT);
           break;
        case 's':
           s = set_insert(s,SHELL_SORT);
           break;
        case 'q':
           s = set_insert(s,QUICK_SORT_STACK);
           break;
        case 'Q':
           s = set_insert(s,QUICK_SORT_QUEUE);
           break;

        }
        A[elements] = []*0

        Loop through  i in s{
                srandom(seed)
                Loop through A (array) {
                A[index] = random();
                }

                Switch (i){
                case BUBBLE_SORT:
```

```
                printf("Bubble Sort\n");
                bubble_sort(A, elements);
                print_stats(elements);
                break;
            case SHELL_SORT:
                printf("Shell Sort\n");
                shell_sort(A, elements);
                print_stats(elements);
            case QUICK_SORT_STACK:
                printf("Quick Sort (Stack)\n");
                quick_sort_queue(A, elements);
                print_stats(elements);
            case QUICK_SORT_QUEUE:
                printf("Quick Sort (Queue)\n");
                quick_sort_queue(A, elements);
                print_stats(elements);


            }
            Loop through array{
            printf(A[index])
            }
        }

        Return 0
    }
```

Stack:

```
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) {
    if ((*s)->items) {
        free((*s)->items);
        free((*s));
        *s = NULL;
    }
    return;
}
bool stack_full(Stack *s) {
    if (s->top - 1 == s->capacity) {
        return true;
    }
    return false;
}

bool stack_empty(Stack *s) {
    if (s->top == 0) {
        return true;
    }
    return false;
}

stack_size(Stack *s) {
    return s->top;
}

bool stack_push(Stack *s, int64_t x) {
    if (!stack_full(s)) {
        s->items[s->top] = x;
        s->top += 1;
        return true;
    }
    return false;
}

bool stack_pop(Stack *s, int64_t *x) {
    if (s && !stack_empty(s)) {
        s->top -= 1;
        *x = s->items[s->top];
        return true;
    }
    return false;
}

void stack_print(Stack *s) {
    for (uint32_t i = 0; i < s->top; i++) {
        printf("%ld \t", s->items[i]);
    }
    printf("\n");
}
```

Collecting_statistics:

```
moves = 0;
compares = 0;
max_stack = 0;
max_queue = 0;

get_moves() {
    return moves;
}
get_compares() {
    return compares;
}

set_zero_moves() {
    moves = 0;
}
set_zero_compares() {
    compares = 0;
}
stats_zeros() {
    set_zero_compares();
    set_zero_moves();
    max_queue = 0;
    max_stack = 0;
}
```

```
moved() {
    moves += 1;
    return;
}
compared() {
    compares += 1;
    return;
}
compare(x) {
    compares += 1;
    return x;
}
count_stack(x) {
    max_stack = x;
    return;
}
count_queue(x) {
    max_queue = x;
}
print_stack_size(){
    printf("Max Stack:", max_stack);

}
print_queue_size(){
    print("Max Queue:", max_queue);

}

print_stats(e) {
    print( e, moves, compares);

}
```

- Creates a global variable which stay constant and functions which can be called to manipulate the numbers;

Design Process:

1. One problem I have right now is running a quick sort. For some reason it currently throws a segfault error and I am unable to fix it so far but it seems that the index for the pivot for partition is really high which leads to problems. In addition, I don’t believe that the problem is because of my stack implementation.

2. I was able to fix it. It had to do with my parameters for partition being srinked which causes problems.
3.

What I learned:

1. I learned how to use stacks and queues and implement them in sorting algorithms.
2. I learned how to test my program using valgrind in order to test for memory leaks.
3. How to graph my data.

Resources Used:

- Asgn3.pdf
- asgn5Design by William Santosa from Piazza
- Example_design.pdf found in CSE 13s Discord.