Shweta Jones
CSE 13S - Winter 2021
Prof Darrell Long

Assignment 3 Design - Sorting: Putting Your Affairs in Order

Design:

The program executes different sorting algorithms: Bubblesort, Shell sort, Quicksort that is implemented with a stack and queue. This allows students to familiarize themselves with different sorting algorithms and what works best when. The main program is run with sorting.c file which is run on terminal.

PreLab Questions:

PreLab 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?**

    Original Set: 8, 22, 7, 9, 31, 5, 13

    Round 1:

    8, 22, 7, 9, 31, 5, 13 → 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, 31, 5, 13 → 8, 7, 9, 22, 5, 31, 13 → 8, 7, 9, 22, 5, 13, 31

    Round 2:

    8, 7, 9, 22, 5, 13, 31 → 7, 8, 9, 22, 5, 13, 31 → 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, 9, 5, 13, 22, 31 → 7, 8, 9, 5, 13, 22, 31 → 7, 8, 9, 5, 13, 22, 31 → 7, 8, 5, 9, 13, 22, 31 → 7, 8, 5, 9, 13, 22, 31

    Round 4

    7, 8, 5, 9, 13, 22, 31 → 7, 8, 5, 9, 13, 22, 31 → 7, 5, 8, 9, 13, 22, 31 → 7, 5, 8, 9, 13, 22, 31

    Round 5:

    7, 5, 8, 9, 13, 22, 31 → 5, 7, 8, 9, 13, 22, 31 → 5, 7, 8, 9, 13, 22, 31

    Round 6:

    5, 7, 8, 9, 13, 22, 31 → 5, 7, 8, 9, 13, 22, 31

NOTE: RED stands for a comparison BLUE stands for a number that I don't need to check

    Number of Swaps: 10

    Number of Comparisons: 21

2. **How many comparisons can we expect to see in the worst-case scenario for Bubble Sort?**
    Hint: make a list of numbers and attempt to sort them using Bubble Sort.

    Original set: 4, 21, 6, 11, 17, 22, 26, 30

    Round 1:

    4, 21, 6, 11, 17, 22, 26 → 21, 4, 6, 11, 17, 22, 26 → 21, 6, 4, 11, 17, 22, 26 → 21, 6, 11, 4, 17, 22, 26 → 21, 6, 11, 17, 4, 22, 26 → 21, 6, 11, 17, 22, 4, 26 → 21, 6, 11, 17, 22, 26, 4

Round 2:

21, 6, 11, 17, 22, 26, 4 → 21, 6, 11, 17, 22, 26, 4 → 21, 11, 6, 17, 22, 26, 4 → 21, 11, 17, 6, 22, 26, 4 → 21, 11, 17, 22, 6, 26, 4 → 21, 11, 17, 22, 26, 6, 4

Round 3:

21, 11, 17, 22, 26, 6, 4 → 21, 11, 17, 22, 26, 6, 4 → 21, 17, 11, 22, 26, 6, 4 → 21, 17, 22, 11, 26, 6, 4 → 21, 17, 22, 26, 11, 6, 4

Round 4:

21, 17, 22, 26, 11, 6, 4 → 21, 17, 22, 26, 11, 6, 4 → 21, 22, 17, 26, 11, 6, 4 → 21, 22, 26, 17, 11, 6, 4

Round 5:

21, 22, 26, 17, 11, 6, 4 → 22, 21, 26, 17, 11, 6, 4 → 22, 26, 21, 17, 11, 6, 4

Round 6:

22, 26, 21, 17, 11, 6, 4 → 26, 22, 21, 17, 11, 6, 4

NOTE: RED stands for a comparison BLUE stands for a number that I don't need to check

Number of Swaps: 18

Number of Comparisons: 21

PreLab 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.**

   Having the perfect number of gaps and better gap sizes can allow for shell sort to be more efficient and improve the time complexity for Shell Sort. If these gap sizes are too large, then it would take a large amount of time since insertion sort would have to do a lot of work. However, in the case of numerous smaller gap sizes, the comparisons would take a lot of time, making shell sort extremely inefficient. In order to improve time complexity, these gaps have to be a sufficient size and these gaps should be spaced out in a balanced way.

   Sources:
   https://www.tutorialspoint.com/data_structures_algorithms/shell_sort_algorithm.htm,
   https://en.wikipedia.org/wiki/Shellsort#:~:text=Shellsort%20is%20an%20optimization%20of,said%20to%20be%20h%2Dsorted,
   https://www.programiz.com/dsa/shell-sort

PreLab 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.**

   The worse case for quicksort is when the pivot for the sort is an "extreme value" so either a very higher number or a very low number. Since quicksort has a "divide and conquer" approach, it makes even a "worse-case scenario" be comparable to a "best-case scenario". For quicksort, the best-case scenario is a

situation where the partitions are split equally, however in a worse case time complexity, these partitions are not evenly balanced. Since the values are switched about pivot positions, it doesn't make the worst-case scenario as bad as it can be, especially when compared to other sorts.
Sources:https://www.geeksforgeeks.org/quick-sort/ ,
https://www.interviewbit.com/tutorial/quicksort-algorithm/

PreLab 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.**

   In order to keep track of the number of moves and comparisons for each sort, I would have a global variable that is accessible between the different functions. This can allow for these values to be printed.

Files

bubble.c: Performs bubble sort
shell.c: Performs shell sort
quick.c: Performs quick sort
stack.c: Performs heap sort with stack
queue.c: Performs heap sort with queue
sorting.c: The main function that uses the user's inputs to run different sorting algorithms
counts.c: Calculates the number of comparisons and moves

PseudoCode:

Sorting.c:

<u>Create Array Function</u>

```
for (int r=0; r<length; r++)
   A[r] = random()
```

<u>Prints Statistics</u>

```
printf("%s\n", sortName)
printf("%d elements, %d moves, %d compares\n", length, sCount, cCount)
if (elements> 0)
   int i = 0
      while (i<elements)
         printf("%13" PRIu32, A[i])
         if ((i+1)%5 == 0)
            printf("\n")
         i++
      if (i%5 > 0)
         printf("\n")
```

```
enum Sorting{
   bubble,
   shell,
   stack,
   queue,
```

```
}Sorting;
Main:
    Set s = set_empty()
    int opt = 0
    int seed = 13371453
    int length = 100
    int elements = 100
    while ((opt = getopt(argc, argv, OPTIONS)) != -1)
        switch (opt)....
    if (elements > length)
        elements = length
    void (*sortCalls[4])(uint32_t *, uint32_t)
    sortCalls[0] = bubble_sort
    sortCalls[1] = shell_sort
    sortCalls[2] = quick_sort_stack
    sortCalls[3] = quick_sort_queue
    char*sorts[4]={ sets names... }
    uint32_t *A = (uint32_t *) malloc (length * sizeof(uint32_t))
    int i = bubble
    while (i <= queue)
        if (set_member(s, i))
                cCount = 0
                sCount = 0
                srandom(seed)
                createArray(A, length)
                sortCalls[i](A, length)
                stats(sorts[i], elements, length, A)
            i++
    free(A)
    return 0
```

Bubble.c

```
uint32_t ival = n-1
uint32_t i = 0
while (i<ival)
    uint32_t j = i+1
        while(j<n)
            if (COMP(&A[j], &A[i]) == true)
                    SWAP(&A[j], &A[i])
            j++
        i++
```

Shell.c

```
int i = 0
while (i<GAPS)
    uint32_t gap = gaps[i]
```

```
for (uint32_t k = gap; k<n; k++)
    uint32_t j = k
    uint32_t m = A[k]
    while (j >= gap && COMP(&m, &A[j - gap]) == true)
        SWAP(&A[j], &A[j - gap])
            j -= gap
    A[j] = m
i++
```

Quick.c

<u>Partition Code:</u>

```
uint32_t pivot = A[lo + ((hi-lo)/2)]
int64_t b = lo-1
int64_t c = hi+1
while (b < c)
    b++
        while(COMP(&A[b], &pivot) == true)
            b++

        c--
        while(COMP(&pivot, &A[c]) == true)
            c--
        if (b<c)
            SWAP(&A[b], &A[c])
return c
```

<u>Quick Sort Stack Code:</u>

```
int64_t hi = 0
int64_t lo = n-1
Stack *Q = stack_create(n)
if (stack_push(Q, hi) == true && stack_push(Q, lo) == true)
    while (stack_empty(Q) != true)
            if (stack_pop(Q, &hi) == true && stack_pop(Q, &lo) == true)
                int64_t p = partition(A, lo, hi)
                    if (lo < p)
                        stack_push(Q, lo)
                        stack_push(Q, p)
                    if (hi > p+1)
                        stack_push(Q, (p+1))
                        stack_push(Q, hi)
        stack_delete(&Q)
```

<u>Quick Code Queue Code:</u>

```
int64_t hi = 0
int64_t lo = n-1
Queue *Q = queue_create(n)
```

```
    if(enqueue(Q, hi) == true && enqueue(Q, lo) == true)
        while(queue_full(Q) == true)
                if(dequeue(Q,&lo) == true && dequeue(Q,&hi) == true)
                    int64_t p = partition(A, lo, hi)
                        if (lo<p)
                            enqueue(Q, lo)
                            enqueue(Q, p)
                        if (hi > p+1)
                            enqueue(Q, (p+1))
                            enqueue(Q, hi)
            queue_delete(&Q)
```

Stack.c:

```
struct Stack {
    uint32_t top
    uint32_t capacity
    int64_t *items
}
Stack Create
    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
Stack Delete
    if (*s && (*s)->items)
        free((*s)->items)
            free(*s)
            *s = NULL
    return
Stack Empty
    return s->top == 0
Stack Full
    return s->top == 1
Stack Size
    return s->top
Stack Push
    if (s->top == s->capacity)
        s->capacity = 2 * s-> capacity
            s->items = (int64_t *) realloc (s->items, s->capacity * sizeof(int64_t))
```

```
        if (s->items == NULL)
            return false
    s->items[s->top] = x
    s->top += 1
    return true
Stack Pop
    if (s->top == 0)
        return false
    s->top -= 1
    *x = s->items[s->top]
    return true
Stack Print
    for (uint32_t p = 0; p<s->top; p++)
        printf("%ld ", s->items[s->top])
    printf("\n")
```

Queue.c:

```
struct Queue {
    uint32_t head
    uint32_t tail
    uint32_t size
    uint32_t capacity
    int64_t *items
}
Create Queue
    Queue *q = (Queue *) malloc (sizeof(Queue))
    if(q)
        q->head = 0
            q->tail = 0
            q->size = 0
            q->capacity = capacity
            q->items = (int64_t *) calloc (capacity, sizeof(int64_t))
            if (!q->items)
                free(q)
                q = NULL
    return q
Delete Queue
    if (*q && (*q)->items)
        free((*q)->items)
            free(*q)
            *q = NULL
    return
Empty Queue
    if (q->size == 0)
```

return true
else
return false

Queue Full

if (q->size == q->capacity)
return true
else
return false

Queue Size

return q->size

Enqueue

if(q->size == q->capacity)
return false
q->size +=1
q->items[q->tail] = x
q->tail=(q->tail+1)%q->capacity
return true

Dequeue

if(q->size == 0)
return false
q->size -=1
*x = q->items[q->head]
q->head = (q->head+1)%q->capacity
return true

Print Queue

for (uint32_t i=q->head; i<=q->tail;i++)
printf("%ld ", q->items[i])

Counts.c

uint32_t cCount = 0
uint32_t sCount = 0

Comparison Function:

cCount+=1
if(*i<*j == true)
return true
else
return false

Swap Function

int temp
temp = *i
*i = *j
*j = temp
sCount += 3