Design Document - Assignment 3

Hsiang Yun Lu

February 6, 2023

1 Purpose of Programs

We will implement four different sorting programs and test their performance by writing a test harness using bit-wise Set operations. The four sorting methods are Shell sort, Heap sort, Quick sort and Batcher sort, respectively. The Set operation implementing functions and statistics module will be separate files that can be used in the main `sorting.c` testing program.

2 Files To Be Included in Directory *asgn3*

- `shell.c` and `shell.h`
  - The `C` program contains implementation of Shell sort. The `shell.h` is the header file that specifies the interface to `shell.c`.
- `heap.c` and `heap.h`
  - The `C` program contains implementation of Heap sort. The `heap.h` is the header file that specifies the interface to `heap.c`.
- `quick.c` and `quick.h`
  - The `C` program contains implementation of recursive Quick sort. The `quick.h` is the header file that specifies the interface to `quick.c`.
- `batcher.c` and `batcher.h`
  - The `C` program contains implementation of Batcher merge sort. The `batcher.h` is the header file that specifies the interface to `batcher.c`.
- `gaps.h`
  - This file contains a gap sequence and its length to be used by Shell sort.
- `set.c` and `set.h`

  - The `C` program contains implementation of bit-wise Set operations. The `set.h` is the header file that specifies the interface to `set.c`.

- `stats.c` and `stats.h`
  - The `C` program contains implementation of statistics module. The `stats.h` is the header file that specifies the interface to `stats.c`.

- `sorting.c`
  - This file contains `main()` and is the test harness for the four sorting implementation and their statistics.

- `Makefile`
  - This file that compiles the sorting programs and builds the `sorting` executable

- `README.md`
  - This file describes how to use all the programs and `Makefile`, including explanations on command-line options

- `DESIGN.pdf`
  - This file describes the design and design process for all the programs
  - Include pseudocode

- `WRITEUP.pdf`
  - Findings and things learnt from different sorting algorithms
  - Include the graphs that displays the performance of the sorts on a variety of inputs
  - Include analysis and explanations for the graphs

3 Design/Structure

(1) `shell.c`

```
SET stats moves = 0
SET stats compares = 0

function shell_sort(stat, array, length)
    FOR k = 0 to gaps_len
        FOR i = gaps[k] to length
            j ← i
            temp ← array[i]
```

```
            WHILE j >= gaps[k] and (arr[j - gaps[k]] > temp)
                compares += 1
                array[j] ← array[j - gaps[k]];
                moves += 1
                j -= gaps[k]
            END WHILE
            array[j] ← temp
            moves += 1
        END FOR
    END FOR
END function
```

(2) `heap.c`

```
SET stats moves = 0
SET stats compares = 0

function max_child(stat, array, first, last)
    left ← 2 × first
    right ← left + 1
    IF right <= last and (array[right - 1] > array[left - 1])
        compares += 1
        RETURN right
    END IF
    RETURN left
END function


function fix_heap(stat, array, first, last)
    found ← FALSE
    mother ← first
    great ← max_child(stat, array, mother , last)
    WHILE mother <= last / 2 and not found
        IF array[great - 1] > array[mother - 1]
            compares += 1
            SWAP array[great - 1] and array[mother - 1]
            moves += 3
            mother ← great
            great ← max_child(stat, array, mother, last)
        ELSE
            found ← TRUE
```

```
        END IF
    END WHILE
END function


function build_heap(stat, array, first, last)
    FOR i = last / 2 to (first - 1) with each step = -1
        fix_heap(stat, array, i, last)
    END FOR
END function


function heap_sort(stat, array, length)
    first ← 1
    last ← length
    build_heap(stat, array, first, last)
    FOR i = last to first with each step = -1
        SWAP array[first - 1] and array[i - 1]
        moves += 3
        fix_heap(stat, array, first, i - 1)
    END FOR
END function
```

(3) `quick.c`

```
SET stats moves = 0
SET stats compares = 0

function partition(stat, array, lo, hi)
    i ← lo - 1
    FOR j = lo to hi
        IF array[hi - 1] > array[j - 1]
            compares += 1
            i += 1              SWAP array[i - 1] and array[j - 1]
            moves += 3
        END IF
    END FOR
    SWAP array[i] and array[hi - 1]
    RETURN i + 1
END function
```

```
function quick_sorter(stat, array, lo, hi)
    IF lo < hi
        p ← partition(stat, array, lo, hi)
        quick_sorter(stat, array, lo, p - 1)
        quick_sorter(stat, array, p + 1, hi)
    END IF
END function


function quick_sort(stat, array, length)
    quick_sorter(stat, array, 1, length)
END function
```

(4) `batcher.c`

```
SET stats moves = 0
SET stats compares = 0

function comparator(stat, array, x, y)
    IF array[x] > array[y]
        compares += 1
        SWAP array[x] and array[y]
        moves += 3
    END IF
END function


function batcher_sort(stat, array, length)
    IF n == 0
        RETURN
    END IF

    t ← bit length of length
    p ← 1 « (t - 1)
    WHILE p > 0
        q ← 1 « (t - 1)
        r ← 0
        d ← p
        WHILE d > 0
            FOR i = 0 to n - d
```

```
                IF (i AND p) == r
                    comparator(stat, array, i, i + d)
                END IF
            END FOR
            d ← q - p
            q » ← 1
            r ← p
        END WHILE
        p » ← 1
    END WHILE
END function
```

(5) `set.c`

```
function set_empty()
    RETURN 0
END function


function set_universal()
    RETURN NOT 0
END function


function set_insert(Set s, x)
    RETURN s OR 1 « x
END function


function set_remove(Set s, x)
    RETURN s AND NOT(1 « x)
END function


function set_member(Set s, x)
    IF (s AND (1 « x)) not 0
        RETURN TRUE
    ELSE
        RETURN FALSE
    END IF
END function
```

```
function set_union(Set s, Set t)
    RETURN s OR t
END function


function set_intersect(Set s, Set t)
    RETURN s AND t
END function


function set_difference(Set s, Set t)
    RETURN s AND (NOT t)
END function


function set_complement(Set s)
    RETURN NOT s
END function
```

(6) `sorting.c`

```
DEFINE command-line options “Hasbhqn:p:r:”

DEFINE enum type for [SHELL, QUICK, HEAP, BATCHER, HELP, END] as SORTS
function get_randarr(array, length, seed)
    SET random seed
    SET mask ← NOT ((1 « 31) OR (1 « 30))
    FOR i = 0 to length
        array[i] ← random number
        array[i] ← array[i] AND mask
    END FOR
END function


function usage(program)
    print(program synopsis and usage)
END function
```

```
function main(arguments)
    SET function pointer array func_ptr = [shell_sort, quick_sort, heap_sort, batcher_sort]
    SET opt ← 0
    SET vector ← 0
    SET len ← 100
    SET elements ← 100
    SET seed ← 13371453
    INITIATE Stats struct array arr_stats = [ 0, 0 ,  0, 0 ,  0, 0 ,  0, 0 ]
    SET string array arr_name = [“Shell Sort”, “Quick Sort”, “Heap Sort”, “Batcher Sort”]

    IF no input argument
        PRINT “Select at least one sort to perform.”
        usage()
    END IF

    WHILE here is at least an argument
        SWITCH argument
            case argument “s” do insert SHELL to vector
                break
            case argument “q” do insert QUICK to vector
                break
            case argument “h” do insert HEAP to vector
                break
            case argument “b” do insert BATCHER to vector
                break
            case argument “a” do insert SHELL, QUICK, HEAP, BATCHER to vector
                break
            case argument “H” do insert HELP to vector
                break
            case argument “n” do len ← number converted from read-in string
                break
            case argument “p” do elements ← number converted from read-in string
                break
            case argument “r” do seed ← number converted from read-in string
                break
            default do usage()
                return EXIT_FAILURE
        END SWITCH
    END WHILE

    SET *Array ← calloc(len, sizeof(32 bits))
```

```
WHILE TRUE
    IF HELP is in vector
        usage()
        break
    ELSE IF SHELL, QUICK, HEAP, BATCHER are all in vector
        FOR s = SHELL to BATCHER
            get_randarr(Array, len, seed)
            function pointer array func_ptr[s](Stats struct array arr_stats[s], Array, len)
            PRINT sorting method, array length, number of moves, number of compares
            IF elements >= len
                FOR i = 0 to len
                    IF (i + 1) is not multiple of 5
                        IF i == len - 1
                            PRINT Array[i] + newline
                        ELSE
                            PRINT Array[i]
                        END IF
                    ELSE
                        PRINT Array[i] + newline
                    END IF
                END FOR
            ELSE IF elements < len
                FOR i = 0 to elements
                    IF (i + 1) is not multiple of 5
                        IF i == elements - 1
                            PRINT Array[i] + newline
                        ELSE
                            PRINT Array[i]
                        END IF
                    ELSE
                        PRINT Array[i] + newline
                    END IF
                END FOR
            END IF
        END FOR
        break
    ELSE
        FOR s = SHELL to BATCHER
            IF s in vector
                get_randarr(Array, len, seed)
                function pointer array func_ptr[s](Stats struct array arr_stats[s], Array, len)
                PRINT sorting method, array length, number of moves, number of compares
```

```
                IF elements >= len
                    FOR i = 0 to len
                        IF (i + 1) is not multiple of 5
                            IF i == len - 1
                                PRINT Array[i] + newline
                            ELSE
                                PRINT Array[i]
                            END IF
                        ELSE
                            PRINT Array[i] + newline
                        END IF
                    END FOR
                ELSE IF elements < len
                    FOR i = 0 to elements
                        IF (i + 1) is not multiple of 5
                            IF i == elements - 1
                                PRINT Array[i] + newline
                            ELSE
                                PRINT Array[i]
                            END IF
                        ELSE
                            PRINT Array[i] + newline
                        END IF
                    END FOR
                END IF
            END IF
        END FOR
        break
    END IF
END WHILE
free Array memory
RETURN 0
End function
```

4 Credit

- Omar's section on Feb. 1st.
- Assignment 1 `monte_carlo.c`