Assignment 4 – Sets and Sorting

Sebastian Avila

CSE 13S – Fall 2023

Purpose

The purpose of this program is to implement a set of *Set* functions and a set of Sorting algorithms. These sorting algorithms are heap sort, shell sort, quick sort, insert sort, and batcher sort.

How to Use the Program

To use this program you will need to have several files in the same directory. The files are `set.c\,`, `set.h`, `insert.c`, `insert.h`, , `shell.c`, `shell.h`, `heap.c`, `heap.h`, `quick.c`, `quick.h`, `batcher.c`, `batcher.h`, `sorting.c`, `stats.c`, `stats.h`, and `Makefile`. With these files in the same directory, use command `make clean` to remove any preexisting object files. Then, use command `make format` to format all the header and source files. Then the same directory use command `make` to compile the program. Then run the program using `./sorting` with a command line argument. The possible command line arguments and their meanings are as follows:

- `-H` : Display program help and usage message
- `-a` : Enable all sorting algorithms
- `-h` : Enable Heap sort
- `-b` : Enable Batcher sort
- `-s` : Enable Shell sort
- `-q` : Enable Quick sort
- `-i` : Enable Insertion sort
- `-n length` : Set the array size to `length`. If this argument is missing, the default length is 100.
- `-p elements` : Print out `elements` number of elements from the array. If this agrument is missing, the default value is 100. If the array size is less than `elements` the entire array will be printed out and nothing more. If `elements` is set to 0, nothing will be printed.
- `-r seed` : Set the random seed to `seed`. If this argument is missing, the default seed is 13371453.

The program uses several optional compiler flags:

- `-Wall`: This flag enables all warning messages.
- `-Werror`: This flag turns all warnings into errors.
- `-Wextra`: This flag enables extra warning flags that are not enabled by `-Wall`.
- `-Wstrict-prototypes`: This flag warns if a function is declared or defined without specifying the argument types.
- `-pedantic`: This flag issues all the warnings demanded by strict ISO C and ISO C++.
- `-lm`: This flag links the math.h library. This allows the program to access and use the functions from the math.h library.

Program Design

The program starts by initializing a set with 8 bits all set to 0. This set represents a set of sorting algorithms. It then designates each type of sorting as a certain bit. It also initializes three variable, `elements`, `seed`, `size`, and sets them to their default values. It then checks the command line arguments. If an incorrect argument is received, an error message and a usage message is printed. The arguments pertaining to the sort options add the corresponding bit to the set when the argument is received. The other three arguments take in a string and covert it into an integer then store that integer in a variable that was initialized before. After the command arguments are dealt with, it initializes a variable `stats` of type Stats, creates a variable with a hexadecimal value of 0x3fffffff. That is 32 bits, with the lower 30 bits set to 1. This variable is used to mask the random numbers generated into only the lower 30 bits. Memory is then allocated using the size of the first element of the array by the size of the array. Next, a series of if statements checks whether each sort is a member of the set. If it is, the stats variable and the array is reset. The array is then sorted, the stats are printed, then the elements of the array are printed. Once all the if statements have been checked, the program frees the memory previously allocated and then returns 0.

Algorithms

All algorithms in this section were given in the assignment instructions. [1] Psuedocode for the algorithms is shown in the function description section of this report.

- Insertion Sort : This algorithm assumes the first element is already sorted. It then moves to the next element and compares it to the first. If it is less than the first element it swaps the elements. It repeats this process for the remaining elements, each time assuming the previous element is already sorted.
- Shell Sort : This is a version of insertion sort. Shell sort repeatedly sorts sub arrays that are separated by a gap that is reduced continuously. After the gap is 1, the array is completely sorted.
- Heap Sort : The first step in the heap sort algorithm is to create a max heap from the array that is to be sorted. A max heap array is a binary tree where each node is greater than or equal to its children. Once the array is transformed into a max heap, the root of the heap is swapped with the last element in the array and the heap size is reduced by 1. Next, the heap is fixed to maintain the max heap structure. The last two steps are repeated until the entire array is sorted.
- Quick Sort : Quick sort works by choosing a pivot element and partitioning the other elements into two sub arrays according to whether they are less than or greater than the pivot. The sub arrays are then sorted recursively.
- Batcher Sort : Batcher sort k-sorts the even and odd sub-sequences of an array, where k is a power of 2, and then merges them.

Function Descriptions

Sorting

- `void init_array(seed, size, array, mask)` : This function takes 4 parameters: three integers `seed`, `size`, `mask`, and an integer array `array`. It does not return anything. This function resets the array to the same random unsorted values that it was before being sorted.

```
init_array(seed, size, *array, mask)
    srandom(seed)
    for i, 0 to i < size
        array[i] = random() AND mask
```

- `void print_elements(*array, size)` : This function takes in two parameters: an integer array and an integer representing its size. It does not return anything. Its purpose is to print the elements of the array in 5 columns with each element having a width of 13.

```
print_elements(*array, size)
    for i, 0 to i < size
        print array[i] with width of 13
        if ((i + 1) % 5) = 0) then
            print newline
```

Sets

- `Set set_empty(void)` : This function takes no parameters and returns a set. It purpose is to return an empty set. This is a set where all bits are equal to 0.

```
Set set_empty(void)
        Set s = 00000000 (0x00)
        return s
```

- `Set set_universal(void)` : This function takes no parameters and returns a set. Its purpose is to return a set in which every possible number is part of the set.

```
Set set_universal(void)
    Set u = 11111111 (0xff)
    return u
```

- `Set set_insert(Set s, int x)` : This function takes two parameters: a set `s` and an integer `x`. It returns a set. Its purpose is to insert the integer `x` into the set `s` and return the set.

```
Set set_insert(Set s, int x)
    s = s OR (0x01 << x)
    return s
```

- `Set set_remove(Set s, int x)` : This function takes two parameters: a set `s` and an integer `x`. It returns a set. Its purpose is to remove the integer `x` from the set `s` and return the set.

```
Set set_remove(Set s, int x)
    s = s AND set_complement(0x01 << x)
    return s
```

- `bool set_member(Set s, int x)` : This function takes two parameters: a set `s` and an integer `x`. It returns a bool (true or false) indicating whether the integer `x` is within the set `s`.

```
bool set_member(Set s, int x)
    if (s AND (0x01 << x) == 1 then
        return true
    return false
```

- `Set set_union(Set s, Set t)` : This function takes two parameters of type set: `s` and `t`. It returns a set containing only the bits corresponding to members that are equal to 1 in either `s` or `t`.

```
Set set_union(Set s, Set t)|
    s = s OR t
    return s
```

- `Set set_intersect(Set s, Set t)`— : This function takes two parameters of type set: `s` and `t`. It returns a set containing only the bits corresponding to members that are equal to 1 in both `s` and `t`.

```
Set set_intersect(Set s, Set t)|
    s = s AND t
    return set
```

- `Set set_difference(Set s, Set t)` : This function takes two parameters of type set: `s` and `t`. It returns a set containing only the elements of `s` that are not in `t`.

```
Set set_difference(Set s, Set t)|
    s = s AND set_complement(t)
    return s
```

- `Set set_complement(Set s)`— : This function takes one parameter of type set: `s`. It returns the complement set `s`. This is the set that contains all the elements of the universal set $\mathbb{U}$ that are not in `s` and contains none of the elements that are in `s`.

```
Set set_complement(Set s)|
    s = NOT s
    return s
```

Insertion Sort

- `void insertion_sort(Stats *stats, int *arr, int length)` : This function takes three parameters: a stats pointer `stats`, and integer pointer `arr`, and an integer `length`. It does not return anything. Its purpose is to sort the elements of `arr`.

```
void insertion_sort(Stats *stats, int *arr, int length)
    for i, 1  to i < length
        j = i
        temp = arr[i]
        while j >= 1 and comp(stats, temp, arr[j-1]) == -1
            arr[j] = move(stats, arr[j-1])
            j -= 1
        arr[j] = temp
```

Shell Sort

- `void shell_sort(Stats *stats, int *A, int n)` : This function takes three parameters: a stats pointer `stats`, and integer pointer `A`, and an integer `n`. This function does not return anything. Its purpose is to sort the elements of `A`.

```
void shell_sort(Stats *stats, int *A, int n)
    for i, 0 to length of gaps array
        for j, gaps[i] to j < length of A array
            k = j
            temp = A[j]
            while k >= gap[i] and cmp(stats, temp, A[k - gap[i]]) == -1
                A[k] = move(stats, A[k - gap[i]])
                k = k - gap[i]
            A[k] = move(stats, temp)
```

Heap Sort

- `void heap_sort(Stats *stats, int *A, int n)` : This function takes three parameters: a stats pointer `stats`, an integer array pointer `A`, and an integer `n`. It does not return anything. Its purpose is to sort the elements of array `A`.

```
void heap_sort(Stats *stats, int *A, int n)
    first = 0
    last  = length of A - 1
    build_heap(A, first, last)
    for leaf, last to leaf > first
        swap(stats, A[leaf], A[first])
        fix_heap(A, first, leaf-1)
```

- `int max_child(Stats *stats, int *A, int first, int last)` : This function takes three parameters: an integer array `A`, an integer `first`, and an integer `last`. It returns an integer. It purpose is to determine which element is greater and return it.

```
int max_child(Stats *stats, int *A, int first, int last)
    left = 2 * first + 1
    right = 2 * first + 2
    if right <= last and cmp(stats, A[left], A[right]) == -1 then
        return right
    return left
```

- `bool fix_heap(Stats *stats, int *A, int first, int last)` : This function takes in three parameters: an integer array `A`, an integer `first`, and an integer `last`. It returns true once it is done. Its purpose is to fix the heap to once again obey the constraints of a max heap after removing the largest element.

```
bool fix_heap(Sats *stats, int *A, int first, int last)
    done = false
    parent = first

    while ((2 * parent + 1 ) <= last) and done = false
        largest_child = max_child(A, parent, last)
        if cmp(stats, A[parent], A[largest_child]) = -1 then
            swap(stats, A[parent],A[largest_child])
            parent = larges_child
        else
            done = true
```

- `void build_heap(int *A, int first, int last)` : This function takes in three parameters: an integer array `A`, an integer `first`, and an integer `last`. It does not return anything. Its purpose is to build a max heap.

```
void build_heap(int *A, int first, int last)
    if last > 0 then
        for parent, (last - 1) / 2 to parent > first - 1
            fix_heap(A, parent, last)
```

Quick Sort

- `int partition(Stats *stats, int *A, int lo, int hi)` : This function takes three parameters, an integer array `A`, and two integers `lo` and `hi`. It returns an integer. Its purpose is to divide the array into two partitions.

```
int partition(Stats *stats, int *A, int lo, int hi)
    i = lo -1
    for j, lo to j < hi
        if cmp(stats, A[j], A[hi]) == -1 then
            i++
            swap(stats, A[i], A[j])
    i++
    swap(stats, A[i], A[j])
    return i
```

- `void quick_sorter(Stats *stats, int *A, int lo, int hi)` : This function takes three parameters, an integer array `A`, and two integers `lo` and `hi`. It does not return anything. Its purpose is to recursively call itself and sort the array `A`.

```
void quick_sorter(Stats *stats, int *A, int lo, int hi)
    if lo < hi then
        p = partition(stats, A, lo, hi)
        quick_sorter(stats, A, lo, p - 1)
        quick_sorter(stats, A, P + 1, hi)
```

- `void quick_sort(Stats *stats, int *A, int n)` : This function takes three parameters: a stats pointer `stats`, an integer array pointer `A`, and an integer `n`. It does not return anything. Its purpose is to sort the elements in the array `A`.

```
void quick_sort(Stats *stats, int *A, int n)
    quick_sorter(stats, A, 0, length of A - 1)
```

Batcher Sort

- `void comparator(Stats *stats, int *A, int x, int y)` : This function takes three parameters: an integer array `A`, and two integers `x` and `y`. It does not return anything. Its purpose is to compare `A[x]` to `A[y]` and it it is greater, than swap the values.

```
void comparator(Stats *stats, int *A, int x, int y)
    if cmp(stats, A[y], A[x]) == -1 then
          Swap A[x] and A[y]
```

- `void batcher_sort(Stats *stats, int *A, int n)` : This function takes three parameters: a stats pointer `stats`, an integer array pointer `A`, and an integer `n`. It does not return anything. Its purpose is to sort the array `A`.

```
void batcher_sort(Stats *stats, int *A, int n)
    if n = 0 the
        return
    t = bit length of n
    p = 1 << (t - 1)

    while p > 0
        q = 1 << (t - 1)
        r = 0
        d = p
          while d > 0
```

```
                for i, 0 to i < n - d
                    if (i AND p) = r then
                        comparator(A, i, i + d)
              d = q - p
              q >>= 1
              r = p
          p >>= 1
```

Psuedocode

```
main
    set = 00000000 (0x00)
    heap = 0
    batcher = 1
    shell = 2
    quick = 3
    insert = 4
    elements = 100
    seed = 13371453
    size = 100

    while (opt = getopt) != -1
        switch opt
            case "H":
                print usage message
                return 0
                break
            case "a":
                set = set_universal()
                break;
            case "h":
                set = set_insert(set, heap)
                break;
            case "b":
                set = set_insert(set, batcher)
                break
            case "s":
                set = set_insert(set, shell)
                break
            case "q":
                set = set_insert(set, quick)
                break
            case "i":
                set = set_insert(set, insert)
                break
            case "n":
                size = optarg
                break
            case "p":
                elements = optarg
                break
            case "r":
                seed = optarg
                break
        default:
                print(USAGE, argv[0]);
                return 1;
    if set == 0x00 then
        print "Select at least one sort to perform"
        print usage message

    initialize Stats variable stats
    mask = 0x3FFFFFFF
    srandom(seed)
    first element = random()
    first element = first element AND mask
    array = allocated memory of size (first element size * number of elements)

    if set_member(set, insertion) then
        reset(stats)
        init_array(seed, size, array)
        insertion_sort(stats, array, size)
        print_stats(stats, "Insertion Sort", size)
        print_elements
    if set_member(set, heap) then
        reset(stats)
        init_array(seed, size, array)
        heapsort(stats, array, size)
        print_stats(stats, "Heap Sort", size)
        print_elements
    if set_member(set, shell) then
        reset(stats)
        init_array(seed, size, array)
        shell_sort(stats, array, size)
        print_stats(stats, "Shell Sort", size)
        print_elements
    if set_member(set, quick) then
        reset(stats)
        init_array(seed, size, array)
        quick_sort(stats, array, size)
        print_stats(stats, "Quick Sort", size)
        print_elements
    if set_member(set, batcher) then
        reset(stats)
        init_array(seed, size, array)
        batcher_sort(stats, array, size)
        print_stats(stats, "Batcher Sort", size)
        print_elements

    free(array)
    return 0
```

Error Handling

- Invalid command line arguments : For the three command line arguments that require additional input, if any of them recieve invalid input, they will remain their default values.

- No members in set : If no sorts are chose through command line arguments, the program the usage message is printed and and the program terminates.

Testing

The program will test each sorting algorithm with the randomly generated number. These randomly generated numbers will be bit-masked to fit in 30 bits. The program will be tested with an array that shows the program is capable of using an array up to the size of the available memory. To ensure no memory leaks, the program will be run with valgrind using the commands `make debug` and `valgrind ./sorting`.

To confirm the output of the program is correct, the `diff` command is utilized. To do this, the binary given in the resources repository will be run and the output will be directed into a file. This is done using a command such as `./sorting_binary -a > expect1.txt` or `./sorting_binary -a 2> expect_err1.txt`. The same commands are used with this programs executable. The files generated can then be compared using the `diff` command.

Results

References

[1] Dr. Keery Veenestra and TAs. Assignment 4: Sets and sorting. `https://git.ucsc.edu/cse13s/fall-2023-section-01/resources`, Fall 2023.