asgn3 – Sorting: Putting your affairs in order

Student name: *Serjo Barron*

Course: *CSE13S* – Professor: *Prof. Long*
Due date: *10/14/21*

1 Program

This program implements 4 sorting algorithms (all which vary in efficiency): insertion sort, shell sort, heapsort, and quicksort. These sorting algorithms will all be defined in their respective file, and be linked and compiled to be used inside a Test Harness. The Test Harness will be the main point of interaction between the user and program.

The Test Harness is where the user can run each sorting algorithm individually, many, or all at once. The user may also define or not define the number of elements printed, the length of the array being sorted, and the seed used to pseudo-randomly generate elements bit-masked to fit in 30 bits.

When a sorting algorithm is ran the program will print the corresponding statistics listing the number of elements being sorted, the amount of moves made, and the amount of comparisons made all in order to sort the array of length n.

2 Required Files

1. insert.c & insert.h
    - The ".c" file includes the implementation of the insertion sort algorithm, and the ".h" file is a header file used to link to the Test Harness (sorting.c).
2. heap.c & heap.h
    - The ".c" file includes the implementation of the heap sort algorithm, and the ".h" file is a header file used to link to the Test Harness (sorting.c).
3. quick.c & quick.h
    - The ".c" file includes the implementation of the quick sort algorithm, and the ".h" file is a header file used to link to the Test Harness (sorting.c).
4. shell.c & shell.h
    - The ".c" file includes the implementation of the shell sort algorithm, and the ".h" file is a header file used to link to the Test Harness (sorting.c).
5. set.h

- A header file included in sorting.c which used to keep track of the command line options.
- This file has functions for basic set operations such as union, intersect, complement. And has functions for modifying sets such as empty, member, and insert.
- All the functions in this header may or may not be used.

6. stats.c & stats.h
   - stats.c includes basic functions to compare, move, swap, and reset the statistics that each of the functions keep track of, which are defined in a struct.
   - stats.h is a header file which defines the struct that stats.c modifies, this header file will be included in all sorting functions and the test harness.
7. sorting.c
   - This file will be the Test Harness for all the sorting algorithms, this is where the user will run the program and enter user-defined options.
8. Makefile
   - This file is used to link all the required files to each other to be linked to the Test Harness (sorting.c), and compile them into an executable.
9. README.md
   - A file in markdown syntax that provides instructions on how to run the program, and a program description.
10. WRITEUP.pdf
    - A file which will analyze each of the sorting algorithms, compare them against each other, and discuss interesting findings/behavior of graphs.
11. DESIGN.pdf
    - A file detailing the design process of the program (this current document).

3 Psuedocode, Structure

3.1 Additional Functions: Stats

In addition to all the sorting functions, there is a file **stats.c** which includes functions to move, swap, compare array elements. The following functions will be given a brief pseudocode to visualize what each of them do, since the sorting functions will make use of these functions to keep track of moves and comparisons of array elements.

The moves and comparisons will be kept inside of a struct defined inside **stats.h**, a header file which will be included inside all sorting functions, stats.c, and the test harness.

stats.c also includes a reset function, but this will be used inside the test harness, not inside the actual sorting algorithm functions.

```
1  define cmp(stats, x, y):
2        compares += 1 (access by using stat pointer to struct element)
3        if x < y:
4              return -1
5        else if x > y:
6              return 1
7        else
8              return 0
9
10 define move(stats, x):
11       moves += 1 (access by using stat pointer to struct element)
12       return x
13
14 define swap(stats, x, y):
15       moves += 3 (access by using stat pointer to element)
16       t = x
17       x = y
18       y = t
19
20 define reset(stats):
21       moves = 0 (resets moves & compares to 0)
22       compares = 0
```

3.2 Additional Functions: Sets

A **sets.h** (header file) will also be provided which includes various functions to compute set operations. Not all of the functions will be used, but the main functions to be used in the Test Harness goes as follows:

1. **empty_ set()**
    - This function when called will initialize an empty set.
2. **insert_ set()**
    - This function allows to insert a element into a set.
3. **member_ set()**
    - This function checks whether a certain element exists within a set, and returns true if so, otherwise it returns false.

3.3 Insertion Sort

```
1  define insertion_ sort(stats, A, n):
2      for k in range of 1 to n:
3          j = k
4          temp = move(stats, A[k]) (+1 move)
5          while j > 0 and cmp(stats, A[j − 1], temp) > 0:
6              A[j] = move(stats, A[j − 1]) (+1 move)
7              j −= 1
8          A[j] = move(stats, temp) (+1 move)
9      return (returns nothing, void)
10
```

3.4 Shell Sort

```
1  define static 32 bit unsigned variables: gap and gap_ index
2
3  define gaps(): (has no parameters, void)
4      gap = (3^gap_index − 1)/2
5      gap_ index −= 1
6      return gap
7
8  define shell_ sort(stats, A, n):
9      gap_ index = (log(3 + 2 ∗ n)/log(3)) (calculates initial max gap_ index)
10     do:
11         gaps()
12         for i in range of gap to n:
13             j = i
14             temp = move(stats, A[i]) (+1 move)
15             while j ≥ gap and cmp(stats, temp, A[j − gap]) < 0:
16                 A[j] = move(stats, A[j − gap])(+1 move)
17                 j −= gap
18             A[j] = move(stats, temp) (+1 move)
19     while (gap > 1) (condition for the do: part of do-while loop)
20
21     return (returns nothing, void)
```

3.5 Heapsort

```
1  (the heapsort sorting algorithm requires 3 helper functions:
2  max_ child, fix_ heap, and build_ heap)
3
4  define max_ child(stats, A, first, last):
5      left = 2 × first
6      right = left + 1
7      if right ≤ last and cmp(A[right − 1], A[left − 1] > 0:
8          return right
9      return left
10
11 define fix_ heap (stats, A, first, last):
12     define boolean for found, and set to false
13     mother = first
14     great = max_ child(stats, A, mother, last)
15
16     while mother ≥ (floor division: last / 2) and not found:
17         if cmp(A[mother − 1], A[great − 1]) < 0:
18             swap(A[mother − 1], A[great − 1])
19             mother = great
20             great = max_ child(stats, A, mother, last)
21         else:
22             set boolean found to true
23
24     return (returns nothing, void)
25
26 define build_ heap(stats, A, first, last):
27     for i in range(father, first - 1) (decrement by 1)
28         fix_ heap(stats, A, father, last)
29     return (returns nothing, void)
30
31 define heap_ sort(stats, A, n):
32     first = 1
33     last = n
34     build_ heap(stats, A, first, last)
35     for leaf in (last, first): (decrement by 1
36         swap(A[first − 1], A[leaf − 1])
37         fix_ heap(stats, A, first, leaf − 1)
38
39     return (returns nothing, void)
```

3.6 Quicksort

```
1  (the quicksort sorting algorithm requires 2 helper functions:
2  partition and quick_ sorter)
3
4  define partition(stats, A, low, high):
5      i = low − 1
6      for j in range(low, high):
7          if cmp(A[j − 1], A[high − 1]) < 0:
8              i += 1
9              swap(A[i − 1], A[j − 1])
10     swap(A[i], A[high − 1])
11     return (i + 1)
12
13 define quick_ sorter(stats, A, low, high):
14     if low < high:
15         p = partition(stats, A, low, high)
16         quick_ sorter(stats, A, low, p - 1) (recursive call)
17         quick_ sorter(stats, A, p + 1, high) (recursive call)
18     return (returns nothing, void)
19
20 define quick_ sort(stats, A, n):
21     quick_ sorter(stats, A, 1, n)
22     return (returns nothing, void)
```

3.7 Test Harness

```
1  define options "haeisqn:p:r:" (the colon after a character signifies it takes user input)
2
3  define an enumeration including all the options (except 'a') with type Sort:
4  (NUM_ SORTS is used as a range value to iterate over HEAP, INSERT, SHELL, and QUICK)
5
6  define a const array for the names of the sorting functions:
7      names[] = ["Heap Sort", "Shell Sort", "Insertion Sort", "Quick Sort"]
8
9  define a static (32-bit unsigned int) for the bitmask, set to the hex value 0x3fffffff
10
11 define array for function pointers:
12     (pointer[4]) (stats, A, n)
13
14 pointer[0] = heap_ sort (elements for pointer[] array)
15 pointer[1] = shell_ sort (the elements are the names of the function)
16 pointer[2] = insertion_ sort
17 pointer[3] = quick_ sort
18
19 define make_ array(A, n, seed): (function used to make/reset the array)
20     srandom(seed) (use seed for random() generation)
21     for i in range 0 to n (in increments of 1):
22         A[i] = random() & bitmask (bit-wise mask the random() number)
```

```
1  (continuation of Test Harness pseudo-code)
2      (define static variables for n, p, and r optarg, all using type 32 bit unsigned int)
3      n
4      print_ num
5      seed
6
7      Set s = empty_ set() (initialize an empty set 's')
8
9      (begin while loop and main function for getopt)
10     while ((opt = getopt(argc, argv, options)) is not -1):
11         switch(opt)
12         case 'h':
13             s = insert_ set(HEADER, s)
14             break
15         case 'a':
16             s = insert_ set(HEAP, s)
17             s = insert_ set(INSERT, s)
18             s = insert_ set(SHELL, s)
19             s = insert_ set(QUICK, s)
20             break
21         case 'e':
22             s = insert_ set(HEAP, s)
23             break
24         case 'i':
25             s = insert_ set(INSERT, s)
26             break
27         case 's':
28             s = insert_ set(SHELL, s)
29             break
30         case 'q':
31             s = insert_ set(QUICK, s)
32             break
33         case 'n':
34             n = atoi(optarg);
35             textbf(atoi() converts string to integer) s = insert_ set(LEN, s)
36             break
37         case 'p':
38             print_ num = atoi(optarg)
39             s = insert_ set(PRINT, s)
40             break
41         case 'r':
42             seed = atoi(optarg)
43             s = insert_ set(SEED, s)
44             break
45         default:
46             s = insert_ set(HEADER, s)
47             break
```

```
1  (continuation of Test Harness pseudo-code)
2       if HEADER is in Set s:
3            print synopsis message
4            return 0 (terminate program)
5
6       (conditionals to check whether to set n, p, or r to default values)
7       if not member_ set(LEN, s):
8            n = 100
9       if not member_ set(PRINT, s):
10           print_ num = 100
11      if not member_ set (SEED, s):
12           seed = 13371453
13
14      (Conditional to check whether the number of printed elements exceeds
15      the available array length)
16      if print_ num > n:
17           print_ num = n (If so, default print_ num to n)
18
19      define pointer to Stats called stat
20      stat.moves = 0
21      stat.compares = 0
22
23      (Allocate memory)
24
25      for loop to iterate over enum starting from HEAP to NUMBER_ SORTS using x:
26           if member_ set(x, s): (x is initialized to HEAP (value of 0 in enum)
27                reset(address of stat)
28                make_ array(A, n, seed) (make array)
29                (pointer[x]) (address of stat, A, n) (call the function pointer array)
30                print ("names[x]")
31
32                print number of elements, n
33                print number of moves, stat.moves
34                print number of comparisons, stat.compares
35
36                for loop to print number of specified array elements:
37                     print the elements to %13 precision in rows of 5
38
39           make_ array(A, n, seed) (resets array to be unsorted)
40
41      (Deallocate memory)
42      return 0 (terminate program)
```

4 Function Usage

1. **Sorting Functions:**
    - As seen in each of the functions, as well as the helper functions, they may take the following parameters: stats, A, and n.
    - stats is a pointer to the Stats struct, and when calling a function the parameter passed should be the address of a pointer to Stats.
    - A an array, which when passed as a parameter should simply be the array name.
    - n is the length of the array. This value should be stored in a variable when the user enters the number of pseudo-random elements to generate in an array of size n.
2. **Test Harness:**
    - When the user runs the executable sorting without choosing any of the options, the user will be prompted with a synopsis message detailing the usage of the program.
    - The user has the following options 'h' (displays program usage), 'a' (runs all sorting tests), 'e' (runs heapsort), 'i' (runs insertion sort), 's' (runs shellsort), 'q' (runs quicksort), 'n' (user-defined array length), 'p' (user-defined printed elements), and 'r' (user-defined seed).
    - If 'n' (array length) is not defined, n will default to 100.
    - If 'p' (number of printed elements) is not defined, p will default to 100.
    - If 'r' (seed) is not defined, r will default to 13371453.

5 Credit

1. All sorting algorithms have been based on the python code provided by Prof. Long inside the assignment 3 (specifications) document.
2. Parts of the sorting.c "Test Harness" implementation has also been inspired by an example/explanation provided in Eugene's section held on Tuesday (10/12). Particularly I used ideas such as function pointers, creating an enumeration to iterate over, and making an array that stores the names of the sorting algorithms.