Nicolas Corfmat
CSE13S, Winter 2023
ncorfmat@ucsc.edu

Assignment 3 DESIGN.pdf

1 Description of Program:

This program sorts arrays of pseudorandom numbers using the following four algorithms: Shell Sort, Heap Sort, Batcher Sort, and Quicksort. The program simultaneously tracks the number of moves, compares, and swaps performed by each sorting algorithm, and conveniently plots this information on a graph. The user can then independently compare differences between each sorting algorithm.

2 Files to be included in directory “asgn3”:

1. Makefile:
    - Supports the compilation of the all .c programs in the “asgn3” directory
2. sorting.c:
    - Test file containing the main() and command-line options to run each sorting algorithm.
3. stats.c:
    - Statistics module that collects information about moves, compares, and swaps.
4. stats.h:
    - Header file containing the interface to stats.c
5. set.c:
    - Bit-wise operations to be implemented with Set
6. set.h:
    - Header file containing the interface to set.c
7. shell.c:
    - File containing the Shell Sort algorithm.
8. shell.h:
    - Header file containing the interface to shell.c
9. gaps.h:
    - Header file containing the gaps sequence to be used in shell.c

10. heap.c:
    - File containing the Heap Sort algorithm.
11. heap.h:
    - Header file containing the interface to heap.c
12. batcher.c:
    - File containing the Batcher Sort algorithm.
13. batcher.h:
    - Header file containing the interface to batcher.c
14. quick.c:
    - File containing the Quicksort algorithm.
15. quick.h:
    - Header file containing the interface to quick.c
16. README.md:
    - Proper explanation of how to run the program and *Makefile*. It also contains a list of available command-line options and explains their functions.
17. DESIGN.pdf:
    - Explains the thought process behind the program with sufficient detail. Comes in PDF format.
18. WRITEUP.pdf:
    - PDF file created using LaTeX that includes graphs displaying the differences between approximate and actual values and an analysis as to what they mean.

3 Pseudocode / Structure:

sorting.c:

```
Create getopt variable and set it to 0
Set default array size to 100
Set default seed to 13371453
Create an array
Create a stats variable
While retrieving arguments from command line:
    If user types -a:
        Add Heap Sort to set
        Add Batcher Sort to set
        Add Shell Sort to set
        Add Quicksort to set
    If user types -h:
        Add Heap Sort to set
```

If user types -b:
    Add Batcher Sort to set
If user types -s:
    Add Shell Sort to set
If user types -q:
    Add Quicksort to set
If user types -r:
    Accepts an argument for setting a new seed
If user types -n:
    Accepts an argument for array size
If user types -p:
    Prints out *element* number of elements
If user types -H:
    Print out program usage

Set array to given size

If number corresponding to Heap Sort is in the Set:
    Fill the array with pseudorandom numbers
    Perform Heap Sort on the array
    Print the elements in array with appropriate column labels

    Reset stats
    Free the array

If number corresponding to Batcher Sort is in the Set:
    Fill the array with pseudorandom numbers
    Perform Batcher Sort on the array
    Print the elements in array with appropriate column labels

    Reset stats
    Free the array

If number corresponding to Shell Sort is in the Set:
    Fill the array with pseudorandom numbers
    Perform Shell Sort on the array
    Print the elements in array with appropriate column labels

    Reset stats
    Free the array

If number corresponding to Quicksort is in the Set:
    Fill the array with pseudorandom numbers
    Perform Quicksort on the array
    Print the elements in array with appropriate column labels

    Reset stats
    Free the array

Define a function that takes an array, array size, and seed as inputs:
    Perform random on seed

    For each element in the array:
        Place a randomly generated number into the element's location

Define a function that takes an array and array size as inputs:
    For each element in the array:
        If the location of the element is a multiple of 5:
            Print a new line
        Print the element

shell.c:

Define the shell sort function which takes stats, an array pointer, and array size as arguments:
    For each element in the given gaps array:
        For each location value between the range element and array size:
            Record the current location value into a new variable *j*
            Store the value of the element at current location in the passed array into a temporary variable

            While *j* is greater than current gap value and the temporary variable is less than the element at array[j-gap]:
                Store element at array[j-gap] into array element at location *j* and increase moves by 1
                Subtract the current gap value from *j*

            Set the array element at location *j* to the temporary variable and increment moves by 1

heap.c:

Define a max_child function that takes stats, an array pointer, and two integers (*first* and *last*) as inputs:

Create an integer *left* that is double the amount of *first*
Create an integer *right* that is 1 more than the value of *left*

If *right* is less than or equal to *last* and the value of array[right-1] is greater than array[left-1] (increment compares by 1):

Return the value of *right*

Else:

Return the value of *left*

Define a fix_heap function that takes stats, an array pointer, and two integers (*first and last*) as arguments:

Create a boolean *found* and set it to false
Create an int *mother* and assign it the value of *first*
Create an int *great* and assign it the value returned by calling the function max_child() with stats, current array, *mother*, and *last* as arguments.

While *mother* is less than or equal to the rounded down integer of *last* / 2 and found is true:

If the value of array[*mother*-1] is less than the value of array[*great*-1] (increment compares by 1):

Swap array[*mother*-1] and array[*great*-1], making sure to increment swaps
Assign *great* to mother

Else:

Assign true to *found*

Define a build_heap function that takes stats, an array pointer, and two integers (*first* and *last*) as arguments:

For every element between the rounded down integer of *last*/2 and one less than *first*:

Perform fix_heap() with stats, provided array, *father*, and *last* as inputs

Define a heap_sort function that takes stats, an array pointer, an array size as arguments:

Assign 1 to variable *first*
Assign array size to variable *last*

Call build_heap, giving it stats, provided array, *first*, and *last* as inputs

For every element between last and first, in descending order:
Swap the values of array[*first*-1] and array[element-1], and increment swaps by 1
Call fix_heap(), giving it stats, an array, *first*, and *leaf* - 1 as inputs

quick.c:

Define a function partition() that takes stats, an array pointer, and two ints (*lo* and *hi*) as arguments:
Assign 1 less than *lo* to a new integer *i*

For every number between *lo* and *hi*:
If array[*j*-1] is less than array[*hi*-*i*] (remembering to increment compares by 1):
Increment *i* by 1
Swap the values of array[*i*-1] and array[*j*-1], incrementing swaps by 1

Swap the values of array[*i*] and array[*hi*-1], incrementing swaps by 1

Return one more than *i*

Define a function quick_sorter that takes stats, an array pointer, and two ints (*lo* and *hi*) as arguments:
If *lo* is less than *hi* (remembering to increment compares by 1):
Assign the result of calling partition() with inputs, provided array, *lo*, and *hi*, as inputs, to a new variable *p*

Call quick_sorter, giving it stats, the provided array, *lo*, and *p - 1* as inputs
Call quick_sorter, giving it stats, the provided array, *p* + 1, and *hi* as inputs

Define a function quick_sort that takes stats, an array pointer, and array size as arguments:
Call quick_sorter, giving it stats, the provided array, and array size as inputs

batcher.c:

Define a function comparator that takes stats, an array pointer, and two ints (*x* and *y*) as arguments:
If element of array at location *x* is greater than element at location *y* (remembering to increment compares by 1):
Swap both elements, incrementing swaps by 1

Define a function batcher_sort that takes stats, an array pointer, and array size as arguments:
    If array size is 0:
        Exit function
    Assign the bit length of array size to a variable *t*
    Assign the result of left shifting 1, *t* - 1 times, to variable *p*

    While p is greater than 0:
        Assign the result of left shifting 1, *t* - 1 times, to variable *q*
        Assign 0 to a new variable, *r*
        Assign the value of *p* to a new variable *d*

        While *d* is greater than 0:
            For every number between 0 and the difference between *n* and *d*:
                If the bit-wise AND of *i* and *p* is equal to *r*:
                    Call comparator, giving it stats, the array, *i*, and sum of *i* and *d* as inputs

            Assign the difference between *q* and *p* to *d*
            Right shift *q* by 1 bit
            Assign the value of *p* to *r*

        Right shift *p* by 1 bit

set.c:

    Create a function set_empty:
        Return 0

    Create a function set_universal:
        Return the inverse of the 32-bit int 0

    Create a function set_insert that takes a set and 8-bit int as arguments:
        Returns the set with the bit corresponding to x set to 1

    Create a function set_remove that takes a set and 8-bit int as arguments:
        Return the set with the 8-bit value removed from it

    Create a function set_member that takes a set and 8-bit int as arguments:
        Returns true if the passed int value is an element in the set, else otherwise

Create a function set_union that takes two sets as inputs:
Returns a set with all the values of both sets

Create a function set_intersect that takes two sets as inputs:
Returns a set with all the elements that both sets share

Create a function set_difference that takes two sets as inputs:
Returns a set of elements found in the first set that aren't in the second set

Create a function set_complement that takes a set as an input:
Returns the complement of the given set (all bits flipped)

4 Credit:

- All four sorting algorithms were derived from python code provided by Darrell Long. Although deviations from his code exist in the provided pseudo code, most of it (especially the structure) remains the same.