Assignment 3 Design Document

Program Requirements

This program creates an array of a given size, fills it with pseudorandom integers through a given seed, and sorts it using a specified algorithm. The user has the option of sorting the array via heap sort, batcher sort, insertion sort, quicksort, or any combination of them. If the user opts not to provide a size or seed, they will default to the values, 100 and 13371453, respectively. The user also has the option of specifying how many elements of the entire array (once sorted) will be printed in the final output. In the case that the specified or default array size is less than the given number of elements to be printed, the program must print out the entire array and nothing more. In the final output, before the sorted array is printed, the program must also specify what sorting method was used, the number of elements printed, as well as the number of moves and comparisons made when sorting. If the user uses all the algorithms, each of those specifications must be provided for each sorted array printed.

sorting.c Pseudocode

Include all header files as well as the following libraries:

- stdio.h
- stdlib.h
- inttypes.h
- unistd.h

Create a function to print the sorted array in the format given in the document with the following input parameters:

- A
    - This is a sorted array containing elements of the type uint32_t.
- arr_size
    - This is a variable of type uint32_t which holds the size of the array.
- num_elements
    - This is another variable of type uint32_t which holds the number of elements to print.

The function must return void.

Create a function to print the usage message. This should specify what are all the possible options the user can enter and what their operation is. The only input parameter should be the executable name, and the function should return void.

Now we enter the main function…

First we must declare the following variables:

- arr_size
    - Size of the array generated

    - Type: uint32_t
    - Default value: 100
- *A, *B
    - Pointers to arrays that each hold elements of type uint32_t
- stats
    - References the functions in the provided Stats structure
    - Will be relevant for measuring the number of moves and compares
- num_elements
    - Default value: 0xFFFFFFFF
        - In order for the pseudorandom numbers generated by random() to be bit-masked to fit in 30 bits, the user-provided value of num_elements will be “AND-ed” with a series of “true” values.
- seed
    - Default value: 13371453
        - As per the guidelines
- opt
    - Type: int
- input_options
    - References the functions provided in the provided Set structure
    - Will be relevant for running multiple sorting algorithms at once
    - Default value: empty_set()

Next, we need to parse the input options using getopt(), and store the sorting options in a set. Here are the following options we need to define:

- ‘a’
    - Use all four algorithms.

- ‘b’
    - Use batcher sort.
- ‘h’
    - Use heap sort.
- ‘i’
    - Use insertion sort.
- ‘q’
    - Use quick sort.
- ‘p <print_size>’
    - Print at most the specified number of elements.
- ‘n <num_to_sort>’
    - This is the number of elements to sort.
    - As mentioned above, the default value is 100.
- ‘r <seed>’
    - Set the seed for the pseudorandom numbers.
    - As mentioned above, the default value is 13371453.
- Default
    - This option will catch the ‘-H’ option.
    - This runs the usage function.

Next, Set the seed using srandom().

After that, allocate memory for A. Use another array to copy the unsorted array to ensure correct statistics if multiple sorting options are given. In each case, copy the original unsorted array to ensure proper statistics. For the implementation of this, it may be necessary to know how to print a variable of type uint64_t. The following link will be helpful in that regard: https://stackoverflow.com/questions/9225567/how-to-print-a-int64-t-type-in-c.

Finally, we need to free the memory allocated for both A and B.

insert.c Pseudocode

Include insert.h and the following libraries:

- stdio.h
- inttypes.h

This file will contain only one function which sorts a given input array A using insertion sort. The number of comparisons and moves done are saved to achieve this in the Stats data structure. The function will take the following input parameters:

- stats
    - References the functions in the Stats structure (Stats *)
    - Used to store the number of moves and compares
- A
    - This is an unsorted array, which is sorted in the function.
    - The elements are pointers to unsigned integers (uint32_t *).
- n
    - n
        - Type: uint32_t
        - Number of elements in A

The function must return void.

This function must contain a loop which examines the elements of the array in order, comparing the element of the current iteration to all the previous elements until it finds its place in an ascending numerical order.

For further clarification, here is the insertion sorting algorithm implemented in Python:

Insertion Sort in Python

```python
def insertion_sort(A: list):
    for i in range(1, len(A)):
        j = i
        temp = A[i]
        while j > 0 and temp < A[j - 1]:
            A[j] = A[j - 1]
            j -= 1
        A[j] = temp
```

heap.c Pseudocode

Include heap.h as well as the following libraries:

- stdio.h
- inttypes.h

A heap sort is done by taking the first element and making it the “parent” for which the next two elements will be its “children.” Each child element will have its own two children until all the

elements in the array (save the first) are assigned a parent. In a max heap, a parent node is always greater than or equal to the child nodes. Therefore to sort the original array, keep creating max heaps and swapping the first and last nodes. When there is only a single node left, the array will have been sorted.

To implement this…

First, create a function that will compare and return the higher of two “children.” The following should be the input parameters:

- stats
    - This is a pointer to the provided Stats structure (Stats *).
    - It is used to store the number of moves and compares.
- A
    - This is the input array.
    - It contains pointers to unsigned integers (uint32_t *).
- first, last
    - They are both of type uint32_t.
    - These are the first and last elements of the array.

This function should return the type uint32_t.

Next, create a function that fixes the heap to ensure it obeys the constraints of a heap. It should contain all of the same parameters as for the previous function. This function, however, should return void.

The third function should build a heap from the given input array. The only parameter for this function should be the stats parameter explained for the first function. This function should also

return void. It must accomplish this by invoking the function to fix the heap across a descending for loop from last to first.

The final function required should contain the sorting algorithm. Once again, save the number of comparisons and moves done to achieve this in the Stats data structure. The input parameters for this function should be the same as for the first function, and this function should return void.

The implementation for the final function must be to first reset the stats variable, and then invoke the functions to swap the nodes in the heap and to fix the heap under another descending for loop ranging from last to first.

For further clarification, here is the heap sorting algorithm implemented in Python:

Heap maintenance in Python

```
def max_child(A: list, first: int, last: int):
    left = 2 * first
    right = left + 1
    if right <= last and A[right - 1] > A[left - 1]:
        return right
    return left

def fix_heap(A: list, first: int, last: int):
    found = False
    mother = first
    great = max_child(A, mother, last)

    while mother <= last // 2 and not found:
        if A[mother - 1] < A[great - 1]:
            A[mother - 1], A[great - 1] = A[great - 1], A[mother - 1]
            mother = great
            great = max_child(A, mother, last)
        else:
            found = True
```

Heapsort in Python

```python
def build_heap(A: list, first: int, last: int):
    for father in range(last // 2, first - 1, -1):
        fix_heap(A, father, last)

def heap_sort(A: list):
    first = 1
    last = len(A)
    build_heap(A, first, last)
    for leaf in range(last, first, -1):
        A[first - 1], A[leaf - 1] = A[leaf - 1], A[first - 1]
        fix_heap(A, first, leaf - 1)
```

quick.c Pseudocode

Include quick.h, stdio.h, and inttypes.h.

The first function should partition the input array to separate all elements higher than a key on one side and lower than the key on the other. It should contain the following input parameters:

- stats
    - Pointer to the Stats structure (Stats *)
    - Used to store the number of moves and compares
- A
    - Array that needs to be partitioned
    - Contains pointers to unsigned integers (uint32_t *)
- lo
    - Type: uint32_t

    - Lower index of the array
- hi
    - Type: uint32_t
    - Upper index of the array

The function should return the index of the key.

Next, we need a recursive function to partition the array. This function will contain the same parameters as before and will return void.

For every case where lo is less than hi, the following should work:

1. Apply the parameters for this function to calling the previous one, and assign the value to some uint32_t variable defined here.
2. Recur this function using the parameters stats, A, lo, and p - 1.
3. Recur this function using the parameters stats, A, p + 1, and hi.

The above should continue until the list contains only one element.

Finally, we need to sort the given input array A using quicksort. As always, we also need to save the number of comparisons and moves done to achieve this in the stats data structure. The following should be the input parameters:

- stats
    - As above
- A
    - Unsorted array, which is sorted in the function
    - Contains pointers to unsigned integers (uint32_t *)
- n
    - Type: uint32_t
    - Number of elements in A

This function should also return void. It must accomplish its aim by first resetting stats, and then calling the previous function using the parameters stats, A, 1, and n.

For further clarification, here is the sorting algorithm implemented in Python:

Partition in Python

```python
def partition(A: list, lo: int, hi: int):
    i = lo - 1
    for j in range(lo, hi):
        if A[j - 1] < A[hi - 1]:
            i += 1
            A[i - 1], A[j - 1] = A[j - 1], A[i - 1]
    A[i], A[hi - 1] = A[hi - 1], A[i]
    return i + 1
```

Recursive Quicksort in Python

```python
# A recursive helper function for Quicksort.
def quick_sorter(A: list, lo: int, hi: int):
    if lo < hi:
        p = partition(A, lo, hi)
        quick_sorter(A, lo, p - 1)
        quick_sorter(A, p + 1, hi)

def quick_sort(A: list):
    quick_sorter(A, 1, len(A))
```

batcher.c Pseudocode

Include batcher.h, stdio.h, and inttypes.h.

First, create a function that returns the bit length of an input number. That input number should be the only function parameter.
It is most efficient to accomplish this by continually dividing the initial number by two, and incrementing the value of a local variable by one (starting from zero) with every iteration, until the initial number is whittled down to one. Then return the local variable.

Next, we need a function to compare two elements of an array, and swap, if required. The parameters needed for this function will be the following:

- stats
    - Same as for all previous files
- A
    - Type: uint32_t *
    - The unsorted array containing pointers to unsigned integers
- x, y
    - Type: uint32_t
    - The indices that need to be compared

This function should return void. The logic for this function is already implemented in the given file stats.c.

Now we arrive at the function that will perform the sorting algorithm. Batcher sort is described as a divide-and-conquer algorithm. It continually finds the midpoint of a section of an array (starting with the entire array) and partitions it into even more sections until each element is its own section. This is when the “merging” begins. Each section is reunited, but in order, moving up the level of partitioning to get back to the original array.
This final function uses that method to sort an input array. As always, we must save the number of comparisons and moves done to achieve this in the Stats data structure.

Input parameters:

- stats
    - As above
- A
    - As above
- n
    - Type: uint32_t
    - Number of elements in A

This function should return void, and function according to this given Python code:

Merge Exchange Sort (Batcher's Method) in Python

```python
def comparator(A: list, x: int, y: int):
    if A[x] > A[y]:
        A[x], A[y] = A[y], A[x]

def batcher_sort(A: list):
    if len(A) == 0:
        return

    n = len(A)
    t = n.bit_length()
    p = 1 << (t - 1)

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

        while d > 0:
            for i in range(0, n - d):
                if (i & p) == r:
                    comparator(A, i, i + d)
            d = q - p
            q >>= 1
            r = p

        p >>= 1
```

Expected Output for the Shell Script for the Write-Up