Assignment 4
Sets and Sorting

by Prof. Darrell Long
edited by Dr. Kerry Veenstra and Jess Srinivas
CSE 13S, Fall 2023
Document version 1

Due Sunday November 5th, 2023, at 11:59 pm
Draft Due Friday November 3rd, 2023, at 11:59 pm

1   Introduction

> *Any inaccuracies in this index may be explained by the fact that it has been sorted with the help of a computer.*
>
> —Donald Knuth, Vol. III, *Sorting and Searching*

You will be implementing two kinds of functions in this assignment. The simpler kind will be a set of *Set* functions. You will use the Set functions and `getopt()` to record which which command-line options are provided. (See Section 2.)

Then you will implement four Sorting algorithms. (This assignment has the pseudocode for *five* sorting algorithms. You will implement at least *four* of them—you get to choose which ones.) The Sorting algorithms will order arrays of integers. (See Sections 3 through 8.)

To help you collect and report statistics about the operation of your sorting algorithms, your implementations will manipulate the integer arrays using functions that we provide you. These provided functions will record the number of operations that are performed and then will let you print the recorded totals. (See Section 9.)

Sections 10 through 13 discuss your task and what you will be submitting.

2   Sets

For this assignment, you are required to use a *set* to track which command-line options are specified when your program is run. For example, the command `sorting -s` should "insert" SHELL into

The function declarations for sets is given in the resources repository in `set.h`. You may not modify this file.

You are tasked with implementing the functions listed within `set.h` in a separate file named `set.c`. In the context of this assignment, a `Set` is nothing more than a type alias for an 8-bit integer. As you should recall from CSE 12, a bit may have one of two states, either 0 or 1. Membership of a set will be handled similarly, either an index is a member of the set, or it isn't. Every bit within the set represents membership of a different element up until a maximum of 8 possible elements, numbered 0 through 7 inclusively.

For manipulating the bits in a set, we use bit-wise operators. These operators, as the name suggests, will perform an operation on every bit in a number. The following are the six bit-wise operators specified in **C**:

&	bit-wise AND	Performs the AND operation on every bit of two numbers.
|	bit-wise OR	Performs the OR operation on every bit of two numbers.
~	bit-wise NOT	Inverts all bits in the given number.
^	bit-wise XOR	Performs the exclusive-OR operation on every bit of two numbers.
<<	left shift	Shifts bits in a number to the left by a specified number of bits.
>>	right shift	Shifts bits in a number to the right by a specified number of bits.

Recall that the basic set operations are: *membership*, *union*, *intersection* and *negation*. Using these functions, you will set (make the bit 1) or clear (make the bit 0) bits in the Set depending on the command-line options read by getopt(). You can then check the states of all the bits (the members) of the Set using a single for loop and execute the corresponding sort. Note: you most likely won't use all the functions, but you *must* use sets to track which command-line options are specified when running your program.

Set set_empty(void)

This function is used to return an empty set. In this context, an empty set would be a set in which all bits are equal to 0.

Set set_universal(void)

This function is used to return a set in which every possible member is part of the set. While you may not need to use this function in this program, you are still required to write it.

Set set_insert(Set s, int x)

This function inserts x into s. That is, it returns set s with the bit corresponding to x set to 1. Here, the bit is set using the bit-wise OR operator (|). The first operand for the OR operation is the set s. The second operand is value obtained by left shifting 1 by x number of bits (0x01 << x). None of the other bits may change.

Set set_remove(Set s, int x)

$$s - x = \{y | y \in s \land y \neq x\}$$

This function deletes (removes) x from s. That is, it returns set s with the bit corresponding to x cleared to 0. Here, the bit is cleared using the bit-wise AND operator (&). The first operand for the AND operation is the set s. The second operand is a negation (~) of the number 1 left shifted to the same position that x would occupy in the set (0x01 << x). This means that the bits of the second operand are all 1s except for the bit at x's position. The function returns set s after removing x. Again, none of the other bits should change.

bool set_member(Set s, int x)

$$x \in s \iff x \text{ is a member of set } s$$

This function returns a bool indicating the presence of the given value x in the set s. The bit-wise AND operator (&) is used to determine set membership. The first operand for the AND operation is the set s. The second operand is the value obtained by left shifting 1 x number of times (0x01 << x). If the result of the AND operation is a non-zero value, then x is a member of s and true is returned to indicate this. false is returned if the result of the AND operation is 0.

Set set_union(Set s, Set t)

$$s \cup t = \{x | x \in s \lor x \in t\}$$

The union of two sets is a collection of all elements in both sets. Here, to calculate the union of the two sets `s` and `t`, we need to use the OR operator (`|`). Only the bits corresponding to members that are equal to 1 in either `s` or `t` are in the new set returned by the function.

Set set_intersect(Set s, Set t)

$$s \cap t = \{x | x \in s \land x \in t\}$$

The intersection of two sets is a collection of elements that are common to both sets. Here, to calculate the intersection of the two sets `s` and `t`, we need to use the AND operator (`&`). Only the bits corresponding to members that are equal to 1 in both `s` and `t` are in the new set returned by the function.

Set set_difference(Set s, Set t)

The difference of two sets refers to the elements of set `s` which are not in set `t`. In other words, it refers to the members of set `s` that are unique to set `s`. The difference is calculated using the AND operator (`&`) where the two operands are set `s` and the negation of set `t` (`~t`). The function then returns the set of elements in `s` that are not in `t`.

This function can be used to find the complement of a given set as well, in which case the first operand would be the universal set $\mathbb{U}$ and the second operand would be the set you want to complement as shown below.

$$\overline{s} = \{x | x \notin s\} = \mathbb{U} - s$$

Set set_complement(Set s)

This function is used to return the complement of a given set. By complement we mean that all bits in the set are flipped using the NOT operator. Thus, the set that is returned contains all the elements of the universal set $\mathbb{U}$ that are not in `s` and contains none of the elements that are present in `s`.

3 Sorting

Putting items into a sorted order is one of the most common tasks in Computer Science. As a result, there are a myriad of library routines that will do this task for you, but that does not absolve you of the obligation of understanding how it is done. In fact, it behooves you to understand the various algorithms in order to make wise choices.

The best execution time that can be accomplished, also referred to as the *lower bound*, for sorting using *comparisons* is $\Omega(n \log n)$, where n is the number is elements to be sorted. If the universe of elements to be sorted is small, then we can do better using a *Count Sort* or a *Radix Sort* both of which have a time complexity of $O(n)$. The idea of *Count Sort* is to count the number of occurrences of each element in an array. For *Radix Sort*, a digit by digit sort is done by starting from the least significant digit to the most significant digit.

What is this O and Ω stuff? It's how we talk about the execution time (or space used) by a program. We will discuss it in lecture and in section, and you will see it again in your Data Structures and Algorithms class, now named CSE 101.

You will be implementing at least four sorting algorithms from these five: Insertion Sort, Shell Sort, Heap Sort, recursive Quicksort, and Batcher Sort. **That is, you get to choose which four algorithms you want to implement from among those five.** The purpose of this assignment is to get you fully familiarized with each sorting algorithm and for you to get a *feel* for computational complexity. These are well-known sorts. You can use the Python pseudocode provided to you as guides. Do not get the code for the sorts from the Internet or you will be referred to for cheating. We will be running plagiarism checkers.

4   Insertion Sort

Insertion Sort is a sorting algorithm that considers elements one at a time, placing them in their correct, ordered position. It is so simple and so ancient that we do not know who invented it. Assume an array of size n with elements A[0] through A[n-1]. For each k in increasing value from $1 \le k \le n - 1$, Insertion Sort compares the k-th element with each of the preceding elements in descending order until its position is found. Assume we're sorting an array A in increasing order. We start from $A[k]$ and check if it is in the correct order by comparing it with the element $A[k-1]$. There are two possibilities at this point:

1. $A[k]$ **is in the right place.** This means that $A[k]$ is greater or equal to $A[k-1]$, and thus we can move onto sorting the next element.

2. $A[k]$ **is in the wrong place.** Thus, $A[k-1]$ is shifted up to $A[k]$, and the original value of $A[k]$ is further compared to $A[k-2]$. Intuitively, Insertion Sort simply shifts elements back until the element to place in sorted order is slotted in.

Below is Python code for Insertion Sort. Remember that the upper limit of the Python range() function is "noninclusive," meaning that, for example, the statement `for k in range(0, 4)` will result in k receiving values 0, 1, 2, and 3.

You will be converting this Python code into C. Since part of this assignment is reporting the number of moves and comparisons that each of the sort algorithms performs, use the move() function of Section 9 in the three places where moves are indicated, and use the cmp() function where a comparison is needed.

```
─────────────────── Insertion Sort in Python ───────────────────
def insertion_sort(A: list):
    # Sort values in A[0] through A[len(A) - 1]
    for k in range(1, len(A)):
        # k will take on values from 1 through len(A) - 1.
        # A[0] through A[k - 1] already are sorted.
        # Shift values until A[0] through A[k] are sorted.

        # Example with k == 3, so A[0] through A[2] already are sorted:
        # A[0] A[1] A[2] A[3]    temp
        #  20   40   50   30      --
        #  20   40   50   --      30     temp = A[3]
        #  20   40   --   50      30     A[3] = A[2]
        #  20   --   40   50      30     A[2] = A[1]
        #  20   30   40   50      --     A[1] = temp

        j = k
        temp = A[k]                       # move A[k]
        while j >= 1 and temp < A[j - 1]:  # compare temp and A[j - 1]
            A[j] = A[j - 1]               # move A[j - 1]
            j -= 1
        A[j] = temp                       # move temp
```

5 Shell Sort

> *There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*
>
> —C.A.R. Hoare

Donald L. Shell (March 1, 1924–November 2, 2015) was an American computer scientist who designed the Shell sort sorting algorithm. He earned his Ph.D. in Mathematics from the University of Cincinnati in 1959, and published the Shell Sort algorithm in the Communications of the ACM in July that same year[1].

Shell Sort is a variation of insertion sort, which sorts pairs of elements which are far apart from each other. The *gap* between the compared items being sorted is continuously reduced. Shell Sort starts with distant elements and moves out-of-place elements into position faster than a simple nearest neighbor exchange. What is the expected time complexity of Shell Sort? It depends entirely upon the gap sequence.

The following is the pseudocode for Shell Sort. The gap sequence is represented by the array `gaps`. You will be given a gap sequence, the Pratt sequence ($2^p 3^q$ also called 3-smooth), in the header file `gaps.h`. For each `gap` in the gap sequence, the function compares all the pairs in `A` that are `gap` indices away from each other. Pairs are swapped if they are in the wrong order.

```
────────── Shell Sort in Python ──────────

def shell_sort(A: list):
    # Sort values in A[0] through A[len(A) - 1]
    for gap in gaps:
        # gap will take on a series of non-contiguous, decreasing values ending with 1.
        # (We provide you with an array of these "gap" values.)
        for k in range(gap, len(A)):
            # k will take on values from gap through len(A) - 1.
            # Sort several interleaved lists, one list for each value of gap.

            # For example, when gap == 5:
            # The list ending    ..., A[k-15], A[k-10], A[k-5] already is sorted.
            # Shift values until ..., A[k-15], A[k-10], A[k-5], A[k] are sorted.
            # (Compare these statememnts to those of insertion_sort().)

            j = k
            temp = A[k]                        # move A[k]
            while j >= gap and temp < A[j - gap]:  # compare temp and A[j - gap]
                A[j] = A[j - gap]              # move A[j - gap]
                j -= gap
            A[j] = temp                        # move temp
```

6 Heapsort

> *Increasingly, people seem to misinterpret complexity as sophistication, which is baffling – the incomprehensible should cause suspicion rather than admiration.*
>
> —Niklaus Wirth

Heapsort, along with the heap data structure, was invented in 1964 by J. W. J. Williams[2]. The heap data structure is typically implemented as a specialized binary tree. There are two kinds of heaps: *max* heaps and *min* heaps. In a max heap, any parent node *must* have a value that is greater than or equal to the
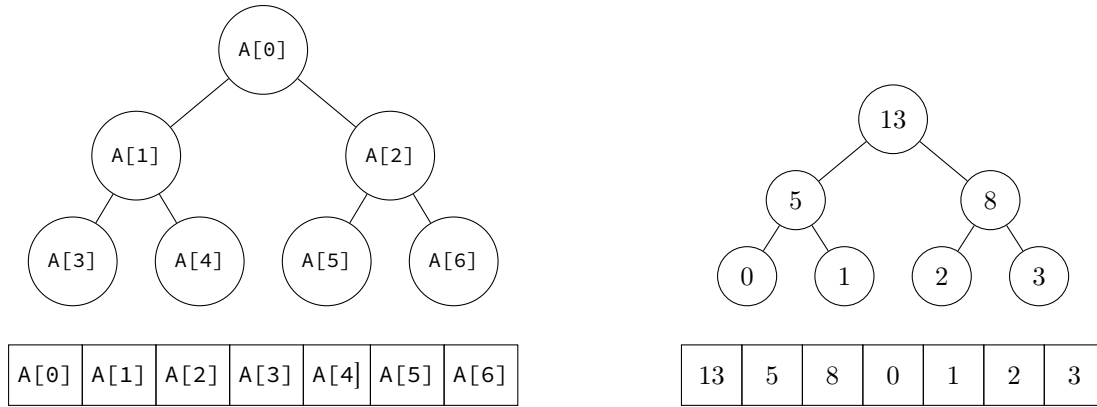
Figure 1: A max heap and its array representation. Children of any node at index k are at indexes $2k + 1$ and $2k + 2$. The parent of any non-root node k is at index $\lfloor \frac{k-1}{2} \rfloor$.

values of its children. For a min heap, any parent node *must* have a value that is less than or equal to the values of its children. The heap is typically represented as an *array*, in which for any parent at index k, the index of its left child is $2k + 1$ and the index of its right child is $2k + 2$. Also, the parent of any non-root node at index $k > 0$ has index $\lfloor \frac{k-1}{2} \rfloor$.

Look at the example in Fig. 1. The root at $k = 0$ is A[0]. It has two children at A[1] and A[2], as is predicted by the equations above, which say that for $k = 0$, the left child is $2k + 1 = 1$ and the right child is $2k + 2 = 2$. Similarly, the children of the node at $k = 1$ are at $2k + 1 = 3$ and $2k + 2 = 4$.

Hearing the name Heapsort, you would be correct to guess as that it utilizes a heap to sort elements. Heapsort sorts its elements using two routines that (1) build a heap, (2) fix a heap.

1. **Building a heap.** The first routine starts with an array that we want to sort and builds a heap from it. This means ordering the array elements such that they obey the constraints of a max or min heap. For our purposes, the constructed heap will be a *max heap*. This means that the largest element, the root of the heap, is the first element of the array from which the heap is built.

2. **Fixing a heap.** The second routine is needed as we sort the array. The gist of Heapsort is that the largest array elements are repeatedly removed from the top of the heap and placed at the end of the sorted array (if the array is to be sorted in increasing order). After removing the largest element from the heap, the heap needs to be *fixed* so that it once again obeys the constraints of a heap.

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

def fix_heap(A: list, first: int, last: int):
    done = False
    parent = first

    # Move the parent down until the max-heap condition is met.
    while 2 * parent + 1 <= last and not done:
        # parent has at least one child
        largest_child = max_child(A, parent, last)
        if A[parent] < A[largest_child]:

            # This Python code swaps parent and largest_child.
            A[parent], A[largest_child] = A[largest_child], A[parent]

            parent = largest_child
        else:
            done = True
```

```
─────────────────── Heapsort in Python ───────────────────
def build_heap(A: list, first: int, last: int):
    if last > 0:
        for parent in range((last - 1) // 2, first - 1, -1):
            fix_heap(A, parent, last)

def heap_sort(A: list):
    first = 0
    last = len(A) - 1
    build_heap(A, first, last)
    for leaf in range(last, first, -1):
        # Visit all non-root nodes starting with the last.
        # That is, the range() above ensures that leaf never will equal first.

        # This Python code swaps first and leaf.
        A[first], A[leaf] = A[leaf], A[first]

        fix_heap(A, first, leaf - 1)
```

7 Quicksort

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

—Edsger Dijkstra

Quicksort (sometimes called partition-exchange sort) was developed by British computer scientist C.A.R.

"Tony" Hoare in 1959 and published in 1961[3]. It is perhaps the most commonly used algorithm for sorting (by competent programmers). When implemented well, it is the fastest known algorithm that sorts using *comparisons*. It is usually two or three times faster than its main competitors, Merge Sort and Heapsort. It does, though, have a worst case performance of $O(n^2)$ while its competitors are strictly $O(n \log n)$ in their worst case.

Quicksort is a divide-and-conquer algorithm. It partitions arrays into two sub-arrays by selecting an element from the array and designating it as a pivot. Elements in the array that are less than the pivot go to the left sub-array, and elements in the array that are greater than or equal to the pivot go to the right sub-array.

Note that Quicksort is an *in-place* algorithm, meaning it doesn't allocate additional memory for sub-arrays to hold partitioned elements. Instead, Quicksort utilizes a subroutine called `partition()` that places elements less than the pivot into the left side of the array and elements greater than or equal to the pivot into the right side and returns the index that indicates the division between the partitioned parts of the array. Quicksort is then applied recursively on the partitioned parts of the array, thereby sorting each array partition containing at least one element. Like with the Heapsort algorithm, the provided Quicksort pseudocode operates on 1-based indexing, subtracting one to account for 0-based indexing whenever array elements are accessed.

──────── Partition in Python ────────

```python
def partition(A: list, lo: int, hi: int):
    # Use element hi as the pivot.
    # Divide the subarray into two partitions.
    i = lo - 1
    for j in range(lo, hi):
        # j takes values from lo to hi - 1
        if A[j] < A[hi]:
            i += 1

            # Swap elements i and j.
            A[i], A[j] = A[j], A[i]

    i += 1

    # Swap the pivot and element i.
    A[i], A[hi] = A[hi], A[i]

    # Return where the pivot is now.
    return i
```

──────── 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, 0, len(A) - 1)
```

8 Batcher's Odd-Even Merge Sort

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

—C.A.R. Hoare

Batcher's odd-even mergesort, or Batcher's method[4], is unlike the other presented sorts in that it is actually a *sorting network*. What is a sorting network? Sorting networks, or *comparator networks*, are circuits built for the express purpose of sorting a set number of inputs. Sorting networks are built with a fixed number of wires, one for each input to sort, and are connected using *comparators*. Comparators compare the values traveling along the two wires they connect and swap the values if they're out of order. Since there are a fixed number of comparators, a sorting network must necessarily sort any input using a fixed number of comparisons. Refer to Figure 2 for a diagram of a sorting network using Batcher's method.

Sorting networks are typically limited to inputs that are powers of 2. Batcher's method is no exception to this. To remedy this, we apply Knuth's modification to Batcher's method to allow it sort arbitrary-size inputs[5]. This modification on Batcher's method is also referred to as the *Merge Exchange Sort*. You will be implementing the merge exchange sort, or Batcher's method, using the provided Python pseudocode.

Be aware that the Python code swaps two array elements using a "target list" in a single assignment statement: `A[x], A[y] = A[y], A[x]`. While C does not support such a feature directly, meaning that normally you would swap two values using a temporary variable and three assignment statements, instead your C code will be calling a swap function that we have provided (see Section 9).

───── Merge Exchange Sort (Batcher's Method) in Python ─────

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

The pseudocode for Batcher's method can be a little mysterious, but it effectively acts as a parallel *Shell*

Sort. Shell Sort acts a variation of Insert Sort and first sorts pairs of elements which are far apart from each other. The distance between these pairs of elements is called a *gap.* Each iteration of Shell Sort decreases the gap until a gap of 1 is used.

Batcher's method is similar, but instead of sorting pairs of elements that are a set gap apart, it *k-sorts* the even and odd subsequences of the array, where k is some power of 2. Given an array A where A_i denotes the i-th index of A, an even subsequence refers to the sequence of values $\{A_0, A_2, A_4, \dots\}$. Similarly, an odd subsequence refers to the sequence of values $\{A_1, A_3, A_5, \dots\}$. The topic of k-sorting is beyond the scope of this course, but all that you are required to understand is that if an array is k-sorted, then all its elements are at most k indices away from its sorted position.

Consider an array of 16 elements. Batcher's method first 8-sorts the even and odd subsequences, then 4-sorts them, then 2-sorts them, and finally 1-sorts them, *merging* the sorted even and odd sequences. We will call each level of k-sorting a *round.* As stated previously, Batcher's method works in *parallel.* By virtue of the algorithm, any indices that appear in a pairwise comparison when k-sorting a subsequence do not appear as indices in another comparison during the same round. A clever use of the bitwise AND operator, `&`, guarantees this property.

When computing the bitwise AND operator of two integers x and y, the resulting integer is composed of 0-bits except in the positions where both x and y have a 1-bit. As an example, let $x = 10 = 1010_2$ and $y = 8 = 1000_2$. Bitwise AND-ing x and y yields $z = 8 = 1000_2$. In the provided pseudocode, the variable `p` tracks the current round of k-sorting. It is always a power of 2 since it starts off as a power of 2, and is only halved in value using the right-shift operator (`>>`). The condition on line 20 of the pseudocode first computes the bitwise AND of `i` and `p`. This effectively partitions values of `i` into partitions of size `p`. The variable `r` effectively represents which partitions can be considered for comparison. Thus, `(i & p) == r` checks if the partition that `i` falls into is eligible for comparison, and if it is, to execute the comparison.

Indices are only ever compared with indices that are `d` indices away. Since `p` is a power of 2 and `d` is an odd multiple of `p`, it follows that `i & p != (i + d) & p` for any `i`. This means there is no overlap with any of the pairs of indices, which therefore means that these comparisons can be run simultaneously, or in *parallel,* with no ill effect. These parallel comparisons are shown clearly in Figure 2.

Although Batcher's method can be run in parallel, your implementation of the sort will run *sequentially,* sorting the input over several rounds. For an array size of n, the initial value to k-sort with is $k = \lceil \log_2(n) \rceil$. The even and odd subsequences are first k-sorted, then $\frac{k}{2}$-sorted, then $\frac{k}{4}$-sorted, and so on until they are 1-sorted. You will want to edit the provided pseudocode to print out the pairs of indices that are being compared to see what is happening.
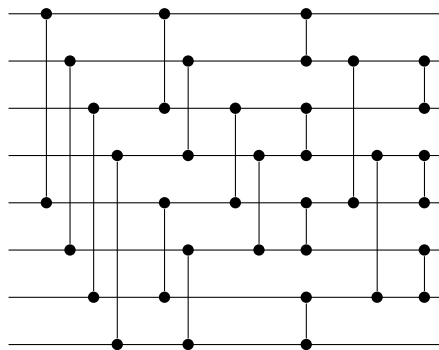

Figure 2: Batcher's odd-even mergesort sorting network with eight inputs. Inputs traveling along the wires are sorted as they move from left to right.

9   Statistics

> *There are three types of lies—lies, damn lies, and statistics.*
>
> —Benjamin Disraeli

To facilitate the gathering of statistics, you will be given, *and must use*, a small statistics module, which you will find defined in `stats.h` and implemented in `stats.c`. The module itself revolves around the following struct:

```
typedef struct {
  uint64_t moves;
  uint64_t comparisons;
} Stats;
```

The module also includes functions to *compare*, *swap*, and *move* elements.

int cmp(Stats *stats, int x, int y)

Compares x and y and increments the `comparisons` field in `stats`. Returns `-1` if x is less than y, `0` if x is equal to y, and `1` if x is greater than y.

int move(Stats *stats, int x)

"Moves" x by incrementing the `moves` field in `stats` and returning x. This is intended for use in Insertion Sort and Shell Sort, where array elements aren't swapped, but instead moved and stored in a temporary variable.

void swap(Stats *stats, int *x, int *y)

Swaps the elements pointed to by pointers x and y, incrementing the `moves` field in `stats` by 3 to reflect a swap using a temporary variable.

void reset(Stats *stats)

Resets `stats`, setting the `moves` field and `comparisons` field to 0. It is possible that you don't end up using this specific function, depending on your usage of the `Stats` struct.

void print_stats(Stats *stats, const char *algorithm_name, uint8_t num_elements)

Print the statistics for `stats`, listing the specified algorithm name and number of elements. You will use this function as part of generating the output, as mentioned in Section 12.

10   Your Task

Your task for this assignment is as follows:

1. We provided you with a header file `set.h`. Within that header file are function prototypes for a collection of Set functions. Create a file `set.c` that implements the Set functions. (Normally you will compile your project by typing `make`, but if you want to check the syntax of just `set.c`, type the command `make set.o` .)

2. Choose *four* of the *five* sorting algorithms. Create a file for all five algorithms. For each of the four algorithms that you chose, implement its associated sorting algorithm in **C**, based on the algorithm's Python pseudocode. For the last algorithm, return the array unmodified. You may also chose to implement all 5 sorts.

- `insert.c`: Insertion sort
- `shell.c`: Shell Sort
- `heap.c`: Heapsort
- `quick.c`: Quicksort
- `batcher.c`: Batcher Sort (Batcher's method)

The interface for these sorts are given in the header files `insert.h`, `shell.h`, `heap.h`, `quick.h`, and `batcher.h`. You are not allowed to modify these header files for any reason.

3. Create file `sorting.c`. Within this file, implement a test harness for your sorting algorithms. In your test harness, you will create an array of pseudorandom elements and test each of the sorts.

4. We provide you with files `stats.c` and `stats.h`. Your sorting algorithms should perform comparison, move, and swap operations by calling functions that are defined in these files. Gather statistics about each sorting algorithm and its performance. The statistics you will gather are the *size* of the array, the number of *moves* required, and the number of *comparisons* required. Note: a comparison is counted only when two array elements are compared.

Your test harness must support all of the following command-line options. `-a -i, -s, -h, -q, and -b` , but only 4 of the sorts need to work. For the sorting algorithm that you choose not to implement, you must leave the elements unsorted and return with no errors.

- `-a` : Employs *all* sorting algorithms that your implemented.

- `-i` : Enables Insertion Sort.

- `-s` : Enables Shell Sort.

- `-h` : Enables Heap Sort.

- `-q` : Enables Quicksort.

- `-b` : Enables Batcher Sort.

- `-r seed` : Set the random seed to `seed`. The *default* seed should be 13371453.

- `-n size` : Set the array size to `size`. The *default* size should be 100.

- `-p elements` : Print out `elements` number of elements from the array. The *default* number of elements to print out should be 100. If the size of the array is less than the specified number of elements to print, print out the entire array and nothing more. You should be able to specify `-p 0` to suppress printing of the elements.

- `-H` : Prints out program usage. See reference program for example of what to print.

It is important to read this *carefully*. None of these options are *exclusive* of any other (you may specify any number of them, including *zero*). The most natural data structure for this problem is a *set* (see Section 2). If you chose not to implement a sorting algorithm, you must still create an option for it in main, and call it. Remember that if you chose not to implement a sorting algorithm, it must not modify the array passed in.

11 Testing

- You will test each of the sorts specified by command-line option by sorting an array of pseudorandom numbers generated with `random()`. Each of your sorts should sort the *same* pseudorandom array. Hint: make use of `srandom()`.

- The pseudorandom numbers generated by `random()` should be *bit-masked* to fit in *30* bits. Hint: use bit-wise AND (`&`).

- Your test harness *must* be able to test your sorts with array sizes *up to the memory limit of the computer*. That means that you will need to dynamically allocate the array.

- Your program should have no *memory leaks*. Make sure you `free()` before exiting. `valgrind` should pass cleanly with any combination of the specified command-line options.

- Your algorithms *must* correctly sort. Any algorithm that does not sort correctly will receive a *zero*.

A large part of this assignment is understanding and comparing the performance of various sorting algorithms. You essentially conducting an experiment. As stated in §10, you *must* collect the following statistics on each algorithm:

- The *size* of the array,

- The number of *moves* required (each time you transfer an element in the array, that counts), and

- The number of *comparisons* required (comparisons *only* count for *elements*, not for logic).

12   Output

> *Books are not made to be believed, but to be subjected to inquiry. When we consider a book, we mustn't ask ourselves what it says but what it means.*
>
> —Umberto Eco

The output your test harness produces *must* be formatted like in the following examples. You can use `print_stats()` when printing the lines that show the algorithm name, number of elements, and the counts of the moves and comparisons. Note that our reference design implements all five of the sorting algorithms, and so its `-a` option shows *five* sort results. Your program may or may not implement all of the sorting algorithms, but if it doesn't, using `-a` must still run all of the algorithms.

```
$ ./sorting -q -n 1000 -p 0
Quick Sort, 1000 elements, 18642 moves, 10531 compares
$ ./sorting -h -n 15 -p 0
Heap Sort, 15 elements, 144 moves, 70 compares
./sorting -a -n 15
Insertion Sort, 15 elements, 82 moves, 65 compares
      34732749      42067670      54998264     102476060     104268822
     134750049     182960600     538219612     629948093     783585680
     954916333     966879077     989854347     994582085    1072766566
Heap Sort, 15 elements, 144 moves, 70 compares
      34732749      42067670      54998264     102476060     104268822
     134750049     182960600     538219612     629948093     783585680
     954916333     966879077     989854347     994582085    1072766566
Shell Sort, 15 elements, 170 moves, 87 compares
      34732749      42067670      54998264     102476060     104268822
     134750049     182960600     538219612     629948093     783585680
     954916333     966879077     989854347     994582085    1072766566
Quick Sort, 15 elements, 135 moves, 51 compares
      34732749      42067670      54998264     102476060     104268822
     134750049     182960600     538219612     629948093     783585680
     954916333     966879077     989854347     994582085    1072766566
Batcher Sort, 15 elements, 90 moves, 59 compares
      34732749      42067670      54998264     102476060     104268822
     134750049     182960600     538219612     629948093     783585680
     954916333     966879077     989854347     994582085    1072766566
```

When printing the array elements, create a table with five columns. Each array element should be printed with a width of 13. You should make use of the following `printf()` statement.

```
printf("%13d", A[i]);
```

13   Deliverables

> *Dr. Long, put down the Rilke and step away from the computer.*
>
> —Michael Olson

You will need to turn in the following source code and header files:

1. Your program *must* have your source files.

   - `sorting.c` contains `main()` and *may* contain any other functions necessary to complete the assignment.
   - `set.c` implements bit-wise Set operations.

2. Your program also *must* have *all five* of these five source files:

   - `heap.c` implements Heap Sort.
   - `batcher.c` implements Batcher Sort.
   - `shell.c` implements Shell Sort.
   - `quick.c` implements recursive Quicksort.
   - `insert.c` implements Insertion Sort.

   Even if you chose not to write one of the sorts, the file must be present in your function, and it must compile. The array that is returned by the function does not need to be sorted, nor does the number of moves or compares need to be accurate. If you chose to not implement a sort, your code must not modify the array passed to it.

3. Your program also *must* have these source files, which we have provided:

   - `batcher.h` specifies the interface to `batcher.c`.
   - `shell.h` specifies the interface to `shell.c`.
   - `gaps.h` provides a gap sequence to be used by Shell sort.
   - `heap.h` specifies the interface to `heap.c`.
   - `quick.h` specifies the interface to `quick.c`.
   - `set.h` specifies the interface to `set.c`.
   - `stats.c` implements the statistics module.
   - `stats.h` specifies the interface to the statistics module.

You can have other source and header files, but *do not try to be overly clever*. The header files for each of the sorts are provided to you and may not be modified. Each sort function takes a pointer to a `Stats` struct, the array of `int`s to sort as the first parameter, and the length of the array as the second parameter. You will also need to turn in the following:

1. Makefile:

   - `CC = clang` must be specified.
   - `CFLAGS=-Wall -Wextra -Wstrict-prototypes -Werror -pedantic` must be specified.

- make must build the sorting executable, as should make all and make sorting.
- make clean must remove all files that are compiler generated.
- make format should format all your source code, including the header files.

2. report.pdf: The report has a few sections: The usage must describe how to use your program and Makefile. It should also list and explain any command-line options that your program accepts. Any false positives reported by scan-build should be documented and explained here as well. Note down any known bugs or errors in this file as well for the graders. The design must describe your design and design process for your program with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode. The results section must include the following

- What you learned from the different sorting algorithms. Under what conditions do sorts perform well? Under what conditions do sorts perform poorly? What conclusions can you make from your findings?
- Tables or Graphs explaining the performance of the sorts on a variety of inputs, such as arrays in reverse order, arrays with a small number of elements, and arrays with a large number of elements. You will find it helpful to write a script to handle the plotting. (The program awk will be helpful for parsing the output of your program. Visit office hours for more information.)
- Analysis of the graphs you produce. Here is an example graph produced by gnuplot with a different set of sorts for reference (axes are log-scaled):
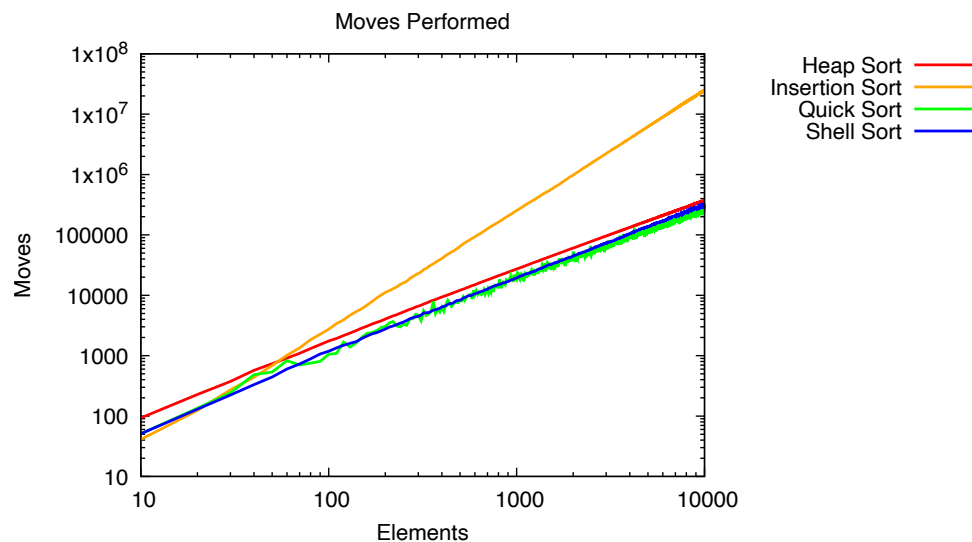
You should look carefully at any graphs that you produce. Are all of the lines smooth? For example, in this graph there are some features of interest. What could be causing features that appear in your own graphs?

14 Submission

For the report draft, you must submit a commit ID on canvas *before Friday November 3rd at 11:59 PST*. You must have a PDF called report.pdf.

For the final version, you must submit a commit ID to canvas before Sunday November 5th. Not doing so will result in using late days.

Refer back assignment 0 for the instructions on how to properly submit your assignment through `git`. Remember: *add*, *commit*, and *push*!

Your assignment is turned in *only* after you have pushed and submitted the commit ID you want graded on Canvas. "I forgot to push" and "I forgot to submit my commit ID" are not valid excuses. It is *highly* recommended to commit and push your changes *often*.

Notes:

1. Your code must be formatted using the `make format` command in the provided makefile.

2. Your code must build using the `make` or `make all` commands in the provided makefile

3. You may not modify any header files.

4. You should clone the resources repository, which contains all required files, and use `cp` to move files to your personal repository.

5. Do not push any object files or binaries. You can use `make clean` before you push to ensure this.

15   Supplemental Readings

- *The C Programming Language* by Kernighan & Ritchie

  - Chapter 1 §1.10
  - Chapter 3 §3.5
  - Chapter 4 §4.10–4.11
  - Chapter 5 §5.1–5.3 & 5.10

- *C in a Nutshell* by T. Crawford & P. Prinz.

  - Chapter 6 – Example 6.5 – Nested loops in the bubble-sort algorithm (p. 105)
  - Chapter 7 – Recursive Functions (pp. 126–127)
  - Chapter 8 – Arrays as Arguments of Functions (pp. 138–140)
  - Chapter 9 – Array Pointers (pp. 152–153)

16   Revisions

Version 1 Original.

References

[1] D. L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.

[2] J. W. J. Williams. Algorithm 232 – heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[3] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[4] Ken Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring), Atlantic City, New Jersey*, AFIPS, pages 307–314, New York, NY, 1968. Association for Computing Machinery.

[5] D.E. Knuth. *Section 5.3.4: Networks for Sorting*, pages 219–247. Addison-Wesley, Boston, MA, 1998.