Asignment 3: Putting your affairs in order - Design

Thadeus Ballmer

October 2021

1 Introduction

In assignment 3, multiple sorting algorithms are implemented and compared including: insertion sort, shell sort, heapsort, and quicksort. The test harness sorting.c links all these sorts together and takes several CLI options for testing.

2 Sorting Algorithms

2.1 Statistics

All sorting algorithms must produce statistics on their operations because the purpose of this assignment is to compare the speed and efficiency of these algorithms. The given **stats.c** file contains several functions to use in each algorithm to keep track of operations including: comparing two values, moving a value, swapping two values, and resetting statistics.

```
int cmp(Stats *stats, uint32_t x, uint32_t y) {
    stats->compares += 1;
    if (x < y) {
        return -1;
    } else if (x > y) {
        return 1;
    } else {
        return 0;
    }
}

uint32_t move(Stats *stats, uint32_t x) {
    stats->moves += 1;
    return x;
}

void swap(Stats *stats, uint32_t *x, uint32_t *y) {
```

```
    stats->moves += 3;
    uint32_t t = *x;
    *x = *y;
    *y = t;
}

void reset(Stats *stats) {
    stats->moves = stats->compares = 0;
}
```

2.2 Insertion Sort

Insertion sort is the most basic and intuitive way to sort through an array—although inefficient. Insertion sort compares every element with each of the preceding elements until the correct placement is found. While **insert.h** includes the correct declaration of **insertion_sort()**, the included assignment 3 pseudocode matched with the proper stats functions define the structure of insertion sort:

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

2.3 Shell Sort

Shell sort is an alteration of insertion sort. Instead of comparing each element to its neighbors directly, two elements are compared a set *gap* away from each other. This *gap* changes in a specified order based on the current iteration of each sorting sequence. The given Python pseudocode presents a slight challenge by including a yield statement. This is easily solved by using a variable to save the current *gap* calculation, then calculating a new *gap* when needed.

```
def gaps(n: int):
    for i in range(in(log(3 + 2 * n) / log(3)), 0, -1):
        yield (3**i -1) // 2

def shell_sort(A: list):
    for gap in gaps(len(A)):
        for i in range(gap, len(A)):
            j = i
```

```
            temp = move(stats, A[i])
            while j >= gap and cmp(stats, A[j - gap], temp) == 1:
                A[j] = move(stats, A[j - gap])
                j -= gap
            A[j] = move(stats, temp)
```

2.4 Heapsort

Heapsort utilizes the heap data structure in order to sort an array. Before the Heapsort algorithm may begin or continue, the heap data structure needs to be *built* from the given or remaining data. In this case, our algorithm uses *max* heap which means that each parent node must have a value greater than or equal to its children. While iterating through Heapsort, elements are removed whenever their position is found, so the heap needs to be *fixed* to match the *max* heap constraints. This program structure requires multiple functions to implement efficiently.

```
def max_child(A: list, first: int, last: int):
    left = 2 * first
    right = left + 1
    if right <= last and cmp(stats, A[right - 1], A[left - 1]) == 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 cmp(stats, A[great - 1], A[mother - 1]) == 1:
            swap(stats, A[mother - 1], A[great - 1])
            mother = great
            great = max_child(A, mother, last)
        else:
            found = True

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):
        swap(stats, A[first - 1], A[leaf - 1])
        fix_heap(A, first, leaf - 1)
```

2.5 Quicksort

Quicksort is a recursive *divide-and-conquer* algorithm. It first selects a pivot point. Then it places values greater than the pivot to the right of the pivot and values less than the pivot to the left. Once successful, it repeats this process on each side through recursion until the data is sorted.

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

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))
```

3 Testing

sorting.c is the test harness for the 4 sorting algorithms. The test harness will take the following inputs:

- -a: Employs *all* sorting algorithms.
- -e: Enables Heap Sort.
- -i: Enables Insertion Sort.
- -s: Enables Shell Sort.
- -q: Enables Quicksort.

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

Each option may be activated independent of other options.

3.1 Sets

Sets must be used to keep track of CLI options. **set.h** defines vaious set functions but the functions applicable to **sorting.c** are **empty_set()**, **member_set()**, **delete_set()** and **insert_set()**. These functions work by manipulating bits in various ways. Using sets by manipulating a few bits is more efficient than using booleans like in assignment 2.

3.2 Pseudocode

This pseudocode was developed from Eugene's section.

```
main()
    seed = 13371453
    size = 100
    elements to print = 100
    while (Use getopt to loop through options)
        switch (opt)
        case: 'a'
            insert_set(INSERTION)
            insert_set(SHELL)
            insert_set(HEAP)
            insert_set(QUICK)
        case: 'e'
            insert_set(HEAP)
        case: 'i'
            insert_set(INSERTION)
        case: 's'
            insert_set(SHELL)
        case: 'q'
            insert_set(QUICK)
        case: 'r'
            seed = optarg
```

```
case: 'n'
    size = optarg
case: 'p'
    elements to print = optarg
case: 'h'
    insert_set(USAGE)

if (member_set(USAGE))
    print(Usage information)
    empty_set()

dynamical allocate array numbers of size
bitmask each array index to fit in 30 bits

if (member_set(HEAP))
    dynamically allocate an array of size
    copy numbers into heap array
    heap_sort(stats, array, size)
    print(Heap sort, size , moves, compares)
    for (Loop through number of elements to print)
        print(elements to print)
    free memory
    reset(stats)

if (member_set(SHELL))
    dynamically allocate an array of size
    copy numbers into heap array
    shell_sort(stats, array, size)
    print(Shell sort, size , moves, compares)
    for (Loop through number of elements to print)
        print(elements to print)
    free memory
    reset(stats)

if (member_set(INSERTION))
    dynamically allocate an array of size
    copy numbers into heap array
    insertion_sort(stats, array, size)
    print(Insertion sort, size , moves, compares)
    for (Loop through number of elements to print)
        print(elements to print)
    free memory
```

```
    reset(stats)

if (member_set(QUICK))
    dynamically allocate an array of size
    copy numbers into heap array
    quick_sort(stats, array, size)
    print(Insertion sort, size , moves, compares)
    for (Loop through number of elements to print)
        print(elements to print)
    free memory
    reset(stats)

free memory
```