Sorting: Putting your affairs in order - CSE13S Assignment 3

Nicholas Reis

October 12, 2021

Program Description

This program will implement a total of four sorting algorithms, including insertion sort, heap sort, shell sort, and quick sort. Each algorithm takes in a structure called stats, an array, and the size of the array as parameters, and will count the number of compares and moves that occur during the sorting process.

Test Harness

Within sorting.c, there will be code that supports command line options for each of the functions written in this assignment. I utilize sets, enums, getopt(), and a switch statement in order to take in options from the user and then run specific algorithms that were selected. The test harness also includes the use of srandom() and seeds in order to generate a specific amount of pseudorandom numbers to be added to an array that is then used in each algorithm.

Pseudocode

```
include all function header files
include inttypes.h, stdio.h, stdlib.h, unistd.h

define OPTIONS "aeisqhr:n:p:"

create a static uint32_t bit_mask variable to fit in 30 bits
create enums for all the sorts and options
create an array to store the names of the sorts
set elements and size to 100 default
set seed to 13371453 default

create a function print_help() to print out all of the help information
```

```
return void

create a function reset_array() to reset the array to its initial values
arguments are sorted array A and an array B that stored the original values
    for loop that iterates through each index:
        set A[i] = B[i]
return void

main:

initialize a Stats struct
set stats.moves and stats.compares to 0
create a Set an make it an empty set

while loop that uses getopt to access the options:
    switch:
    case 'a' complements the set and sets all bits to 1
    case 'eisq' inserts the specific enum into the set
    case 'rnp' inserts the specific enum into the set and
               accesses the int input by using atoi(optarg)
    case 'h', '?', and default inserts the HELP enum into the set

if statement checks if HELP is a member of the set or if the set is empty
    calls print_help()

uses srandom with seed as the argument

use calloc to allocate memory for array A
use calloc to allocate memory for array B

for loop that iterates until it reaches size of the array:
    creates a random number using random()
    sets the index of A[i] equal to (random number and bit_mask)
    sets B[i] equal to A[i]

for loop iterates through sorts enums:
    if enum is HEAP:
        heap_sort(stats, A, size)
    else if enum is SHELL:
        reset_array(A, B)
        shell_sort(stats, A, size)
    else if enum is INSERTION:
        reset_array(A, B)
        insertion_sort(stats, A, size)
    else if enum is QUICK:
        reset_array(A, B)
```

```
        quick_sort(stats, A, size)
    prints out the current sort, the size of the array,
               the number of moves, and number of compares
    for loop iterates until i < elements and i < size
        if 5 number have been printed
            print a new line
        print out each number with 13 spaces allocated
    print a new line
free memory allocated to arrays
return void
```

Insertion Sort

Within insert.c, there is code that sorts a given array of length n using the insertion sort algorithm. This algorithm begins with the second element of the array and checks whether or not it is less than the first element. If it is, it replaces it and moves on to check the next number, otherwise it just moves on to the third element. It checks if this element is smaller than the previous, and if it is, it shifts the second element into its place and then checks the first number. Essentially it isolates an element and checks it against the previous elements, shifting it into the correct position from smallest to largest number. Using the following python pseudocode from the assignment 3 pdf (credit to Professor Long for this), I was able to write this program.

Pseudocode

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
```

Figure 1: Insertion Pseudocode

- The for loop in this pseudocode iterates from 1 to the length of the array minus 1 (range in python stops before it reaches the second value). The code begins with j = 1 because the algorithm needs to begin with the second index of the array, and because C is 0-based, index 1 is the second element of the array. We store the value of A[i] in temp so that we can place the selected number in the right spot at the end of the for loop. The while loop will decrement j each iteration to work backwards and check the previous numbers, and only runs while j is greater than 0 and while

temp (A[i]) is smaller than the previous number. Basically, the while loop moves the value of A[j - 1] to A[j] if it is a larger number than A[j]. After the while loop breaks, the original number is then put into the correct spot at A[j].

- To keep track of compares within this algorithm, I replaced the second condition of the while loop (temp < A[j - 1]) with the cmp() function defined in stats.h, as this is the spot in the program where two elements of the array are being compared to each other before moving them to different locations. I kept track of the moves in the lines (temp = A[i]), (A[j] = A[j - 1]), and (A[j] = temp), because this is where the program actually performs the action of moving the values to a different spot. To do this, I used the move() function also defined in stats.h.

Shell Sort

Within shell.c, there is code that sorts a given array of length n using the shell sort algorithm. This algorithm utilizes gaps to compare each number and place it in the correct location. The way the gap is calculated is by using the formula 3^i - 1 // 2, with i being the value of log(3 + 2 * n) / log(3), and with n being the size of the array. i decrements after each gap iterates through the entire array, with each value being checked at least once. The gap will determine the distance of the two indexes being compared, so if the gap size was 4, the algorithm would begin by comparing the value at index 0 and index 3, switch them if the value at 3 is smaller than the one at 0, and then move on to compare index 1 and index 4. Once each value has been compared once (aka the gap compares index x to the last index), the gap size decreases by 1 and begins from index 0 once again. This continues until the gap size is one and then finishes once it reaches the ending index. Using the following python pseudocode from the assignment 3 pdf (credit to Professor Long for this), I was able to write this program.

Pseudocode

- The gaps function in this psuedocode works as a generator, basically meaning that each time this function is called, it yields (returns) a different value. It calculates the gap that will be used by first setting the value of i to log(3 + 2 * n) / log(3), and then returning the value of 3^i - 1 // 2. The value of i will be an integer that is decremented each time the function is called until it is 0, in which it will end the loop and not return anything. The shell_sort function uses the largest gap first, does the sorting with it, and then uses the previous gap - 1. The nested for loop then does a similar process that was used in insertion sort, except it compares indexes that are gap distance away from each other, as opposed to next to each other.
- To keep track of compares within this algorithm, I replaced the second condition of the while loop (temp < A[j - gap]) with the cmp() function defined in stats.h, as this is the spot in the program where two elements of

Shell Sort in Python

```python
from math import log

def gaps(n: int):
    for i in range(int(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 = A[i]
            while j >= gap and temp < A[j - gap]:
                A[j] = A[j - gap]
                j -= gap
            A[j] = temp
```

Figure 2: Shell Pseudocode

the array are being compared to each other before moving them to different locations. I kept track of the moves in the lines (temp = A[i]), (A[j] = A[j - gap]), and (A[j] = temp), because this is where the program actually performs the action of moving the values to a different spot. To do this, I used the move() function also defined in stats.h.

Heap Sort

Within heap.c, there is code that sorts a given array of length n using the heap sort algorithm. This algorithm utilizes heaps to order an array by first building a heap (a max heap) out of the array, and then fixing the heap. My code builds a max heap, meaning that each parent node will be larger than or equal to the children nodes. Once a max heap is created, the first and last elements of the heap will be swapped, the largest element (that was the first element) is removed from the heap, and the heap must then be fixed once again to obey the requirements of a max heap. This occurs until the heap is down to the last element, meaning all other elements have been sorted. Using the following python pseudocode from the assignment 3 pdf (credit to Professor Long for this), I was able to write this program.

Pseudocode

Heap maintenance in Python

```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)
```

- The max_child() function tells the algorithm where the current parent node's children are located in the array, and then returns which one of them is the largest.
- The fix_heap() function takes in an int value an assigns it to mother. It then finds the max_child of mother, and if the max_child is less than the mother node, they are swapped. This continues all the way down the heap until it becomes a max heap, in which it stops swapping the nodes and the boolean value of found becomes true.
- The build_heap() function is run once and orders the given array so that it fits the constraints of a max heap. It does so by calling fix_heap() on the original array.
- The heap_sort() function first begins by building a max heap with the given array, a value of 1 for first, and the length of the array for last. It then will iterate through each index (decreasing) of the array and swap the first and last elements before fixing the heap and doing it over again until the for loop reaches the first index.
- To keep track of compares within this algorithm, I changed the section of max_child that compares (A[right - 1] > A[left - 1]) and the section of fix_heap that compares (A[mother - 1] < A[great - 1]) to instead use

the cmp() function defined in the stats.h file. To keep track of moves in this function, I replaced the section of fix_heap that swaps the values of (A[mother - 1] and A[great - 1]) and the section of heap_sort that swaps the values of (A[first - 1] and A[leaf - 1]) to instead use the swap() function also defined in the stats.h file.

Quick Sort

Within quick.c, there is code that sorts a given array of length n using the quick sort algorithm. This algorithm utilizes partitions to split the elements of an array into sections based on whether or not they are larger than a specified pivot element. If an element is less than the pivot, it moves to the right side of the array, and if it is larger than the pivot it moves to the right side of the array. The function then becomes recursive, as each half of the array is then put back into the quick sorter until each array partition is cut down to only one element left, meaning the array is sorted. Using the following python pseudocode from the assignment 3 pdf (credit to Professor Long for this), I was able to write this program.

Pseudocode

Partition in 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

```
# 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))
```

Figure 3: Quicksort Pseudocode

- The partition() function takes in an array and a low and high index as the parameters. The loop iterates from lo to hi - 1 (the pivot), and checks whether the first element is less than the last element of the array. If it is, it is swapped with the element at index i. This places the smaller values in the left side of the array. After the for loop ends, the pivot is placed at

index i, which will end up being to the right of all the elements smaller than it, and thus to the left of all the elements that are larger than it. The function returns the index + 1 of this final pivot location.
- The quick_sorter() function makes sure that the array can be sorted (that there are multiple elements in it) and then sets the value of p to the index of the pivot when it is partitioned. The function then recursively sorts each half of the partition separately, until it is only one element left in the partitions.
- The quick_sort() function takes in the array as a parameter and then calls the quick_sorter() helper function.
- To keep track of compares within this algorithm, I changed the section of partition() that checks (A[j - 1] < A[hi - 1]) to instead use the cmp() function defined in the stats.h file. To keep track of moves in this function, I replaced the section of partition() that swaps (A[j - 1] and A[i - 1]), along with (A[i] and A[hi - 1]) to instead use the swap() function also defined in the stats.h file.

Design Process

- Began by working on insert.c, utilized the pseudocode in the assignment 3 pdf to structure my program
- Did not do any testing once complete, instead began on test harness to make sure I could get the options down in code
- Went to Eugene's section on Tuesday, helped tremendously to understand each algorithm as well as how to use sets, stats, and dynamic memory allocation
- Wrote much of my test harness (specifically testing for insert.c first), still need to work on seed functionality and -n -p functionality for options
- Tested insert.c, made some changes and added the correct print statements to print out the elements
- Began working on shell.c, utilizing the pseudocode in the assignment 3 pdf to structure my program
- Used a static uint32_t variable to keep track of gaps
- Had some issues with typing of my variables (because of division), made some adjustments using temp variables and the floor() function from math.h to fix the problems
- Noticed that the nested for loop in the code resembles insertion sort, so I could put my move() and cmp() in the same spots
- Began working on heap.c, utilizing the pseudocode to structure my program
- Worked first try! Used the floor function from math.h for floor division and made sure to use cmp() and swap() functions to keep track of the compares and moves
- Began working on quick.c, utilizing the pseudocode to structure my program
- Spent a long time trying to figure out why my function wasn't sorting

properly and finally realized I had a swap() function inside a loop on accident
- Updated sorting.c with all functions and print statements to have everything formatted properly
- Kept getting a malloc() corrupted top size error when I tried to use -p option for values under 50
- Realized I was allocating the wrong amount of memory using calloc(), that change fixed the issue