Karthi Sankar
kasankar@ucsc.edu
10/18/21

CSE 13S Assignment 3 Design Document

Purpose:

The purpose of this program is to simulate 4 different sorting algorithms: insertion, shell, heap, and quick sort. We will do this in order to examine the effectiveness of these sorting algorithms for different numbers of elements and the time complexity of each of these algorithms. We will test each algorithm with a different amount of elements and different numbers. This program will teach us about these algorithms and how to analyze them.

Structure:

The program is set up with a test harness, as with the last assignment. Each of the sorts and corresponding helper functions will be contained in different .c files, and these files will be included in the test harness sorting.c. Sorting.c will contain the main function, which will call all the other sorting functions. Again, the functions will be run using command line options for each of the different files, user input, and the help menu. The main difference is that the functions will be triggered and run using a set. When a command line option is encountered, an enumeration will be inserted into the set. Then, when that enumeration is detected as a member of the set, the corresponding function will be run.

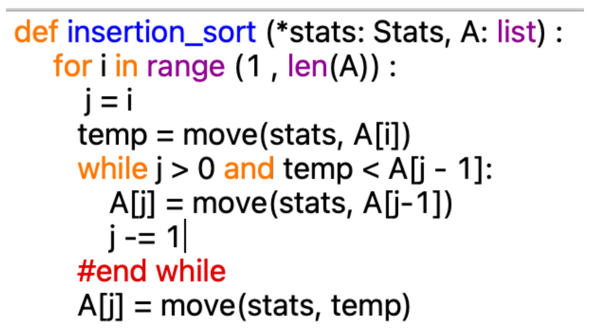
Description:

(pseudocode comes directly from assignment PDF + moves/compares/swaps added)

stats.c:

This file is given to us. It contains the implemented functions compare(), move(), swap(). Whenever any of the three functions are used in the separate sort files, moves and comparisons (in the Stats struct) will be incremented. Using compare() will increment comparisons by one, using move() will increment moves by one, and using swap() will increment moves by 3.

insert.c:



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

Parameters: Stats struct, array A / Return type = void

The for loop will go through the length of the array. We will start from the first element (element 1) and compare each element to the one before it. The variable temp represents the element being compared to its previous element. While j is positive, and if the element we are on is less than the previous element, the previous element is copied to the next position and j is decremented. Finally, at the conclusion of the while loop, the element at A[j] is changed to temp, essentially swapping two elements if the previous element is greater than the element being checked (not using the swap() function). This will loop through the whole array to sort every element. Move() is used three times, when storing new values into array elements or assigning an array element to a temporary variable. Compare() is used once when checking if an element is less than the previous element in the array.

shell.c:

```
def shell_sort (*stats: Stats, A: list):
    int gaps = [((log(3.0 + (2.0 * len(A)))) / log(3.0))]
    int index = 0;
    for (int exp = len(gaps): exp > 0: exp -= 1):
        gaps[index] = (3** exp - 1) // 2
        index += 1
    #end if
    int len_gaps = index
    index = 0
    for (index = 0: index < len_gaps; i += 1)
        for (int i = gaps[index]: i < len(A): i += 1)
            int j = i
            temp = move(stats, A[i])
            while j >= gaps[index] and cmp(stats, temp, A[j-gaps[index]]) < 0:
                A[j] = move(stats, A[j - gaps[index]])
                j -= gaps[index]
            #end while
            A[j] = move(stats, temp)
        #end for
    #end for
```

Parameters: Stats struct, array A / Return type: void

This function will sort the given array using the shell sort method. I start by creating an array called gaps with the size of log(3 + 2 (len(A))) // log (3), with the length of A being represented by n in the actual program. In my actual program, I will be using a dynamically allocated array. Then, I use a for loop to iterate through the exponents necessary for the array gaps, placing the calculated gap, (3^exp -1) //2, into each index of the array gaps and incrementing the variable index. Then I assign the value of index to a variable, len_gaps, and then reset index to 0. Then I use another for loop to iterate through each gap in the array gaps, and another nested for loop within that, to iterate from the current gap through the length of the parameter array A. Within these nested for loop is a modified insertion sort using the gaps array when necessary. It works the same way, checking previous elements and moving them if necessary, and the end result is a sorted array. The move() function is used three times, when storing new values into array elements or assigning an array element to a temporary variable; the compare() function is used once when checking if an element is less than the previous element in the array.

heap.c:

We will be using a max heap, in which a parent node is always greater than or equal to its child nodes.

```
def max_child (*stats: Stats, A: list , first : int , last : int):
    left = 2 * first
    right = left + 1
    if right <= last and cmp(stats, A[right-1], A[left-1]) > 0:
        return right
    # end if
    return left
```

max_child:

Parameters: Stats struct, array A, int first, int last / Return type int

This function will return a parent's max child. Essentially it uses the fact that a parent's left child will be located at its parents index*2, and its right child will be located at its parents index*2 + 1. So if the right child is less than or equal to the last element, and if the right child is greater than the left child, the right child index is returned; otherwise, the left child index is returned. Compare is used once to compare two array elements.

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

    while mother <= last // 2 and not found :
        if cmp(stats, A[mother - 1], A[great - 1]) < 0:
            swap(stats, &A[mother - 1], &A[great -1])
            mother = great
            great = max_child (stats, A, mother , last )
        #end if
        else :
            found = True
        #end else
    #end while
```

fix_heap():

Parameters: Stats struct, array A, int first, int last / Return type void

This function will fix the heap by making it a max heap. First, the int mother is set to the first index, a boolean found is set to false, and the variable great is set to the max child index. So while found isn't true, and mother is less than or equal to the last element index/2, the elements at mother-1 and great-1 are compared, and swapped if the element are mother-1 is less than great-1, which makes sense because the mother should be greater than its child. Then, mother and great are reset, and the other elements are compared. When the array is correctly indexed as a max heap, found is set to true, so the function will end. Compare is used once to compare two array elements, and swap is used once.

```
def build_heap (*stats: Stats, A: list , first : int , last : int) :
    for father in range ( last // 2 , first - 1 , -1) :
        fix_heap (stats, A, father , last )
    #end for
```

build_heap():

Parameters: Stats struct, array A, int first, int last / Return type void

This function essentially will run fix_heap as necessary, making sure the array follows max heap properties. The function will start from approximately the middle of the array. It will run fix_heap, which will determine if the parent node's max child is greater than itself, in which case, they will be swapped. The for loop iterates through each of the parents nodes and checks if they follow the max heap properties. If not, they will be fixed. This function will end with checking the first (top) parent node. Compare() and swap will be used by calling max_child and fix_heap.

```
def heap_sort (*stats: Stats, A: list) :
    first = 1
    last = len(A)
    build_heap (stats, A, first , last )
    for leaf in range (last , first , -1) :
        swap(stats, &A[first - 1], &A[leaf -1])
        fix_heap (stats, A, first , leaf - 1)
    #end for
```

heap_sort:

Parameters: Stats struct, array A / Return type void

This function will sort the heap. First, build_heap is run, which will effectively make the array a max heap, sorting it in decreasing order. Then, the first and last elements are swapped, essentially shrinking the array. This means that from now on, we will fix the heap, not including the last element. This process of "shrinking" and fixing a smaller heap, will continue until the final array is sorted in increasing order, which will occur as smaller and smaller "heaps" are fixed. Swap is used once in this function, and compare() and swap() will be used many times by calling build_heap and fix_heap.

quick.c:

```
def partition (*stats: Stats, A: list , lo: int , hi: int) :
   i = lo - 1
   for j in range (lo , hi) :
      if cmp(stats, A[j-1], A[hi-1]) < 0:
         i += 1
        swap(stats, &A[i-1], &A[j-1])
      #end if
   #end for
   swap(stats, &A[i], &A[hi-1])
   return i + 1
```

partition():

Parameters: Stats struct, array A, int lo, int hi / Return type: int

This function will set a pivot (and return it), and any elements less than the pivot will be moved to the left of the pivot, and any elements greater than the pivot will be moved to the right of the pivot. The integer i is set to lo - 1. Then, the for loop iterates from lo to hi, index 1 to the last index. If A[j] is less than the A[hi], the pivot, then i will be incremented, and A[i] and A[j] will be swapped. When j reaches hi, the loop will stop. Then, the element at i + 1 will be swapped with j, the last element. In effect, this will place the pivot in the middle of the array and set up the halves that are less than and greater than the pivot. The extra -1's used in the algorithm are there for the purpose of 1-based indexing in C. Compare() is used once to compare array elements, and swap() is used twice in this function.

```
def quick_sorter (*stats: Stats, A: list , lo: int , hi: int ) :
   if lo < hi:
      p = partition (stats, A, lo , hi)
      quick_sorter (stats, A, lo , p - 1)
      quick_sorter (stats, A, p + 1 , hi)
   #end if
```

quick_sorter():

Parameters: Stats struct, array A, int lo, int hi / Return type: void

This function will essentially sort the array. If the integer lo is less than hi, the initial array will be partitioned. Then, quick_sorter() will be called recursively for both the left half (less than pivot) and the right half (greater than the pivot) of the array. This will, in effect, eventually sort the left and right halves of the array in increasing order, with the pivot already being in place. Compare() and swap() will be called many times, as partition is called and quick sorter is called recursively.

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

quick_sort():

Parameters: Stats struct, array A / Return type: void

This function will sort the array given as a parameter. This function calls quick_sorter() for the given array, with parameters array A, 1 (as we are using 1-based indexing), and the length of the array.

sorting.c:

(psuedocode in Python + English)

This file serves as the test harness for the program, and has multiple parts.

```
create enum ( INSERTION, SHELL, HEAP, QUICK ) named Sorts

int main() :
   Set s = empty_set()
   int seed = 13371453
   int elements = 100
   int print_elements = 100
   bool no_print = true
   bool help = false
   bool no_input = true
   bool bad_input = false
```

I started with creating a “list” of enumerations, with each name being one of the sorts. I start the main function. Within it, I create an empty set. I also set the seed, variable elements, and variable print_elements to what should be their default without user input.

I then declare some booleans which will be used to print out array elements as well as the print menu.

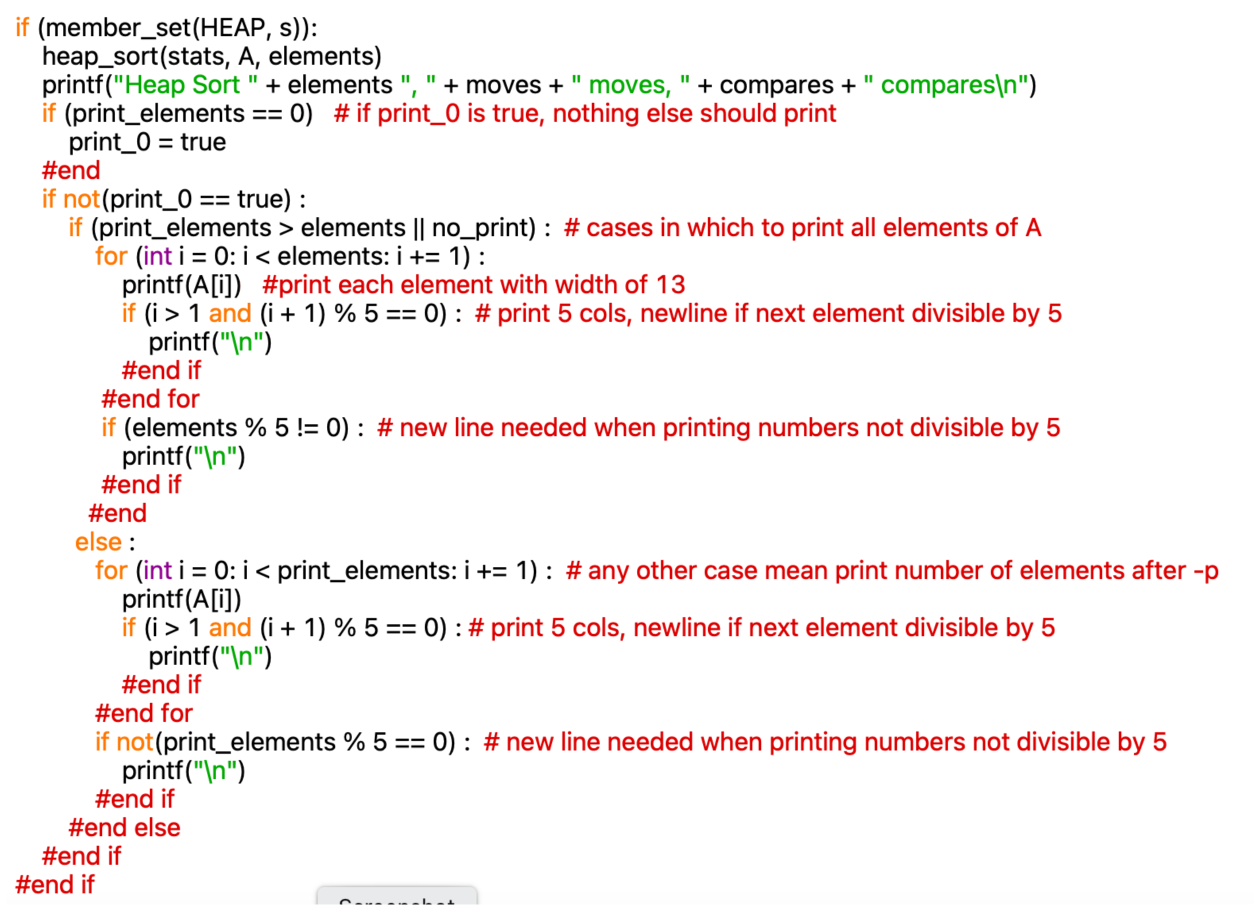
```
while there is input from the user:
           no_input = false
   switch (opt) :
           case 'a':
              s = insert_set(HEAP, s)
              s = insert_set(SHELL, s)
              s = insert_set(INSERTION, s)
              s = insert_set(QUICK, s)
              break
            case 'h':
               s = insert_set(HEAP, s)
               break
            # repeat for each of the sorts
            case 'r':
               seed = input("-r  ")
               if invalid:
                  seed = 13371453
               #end
               break
            case 'n':
               elements = input("-n  ")
               if invalid:
                 elements = 100
               #end
               break
            case 'p':
               print_element = input("-p  ")
               no_print = false
               if invalid:
                 print_elements = 100
               #end
               break
             case 'h':
               help = true
               break
            default:
               bad_input = true
               break
      #end switch
   #end while
```

This while loop will run until the user is done giving the input, otherwise known as using the command line options. First, if the loop is entered, the boolean no_input is set to false. Then, for each of the sorts, in the case of their specified command line options, the enumeration will be inserted into the set (in pseudo, I only put heap sort). The case ‘a’ or all, will insert all enumerations into the set s. For the case of the seed, elements, and print elements, I take the input of the user for their values; if their input is invalid, I set them to the default values. In case ‘h’, for help, I set boolean help to true, which will be used later. Finally, in the default case, which will be reached if the user uses invalid command line options, I set boolean bad_input to true, which will be used later on.

```
create stats structs
set moves to 0
set compares to 0

srandom(seed)
int A = [elements]
for (int i = 0: i < elements: i += 1) :
    A[i] = random() & 0x3FFFFFFF
#end
```

In this snippet, I declare the stats struct and set the moves and compares to 0. Then, I generate random numbers with the given seed. I initialize array A to be the size of elements, the variable declared earlier. Then, using a for loop, each index of A, A[i], is set to a random number that has been bit-masked (limited) to 30 bits.



```
if (member_set(HEAP, s)):
    heap_sort(stats, A, elements)
    printf("Heap Sort " + elements ", " + moves + " moves, " + compares + " compares\n")
    if (print_elements == 0)   # if print_0 is true, nothing else should print
        print_0 = true
    #end
    if not(print_0 == true) :
        if (print_elements > elements || no_print) :  # cases in which to print all elements of A
            for (int i = 0: i < elements: i += 1) :
                printf(A[i])   #print each element with width of 13
                if (i > 1 and (i + 1) % 5 == 0) :  # print 5 cols, newline if next element divisible by 5
                    printf("\n")
                #end if
             #end for
             if (elements % 5 != 0) :  # new line needed when printing numbers not divisible by 5
                printf("\n")
             #end if
           #end
         else :
            for (int i = 0: i < print_elements: i += 1) :  # any other case mean print number of elements after -p
                printf(A[i])
                if (i > 1 and (i + 1) % 5 == 0) : # print 5 cols, newline if next element divisible by 5
                    printf("\n")
                #end if
            #end for
            if not(print_elements % 5 == 0) :  # new line needed when printing numbers not divisible by 5
                printf("\n")
            #end if
        #end else
    #end if
#end if
```

This large portion of code is for printing the result of the command line options for each of the sorts. If the sort-specific enumeration is a member of the set, I run the corresponding sort, and print the number of elements, moves, and compares. If print_elements was set to 0 (using -p), nothing else will be printed. Otherwise, using the boolean declared in the first part of the program, I determine the number of elements to print. If print_elements is greater than elements or input was given to -p, the whole array is printed, by iterating in a for loop using elements. The data is printed into 5 columns and formatted as specified. In any other case, the number of elements to print should be print_elements (the input given to -p), and the output is printed as such.

(This section of code is repeated 4 times, once with each of the different sorts.)

```
reset(stats)   # must reset stats before running another sort
srandom(seed)  # re-randomize array after each sort
for (int i = 0: i < elements: i += 1) :
    A[i] = random() & 0x3FFFFFFF
#end for
```

After sorting and printing one of the sorts' output, I must reset stats. This ensures the correct number of moves and compares is counted, by starting fresh for each sort. Then, because all elements of A will be sorted after a sort function is run, I re-randomize the elements of array A, generating new random numbers with the seed and re-populating the array.

(This section of code is repeated 4 times, once after running each of the different sorts.)

```
if (help or no_input or bad input):
    print help menu

# freeing memory (specific to C)
free(A)
return 0
```

This last part of this file is to print the help menu when necessary. I print the help menu if one of three cases is true; the user typed '-h', the user gave no input (command line options), or the user attempted to use invalid command line options. Then, as array A is dynamically allocated in my actual program, I use free(A) at the end, as is necessary when using dynamic allocation. Finally, I return 0, signaling the end of the program.