Sorting Algorithm Designs

Moore Macauley
University of California, Santa Cruz

October 23, 2022

1 Introduction

In this assignment, I was tasked to create and then compare the runtimes of four different sorting algorithms. To do this, for each of the algorithms, I will keep track of the number of comparisons and moves made using the provided stats module, to make understanding the difference between the algorithms easier.

2 Bubble Sort

The first and simplest sorting algorithm is bubble sort. The idea behind it is very simple. We iterate over the array, comparing every element to the next element in the array. If the next element is smaller than the current element, we swap them. Once the whole list has been iterated over, if no swaps were made, the list is sorted. Otherwise, the list is not sorted, but the largest element must be the last position, so we can ignore it as we iterate over the list again. We do this until the list is sorted, either because no swaps were made, or because there is only one item left to be sorted, as an array with a length of 1 is, by definition, sorted.

```
Pseudocode:

Include bubble.h

Get passed a pointer to a statistics structure stats, a pointer arr to an array of
unsigned 32 bit integers, and the number of elements in the array.

Set the int length to the number of elements in the array minus one.
Create the int swapped.

For every value i between 0 and the number of elements - 1
    swapped = false
    For every value j between the number of elements -1 and i, counting down
        Compare element j and element j-i with stats
        If element j is less than element j-1
            Swap element j and element j-1 with stats
```

```
            Set swapped to true
    If swapped is false
        Break out of the loop
```

3 Shell Sort

Shell sort is a little bit more complex. It works by comparing elements that are separated by a gap value, and swapping them if the earlier value is less than the later value. It then decreases the gap value and repeats until the value is equal to 1. This is much faster than bubble sort, as while it works similarly, the gap allows the list to be almost sorted much more quickly, making the final pass much faster. People much smarter than me have determined that a very good gap sequence starts by multiplying the length of the array by 5, then dividing that by 11 (rounding down) to get the first gap. This same equation is applied to the first gap to get the second gap, and so on and so forth until the gap is less than or equal to 2, at which point we set the gap to 1.

To use this algorithm, I will first create an additional function next gap to calculate the next gap, given the previous gap. It returns 0 if the gap it is given is 1, as this will end the for loop that it is being used in, discussed later. Back in the shell sort function, I will use a for loop that will continue calculating the next gap based on the previous gap until the gap is equal to or less than 1. Inside this for loop, I will iterate over the value i, between the current step and the length of the array to be sorted. I will then create a third and final for loop, where I will iterate over the value j, between i and gap - 1, using subtracting gap from i each time. If element j is less than element j-gap of the array, then we use stats to swap them.

```
Pseudocode:

Include shell.h

int next gap(int current gap, int *end):

    If n == 1
        Return 0

    Multiply current gap by 5, then divide by 11, using int division to round down.
    Save this in int next gap.
    If next gap less than or equal to 2
        Set next gap to 1
    Return next gap



void shell sort(Stats stats, int *arr, int number of elements)

    create *int end, set to false
```

```
Set initial gap to gap(number of elements, end)

For gap = initial gap, gap > 0, gap = next gap(gap, end)
    For i = gap, i < number of elements, i += 1
        Set int j to i
        Save arr[i] in temp
        While j is greater than or equal to gap and temp is
        less than or equal to element j - gap
            Use stats to swap element j and element j-gap
            j-=gap
       arr[j] = temp
```

4 Quick Sort

Quick sort is, on average, the fastest sorting algorithm that uses comparisons. It works by dividing up the array that will be sorted into three pieces around a pivot value. This pivot value is equal to the average of the first element of the array, the middle element, and the last element. The three pieces contain all elements smaller than the pivot value, all elements equal to the pivot value, and all elements larger than the pivot value respectively. Quicksort is then run recursively on the first and third pieces. As quicksort handles small arrays poorly, we use shell sort if there are fewer than 8 elements in the array.

In order for me to implement this, I will create another function, actual quick sort, which will accept four arguments. The first will be a pointer to the array to be sorted, the second will be a pointer to a Stats struct, the third, the number of the first element in the sub array, and the fourth the number of the last element in the sub array. When the quick sort function is called, it will then immediately call actual quick sort, with the arguments arr, stats, 0, and number of elements.

Inside actual quick sort, there will be two for loops, with three additional integers, mid start, mid end, and mid buffer. Mid start should start equal to the position of the first element, mid buffer should be start equal to the position of the final element, and mid end will remain undefined for now. The first for loop will iterate over the int i, representing all the values between the position of the first element and mid buffer. If the value of i is less than the pivot and i is not equal to mid start, we swap the value at mid start with the value at i, before incrementing mid start. If the value of i is equal to the pivot, we swap the value of i with the value at mid buffer - 1. We then subtract 1 from i so whatever value that had just been swapped can be compared to the pivot, and then subtract 1 from mid buffer as wee.

Once this loop has finished, all values smaller than the pivot should be between the initial first element and mid start, all values larger than the pivot should be between mid start and mid buffer, while all values equal to the pivot should end up between mid buffer and the inital final element. We should then set mid end to be equal to mid start, before iterating over all the values between mid buffer and the inital ending element, swapping element mid buffer with element mid end, and then incrementing mid end. We should then call quicksort on the elements between start and mid start, and mid end and end.

Finally, if the difference between the position of the initial starting value and the position of the initial ending value are less than 8, I will use an altered version of the above shell sort, which takes the initial and ending positions rather than the number of elements, but is otherwise the same.

Pseudocode:

```
Include quick.h

int next gap(int current gap, int *end):

    If *end is true
        Return 0

    Multiply current gap by 5, then divide by 11, using int division to round down.
    Save this in int next gap.
    If next gap less than or equal to 2
        Set *end to true
        Set next gap to 1
    Return next gap



void altered shell sort(Stats stats, int *arr, int start, int final)

    create *int end

    Set initial gap to gap(final - start, end)

    For gap = initial gap, gap > 0, gap = next gap(gap, end)
        For i = gap + start, i < final, i += i
            For j = i; j>=start + gap-1; j-=gap
                Compare element j and element j - gap with stats
                If element j is less than element j - gap
                    Use stats to swap elements j and j - gap



void actual quick sort(Stats *stats, int *arr, int start, int end)

    If end - start is less than 8
        altered shell sort(stats, arr, start, end)

    Create int mid start = start;
    Int pivot = the average of element 0, element mid (start + end/2), and element mid end

    Create int middle buffer = end

    For int i, between start and middle buffer, increment by 1
        Compare element i and pivot with stats
```

```
        If element i is less than pivot
            If i is not equal to mid start
                Use stats to swap element mid start with element i
            Increment mid start
        If element i equals pivot
            use stats to swap element middle buffer - 1 and element i
            Reduce i by 1
            Subract 1 from middle buffer

    int mid end = mid start

    For the values between middle buffer and end, incrementing by 1
        swap element mid end with element middle buffer

    actual quick sort(stats, arr, start, mid start)
    actual quick sort(stats, arr, mid end, end)



void quick sort(Stats *stats, int *arr, int number of elements)
    actual quick sort(stats, arr, 0, number of elements)
```

5 Heap Sort

Heap sort works by creating a heap, then using it to quickly sort an array. A heap is a tree like structure, consisting of nodes. Each node has a value, and is connected to up to two nodes that are larger than it, with the first node being the smallest value. To sort using a heap, there are two steps. We first create a heap using the array to be sorted, and then repeatedly take the first element of the heap, fixing the heap after every removal.

In order to do this, I will create a number of functions. The first three left child, right child, and parent will simply return the position of the left child, the right child, and the parent of a node respectively, given the position of that node, which will be trivial to implement.

Up heap will, given the position of a node, repeatedly compare that node with its parent, and if the parent is larger than the node, it will swap the nodes. Otherwise, it stops.

Build heap will iterate over all of the values in the array to be sorted, and add a node containing the iterated value as a child to the first node that is lacking a child, before passing the position of the new node to up heap until the new node is larger than its parent. This will then organize the heap appropriately.

Down heap will, given the current heap size, move the value in the first slot of the heap down until it reaches an appropriate place. It does this by iterating in a while loop while the position of the position of the left child is less than the heap size. Inside this while loop, it first checks to see if there is a right child, marking the left as the smaller child if there is one. If both children are there, it compares the two to determine which is smaller. If the smaller child is smaller than the parent, the smaller child and parent are swapped. Otherwise, the function breaks out of the while loop, as the heap is ordered correctly.

Finally, heap sort will use all of these previous functions to sort the array. It first uses build heap to

build the heap, saving it into a new array, before creating a new array, sorted array, with the same length as heap and the array passed to it. It then iterates i between 0 and the length of the array, setting the ith element of sorted array to the first element of heap, and then swapping the last element of heap with the first element element. It then uses down heap to reorganize the heap array properly.

Pseudocode:

```
int left child(int position)
    return position * 2 + 1

int right child(int position)
    return position * n + 2

int parent(int position)
    return (n-1)/2



void up heap(int *arr, Stats *stats, int position)
    While position is greater than 0, and the node at position is less than the
    parent of position
        Use stats to compare the node at position and the parent of position
        (The above doesn't do anything, but compare needs to be incremented)
        Use stats to swap the node at position and the parent of position
        Set position to the parent of position



void build heap(int *arr, int *heap, int number of elements, Stats *stats)
    For every number i between 0 and the number of elements
        Element i of heap is equal to element i of array
        up heap(heap, stats, i)



void down heap(int *heap, int heap size, Stats *stats)
    position = 0
    create int smaller
    while left child(position) is less than heap size
        if right child(position) is equal to heap size
            smaller = right child(position)
        else
            Use stats to compare left child and right child
            if left child is smaller
                set smaller to left child position
```

```
            else
                set smaller to right child position
        Use stats to compare the node and the smaller child
        if node is smaller than the smaller child
            break out of the loop
        Use stats to swap the node and the smaller child
        Set position to the position of the smaller child



void heap sort(Stats *stats, int *arr, int number of elements)
    int heap[number of elements] = {0}
    build heap(arr, heap, number of elements, stats)
    int sorted list[number of elements] = {0}

    for every value i between 0 and number of elements
        Use stats to set sorted list[i] to heap[0]
        Use stats move to set heap[0] to heap[number of elements - i - 1]
        down heap(heap, number of elements - i, stats)
```

6 Main

The main program will be used to test the efficiency of all of the sorting algorithms. In order to do this, main will accept arguments from the command line, telling it what tests to run. These arguments will be parsed with a switch and getopt, and the provided Set will store a number that corresponds to the argument that was passed. Using the length of the array to be sorted, I will then use the seed, mtrand.c, and a for loop to generate an array that needs to be sorted by my function. Then, I will use a for loop to iterate over all of the possible arguments that could be used, and run the tests for them if they appear inside the Set. I plan to use an array of function pointers to determine when a function should be run inside of this for loop, which should contain a pointer in the slot corresponding to the value set inside the Set. Ie, if bubble sort corresponded to 1, then the pointer in slot 1 of the array would point to the bubble sort function.

Inside the for loop, I start by copying the unsorted starting array into a new location, and create a new Stats. I then pass the new array, new stats, and previously defined array length variables into the corresponding sorting function. I then print out the name of the sort, the number of elements in the array, the number of moves stats made, the number of compares stats made, and finally, the first x elements in the sorted array, with x being either 100 or whatever the -p argument set it to.

```
Pseudocode:

Start by getting argc and argv from command line
Create int seed, set to 13371453
```

```
Create int size, set to 100
Create int print, set to 100
Create Set set

Iterate over option with getopt
    Switch
    case a
        Insert 2-5 in set
        Insert 6 in set
    case s
        Insert 2 in set
    case b
        Insert 3 in set
    case q
        Insert 4 in set
    case h
        Insert 5 in set
    case r
        Set seed to the seed in optarg
        Insert 7 in set
    case n
        Set size to the size in optarg
        If size is not between 1 and 250 million
            Set error to true
        Insert 8 in set
    case p
        Set print to the size in optarg
        Insert 9 in set
    case H
        Insert 11 in set
    default
        Insert 10 in set

If 10 or 11 is in set
    Print out program usage
    if 11 is in set
        Return 0
    Return 1

Create an array of strings with the names of sort algorithms called names
Shell sort should be in slot 0, bubble sort in slot 1,
quick sort in slot 2, and finally heap sort in slot 3.

Create an array of pointers to sorting functions, called
sorters. These should be in the same order as the array above.
```

```
Use calloc to create two pointers, start arr and arr, both count size and
size 4.

Set the mtrand seed to seed

For all values n between 0 and size
    Set start arr[n] to a random number generated with mtrand

For all values i between 0 and 4
    if i + 2 is in set
        Copy start arr into arr with a for loop
        Create a new Stats, stats
        Pass arr, stats, and size into sorters[i]
        Print names[i], size, stats move, and stats compares
        Create int i = 0
        If print is greater than size,
            print = size
        While int i is less than print
            for all values b between 0 and 5
                print arr[i]
                i++
            Print newline

Free arr and start arr
```

7 Graphing

Finally, in order to complete my writeup, I need to produce graphs showing the performance of the different sorts on a variety of inputs. The inputs I will be considering will be smaller arrays, very large arrays, smaller arrays in reverse order, and smaller arrays already in order. To do this, I will create a grand total of eight graphs, two for each scenario, one plotting the number of comparisons an algorithm makes, and a second for the number of moves an algorithm makes. To make things easier for myself, I will create a script, detailed below, and slightly edit main for each scenario.

The script will be fairly simple. It will iterate some value i from start to end, with start and end being changed to different values for each scenario. It will then run sorting, the program built with main, with the argument -a to run all the algorithms, -r i to use seed i for generating the array, -n i to set the size of the array to i, and finally -p 0, to make it not print out any element from the array. The program will then take line 0 of the output and write it into shell.dat, line 1 into bubble.dat, line 2 into quick.dat, and finally line 3 into heap.dat. This will loop until i has reached the start value, at which point it will graph shell.dat, bubble.dat, quick.dat, and heap.dat into a graph, with each being its own line.

Pseudocode:

```
for i between start and end
    ./sorting -a -r i -n i -p 0, and overwrite temp.dat
    Take the first line of temp.dat, redirect it into bubble.dat
    Take the second line of temp.dat, redirect it into heap.dat
    Take the third line of temp.dat, redirect it into quick.dat
    Take the fourth line of temp.dat, redirect it into shell.dat

Graph shell.dat, bubble.dat, quick.dat, and heap.dat using gnuplot with lines
```

This is a very simple script, and to make it work properly, I plan to edit main slightly, as I find using bash much more difficult than c. For all four scenarios, I plan to first edit main to make it only output the array size and the number of comparisons, before running the script. I will then run the script a second time, editing main again to now make it only output the array size and the number of moves. As this is the format required by gnuplot, no further manipulation will need to be done by the script in order to graph the lines. I could leave the output unchanged and manipulate it with bash, but as I am only making eight graphs, this seemed easier.

For the first scenario, small arrays, I plan to make start 1, and end 1000. This will give us a good idea of how long it takes the different algorithms to sort smaller array sizes, as compared to the 250 million max size 10k is relatively small, but not so small an idea of the data's trend cannot be established.

For the second scenario, large arrays, I plan to make start 1 million, and end 1 million, 10000. This will give us a good idea of how long it takes the different algorithms to sort very large arrays, as 1 million is very large, and this will analyze several arrays in that ballpark so we can get an idea of the data's trend. I will, unfortunately, have to edit the script to not run bubble sort, as bubble sort simply takes too long when sorting a million elements.

For the third scenario, smaller arrays in reverse order, I plan to once again use a start of 1 and an end of 10000, but as sorting does not natively support arrays in reverse order, some edits will need to be made. I will implement a small function, reverse order, detailed below, that when given a pointer to an array and the length of the array will sort that array from largest to smallest. This function will use bubble sort, simply because it is the easiest to implement. This function will be run on the arr in main immediately before it is passed to the sorting algorithm being used, causing the array that the sorting algorithm is used on to be in reverse order.

```
Pseudocode:

Get the pointer arr and the int len

Create the int sorted

do
    set sorted to 1
    subtract 1 from len
    for int i between 0 and len
        if element i is smaller than element i+1
```

```
            swap element i and i+1
            set sorted to false
While sorted is false
```

For the fourth and final scenario, smaller arrays in order, I plan to use the same start and end as before. I will simply use quick sort to sort the array before it is passed to any other function.