Jason Russo
ECE 2400

PA3-Algo-Report

1. Introduction

PA3 focuses on implementing various sorting algorithms. This task allows us to not only practice our coding skills, but to sharpen new ones. We consider time complexity using Big-O notation and compare various algorithms against each other, even though. At a high level, they all do the same thing. This assignment allows us to see for ourselves how different algorithms have their own unique tradeoffs and how these tradeoffs must be considered when writing a program. I successfully implemented all of the sorting algorithms in this assignment. Each implementation relied on at least one helper function to either aid in the sorting, or handle the sorting itself. My selection sort implementation uses a helper function to find the index of the minimum value in the array so it be moved to the front. This process is repeated until the entire array is sorted. My merge sort implementation uses two helper function. The first, merge_recur, splits the array into two and then recurses, splitting the array even further. When the recursion unwinds, the helper function merge is called to merge all the arrays into one sorted array. My quicksort implementation also uses two helper functions. The partition helper function is used to push all values in the array greater than the partition value to a spot in the array after the partition and then returns the index of the partition. Then, quick_recur is called on either side of the partition, splitting the array further. When the recursion unwinds, the array is sorted. Finally, my pseudo-radix sort implementation also uses two helper functions. Radix_help is sorts the values into the array into their proper bin, using the given get_digit_at function to find which bin a number belongs in. Then, quicksort is used to sort each bin. In the end, I found selection sort to be quite slower than the other implementations, which performed fairly similar given 10 random arrays of size 50,000. The only instance in which selection sort did not perform the worst was in the case of a reversed array. In this case, quicksort was the slowest. Merge sort performed the best for each targeted test, but lost to quicksort by .0004 seconds in the 10 random tests. All things considered, I think merge sort would perform the best since it does not have as bad of a worst case as quicksort does.

2. Implementations

The first algorithm I implemented was selection sort (Figure 1). All of the sorting algorithms have the same interface, so they all take an int array and the array size as their inputs. For selection sort, these inputs are enough. My implementation iterates through the array, finding the minimum value, and then swapping the minimum value with the current index. The find_min (Figure 2) helper function is used to find the minimum value of the array. To prevent the function from finding the same minimum value over and over, it takes a begin input along with array and size. This begin index tells find_min where to start in the array. Begin is the value of the iterator in selection_sort, so the last minimum value will be at iterator - 1 and thus will not be found by find_min. The swapping of values was done using the swap helper function, which was provided for us.

The next implementation was merge sort (Figure 3). In this case, having only the array and size are not sufficient to properly implement merge sort, so a helper function merge_recur (Figure 4) is used. This helper function takes an array and its beginning and ending indexes as inputs. It then splits this array at the midpoint and calls merge_recur on each of the two halves of the original array. This recursion will continue until the base case (end - begin <= 1) is met. Now that we have a bunch of one element arrays, the recursion will unwind, taking two of the one element arrays and calling the merge function (Figure 5) on them, merging them into a sorted array. After all of the one element arrays are merged into two element arrays, those will merge into four element arrays and so on until we have an array of the original size with all of the elements sorted. Once this is done, the values of this temporary array are copied into the original, finishing the sort. While the merge_recur function handles the splitting, the merge function handles the merging and, most importantly, sorting. This functions inputs are a temporary array, two arrays to be merged, and beginning and ending indexes for the two arrays to be merged. This function iterates through the size of the temporary array (which is the combined size of the two arrays to be merged) and checks multiple if/else statements to see from which array the next value should be taken. If the next value of array_a is less than the next value of array_b, it will take from array_a and vice versa. It will also check if it has taken all values

from one array, in which case it will simply take values from the other. This is kept track of through iterator variables that keep track of the next index in each array.

Similar to merge sort, quicksort (Figure 6) uses an array and a beginning and ending index instead of merely the size. Once again, this requires the creation of a helper function, quick_recur (Figure 7). Quick_recur gets a pivot index, p, by using the partition helper function (Figure 8). It then calls quick_recur on either side of the pivot. This recursion will stop when the base case of begin equal to end is met. At this point, the function will unwind, and the different sub-arrays, which were each 'sorted' around their pivot will create one fully sorted array. The important part of this quicksort implementation is the partition function, which takes an array along with a beginning and ending index. This function then sets a pivot value equal to the second to last index of the array. It will then check each value in the array and, if it finds a value that is less than or equal to the pivot, it will swap it to an index before the pivot, which is tracked by an iterator value that starts at the first index of the array. This creates an array with values less than the pivot near the start of the array, and values greater than the array near the back, after the pivot index. These values, however, are not necessarily sorted. As we call partition on smaller and smaller arrays, it will eventually lead to the case where these arrays are fully sorted. When quick_recur unwinds, the values will now be sorted.

In this PA, we were asked to implement a pseudo-radix sort. A real radix sort would sort the values in the array by each digit. For example, the algorithm might start with the least significant digit and sort the array by that value. It would then sort by the next most significant digit and so on, until the entire array was sorted. In our case, we simply sorted by the K - 1 digit, placing all numbers with the same K - 1 digit into a 'bin', and then sorted those bins and returned them in order. I used a K value of 5 which represents the hundred thousands place of a decimal number. A radix (R) of 10 was used, meaning that we would be evaluating the numbers as decimal numbers in which each digit represents a power of 10. My implementation of this pseudo radix sort (Figure 9), uses a helper function called radix_help (Figure 10) to handle the sorting. Radix_help takes an array, a K - 1 value, and beginning and ending indexes as inputs. A while loop is run as long as the iterator variable is at index within the array (between begin and end) and that we are in a valid bin (0 <= r <= 10). Inside this while loop, a for loop iterates through the array and checks the digit at the current index, using the get_digit_at helper function provided to us. If the digit is equal to the current bin (r) , then it is swapped with the iterator value to move it into the proper bin within the array. After the entire array has been iterated through and all values of the bin have been placed together, quicksort is called on the bin, which runs from hold to iterator and then r is incremented to check the next bin. Hold is set to iterator to hold the new starting point of this new bin.

Despite each algorithm having the same interface and serving the same purpose, their implementations were quite different. The selection sort implementation was the simplest for me to implement. The find_min helper function was fairly straightforward, and the sort merely swaps the next min value into place. The simplicity of this implementation comes at a cost to speed, which we will discuss more in depth later on. Merge sort got a lot more complicated, requiring a recursive helper function, along with a merge function to actually do the sorting. In terms of complexity, I found merge sort to be the second most complex implementation, right behind radix sort. Radix sort was the most complex to implement largely because it had not been covered in lecture. Since we only had pseudo-code to go off of, I found that I had to try and test various implementations just to get one that worked. This struggle was perhaps not worth it, as radix sort simply was not the best performing algorithm. Finally, there was quicksort, which was relatively on par with merge sort. Quicksort also relied on a recursive helper function along with another helper function to 'sort' the array. These functions were a bit more straightforward than merge sort, however.

3. Testing Strategy

My testing strategy consisted of three kinds of testing: basic, directed, and random. In each case, white box testing was used. White box testing means that only the interface of the function was tested. In this case, it means the test only checked if the array was sorted. Nothing was checked in regards to how exactly the function sorted the array, only that it was sorted in the end. Since all the functions had the same interface, this means that the same tests could be used for each function, which is what I did. Each function was tested with the same basic, directed, and

random tests. The basic test for each function was an unsorted array with 5 elements. This obviously is a very simple case and is used to see if our implementations are on the right track. We obviously need far more testing than just this basic case, but it is a good start. Next up was the directed testing. Directed testing is used to target edge, corner, and special cases that our implementations must be capable of handling. Directed testing is extremely important because one corner case that is handled improperly can lead to many issues down the line. As such, it is crucial to spend a lot of time thinking about these cases to make sure we cover as many as possible in our directed testing. After we finish directed testing, we move on to random testing. Random testing is used to cover as many general cases as possible. After all, the inputs that the functions will receive from users will be random testing, so this testing is a good way to see how the functions will handle regular use. In the best case, random testing may even uncover a new corner case that the programmer may have missed in the directed testing. Either way, it is still a very good way to make sure that the implementation can handle most general cases.

The goal of my directed testing was to cover as many corner and special cases as possible. It is stated in the PA handout that the values contained within the int arrays are limited to 0 to 99,999 inclusive. As such, negative values and values greater than 99,999 were not tested and are left to undefined behavior. The cases that were tested are: an empty array, an array of a single element, an array that is already sorted, and an array with duplicates. Admittedly, it is hard to test an empty array as there is nothing to compare, but the fact that the test runs shows that there is no issues of infinite recursion due to insufficient base cases. The case of an array of size 1 had a similar purpose since a few implementations rely on recursion and splitting the arrays in half. Already sorted arrays were tested to make sure that the implementations would not scramble any values and that they could safely return the same array that was inputted. Lastly, arrays with duplicates were tested to make sure that, when swapping and moving values, the implementations would not disregard matching values and would properly place them in the array.

Random tests were used to check a large amount of general cases and to simulate the random use that the functions would receive in the hands of general users. To ensure that all the inputs to the functions were valid, a max int of 100,000 was declared and the values added into the arrays were done in the manner of rand() % 100,000 to limit the values between 0 and 99,999 inclusive. 100 random tests were run and the random arrays had a random size between 0 and 1,000. A reference array was then created and was given all of the values of the random array. The reference array was sorted using the C standard library qsort function. The proper sorting algorithm was then called on the random array and then it was compared to the reference array to ensure it was sorted properly.

In the end, I believe my testing covered most, if not all, edge and corner cases. This is backed even further by passing all of the tests used to run the function evaluations, which will be discussed in the next section. In terms of code coverage, I had 97.9% coverage, or 457 lines out of 467. This means that, during my testing, 457 of 467 lines of code were used. While a high code coverage percentage does not ensure that my code is entirely correct, it does reassure me that my implementations have been tested thoroughly and that almost all of the code I wrote had a definitive purpose and was used to pass the various tests. Based on all of this, I believe that my implementations are functionally correct. There may be ways to further optimize these implementations, but I believe that ones I have written will accomplish their purpose.

4. Evaluation

Since all of the implementations share the same purpose, the key characteristic that was evaluated was time. Each implementation was run through multiple trials of different targeted and random tests and were evaluated for time. For each test, five trials were ran and the average time was taken to give a more reliable result. This testing was all done in a build-eval directory with a release build of the code. The different tests run were as follows: reversed arrays, almost sorted arrays, arrays with few unique elements, a random array, five random arrays, and ten random arrays. Testing reversed arrays was quite important as it proved to be the worst case scenario for my quicksort implementation, costing lots of time. The different implementations had surprisingly different results for each test, compared to not just the other algorithms, but also to themselves.

The results of the speed testing are found in Figure 11. Using the random-10x test as the general case, we can see that selection sort was by far the slowest algorithm, taking nearly 195 times as long as the next slowest

algorithm! This difference in speed is tremendous. The only instance in which selection sort is not the slowest is in the case of a reversed array, in which quicksort performs the slowest. This makes sense, as a reversed array is the worst case for quicksort as it must swap every value in the array. Quicksort redeemed itself, however, by being the fastest algorithm in the random-10x test, beating out merge and radix sort, which took the same amount of time. Looking at merge sort, it took nearly identical time for each of the targeted cases, along with the random tests. Radix sort, on the other hand, was a bit up and down. It matched radix sort on the random tests and on the almost sorted array, but was far slower in the case of a reversed array and an array with few unique elements. This makes sense, as quicksort struggled in those areas, and radix sort uses quicksort to sort the various bins.

Despite performing best in the 'general' test, I do not believe quicksort to be the most effective sorting algorithm of those implemented. Instead, I give that title to merge sort. Merge sort performed the best on the targeted tests and, most importantly, was consistent. Merge sort took anywhere from .1ms to .3ms for the targeted tests, which quicksort took anywhere from .3ms to 8.4ms, which is a huge range! Radix sort admittedly suffered from using quicksort in order to sort its bins. Finally, selection sort is just too simplified of an algorithm and leaves much to be desired. The implementation of selection sort is simple and easy, but comes at a drastic cost to speed. Merge sort was a rather complex implementation, but the complexity paid off in the form of consistent speed. For these reasons, I believe merge sort to be the most effective sorting algorithm implemented.

5. Appendix

```
def selection_sort ( array, size)

   for index in array
      idx = find_min( array, index, size)
      swap( array, index, idx)
```

(Figure 1)

```
def find_min ( array, begin, size )

   min_value = array[begin]
   min_index = begin

   for index in array
      if array[index] less than min_value
         min_value = array[index]
         min_index = index

   return min_index
```

(Figure 2)

```
def merge_sort ( array, size )

   merge_recur( array, 0, size )
```

(Figure 3)

```
def merge_recur ( array, begin, end )

   size = end - begin
   if  size is less than or equal to 1 return
   middle = ( begin + end ) / 2

   merge_recur( array, begin, middle )
   merge_recur ( array, middle, end )

   merge( temp_array, array, begin, middle, array, middle, end )

   for index in begin to end
      array[index] = temp[index]
```

(Figure 4)

```
def merge ( output_array, array_a, begin_a, end_a,
            array_b, begin_b, end_b )

   size = ( end_a - begin_a ) + ( end_b - begin_b )
   index_a = begin_a
   index_b = begin_b

   for index in 0 to size

      if  at end of array_a
         output_array[index] = array_b[index_b]
         index_b = index_b + 1
      elif at end of array_b
         output_array[index] = array_a[index_a]
         index_a = index_a + 1
      elif array_a[index_a] less than array_b[index_b]
         output_array[index] = array_a[index_a]
         index_a = index_a + 1
      else
         output_array[index] = array_b[index_b]
         index_b = index_b + 1
```

(Figure 5)

```
def quick_sort ( array, size )

   quick_recur( array, 0, size )
```

(Figure 6)

```
def quick_recur ( array, begin, end )

   if begin is end return

   part = partition( array, begin, end )

   quick_recur( array, begin, part )
   quick_recur( array, part + 1, end )
```

(Figure 7)

```
def partition ( array, begin, end )

   pivot = array[end - 1]
   index = begin

   for i in begin to end

      if array[i] less than or equal to pivot
         swap( array, i, index)
         index = index + 1

   return index - 1
```

(Figure 8)

```
def radix_sort ( array, size )

   radix_help( array, K - 1, 0, size )
```

(Figure 9)

```
def radix_help ( array, K - 1, begin, end )

    hold = 0
    r = 0
    iterator = begin

    while iterator in array and r less than 10

        for i in begin to end

            if get_digit_at( array[i], K - 1, 10 ) is r
                swap( array, i, iterator )
                iterator = iterator + 1

        quick_recur( array, hold, iterator )
        hold = iterator
        r = r + 1
```

(Figure 10)

	Selection	Merge	Quick	Radix
Reversed	0.0077	0.0001	0.0084	0.0009
Almost Sorted	0.0077	0.0002	0.0002	0.0003
Few Unique	0.0076	0.0002	0.0010	0.0009
Random	0.0076	0.0003	0.0003	0.0003
Random - x5	0.1832	0.0017	0.0014	0.0017
Random - x10	0.7269	0.0036	0.0032	0.0039

(Figure 11)