Fall
2015

An Experimental Analysis of Three Efficient Sorting Algorithms

BY
CONNOR DELAPLANE, RICKY VALENCIA, AND ALEX GODDARD

PORTLAND STATE UNIVERSITY | CS 350 Algorithms and Complexity

AN INTRODUCTION TO THIS DOCUMENT

In this document, we analyze the behavior of three industry-standard sorting algorithms on large data sets. Namely, we will be studying, in turn, **quicksort**, **mergesort**, and **heapsort**. Each of these sorting algorithms has an asymptotic efficiency of O(n log n), which has been proven to be a lower bound for the sorting problem. In our analysis of these algorithms, our primary focus will be on the relationship between growth of input size and amount of work done by the algorithm (measured in basic operation execution counts and execution times on our running machine). We will also analyze the stability, complexity, space requirements, and other notable pros and cons of each algorithm. Upon completion of our analysis, we will conclude with a reflection on our findings and use them to select our "favorite" algorithm; the one that we perceive to have the highest net benefit, widest applicability in professional settings, and most probable longevity (predicted usage in the years to come).

To obtain our test results, we developed a program to automatically construct randomly generated arrays of sizes from 10 to 1,000,000 in various states of pre-assortment and run each algorithm on them (however, we found the results for sizes of 100,000 and 1,000,000 to be of primary interest, and the data for smaller input sizes is therefore ignored here). For each pre-assortment "case", the algorithm was run 100 times on different pseudorandom inputs, and the basic operation counts as well as the execution times in milliseconds were accumulated and averaged to obtain the final results (contained in the tables found later in this document). Prior to automated testing, each individual algorithm was thoroughly tested and debugged to ensure correct sorting behavior. The next page contains the code for the main function of our automated testing program.

```
public void run() throws Exception {
    Map<String, int[]> numListsMap = new HashMap<String, int[]>();
    List<Sort> sorts = new ArrayList<~>();
    int[] sizes = new int[SIZE];

    //Creates a list of the sorts that will be used
    generateSorts(sorts);
    //Generates a list of the sizes used for each list
    generateListSizes(sizes);
    //Deletes the previous .txt files that store the results
    deleteExistingResults(sorts);
    for (int size: sizes) {
        //Generate the pre-assortment of integer lists of length size
        generateLists(numListsMap, size);
        //For each item in the list where the key is the name/type of the list
        //and the value is the list itself.
        for (Map.Entry<String, int[]> list: numListsMap.entrySet()) {
            for (Sort sort: sorts) {
                //Run the sort on the pseudo random generated
                // pre-assorted lists NUM_OF_TEST times
                for (int i = 0; i < NUM_OF_TESTS; ++i) {
                    //Sorts the list
                    int[] sortedList = sort.sort(list.getValue());
                    //Test to see if the list was actually sorted
                    int[] sortedClone = list.getValue().clone();
                    Arrays.sort(sortedClone);
                    if (!Arrays.equals(sortedList, sortedClone)) {
                        throw new Exception("Failure to sort "
                                + list.getKey() + " for sort "
                                + sort.getTypeOfSort());
                    }
                }
            }
            //Write the average results for each sort in a file
            writeToFile(list.getKey(), sorts);
            //Resets the time, size, and basic operation values for each sort
            for (Sort sort: sorts) {
                sort.reset();
            }
        }
    }
}
```

ALGORITHM #1: QUICKSORT

What is Quicksort?

Quicksort is one of the divide and conquer sorting algorithms; it dynamically divides itself as it sorts itself. Developed by Tony Hoare in 1959, it continues to thrive as one of the fastest sorting algorithms. The dynamic division is based off what is called the pivot, which helps split the list into two parts with values less than the pivot stored on the left side of the array, and values greater than or equal to the pivot on the right side of the array relative to the pivot. From here, it recursively calls itself on the left and right sub-array and does the same division until the sub-arrays are sorted. One advantage to this algorithm is that it works inward from both ends of the list instead of working from one only one end to swap numbers. Another advantage to this sorting algorithm is that it is all done in place, so there is no combining new arrays when returning from the left and right sub-arrays meaning there is no extra memory allocated. This is beneficial when the size of list become larger and memory is expensive. However, one of the downsides to quicksort is that it is not stable, meaning it doesn't preserve the order of equal items in a list as it sorts. Also, given that this method does not use additional memory, it still has to use a large part of the stack to function correctly.

```java
protected int[] hoarePartition(int[] numList, int low, int high) {
    Random random = new Random();
    int index = random.nextInt(high+1-low) + low;
    int pivot = numList[index];
    int i = low;
    int j = high + 1;
    int basicOpCount = 0;

    //swap the lowest index with the pivot
    swap(numList, index, low);
    while (true) {
        do {
            ++basicOpCount;
            ++i;
            if (i > high) {
                break;
            }
        } while (numList[i] < pivot);
        do {
            ++basicOpCount;
            --j;
        } while (numList[j] > pivot);

        if (i < j) {
            //Swap the numbers so the lower number
            // goes to the left and higher goes to
            // the right
            swap(numList, i, j);
        } else {
            int[] results = new int[2];
            //Swap the pivot back into the middle
            // of the left and right side of the list
            swap(numList, j, low);
            results[0] = j;
            results[1] = basicOpCount;
            return results;
        }
    }
}
```

How Quicksort Works

First, it starts by selecting a pivot within the list. There are various ways of effectively selecting a pivot, she of which include randomly selecting, getting the middle index, or selecting the median between the first, middle, and last index. Then, it swaps the pivot with the lowest index's value followed by traversing up the list starting on the new lowest index, low, until it finds a value that is greater than the pivot, and

```java
private int quickSort(int[] numList, int low, int high) {
    int splitIndex;
    int[] results;
    int basicOpCount = 0;

    if (low < high) {
        results = hoarePartition(numList, low, high);
        splitIndex = results[0];
        basicOpCount = results[1];
        basicOpCount += quickSort(numList, low, splitIndex - 1);
        basicOpCount += quickSort(numList, splitIndex + 1, high);
    }
    return basicOpCount;
}
```

down the right side of the list from the highest index, high, until it finds a value less than the pivot. Once these values are found, they are swapped, and it continues to search through the list while swapping values until the left side, i, passes the right side's index, j (when j < i). Then, the pivot is put back in the middle of these two sub-lists by swapping the pivot with j. An example of what this may look like in code may be seen in the examples above. The partition method returns both a count of the basic operation and the index of the pivot that was used. Then the left sub-array is called with a low of index low and the high index of the pivot - 1, while the right sub-array is called with a low of the pivot + 1 and the high index of high, and their respective sub-arrays are recursively called until all the sub-arrays are sorted.

Quicksort in Action

1. The pivot, *p,* is randomly selected.

1	7	5	3^{p}	8	6	2	9

2. Swap *p* with the lowest index.

3^{p}	7	5	1	8	6	2	9

3. Iterate up the left side with *i,* and down the right with *j* until j < i.

3^{p}	7^{i}	5	1	8	6	2	9^{j}

4. Stop when *i > p* and *j < p.*

3^{p}	7^{i}	5	1	8	6	2^{j}	9

5. Swap *i* with *j.*

3^{p}	2^{i}	5	1	8	6	7^{j}	9

6. Stop when *i > p and j < p.*

3^{p}	2	5^{i}	1^{j}	8	6	7	9

7. Swap *i* with *j.*

3^{p}	2	1^{i}	5^{j}	8	6	7	9

8. *j < i* so it now stops traversing.

3^{p}	2	1^{j}	5^{i}	8	6	7	9

9. Swap *p* with *j.*

1^{j}	2	3^{p}	5^{i}	8	6	7	9

10. Repeat steps 1 - 5 for sub-array to the left of split index 3, then to it's left and right sub-arrays recursively.

1	2	3	5	8	6	7	9

11. Repeat steps 1 - 5 for sub-array to the left of split index 3, then to it's left and right sub-arrays recursively.

1	2	3	5	6	7	8	9

12. Return from the recursive calls to a sorted list.

Analysis of Experimental Data for Quicksort

Since quicksort is a divide and conquer algorithm, then we can expect a logarithmic efficiency class given that it divides the problem, in the best case scenario, in half every iteration. Therefore, it will divide the problem into 2 sub problems, where both sides need to be solved for all instances of size n divided into 2. This gives us the recurrence $C(n) = 2C(n/2)+n$ for $n > 1$ and $C(1) = 0$ since there are no comparisons with a list of size 1. Using the master theorem where $a = 2$, $b = 2$, and $d = 1$, we find that $a == b^d$, so we get the $\Theta(n^d*\log(n)) = \Theta(n*\log(n))$ in the best-case scenario. In the worst case, a pivot will be picked that is at an extreme, which will leave one of the two sub-arrays empty. After one iteration through this worst-case scenario, then there are still n-1 integers to be sorted. If an extreme is picked every time, then each integer will have to be compared with every integer left in the sub-array, giving an efficiency of $\Theta(n^2)$. This makes it no better than bubble sort at its worst. In our testing, however, we tested over 100 lists for each list length and averaged the results, so we are testing the average efficiency. After one iteration through the partitioning of the list there will be n + 1 comparisons. Given that the split index, s, for the partition is between 0 and n - 1, then the sub-arrays will be of length s and n - 1 - s. This gives us the recurrence equation of $C(n) = 1/n*\sum^{0 \text{ to } n-1} ((n+1) + C(s) + C(n-1-s))$ for $n >1$, $C(0) = 0$, and $C(1) = 0$. The solution to the average case efficiency ends up being $C(n) \approx 1.39n*\log(n)$. The efficiency is highly contingent on the pivot you select before comparing items in the list, however. There have been many improvements to quicksort that have been found by altering the pivot chosen, some of which include randomly selecting the pivot or selecting the middle index of the list, both of which we tested.

	Size 100,000				Size: 1,000,000			
	Random Pivot		Middle Pivot		Random Pivot		Middle Pivot	
Lists	Avg. Basic Op. Count	Time (ms)	Avg. Basic Op. Count	Time (ms)	Avg. Basic Op. Count	Time (ms)	Avg. Basic Op. Count	Time (ms)
Ascending	2,149,852	9.3	1,600,016	2.7	26,063,239	102.56	19,000,019	35.11
Descending	2,153,186	9.42	1,648,604	3.01	26,204,207	103.45	19,708,730	35.3
Sorted 25%	2,150,472	13.72	2,204,773	9.69	25,771,301	172.21	29,225,807	121.56
Sorted 50%	2,139,651	12.48	2,155,916	7.89	25,729,230	168.5	25,016,665	111.65
Sorted 75%	2,153,016	11.28	2,183,651	6.75	26,027,454	146.68	24,251,253	93.35

	Size 100,000				Size: 1,000,000			
	Random Pivot		Middle Pivot		Random Pivot		Middle Pivot	
Lists	Avg. Basic Op. Count	Time (ms)	Avg. Basic Op. Count	Time (ms)	Avg. Basic Op. Count	Time (ms)	Avg. Basic Op. Count	Time (ms)
Random	2,143,811	15.09	2,093,255	10.83	25,764,095	173.91	25,941,921	124.38
Alternating Pattern	1,572,165	7.72	1,561,130	3.97	18,818,572	81.9	18,818,572	41.98
Random Pattern	1,572,195	8.2	1,568,681	4.24	19,001,182	94.02	19,114,449	47.08
Single Integer	1,561,130	8.34	1,561,130	3.66	18,818,572	86.89	18,818,572	40.69

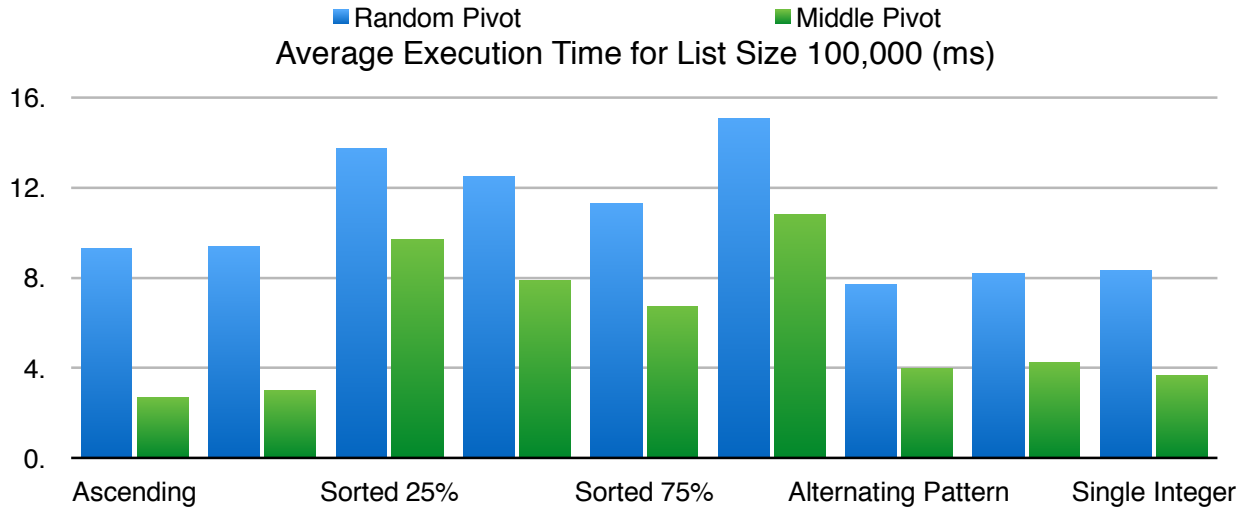
These are the results we recorded after running a variety of lists through each quicksort variation for a total of 100 times before taking the average of each performance. Given the average case efficiency of quicksort being $C(n) \approx 1.39n*\log(n)$, then theoretically we should be getting a basic operation average of 2,308,740 and 27,704,900 for lists of length n = 100,000 and 1,000,000 respectively. In our best-case scenario we get $\Theta(n*\log(n))$, meaning our results should be 1,660,964 and 19,931,568 for lists of size 100,000 and 1,000,000 respectively.

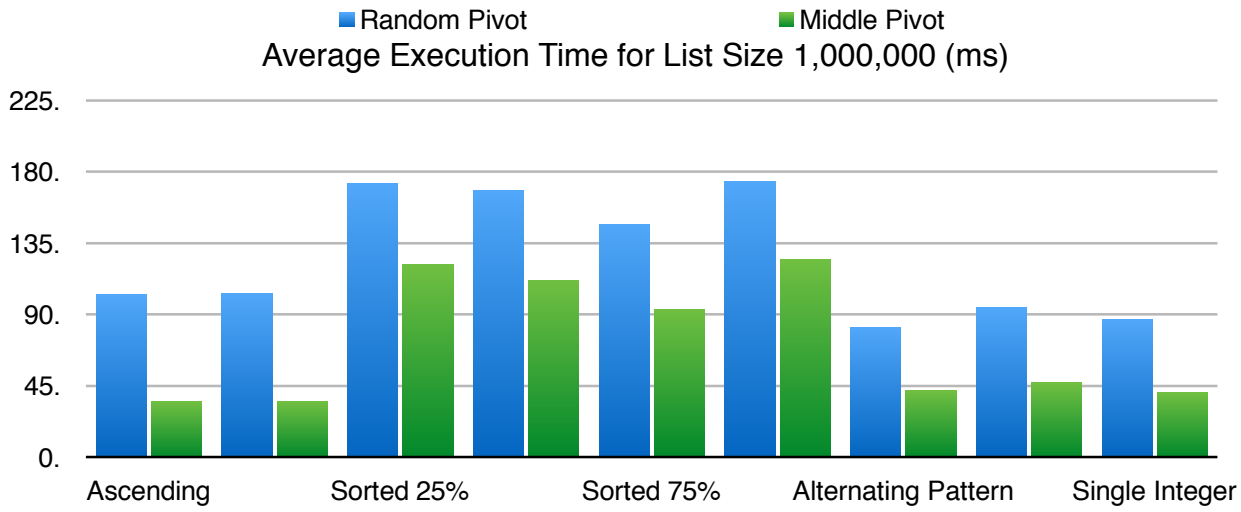
As you can see from our results, the average basic operation count for both sorts lie very close to the expected average performance for sorted 25%-75% and random lists, except in the cases where the list is ascending and descending where the middle index seems to thrive. In both of these lists, the middle pivot quicksort performs near its best-case efficiency. This happens for the ascending list because the middle pivot will always be the middle value for both the left and right sides, so there will be no swapping done and the lists will split equally in half every time since it picks the middle value, similar to merge sort. For the random pivot, it could potentially pick one of the extremes, which would drive its average basic operation count up since it would end up making more comparisons. The same idea is followed for the descending list where the middle pivot quicksort will select the middle value between the right and left side causing it to divide in half every time, where as the random pivot could pick an extreme causing it to make more comparisons.

When we get to the alternating pattern, random patter, and single integer (sorted) list, the basic operation is closer to the best-case performance for both sorts. We can conclude that in the cases where there is an alternating pattern, a random pattern, and a single integer sorted list, quicksort performs better in both methods of quicksort.

As we found form our results, selecting a random pivot is comparable to selecting a middle pivot only in the cases where the lists are not ascending or descending. Both sorts managed to perform as we expected and are accurate representations of our theoretical recurrence relations since our experiments lie close to the expected values. In the case where the list is already sorted in ascending order though, then bubble sort would out perform both of these quicksort variations since it would only have to iterate through he list once to know if it's sorted. Since knowing when a list will be sorted is nebulous, however, then the average case for bubble sort, $O(n^2)$ wouldn't stand a chance against quicksort's average case efficiency.

Although both these quicksort methods perform as expected given the recurrence relation, the middle pivot sort seems out outperform the random pivot in terms of execution time as you can see above. Although they share similar basic operation counts, it seems fair to give the middle pivot sort the upper hand as it performs its sorting substantially faster in every list variation we tested.





Quicksort: Programming Difficulties

One problem we were having while implementing Hoare's partition is that the index i would go out of index every time we ran the sorting method. After running it through the debugger, we found that i would iterate past the end of the list if it was not checked and cause an array out of bounds exception. The reason it doesn't happen to j is because it was always caught when it reaches the pivot at the 0th index. Since the pivot is there, it will always catch j before it goes below the bounds of the array index.

ALGORITHM #2: MERGESORT

What is Mergesort?

Mergesort is a comparison-based sorting algorithm invented by John von Neumann in 1945. It is designed using the idea of divide and conquer where you break a larger problem up into smaller sub problems in order to solve it more efficiently.

The idea is that you are given an array of orderable elements. The first step is to copy the bottom half of the array to a new array, and the top half to another new array. Once you have done this, you will recursively call the algorithm again on each new subarray. Once your subarrays become size one the recursive calls end and you begin merging into new sorted arrays. At this point, you compare elements in one subarray to elements in the other, placing the smallest element at the first available position in the new sorted array. Once you reach the end of one subarray, you know everything left in the other subarray is larger so you just append the remaining elements to the new list. This process is repeated with each recursively created subarray until the first call returns and your entire list has been sorted.

In simplified terms, you continue to split an array in half until there Is only one element in each sub array, at which point you begin merging single element arrays into sorted arrays of size two, then sorted arrays of size two to sorted arrays of size 4, and so on until your left with a sorted array containing all your initial elements. Mergesort is quite efficient with an average and worst-case performance of Θ (n log n). Mergesort is also considered a stable sort with most implementations preserving the input order of equal elements in the sorted output. The number of key comparisons made by Mergesort in the worst case happens to come very close to the theoretical minimum (Discussed in section 11.2 of Levitin: $\lceil log_2 n! \rceil \approx \lceil nlog_2 n - 1{,}44n \rceil$).

While Mergesort is quite powerful, there are some drawbacks. There is a time and space overhead required for using a stack to handle recursive calls. However, there are variations of merge sort that start bottom up by merging pairs of an arrays elements, then merging sorted pairs, and so on. I won't be analyzing this implementation but it is still interesting to talk about. The algorithm also does quite a bit of copying work, copying elements recursively into subarrays and then later into the resulting sorted array.

Mergesort in Action

Here is an example of how merge sort would work on a small array A = [6, 2, 8, 7, 3, 1, 4, 5]

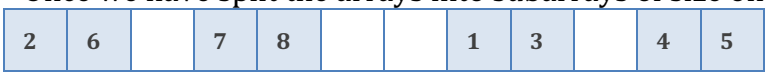
6	2	8	7	3	1	4	5

We start out with our orderable array.

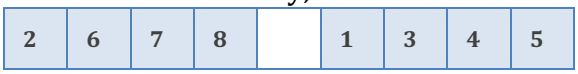
The array is split into a bottom half and top half, then the function is called recursively on each new half.

Each half of the array is then split again into a bottom and top half, then the function is called again recursively on each half of the old halves.

Once we have split the arrays into subarrays of size one, the merge portion begins.



We compare each element to its other half array (denoted by having only one space between them. We see that 2 is less than 6 so we insert 2 first, followed by 6 into our first subarray. Next we see that 7 is less than 8 so we insert 7 first, followed by 8 in our second subarray, and so on.



Once we have our sorted pair arrays, we simply continue the same process, comparing each element in the bottom half to each element in the top half and inserting the smallest element first, followed by the next largest, etc.

We repeat the last step one more time to produce our final sorted array.

Implementation of Mergesort

Below is our implementation of the two functions that comprise Mergesort. These implementations were based off the algorithms described in Levitin with the added functionality of returning our basic operation count. The timing was done outside of these functions by a main function, which simply took the time before and after execution to find the run time.

```java
private int mergeSort(int[] numList) {
    double size = (double) numList.length;
    int firstHalfSize = (int) Math.floor(size/2.0);
    int secondHalfSize = (int) Math.ceil(size / 2.0);
    int[] firstHalf = new int[firstHalfSize];
    int[] secondHalf = new int[secondHalfSize];
    int basicOpCount = 0;

    if (size > 1) {
        System.arraycopy(numList, 0, firstHalf, 0, firstHalfSize);
        System.arraycopy(numList, firstHalfSize, secondHalf, 0, secondHalfSize);
        basicOpCount += mergeSort(firstHalf);
        basicOpCount += mergeSort(secondHalf);
        basicOpCount += merge(firstHalf, secondHalf, numList);
    }
    return basicOpCount;
}

private int merge(int[] firstHalf, int[] secondHalf, int[] numList) {
    int firstHalfSize = firstHalf.length;
    int secondHalfSize = secondHalf.length;
    int firstHalfIndex = 0, secondHalfIndex = 0, listIndex = 0;
    int basicOpCount = 0;

    while (firstHalfIndex < firstHalfSize && secondHalfIndex < secondHalfSize) {
        ++basicOpCount;
        if (firstHalf[firstHalfIndex] < secondHalf[secondHalfIndex]) {
            numList[listIndex] = firstHalf[firstHalfIndex];
            ++firstHalfIndex;
        } else {
            numList[listIndex] = secondHalf[secondHalfIndex];
            ++secondHalfIndex;
        }
        ++listIndex;
    }
    if (firstHalfIndex == firstHalfSize) {
        System.arraycopy(secondHalf, secondHalfIndex, numList, listIndex, secondHalfSize - secondHalfIndex);
    } else {
        System.arraycopy(firstHalf, firstHalfIndex, numList, listIndex, firstHalfSize - firstHalfIndex);
    }
    return basicOpCount;

}
```

Measured Results from Mergesort Testing

Basic Operation Count (Key Compare)

Array Size	Ascending	Descending	25% Sorted	50% Sorted	75% Sorted	Random	Alternating Pattern	Random Pattern	Single Integer
10^5	815,024	853,904	13,565,17	1,153,217	973,510	1,536,267	1,228,640	1,217,685	853,904
10^6	9,884,992	10,066,432	16,539,836	14,279,291	12,144,577	18,674,434	10,066,432	14,597,035	10,066,432

Execution Time (Milliseconds)

Array Size	Ascending	Descending	25% Sorted	50% Sorted	75% Sorted	Random	Alternating Pattern	Random Pattern	Single Integer
10^5	11.78	12.16	17.96	15.97	14.05	20.82	12.71	13.89	12.08
10^6	122.4	131.26	223.32	187.85	157.63	243.91	135.84	157.55	143.24

Analysis of Expected and Gathered Data for Mergesort

When analyzing our experimental data we decided to focus only on arrays of size 100,000 and 1,000,000. This choice occurred because we wanted to reduce the amount that any irregularities may have upon the measured results by increasing our sample size.

For basic operation count, we want to analyze how many times a key comparison occurred on an array of size N. To understand how many key comparisons we should theoretically expect to make in the worst case we need to understand how Mergesort works.

Mergesort consists of the following steps at each level of recursion, simplified from our earlier description:

1. Split the array in half
2. Recursively sort each half
3. Use the merge algorithm to combine the two halves together

The step we are concerned with is step 3. During this step we have to do our key comparisons in order to combine the two halves together into a sorted array. The worst case for comparisons made at this step is n if each half is of size n/2. This is because after each comparison, one element is consumed and placed into our sorted array. At worst we will do n comparisons during step 3. From this information we produce the following recurrence relation:

$$C(1) = 0$$
$$C(n) = 2C\left(\frac{n}{2}\right) + n$$

Following this analysis we find the following pattern (let $n = 2^k$):

$$C(n) = 2^k C\left(\frac{n}{2^k}\right) + kn$$
$$C(n) = 2^{\log_2 n} C\left(\frac{n}{2^{\log_2 n}}\right) + n \log_2 n$$
$$C(n) = n + n \log_2 n$$

From this, we can determine that the maximum theoretical basic operation count for array size 10^5 or 100,000 is:

$$C(100000) = 100000 + 100000 * \log_2 100000 \approx 1760000$$

This is in line with our experimental data, where the random array of size 100,000 produced a basic operation count of 1,536,267, which is slightly below our theoretical maximum, with all other arrays producing even less basic operations. Similarly, with an array of size 10^6 (1,000,000), we observed the following:

$$C(1{,}000{,}000) = 1{,}000{,}000 + 1{,}000{,}000 * \log_2 1{,}000{,}000 \approx 20{,}900{,}000$$

This is also in line with our experimental data, where the random array of size 1,000,000 produced a basic operation count of 18,674,434. Again this data is slightly below our theoretical maximum, with all other arrays falling below the count we found for our random array.

For execution time, we observed that the algorithm always executed in less than 21 milliseconds for size 10^5 and less than 244 milliseconds for size 10^6. Execution time is much harder to analyze in the same detail as basic operation count. This is because it can be subject to many different things including background applications, execution environment, setup times, etc.

Mergesort: Conclusions and Reflections

Mergesort is a fairly simple algorithm to understand and analyze, making our theoretical and experimental results line up quite well. We hypothesized that there may be some irregularities due to the complexity of our testing environment. However, the data that we measured seems to be fairly accurate and helps to support the ideas behind Mergesort accurately.

ALGORITHM #3: HEAPSORT

What is Heapsort?

Heapsort is an efficient sorting algorithm invented by J. W. J. Williams. It works by first converting an input array into a heap, and then repeatedly shifting the largest (first) element in the heaped array to the end of the list, re-heaping the remaining elements afterwards.

How Heapsort Compares

While a little slower than quicksort on most large inputs, heapsort is in O(n log n) in both the average and worst cases, compared with quicksort's unlikely, yet still possible quadratic worst case running time. Its computations on the input are performed in-place and therefore it does not require extra storage based on input size, as is the case with mergesort. Additionally, unlike both quicksort and mergesort, it is not recursive, saving stack space. However, like quicksort, it is not stable, as the heaping process does not preserve ordering of equal elements.

Heapsort in Action

Here is how an array of 10 integers would be sorted using heapsort:

Original array:

4	72	32	60	87	11	22	88	10	48

The first step is to heap the array. A heap is an essentially complete binary tree in which the value of each parent node is larger than either of its children (it may have only one child). This property is called the *parental dominance rule*. A heap structure can be represented in an array A with indices numbered A[1...n], where for each index j such that $1 \leq j \leq n$, $A[j] \geq A[2j]$ and $A[j] \geq A[2j+1]$.

Because arrays of size n are conventionally represented in a program as going from index 0 to index n-1, we have to make some special accommodations so that our index arithmetic will work correctly. One way to do this is to copy the elements of A into a new array B of size n+1, where indices B[1...n] contain the elements of A[0...n-1], respectively, and B[0] is either left unused or contains a "dummy value". In our implementation, to avoid unnecessary array copying, we simply subtract 1 from our calculated index position to obtain the position of the element in the actual array. Thus, a calculated index of [j] in an array indexed [1...n] corresponds to [j-1] in an array indexed [0...n-1].

To clarify, the following is the code for our shiftDown method:

```
// Shifts the element at rootPos down the array until it satisfies the parental
//   dominance rule.
// Returns the basic operation count (key comparison).
  static int shiftDown(int[] numbers, int rootPos, int size) {
    int basicOpCount = 0;
```

```
    int k = rootPos+1;        // The first element is internally represented
    int v = numbers[k-1];     //    as index 1 to aid with arithmetic.
    boolean heap = false;
    while (!heap && 2*k <= size) {
      int j = 2*k;
      if (j < size) {
        basicOpCount++;
        if (numbers[j-1] < numbers[j])   // Corresponds to
          j++;                           //    numbers[j] < numbers [j+1]
      }                                  //    for an array indexed from
      basicOpCount++;                    //    1 to n
      if (v >= numbers[j-1])
        heap = true;
      else {
        numbers[k-1] = numbers[j-1];     // If the parental dominance rule is not
        k = j;                           //    satisfied, swap with the larger child.
      }
    }
    numbers[k-1] = v;
    return basicOpCount;
  }
```

The heap is constructed by repeatedly performing this shift from root positions of floor(n/2)-1 down to 0 (all indices of values greater than floor(n/2)-1 contain leaves):

```
// Converts an input array of integers into a heaped array.
// Returns the basic operation count.
  static int makeHeap(int[] numbers) {
    int basicOpCount = 0;
    for (int i = (int) Math.floor(((double) numbers.length)/2)-1; i >= 0; i--)
      basicOpCount += shiftDown(numbers, i, numbers.length);
    return basicOpCount;
  }
```

After executing makeHeap(A), our array will look like this:

Heaped array:

88	87	32	72	48	11	22	60	10	4

Now that the array is heaped, we can perform the sorting. This requires 3 steps, performed once for each element in the array:

1. Swap the first and last elements. (This puts the largest element last in the array.)
2. Decrease the array size by 1. (The last element is already in its correct position, so we can ignore it for the remainder of the procedure.)
3. Execute shiftDown on the new root element. (Verify its parental dominance. If it doesn't hold, shift it down the heap until it does.)

We will perform the first iteration of these steps on our example array. First, we swap the first and last elements: (red indicates swapped elements)

Swapped first and last elements:

4	87	32	72	48	11	22	60	10	88

Then we decrease the array size by 1. Essentially, this has the effect of "fixing" the last element in place, so that it can no longer be considered: (fixed element is in blue)

Last element fixed:

4	87	32	72	48	11	22	60	10	88

Now, we shift the new first element down until it satisfies the parental dominance rule. We do this by repeatedly swapping it with the larger of its children. Our root node is 4 and its children are 87 and 32. Since 87 is larger than both its parent and its sibling, we swap it with 4:

Swapped with 87:

87	4	32	72	48	11	22	60	10	88

Now 4 is in position 2. Its left child is in position 2*2=4 (72), and its right child is in position 2*2+1=5 (48). 72 is the largest of the three elements, so we swap it with its parent:

Swapped with 72:

87	72	32	4	48	11	22	60	10	88

This puts 4 in position 4. Its children are in positions 2*4=8 and 2*4+1=9. The values of these elements are 60 and 10, respectively. We swap 4 with 60:

Swapped with 60:

87	72	32	60	48	11	22	4	10	88

Now we can see that 4 is in position 8 and is therefore a leaf node, having no children (2*8=16, which is out of bounds). This means the parental dominance of 4 is satisfied by default. We can also see that the resulting array of size 9 is once again in heap form. This means we can start again with our array of smaller size.

We continue in this manner until all of the elements are sorted. We show the content of the array at the end of each successive iteration: (elements swapped during step 3 are in red)

After iteration 2:

72	60	32	10	48	11	22	4	87	88

After iteration 3:

60	48	32	10	4	11	22	72	87	88

After iteration 4:

48	22	32	10	4	11	60	72	87	88

After iteration 5:

32	22	11	10	4	48	60	72	87	88

After iteration 6:	22	10	11	4	32	48	60	72	87	88
After iteration 7:	11	10	4	22	32	48	60	72	87	88
After iteration 8:	10	4	11	22	32	48	60	72	87	88
After iteration 9 (final):	4	10	11	22	32	48	60	72	87	88

As you can see, the output of this algorithm is a sorted array. This concludes our example.

Experimental Analysis of Heapsort

In our analysis of this algorithm, we ran it 100 times each on 9 different randomly generated arrays of size 100,000 and 1,000,000. We averaged the basic operation counts and execution times in milliseconds of these runs to obtain the following data tables.

	Size 100,000		Size 1,000,000	
Array	Basic op count	Exec time (ms)	Basic op count	Exec time (ms)
Random	3,019,555	11.93	36,793,266	205.44
Ascending	3,112,517	8.06	37,692,069	101.3
Descending	2,926,640	7.95	36,001,436	101.55
25% sorted	3,033,630	11.81	36,861,810	167.33
50% sorted	3,046,906	11.5	36,834,816	158.97
75% sorted	3,091,766	9.58	37,100,912	128.44
Equal values	299,994	0.01	2,999,994	6.21
Alternating values	1,552,369	3.67	2,999,994	5.62
Randomly alternating	1,543,066	3.8	18,351,589	50.94

The worst-case basic operation count of heapsort can be calculated by the formula:

$$C_{worst}(n) = 2n\ log_2 n$$

Where n is the size of the input. We can see from this relation that heapsort's asymptotic efficiency is O(n log n). Calculating the worst-case basic operation counts for inputs of sizes 100,000 and 1,000,000 manually, we obtain:

$C_{worst}(100{,}000) \approx 3.32 \times 10^6$

and

$C_{worst}(1{,}000{,}000) \approx 3.99 \times 10^7$

We can see that the experimental data for these input sizes is always lower than these upper bounds, although it frequently approaches them. It turns out that the average case efficiency of heapsort is bounded below by half the value of the worst case:

$$C_{avg} \geq \frac{1}{2} C_{worst}$$

When the array contains all equal elements, we get the best case of heapsort, because every node satisfies the parental dominance rule, and therefore there is no need to shift elements down the heap in step 3.

Note also the execution times of the algorithm for each array; for inputs of size 1 million, heapsort consistently took less than 300 ms (.3 seconds) to execute on our machine. Compare this with a brute force algorithm like insertion sort (considered to be better than most other naïve sorting algorithms), which takes about 3 minutes to sort arrays of the same size on the same computer!

Heapsort: Implementation Difficulties

The main difficulty in using heapsort stems from the necessity of indexing the input arrays from 1 to n, rather than 0 to n-1. However, this sneaky specification is easy to implement once a deeper understanding of the algorithm's inner mechanical behavior is acquired.

CONCLUSION AND REFLECTION

Overall, each of these algorithms is quite efficient, substantially more so than brute force algorithms. Even so, each has unique pros and cons. Mergesort may be the "fastest", but consumes extra storage proportional to input size. On the other hand, quicksort and heapsort are performed in place, but lack the stability of mergesort. The obvious implication of these findings is that, as of yet, the "perfect sort" (if such a thing exists) has yet to be discovered. Overall, however, the authors of this document believe quicksort to be the most promising algorithm to date. Its versatility, speed, and elegantly simple design have led many leaders in the field of computer science to the same conclusion. From our results, we found that quicksort was able to sort the same lists as mergesort and heapsort at quicker speeds even though it may have had a higher basic operation average. Consequently, many variations to the basic quicksort approach have been developed in hopes of capitalizing on its strengths and minimizing its weaknesses. These continued improvements are bringing us closer and closer to the theoretical lower bound for the sorting problem. However, as previously stated, no one-size-fits-all method for sorting has yet been devised; a sorting algorithm for a particular application should be chosen with great care after carefully weighing the costs and benefits of each with regards to the requirements of the project.