Analysis of Sorting Algorithms

Chamara Paul

CPSC 420 Algorithms

23 March 2004

# I. Introduction

In computer science and mathematics, a sorting algorithm is a sequence of computational steps that puts elements of a list into order by means of a certain ordering, often in an ascending or descending order. Efficient sorting is important to optimizing the use of other algorithms that require sorted lists to work correctly; it is also often used for putting data in canonical order and for producing human-readable output. In this analysis of sorting algorithms, the standard sorting algorithms, Insertion sort, Merge sort, Heapsort, Quicksort, and Radix sort will be demonstrated and tested for its efficiencies and inefficiencies of running time.

# II. Assumptions and Formulations

Sort algorithms are often classified by computational complexity in terms of the size of the list (*n*) and its running time. The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed. Typically, the order of growth of a function is relatively efficient if it is on the order of O(*n* log *n*). A function is inefficient if its running time is O(*n*2). Sort algorithms which only use an abstract key comparison operation always need at least O(*n* log *n*) comparisons on average; sort algorithms which exploit the structure of the key space cannot sort faster than O(*n* log *k*) where *k* is the size of the key space.

# Insertion sort

Insertion sort is a simple sort algorithm in which the sorted array (or list) is built one entry at a time. It is much less efficient than the more advanced algorithms such as Merge sort, Quicksort, or Heapsort, but its advantages are:

* Simple to implement
* Efficient on (quite) small data sets
* Efficient on data sets which are already substantially sorted
* Stable
* Does not suffer from poor ‘worst case input’ performance
* Minimal memory requirements

In abstract terms, an element is compared to all the prior elements until a lesser element is found. The choice of which element to remove from the input is arbitrary and can be made using almost any choice algorithm. The inefficiency of the insertion sort stems from the fact that it moves elements only one position at a time.

Sorting is typically done in-place. The result array after *k* iterations contains the first *k* entries of the input array and is sorted. In each step, the first remaining entry of the input is removed, inserted into the right position, thus extending the result as demonstrated in figure 1.

(a)

|  |  |  |  |
| --- | --- | --- | --- |
| Sorted partial result | | Unsorted data | |
| <= *x* | > *x* | *x* | ... |

(b)

|  |  |  |  |
| --- | --- | --- | --- |
| Sorted partial result | | | Unsorted data |
| <= *x* | *x* | > *x* | ... |

**Figure 1:** In the operation of insertion sort, (a) becomes (b) with each element > x copied to the right as it is compared against x.

The algorithm can be described as:

1. Start with the result being the first element of the input.
2. Loop over the input array until it is empty, “removing” the first remaining (leftmost) element.
3. Compare the removed element against the current result, starting from the highest (rightmost) element, and working left towards the lowest element.
4. If the removed input element is lower than the current result element, copy that value into the following element to make room for the new element below, and repeat with the next lowest result element.
5. Otherwise, the new element is in the correct location; save it in the cell left by copying the last examined result up, and start again from (2) with the next input element.

A simple pseudocode and analysis of this follows:

|  |  |
| --- | --- |
| INSERTION-SORT(A) | *Times* |
| 1         for *j*  2 to *length* [A] | *n* |
| 2         do key  A[*j*] | *n* - 1 |
| 3         # Insert A[*j*] into the sorted sequence A[1 .. *j* - 1] |  |
| 4         *i*  *j* – 1 | *n* - 1 |
| 5         while *i* > 0 and A[*i*] > key | Sum *tj* from *j*=2 to *n* |
| 6         do A[*i* + 1]  A[*i*] | Sum (*tj* -1) from *j*=2 to *n* |
| 7         *i*  *i* – 1 | Sum (*tj* -1) from *j*=2 to *n* |
| 8 A[*i* + 1]  key | *n* – 1 |

The running time of insertion sort is:

T(*n*) = *n* + (*n* – 1) + (*n* – 1) + (*n*(*n*+1)/2 – 1) + (*n*(*n* – 1)/2) + (*n*(*n* – 1)/2) + (*n* – 1)

This worst case running time can be expressed as *an*2 + *bn* + *c* for constants *a*, *b*, *c*, thus a quadratic function of *n*. T(*n*) for insertion sort is O(*n*2). The average case is often roughly the same as the worst case. In the average case *tj* = *j*/2, which turns out to be a quadratic function just like the worst case running time. It takes O(*n*2) time in the average and worst cases, which makes it impractical for sorting large numbers of elements. However, insertion sort’s inner loop is very fast, which often makes it one of the fastest algorithms for sorting small numbers of elements, typically less than ten or so.

From the running time analysis above, a prediction can be made about the running time of insertion sort later in this analysis:

|  |  |  |
| --- | --- | --- |
| Array Size | Average Case | Worst Case |
| 2 | 4 | 4 |
| 4 | 16 | 16 |
| 8 | 64 | 64 |
| 16 | 256 | 256 |
| 32 | 1024 | 1024 |
| 64 | 4096 | 4096 |
| 100 | 10000 | 10000 |
| 1000 | 1000000 | 1000000 |
| 10000 | 100000000 | 100000000 |

# Merge sort

The Merge sort takes advantage of the ease of merging already sorted lists into a new-sorted list. It is a particular good example of the divide and conquer algorithmic paradigm. It starts by comparing every two elements (e.g. 1 and 2, 3 and 4…) and swapping them if the first should come after the second. It then merges each of the resulting two, then sorting the lists of four, and so on. Conceptually, merge sort works as follows:

If the list to be sorted is longer than one item:

* Divide the *n*-element sequence to be sorted into two subsequences of *n*/2 elements each
* Sort the tow subsequences recursively using merge sort
* Merge the two sorted subsequences to produce the sorted answer

This is demonstrated in figure 2.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sorted sequence | | | | | | | | | | | | | | |
| 1 2 2 3 4 5 6 7 | | | | | | | | | | | | | | |
| Merge | | | | | | | | | | | | | | |
| 2 4 5 7 | | | | | | |  | 1 2 3 6 | | | | | | |
| Merge | | | | | | |  | Merge | | | | | | |
| 2 5 | | |  | 4 7 | | |  | 1 3 | | |  | 2 6 | | |
| Merge | | |  | Merge | | |  | Merge | | |  | Merge | | |
| 5 |  | 2 |  | 4 |  | 7 |  | 1 |  | 3 |  | 2 |  | 6 |

**Figure 2:** The operation of merge sort on the array A = <5, 2, 4, 7, 1, 3, 2, 6>. The lengths of the sorted sequences being merged increase as the algorithm progresses from the bottom to top.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the “merge” step. To perform the merging, the procedure MERGE(A, *p*, *q*, *r*) is used, where A is an array and *p*, *q*, and *r* are indices numbering elements of the array such that *p* <= *q* < *r*. This procedure takes O(*n*), where *n* = *r* – *p* + 1 is the number of elements being merged, and it works as shown in the following pseudocode and analysis:

|  |  |
| --- | --- |
| MERGE(A, *p*, *q*, *r*) | *Times* |
| 1         *n*1  *q* – *p* + 1 | 1 |
| 2         *n*2  *p* - *q* | 1 |
| 3         create arrays L[1 .. *n*1 + 1] and R[1 .. *n*2 + 1] | 1 |
| 4         for *i*  1 to *n*1 | *n*/2 |
| 5         do L[*i*]  A[*p* + *i* –1] | *n*/2 |
| 6         for *j*  1 to *n*2 | *n*/2 |
| 7         do R[*j*]  A[*p* + *i* – 1] | *n*/2 |
| 8         L[*n*1 + 1]  *infinity* | 1 |
| 9         R[*n*2 + 1]  *infinity* | 1 |
| 10      *i*  1 | 1 |
| 11      *j*  1 | 1 |
| 12      for *k*  *p* to *r* | *n* |
| 13      do if L[*i*] <= R[*j*] | *n* |
| 14      then A[*k*]  L[*i*] | *n* |
| 15      *i*  *i* + 1 | *n* |
| 16      else A[*k*]  R[*j*] | *n* |
| 17      *j*  *j* + 1 | *n* |

The pseudocode and analysis of merge sort follows:

|  |  |
| --- | --- |
| MERGE-SORT(A, *p*, *r*) | *Times* |
| 1         if *p* < *r* | 1 |
| 2         then *q*  floor (*p* + *r*)/2 | 1 |
| 3         MERGE-SORT(A, *p*, *q*) | T(*n*/2) |
| 4         MERGE-SORT(A, *q*+1, *r*) | T(*n*/2) |
| 5         MERGE (A, *p*, *q*, *r*) | O(*n*) |

The running time of merge sort is:

T(*n*) = T(*n*/2) + T(*n*/2) + O(*n*) = 2T(*n*/2) + O(*n*)

The master theorem yields:

a = 2, b = 2, f(*n*) = O(*n*)

*n*logb a = *n*log2 2 = *n*

Since f(*n*) = *n*log2 2), case 2 of the master theorem can be applied. Thus, T(*n*) = (*n* log *n*). Based on the recurrence of this algorithm, merge sort has an average and worst-case performance of O(*n* log *n*).

From the running time analysis above, a prediction can be made about the running time of merge sort later in this analysis:

|  |  |  |
| --- | --- | --- |
| Array Size | Average Case | Worst Case |
| 2 | 0.602059991 | 0.60205999 |
| 4 | 2.408239965 | 2.40823997 |
| 8 | 7.224719896 | 7.2247199 |
| 16 | 19.26591972 | 19.2659197 |
| 32 | 48.16479931 | 48.1647993 |
| 64 | 115.5955183 | 115.595518 |
| 100 | 200 | 200 |
| 1000 | 3000 | 3000 |
| 10000 | 40000 | 40000 |

# Heapsort

Heapsort is one of the best general-purpose sort algorithms. It has very efficient running time performance on randomly ordered arrays, has well behaved memory use, and its worst-case performance is effectively the same as the average case. Some fast sort algorithms have spectacularly bad worst-case performances, both in timing and in memory use. Heapsort works in-place and the worst case running time to sort *n* elements is O(*n* log *n*). For reasonably large values of *n*, the log *n* term is almost constant, so that the sorting time is close to linear in the number of items to sort.

Heapsort works by determining the largest (or smallest) element of the lists, placing that at the end (or beginning) of the list, then continuing with the rest of the list. This algorithm accomplishes its task efficiently by using a data structure called a heap. A heap is a binary tree where each parent is larger than either of its children. Once the data list has been made into a heap, the root node is guaranteed to be the largest element. It is removed and placed at the end of the list, then the remaining list is “heapified” again.

The parent, left child, and right child can be computed simply:

PARENT(*i*)

return floor *i*/2

LEFT(*i*)

return 2*i*

RIGHT(*i*)

return 2*i* + 1

The MAX-HEAPIFY procedure, which runs in O(log *n*) time, is the key to maintaining the max-heap property. This function lets the value at A[I] “float down” in the max-heap so that the sub tree rooted at index I becomes a max-heap. The pseudocode and analysis follows:

|  |  |
| --- | --- |
| MAX-HEAPIFY(A, *i*, *n*) | *Times* |
| 1 *l*  LEFT(*i* ) | 1 |
| 2 *r*  RIGHT(*i* ) | 1 |
| 3 if *l* <= *heap-size* and A[*l*] > A[*i* ] | 1 |
| 4 then *largest*  *l* | 1 |
| 5 else *largest*  *i* | 1 |
| 6 if *r* <= *heap-size* and A[*r*] > A[*largest*] | 1 |
| 7 then *largest*  *r* | 1 |
| 8 if *largest*  != *i* | 1 |
| 9 then exchange A[*i* ]  A[*largest*] | 1 |
| 10 MAX-HEAPIFY(A, *largest*, *n*) | T(*2n/*3) |

The running time of MAX-HEAPIFY is:

T(*n*) = T(2*n*/3) + 1)

By case 2 of the master theorem, this yields T(*n*) = O(log *n*).

The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max heap from an unordered input array. It does this by going through the remaining nodes of the tree and runs MAX-HEAPIFY on each one. The pseudocode and analysis follows:

|  |  |
| --- | --- |
| BUILD-MAX-HEAP(A) | *Times* |
| 1         *heap-size*[A]  *length*[A] | 1 |
| 2         for *i*  floor *length*[A]/2 downto 1 | 1 |
| 3 do MAX-HEAPIFY(A, *i*) | *n*/2 |

BUILD-MAX-HEAP has a running time of O(*n*).

The Heapsort procedure, which runs in O(*n* log *n*) time, sorts an array in place. The heapsort algorithm can be explained as follows:

* Given array A[1, …, *n*], Let *i*:=floor(*n*/2)+1. The heap property holds for A[*i*, …, *n*] because 2*i* > *n* so none of the elements in this range has a child.
* Apply the heapify function to positions *i*-1, *i*-2, …, 1. Each heapify function extends the heap property by one index leftward until the entire array is a heap.
* Set *i* = *n*.
* The largest (remaining) element is not at A[1]. Swap A[1] and A[*i*] to put it in its correct place. The heap property still holds for A[2, …, *i*-1]. Use heapify to extend it to A[1, ..,*i*-1]. Subtract one from *i*, and repeat this step until the whole array is sorted.

Figure 3 illustrates the operation of heapsort.



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | 1 | 2 | 3 | 4 | 5 |

**Figure 3:** The operation of Heapsort. (a) The max-heap data structure just after it has been built by BUILD-MAX-HEAP. (b-e) The max-heap just after each call of MAX-HEAPIFY in line 5. The value of *i* at the time is shown. Only lightly shaded nodes remain in the heap. (f) The resulting sorted array A.

Here’s the pseudocode and analysis of this:

|  |  |
| --- | --- |
| HEAPSORT(A) | *Times* |
| 1         BUILD-MAX-HEAP(A) | O(*n*) |
| 2         for *i* *length* [A] downto 2 | *n* |
| 3         do exchange A[1]  A[*i*] | *n* |
| 4         *heap-size*[A]  *heap-size* [A] – 1 | *n* |
| 5 MAX-HEAPIFY (A, 1) | O(log *n*) |

The running time for heapsort is:

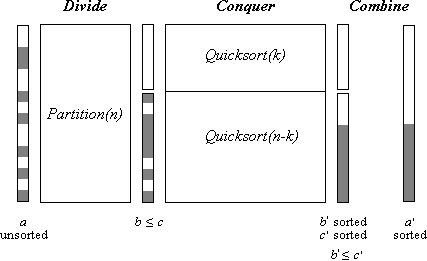
T(*n*) = O(*n*) \* O(log *n*) = O(*n* log *n*)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A complete binary tree with *n* vertices has a depth of at most log(*n*). Therefore, procedure MAX-HEAPIFY requires at most log *n* steps. Procedure BUILD-MAX-HEAP calls MAX-HEAPIFY for each vertex, therefore it requires at most *n* log *n* steps. Heapsort calls BUILD-MAX-HEAP once; then it calls MAX-HEAPIFY for each vertex, together it requires at most 2(*n* log *n*) steps.  From the running time analysis above, a prediction can be made about the running time of heapsort later in this analysis:   |  |  |  | | --- | --- | --- | | Array Size | Average Case | Worst Case | | 2 | 0.602059991 | 0.60205999 | | 4 | 2.408239965 | 2.40823997 | | 8 | 7.224719896 | 7.2247199 | | 16 | 19.26591972 | 19.2659197 | | 32 | 48.16479931 | 48.1647993 | | 64 | 115.5955183 | 115.595518 | | 100 | 200 | 200 | | 1000 | 3000 | 3000 | | 10000 | 40000 | 40000 | |

# Quicksort

Quicksort, like merge sort, is based on the divide-and-conquer paradigm. It is a sorting algorithm that, on average, needs O(*n* log *n*) comparisons to sort *n* items, but in worst case requires O(*n*2) comparisons. Its inner loop is inherently very fast on nearly all computers, which makes it significantly faster than other O(*n* log *n*) algorithms that can sot in-place or nearly so in the average case. This algorithm’s worst-case performance is much worse than some other sorting algorithms such as heapsort or merge sort. Because of its good average performance and simple implementation, quicksort is one of the most popular sorting algorithms in use.

To implement quicksort, first, the sequence to be sorted *a,* is partitioned into two parts, such that all elements of the first part *b* are less than or equal to all elements of the second part *c* (divide). Then the two parts are sorted separately by recursive application of the same procedure (conquer). Recombination of the two parts yields the sorted sequence (combine). Figure 4 illustrates this approach.



**Figure 4:** The operation of quicksort on list *n*.

The key to the algorithm is the PARTITION procedure, which rearranges the sub array in-place. The running time for this procedure is O(*n*). The pseudocode and analysis of this follows:

|  |  |
| --- | --- |
| PARTITION (A, *p*, *r*) | *Times* |
| 1         *x*  A[*r*] | 1 |
| 2         *i*  *p* – 1 | 1 |
| 3         for *j*  *p* to *r* – 1 | *n* |
| 4         do if A[*j*] <= *x* | *n* |
| 5         then *i*  *i* + 1 | *n* |
| 6         exchange A[*i*]  A[*j*] | *n* |
| 7       exchange A[*i* + 1]  A[*r*] | 1 |
| 8 return *i* + 1 | 1 |

The quicksort algorithm uses a recursive divide-and-conquer strategy to sort a list. The steps are:

1. Pick a pivot element from the list.
2. Reorder the list so that all elements less than the pivot precede all elements greater than the pivot. This means that the pivot is in its final place; the algorithms puts at least one element in its final place on each pass over the list. This step is commonly referred to as “partitioning”.
3. Recursively sort the sub list of elements less than the pivot and the sub list of elements greater than the pivot. If one of the sub lists is empty or contains one element, it can be ignored.

In pseudocode, the complete algorithm in its simplest form follows with its analysis:

|  |  |
| --- | --- |
| QUICKSORT(A, *p*, *r*) | *Times* |
| 1         if *p* < *r* | 1 |
| 2         then *q*  PARTITION(A, *p, r*) | O(*n*) |
| 3         QUICKSORT(A, *p*, *q* – 1) | T(*n*/2) |
| 4 QUICKSORT(A, *q* + 1, *r*) | T(*n*/2) |

The running time of quicksort is:

T(*n*) = T(*n*/2) + T(*n*/2) + O(*n*) = 2T(*n*/2) + O(*n*)

The master theorem yields:

a = 2, b = 2, f(*n*) = O(*n*)

*n*logb a = *n*log2 2 = *n*

Since f(*n*) = *n*log2 2), case 2 of the master theorem can be applied. Thus, T(*n*) = (*n* log *n*). Based on the recurrence of this algorithm, quicksort has an average case performance of O(*n* log *n*).

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, and this in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm’s run time can be as fast as O(*n* log *n*). If the partitioning is unbalanced, however, it can run as slow as O(*n*2).

From the running time analysis above, a prediction can be made about the running time of quicksort later in this analysis:

|  |  |  |
| --- | --- | --- |
| Array Size | Average Case | Worst Case |
| 2 | 0.602059991 | 4 |
| 4 | 2.408239965 | 16 |
| 8 | 7.224719896 | 64 |
| 16 | 19.26591972 | 256 |
| 32 | 48.16479931 | 1024 |
| 64 | 115.5955183 | 4096 |
| 100 | 200 | 10000 |
| 1000 | 3000 | 1000000 |
| 10000 | 40000 | 100000000 |

# Radix sort

The Radix sort is a fast, stable sort algorithm, which can be used to sort items that are identified by unique keys. Every key is a string or a number, and radix sort sorts these keys in a particular ascending or descending order. It does this by taking a list in as a list of binary strings and sorts them on the least significant bit, preserving their relative order. This “bitwise” sort must be stable, otherwise the algorithm will not work.

The algorithm operates in O(*nk*) time, where n is the number of items, and k is the average key length. This algorithm was originally used to sort punched cards in several passes. Radix sort has resurfaced as an alternative to other high performance sorting algorithms like quicksort, heapsort and merge sort, which require O(*n* log *n*) comparisons, where *n* is the number of items to be sorted.

Radix sort typically uses the following sorting order: short keys come before longer keys, and keys of the same length are sorted in ascending or descending order as illustrated in figure 5. This coincides with the normal order of numbers, if the numbers are represented as digit strings.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 8 | 7 |  | 3 | 2 | 1 |  | 3 | 2 | 1 |  | 3 | 2 | 1 |
| 6 | 5 | 4 |  | 6 | 5 | 4 |  | 6 | 5 | 4 |  | 6 | 5 | 4 |
| 3 | 2 | 1 |  | 9 | 8 | 7 |  | 9 | 8 | 7 |  | 9 | 8 | 7 |

**Figure 5:** The operation of radix sort on a list of three 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicated the digit position sorted on to produce each list from the previous one.

A radix algorithms works as follows:

1. Take the least significant digit (or group bits) of each key.
2. Sort the list of elements based on that digit, but keep the order of elements with the same digit (making it a stable sort).
3. Repeat the sort with each more significant digit.

The sort in step 2 is done with the counting sort in this analysis, which works since there are only finitely many digits.

An example of radix sort is as follows:

Sort the list:

190, 55, 85, 40, 3, 44, 602, 77

1. Sorting by least significant digit (1s place) gives:

190, 40, 602, 3, 44, 55, 85, 77

1. Sorting by next digit (10s place) gives:

3, 602, 40, 44, 55, 77, 85, 190

1. Sorting by the most significant digit (100s place) gives:

3, 40, 44, 55, 77, 85, 190, 602

The code for radix sort is straightforward. The following procedure assumes that each element in the *n*-element array A had *d* digits, where 1 is the lowest-order digit and digit *d* is the highest-order digit. The pseudocode and analysis follows:

|  |  |
| --- | --- |
| RADIX-SORT(A, *d*) | *Times* |
| 1         for *i*  1 to *d* | *d* |
| 2 do use a stable sort to sort array A on digit *i* | O(*n*) \* *d* |

The running time for radix sort is:

T(*n*) = O(*n* \* *d*)

When d is constant, T(*n*) = O(*n*).

The stable sort used by radix sort in this analysis is COUNTING-SORT. This algorithm assumes that each of the *n* input elements is an integer in the range 0 to *k*. When *k* = O(*n*), the sort runs in *n*) time. Counting sort determines, for each input element, *x*, the number of elements less than *x*. This information can be used to place element *x* directly into its position in the output array. The pseudocode and analysis of this procedure follows:

|  |  |
| --- | --- |
| COUNTING-SORT(A, B, *k*) | Times |
| 1         for *i*  to *k* | *n* |
| 2         do C[*i*]  | *n* |
| 3         for *j* 1 to *length* [A] | *n* |
| 4         do C[A[*j*]] C[A[*j*]] + 1 | *n* |
| 5         # C[*i*] now contains the number of elements equal to *i* |  |
| 6         for *i*  1 to *k* | *n* |
| 7         do C[*i*] C[*i*] + C[*i* –1] | *n* |
| 8         # C[*i*] now contains the number of elements less than or equal to *i* |  |
| 9         for *j*  *length*[A] downto 1 | *n* |
| 10      do B[C[A[*j*]]] A[*j*] | *n* |
| 11 C[A[*j*]]  C[A[*j*]] – 1 | *n* |

The running time for COUNTING-SORT is:

T(*n*) = O(*n*)

From the running time analyses above, a prediction can be made about the running time of radix sort later in this analysis:

|  |  |  |
| --- | --- | --- |
| Array Size | Average Case | Worst Case |
| 2 | 2 | 2 |
| 4 | 4 | 4 |
| 8 | 8 | 8 |
| 16 | 16 | 16 |
| 32 | 32 | 32 |
| 64 | 64 | 64 |
| 100 | 100 | 100 |
| 1000 | 1000 | 1000 |
| 10000 | 10000 | 10000 |

# III. Experiment Design

**The Wonderful World of Perl… yeah.**

This analysis was done using the programming language, Perl. The issue of Perl’s timing efficiency must be addressed in the design of this analysis. Perl’s set of operators, data types, and control constructs are not necessarily intuitive when it comes to speed and space optimization. Perl is an interpretive language, and many trade-offs were made during Perl’s design, which basically means that while it is easier to use, it is generally slower than compiled languages such as C and C++.

It was not anticipated, however, that this inefficiency would make such a huge difference. While the sorting algorithms were running and attempting to sort arrays of size 100, 000, it became apparent that an array size of 1,000,000 could not run for the duration of this experiment. It took approximately 54 hours to run a single iteration of the sorting algorithms to 100,000, and it was calculated that it would take about half a year for the algorithms to run to 1,000,000! Thus, the maximum array size was cut down to 10,000. To run ten iterations to 10,000, it would take approximately 10 hours (which seemed much more reasonable).

# Algorithm Design

The sorting algorithms implemented in this analysis came directly from *Introduction to Algorithms* by Thomas H. Cormen. However, there is an additional feature added to these algorithms that must be noted. An operation count variable was buried in each sorting algorithm (usually in the deepest for loop) to try to accurately reflect the running time of each sort. This was also to compare the real time results with the predicted results described above.

# IV. Results

|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | Average Case | Worst Case | Actual Case |
| 2 | 0.602059991 | 0.60205999 | 0.4 |
| 4 | 2.408239965 | 2.40823997 | 2.6 |
| 8 | 7.224719896 | 7.2247199 | 13.1 |
| 16 | 19.26591972 | 19.2659197 | 63.6 |
| 32 | 48.16479931 | 48.1647993 | 242 |
| 64 | 115.5955183 | 115.595518 | 1123.4 |
| 100 | 200 | 200 | 2427 |
| 1000 | 3000 | 3000 | 251263 |
| 10000 | 40000 | 40000 | 24990050 |

The running times for the insertion sort algorithms are a lot bigger than expected. This may be due to the range for insertion sort between *n* and *n*2. The larger numbers may also reflect the placement of the Operation Count variable.

|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | Average Case | Worst Case | Actual Case |
| 2 | 0.602059991 | 0.60205999 | 5 |
| 4 | 2.408239965 | 2.40823997 | 12 |
| 8 | 7.224719896 | 7.2247199 | 29 |
| 16 | 19.26591972 | 19.2659197 | 70 |
| 32 | 48.16479931 | 48.1647993 | 167 |
| 64 | 115.5955183 | 115.595518 | 392 |
| 100 | 200 | 200 | 680 |
| 1000 | 3000 | 3000 | 9987 |
| 10000 | 40000 | 40000 | 133631 |

The results for the merge sort algorithm are larger than expected, most likely due to the placement of the Operation Counter.

|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | Average Case | Worst Case | Actual Case |
| 2 | 0.602059991 | 0.60205999 | 3.6 |
| 4 | 2.408239965 | 2.40823997 | 9.6 |
| 8 | 7.224719896 | 7.2247199 | 24.5 |
| 16 | 19.26591972 | 19.2659197 | 62.5 |
| 32 | 48.16479931 | 48.1647993 | 149.8 |
| 64 | 115.5955183 | 115.595518 | 366.5 |
| 100 | 200 | 200 | 635.8 |
| 1000 | 3000 | 3000 | 9595.1 |
| 10000 | 40000 | 40000 | 129196.5 |

The heapsort algorithm also shows larger numbers for the actual results, which will also be attributed to the placement of the Operation Counter.

|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | Average Case | Worst Case | Actual Case |
| 2 | 0.602059991 | 4 | 3 |
| 4 | 2.408239965 | 16 | 8.7 |
| 8 | 7.224719896 | 64 | 25.2 |
| 16 | 19.26591972 | 256 | 68.2 |
| 32 | 48.16479931 | 1024 | 167.2 |
| 64 | 115.5955183 | 4096 | 426.8 |
| 100 | 200 | 10000 | 735.5 |
| 1000 | 3000 | 1000000 | 11732.6 |
| 10000 | 40000 | 100000000 | 167346.3 |

The results of the quicksort algorithm are as expected. The actual number of operations falls within the average to worst case values of this algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | Average Case | Worst Case | Actual Case |
| 2 | 2 | 2 | 10 |
| 4 | 4 | 4 | 20 |
| 8 | 8 | 8 | 40 |
| 16 | 16 | 16 | 80 |
| 32 | 32 | 32 | 160 |
| 64 | 64 | 64 | 320 |
| 100 | 100 | 100 | 500 |
| 1000 | 1000 | 1000 | 5000 |
| 10000 | 10000 | 10000 | 50000 |

The results of the radix sort algorithm are also as expected. The reason why the numbers are five times the predicted running times is because there were five digits in each number that the counting sort had to iterate through.

# V. Conclusions

All of the standard sorts in this analysis appear to run correctly. Any discrepancies in the processing times are most likely a result of the CPU of the machine on which it was utilized. The operation count results of the sorting algorithms were partially what was expected. It was clear that the placement of the Operation Counter effected the differences between the predicted results and the actual results. Overall, the project was successful in that the sorts executed as expected, save for a few minor discrepancies.

# VI. Appendices

# Charts

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Insertion Sort for Size 2 | | |  | Insertion Sort for Size 4 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 1 | 0.00 |  | 1 | 3 | 0.00 |
| 2 | 0 | 0.00 |  | 2 | 0 | 0.00 |
| 3 | 1 | 0.00 |  | 3 | 3 | 0.00 |
| 4 | 0 | 0.00 |  | 4 | 3 | 0.00 |
| 5 | 0 | 0.00 |  | 5 | 3 | 0.00 |
| 6 | 0 | 0.00 |  | 6 | 4 | 0.00 |
| 7 | 0 | 0.00 |  | 7 | 3 | 0.00 |
| 8 | 1 | 0.00 |  | 8 | 2 | 0.00 |
| 9 | 0 | 0.00 |  | 9 | 3 | 0.00 |
| 10 | 1 | 0.00 |  | 10 | 2 | 0.00 |
| Average | 0.4 | 0.00 |  | Average | 2.6 | 0.00 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Insertion Sort for Size 8 | | |  | Insertion Sort for Size 16 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 17 | 0.00 |  | 1 | 67 | 0.00 |
| 2 | 16 | 0.00 |  | 2 | 71 | 0.00 |
| 3 | 18 | 0.00 |  | 3 | 59 | 0.00 |
| 4 | 13 | 0.00 |  | 4 | 74 | 0.00 |
| 5 | 10 | 0.00 |  | 5 | 53 | 0.00 |
| 6 | 8 | 0.00 |  | 6 | 54 | 0.00 |
| 7 | 18 | 0.00 |  | 7 | 57 | 0.00 |
| 8 | 7 | 0.00 |  | 8 | 72 | 0.00 |
| 9 | 8 | 0.00 |  | 9 | 61 | 0.00 |
| 10 | 16 | 0.00 |  | 10 | 68 | 0.00 |
| Average | 13.1 | 0.00 |  | Average | 63.6 | 0.00 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Insertion Sort for Size 32 | | |  | Insertion Sort for Size 64 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 197 | 0.00 |  | 1 | 1030 | 0.00 |
| 2 | 205 | 0.00 |  | 2 | 855 | 0.00 |
| 3 | 263 | 0.00 |  | 3 | 2425 | 0.01 |
| 4 | 236 | 0.00 |  | 4 | 954 | 0.00 |
| 5 | 232 | 0.00 |  | 5 | 1017 | 0.00 |
| 6 | 289 | 0.00 |  | 6 | 906 | 0.01 |
| 7 | 246 | 0.00 |  | 7 | 1065 | 0.00 |
| 8 | 239 | 0.01 |  | 8 | 988 | 0.00 |
| 9 | 241 | 0.00 |  | 9 | 991 | 0.00 |
| 10 | 272 | 0.00 |  | 10 | 1003 | 0.00 |
| Average | 242 | 0.00 |  | Average | 1123.4 | 0.00 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Insertion Sort for Size 100 | | |  | Insertion Sort for Size 1000 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 2512 | 0.00 |  | 1 | 248234 | 0.36 |
| 2 | 2233 | 0.00 |  | 2 | 256768 | 0.36 |
| 3 | 2425 | 0.01 |  | 3 | 242983 | 0.34 |
| 4 | 2692 | 0.01 |  | 4 | 255678 | 0.33 |
| 5 | 2566 | 0.01 |  | 5 | 250027 | 0.35 |
| 6 | 2209 | 0.01 |  | 6 | 249397 | 0.23 |
| 7 | 2383 | 0.01 |  | 7 | 252880 | 0.22 |
| 8 | 2440 | 0.00 |  | 8 | 247851 | 0.22 |
| 9 | 2330 | 0.00 |  | 9 | 251642 | 0.23 |
| 10 | 2480 | 0.00 |  | 10 | 257170 | 0.22 |
| Average | 2427 | 0.01 |  | Average | 251263 | 0.29 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Insertion Sort for Size 10000 | | |  | Insertion Sort | | |
| Trial | Operation Count | Time (seconds) |  | Size | Operation Count | Time (seconds) |
| 1 | 24863970 | 34.96 |  | 2 | 0.4 | 0 |
| 2 | 24958603 | 34.83 |  | 4 | 2.6 | 0 |
| 3 | 24880573 | 34.69 |  | 8 | 13.1 | 0 |
| 4 | 25141933 | 35.49 |  | 16 | 63.6 | 0 |
| 5 | 24736165 | 35.34 |  | 32 | 242 | 0.001 |
| 6 | 24820981 | 22.47 |  | 64 | 1123.4 | 0.002 |
| 7 | 25150620 | 21.33 |  | 100 | 2427 | 0.005 |
| 8 | 25155670 | 21.17 |  | 1000 | 251263 | 0.286 |
| 9 | 25162870 | 21.21 |  | 10000 | 24990050 | 29.551 |
| 10 | 25029115 | 34.02 |  |  |  |  |
| Average | 24990050 | 29.55 |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Merge Sort for Size 2 | | |  | Merge Sort for Size 4 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 5 | 0 |  | 1 | 12 | 0 |
| 2 | 5 | 0 |  | 2 | 12 | 0 |
| 3 | 5 | 0 |  | 3 | 12 | 0 |
| 4 | 5 | 0 |  | 4 | 12 | 0 |
| 5 | 5 | 0 |  | 5 | 12 | 0 |
| 6 | 5 | 0 |  | 6 | 12 | 0 |
| 7 | 5 | 0 |  | 7 | 12 | 0 |
| 8 | 5 | 0 |  | 8 | 12 | 0 |
| 9 | 5 | 0 |  | 9 | 12 | 0 |
| 10 | 5 | 0 |  | 10 | 12 | 0 |
| Average | 5 | 0 |  | Average | 12 | 0 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Merge Sort for Size 8 | | |  | Merge Sort for Size 16 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 29 | 0 |  | 1 | 70 | 0 |
| 2 | 29 | 0 |  | 2 | 70 | 0 |
| 3 | 29 | 0 |  | 3 | 70 | 0 |
| 4 | 29 | 0 |  | 4 | 70 | 0 |
| 5 | 29 | 0 |  | 5 | 70 | 0.01 |
| 6 | 29 | 0 |  | 6 | 70 | 0 |
| 7 | 29 | 0 |  | 7 | 70 | 0 |
| 8 | 29 | 0.01 |  | 8 | 70 | 0 |
| 9 | 29 | 0 |  | 9 | 70 | 0.01 |
| 10 | 29 | 0.01 |  | 10 | 70 | 0 |
| Average | 29 | 0.002 |  | Average | 70 | 0.002 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Merge Sort for Size 32 | | |  | Merge Sort for Size 64 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 167 | 0.01 |  | 1 | 392 | 0.01 |
| 2 | 167 | 0 |  | 2 | 392 | 0.02 |
| 3 | 167 | 0 |  | 3 | 392 | 0.02 |
| 4 | 167 | 0 |  | 4 | 392 | 0.01 |
| 5 | 167 | 0.01 |  | 5 | 392 | 0.02 |
| 6 | 167 | 0 |  | 6 | 392 | 0.01 |
| 7 | 167 | 0.01 |  | 7 | 392 | 0.01 |
| 8 | 167 | 0 |  | 8 | 392 | 0.01 |
| 9 | 167 | 0 |  | 9 | 392 | 0.01 |
| 10 | 167 | 0.01 |  | 10 | 392 | 0.01 |
| Average | 167 | 0.004 |  | Average | 392 | 0.013 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Merge Sort for Size 100 | | |  | Merge Sort for Size 1000 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 680 | 0.04 |  | 1 | 9987 | 2.65 |
| 2 | 680 | 0.03 |  | 2 | 9987 | 2.75 |
| 3 | 680 | 0.03 |  | 3 | 9987 | 2.74 |
| 4 | 680 | 0.03 |  | 4 | 9987 | 2.77 |
| 5 | 680 | 0.03 |  | 5 | 9987 | 2.72 |
| 6 | 680 | 0.02 |  | 6 | 9987 | 2.42 |
| 7 | 680 | 0.03 |  | 7 | 9987 | 2.51 |
| 8 | 680 | 0.02 |  | 8 | 9987 | 2.38 |
| 9 | 680 | 0.03 |  | 9 | 9987 | 2.5 |
| 10 | 680 | 0.03 |  | 10 | 9987 | 2.38 |
| Average | 680 | 0.029 |  | Average | 9987 | 2.582 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Merge Sort for Size 10000 | | |  | Merge Sort | | |
| Trial | Operation Count | Time (seconds) |  | Size | Operation Count | Time (seconds) |
| 1 | 133631 | 489.78 |  | 2 | 5 | 0 |
| 2 | 133631 | 485.77 |  | 4 | 12 | 0 |
| 3 | 133631 | 486.68 |  | 8 | 29 | 0.002 |
| 4 | 133631 | 491.03 |  | 16 | 70 | 0.002 |
| 5 | 133631 | 488.33 |  | 32 | 167 | 0.004 |
| 6 | 133631 | 304.24 |  | 64 | 392 | 0.013 |
| 7 | 133631 | 302.33 |  | 100 | 680 | 0.029 |
| 8 | 133631 | 301.56 |  | 1000 | 9987 | 2.582 |
| 9 | 133631 | 315.91 |  | 10000 | 133631 | 398.361 |
| 10 | 133631 | 317.98 |  |  |  |  |
| Average | 133631 | 398.361 |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Heap Sort for Size 2 | | |  | Heap Sort for Size 4 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 3 | 0 |  | 1 | 10 | 0 |
| 2 | 4 | 0 |  | 2 | 11 | 0 |
| 3 | 3 | 0 |  | 3 | 10 | 0 |
| 4 | 4 | 0 |  | 4 | 10 | 0 |
| 5 | 4 | 0 |  | 5 | 8 | 0 |
| 6 | 4 | 0 |  | 6 | 9 | 0 |
| 7 | 4 | 0 |  | 7 | 10 | 0 |
| 8 | 3 | 0 |  | 8 | 9 | 0 |
| 9 | 4 | 0 |  | 9 | 10 | 0 |
| 10 | 3 | 0 |  | 10 | 9 | 0 |
| Average | 3.6 | 0 |  | Average | 9.6 | 0 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Heap Sort for Size 8 | | |  | Heap Sort for Size 16 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 24 | 0 |  | 1 | 62 | 0 |
| 2 | 23 | 0 |  | 2 | 58 | 0 |
| 3 | 23 | 0 |  | 3 | 62 | 0.01 |
| 4 | 26 | 0.01 |  | 4 | 57 | 0 |
| 5 | 25 | 0 |  | 5 | 66 | 0 |
| 6 | 27 | 0 |  | 6 | 63 | 0.01 |
| 7 | 23 | 0.01 |  | 7 | 64 | 0 |
| 8 | 24 | 0 |  | 8 | 57 | 0 |
| 9 | 25 | 0 |  | 9 | 66 | 0 |
| 10 | 25 | 0 |  | 10 | 70 | 0 |
| Average | 24.5 | 0.002 |  | Average | 62.5 | 0.002 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Heap Sort for Size 32 | | |  | Heap Sort for Size 64 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 148 | 0.01 |  | 1 | 367 | 0.03 |
| 2 | 158 | 0.01 |  | 2 | 360 | 0.02 |
| 3 | 152 | 0 |  | 3 | 369 | 0.03 |
| 4 | 145 | 0.01 |  | 4 | 363 | 0.03 |
| 5 | 153 | 0.01 |  | 5 | 374 | 0.04 |
| 6 | 146 | 0.01 |  | 6 | 371 | 0.02 |
| 7 | 151 | 0.01 |  | 7 | 366 | 0.02 |
| 8 | 146 | 0.01 |  | 8 | 365 | 0.02 |
| 9 | 154 | 0.01 |  | 9 | 366 | 0.03 |
| 10 | 145 | 0.01 |  | 10 | 364 | 0.03 |
| Average | 149.8 | 0.009 |  | Average | 366.5 | 0.027 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Heap Sort for Size 100 | | |  | Heap Sort for Size 1000 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 639 | 0.09 |  | 1 | 9613 | 7.61 |
| 2 | 648 | 0.09 |  | 2 | 9615 | 7.42 |
| 3 | 635 | 0.08 |  | 3 | 9642 | 7.56 |
| 4 | 629 | 0.09 |  | 4 | 9581 | 7.38 |
| 5 | 625 | 0.08 |  | 5 | 9555 | 7.53 |
| 6 | 640 | 0.07 |  | 6 | 9603 | 5.51 |
| 7 | 638 | 0.07 |  | 7 | 9583 | 5.63 |
| 8 | 636 | 0.08 |  | 8 | 9598 | 5.58 |
| 9 | 635 | 0.08 |  | 9 | 9598 | 6.08 |
| 10 | 633 | 0.08 |  | 10 | 9563 | 6.03 |
| Average | 635.8 | 0.081 |  | Average | 9595.1 | 6.633 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Heap Sort for Size 10000 | | |  | Heap Sort | | |
| Trial | Operation Count | Time (seconds) |  | Size | Operation Count | Time (seconds) |
| 1 | 129282 | 2161.51 |  | 2 | 3.6 | 0 |
| 2 | 129218 | 2127.93 |  | 4 | 9.6 | 0 |
| 3 | 129170 | 2120.61 |  | 8 | 24.5 | 0.002 |
| 4 | 129143 | 2142.95 |  | 16 | 62.5 | 0.002 |
| 5 | 129140 | 2124.11 |  | 32 | 149.8 | 0.009 |
| 6 | 129300 | 1206.84 |  | 64 | 366.5 | 0.027 |
| 7 | 129137 | 1212.56 |  | 100 | 635.8 | 0.081 |
| 8 | 129200 | 1265.78 |  | 1000 | 9595.1 | 6.633 |
| 9 | 129135 | 1267.33 |  | 10000 | 129196.5 | 1690.414 |
| 10 | 129240 | 1274.52 |  |  |  |  |
| Average | 129196.5 | 1690.414 |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Quicksort for Size 2 | | |  | Quicksort for Size 4 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 3 | 0.00 |  | 1 | 8 | 0.00 |
| 2 | 3 | 0.00 |  | 2 | 10 | 0.00 |
| 3 | 3 | 0.00 |  | 3 | 8 | 0.00 |
| 4 | 3 | 0.00 |  | 4 | 8 | 0.00 |
| 5 | 3 | 0.00 |  | 5 | 10 | 0.00 |
| 6 | 3 | 0.00 |  | 6 | 8 | 0.00 |
| 7 | 3 | 0.00 |  | 7 | 10 | 0.00 |
| 8 | 3 | 0.00 |  | 8 | 8 | 0.00 |
| 9 | 3 | 0.00 |  | 9 | 8 | 0.00 |
| 10 | 3 | 0.00 |  | 10 | 9 | 0.00 |
| Average | 3 | 0.00 |  | Average | 8.7 | 0.00 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Quicksort for Size 8 | | |  | Quicksort for Size 16 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 27 | 0.00 |  | 1 | 61 | 0.00 |
| 2 | 26 | 0.00 |  | 2 | 67 | 0.01 |
| 3 | 24 | 0.00 |  | 3 | 58 | 0.00 |
| 4 | 22 | 0.00 |  | 4 | 65 | 0.00 |
| 5 | 24 | 0.00 |  | 5 | 86 | 0.00 |
| 6 | 31 | 0.00 |  | 6 | 63 | 0.00 |
| 7 | 23 | 0.00 |  | 7 | 62 | 0.00 |
| 8 | 27 | 0.00 |  | 8 | 84 | 0.00 |
| 9 | 24 | 0.00 |  | 9 | 72 | 0.00 |
| 10 | 24 | 0.00 |  | 10 | 64 | 0.00 |
| Average | 25.2 | 0.00 |  | Average | 68.2 | 0.00 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Quicksort for Size 32 | | |  | Quicksort for Size 64 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 158 | 0.00 |  | 1 | 390 | 0.01 |
| 2 | 162 | 0.01 |  | 2 | 451 | 0.01 |
| 3 | 166 | 0.00 |  | 3 | 445 | 0.01 |
| 4 | 156 | 0.00 |  | 4 | 389 | 0.01 |
| 5 | 154 | 0.01 |  | 5 | 409 | 0.01 |
| 6 | 174 | 0.00 |  | 6 | 474 | 0.01 |
| 7 | 185 | 0.00 |  | 7 | 410 | 0.01 |
| 8 | 154 | 0.00 |  | 8 | 412 | 0.01 |
| 9 | 196 | 0.00 |  | 9 | 431 | 0.00 |
| 10 | 167 | 0.00 |  | 10 | 457 | 0.01 |
| Average | 167.2 | 0.00 |  | Average | 426.8 | 0.01 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Quicksort for Size 100 | | |  | Quicksort for Size 1000 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 844 | 0.03 |  | 1 | 11900 | 2.42 |
| 2 | 691 | 0.03 |  | 2 | 12815 | 2.32 |
| 3 | 748 | 0.02 |  | 3 | 11874 | 2.41 |
| 4 | 745 | 0.02 |  | 4 | 11571 | 2.45 |
| 5 | 710 | 0.01 |  | 5 | 12077 | 2.39 |
| 6 | 701 | 0.03 |  | 6 | 11018 | 1.89 |
| 7 | 715 | 0.02 |  | 7 | 11082 | 1.88 |
| 8 | 793 | 0.02 |  | 8 | 11678 | 1.91 |
| 9 | 741 | 0.02 |  | 9 | 11874 | 2.01 |
| 10 | 667 | 0.02 |  | 10 | 11437 | 2.07 |
| Average | 735.5 | 0.02 |  | Average | 11732.6 | 2.18 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Quicksort for Size 10000 | | |  | Quicksort Analysis | | |
| Trial | Operation Count | Time (seconds) |  | Size | Operation Count | Time (seconds) |
| 1 | 169721 | 468.05 |  | 2 | 3 | 0 |
| 2 | 166652 | 520.64 |  | 4 | 8.7 | 0 |
| 3 | 167316 | 525.63 |  | 8 | 25.2 | 0 |
| 4 | 176258 | 482.99 |  | 16 | 68.2 | 0.001 |
| 5 | 155749 | 505.40 |  | 32 | 167.2 | 0.002 |
| 6 | 157079 | 306.08 |  | 64 | 426.8 | 0.009 |
| 7 | 157516 | 315.70 |  | 100 | 735.5 | 0.022 |
| 8 | 174557 | 281.14 |  | 1000 | 11732.6 | 2.175 |
| 9 | 170303 | 286.61 |  | 10000 | 167346.3 | 396.996 |
| 10 | 178312 | 277.72 |  |  |  |  |
| Average | 167346.3 | 397.00 |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Radix Sort for Size 2 | | |  | Radix Sort for Size 4 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 10 | 0 |  | 1 | 20 | 0 |
| 2 | 10 | 0 |  | 2 | 20 | 0 |
| 3 | 10 | 0 |  | 3 | 20 | 0 |
| 4 | 10 | 0 |  | 4 | 20 | 0 |
| 5 | 10 | 0 |  | 5 | 20 | 0 |
| 6 | 10 | 0 |  | 6 | 20 | 0 |
| 7 | 10 | 0 |  | 7 | 20 | 0 |
| 8 | 10 | 0 |  | 8 | 20 | 0 |
| 9 | 10 | 0 |  | 9 | 20 | 0 |
| 10 | 10 | 0 |  | 10 | 20 | 0 |
| Average | 10 | 0 |  | Average | 20 | 0 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Radix Sort for Size 8 | | |  | Radix Sort for Size 16 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 40 | 0 |  | 1 | 80 | 0 |
| 2 | 40 | 0 |  | 2 | 80 | 0 |
| 3 | 40 | 0 |  | 3 | 80 | 0 |
| 4 | 40 | 0 |  | 4 | 80 | 0 |
| 5 | 40 | 0 |  | 5 | 80 | 0 |
| 6 | 40 | 0 |  | 6 | 80 | 0 |
| 7 | 40 | 0 |  | 7 | 80 | 0 |
| 8 | 40 | 0 |  | 8 | 80 | 0 |
| 9 | 40 | 0 |  | 9 | 80 | 0 |
| 10 | 40 | 0 |  | 10 | 80 | 0 |
| Average | 40 | 0 |  | Average | 80 | 0 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Radix Sort for Size 32 | | |  | Radix Sort for Size 64 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 160 | 0.01 |  | 1 | 320 | 0.01 |
| 2 | 160 | 0 |  | 2 | 320 | 0 |
| 3 | 160 | 0 |  | 3 | 320 | 0 |
| 4 | 160 | 0.01 |  | 4 | 320 | 0 |
| 5 | 160 | 0 |  | 5 | 320 | 0 |
| 6 | 160 | 0 |  | 6 | 320 | 0 |
| 7 | 160 | 0 |  | 7 | 320 | 0 |
| 8 | 160 | 0 |  | 8 | 320 | 0 |
| 9 | 160 | 0.01 |  | 9 | 320 | 0 |
| 10 | 160 | 0 |  | 10 | 320 | 0 |
| Average | 160 | 0.003 |  | Average | 320 | 0.001 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Radix Sort for Size 100 | | |  | Radix Sort for Size 1000 | | |
| Trial | Operation Count | Time (seconds) |  | Trial | Operation Count | Time (seconds) |
| 1 | 500 | 0.01 |  | 1 | 5000 | 0.07 |
| 2 | 500 | 0 |  | 2 | 5000 | 0.07 |
| 3 | 500 | 0.01 |  | 3 | 5000 | 0.08 |
| 4 | 500 | 0.01 |  | 4 | 5000 | 0.08 |
| 5 | 500 | 0.01 |  | 5 | 5000 | 0.07 |
| 6 | 500 | 0 |  | 6 | 5000 | 0.05 |
| 7 | 500 | 0.01 |  | 7 | 5000 | 0.06 |
| 8 | 500 | 0 |  | 8 | 5000 | 0.06 |
| 9 | 500 | 0.01 |  | 9 | 5000 | 0.06 |
| 10 | 500 | 0.01 |  | 10 | 5000 | 0.06 |
| Average | 500 | 0.007 |  | Average | 5000 | 0.066 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Radix Sort for Size 10000 | | |  | Radix Sort | | |
| Trial | Operation Count | Time (seconds) |  | Size | Operation Count | Time (seconds) |
| 1 | 50000 | 1.02 |  | 2 | 10 | 0 |
| 2 | 50000 | 1.01 |  | 4 | 20 | 0 |
| 3 | 50000 | 1.03 |  | 8 | 40 | 0 |
| 4 | 50000 | 1.03 |  | 16 | 80 | 0 |
| 5 | 50000 | 1 |  | 32 | 160 | 0.003 |
| 6 | 50000 | 0.7 |  | 64 | 320 | 0.001 |
| 7 | 50000 | 0.71 |  | 100 | 500 | 0.007 |
| 8 | 50000 | 0.56 |  | 1000 | 5000 | 0.066 |
| 9 | 50000 | 0.77 |  | 10000 | 50000 | 0.864 |
| 10 | 50000 | 0.81 |  |  |  |  |
| Average | 50000 | 0.864 |  |  |  |  |

# Source Code

#!/usr/bin/perl

# sort.pl

#

# Author : Chamara Paul

# Date : 23 March 2004

#

# CPSC 420 Algorithms

# Project 1

#

# Program to test insertion sort, merge-sort, heap-sort, quick-

# sort and radix sort.

use Benchmark;

use POSIX;

use Time::HiRes;

sub COUNTINGSORT

{

my($k, $d, @C) = @\_;

my(@D, @E);

my $size = scalar(@C);

for (my $i = 0; $i < $k; $i++)

{

$E[$i] = 0;

}

for (my $j = 0; $j < $size ; $j++)

{

my $digit = substr $C[$j], -$d, 1;

$E[$digit] = $E[$digit] + 1;

}

for (my $i = 1; $i < $k; $i++)

{

$E[$i] = $E[$i] + $E[$i-1];

}

for (my $j = $size-1; $j >= 0; $j--)

{

my $digit = substr $C[$j], -$d, 1;

$D[$E[$digit]-1] = $C[$j];

$E[$digit] = $E[$digit] - 1;

$opCount++;

}

return @D;

}

sub RADIXSORT

{

my($d, @C) = @\_;

for (my $i = 1; $i <= $d; $i++)

{

@C = COUNTINGSORT(10, $i, @C);

}

return @C;

}

sub PARTITION

{

my($p, $r, @C) = @\_;

my($x, $i);

$x = $C[$r];

$i = $p - 1;

for (my $j = $p; $j <= $r - 1; $j++)

{

$opCount++;

if ($C[$j] <= $x)

{

$i++;

my $temp1 = $C[$i];

$C[$i] = $C[$j];

$C[$j] = $temp1;

}

}

my $temp2= $C[$i+1];

$C[$i+1] = $C[$r];

$C[$r] = $temp2;

return ($i + 1, @C);

}

sub QUICKSORT

{

my($p, $r, @C) = @\_;

my $q;

if ($p < $r)

{

($q, @C) = PARTITION($p, $r, @C);

@C = QUICKSORT($p, $q - 1, @C);

@C = QUICKSORT($q + 1, $r, @C);

}

return @C;

}

sub LEFT

{

my $i = $\_[0];

return $i \* 2 - 1;

}

sub RIGHT

{

my $i = $\_[0];

return 2 \* $i;

}

sub MAXHEAPIFY

{

my($i, @C) = @\_;

$i++;

my $l = LEFT($i);

my $r = RIGHT($i);

my $size = scalar(@C);

my $largest;

if (($l <= $size) && ($C[$l] > $C[$i-1]))

{

$largest = $l;

}

else

{

$largest = $i-1;

}

if (($r <= $size) && ($C[$r] > $C[$largest]))

{

$largest = $r;

}

if ($largest != $i-1)

{

my $temp = $C[$i-1];

$C[$i-1] = $C[$largest];

$C[$largest] = $temp;

@C = MAXHEAPIFY($largest, @C);

}

$opCount++;

return @C;

}

sub BUILDMAXHEAP

{

my @C = @\_;

my $size = scalar(@C)/2;

for (my $i = $size; $i >= 0; $i--)

{

@C = MAXHEAPIFY($i, @C);

}

return @C;

}

sub HEAPSORT

{

my @C = @\_;

my @D;

my $size = scalar(@C);

@C = BUILDMAXHEAP(@C);

for (my $i = $size; $i > 1; $i--)

{

my $temp = $C[0];

$C[0] = $C[$i-1];

$C[$i-1] = $temp;

push @D, pop @C;

@C = MAXHEAPIFY(0, @C);

}

push @D, pop @C;

return @D;

}

sub MERGE

{

my($p, $q, $r, @C) = @\_;

my $inf = INT\_MAX;

my $n1 = $q - $p;

my $n2 = $r - $q - 1;

my(@R, @L);

for (my $i = 0; $i <= $n1; $i++)

{

$L[$i] = $C[$p+$i];

}

for (my $j = 0; $j <= $n2; $j++)

{

$R[$j] = $C[$q+1+$j];

}

$L[$n1+1] = $inf;

$R[$n2+1] = $inf;

my $i = 0;

my $j = 0;

for (my $k = $p; $k <= $r; $k++)

{

$opCount++;

if ($L[$i] <= $R[$j])

{

$C[$k] = $L[$i];

$i++;

}

else

{

$C[$k] = $R[$j];

$j++;

}

}

return @C;

}

sub MERGESORT

{

my($p, $r, @C) = @\_;

if ($p < $r)

{

my $q = int(($p + $r)/2);

@C = MERGESORT($p, $q, @C);

@C = MERGESORT($q+1, $r, @C);

@C = MERGE($p, $q, $r, @C);

}

return @C;

}

sub INSERTIONSORT

{

my @C = @\_;

my $size = scalar(@C);

for (my $j = 1; $j < $size; $j++)

{

my $key = $C[$j];

my $i = $j - 1;

while (($i >= 0) && ($C[$i] > $key))

{

$C[$i+1] = $C[$i];

$i--;

$opCount++;

}

$C[$i+1] = $key;

}

return @C;

}

# Main

my @num = (2, 4, 8, 16, 32, 64, 100, 1000, 10000, 100000, 1000000);

for (my $i = 0; $i < 9; $i++)

{

my(@A, @B, @C);

my($time0, $time1);

for (my $k = 0; $k < 10; $k++)

{

for (my $j = 0; $j < $num[$i]; $j++)

{

push @A, int(rand 65535);

}

@B = @A;

my $size = scalar(@B);

# Insertion Sort

$opCount = 0;

$time0 = new Benchmark;

@C = INSERTIONSORT(@B);

$time1 = new Benchmark;

print "OpCount for Insertion Sort = $opCount\n";

print "Insertion Sort Elapsed Time for Size $num[$i]: ".

timestr(timediff($time1, $time0)).

"\n\n";

# Merge Sort

$opCount = 0;

$time0 = new Benchmark;

@C = MERGESORT(0, $size, @B);

$time1 = new Benchmark;

print "OpCount for Merge Sort = $opCount\n";

print "Merge Sort Elapsed Time for Size $num[$i]: ".

timestr(timediff($time1, $time0)).

"\n\n";

# Heap Sort

$opCount = 0;

$time0 = new Benchmark;

@C = HEAPSORT(@B);

$time1 = new Benchmark;

print "OpCount for Heap Sort = $opCount\n";

print "Heap Sort Elapsed Time for Size $num[$i]: ".

timestr(timediff($time1, $time0)).

"\n\n";

# Quick Sort

$opCount = 0;

$time0 = new Benchmark;

@C = QUICKSORT(0, $size, @B);

$time1 = new Benchmark;

print "OpCount for Quick Sort = $opCount\n";

print "Quick Sort Elapsed Time for Size $num[$i]: ".

timestr(timediff($time1, $time0)).

"\n\n";

# Radix Sort

$opCount = 0;

$time0 = new Benchmark;

@C = RADIXSORT(5, @B);

$time1 = new Benchmark;

print "OpCount for Radix Sort = $opCount\n";

print "Radix Sort Elapsed Time for Size $num[$i]: ".

timestr(timediff($time1, $time0)).

"\n\n";

}

}

print "\n";

# Works Cited

Cormen, Thomas H. *Introduction to Algorithms.* The MIT Press. 2001

Wall, Larry. *Programming Perl*. O’Reilly. 2000

[www.wikipedia.org](http://www.wikipedia.org)

<http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/algoen.htm>