Sorting Algorithms

This chapter is intended to cover various sorting algorithms. We perform various kinds of sorts in our everyday lives, such as sorting a pack of cards, ordering a pile of books, comparing multiple bills, and so on. These sorts are primarily based on intuition. Sometimes, sorting can be an essential part of other algorithms which are used for route optimization. In the current chapter, two different types of sorts will be covered. The first is comparison-based sorting, wherein all the key values of the input vector are directly compared with each other prior to ordering. The second type is non-comparison-based sorting, wherein computations are performed on each key value, and then the ordering is performed based on the computed values. Overall, all algorithms primarily follow the principle of divide and conquer. Some of the ways of dividing covered in the chapter are length-based splits (used in merge sort), pivot-based splits (used in quick sort), and digit- based splits (used in radix sort). The initial part of the chapter deals with comparison-based sorts with three simple and intuitive sorting algorithms, which are relatively slow and have an asymptotic complexity of θ(n 2 ) in average and worst-case scenarios. Then we'll cover algorithms with better asymptotic performance in worst-case scenarios, such as θ(nlog n) . Finally, non-comparison-based sorts are covered, which show better asymptotic performance in worst-case scenarios, such as θ(n) under special conditions. The current chapter will cover the following topics:

Sorting terminology and notation Three Θ(n 2 ) sorting algorithms

Insertion sort Bubble sort

Selection sort

The cost of exchange sorting

Shell sort

Merge sort Quick sort

Page 119

﻿

Sorting Algorithms

Heap sort

Bin sort and radix sort

An empirical comparison of sorting algorithms Lower bounds for sorting

Sorting terminology and notation

In this chapter, the input for any algorithm is a vector of elements (key values) unless stated otherwise. These elements can be of any type: numeric, character, logical, or complex. Consider an input vector V of elements i 1 ,i 2 , … ,i n . These elements are said to be sorted provided their corresponding values satisfy a particular order. In other words, the elements of vector V are said to be sorted in non-decreasing order provided their values satisfy the condition i 1 <i 2 <, … ,<i n .

All the algorithms presented in this chapter can handle the special case of sorting, that is, duplicate elements in a given input vector; however, only some of them perform it optimally. An algorithm is said to be performing optimally provided it retains the original position of duplicate elements without redundantly ordering them, thereby reducing computation time.

The simplest way to compare performances of two algorithms is by assessing their

computational system runtime. Table 5.2 shows the system runtime of sorting algorithms for comparison purposes. The runtime depends on the following factors:

Parameters of the input data

Number of elements in the input vector

Memory size of the elements and their respective keys Allowable range of elements (key values) Original order of the input vector

Parameters of the algorithm (approach) Number of comparisons between keys

Number of swapping or element interchange operations List of vectors that need to be sorted and their frequency of recurrence

Asymptotic analysis — functional forms of runtime

[ 108 ]

Page 120

﻿

Chapter 5

Three Θ(n²) sorting algorithms

The following sections deal with three different sorting algorithms, which require θ(n 2 ) system runtime to execute both average-and worst-case scenarios. These algorithms are simple to implement, but show poor computational performance.

Insertion sort

Consider a vector V of numeric elements which needs to be sorted in ascending order. The most intuitive way is to iterate through the vector of elements and then perform element insertions at relevant positions within the vector, satisfying the ordering criterion. This kind of ordering based on a series of insertions is termed insertion sorting. The following Figure 5.1 illustrates the approach for insertion sorting in which each row represents the modified vector for the corresponding ith iteration. The sorting operation is indicated by the arrows:

Figure 5.1: Illustration of insertion sort

[ 109 ]

Page 121

﻿

Sorting Algorithms

The following is the code in R which performs insertion sorting to arrange a vector in increasing order:

Insertion\_Sort <- function(V,n) {

if(n==0) stop("No elements to sort") for(i in 2:(length(V))) {

val <- V[i] j <- i - 1

while (j >= 1 && val <= V[j]) {

V[j+1] <- V[j] j <- j-1

}

V[j+1] <- val

}

return(V)

}

Let's analyze the code for best-, worst-, and average-case scenarios using the metric as the number of comparisons ( val < V[j] ).

Worst-case : Assume a vector of n elements in decreasing order. The number of comparisons using the first for loop is one, second for loop is two, and so on till n-1 . Thus, the total number of comparisons for the complete execution of sorting is given as follows:

Best-case : Assume a vector of n elements already sorted. The number of comparisons for each of the n-1 for loops is one, as the while condition fails for each iteration. Thus, the total number of comparisons for complete execution of sorting is given as:

[ 110 ]

Page 122

﻿

Chapter 5

Average-case : Assume a vector of n elements in any order. For the sake of simplicity, consider the first half of the elements to be sorted, and the remaining as unsorted. Then, the first half would require only comparisons, while the second half would require comparisons. Thus, the functional form of system runtime for an average-case

2

asymptotically.

Similar to the number of comparisons, the number of swaps ( V[j+1] <- V[j] ) within the while loop is also a measure of an algorithm's performance in assessing its computation runtime. The while conditional loop comprises both comparisons and swaps, wherein the number of swaps is one less than the number of comparisons, as the while loop condition fails for every last iteration of comparison. Thus, the total number of swaps is n-1 less than the total number of comparisons for the complete execution of sorting. Thereby, the functional form of system runtime remains the same for both worst-and average-case scenarios ( θ(n 2 ) ), whereas it becomes 0 for the best-case scenario.

Bubble sort

Unlike insertion sort, bubble sort is non-intuitive, tougher to comprehend, and has poor performance even for the best-case scenario. In every iteration, each element within the vector is compared with the rest of the elements, and the smaller (or larger) element amongst all is pushed toward the first (or last) position, just as a water bubble pops out on the water surface; hence, the algorithm is named as bubble sort. It is a step-by-step approach of comparing adjacent elements and swapping key values.

The following is the R code which performs bubble sorting. The code is implemented in an adaptive format, and performs iterations differently. This is explained in detail below the code along with an illustration ( Figure 5.2 ):

Bubble\_Sort <- function(V,n) {

if(n==0) stop("No elements to sort") for(i in 1:length(V)) {

flag <- 0

for(j in 1:(length(V)-i)) {

if ( V[j] > V[j+1] ) {

val <- V[j]

V[j] <- V[j+1] V[j+1] <- val flag <- 1

}

[ 111 ]

scenario is θ(n ) . So, the average-case scenario is similar to the worst-case scenario

Page 123

﻿

Sorting Algorithms

}

if(!flag) break

}

return(V)

}

As seen in the preceding code, bubble sort comprises two for loops along with a flag condition to keep a check on the swapping condition to avoid any redundant iterations. The inner for loop is meant to check all adjacent element comparisons and undergo the required swapping operations. If the lower-indexed key value is greater than its higher- indexed key value, then the elements within the vector are swapped. Therefore, the highest key value element is pushed toward the end, and the subsequent lower key value elements are pushed toward the start. As the highest value is pushed right-most, the second iteration of the inner for loop does not consider the last key value for comparison with the second- last key value:

Figure: 5.2: Illustration of bubble sort

Subsequently, iterations are performed using the outer for loop, barring each key value one less than the preceding iteration. The flag condition helps in avoiding redundant loops once the vector is intermediately sorted, as can be seen in the preceding illustrated example. Let's analyze the code for best, worst, and average-case scenarios for a number of comparisons ( V[j] > V[j+1] ) without considering the flag condition. Then we can observe that for any order of input vector V , the number of comparisons using both the for loops increases with a factor of 1 for each iteration. Therefore, the asymptote of system runtime using a number of comparisons as an evaluation metric for all the three cases is θ(n 2 ) .

[ 112 ]

Page 124

﻿

Chapter 5

Similar to the number of comparisons, the number of swaps can also be considered as an evaluation metric of an algorithm's system runtime. The number of element swaps in a bubble sort depends on the adjacent values within the vector. Assuming half the number of elements to be unsorted for an average-case scenario, the asymptote would be θ(n 2 ) , estimated using the number of required swaps.

Selection sort

Again, consider a numeric vector which is to be sorted in ascending order. Another intuitive approach for sorting the vector is to first select the smallest element and place it in the first position, then select the second smallest and place it in the second position, and so on. This kind of select and sort approach is termed selection sort. Selection sort follows an iii principle, that is, in the i th iteration, select the i th order element from the vector, and place it in the i th position. This approach boils down to a unique feature wherein in the number of swaps required in each iteration is only one unlike what was observed in bubble sort. In other words, for a vector of length n , only n-1 swaps are required for a complete execution of sorting; however, the number of comparisons are similar to the bubble sort algorithm. In selection sort, the position of the smallest element is first remembered, and then swapped accordingly. Figure 5.3 further illustrates the selection sort for a numeric vector:

Figure 5.3: Illustration of selection sort

[ 113 ]

Page 125

﻿

Sorting Algorithms

The following R code performs selection sorting. This current raw code is implemented using the for loop:

Selection\_Sort\_loop <- function(V,n) { if(n==0) stop("No elements to sort") keys <- seq\_along(V) for(i in keys) {

small\_pos <- (i - 1) + which.min(V[i:length(V)]) temp <- V[i]

V[i] <- V[small\_pos] V[small\_pos] <-temp

}

return(V)

}

As the number of comparisons is similar to the bubble sort, the function form of system runtime is θ(n 2 ) for all three cases. However, we have seen till now that the swapping operations are much lower. This is advantageous in scenarios where the cost of swaps is higher than the cost of comparisons, such as vectors with large elements or long strings. Also, selection sort performs very well in vectors with a large number of elements. This is because the swap operations exchange only the position keys (or pointers) instead of position key values (or elements). Thus, additional space is required to store the position keys (or pointers); however, the return of swapping is much faster. The following illustration shows an example of swapping position keys. Consider a numeric vector with four elements. Fig (a) in Figure 5.4 shows the pre-swapped position keys (pointers) along with values, and Fig (b) in Figure 5.4 shows the post-swapped position keys (pointers) along with values:

Figure 5.4: Example of swapping pointers toward key values

Nevertheless, there is a caveat in the implementation of selection sort. The swapping operation is performed even when the position and order of an element is the same. That is to say, even if the i th order element is present at the i th position, the swapping operation will continue. This can be avoided using a test condition. However, in general, the cost incurred due to the test condition is higher than the cost saved by avoiding swaps. Thus, the functional form of the system runtime based on the number of swaps is θ(n) for all three cases. [ 114 ]

Page 126

﻿

Chapter 5

The cost of exchange sorting

Insertion sort, bubble sort, and selection sort are three typical sorting algorithms which are costly to execute for large size vectors and a large number of recurrent vectors. The reasons pertaining to their higher cost of execution are as follows:

Comparison of only adjacent elements in a vector

Swapping of adjacent elements (except in selection sort) based on comparisons

The swapping of adjacent elements is called an exchange; hence, these three algorithms are also called exchange sorts. The cost of execution of these exchange sorts is the total number of cell-to-cell movements (also known as number of inversions) by each element before forming into a right-order vector.

Consider a vector of length n . Then, the total number of exchanges or inversions possible is equal to the total number of available pairs, that is, . On average, the total number of inversions possible is equivalent to per vector. Thus, we can comment with certainty that any sorting algorithm performing adjacent pair comparisons (and swaps) will have an associated cost of at least ~ Ω(n 2 ) for the average-case scenario.

The following Table 5.1 summarizes the system runtime asymptotes for all the three case scenarios (best, worst, and average) using the number of comparisons and number of swaps for insertion sort, bubble sort, and selection sort algorithms. The system runtime of insertion

2

Table 5.1: Comparison of asymptotic complexities using diﬀerent evaluation metrics

[ 115 ]

sort and bubble sort is θ(n ) across both average and worst-case scenarios:

Page 127

﻿

Sorting Algorithms

Let's continue with other sorting algorithms, which show considerably better performance as compared to the three exchange-sort algorithms.

Shell sort

Shell sort (also called diminishing increment sort) is a non-intuitive (real-life) and a non- adjacent element comparison (and swap) type of sorting algorithm. It is a derivative of insertion sorting; however, it performs way better in worst-case scenarios. It is based on a methodology adopted by many other algorithms to be covered later: the entire vector (parent) is initially split into multiple subvectors (child), then sorting is performed on each subvector, and later all the subvectors are recombined into their parent vector. Shell sort, in general, splits each vector into virtual subvectors. These subvectors are disjointed such that each element in a subvector is a fixed number of positions apart. Each subvector is sorted using insertion sort. The process of selecting a subvector and sorting continues till the entire vector is sorted. Let us understand the process in detail using an example and illustration ( Figure 5.5 ):

Figure 5.5: Illustration of shell sort

[ 116 ]

Page 128

﻿

Chapter 5

Consider a numeric vector V of even length (16 elements) which needs to be sorted in ascending order. Also, let us assume that the subvector split is a multiple of two. Then, the shell sort arrange vector using iterative process as discussed below:

Iteration 1 : Split the entire vector V into eight subvectors of two elements each such that each element within a subvector is eight positions apart, and the first element of all subvectors are in sequence, as shown in ( i 1 ). Then, perform insertion sorting on each subvector separately.

Iteration 2 : Now increase the length of the subvectors by decreasing the splits. Next, split the entire vector V into four subvectors of four elements each such that each element within a subvector is four positions apart, and the first element of all subvectors are in sequence, as shown in ( i 2 ).

Similarly, perform iterations till the length of the subvector equals the entire vector, and finally, culminate the sorting with a normal insertion sort of all the elements. The following R code performs shell sorting on both even and odd length vectors:

Shell\_Sort <- function(V,n) {

if(n==0) stop("No elements to sort") increment=round(n/2) ## as.integer while(increment>0) {

for(i in (increment+1):n) {

temp <- V[i] j=i

while(j >= (increment+1) && V[j-increment] > temp) {

V[j] <- V[j-increment] j <- j-increment

}

V[j] <- temp

}

if(increment==2) {

increment <- 1} else{

increment <- round(increment/2.2)

} }

return(V)

}

Shell sort is an improvement over the insertion sort, as the sorting is performed initially on subvectors before being performed on the entire vector. All the intermediate iterations nearly sort the entire vector prior to the final iteration. Now, the cost of iterating a nearly sorted vector is relatively much cheaper than performing insertion sorting on the raw input vector.

[ 117 ]

Page 129

﻿

Sorting Algorithms

Another way of further improving shell sort performance is by increasing the length of the subvectors in the initial iteration. For example, in the preceding example, we started the iterations from two elements in each subvector, which can be increased to three. The advantages of increasing the length of the initial subvector are as follows: The entire vector would be more nearly sorted for the final iteration The number of iterations would reduce

In R, shell short implementation uses gap as 4 k +3.2 k-1 +1 (with prefix of 1 and k ≥ 1 ) which is a variant from Sedgewick (1986), which has a worst-case scenario of θ(n 4/3 ). T he 1 in the prefix is added to ensure sorting yields correct results. Thus, shell sort performs much better than lone insertion sort asymptotically. Shell sort also demonstrates how special properties of other sorting algorithms can be exploited to enhance their existing performance.

Merge sort

Merge sort follows the principle of divide and conquer, wherein the input vector is first divided into two halves, then each half is independently sorted and later merged into a single sorted vector. Its key features are as follows:

Conceptually simple to understand. Asymptotically better performance. Empirically lower system runtime.

Concept of merge – as mentioned earlier, merge is performed on two sorted subvectors (halves). The first element of each subvector is compared, and the smallest is picked up and placed in the first position of the output vector. Subsequently, the picked-up element is removed from its corresponding

subvector. This process of first element comparison continues till all the elements in both the subvectors become empty, and are orderly filled in the output vector. Requires recursive implementation for effective execution. The following R code recursively implements merge sort:

Merge\_Sort <- function(V) {

if(length(V) == 0) stop("Not enough elements to sort")

## Merge function to sort two halves or sub-vectors merge\_fn <- function(first\_half, second\_half) {

result <- c()

while(length(first\_half) > 0 && length(second\_half) > 0) {

if(first\_half[1] <= second\_half[1]) {

[ 118 ]

Page 130

﻿

Chapter 5

result <- c(result, first\_half[1]) first\_half <- first\_half[-1]

} else {

result <- c(result, second\_half[1]) second\_half <- second\_half[-1]

} }

if(length(first\_half) > 0) result <- c(result, first\_half) if(length(second\_half) > 0) result <- c(result, second\_half) return(result)

}

## Recursively split the parent vector into two halves (sub- vectors)

if(length(V) <= 1) V else {

middle <- length(V) / 2

first\_half <- V[1:floor(middle)]

second\_half <- V[floor(middle+1):length(V)] first\_half <- Merge\_Sort(first\_half) second\_half <- Merge\_Sort(second\_half)

if(first\_half[length(first\_half)] <= second\_half[1]) {

c(first\_half, second\_half)

} else {

merge\_fn(first\_half, second\_half)

} } }

The R code comprises two subcodes. One explains how to execute the merge operation ( merge\_fn ), and the other how to operate the main function ( Merge\_Sort ) recursively. The former function executes the merge operation on two input vectors (or two halves of a subvector), whereas the latter function recursively splits the main vector ( V ) to its lowest possible half ( log n levels of recursion), and accordingly, performs the merge operation.

[ 119 ]

Page 131

﻿

Sorting Algorithms

Figure 5.6 illustrates the methodology of merge sort in operation:

Figure 5.6: Illustration of merge sort

One of the main drawbacks of merge sort is memory management. It requires almost twice the memory required by most of the sorting algorithms. Initially, the main input vector is recursively split into multiple subarrays. These subarrays are again recursively merged into multiple secondary vectors, until a final sorted vector is obtained. Thus, a complete execution requires two sets of supplementary vectors (one while splitting, and the other while merging); a bypass of either step is extremely difficult to implement.

Despite the fact of its recursive implementation, analyzing merge sort asymptotically is not very difficult. The analysis can be divided into two stages:

Stage I : The main input vector ( V ) with n elements is split recursively into n subvectors, as illustrated in Figure 5.6 . The vector V is initially divided into two subvectors each with n/2 elements, which, in turn, is divided into two with n/4 elements each, and so on till all the subvectors have only a single element. Assuming n to be a power of 2, the depth of recursion is log n .

[ 120 ]

Page 132

﻿

Chapter 5

Stage II : The n subvectors are merged iteratively into a final sorted vector, as illustrated in Figure 5.6 . In the first iteration, each subvector with a single element is merged (along with sort) into n/2 subvectors, each with two elements. In the second iteration, merged subvectors are remerged into n/4 sub vectors, each with four elements and so on, until a single sorted vector is obtained. Thereby, the asymptote for each iteration is θ(n) , as n steps are required for its completion.

Thus, the functional form of the system runtime of the merge sort algorithm in terms of the number of execution steps is θ(n log n) , because each log n level of recursion requires an n number of total merge operation steps. As the cost function is independent of the order of the initial input vector, the asymptote for the best, average, and worst-cases remain the same.

Quick sort

The quick sort algorithm is an updated version of the merge sort algorithm with faster in- memory sorting capability. It is widely used in average-case as against worst-case scenarios. It is also efficient in terms of memory utilization, as it does not require the secondary vector when performing the merge operation. Quick sort can be accessed in R using functions such as sort (base) and quick sort (rje). It is also called partition-exchange sort. Like merge sort, quick sort also requires recursive implementation for effective execution.

The following is the three-step execution methodology of the quick sort algorithm for a given input vector V with n elements:

1. Select the pivot or root element of the given input vector. The pivot element is used to partition the entire vector into two subvectors such that all the elements in the first vector or left vector are less than the pivot, and all the elements in the second vector or right vector are greater than or equal to the pivot. However, the elements within both the partitioned subvectors need not be sorted. Usually, the element with the median value is considered for pivot. However, in our

algorithm, we have considered the last element as the pivot for the corresponding vector. The pivot is said to be best when the partitioned subvectors are of the same length, and worst when one of the subvectors is empty.

2. Perform recursive sorting on each of the subvectors (excluding the pivot)

obtained after the split.

3. Join the first sorted subvector, the pivot, and the second sorted subvector to

obtain the final sorted output.

[ 121 ]

Page 133

﻿

Sorting Algorithms

The following R code implements the recursive form of the quick sort algorithm:

Quick\_Sort <- function(V,n) {

if (n <= 1) return(V)

left <- 0 ##start from left prior first element right <- n ##start from rightmost element

v <- V[n] ## initialize last element as pivot element

## Partition implementation repeat {

while (left < n && V[left+1] < v) left <- left+1 while (right > 1 && V[right-1] >= v) right <- right-1 if (left >= right-1) break ## Swap elements temp <- V[left+1]

V[left+1] <- V[right-1] V[right-1] <- temp

}

## Recursive implementation of Quick sort

if (left == 0) return(c(V[n], Quick\_Sort(V[1:(n-1)],n=(n-1)))) if (right == n) return(c(Quick\_Sort(V[1:(n-1)],n=(n-1)), V[n])) return( c(Quick\_Sort(V[1:left],n=left), V[n], Quick\_Sort(V[(left+1):(n-1)],n=(n-left-1)))) }

The R code begins with initializing the left and right indices. The left index represents the position of an element prior to the first element in the vector, and the right index represents the position of the last element in the vector. Then, the last element is considered as the pivot element for the corresponding input vector. Consider the following numeric vector with 16 elements:

Figure 5.7: An example of a 16-elements numeric vector

Now, the left and right indices start moving inward under the repeat loop till the indices meet. The inner while loops checks for bounds along with the pivot element prior to updating the left and right indices. Subsequently, the elements are swapped such that all elements toward the left of the pivot are lower than the pivot element, and all the elements toward the right are higher than the pivot element. However, the elements within the left and right subvectors need not be ordered. Figure 5.8 illustrates the first swap iterations being performed under the repeat loop:

[ 122 ]

Page 134

﻿

Chapter 5

Figure 5.8: Illustration of swap iterations

Once the left index meets the right index, the repeat loop breaks, and the recursive implementation of quick sort begins. Here, the pivot element is correctly positioned, and the remaining elements within the left and right subvectors are subject to recursive sorting. Figure 5.9 illustrates the complete implementation of the quick sort algorithm:

Figure 5.9: Illustration of quick sort

[ 123 ]

Page 135

﻿

Sorting Algorithms

Let's analyze the asymptote of the quick sort algorithm in detail based on the number of operations performed at each step. Consider an input vector V of length n . A total of n moves are required to complete the traversing of both the left and right indices, till they meet each other. The repeat loop can be executed at most n times, and the while loop can fail at most n times. Hence, the asymptote of partition based on the pivot element is θ(n) . Consider a worst-case scenario wherein one of the subvectors has no elements upon partitioning. If this scenario occurs at each partition step, then the asymptote of the algorithm becomes θ(n 2 ) . In our algorithm, the worst-case scenario is bound to happen only when the input vector possesses all the elements in the descending order. However, this situation can be minimized upon random selection of the pivot value.

Consider a best-case scenario wherein for each iteration (of the repeat loop), the pivot value partitions the vector into two equal subvectors. Such a perfect kind of pivot will result in log n levels of partitions and n levels of traversing (by the left and right indices), which corresponds to an asymptote of θ(nlog n) .

Consider an average-case scenario. Here, the behavior of the partition is between the best and worst-case scenarios, and there is an equal likelihood for any type of subvector partition. The asymptote which satisfies the recurrence relation can be defined as follows:

Thus, the closed form solution for the average-case scenario is also θ(nlog n) , which is similar to that of the best-case scenario.

Heap sort

Heap sort is an improvised form of selection sort, wherein the algorithm initially splits the input vector into sorted and unsorted vectors, and then iteratively shrinks the unsorted vector by extracting the largest element and placing it in the sorted vector. It is based on the heap data structure which provides a non-quadratic asymptote even for the worst-case scenarios. Heaps are tree-based data structures with the following properties: Shape criterion : Heaps are primarily complete binary trees with both the left and right child nodes filled with values:

[ 124 ]

Page 136

﻿

Chapter 5

Figure 5.10: Illustration of complete and in-complete binary tree

Heap criterion : The ordering of the tree is unidirectional. In other words, all the parent nodes will be greater than the child nodes (max-heap), or all the child nodes will be greater than the parent nodes (min-heap). Either of the heaps can be used for sorting in any required order. In our example, we will use max-heap to sort the input vector in an ascending order. Also, the values in the nodes are independent of each other. It is possible that all the values of the nodes in a right sub-tree are higher than the values of the nodes in the left sub-tree:

Figure 5.11: Illustration of max-heap and min-heap

The heap sort algorithm possesses some structural advantages which enhance its performance efficiency. It adopts the concept of a complete binary tree wherein the tree is balanced. It requires less in-memory, as the values of the input vector are directly stored in the form of a binary tree. The values need not be explicitly inserted into each of the nodes within the tree. Hence, it is also suitable for large size vectors. The asymptotic performance is also non-quadratic in the best, average, and worst-case scenarios. The functional form of the system runtime is nlog n .

[ 125 ]

Page 137

﻿

Sorting Algorithms

The heap sort algorithm is quite easy to implement. The input vector array is first converted into a max-heap ( max\_heap function). Then, the maximum value from the heap is extracted iteratively, and placed at the end of the array, ensuring that the order of the heap remains intact. Consider a vector of length n wherein all the elements are positioned from 1 to n . The first extracted maximum element will be placed in the n th position, the second extracted maximum element will be placed in the (n-1) th position, and so on. The extraction continues till the heap becomes empty.

The following is the R code which implements the recursive form of the heap sort algorithm:

Heap\_Sort <- function(V) {

heapsize <- length(V) ## Initialize with total vector size for (i in floor(length(V)/2):1)

V <- max\_heap(V, i,heapsize) ## Build initial max-heap for (i in length(V):2) {

temp <- V[i] ## replace ith with 1st element (maximum) V[i] <- V[1] V[1] <- temp

heapsize <- heapsize -1 ##Reduce size of input vector V <- max\_heap(V, 1,heapsize) ##Re-build max-heap with reduced

input vecto0072

}

return(V)

}

## Following function recursively builds max-heap max\_heap <- function(V, i,heapsize) {

left <- 2\*i

right <- 2\*i+1

if (left<=heapsize && V[left]>V[i]){ ## build left sub-tree

largest <- left} else{ largest <- i }

if (right<=heapsize && V[right]>V[largest]) largest <- right ## build right sub-tree if (largest != i) {

temp2 <- V[largest] ##replace largest with ith element V[largest] <- V[i] V[i] <- temp2

V <- max\_heap(V, largest,heapsize) ## Recursive run

}

return(V)

}

[ 126 ]

Page 138

﻿

Chapter 5

Figure 5.12 illustrates the step-by-step implementation of the heap sort algorithm. The first step shows the original vector V with 11 elements, which need to be sorted in ascending order. The second step shows the initial max-heap with the largest element in the first node. The third step shows the extraction of the largest element (here, 88 ). The extracted element is then placed in the last position of the array. The max-heap tree is again built with the new largest element as its first node. The fourth step shows the extraction of the corresponding largest element (here, 65 ). The extracted element is then placed in the second last position of the array. The max-heap tree is rebuilt with a new largest element as its first node:

Figure 5.12: Step-by-step illustration of heap sort

[ 127 ]

Page 139

﻿

Sorting Algorithms

These steps continue till all the elements from the max-heap tree are extracted and placed in the relevant positions. The final sorted vector is as follows:

Figure 5.13: Final vector positions of heap sort

Now let's analyze the runtime performance of the algorithm assuming a vector of length n . The max-heap recursive function requires θ(n) runtime, and n extractions of the largest element require θ(log n) runtime. Thus, the total runtime of the heap sort algorithm for the best, average, and worst-case scenarios is θ(nlog n) .

Bin sort and radix sort

Bin sort is one of the most efficient algorithms, wherein an input vector is split into multiple bins, and then sorting is performed within each bin. The elements are assigned to the bins based on the computations performed on each element. The bins can be a list of multiple vectors or a linked list. The current execution uses a list of multiple vectors as bins. The following R code performs the bin sort operation on a numeric vector ( V ) containing n elements. The maxValue variable denotes the element with maximum value within the input vector:

Bin\_Sort=function(V,n,maxValue){

bin <-list("binValues"=list(), "nElement"=NA) ## create empty bins for(i in 1:n){

bin[["binValues"]][[i]]<-NA bin[["nElement"]][i]<-0

}

## add elements into suitable bins

bin <- addItem(V=V,bin=bin,maxValue=maxValue,n=n) ## bind all bins into a single sorted vector output <- bindSorted\_vec(bin=bin,n=n) return(output)

}

Initially, an empty bin is created, which contains a list ( binValues ) and a vector ( nElement ). The list ( binValues ) is meant to act as bins to hold elements of the input vector ( V ), and the vector ( nElement ) is meant to track the count of elements in each bin.

[ 128 ]

Page 140

﻿

Chapter 5

The functions addItem and insertItem are meant to allocate each element into bins in a sorted order. The function insertItem gets activated when a new element is being inserted into a bin already containing elements. While inserting, the value of the new element is compared with the existing elements. Accordingly, the position is assigned to the new element, ensuring the sorting order (ascending in our case):

# add item to bin

addItem=function(V,bin,maxValue,n){

for(i in 1:n){

val<-V[i]

ix<-ceiling((val\*n)/maxValue)

if(is.na(bin[["binValues"]][[ix]][1])){

bin[["binValues"]][[ix]][1]<-val bin[["nElement"]][ix]<-1

} else {

bin <- insertItem(val=val, ix=ix,bin=bin)

} }

return(bin)

}

# insert a item into a bin ensuring sorting insertItem=function(val, ix,bin){ nElement<-bin[["nElement"]][ix] pos<-NULL

for(i in 1:nElement){

if(val<bin[["binValues"]][[ix]][i]){

pos<-i

} }

if(is.null(pos)){

bin[["binValues"]][[ix]][nElement+1]<-val

} else if(pos==1) {

bin[["binValues"]][[ix]]<-c(val, bin[["binValues"]][[ix]][1])

} else {

bin[["binValues"]][[ix]]<-c(bin[["binValues"]][[ix]][1:(pos- 1)], val, bin[["binValues"]][[ix]][pos:nElement])

}

bin[["nElement"]][ix]<-nElement+1 return(bin)

}

[ 129 ]

Page 141

﻿

Sorting Algorithms

Some of the key features of the functions addItem and insertItem are as follows: Direct computations are performed on the element values prior to assigning them to each bin. The computation depends on the length of the input vector ( n ) and the maximum value in the input vector ( maxValue ). This also restricts the input vector to be of integer type rather than numeric.

The length of the binValue list is restricted to n . In other words, the total number of bins is n .

The vector nElement keeps track of the elements in each bin.

The function insertItem intrinsically ensures sorting among the elements within each bin. Whenever a new element needs to be inserted, its position is first determined based on its value, and then inserted accordingly.

Once all the elements of the input vector are allocated to the respective bins, the bins are then bound into a single output vector in an order from 1 to n . During the bind process, the relative positions of the elements within each bin are maintained. Hence, the output received is a completely sorted vector (ascending order in our case):

# bind the list into a sorted vector bindSorted\_vec=function(bin,n){

output <- c() currentIx<-1

for(i in 1:n){

if(!is.na(bin[["binValues"]][[i]][1])){

nElement<-bin[["nElement"]][i] for(m in 1:nElement){

output[currentIx]<-bin[["binValues"]][[i]][m] currentIx<-currentIx+1

} } }

return(output)

}

The following example shows the working of the bin sort algorithm in R:

> V<-c(20,12,65,8,10,16,43,35,23,88,2,56,41,27,67,55) > n<-16

> maxValue<-88

> Bin\_Sort(V=V,n=n,maxValue=maxValue)

[1] 2 8 10 12 16 20 23 27 35 41 43 55 56 65 67 88

[ 130 ]

Page 142

﻿

Chapter 5

The performance of the bin sort algorithm is θ(n) for most of the scenarios. It is evaluated based on the number of operations required to place an element into a bin and then taking out all the elements from the bins into an output vector. However, when the input vector becomes very large, the number of traversing operations required for placement of each element increases considerably, and the performance is drastically affected.

Bucket sort is another representation of the bin sort algorithm, wherein the elements are initially assigned to each bin, and each bin is subjected to a different sorting technique. There is also no initial check on the elements being inserted into non-empty bins. Once all the elements are placed into their respective bins based on a computation criterion, each bin is then exposed to a different sorting algorithm. These individually sorted bins are later bound into a single vector of sorted elements.

Radix sort, on the other hand, is an improvised version of bin sort, wherein the number of bins can be restricted to a smaller number (generally 10 bins), and relative positioning of elements while assigning them into non-empty bins is not required. Consider a vector of n elements ranging from 0 to 999 which needs to be sorted in ascending order. Let us also define bins from 1 to 10 such that bin 1 is meant to store elements with the digit 1, bin 2 is meant to store elements with the digit 2, and so on. We can begin assigning elements to each bin based on their units digit. If the units digit of an element is 1, then the element will be placed in bin 1, and if the units digit is 0, the element will be placed in bin 10, and so on. Also, while inserting elements into non-empty bins, the relative positions need not to taken into account as was the case in bin sort. Once all the elements are inserted into the respective bins based on their units digit, all the 10 bins will then be bound into a single vector (without disturbing the overall order of the bins, that is, the first bin follows the second bin, which follows the third bin, and so on) using the bindSorted\_vec function. Similarly, the process continues for the tens digit and the and hundreds digit. The following R code implements the radix sort algorithm:

# add item to bin

addItem=function(V,bin,digLength,n){

for(i in 1:n){

val<-V[i]

## Extract the required digit from the number ix<-floor((val/digLength) %% 10)+1 ## Assign element to each bin

bin[["binValues"]][[ix]][bin[["nElement"]][ix]+1]<-val ## Track count of elements in each bin

bin[["nElement"]][ix]<-bin[["nElement"]][ix] + 1

}

return(bin)

}

# bind the list into a sorted vector

[ 131 ]

Page 143

﻿

Sorting Algorithms

bindSorted\_vec=function(bin){

output <- c() currentIx<-1

for(i in 1:10){

if(!is.na(bin[["binValues"]][[i]][1])){

nElement<-bin[["nElement"]][i] for(m in 1:nElement){

output[currentIx]<-bin[["binValues"]][[i]][m] currentIx<-currentIx+1

} } }

return(output)

}

# radixsort Algorithm

radix\_Sort=function(V,n,maxValue,digLength){

for(digLength in c(10^(0:digLength))) {

bin <-list("binValues"=list(), "nElement"=NA) # create empty bins for(i in 1:10){

bin[["binValues"]][[i]]<-NA bin[["nElement"]][i]<-0

}

bin <- addItem(V=V,bin=bin,digLength=digLength,n=n) V <- bindSorted\_vec(bin=bin) }

return(V)

}

The following example shows the working of the radix sort algorithm in R:

> V<-c(67,54,10,988,15,5,16,43,35,23,88,2,103,83) > n<-14

> maxValue<-988 > digLength <- 2

> radix\_Sort(V=V,n=n,maxValue=maxValue,digLength=digLength) [1] 2 5 10 15 16 23 35 43 54 67 83 88 103 988

[ 132 ]

Page 144

﻿

Chapter 5

Figure 5.14 , 5.15 , and 5.16 illustrate the implementation of the radix sort algorithm. Consider an integer vector ( V ) with 14 elements, with the maximum element as 988, and the length of digits as 2 (one less than the length of the maximum element):

Figure 5.14: Iterations 0 and 1 of radix sort

Iteration 0 in radix sort uses the units digit from rightmost to arrange data in bins. For example, 10 with 0 in right most goes to the first bin and 43 goes to the third bin. Similarly, next iteration will use tens digit as shown in Figure 5.15 :

Figure 5.15: Iteration 2 of radix sort

[ 133 ]

Page 145

﻿

Sorting Algorithms

The output from the tens digit is then reallocated using the hundreds digit (leftmost digit) as shown in Figure 5.16 :

Figure 5.16: Iteration 3 of radix sort

Now, let's analyze the performance of the radix sort algorithm. The asymptote of radix sort is θ(n) for all types of best, worst, and average-case scenarios irrespective of the length of the input vector. The asymptote primarily depends on the maximum number of digits for a given input vector and the base of the computation. In our algorithm, we have used a base of 10 for performing computations on each element prior to assigning them to the respective bins. The asymptote can be rewritten as θ(nk + sk) , where n represents the total length of the input vector, s represents the base, and k represents the length of the maximum element in the input vector. However, if the length of the input vector is large and most of the values are distinct, then the asymptotic complexity of radix sort changes to Ω(nlog n) . Also, if the range of elements is large, then the radix sort algorithm will show its best performance in terms of the Ω(nlog n) asymptote.

[ 134 ]

Page 146

﻿

Chapter 5

Nevertheless, the radix sort algorithm is very difficult to implement efficiently. The implementation requires a number of loop iterations, which affects the runtime

performance of the algorithm. The following loops form an integral part of radix sort, which is shown in the preceding three images:

Loop to initialize the position of the digit ( digLength ) for an element Loop to create empty bins

Loop to perform radix/index computation on each element prior to assigning elements into the respective bins, and to keep a track on the count of elements within each bin

Loop to extract elements from each bin and assign them to an output vector

Also, radix sort is limited to the integer type of input vectors. Vectors with real numbers and arbitrary element lengths need to be handled with extra care.

An empirical comparison of sorting algorithms

Empirical comparison analysis intends to evaluate the performance of algorithms based on the system runtime. Many algorithms might possess the same asymptote complexity, but their performance might differ based on the size of the input vector. Empirical analysis is performed on the underlying assumption that the system properties and configuration remain the same for all the running algorithms under consideration.

Table 5.2 shows the system runtime for actual implementation of sorting algorithms measured using microbenchmark in R:

Table 5.2: Empirical comparison of sorting algorithms using system conﬁguration of 2.8-GHz Intel i7 CPU running Windows. The system runtime is shown in milliseconds

[ 135 ]

Page 147

﻿

Sorting Algorithms

The input used for empirical analysis is a random vector of integers of various lengths ranging from 10, 100, 1,000 to 10,000. The input for the best-case scenario is an increasing sorted vector of length 1,000. Similarly, the input for the worst-case scenario is a decreasing sorted vector of length 1,000. We can observe that the performance of some algorithms is agnostic of the best and worst-case input. The following are some takeaways from the preceding Table 5.2 :

Algorithms with asymptotic complexity of O(n 2 ) perform poorly for large length of input vectors. Shell sort shows superior performance. Bubble sort shows the worst performance unless the input is a best-case (sorted).

Among the algorithms with asymptote complexity O(nlog n) , heap sort is the worst performer except in the best-and worst-cases due to overhead of class structure (heaps).

Overall, radix sort shows a consistently good performance across all lengths of input vectors as compared to other algorithms.

Lower bounds for sorting

So far, we have covered performance assessment of algorithms based on their time complexity (number of operations). Empirical analysis shows the performance based on actual system runtime, while asymptotic analysis evaluates the performance based on the number of operations (or comparisons). However, for non-comparison-based sorts, such as bin sort and radix sort, the asymptotic complexity is evaluated using the number of iterations based on the value of specific digits as against the whole element itself. Table 5.3 summarizes the asymptotes of sorting algorithms based on the best, average, and worst- case scenarios depending on their type of sort:

Table 5.3: Asymptotic complexities of various assorting algorithms

[ 136 ]

Page 148

﻿

Chapter 5

Now, let's analyze the complexity induced by the problem (of sorting) itself. The upper bound of the sorting problem is the asymptotic complexity of the fastest known algorithm, whereas the lower bound is the best possible efficiency that can be achieved using any sorting algorithm (also includes algorithms which are not invented yet). Once the lower and upper bounds meet using an algorithm, then we can safely assume that no other algorithm can beat this in terms of efficiency.

The best possible bounds of the current sorting algorithms for a given size of input vector are Ω(n) and O(nlog n) . This is because of the following reasons:

Every algorithm takes at least n iterations to read the input vector and write n elements to attain the output vector

Also, every element needs to be scanned before recognizing whether the input vector is sorted or not

To date, no one has ever devised an algorithm which can perform better than the O(nlog n) asymptote in both average and worst-case scenarios owing to the previously mentioned reasons. Thus, for a given worst-case scenario, we can comfortably presume that any sorting algorithm which requires Ω(nlog n) comparisons also requires Ω(nlog n) system runtime, which, in turn, shows that the problem of sorting also requires Ω(nlog n) system runtime. Hence, we can conclude that no comparison-based sorting algorithm with asymptotic complexity of θ(nlog n) can improve more than a constant factor.

Exercises

1. Write a bin sort and radix sort algorithm using linked lists. Compare their

runtime with algorithms implemented using lists.

2. Rewrite the original selection sort algorithm such that redundant swaps (of the same elements) are removed, and also compare its system runtime with the original algorithm.

[ 137 ]

Page 149

﻿

Sorting Algorithms

3. Out of the following, which algorithm preserves the original ordering of duplicate elements in the input vector? Can you suggest modifications which can prevent redundant swaps from occurring?

Insertion sort Bubble sort

Selection sort Shell sort

Merge sort Quick sort Heap sort Bin sort

Radix sort

4. Can you prove why comparison-based sorting algorithms require a minimum

asymptotic complexity of O(nlog n) for worst-case scenarios?

5. Compare the empirical performance of merge sort using vector-based and linked

list based implementation.

Summary

The current chapter builds the fundamental of sorting algorithm. The chapter introduced two kinds of sorting algorithm – comparison-based and non-comparison-based. The chapter introduced fundamentals of insertion sort, bubble sort, and selection sort, which are comparison-based algorithms and cover its implementation in R. The second half of the chapter focused on non-comparison-based sorting algorithms such as shell sort, merge sort, quick sort, heap sort, bin sort, and radix sort. The chapter also provided empirical comparison of various sorting algorithms.

[ 138 ]