Comparison between Merge sort Quicksort and Heapsort.

Fozilbek Kholboev  
 December 20, 2021

Introduction

In this paper, we will address the question of sorting. Let us understand the concept of sorting in the first place. What we mean by sorting? Sorting is rearranging the items such that their keys are ordered according to some well-defined ordering rule (usually numerical or alphabetical order) in the form of ascending keys or descending keys. As in our real life, there are several ways to accomplish one goal, the same idea applies to sorting as well. So, there are several algorithms which can sort the elements in both continuous-memory(array) and non-continuous-memory (linked-list). Specifically in this paper, we will see three comparison-based algorithms Merge sort, Quicksort and Heapsort, how they work, how fast they perform in specific situations, amount of resource they use to accomplish same goal of sorting. Furthermore, we try to compare these algorithms by considering their advantages and disadvantages on certain circumstances.

Methodology

We will investigate each algorithm one by one, try to consider them like following prototype

Classification: Comparison-based  
Time Complexity

* the worst-case scenario
* the average-case scenario
* the best-case scenario

Space Complexity (Memory Usage)

* in-place constant memory O(1)
* auxiliary extra memory usage (grows with input size)

Stability

* Is an attribute of sorting algorithm indicating that data with equal keys maintain their relative input order in the output (result)

Adaptive

* If the list is already sorted, algorithm should take minimum time by taking advantage of already sorted elements.

The algorithms mentioned above are implemented in C++. The data structure of choice is continuous memory (arrays). The results are generated for two categories: sorted array and pseudo-randomly generated values ranging [-len, len] where len is current size of the length of the array ranging 100, … 10000. The algorithms were tested on arrays of sizes from 100, 200, 300, . . . , 10000. In both categories, each algorithm was given the same input data. Each array size was tested 100 times. The result for each size is the average time it took for each algorithm to sort the input array.

Merge Sort

Merge Sort is top-bottom recursive algorithm. It uses divide and conquer method.

The main idea behind the scenes is, take the array and divide it possibly into two halves (left, right). Sort the left subpart of the array, Sort the right subpart of the array and merge those sorted subparts together to get final complete sorted array.   
This is recursive algorithm, it takes the whole array and continuously divides into possible 2 halves (left, right) until one element left in the array, which is sorted, then recursively goes back to previous executions to sort the right subpart in similar manner, finally merging those sorted subparts together.

Merge sort has the wort-case of O(n log(n)) which is way faster than O(n2) (bubble, selection, insertion sorts). The worst-case scenario happens when merging two subparts, it must do maximum number of comparisons. For example: Text, letter

Description automatically generated

And it has average-case time complexity order of O(n log n). As well as its best-case scenario is order of O(n log n), which means it is not adaptive sorting algorithm. In other words, it has no way to figure out whether the array is already sorted before recursively breaking into 2 halves and merging those halves together.

Most implementations give stable Merge sort (this is true in our implementation in this paper).

This algorithm is not in-place algorithm, which means we use O(n log n) extra auxiliary memory.

Quicksort

Quicksort is in-place and recursive comparison-based sorting algorithm. Quicksort, like Merge sort applies the divide-and-conquer paradigm. The main idea is: ->

Divide: Partition (rearrange) the array A[b … e] into two subarrays A[b … pIndex – 1] and A[pIndex+1 … e] such that A[b … pIndex -1] <= A[pIndex] <= A[pIndex+1 … e]. Compute the index of pIndex as part of the PARTITION procedure.

Conquer: Sort two subarrays A[b, pIndex-1] and A[pIndex+1 … e] by recursive calls to QUICKSORT  
Because 2 subarrays are already sorted, no need to combine them (like Merge in Merge sort) so the entire array A[b … e] is sorted.

PSEUDOCODE

QUICKSORT(A, begin, end){

If(begin >= end)  
 return  
 pIndex = PARTITION( A, begin, end)  
 QUICKSORT(A, begin, PIndex-1)  
 QUICKSORT(A, pIndex+1, end)  
}

We achieve complete sort by partitioning then recursively applying the method to subarrays.

The key to the whole algorithm is PARTITION procedure which rearranges (sorts) the A[begin … end]   
in-place.

PSEUDOCODE:  
PARTITION(A, left, right){

int pivot = A[right]  
int pIndex = left;  
for i=left to right-1{

if( A[i] <= pivot ) {

if(i != pIndex)  
exchange A[i] with A[pIndex]

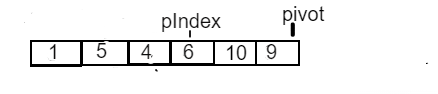
}  
}  
exchange A[pIndex] with A[right]  
return pIndex

}

In my method of implementation, I decided to choose the last element as pivot. (Although, this algorithm can be implemented by choosing the first element, mean and choosing the pivot randomly). And we also have pIndex which starts from left

PARTITION works by choosing the pivot and we loop through from A[left] to Aright-1]. If we find an element which is smaller than or equal to the pivot, we swap that element with the element under index pIndex.  
We have one condition i != pIndex since they start at same place initially and there is no reason to exchange same element with itself. When we exchange element, we increment the pIndex.

Diagram

Description automatically generated  
Of course, ,at the end we must exchange the element under index of pIndex with the pivot because logically PARTITION works that way by rearranging (sorting) the elements in place dividing into two halves, left A[b, pIndex-1] <= pIndex <= A[pIndex+1 … e]  


Then it returns the index of pIndex so that we know the element under this index is ion correct place and we have to apply the same thing for our subarrays A[b … pIndex-1] and A[pIndex+1 … e] recursively.

The running time of Quicksort depends on whether partitioning is balanced or unbalanced which in turn depends on which elements are used for partitioning and choice of the pivot.

If the PARTITION is balanced, algorithm runs asymptotically as fast as Merge Sort (O(n logn)).  
if the PARTITION is unbalanced, it can tun asymptotically as slowly as insertion sort (O(n2)).

It means Quicksort is adaptive comparison-based sorting algorithm such that its total running time depends not only number of input but also state of the input.

QUICKSORT(A, begin, end){

If(begin >= end)  
 return  
 pIndex = PARTITION( A, begin, end)  
 QUICKSORT(A, begin, PIndex-1)  
 QUICKSORT(A, pIndex+1, end)  
}

Let us suppose we have list of elements   
 1 ….. 15  
If we call the QUICKSORT() by passing 1 as beginning and 15 as the end. We assume always PARTITION happened in the middle of the array for all recursive QUICKSORT calls. Text

Description automatically generated with low confidence

So, PARTITION procedure is n time complexity because it goes through given entire list. At each stage (level) we have n time complexity, but how many stages (levels) are there? Well, we assumed PARTITON happens in the middle by dividing the array by 2 and goes recursively until we have 1 element left in a subarray.  
So if something is getting divided by 2 every time, how many times it needs to be divided such that it reaches 1.

A picture containing diagram

Description automatically generated

So there are log n levels which has n each n time complexity, finally giving us order of O(n log n) as the best-case running time complexity for Quicksort. So Quicksort is presumed to do its best when it partitions the array in the middle in each level of recursion, but is this possible in real life anyway?  
So we said that when partitioning is done in the middle of the array right? It means the element chosen as pivot must be median of the given array A[left … right] in each level of the recursion

A picture containing text, clock

Description automatically generated

But here goes the question, how do you know the median unless the list is sorted? Simply, we can’t do that. Randomly it may happen, but it is not under our control. So, achieving the best-case in Quicksort is impossible.

The worst-case happens in Quicksort

1. Array is already sorted in the same order
2. Array is already sorted in the reversed order
3. All elements in the array are same

Let us have a look on how and why Quicksort performs in its worst-case when dealing with already sorted array.  
As our method of choice to implement the algorithm, we choose the last element as our pivot, we get something like this:

Diagram, schematic

Description automatically generated

So, the time complexity of Quicksort algorithm in worst-case is:  
 [ N + ( N-1 ) + ( N-2 ) + ( N-3) + … + 2 ] = N(N+1)/2 – 1 = O( N2 )

However, we can improve the worst-case of sorting the already sorted array by choosing the median of the given in each level of the recursion, in which not only we solve the worst-case slowness, but we achieve the impossible best-case of Quicksort. Truly, this is the candidate to achieve best-case of the Quicksort.

Heap sort

Up until now, we have been utilizing linear data structures (array, vector … ) to accomplish our goal of sorting. But in this special method of sorting, we use hierarchal data structure (tree). Here is the deal, to fully comprehend the whole idea of Heap sort, we need to understand what the heap really is. Heap is complete binary tree. Furthermore, heap has two properties: max-heap, min-heap.   
So, max-heap suggests that each parent node is greater than its child nodes for all levels of tree (from root to leaf nodes).

As soon as we get input array, for example: {11, 2, 48, 13, 9, 21}

We can create such a complete binary tree (heap) like this one

Diagram

Description automatically generated with low confidenceA picture containing text, clock

Description automatically generated

On quick glance, we can say we can represent the heap by writing level by level (from root to leaves) from left to right. More “academic” definition for this is, we can find the left child by 2\*i + 1 and   
right child by 2\*i + 2 for given i as the position of the node for which the child nodes are being found of. For example, left child of root is 2 which under index 1. We give 0 which is position of root, 2\*0 + 1 gives us 1 which means our formula works correctly.

Great we get our heap ready, and we are ready to sort, well not there yet. We claimed that we use   
max-heap property of heap in which every parent node must be greater than child nodes.

We use HEAPIFY procedure to do that. Which essentially, goes through our heap and put the largest value of as root.

Diagram

Description automatically generatedA picture containing text, clock

Description automatically generated

Now, if we use our “homosapian” primate brain, we can come to conclusion that if we take the largest element and exchange it with last element and create max-heap on our affected heap excluding the last element and again taking largest and swapping it with current last element, applying the procedure until one unexchanged element left we get a brand-new sorted array.

Diagram

Description automatically generated

Diagram

Description automatically generated with medium confidenceDiagram

Description automatically generated

A picture containing clock, watch

Description automatically generated Diagram

Description automatically generated with medium confidence

It is obvious that, if we apply this until we left with one unexchanged element (which will be in its correct place anyway) we build sorted array from the right (biggest number) to left.

Text, letter

Description automatically generated

Heapsort is in-place sorting algorithm. It first takes input array and logically build max-heap on it. In max-heap the biggest number will be root node ( arr[0] ) so it places largest number to the end, so that our array logically divides into sorted and unsorted part. We take largest elements from unsorted part and put to sorted part accordingly.

Heapsort is not adaptive algorithm. Which means its best-case scenario is O(n logn), also its average and worst cases are order of O(n logn). However, Heapsort can do a little better according to input’s state, but it still is written as O (n log n).

In order to calculate time complexity of Heapsort, we have to calculate 3 parts: build max-heap, heapfy, and Heapsort itself. Although build method is not written as standalone function, it is there in Heapsort.

First HEAPIFY procedure, time we need to calculate is time taken to fix up relationship among elements A[i], A[left] and A[right], and time taken to run HEAPIFY on a subtree rooted at one of child nodes of i

To find the maximum of the parent and its child nodes is Θ(1) and calling the HEAPIFY for recursive levels of child node which has been exchanged is, if we have n number of nodes, then any subtree of that tree can have maximum of 2n/3 nodes (elements). So, we have this recurrence relationship



If we solve this recurrence, we will get T(n) = O( log n ). We know that if tree has n nodes, then its height will be h = log n. So, we can also say HEAPIFY is O (h).

Furthermore, we iterate through from n/2 – 1 to 1. Number of nodes at height h at most

Chart

Description automatically generated with medium confidence

Complexity will be

Diagram

Description automatically generated if we solve this, we get O(n).

Now we know that HEAPIFY takes O( log n ) and building max-heap takes O (n ) time.

Finally, time complexity of Heapsort = time of build-maxheap + time of HEAPIFY.

O(n) + (n – 1) [ O (log n ) ] -> giving us O (n log n) in worst case.

Results

Graphs of results for the average case of all sorting algorithms are presented in Figure 1. Quick sort took the least time to finish. Merge sort took the most time to sort out of 3 sorting algorithms. (I am presuming that we are taking time by dynamically allocating new arrays and then copying the elements from original array to newly created auxiliary arrays). In theory it has been claimed that Quick sort is faster than Merge sort in small arrays but as the size gets larger Merge sort takes the lead. I am not sure whether my machine took our max length as large array or not.

Chart, line chart

Description automatically generated

But what about for small number of input data? Same can be seen in small arrays also.

Chart, line chart

Description automatically generated

When it comes to sorting the already sorted data for 3 algorithms:

Chart, line chart

Description automatically generated

It is very different story. Merge sort took the lead, followed by Heap sort. Quick sort took several times longer than other 2 sorting algorithms.

As once has been mentioned above, Quicksort performs its worst-case when dealing with already sorted array. So, here this happens. But we must acknowledge that, for smaller sized input sorted arrays Heapsort is in fact faster than Merge sort. But as soon as size increases Merge sort takes the lead.

Conclusion

Assume you want to buy new clothes. You want to buy the best clothes out there. How do you define best clothes? Well, for which weather you want to buy? And price triggers the definition of the “best”. Also, style, fashion of the cloth will contribute to the definition of the “best”.

Same goes for the sorting algorithms as well. The definition of the best algorithm given by you and your circumstances, such as what do you really want from sorting algorithm apart from your list of elements gets sorted? You have no worries of memory resources and need speed when working with large scales, good take Merge sort. You have no worries of time and want memory efficiency, good take Quick sort or Heapsort. You know that most of time you have chance to get sorted array or partially sorted lists, so good avoid Quicksort and take Merge sort or Insertion sort.

From statisticians point of view in this papers experiments, Quick sort is the best of 3 when dealing with pseudo-random generated arrays in our all sizes, it is both memory efficient and considerably faster except array is already sorted.