"The Bottom Up and Many Pivot Study of Merge Sort, Quick Sort, and Heap Sort"

Mark A Avila

CSCI 323 - Analysis of Algorithms

Summer 2018

Abstract

The popular sorting algorithms that I will be talking about is Merge Sort, Quick Sort, and Heap Sort. For Merge and Quick Sort, I will not be talking about their relations with Divide and Conquer since it seems obvious on what the purpose of their use to both of these algorithms. In Merge and Heap sort I will talk about how bottom up is used and a combination of using multiple pivots along with Quick Sort which has a name called Multi pivot Quick Sort. The purpose of this project is not only to experiment with them but comparing on which of the 3 will run better with its own time complexities. My prediction for the efficient algorithm there is Merge Sort, with the time complexity of n log n since. Heap sort will have a time complexity with bottom up will be log n and Multipivot Quick Sort, would be n log n.

Review

Merge Sort: It uses a Divide and Conquer approach that divides an array into sub arrays. It has both sides containing values that needs to be sorted. Once both sides are sorted, we can merge them into one array. It uses an out-of-place which also uses an auxiliary storage that costs us the space complexity to be θ(n).

- Time Complexity
    - Best case: θ(n log n)
    - Average vase: θ(n log n)
    - Worst case: θ(n log n)

Quick Sort: It also uses a Divide and Conquer approach, but it involves in partition. It uses in-place sorting that starts with a pivot. Creating or finding a pivot requires partition. Once it partitions, it will divide a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

1. Pick an element, called a *pivot*, from the array.
2. *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

- Time Complexity:
    - Best Case: θ(n log n)
    - Average Case: θ(n log n)
        - We use recurrence relations to get the average case time complexity, but we use a pseudo average to see what we are trying to get.
        - We get T(n)= n+1 + $\frac{1}{n}\sum_{k=1}^{n}[T(k-1)+T(n-k)]$
        - n+1 is the partition, T(k-1) is the left side of the pivot , T(n-k) is the right side of the pivot.
        - Using Domain/Range Transformation with the follow up of telescoping we get the time complexity of T(n) = (n+1)*2(ln n)≈θ(n log n)
    - Worst Case: θ(n^2)
        - Using recurrence relations, we get T(n) = T(k-1) + T(n-k) +n+1

Heap Sort: involves preparing the list by first turning it into a max heap. The algorithm then repeatedly swaps the first value of the list with the last value, decreasing the range of values considered in the heap operation by one, and sifting the new first value into its position in the heap. This repeats until the range of considered values is one value in length.

1. Call the buildMaxHeap() function on the list. Also referred to as heapify(), this builds a heap from a list in O(n) operations.
2. Swap the first element of the list with the final element. Decrease the considered range of the list by one.
3. Call the siftDown() function on the list to sift the new first element to its appropriate index in the heap.
4. Go to step (2) unless the considered range of the list is one element.

- Time Complexity:
    - Best case: θ(n)
    - Average case: θ(n log n)
    - Worst Case: θ(n log n)

Algorithm Overview

Bottom Up Merge Sort

For the overall Merge sort algorithm, it was created by John von Neumann in 1945. For the bottom up implementation of Merge Sort it was created by von Neumann and Herman Goldstine in 1948. The idea of bottom up merge sort is Even though we are thinking in terms of merging together two large subarrays, the fact is that most merges are merging together tiny subarrays. Another way to implement mergesort is to organize the number of merges so that we do all the merges of tiny arrays on one pass, then do a second pass to merge those arrays in pairs, and so forth, continuing until we do a merge that encompasses the whole array. This method requires even less code than the standard

recursive implementation. We can begin by doing a pass of 1-by-1 merges (considering individual items as subarrays of size 1), then a pass of 2-by-2 merges (merge subarrays of size 2 to make subarrays of size 4), then 4-by-4 merges, and so forth. The time complexity for this algorithm runs at the same θ(n log n), but for the space complexity, it will be less than linear since it used less or close to no memory.

Psuedo-code

```
BottomUpMergeSort(a[], b[], n){

	for (width = 1; width < n; width = 2 * width){
		for (i = 0; i < n; i = i + 2 * width){
			BottomUpMerge(a, i, min(i+width, n), min(i+2*width, n), b);
		}
		CopyArray(b, a, n);
	}
}

BottomUpMerge(a[], iLeft, iRight, iEnd, b[]){
	i = iLeft, j = iRight;
	for (k = iLeft; k < iEnd; k++) {
		if (i < iRight && (j >= iEnd || a[i] <= a[j])) {
			b[k] = a[i];
			i = i + 1;
		} else {
			b[k] = a[j];
			j = j + 1;
		}
	}
}

CopyArray(b[], a[], n){
	for(i = 0; i < n; i++)
		a[i] = b[i];
}
```

Bottom Up Heap Sort

In general, Heap Sort was created by by J. W. J. Williams in 1964. This was also the birth of the heap, presented already by Williams as a useful data structure in its own right. In the same year, R. W. Floyd published an improved version that could sort an array in-place, continuing his earlier research into the treesort algorithm. The change makes little difference to the linear-time heap-building phase, but is significant in the second phase. Bottom-up heapsort instead finds the path of largest children to the leaf level of the tree using only one comparison per level. It finds a leaf which has the property that it and all of its ancestors are greater than or equal to their siblings. From the leaf, it searches upward (comparison based level) for the correct position in that path in order to insert it at the end of the array. This is the same location as ordinary heapsort finds, and requires the same number of exchanges to perform the insert. The time complexity of the algorithm would be for average case θ(1) and worst case would be θ(n).

Pseudo-code

```
function leafSearch(a, i, end) is
    j ← i
    while iRightChild(j) ≤ end do
        (Determine which of j's two children is the greater)
        if a[iRightChild(j)] > a[iLeftChild(j)] then
            j ← iRightChild(j)
        else
            j ← iLeftChild(j)
    (At the last level, there might be only one child)
    if iLeftChild(j) ≤ end then
        j ← iLeftChild(j)
    return j

procedure siftDown(a, i, end) is
    j ← leafSearch(a, i, end)
    while a[i] > a[j] do
        j ← iParent(j)
    x ← a[j]
    a[j] ← a[i]
    while j > i do
        swap x, a[iParent(j)]
        j ← iParent(j)
```

Quick Radix Sort

Quick Sort in general was created by Tony Hoare in 1959, but published his work in 1961. The idea or implementation of multi pivot quick sort is to take two pivots, one in the left end of the array and the second, in the right end of the array. The left pivot must be less than or equal to the right pivot, so we swap them if necessary. We begin partitioning the array into three parts: first part, all elements will be less than the left pivot, second part all elements will be greater or equal to the left pivot and also will be less than or equal to the right pivot, and third part all elements will be greater than the right pivot. Then, we shift the two pivots to their appropriate positions as we see in the below bar, and after that we begin quick sorting these three parts recursively, using this method. Multi pivot quick sort is a little bit faster than the original single pivot quicksort. The worst case will remain $O(n^2)$ when the array is already sorted in an increasing or decreasing order.

Pseudo-code

```
swap(int* a, int* b){
	int temp = *a;
	*a = *b;
	*b = temp;
}

MultiPivotQuickSort(int* arr, int low, int high){
	if (low < high) {
		int lp, rp;
		rp = partition(arr, low, high, &lp);
		DualPivotQuickSort(arr, low, lp - 1);
		DualPivotQuickSort(arr, lp + 1, rp - 1);
		DualPivotQuickSort(arr, rp + 1, high);
	}
}

partition(int* arr, int low, int high, int* lp){
	if (arr[low] > arr[high])
		swap(&arr[low], &arr[high]);
	int j = low + 1;
	int g = high - 1, k = low + 1, p = arr[low], q = arr[high];
	while (k <= g) {
		if (arr[k] < p) {
			swap(&arr[k], &arr[j]);
			j++;
		}
		else if (arr[k] >= q) {
			while (arr[g] > q && k < g)
				g--;
			swap(&arr[k], &arr[g]);
			g--;
			if (arr[k] < p) {
				swap(&arr[k], &arr[j]);
				j++;
			}
		}
		k++;
	}
	j--;
	g++;
	swap(&arr[low], &arr[j]);
	swap(&arr[high], &arr[g]);
	*lp = j;

	return g;
}
```

Results

Bottom Up Merge Sort

n	Iter 1	Iter 2	Iter 3	Iter 4	Iter 5	Iter 6	Iter 7	Iter 8	Iter 9	Iter 10	Avg.
10	8	6	7	6	5	4	8	7	5	6	6.2

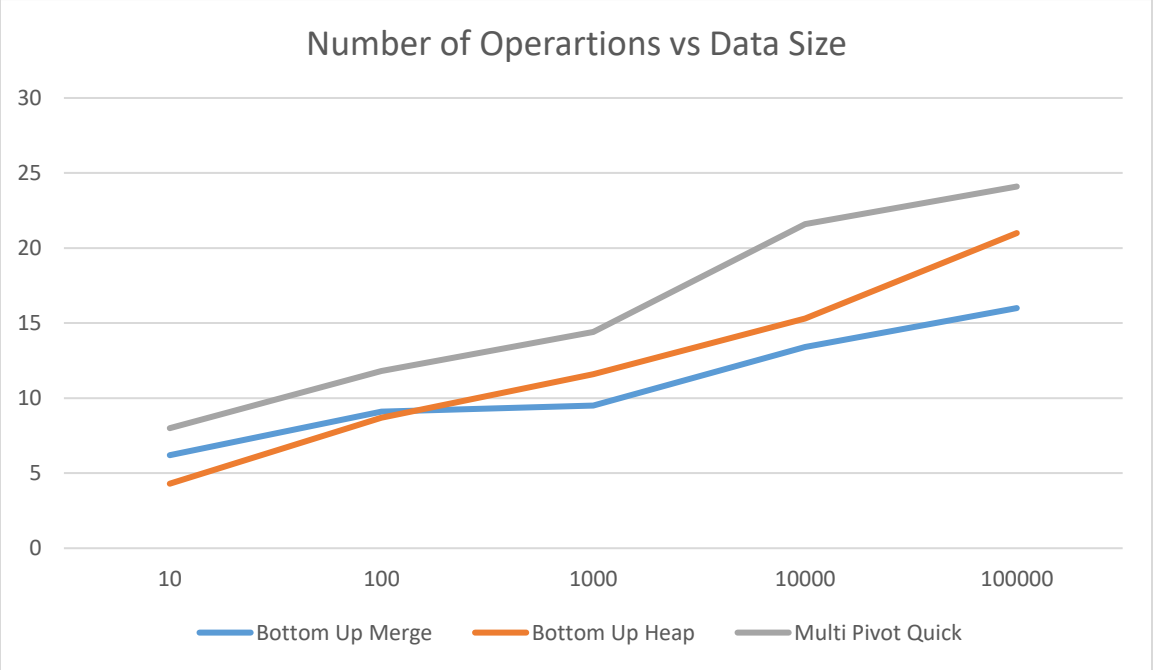
100	7	9	8	10	12	9	8	7	11	10	9.1
1000	12	13	10	9	7	8	10	9	8	9	9.5
10000	18	16	14	15	10	12	11	13	15	10	13.4
100000	20	16	19	15	17	18	16	19	20	18	16

Bottom Up Heap Sort

n	Iter 1	Iter 2	Iter 3	Iter 4	Iter 5	Iter 6	Iter 7	Iter 8	Iter 9	Iter 10	Avg.
10	2	4	5	3	7	6	4	5	2	5	4.3
100	8	9	6	8	11	9	8	7	11	10	8.7
1000	12	14	15	13	9	8	10	12	13	10	11.6
10000	16	14	15	13	10	11	14	18	22	20	15.3
100000	22	24	25	19	20	16	18	19	22	25	21

Multi Pivot Quick Sort

n	Iter 1	Iter 2	Iter 3	Iter 4	Iter 5	Iter 6	Iter 7	Iter 8	Iter 9	Iter 10	Avg.
10	8	6	7	6	5	9	8	10	12	9	8
100	12	11	15	9	8	10	13	15	11	14	11.8
1000	14	13	10	11	12	15	16	19	18	16	14.4
10000	20	22	19	18	23	24	21	23	25	21	21.6
100000	20	26	30	25	27	28	20	19	20	26	24.1



Conclusion

From this “lab” experiment, I realized that the time complexity remains somewhat similar to the general sorting algorithms for merge quick, and heap sort. Multi pivot basically means a shortcut of

finding pivot for both left and right side of the array, bottom up you start at a small value, building up to a greater and large value. If I were to continue studying these topics I would ask the following:

1. Would all 3 of these variations work for Big Data Algorithms
2. If we were to use Parallel Algorithm to have more than n processors working in a shorter amount of time?
3. Can the space complexity be better, without using auxiliary storage?

Reference

https://www.geeksforgeeks.org/dual-pivot-quicksort/
http://www.codebytes.in/2014/10/bottom-up-merge-sort-java-implementation.html
https://stackoverflow.com/questions/680541/quick-sort-vs-merge-sort
https://en.wikipedia.org/wiki/Merge_sort
https://en.wikipedia.org/wiki/Heap_sort
https://en.wikipedia.org/wiki/Quick_sort