Observing trends in running time of various sorting algorithms

Anurag Sudhakar Makade (IIT2018121)

IV Semester, B.Tech. in Information Technology,

Indian Institute of Information Technology, Allahabad

Abstract: In this paper, I have tried to present the trends observed in running time of a few sorting algorithms as a function of the input size.

1. INTRODUCTION

In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists.

2. SORTING ALGORITHMS

2.1 Insertion Sort

It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list.

Algorithm:

```
INSERTION-SORT(A)
1 for j = 2 to A.length
2   key = A[j]
3   // Insert A[j]  into the sorted
sequence A[1..j-1].
4   i = j - 1
5   while i > 0 and A[i] > key
6     A[i + 1] = A[i]
7     i = i - 1
8   A[i + 1] = key
```

Time complexity calculation:

Best Case:

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., $O(n)$). During each iteration of the outer for loop, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array and the inner while loop exits on the first check itself.

Worst Case:

The worst case input is an array sorted in reverse order. In these cases every iteration of the inner while loop will scan and shift the entire sorted subsection of the array before inserting the next element. To insert the last element, we need at most $n-1$ comparisons and at most $n-1$ swaps. To insert the second to last element, we need at most $n-2$ comparisons and at most $n-2$ swaps, and so on. Hence, $(n-1)+(n-2)+...+1 = \frac{n(n-1)}{2}$ number of operations would be required. This gives insertion sort a quadratic running time (i.e., $O(n^2)$).

Observed Trends:

2.2 Quicksort

Quicksort is a divide and conquer algorithm which relies on a partition operation: to partition an array, an element called a pivot is selected. All elements smaller than the pivot are

moved before it and all greater elements are moved after it. The lesser and greater sublists are then recursively sorted.

Algorithm:

```
QUICKSORT(A, p, r)
1 if p < r
2   q = PARTITION(A, p, r)
3   QUICKSORT(A, p, q - 1)
4   QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1 x = A[r]
2 i = p - 1
3 for j = p to r - 1
4   if A[j] ≤ x
5     i = i + 1
6     swap A[i] with A[j]
7 swap A[i + 1] with A[r]
8 return i + 1
```

Time Complexity Calculation:

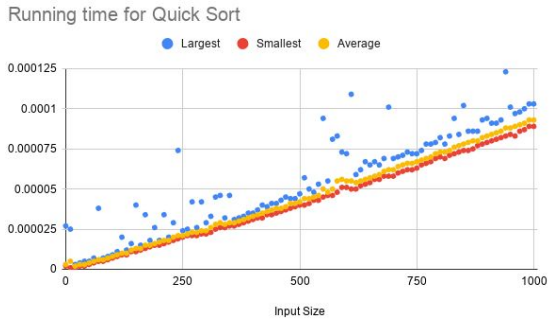
Best Case:

In the most even possible split, PARTITION produces two subproblems, each of size no more than $n = 2$, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$. The recurrence for the running time is $T(n) = 2T(n/2) + \Theta(n)$. By the master theorem, this has the solution $T(n) = \Theta(n \lg n)$ with time complexity $O(n \lg n)$.

Worst Case:

If this happens repeatedly in every partition, then each recursive call processes a list of size one less than the previous list. Consequently, we can make $n - 1$ nested calls before we reach a list of size 1. Assuming the same partitioning happens in each recursive call, we can make $n - 1$ nested calls before we reach a list of size 1. The ith call does $O(n - i)$ work to do the partition, $\sum_{i=0}^{n}(n - i) = O(n^2)$, so in that case Quicksort takes $O(n^2)$ time.

Observed Trends:



2.3 Merge Sort

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list.

Algorithm:

```
MERGE-SORT(A, p, r)
1 if p < r
2   q = (p + r)/2
3   MERGE-SORT(A, p, q)
4   MERGE-SORT(A, q + 1, r)
5 MERGE(A, p, q, r)
```

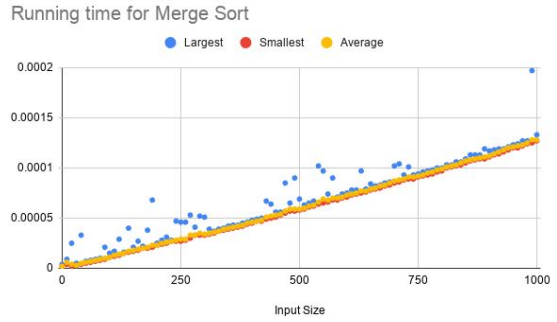
```
MERGE(A, p, q, r)
1 n1 = q - p + 1
2 n2 = r - q
3 let L[1 .. n1 + 1] and R[1 .. n2 + 1]
be new arrays
4 for i = 1 to n1
5   L[i] = A[p + i - 1]
6 for j = 1 to n2
7   R[j]  D A[q + j]
8 L[n1 + 1] = ∞
9 R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13   if L[i] ≤ R[j]
14     A[k] = L[i]
15     i = i + 1
16   else A[k] = R[j]
17     j = j + 1
```

Time Complexity Calculation:

If the running time of merge sort for a list of length n is $T(n)$, then the recurrence relation for merge sort comes out to be $T(n) = T(n/2) + \Theta(n)$, which, by the Master theorem, has the solution $T(n) = \Theta(n \lg n)$.

Best case time complexity: $O(n \lg n)$
Worst case time complexity: $O(n \lg n)$

Observed Trends:



2.4 Heapsort

It works by determining the largest (or smallest) element of the list, placing it at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a heap.

Algorithm:

```
HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 for i = A.length downto 2
3   swap A[1] with A[i]
4   A.heap-size = A:heap-size - 1
5   MAX-HEAPIFY(A, 1)
```

```
BUILD-MAX-HEAP(A)
1 A.heap-size = A.length
2 for i = A.length/2 downto 1
3   MAX-HEAPIFY(A, i)
```

```
MAX-HEAPIFY(A, i)
1 l = LEFT(i)
2 r = RIGHT(i)
3 if l ≤ A.heap-size and A[l] > A[i]
4   largest = l
5 else largest = i
6 if r ≤ A.heap-size and A[r] >
A[largest]
7   largest = r
8 if largest ≠  i
9   swap A[i] with A[largest]
10  MAX-HEAPIFY(A, largest)
```

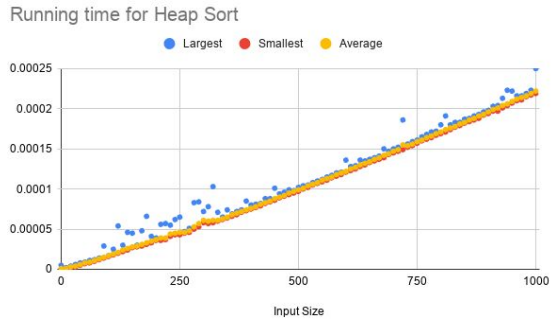
Time Complexity Calculation:

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the $\Theta(1)$ time to fix up the relationships among the elements A[i], A[LEFT(i)], and A[RIGHT(i)], plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children's subtrees each have size at most $2n/3$—the worst case occurs when the bottom level of the tree is exactly half full—and therefore we can describe the running time of MAX-HEAPIFY by the recurrence $T(n) \leq T(2n/3) + \Theta(1)$. The solution to this recurrence, by the master theorem is $T(n) = O(\lg n)$.

The running time for BUILD-MAX-HEAP is bounded by $O(n)$.

Thus, the HEAPSORT procedure takes time $O(n \lg n)$, since the call to BUILD-MAX-HEAP takes time O(n) and each of the $n - 1$ calls to MAX-HEAPIFY takes time $O(\lg n)$.

Observed Trends:



2.5 Sorting by insertion into a hash table

This algorithm works by implementing a hash table with all the possible input elements as the keys and the number of occurrences of the corresponding element as the value.

Algorithm:

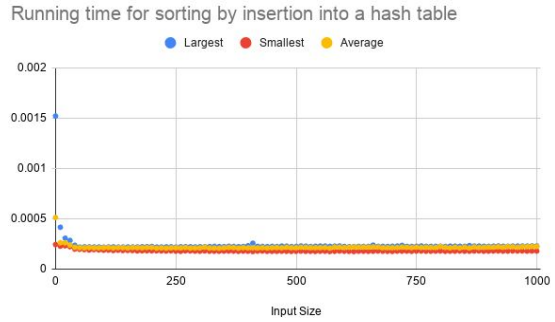
```
HASH-TABLE-SORT(A)
1 for i = 1 to A.length
2   HashTable[A[i]]++
```

```
3 j = 1
4 for key in HashTable
5   while HashTable[key] > 0
6     A[j] = key
7     HashTable[key]--
8     j++
```

Time Complexity Calculation:

We need to traverse the list of elements once in order to perform the hashing. This takes linear running time (i.e. $(O(n))$. We also need to traverse the hash table once in order to form the sorted array. This requires time $O(k)$, k being the number of keys in the hash table. Hence, the best case, worst case and average case time complexity for this algorithm would be $O(n+k)$.

Observed Trends:



3. REFERENCES

[1] Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

[2]https://en.wikipedia.org/wiki/Sorting_algorithm.