Sorting Algorithms Visualization and Analysis

Ishwak Sharda, Kshitij Goyal, Joshua Lepon

September 21, 2024

Course: COMP 359
Instructor: Russell Campbell
Date: September 20, 2024

Bubble Sort

Pseudocode

Algorithm 1 Bubble Sort

```
Input: An array $A$ of length $n$
for $i = 0$ to $n - 1$ do
    for $j = 0$ to $n - i - 2$ do
        if $A[j] > A[j + 1]$ then
            Swap $A[j]$ and $A[j + 1]$
```

Mathematical Analysis

The total number of comparisons in Bubble Sort is:

$$T(n) = \sum_{i=0}^{n-1} (n - i - 1)$$

Simplifying the summation:

$$T(n) = \sum_{i=0}^{n-1} (n - 1 - i) = \sum_{i=0}^{n-1} (n - 1) - \sum_{i=0}^{n-1} i$$

$$\sum_{i=0}^{n-1} (n - 1) = n \cdot (n - 1)$$

$$\sum_{i=0}^{n-1} i = \frac{(n - 1) \cdot n}{2}$$

$$T(n) = n \cdot (n - 1) - \frac{(n - 1) \cdot n}{2} = \frac{n(n - 1)}{2}$$

Therefore, the time complexity is:

$$T(n) = O(n^2)$$

In the best case, when the array is already sorted, the time complexity is:

$$T_{\text{best}}(n) = O(n)$$

Insertion Sort

Pseudocode

Algorithm 2 Insertion Sort

```
Input: An array $A$ of length $n$
for $i = 1$ to $n - 1$ do
    $key = A[i]$
    $j = i - 1$
    while $j \geq 0$ and $A[j] > key$ do
        $A[j + 1] = A[j]$
        $j = j - 1$
    $A[j + 1] = key$
```

Mathematical Analysis

In the worst case (if the array is sorted in reverse order), the total number of comparisons in Insertion Sort is:

$$T(n) = \sum_{i=1}^{n-1} i = \frac{(n-1) \cdot n}{2}$$

Thus, the worst-case time complexity is:

$$T(n) = O(n^2)$$

In the best case (when the array is already sorted), the total number of comparisons is:

$$T_{\text{best}}(n) = O(n)$$

Merge Sort

Pseudocode

Algorithm 3 Merge Sort

```
Input: An array $A$ of length $n$
if $n > 1$ then
    $mid \leftarrow n/2$
    $left \leftarrow A[0 \ldots mid - 1]$
    $right \leftarrow A[mid \ldots n - 1]$
    MergeSort($left$)
    MergeSort($right$)
    Merge($left$, $right$, $A$)
```

Algorithm 4 Merge

```
Input:Input: Arrays $B[0..p1]$ and $C[0..q1]$ both sorted
$i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$
while $i < p$ and $j < q$ do
    if $B[i] \leq C[j]$ then
        $A[k] \leftarrow B[i]$
        $i \leftarrow i + 1$
    else
        $A[k] \leftarrow C[j]$
        $j \leftarrow j + 1$
    $k \leftarrow k + 1$
if $i = p$ then
    Copy $C[j \ldots q - 1]$ to $A[k \ldots p + q - 1]$
else
    Copy $B[i \ldots p - 1]$ to $A[k \ldots p + q - 1]$
```

Mathematical Analysis

Let $T(n)$ be the time complexity of Merge Sort for an input array of size n. The recurrence relation is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Step-by-Step Expansion

Expand the recurrence relation to show how the time complexity unfolds at each level:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4T\left(\frac{n}{4}\right) + 2n$$

$$T(n) = 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n = 8T\left(\frac{n}{8}\right) + 3n$$

General Case

In general, after k expansions, we have:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

Base Case

The recursion terminates when $n = 1$, i.e., when $\frac{n}{2^k} = 1$. Solving for k gives:

$$n = 2^k \implies k = \log_2 n$$

Substitute this value of k into the general equation:

$$T(n) = 2^{\log_2 n} T(1) + n \log_2 n$$

Since $T(1) = O(1)$:

$$T(n) = n \cdot O(1) + n \log_2 n$$

Thus, the total time complexity of Merge Sort is:

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

Quick Sort

Pseudocode

Algorithm 5 Quick Sort

```
Input: An array $A$ of length $n$
if $n > 1$ then
    $pivot \leftarrow A$[some index]
    $left \leftarrow$ elements less than $pivot$
    $right \leftarrow$ elements greater than $pivot$
    QuickSort($left$)
    QuickSort($right$)
    Merge($left$, $pivot$, $right$, $A$)
```

Mathematical Analysis

Let $T(n)$ be the time complexity of Quick Sort for an input array of size n. Quick Sort splits the array into two parts, one with elements smaller than the pivot and the other with elements larger than the pivot. The recurrence relation is:

$$T(n) = T(k) + T(n-k-1) + O(n)$$

where k is the number of elements less than the pivot, and $n-k-1$ is the number of elements greater than the pivot. The $O(n)$ term comes from the partitioning step, which takes linear time.

Worst-Case Time Complexity

In the worst case, the partition is highly unbalanced, with $k = 0$ or $k = n-1$. In this case, the recurrence becomes:

$$T(n) = T(n-1) + O(n)$$

Expand this recurrence step by step:

$$T(n) = T(n-1) + n$$

$$T(n) = T(n-2) + (n-1) + n$$

$$T(n) = T(n-3) + (n-2) + (n-1) + n$$

In general, after k expansions, we have:

$$T(n) = T(n-k) + \sum_{i=0}^{k-1}(n-i)$$

In the worst case, this recurrence continues until $T(1)$, so we sum from 1 to n:

$$T(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Thus, the worst-case time complexity is:

$$T(n) = O(n^2)$$

Best-Case Time Complexity

In the best case, the pivot splits the array into two equal parts. The recurrence becomes:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Using the master theorem or expanding the recurrence as done for merge sort:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4T\left(\frac{n}{4}\right) + 2n$$

$$T(n) = 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n = 8T\left(\frac{n}{8}\right) + 3n$$

In general, after k expansions, we have:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

The recursion stops when $\frac{n}{2^k} = 1$, i.e., $k = \log_2 n$. Thus:

$$T(n) = n \cdot O(1) + n \log_2 n$$

So, the best-case time complexity is:

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

Bogo Sort

Pseudocode

Algorithm 6 Bogo Sort

```
Input: An array $A$ of length $n$
while A is not sorted do
    Randomly shuffle the array $A$
```

Mathematical Analysis

Bogo Sort works by repeatedly shuffling the array until it is sorted. The time complexity depends on the probability of randomly generating a sorted permutation.

Worst-Case Time Complexity

There are $n!$ possible permutations of an array of size n. In the worst case, Bogo Sort must generate every possible permutation before finding the sorted one. This gives us the expected number of permutations as:

$$E(T(n)) = n!$$

The time complexity in the worst case is, therefore:

$$T(n) = O(n * n!)$$

Heap Sort

Pseudocode

Algorithm 7 Heap Sort

```
Input: An array $A$ of length $n$
Build a max heap from the input array $A$
for $i = n - 1$ to 1 do
    Swap $A[0]$ (max element) with $A[i]$
    Reduce the heap size by 1
    Heapify the root $A[0]$ to maintain the heap property
```

Mathematical Analysis

Heap Sort consists of two major phases: building the max heap and extracting elements from the heap while maintaining the heap structure.

Building the Max Heap

Building a max heap takes $O(n)$ time. This is done by calling the 'heapify' function from the middle of the array towards the root (the last non-leaf node up to the root).

$$T_{\text{build}}(n) = O(n)$$

Heapify Operation

Each call to 'heapify' takes $O(\log n)$ time since the height of the heap is $\log n$, and the heap property needs to be restored by percolating down the root element. During the sorting phase, we perform 'heapify' for each element of the array, resulting in $n - 1$ 'heapify' operations. Each heapify takes $O(\log n)$ time.

$$T_{\text{heapify}}(n) = O(\log n)$$

Sorting Phase

In the sorting phase, we extract the maximum element n times. After each extraction, the size of the heap is reduced by one, and we perform a 'heapify' on the reduced heap. The total time complexity of this phase is:

$$T_{\text{sort}}(n) = \sum_{i=1}^{n-1} O(\log i) = O(n \log n)$$

Total Time Complexity

The overall time complexity of Heap Sort is the sum of the time to build the heap and the time to perform the heapify operations:

$$T(n) = O(n) + O(n \log n) = O(n \log n)$$

Thus, the time complexity of Heap Sort is:

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

Space Complexity

Heap Sort is an in-place sorting algorithm. Thus, the space complexity is:

$$S(n) = O(1)$$

Shell Sort

Pseudocode

Algorithm 8 Shell Sort

```
Input: An array $A$ of length $n$
$gap = n/2$
while $gap > 0$ do
    for from initial gap index to $n - 1$ do
        Get and store element at gap
        Get element at left end of gap
        if element at the left of gap is greater than element at the gap then
            Swap element at gap with element at the left of the gap
    $gap = gap/2$ — halve the gap size
```

Mathematical Analysis

Shell Sort is an optimization of insertion sort that swaps 2 elements at 2 ends of an array with a gap between them; this gap usually starts at half the array size, and the gap gradually halves until it reaches 1 and then it becomes a regular insertion sort until the array is sorted. It compares the elements at the right end of the gap with the element at the left end of the gap and if the element at the right end of the gap is smaller than the element at the left end of the gap, a swap occurs. This will continue until the end of the array is reached which triggers a halving of the gap.

0.0.1 Total Time Complexity

Determining the Time Complexity of the Shell Sort is pretty challenging as the time complexity of a shell sort is heavily dependent on the choice of the gap sequence as well as how sorted the array already is.

0.0.2 Best Case Time Complexity

The best case time complexity of Shell Sort is:

$$O(nlog^2n)$$

This is possible because when the array is nearly sorted, similar to an insertion sort, the algorithm takes on a linear time complexity of $O(n)$ as it will only need to pass through the array once to sort the array; and when using optimized gap sequences like the Pratt Sequence which uses powers of 2 and 3s as gaps:

$$2^p3^q = 1, 2, 3, 4, 6, 8, 9, 12, 16...$$

the total number of passes required will be $O(nlogn)$ since the gaps decrease exponentially in size. Mathematically, the total time cost can be written as the following:

$$T(n) = \sum_{i=1}^{logn} O(n * logn)$$

This results in the following sequence:

$$T(n) = O(nlogn) + O(nlogn) + ... + O(nlogn)(logntimes)$$

Thus the best case time complexity becomes:

$$T(n) = O(nlogn * logn) = O(nlog^2 n)$$

0.0.3 Worst Case Time Complexity

Shell sort has a worst case time complexity of:

$$O(n^2)$$

When the array is completely unordered and when using Shell's original gap sequence of:

$$n/2, n/4, ..., 1$$

This is because the large initial gaps do not reduce the number of comparisons, and when the gaps decrease, it basically becomes a regular insertion sort.

Space Complexity

Shell Sort is an in-place sorting algorithm. Thus, the space complexity is:

$$S(n) = O(1)$$

References (MLA Style)

- “Running Parallel Sorting Algorithm With Processing Java.” ChatGPT, https://chatgpt.com/share/66ecc68d-e884-800c-aeb9-b31a1d52f0b3.
- “Selection Between PG Graphics and Instance Based Rendering.” ChatGPT, https://chatgpt.com/share/66ecc6e0-5c70-800c-9fdb-bc43811f16f7.
- Wikipedia contributors. “Heapsort.” Wikipedia, 19 Sept. 2024, en.wikipedia.org/wiki/Heapsort.
- Radix Sort Algorithm. www.tutorialspoint.com/data$_{s}tructures_{a}lgorithms/radix_{s}ort_{a}lgorithm.htm.$
- Sedgewick, Robert, and Kevin Wayne. “Merge Sort.” Algorithms - FOURTH EDITION, vol. 1, Addison-Wesley.
- Shell, D. L. ”A High-Speed Sorting Procedure” *Communications of the ACM*, 1959
- Pratt, Vaughan Ronald (1979). ”Shellsort and Sorting Networks (Outstanding Dissertations in the Computer Sciences)”. *Garland*, 1979
- Shaffer, Clifford A. Data Structures and Algorithm Analysis (Java Version), 3.2 ed., Department of Computer Science, Virginia Tech, 28 Mar. 2013, pp. 2.3-2.4, 3.1-3.1, 4.1, 5.1, 5.2.