Analysis of Internal Sorting Algorithms

==========

Bubble sort

Bubble sort works by moving larger elements to the end of an array A by comparing i and $i+1$ elements, and swapping if $A[i] > A[i+1]$. Bubble sort is stable sort as the relative position of duplicate elements is preserved. Because it does not require extra memory space, it is an in-place sorting algorithm.

Running time = $O(N^2)$

Selection sort

Selection sort is an improvement on bubble sort because elements are moved more than 1 slot per step. Selection sort works by scanning the array, selecting the smallest element and swapping with $A[0]$. You repeat the process by scanning the remaining array, selecting the smallest and performing the swapping until last element is reached.

Selection sort is not stable as the relative position of duplicate elements might change. It is an in-place sorting algorithm that only needs $O(1)$ extra space to store "min_index" temporary variable.

Running time = $O(N^2)$

Insertion sort

Insertion sort works by inserting ith element on sorted $i-1$ elements, for $0 < i < N$. Thus there are $N-1$ passes in total. Insertion sort is in-place and stable sorting algorithm. The implementation consists of nested loops. Outer loop takes $N-1$ iterations. The inner loop takes at max i iterations for the ith pass. Since only one assignment is performed, a shift operation requires approximately a third of the processing work of an exchange operation (like in bubble sort).

Worst-case: The list is in reverse order

In such case, the inner loop takes i iterations for the ith pass. Thus, the total number of iterations is

$$1 + 2 + 3 + ... + N - 1 = N(N-1)/2 = \Theta(N^2)$$

Best-case: The list is already sorted

If the input array is sorted, then the inner loop fails immediately. Thus,the total number of iterations is

$$\Theta(N)$$

Hence, the complexity of insertion sort lies between $\Theta(N)$ and $\Theta(N^2)$, depending on the input.

Average-case:

Inversions are any ordered pair (i, j) in the input array A such that $i < j$ but $A[i] > A[j]$. Each swap removes only one inversion. Thus, the complexity of any sorting algorithm, that performs swapping (explicitly like bubble sort or implicitly like insertion sort) of adjacent elements, is proportional to the number of inversions on the input. Under the assumptions that there are no duplicates, $N > 1$ and any permutation of N distinct elements is equally likely, we can compute average number of inversions in an array of N distinct elements.

Let L be the list of elements and L_r its reverse order list. Then, any element pair (x, y) such that $y > x$, is an inversion in exactly one of L or L_r. Number of ways you can choose 2 distinct elements from N is given as,

$$C_2^N = \frac{N!}{(N-2)! * 2!} = \frac{N(N-1)}{2}$$

Total number of inversion in L and L_r = $N(N-1)/2$

Average number of inversions = $N(N-1)/4$

Any algorithm (not only insertion sort) that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average. This is the case because on average there are $N(N-1)/4 = \Omega(N^2)$ inversions.

Sum up the running time:

Worst case = $\Theta(N^2)$
Best case = $\Theta(N)$
Average case for random array = $\Theta(N^2)$
"Almost sorted" case: $\Theta(N)$

If speaking for all cases in general, it runs in $O(N^2)$

Shell sort

Shell sort improves on insertion sort using the fact that insertion sort runs faster on "somewhat" sorted array. It performs several insertion sorts on different sub-sequences of elements. Shell sort uses a sequence, $h_1, h_2, h_3, ..., h_t$, called increment sequence, such that $h_1 = 1$. Shell sort is an in-place and not-stable sorting algorithm.

Shell sort begins with a suitable h_k increment and performs insertion sort on the elements spaced h_k, such that, for every i we have $A[i] < A[i + h_k]$. The array is then said to be h_k-sorted. The step is repeated for the next increment $h_k - 1$ until h_1. Because the comparison distance between elements decreases until the last phase, shell short is sometimes referred to as "diminishing increment sort".

An important property to remember is that an h_k sorted array that is then $h_k - 1$ sorted remains h_k sorted. That is, work done on early phase is not undone by later phase.

Increment Sequences:

Shell sequence: $N/2, N/4, ..., 1$ (repeatedly divide by 2)
Hibbard sequence: $1, 3, 7, ..., 2^k - 1$
Knuth sequence: $1, 4, 13, ..., (3^k - 1)/2$
Sedgewick sequence: $1, 5, 19, 41, 109, ...$

The running time of Shell sort depends on the choice of increment sequence. The average-case analysis is still an open problem, except for the most trivial sequence. The worst-case complexity for Shell sequence is discussed below:

Worst-case running time of Shell sort using shell sequences is $\Theta(N^2)$

For such tight bound, we need to show the following:

- Upper bound is $O(N^2)$
- For some input it actually takes $\Omega(N^2)$

Consider a case where N is a power of 2 such that all increments are even except 1. Let's place $N/2$ largest numbers in the odd positions and $N/2$ smallest numbers in the even positions in sorted order.

```
Example:
Start           1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16
After 8-sort    1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16
After 4-sort    1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16
After 2-sort    1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16
After 1-sort    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

As shown above, none of the insertion sort, except h_1, changes the list. It is easy to see that for any Shell sequence, you can always come up with an input for which shell sort becomes merely an insertion sort.

All elements in even positions i, moves $i-1$ positions during h_1-sort. Odd positioned elements do not have to move. Thus, to place $N/2$ elements in correct place requires

```
1 + 2 + 3 + 4 + 5 + 6 + 7 = 28 operations.
```

For general N,

$$\sum_{i=1}^{N/2} i - 1 = \Omega(N^2)$$

Now, we show the upper bound. At each h_k sort, about N/h_k elements are insertion sorted. Since insertion sort is worst-case $O(N^2)$, total cost of each pass is

$O((N/h_k)^2) = O(N^2/h_k)$.

Total cost over all passes is given by,

$$O(\sum_{i=1}^{t} N^2/h_i) = O(N^2 * \sum_{i=1}^{t} 1/h_i)$$

Since, $\sum_{i=1}^{t} 1/h_i$ is a geometric series with first element 1, $\sum_{i=1}^{t} 1/h_i < 2$

Hence, total cost over all passes = $O(N^2)$. And the proof is complete!

There are couple of optimization that can be performed on this sequence:

- It is easy to observe that since increments are not relatively prime, chances are same elements will be traced (e.g: 8, 4, 2 increments)
- Smaller is the comparison distance, lesser effect does it have on the list. We want to maintain larger distance comparison.

Hence Hibbard, Knuth, Sedgewick and other sequences were developed to increase the performance of Shell sort.

Heap sort

Heapsort uses binary heap to sort the elements. It first builds heap from the input list of elements and then performs N "delete min" operations. In order to avoid creating new list, the deleted minimum (after each "delete min" operation) is swapped with the last element of the heap. The delete_min method of BinaryHeap class is thus modified to allow such functionality.

Worst case complexity:

Build heap performs in $O(N)$. Each of the N delete min operations, in the worst case, performs in $O(logN)$; as the root element is percolated all the way down to the leaf.

Worst case complexity = $O(NlogN)$

Average case complexity:

The average case complexity analysis of Heap sort is pretty involved. Here, a lower bound of $2NlogN - O(NloglogN)$ is established using simple mathematical tool. Using more complex analytical tools, a tighter bound of $2NlogN - O(N)$ can be established.

Consider an array of a random permutation of N distinct elements. A lower bound on the number of binary heaps H, that can made from N elements, can be shown to be $H > (N/(4e))^N$.

Since, build heap runs in $\Theta(N)$ on average, we need to only show bound on "delete min" operations. Each heap sort is associated with a sequence $D : d_1, d_2, ..., d_N$, where d_i is the depth to which ith root element is percolated down to, while performing $2d_i$ comparisons. The cost associated with these "delete min" operations for a particular D is given as,

$$C_D = \sum_{i=1}^{N} d_i$$

Number of comparisons = $2C_D$

Since, each depth d contains at most 2^d elements, total number of unique "delete min" sequences for a particular D sequence is given as,

$$S_D = 2^{d_1}2^{d_2}2^{d_3}...2^{d_N} = 2^{C_D}$$

The value of d_i lies between 1 and $log(N)$, giving at most $(logN)^N$ possible D sequences. (Actually, it lies between 1 and $log(N - i)$, but we set an upper bound). Furthermore, there are $NlogN - N + 1$ different values of C_D (multiple D sequences can give same C_D).
Let's choose a cost value of C_D as M. How many of the D sequences are associated with cost M?
We set an upper bound $(logN)^N$, i.e all the possible D sequences. Each of those sequences has further at most 2^M possible "delete min" sequences. Thus, the number of distinct "delete min" sequences that require exactly M cost is at most

$$(logN)^N 2^M$$

Consequently, the number of heaps with cost less than M is at most

$$\sum_{i=1}^{M-1}(logN)^N 2^i = (logN)^N \sum_{i=1}^{M-1} 2^i = (logN)^N(2^M - 2) < (logN)^N 2^M$$

Now, for $M = NlogN - O(NloglogN)$, the number of heaps is less than $(N/16)^N$, which is a tiny fraction of total number of possible heaps, $H > (N/(4e))^N$. Hence, average number of comparisons must be at least

$$2M = 2NlogN - O(NloglogN)$$

Even though the analysis looks complex, what we have done here is basically chose a value and argued that the average must be greater than this because, there are very few values less than it. In other words, for a list $[1, 2, 3, 4, 5, 6]$, the average must be greater than 2 because out of 6 values only one value is less than 2.

Merge sort

Merge sort is a recursive algorithm based on divide and conquer strategy. It works by diving the input array into two, recursively sorting each left and right half and then merging two halves.
Merge sort is not in-place sorting algorithm as it requires $O(N)$ memory space for temporary list. It is a stable sorting algorithm that maintains the relative position of duplicate elements (if implemented as such).

Running time of Merge sort can be obtained from the recurrence relation. Let $T(N)$ be the running time to sort N items. Since it takes linear time to merge two lists, the recurrence relation is given by,

$$T(1) = O(1)$$

$$T(N) = 2T(N/2) + N$$

Solving for $T(N)$,

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$

$$...$$

Adding both sides,

$$\frac{T(N)}{N} + \frac{T(N/2)}{N/2} + ... + \frac{T(2)}{2} = \frac{T(N/2)}{N/2} + \frac{T(N/4)}{N/4} + ... + \frac{T(1)}{1} + 1 + 1 + ... + 1$$

Telescoping the sum,

$$\frac{T(N)}{N} = \frac{T(1)}{1} + logN$$

Thus,

$$T(N) = \Theta(NlogN)$$

A major drawback of the merge sort is that it needs linear extra memory (though theoretically possible in $O(1)$).

Quick sort

Quick sort is also a recursive divide and conquer algorithm. It works by partitioning list into left and right sub-lists using a pivot element, such that elements in left $<=$ elements in right. It recursively sorts each sub-lists and then concatenates two sublists along with pivot element. It is a stable and in-place sorting algorithm.

Choosing the pivot element:

- First element: Terrible idea! If the list is sorted or reverse sorted, then all elements go into one of the sub-lists only. $O(N^2)$ time spent doing nothing!
- Random element: A safe option, unless random number generator is flawed. Furthermore, random number generation is rather expensive.
- Median: The best pivot you could choose. However, it is expensive to calculate median of elements.
- Median-of-three: An estimate of median can be obtained by taking median of the three numbers: first, middle and last. The implementation uses this pivot.

Performing the partition:

Partition can be performed in $O(N)$. Here is an illustration of the strategy.
Initialize i and j variables to first and second last element respectively.

```
Initial list
i-->                                    <--j
3       5       4       2       3       6       7       1
```

Swap last element and the pivot element (median of 3, 3, 1 = 3).

```
i-->                                    <--j
3       5       4       2       1       6       7       3
```

Move i towards right until it finds element greater than or equal to the pivot, in which case stop. Simultaneously move j towards left until it finds element less than or equal to the pivot, in which case stop.

```
i-->                        <--j
3       5       4       2       1       6       7       3
```

When i and j both stop and i is to the left of j, swap the elements.

```
i-->                        <--j
1       5       4       2       3       6       7       3

        i-->        <--j
1       5       4       2       3       6       7       3

        i-->        <--j
1       2       4       5       3       6       7       3

     <--j       i-->
1       2       4       5       3       6       7       3
```

At this moment, i and j have crossed. The final step is to swap the pivot element that was placed in the end of the list with the element where i is pointing.

```
1       2       3       5       3       6       7       4
```

Thus the partitions are:

```
[1       2]      [3]       [5       3       6       7       4]
```

As you might have noticed, strategy works with duplicate entries as well.

Since quick sort is recursive, recurrence relation will be used for the complexity analysis. Assume pivot is chosen randomly from the element. Then, since the partition step performs in linear time, the general recurrence relation is given by,

$$T(0) = T(1) = O(1)$$

$$T(N) = T(i) + T(N - i - 1) + N$$

Worst case complexity:

Worst case is the case when pivot chosen is the smallest element all the time. The recurrence relation for such case is:

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

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

$$...$$

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

Telescoping the sum,

$$T(N) = \sum_{i=2}^{N} i = \Theta(N^2)$$

Best case complexity:

Best case is the case when the list is already sorted or the randomly chosen pivot is always the median. The recurrence relation for such case is:

$$T(N) = 2T(N/2) + N$$

The relation is exactly the same as merge sort. Hence,

$$T(N) = \Theta(NlogN)$$

Average case complexity:

Since the pivot is chosen randomly from elements, we assume that each of the N possible sizes of the partition is equally likely.

$$E[T(i)] = E[T(N-i-1)] = \frac{1}{N}\sum_{i=0}^{N-1} i$$

Thus, the recurrence relation is given by,

$$T(N) = \frac{2}{N}\sum_{i=0}^{N-1} T(i) + N$$

$$NT(N) = 2\sum_{i=0}^{N-1} T(i) + N^2$$

$$(N-1)T(N-1) = 2\sum_{i=0}^{N-2} T(i) + (N-1)^2$$

Subtracting them,

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2N - 1$$

$$NT(N) = (N+1)T(N-1) + N$$

Rearranging the terms,

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{1}{N+1}$$

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{1}{N}$$

$$...$$

$$\frac{T(1)}{2} = \frac{T(0)}{1} + \frac{1}{2}$$

Telescoping the sum,

$$\frac{T(N)}{N+1} = \frac{T(0)}{1} + \sum_{i=2}^{N+1} \frac{1}{i} = O(log(N))$$

Thus,

$$T(N) = O(NlogN)$$

Bucket sort

Bucket sort is a linear time stable sorting algorithm that uses an additional information about the input array of items, the maximum value M. Bucket sort works by keeping an array $buckets$ of size M with each elements initialized to 0's. The input array is scanned and the array $buckets$ is updated for each $item$ as such $buckets[item] \mathrel{+}= 1$. The array $buckets$ is then scanned to output the sorted array.

Bucket sort takes $O(N)$ time to scan the element in the array and update the buckets. It takes $O(M)$ time to scan the buckets and output the sorted array.

Thus, the complexity is $O(N+M)$. If M is $O(N)$, then
Complexity = $O(N)$

To come in terms with $\Omega(NlogN)$ running time complexity of comparison based sorting algorithm, understand that the step, $buckets[item] \mathrel{+}= 1$, is actually performing M way comparison in one step.

Radix sort

Radix sort is basically multi-pass Bucket sort. If the size of elements in the input array is relatively small, Bucket sort can be performed sequentially on each indices or digits. Consider Bucket sorting N integers that are between 0 and 999. This would require the array $buckets$ of size 999, which is not ideal. Instead, we perform three Bucket sorts on each digits starting from the least significant digit and use a 10-size array $buckets$.

Unlike Bucket sort, more than one element that are different could fall on the same bucket. Thus we maintain lists. Furthermore, since each pass is stable, ordering determined in $(k-1)th$ pass is retained in future kth pass.

Complexity = $O(p(N+M))$, where p is the number of passes (max length of elements), N is the length of input array and B is the number of buckets.