Homework 2

CSE 2046 Analysis of Algorithms, Spring 2020

Şükriye Soyer 150116010

# Insertion sort

Insertion sort is a simple [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm" \o "Sorting algorithm) that builds the final [sorted array](https://en.wikipedia.org/wiki/Sorted_array" \o "Sorted array) (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as [quicksort](https://en.wikipedia.org/wiki/Quicksort" \o "Quicksort), [heapsort](https://en.wikipedia.org/wiki/Heapsort" \o "Heapsort), or [merge sort](https://en.wikipedia.org/wiki/Merge_sort" \o "Merge sort). However, insertion sort provides several advantages:

Simple implementation, efficient for (quite) small data sets, much like other quadratic sorting algorithms, More efficient in practice than most other simple quadratic (i.e., [O](https://en.wikipedia.org/wiki/Big_O_notation)(*n*2)) algorithms such as [selection sort](https://en.wikipedia.org/wiki/Selection_sort" \o "Selection sort) or [bubble sort](https://en.wikipedia.org/wiki/Bubble_sort) , [Adaptive](https://en.wikipedia.org/wiki/Adaptive_sort), i.e., efficient for data sets that are already substantially sorted: the [time complexity](https://en.wikipedia.org/wiki/Time_complexity) is [O](https://en.wikipedia.org/wiki/Big_O_notation)(*kn*) when each element in the input is no more than *k* places away from its sorted position [Stable](https://en.wikipedia.org/wiki/Stable_sort); i.e., does not change the relative order of elements with equal keys[In-place](https://en.wikipedia.org/wiki/In-place_algorithm); i.e., only requires a constant amount O(1) of additional memory space [Online](https://en.wikipedia.org/wiki/Online_algorithm); i.e., can sort a list as it receives it.

->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*)). ALso ı create sorted integer inputs sizes of 10,30,50,70,100,1000. Theoretically, as I expected, when size is 1000 slower than size of 10 ,100 times.

->The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(*n*2)).

ALso ı create reverse sorted integer inputs sizes of 10,30,50,70,100,1000. Actually size of 10 is faster than size of 100 ,100 times but my experiment 41 times faster.

->The average case insertion sort has a linear running time (i.e., O(*n*2)). ALso ı create randomly integer inputs sizes of 10,30,50,70,100,1000. Actually size of 10 is faster than size of 100 ,100 times but my experiment 25 times faster.Like theoretical expectations when our list was in order it worked faster than the reverse order.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 30 | 50 | 70 | 100 | 1000 |
| Random | 7.759\*10^-4 | 7.3661\*10^-4 | 0.0014103 | 0.0020056 | 0.0034759 | 0.0172073 |
| Order | 5.751\*10^-4 | 0.0010089 | 0.0010008 | 0.0021216 | 0.0037587 | 0.0191028 |
| Reverse Order | 6.107\*10^-4 | 6.815\*10^-4 | 0.0015476 | 0.0023758 | 0.0029184 | 0.0248069 |

Merge Sort

In [computer science](https://en.wikipedia.org/wiki/Computer_science" \o "Computer science), **merge sort** (also commonly spelled **mergesort**) is an efficient, general purpose, [comparison-based](https://en.wikipedia.org/wiki/Comparison_sort" \o "Comparison sort) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm" \o "Sorting algorithm). Most implementations produce a [stable sort](https://en.wikipedia.org/wiki/Sorting_algorithm" \l "Stability" \o "Sorting algorithm), which means that the order of equal elements is the same in the input and output.

In the best case, the input is already sorted (i.e., is one run), so the natural merge sort need only make one pass through the data. O(*n* log *n*) typical, O(*n*) natural variant. Size of 100 ,6.4 times slower than size of 10 .

. Theoretically, when O(nlogn) 23 times slower than size of 10 but when O(n) should be 10 times slower .

Merge sort has an [average](https://en.wikipedia.org/wiki/Average_performance" \o "Average performance) and [worst-case performance](https://en.wikipedia.org/wiki/Worst-case_performance" \o "Worst-case performance) of [O](https://en.wikipedia.org/wiki/Big_O_notation)(*n* log *n*). I create firstly sorted,reverse sorted and randomly inputs ; also ı think there wasn't a big time difference between these series. Like theoretical expectations. As theoretically worst case and average case performances are close. Size of 100 ,9.79 times slower than size of 10 . Theoretically should be 23 times it is close to expected.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 30 | 50 | 70 | 100 | 1000 |
| Random | 4.48\*10^-4 | 7.774\*10^-4 | 0.0017279 | 0.0018746 | 0.0047277 | 0.0207055 |
| Order | 4.712\*10^-4 | 0.002061 | 0.0016957 | 0.0032088 | 0.0029608 | 0.0186171 |
| Reverse Order | 5.262\*10^-4 | 0.0012557 | 0.0015423 | 0.0021491 | 0.0030413 | 0.0242707 |

Max-Heap

A **binary heap** is a [heap](https://en.wikipedia.org/wiki/Heap_(data_structure)" \o "Heap (data structure)) [data structure](https://en.wikipedia.org/wiki/Data_structure) that takes the form of a [binary tree](https://en.wikipedia.org/wiki/Binary_tree" \o "Binary tree). Binary heaps are a common way of implementing [priority queues](https://en.wikipedia.org/wiki/Priority_queue" \o "Priority queue). Heaps where the parent key is greater than or equal to (≥) the child keys are called *max-heaps*; those where it is less than or equal to (≤) are called *min-heaps*. Efficient ([logarithmic time](https://en.wikipedia.org/wiki/Logarithmic_time" \o "Logarithmic time)) algorithms are known for the two operations needed to implement a priority queue on a binary heap: inserting an element, and removing the smallest or largest element from a min-heap or max-heap, respectively. Binary heaps are also commonly employed in the [heapsort](https://en.wikipedia.org/wiki/Heapsort" \o "Heapsort) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm" \o "Sorting algorithm), which is an in-place algorithm because binary heaps can be implemented as an [implicit data structure](https://en.wikipedia.org/wiki/Implicit_data_structure" \o "Implicit data structure), storing keys in an array and using their relative positions within that array to represent child-parent relationships.

Binary heaps, is easily seen to run in *O*(*n* log *n*) time: it performs *n* insertions at *O*(log *n*) cost each.In theoretical expectations; worst ,best, average case ; O(nlogn) but ı think this is true for large size like 1000 integers therefore ı think reverse order is faster than order or random variables like 10 ,30 integers.In the best case size of 10 is faster than 55 times than size of 100 also theoretically should be 23 times. In the worst case and average case size of 10 is faster than 55 times than size of 100 so like theoretically best average and worst cases O(nlogn).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 30 | 50 | 70 | 100 | 1000 |
| Random | 8,086\*10^-4 | 0,0016531 | 0,0024133 | 0,0046877 | 0,0044094 | 0,0186157 |
| Order | 9,588\*10^-4 | 0,0026695 | 0,0011834 | 0,0037172 | 0,0029814 | 0,0211882 |
| Reverse Order | 7,279\*10^-4 | 0,0022325 | 0,0030132 | 0,0036343 | 0,0049663 | 0,0201662 |

Quick Select Algorithm

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **quickselect** is a [selection algorithm](https://en.wikipedia.org/wiki/Selection_algorithm" \o "Selection algorithm) to find the *k*th smallest element in an unordered list. It is related to the [quicksort](https://en.wikipedia.org/wiki/Quicksort" \o "Quicksort) sorting algorithm. Like quicksort, it was developed by [Tony Hoare](https://en.wikipedia.org/wiki/Tony_Hoare" \o "Tony Hoare), and thus is also known as **Hoare's selection algorithm**.[[1]](https://en.wikipedia.org/wiki/Quickselect#cite_note-1) Like quicksort, it is efficient in practice and has good average-case performance, but has poor worst-case performance. Quickselect and its variants are the selection algorithms most often used in efficient real-world implementations.

Quickselect uses the same overall approach as quicksort, choosing one element as a pivot and partitioning the data in two based on the pivot, accordingly as less than or greater than the pivot. However, instead of recursing into both sides, as in quicksort, quickselect only recurses into one side – the side with the element it is searching for. This reduces the average complexity from O(*n* log *n*) to O(*n*), with a worst case of O(*n*2) and best case is O(n).

Worst Case:

The most unbalanced partition occurs when one of the sublists returned by the partitioning routine is of size *n* − 1. 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. This means that the [call tree](https://en.wikipedia.org/wiki/Call_stack" \o "Call stack) is a linear chain of *n* − 1 nested calls. The *i*th call does *O*(*n* − *i*) work to do the partition, and {\displaystyle \textstyle \sum \_{i=0}^{n}(n-i)=O(n^{2})}, so in that case Quicksort takes *O*(*n*²) time. I used random list for it. As theoretically In the worst case size of 10 is faster than 89 times than size of 100 .

Best Case:

In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. there are only *O*(*n*) calls at each level, this is subsumed in the *O*(*n*) factor. As theoretically In the worst case size of 10 is faster than 6.5 times than size of 100 .

Average Case:

To sort an array of *n* distinct elements, quicksort takes *O*(*n* log *n*) time in expectation, averaged over all *n*! permutations of *n* elements with [equal probability](https://en.wikipedia.org/wiki/Uniform_distribution_(discrete)" \o "Uniform distribution (discrete)). We list here three common proofs to this claim providing different insights into quicksort's workings.

Select pivot is very important a quicksort algorithm should always aim to choose the middle-most element as its pivot. Some algorithms will literally select the center-most item as the pivot, while others will select the first or the last element. But when we say “middle-most” element, what we mean is an element at the **median** of the entire unsorted collection. This ends up being super crucial because we want the two partitioned halves — the elements smaller than the pivot and the elements larger than the pivot — to be mostly equal.

In the fourth question ı select pivot first element and ı didnt order it but, fifth question ı chose median as pivot it is faster than fourth question. . As theoretically should be 23 times but my in experiment the worst case size of 10 is faster than 6.31 times than size of 100 .

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 30 | 50 | 70 | 100 | 1000 |
| Random | 5.751\*10^-4 | 7.703\*10^-4 | 0.0014563 | 0.0019938 | 0.0036874 | 0.0195685 |
| Order | 8.146\*10^-4 | 0.0018629 | 0.0011241 | 0.0026508 | 0.0028036 | 0.0228513 |
| Reverse Order | 4.972\*10^-4 | 0.0014624 | 0.0014469 | 0.0026982 | 0.0028519 | 0.0130132 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 30 | 50 | 70 | 100 | 1000 |
| Random | 5.751\*10^-4 | 7.703\*10^-4 | 0.0054496 | 0.0031404 | 0.0091473 | 0.0364774 |
| Order | 0.0015984 | 0.0019972 | 0.0033135 | 0.0080513 | 0.0068321 | 0.0338233 |
| Reverse Order | 0.0012951 | 0.0013265 | 0.0034565 | 0.0070769 | 0.0089619 | 0.0305484 |

In random sort at big size from slowest to fastest in terms of speed : Quick Select Sort - Merge Sort–Quick Select Sort with using first element -Max Heap Sort- Insertion Sort

In random sort at small size from slowest to fastest in terms of speed : Quick Select Sort - Max Heap Sort- Merge Sort–Quick Select Sort with using first element - Insertion Sort

In order sort in big size from slowest to fastest in terms of speed : Quick Select Sort - Merge Sort–Quick Select Sort with using first element -Max Heap Sort- Insertion Sort

In order sort in small size from slowest to fastest in terms of speed : Quick Select Sort -Max Heap Sort- Merge Sort–Quick Select Sort with using first element -Insertion Sort

In reverse order sort in big size from slowest to fastest in terms of speed : Quick Select Sort - Merge Sort–Quick Select Sort with using first element -Max Heap Sort- Insertion Sort

In reverse order sort in small size from slowest to fastest in terms of speed : Quick Select Sort -Max Heap -–Quick Select Sort with using first element -Merge Sort- Insertion Sort