**ALGORITHM PRESENTATION OF INSERTION SORT**

**1. Idea**

- Sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

**2. Insertion Sort working:**

Consider an array: {13, 12, 14, 6, 7}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 13 | 12 | 14 | 6 | 7 |

Step 1: First 2 elements of the array are compared in insertion sort

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 13 | 12 | 14 | 6 | 7 |

* 13 is greater than 12 so they are not in ascending order and 13 is not at it’s correct position. Swap 13 and 12.
* So 12 is stored in a sorted array

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 13 | 14 | 6 | 7 |

Step 2: Move to the next 2 elements and compare them

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 13 | 14 | 6 | 7 |

* Here 14 is greater than 13, both elements are in ascending order, so no swapping. 13 is also stored in a sorted array along with 11

Step 3:

* Move to the next two elements which are 14 and 6

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 13 | 14 | 6 | 7 |

* Both 14 and 6 are not at their correct place so swap them

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 13 | 6 | 14 | 7 |

* After swapping, elements 13 and 6 are not sorted, then swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 6 | 13 | 14 | 7 |

* Here, 12 and 6 are not sorted, swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 12 | 13 | 14 | 7 |

* Here, 6 is at correct position

Step 4:

* Move to the next two elements 14 and 7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 12 | 13 | 14 | 7 |

* Swap between both

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 12 | 13 | 7 | 14 |

* 7 is smaller than 13, swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 12 | 7 | 13 | 14 |

* 7 is smaller than 12, swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 7 | 12 | 13 | 14 |

* Finally, the array is completely sorted.

**3. Time complexity**

- The worst-case time complexity of the Insertion Sort is O(n^2)

- The average case time complexity of the Insertion Sort is O(n^2)

- The time complexity of best case is O(n)

**4. Space complexity**

- The auxiliary space complexity of Insertion Sort is O(1)

**5. Variants/Improvements of Insertion Sort**

- Binary insertion sort is a sorting algorithm which is similar to the [insertion sort](http://www.geeksforgeeks.org/insertion-sort/), but instead of using linear search to find the location where an element should be inserted, we use [binary search](https://www.geeksforgeeks.org/binary-search/). Thus, we reduce the comparative value of inserting a single element from O (N) to O (log N). (Source: <https://www.geeksforgeeks.org/binary-insertion-sort/>)

- Early Termination: During the insertion process, you can break out of the loop early if you find the correct position for the current element. This can prevent unnecessary comparisons when the array is almost sorted or has a lot of elements already in their correct positions.

- Move Instead of Swap: Instead of using swap operations to move elements in the sorted part of the array, you can use a single temporary variable to hold the element being inserted and shift elements in the sorted part by one position. This can be more efficient in terms of memory access and data movement.

- [Shell sort](http://en.wikipedia.org/wiki/Shellsort)is mainly a variation of [Insertion Sort](https://www.geeksforgeeks.org/insertion-sort/). In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of ShellSort is to allow the exchange of far items. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h’th element are sorted.

(Source: <https://www.geeksforgeeks.org/shellsort/>)

- Adaptive Insertion Sort: Adaptive insertion sort is an enhancement that takes advantage of the partially sorted nature of the array. If the array is already partially sorted, the number of comparisons and shifts can be reduced.

- Combining Insertion Sort with Other Sorting Algorithms: Insertion sort can be used in combination with other sorting algorithms, such as merge sort or quicksort, for small sub-arrays. This hybrid approach can take advantage of the strengths of each sorting algorithm.

**ALGORITHM PRESENTATION OF HEAP SORT**

**1. Idea**

- The array will first be converted into a Max-heap using the heapify operation. Next, the root node of the Max-heap will be sequentially removed and replaced with the last node in the heap. After each replacement, the root of the heap will be heapified to maintain the Max-heap property. This process will be repeated until the size of the heap becomes greater than 1.

**2. Working of Heap Sort**

- Consider the array: {5, 11, 4, 6, 2}

Step 1:

* Build complete binary tree from the array

A diagram of a line with numbers and circles

Description automatically generated

Step 2:

* Transform into max heap: to transform into max heap, the parent node should always be greater than the children nodes
  + 5 is smaller than 11 so swap them

A black line with red numbers and a black line

Description automatically generated

* + 5 is smaller than 6 so swap them to build max heap

A diagram of a diagram

Description automatically generated

* + Now we have max heap

Step 3

* Remove the root element (11) from the max heap. To delete it, try to swap it with the last node, heapify the tree again to max heap.
* Resulted heap should like this:

A black line with red numbers and a circle

Description automatically generated

Step 4:

* Repeate the above step

A black line with red numbers and a circle

Description automatically generated

Step 5:

* Remove the root (5) again and perform heapify

A black and white image of a long black stick

Description automatically generated

Step 6:

* Remove the root (4) and heapify again.

A black circle with red numbers

Description automatically generated

Step 7:

* Delete the last root and the sorted array will be:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 4 | 5 | 6 | 11 |

**3. Time complexity**

- The worst-case time complexity of the Heap Sort is O(nlog(n))

- The average case time complexity of the Heap Sort is O(nlog(n))

- The time complexity of best case is O(nlog(n))

**4. Space complexity**

- The auxiliary space complexity of Heap Sort is O(1)

**5. Variants/Improvements of Heap Sort**

- Min Heap Sort: Beside the Heap Sort builds a max heap to sort in ascending oder, we can build a min heap to sort elements descending order.

- Iterative Heap Sort: the standard Heap Sort using recursion to heapify. However, an iterative version can be implemented using loops instead of recursion, which might be more memory-efficient and suitable for language or environments with limited stack space.

- Bottom-up Heap Sort: This variation starts by creating an array-based heap without using the standard heapify operation. Instead, it starts from the first non-leaf node and sifts down each element to build the heap. Afterward, it performs the sorting by repeatedly extracting the root and heapifying the remaining elements.

- D-ary Heap Sort: Basic Heap Sort uses Binary Heap, each node has 2 children. In D-ary Heap Sort, the heap is generalized to have at most D children for each node. This reduce the height of the heap, potentially improving performance.

- Parallel Heap Sort: in parallel computing environments, we can divide the heap into smaller heaps and sort them concurrently. After that, the sorted smaller heaps are merged to get the final sorted result, potentially reducing the sorting time.

- Adaptive Heap Sort: aims to optimize the Heap Sort performance for partially sorted or nearly sorted array. If the input is partially sorted and adjusts the heapify operation accordingly to reduce unnecessary work.

- In-place Heap Sort: In standard Heap Sort, it requires extra space for the heap data structure. However, with some modifications, it can be implemented in an in-place version that sorting without using additional memory for the heap.

**ALGORITHM PRESENTATION OF FLASH SORT**

**1. Idea**

- Flash Sort is an in-place sorting algorithm with a time complexity of O(n), non-recursive, and consists of three steps: (1) Data classification, which means making assumptions about the data distribution, for example, assuming a uniform distribution, to find an estimated formula for the position (class) of an element after sorting. (2) Global permutation, which involves shifting elements within the array to their respective classes. (3) Local sorting, which means sorting the elements within the range of each class.

**2. Working of Flash Sort**

Stage 1: Data classification

* The necessary elements for data classification are:
* The number of classes, denoted by m.
* An array of m elements that record the starting positions of each class.
* Step 1: Determine the minimum (min) and maximum (max) values of the elements in the array A.
* Step 2: Determine which class each element belongs to using the following formula: l[i] = 1 + floor((m - 1) \* (A[i] - min) / (max - min)) The formula above indicates the class that contains element i of array A.
* Step 3: Record the starting positions of each class in the array A. A class i is considered full when its starting position (l[i]) is in the correct position within array A. Therefore, a class is considered empty when its starting position is at the end of its correct position in class A. To place an element into its class, we decrement l[i] until it reaches its correct position, which means it is full. To determine the starting and ending positions of each class, we need to know the size of each class. Thus, to consider classes as empty, the starting position of each class must be the position where it should end in array A. Therefore, we have the following formula: l[i] = l[i] + l[i-1]

Stage 2: Global Permutation

* After the preparation in Stage 1 is completed, we begin the process of sorting the elements into their respective classes. This process will form cycles of permutations: whenever we move an element from one position to another, we have to remove the current element occupying that position and continue with the element that was removed, repeating this process until we return to the original position to complete the cycle.
* In other words, during this stage, we rearrange the elements within the array, swapping them in cycles to place each element in its correct class. This process continues until all elements are sorted into their respective classes, forming a cycle of permutations for each class.

Stage 3: Local Sorting

* The current array has been approximately sorted because the elements are already placed in their correct classes. To achieve the final sorted order, we will use the Insertion Sort algorithm to optimize the sorting process.

Demo of algorithm with 7 elements:

A diagram of a number system

Description automatically generated

A diagram of a flash diagram

Description automatically generated

A diagram of a flash

Description automatically generated

A diagram of a flash diagram

Description automatically generated

A diagram of a flash

Description automatically generated

A diagram of a flash

Description automatically generated

A diagram of a number

Description automatically generated

Source of Demo: https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh

**3. Time complexity**

- The worst-case time complexity of the Flash Sort is O(n^2)

- The average case time complexity of the Flash Sort is O(n)

- The time complexity of best case is O(n)

**4. Space complexity**

- The space complextity of Flash Sort is O(n)

**5. Variants/Improvements of Flash Sort**

- Improved Bucket Selection: in the Flash Sort, the number of classes is calculated by the function: m = floor(0.45\*n). Researchers have proposed alternative formulas to find an optimal value for m based on the input data distribution to further improve the algorithm's efficiency.

- Hybrid Flash Sort: Hybrid combine Flash Sort with other sorting algorithms to utilize the strength of each.

- Parallel Flash Sort: this variant is dividing the input data into smaller subsets and apply Flash Sort concurently on each subset. The results of all subsets are then combined to get the final sorted array. This can speed up the sorting process on multi-core processors and parallel computing architectures.

- Adaptive Flash Sort: similar to Heap Sort, this improved version of Flash Sort detects if input is partially sorted and adapts its behavior accordingly. The algorithm may avoid some unnecessary classification and permutation steps to optimize its performance.

List of references:

Insertion Sort:

https://www.geeksforgeeks.org/insertion-sort/

Heap Sort:

<https://www.geeksforgeeks.org/heap-sort/>

https://visualgo.net/

Flash Sort:

<https://www.studocu.com/vn/document/truong-dai-hoc-su-pham-ky-thuat-thanh-pho-ho-chi-minh/computer-architecture-and-assembly-language/flash-sort/60588066?fbclid=IwAR2wDJv7AifCShnyfWCqY1YVPyRzKV6PbJCHGfCYV_3x5CWye-STRI492bo>

<https://www.youtube.com/watch?v=CAaDJJUszvE&t=445s>

https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh