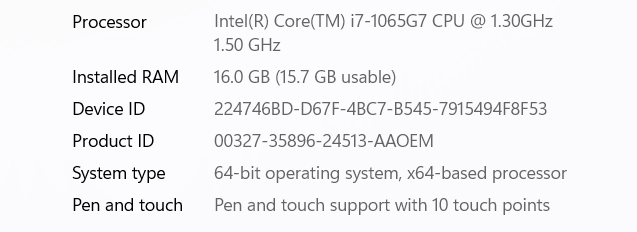
REPORT: SORTING

# Information

|  |  |
| --- | --- |
| Full Name | Student ID |
| Trần Nguyễn Minh Hoàng | 22127131 |
| Đỗ Đình Hải | 22127095 |
| Phạm Thành Đạt | 22127064 |
| Đỗ Phan Tuấn Đạt | 22127057 |

# Introduction

## Specifications of device used for sorting



# Algorithm presentation

## Selection sort

## Insertion sort

### 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.

### Step-by-step description

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.

### Complexity evaluation

* Time complexity:
  + Best case: O(n)
  + Average case: O(n^2)
  + Worst case: O(n^2)
* Space complexity: O(1)

### Variants/Improvements

* 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).
* 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.
* 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.

## Bubble sort

## Shaker sort

## Shell sort

## Heap sort

### 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.

### Step-by-step description

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 achieve this, swap (11) node with the last node and remove it from the heap. Heapify the tree again to get another max heap

A black line with red numbers and a circle

Description automatically generated

Step 4: Repeat the above step

A black line with red numbers and a circle

Description automatically generated

Step 5: Remove the root (5) and perform heapify again

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 |

### Complexity evaluation

* Time complexity:
  + Best case: O(n\*logn)
  + Average case: O(n\*logn)
  + Worst case: O(n\*logn)
* Space complexity: O(1)

### Variants/Improvements

* 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, in which 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.

## Merge sort

## Quick sort

### Idea

Divide the given array into two sections using a partitioning element called a pivot. The division performed is such that all elements to the left side of the pivot are smaller than the pivot, and all elements to the right side of the pivot are greater than the pivot. Pivot element can be first element, last element, middle element, or even a random element in the array.

### Step-by-step description

Below is the algorithm for quick sort using the last element of the array as the pivot:

Step 1 − Choose the highest-index element as Pivot.

Step 2 − Take another variable to indicate Low (lowest index - 1).

Step 5 – Starting from index i = Low + 1, while i < index of Pivot, compare i-th element with Pivot

Step 5.1 – If value of i-th element is smaller than Pivot, increase Low by 1 and swap i-th element with Low-th element.

Step 5.2 – Increase I by 1 and return to step 5.1.

Step 6 – Swap (i+1)-th element with Pivot. The array is now divided into two sub-arrays. The left sub-array consists of all elements smaller than Pivot, and the right sub-array consists of all elements greater (or equal, if any) elements than Pivot.

Step 7 – Recursively perform the algorithm, starting from Step 1, for each sub-array.

When the process completely terminates, the resulting array will be a sorted array.

### Complexity evaluation

* Time complexity:
  + Best case: O(n\*logn)
  + Average case: O(n\*logn)
  + Worst case: O(n^2)
* Space complexity: O(1)

### Variants/Improvements

Quick sort can be sped up significantly by choosing the median of several elements as the Pivot.

## Counting sort

## Radix sort

## Flash sort

# Referenced resources

* Quick sort:
  + <https://www.ques10.com/p/65800/write-quicksort-algorithm-using-last-element-as-pi/>
  + <https://www.geeksforgeeks.org/quick-sort/>
  + [Analysis of quicksort (article) | Quick sort | Khan Academy](https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort)
* Insertion sort:
  + <https://www.geeksforgeeks.org/binary-insertion-sort/>
  + <https://www.geeksforgeeks.org/shellsort/>