VNUHCM - University of Science
fit@hcmus

CSC10004 – Data Structures and Algorithms

Session 04
Sorting Algorithms

Instructors:
Dr. Lê Thanh Tùng

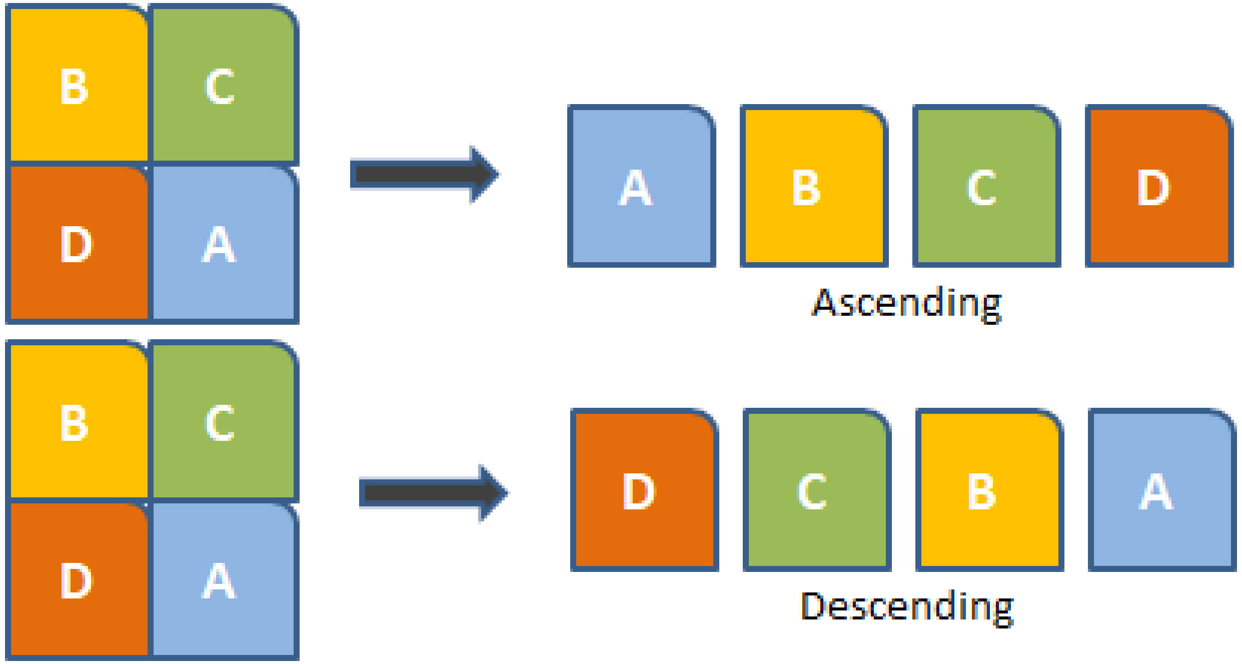
Content

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Heap Sort

Sorting Algorithm

What is Sorting?

- Sorting is a process organizes **a collection of data** into **ascending/descending** order



What is Sorting ?

- Sort key: data item which determines order

	Name	Age	Score
0	Peter	18	7
1	Riff	15	6
2	John	17	8
3	Michel	18	7
4	Sheli	17	5

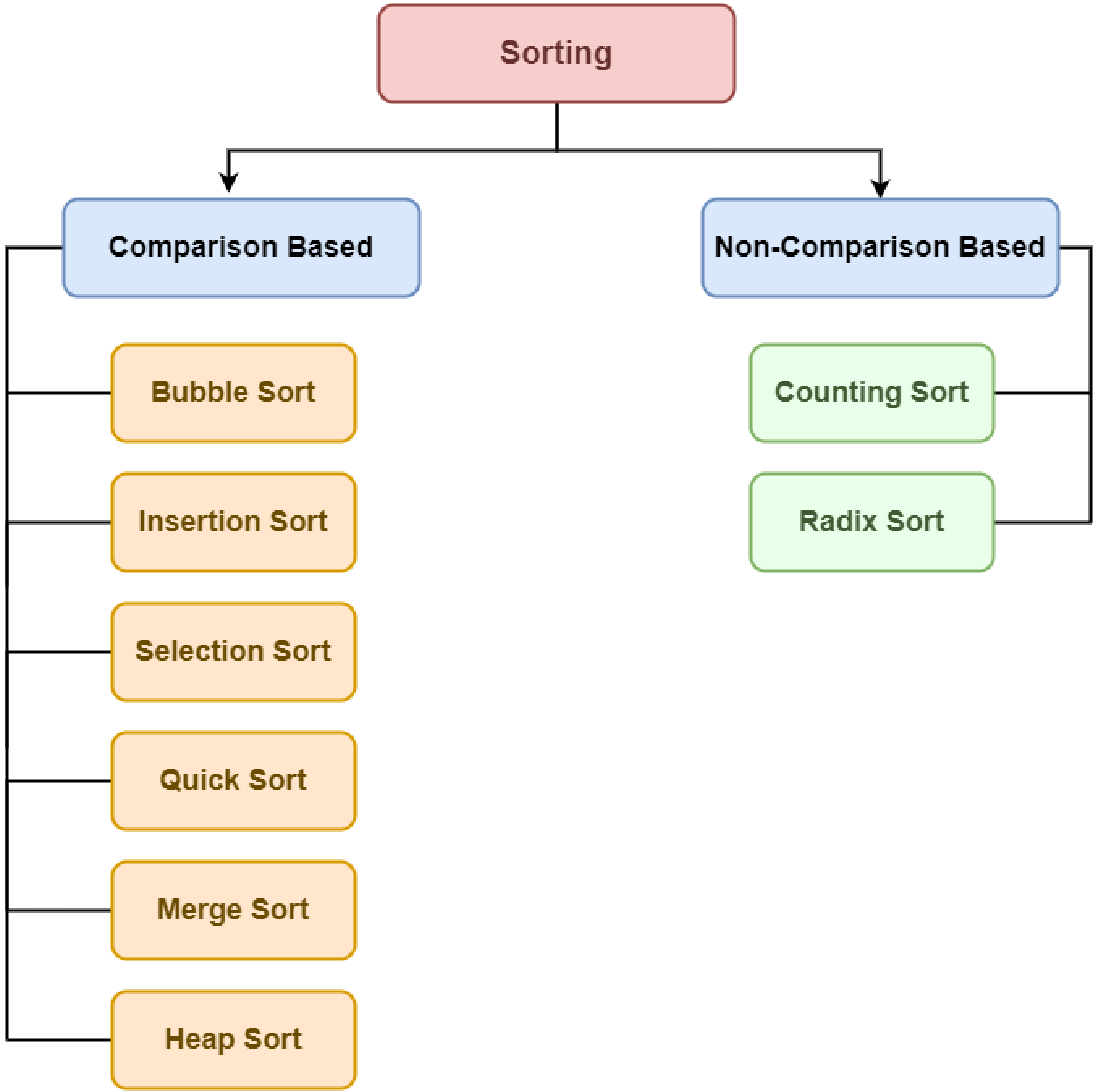
The sort key can be chosen as

- **Name**: arrange student
- **Age**: focus on age
- **Score**: consider the result

Classification of sorting algorithms

- **Memory usage**: in-place sort / not in-place sort
  - **In-place** (algorithm): sorts the data without using any additional memory.
- **By stability**: maintain the relative order of the records with equal keys
- **Comparison:** utilize comparison or not?
- **Adaptability:** whether the pre-sorted-ness of the input affects the running time or not
- **Data Location:** Internal or external
  - **Internal**: data **fits in** memory
  - **External**: data must reside on **secondary storage**
- We will analyze only **internal** sorting algorithms

Classification of sorting algorithms



Sorting algorithms

- Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.
- Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.
- A comparison-based sorting algorithm makes ordering decisions only based on comparisons.

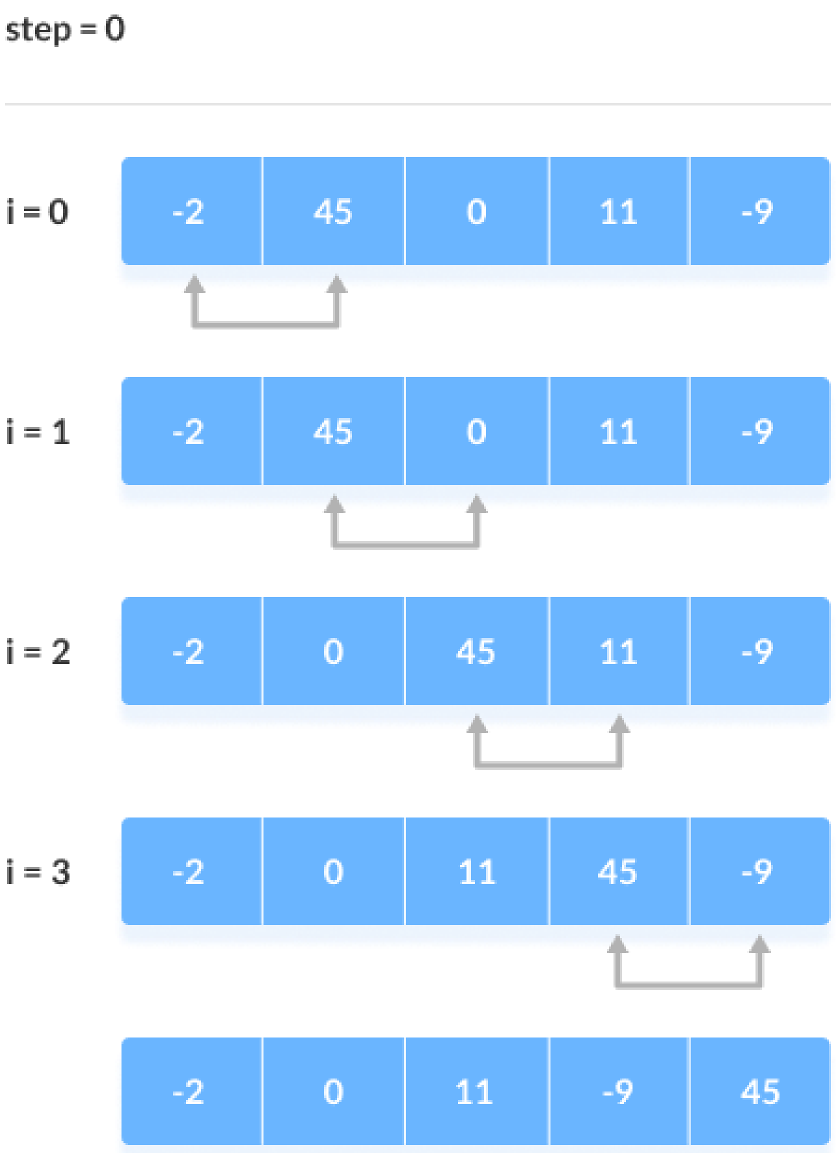
Bubble Sort

Bubble Sort

- **Main idea**: compares two adjacent elements and swaps them until they are in the intended order

5	1	12	-5	16	unsorted
5	1	12	-5	16	5 > 1, swap
1	5	12	-5	16	5 < 12, ok
1	5	12	-5	16	12 > -5, swap
1	5	-5	12	16	12 < 16, ok
1	5	-5	12	16	1 < 5, ok
1	5	-5	12	16	5 > -5, swap
1	-5	5	12	16	5 < 12, ok

- Step 1: Compare and Swap
  - If i^{th} and $(i+1)^{th}$ elements are in the incorrect positions, swap them



Bubble Sort

- Step 2: Remaining Iteration
    - Repeat Step 1
    - Until sorted list or len(list) – 1 times

```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

Bubble Sort


```
BUBBLE-SORT(A,n)
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order

for i ← 0 to n − 2 do
    for j ← 0 to n − 2 − i do
        if A[j+1] < A[j]
            swap A[j+1] and A[j]
```

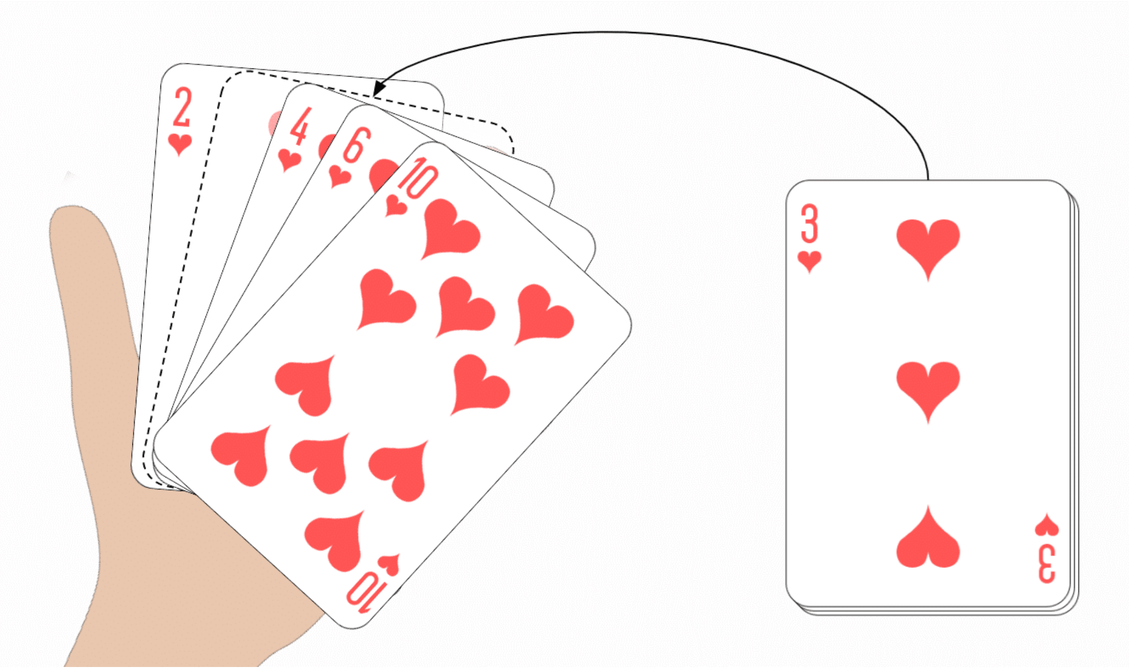
Basic operation

- What is the best and worst case of Bubble Sort
- What is the order of growth function of Bubble Sort ?

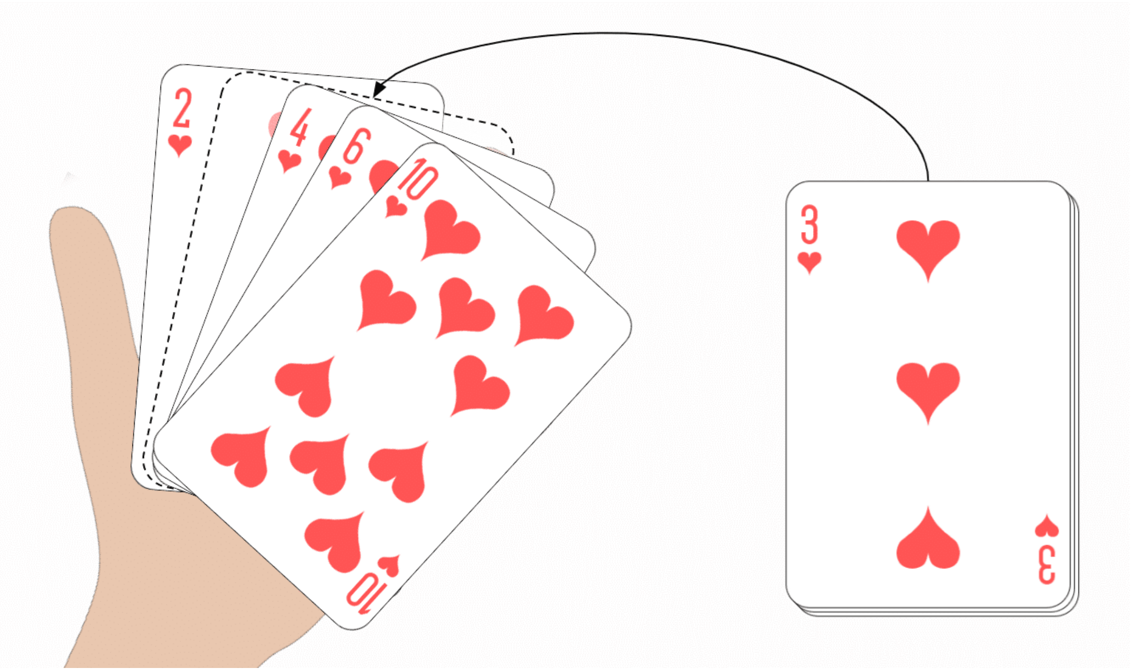
Insertion Sort

Insertion Sort

- Main idea: Insertion sort works similarly as we sort cards in our hand in a card game



- The first element in the array is assumed to be sorted.
- Try to sort the unsorted element – second card

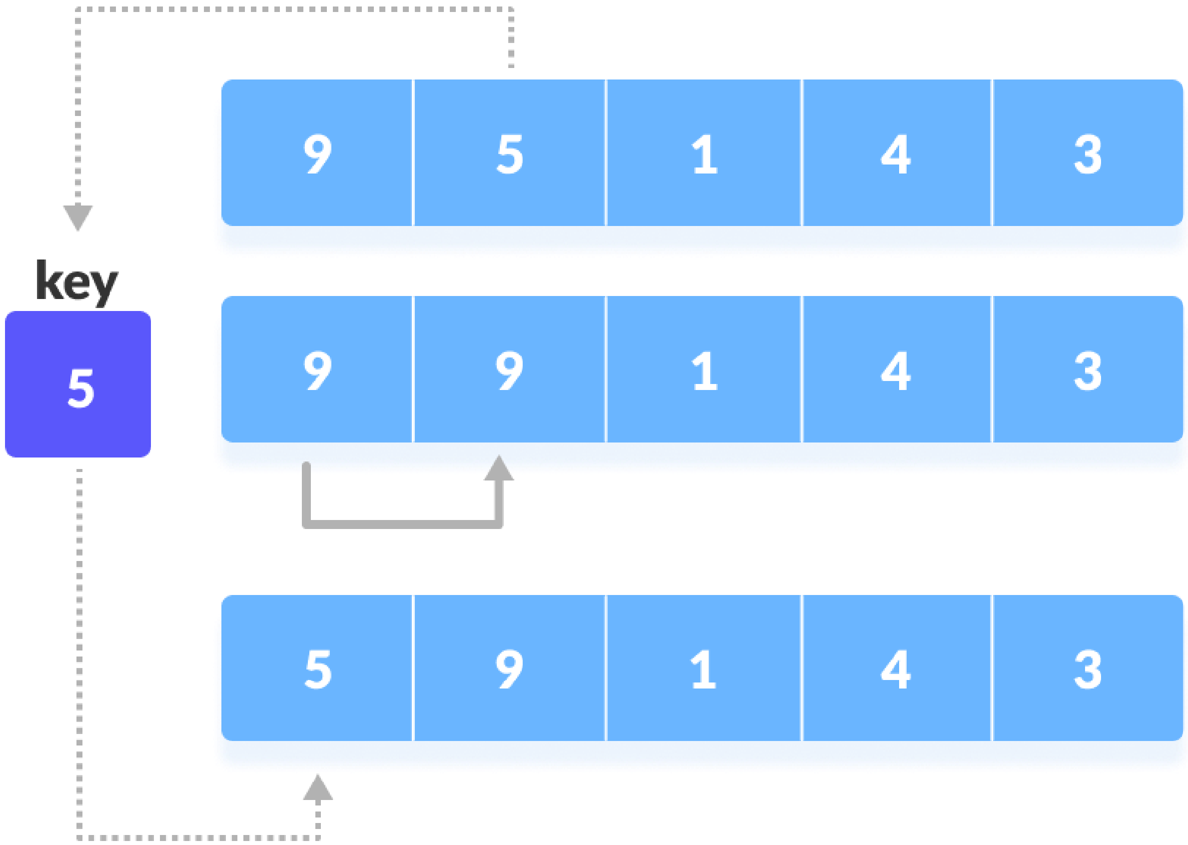


Insertion Sort

- For example, we consider the following array

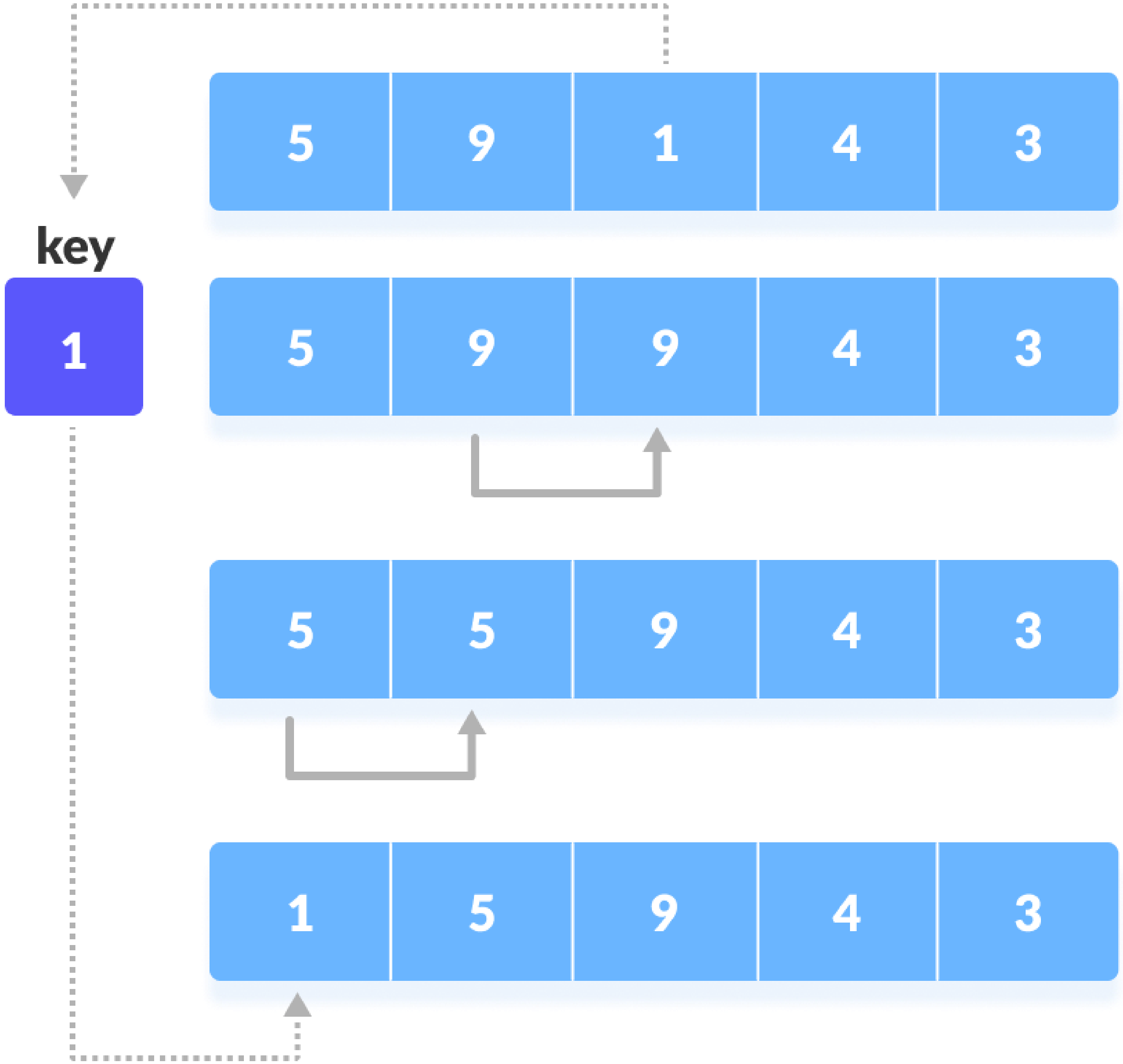
9	5	1	4	3

- Consider the first “unsorted” element – 5
  - Find the place to put the unsorted one



Insertion Sort

- The first two elements are sorted
- Placed the key element just behind the element smaller than it



- Shift the array to right to find the suitable position to put key element

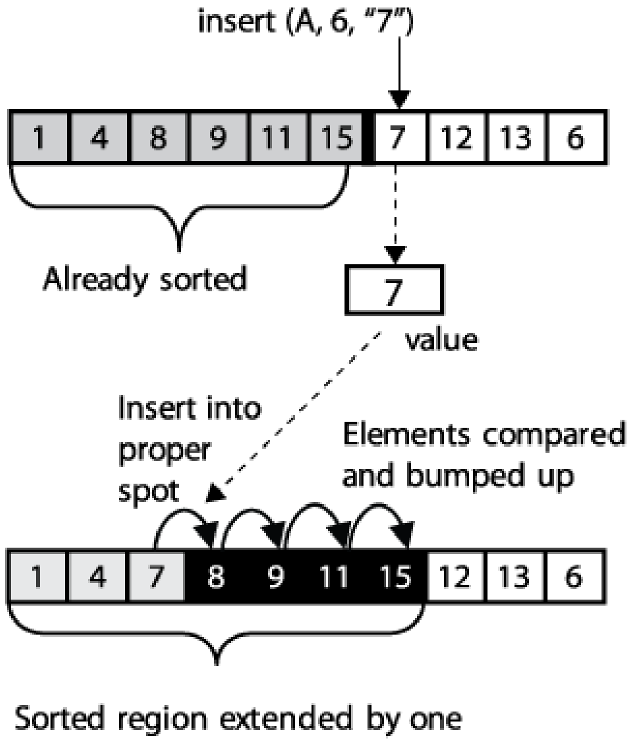
INSERTION SORT

Best	Average	Worst
$O(n)$	$O(n^2)$	$O(n^2)$

Array

```
sort (A)
    for i = 1 to n – 1 do
        insert (A, i, A[i])
end

insert (A, pos, value)
    i = pos – 1
    while (i ≥ 0 and A[i] > value) do
        A[i + 1] = A[i]
        i = i – 1
    A[i + 1] = value
end
```



- **What is the best case of insertion sort**

```
void insertionSort(int array[], int size) {
  for (int step = 1; step < size; step++) {
    int key = array[step];
    int j = step - 1;
    // Compare key with each element on the left of it
    // until an element smaller than it is found.
    while (key < array[j] && j >= 0) {
        array[j + 1] = array[j];
        --j;
    }
    array[j + 1] = key;
  }
}
```

- **What is the worst case of insertion sort**

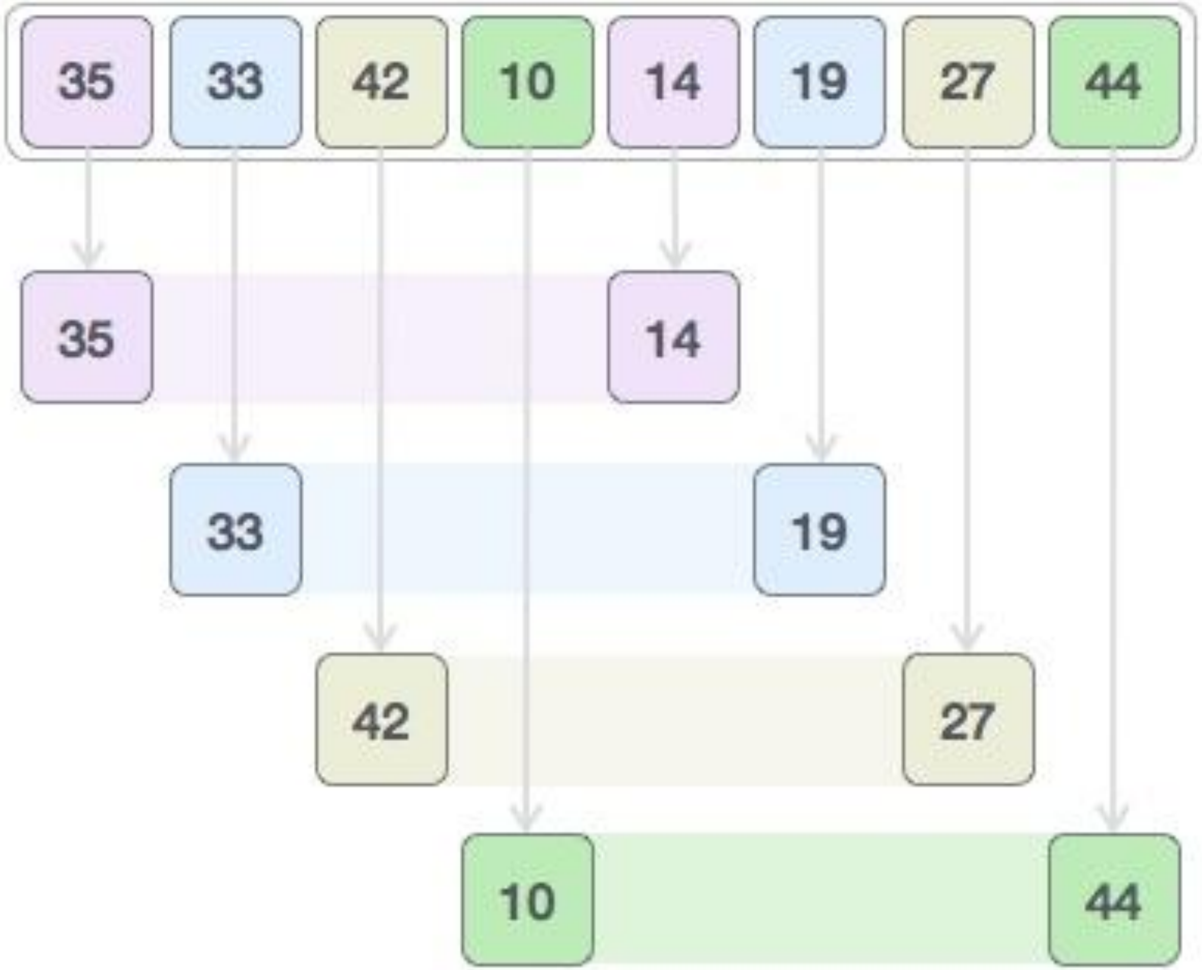
```
void insertionSort(int array[], int size) {
  for (int step = 1; step < size; step++) {
    int key = array[step];
    int j = step - 1;
    // Compare key with each element on the left of it
    // until an element smaller than it is found.
    while (key < array[j] && j >= 0) {
        array[j + 1] = array[j];
        --j;
    }
    array[j + 1] = key;
  }
}
```

Shell Sort

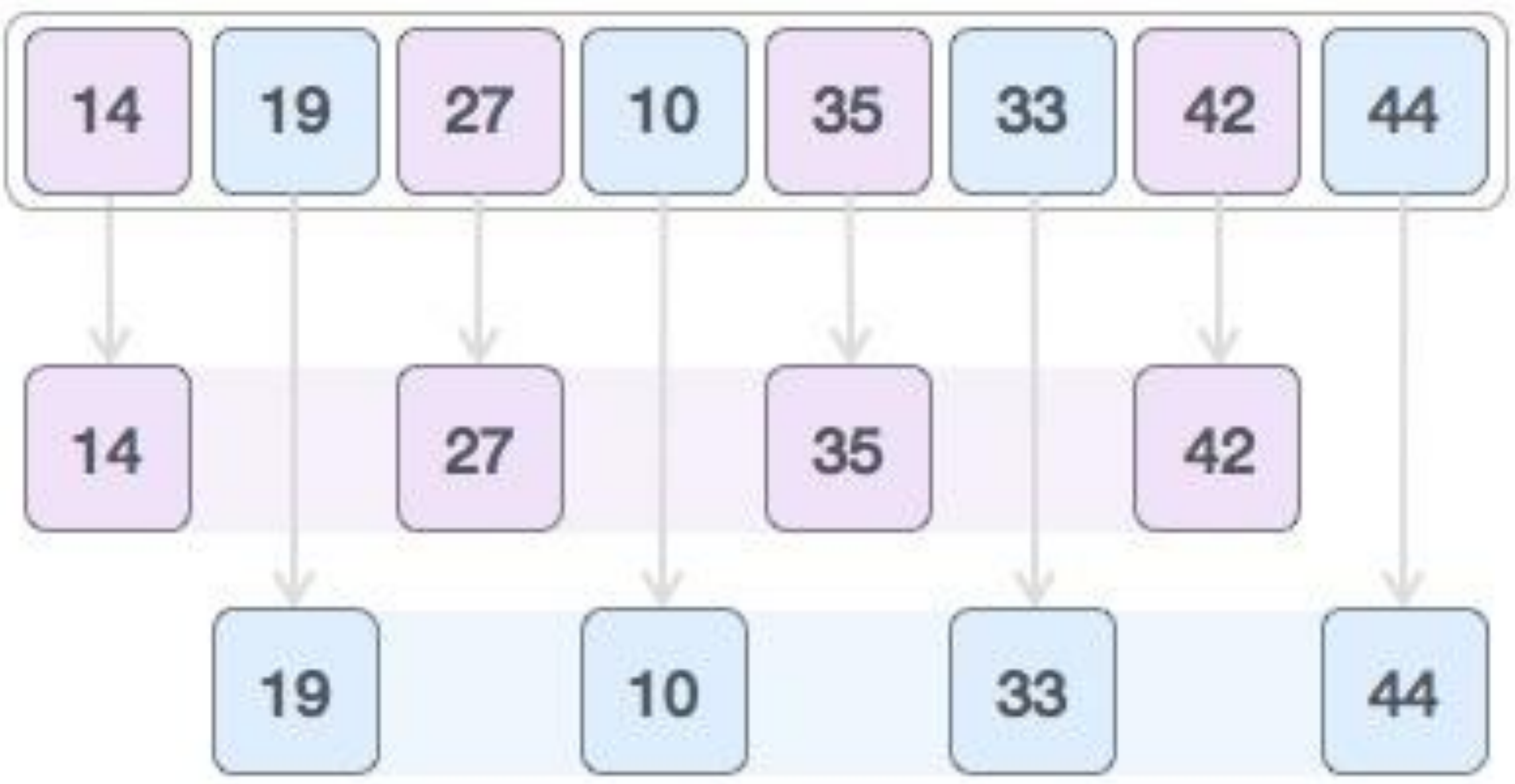
Shell Sort

- Shell sort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead
- 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

Shell Sort



Shell Sort



- However, the Shell Sort also depends on the interval length, which is often chosen by Knuth's increments:

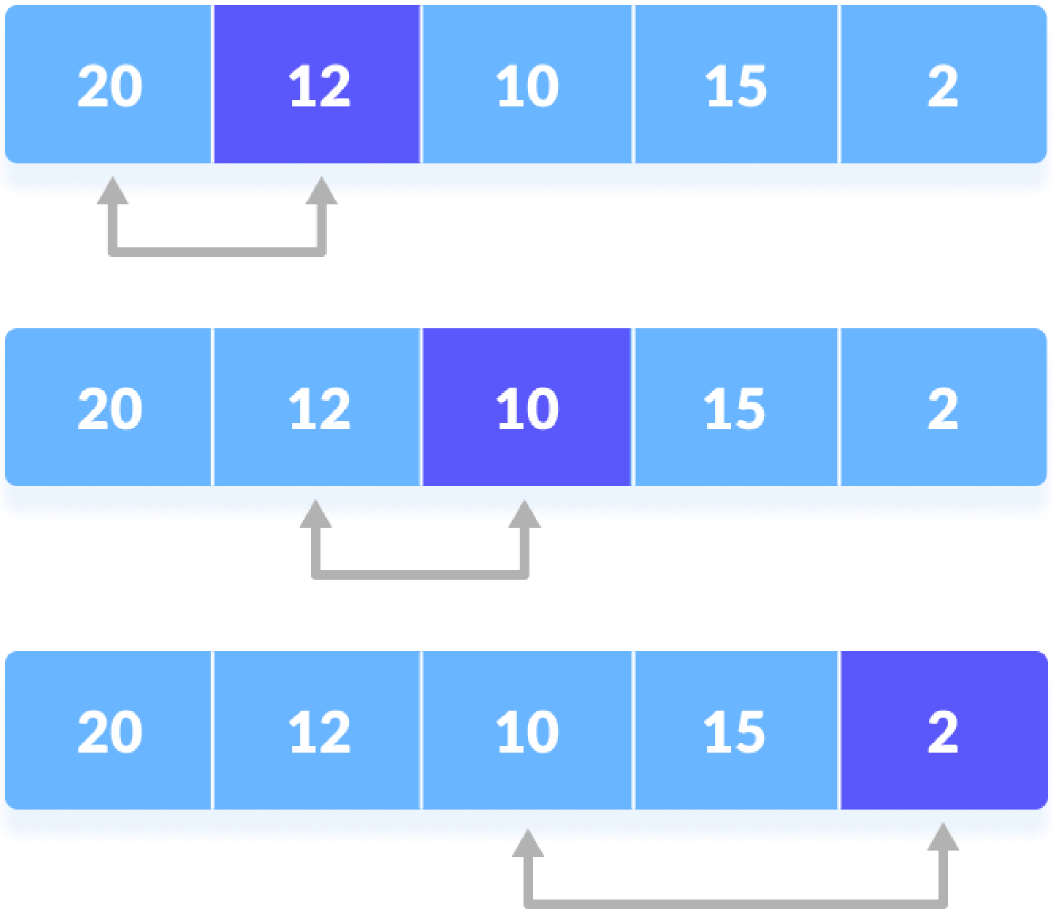
$$1, 4, 13, \ldots, (3k - 1) / 2$$

Selection Sort

Selection Sort

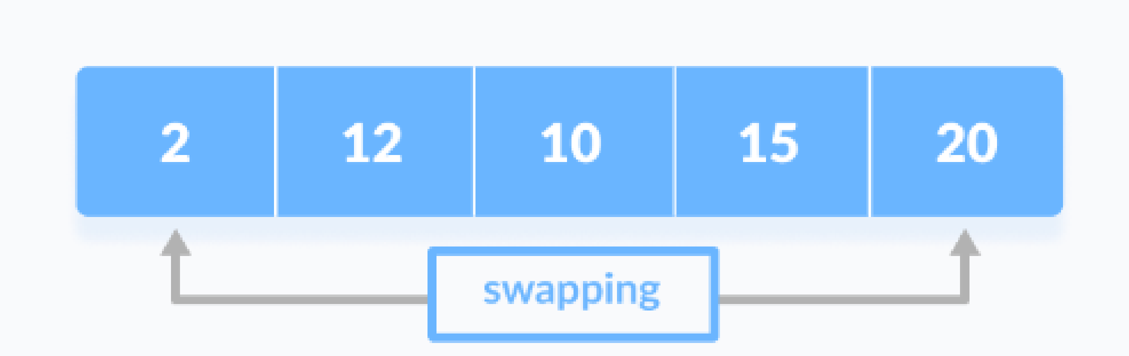
- Sort naturally the same as in real-life:
  - The list is divided into two sub-lists, sorted and unsorted, which are divided by an imaginary wall.
  - Find the **smallest element** from the **unsorted sub-list** and move to the correct position (swap it with the element at the beginning of the unsorted data.)
  - After each selection and swapping, increase the number of sorted elements and decrease the number of unsorted ones.
  - Loop those steps until the unsorted list has only 1 element.

- Step 1: find the minimum value of the list

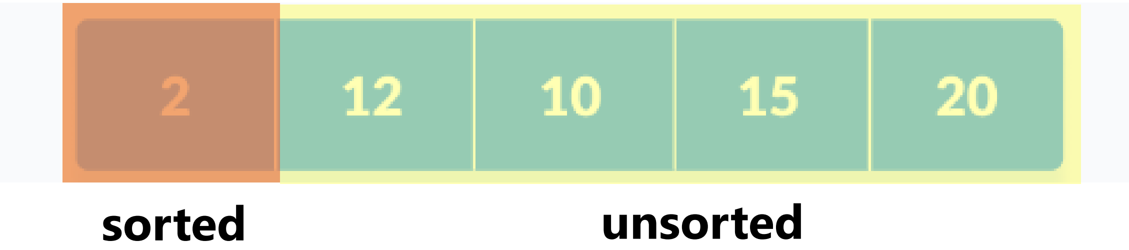


Selection Sort

- Step 2: min val is placed in the front of the unsorted list



- Step 3: repeatedly step 1-2 for the unsorted parts



Selection Sort

i = 0	15	2	8	7	3	6	9	17
i = 1	2	15	8	7	3	6	9	17
i = 2	2	3	8	7	15	6	9	17
i = 3	2	3	6	7	15	8	9	17
i = 4	2	3	6	7	15	8	9	17
i = 5	2	3	6	7	8	15	9	17
i = 6	2	3	6	7	8	9	15	17
i = 7	2	3	6	7	8	9	15	17

Selection Sort

- Which operation should be used for analysis?
- How many operations are there with size of the problem n?
- Best case? Worst case?

Selection Sort

```
SELECTION-SORT(A,n)
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]
            min ← j
    swap A[i] and A[min]
```

Basic operation

- In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).
- To analyze a sorting algorithm, we should count the number of key comparisons and the number of moves.
- Ignoring other operations does not affect our result.
- The outer for loop executes n-1 times. We invoke swap function once at each iteration.
  - Total Swaps: n-1
  - Total Moves: 3*(n-1) (Each swap has three moves)

- The inner for loop executes the size of the unsorted part minus 1 (from 1 to n-1), and in each iteration we make one key comparison.

Number of key comparisons = 1+2+...+n-1 = $\frac{n*(n-1)}{2}$

- The best case, the worst case, and the average case of the selection sort algorithm are same.

- Order of the algorithm: **O(n^2)**

Selection Sort

- If sorting a very large array, selection sort algorithm probably too inefficient to use.

- **What is the advantage of this algorithm?**

Selection Sort

- If sorting a very large array, selection sort algorithm probably too inefficient to use.

- **What is the advantage of this algorithm?**
  - Since selection sort is an in-place sorting algorithm, it does not require additional storage
  - …

Selection Sort

- The behavior of the selection sort algorithm does not depend on the initial organization of data.
- Although the selection sort algorithm requires $O(n^2)$ key comparisons, it only requires $O(n)$ moves.
- A selection sort could be a good choice if data moves are costly but key comparisons are not costly (short keys, long records).

Heap Sort

- Definition (array-based representation):
  - Heap is a collection of n elements (a_0, a_1, … a_{n-1}) in which every element (at position i) in the first half is greater than or equal to the elements at position 2i+1 and 2i+2.
  - (if $2i+2 \geq n$, just $a_i \geq a_{2i+1}$ satisfied).
- i.e., for every i ($0 \leq i \leq n/2 - 1$)

$$\mathbf{a_i \geq a_{2i+1} \text{ and } a_i \geq a_{2i+2}}$$

- Heap in above definition is called **max-heap.** (We also have min-heap structure).

Heap Structure

- Examples:
  - A max-heap: 9, 5, 6, 4, 5, 2, 3, 3
  - A min-heap: 8, 15, 10, 20, 17, 12, 18, 21, 20
- Give some more examples of:
  - A max-heap with 11 elements
  - A min-heap with 7 elements
- Where is the largest element in max-heap?

- **Property:**
  - The **first element** of the max-heap is always the largest.

Heap Construction

- Input: An array `a[]`, n elements
- Output: A heap `a[]`, n elements

```
Step 1.  Start from the middle of the array
         Initialize index = n/2 - 1
Step 2. while (index >= 0){
   heapRebuild at position index
             //heapRebuild(index, a, n)
   index = index - 1
 }
```

heapRebuild (pos, A, n)

- Step 1. Initialize k = pos, v = A[k], isHeap = false
- Step 2.

```
while !isHeap and 2*k+1 < n do
    j = 2*k + 1 //first element
    if j < n - 1 //has enough 2 elements
        if A[j] < A[j + 1] then j = j + 1
        //position of the larger between A[2*k+1] and A[2*k+2]
    if A[k] ≥ A[j] then isHeap = true
    else
        swap between A[k] and A[j]
    k = j
```

Heap Construction - An Example

- Construct a heap from the following list:

2, 9, 7, 6, 5, 8

Heap Construction - An Example

- Construct a heap from the following list:

5, 8, 1, 9, 0, 2, 6, 3, 4

5 - 8 – 6 – 9 – 0 – 2 – 1 – 3 – 4 (a3 → x, a2 → a6)

5 – 9 – 6 – 8 – 0 – 2 – 1 – 3 – 4 (a1 → a3, a3 → x)

9 – 5 – 6 – 8 – 0 – 2 – 1 – 3 – 4 (a0 → a1)

9 – 8 – 6 – 5 – 0 – 2 – 1 – 3 – 4 (a1 → a3, a3 → x)

- An interesting sorting algorithm discovered by J.W.J. Williams (in 1964).
- Idea is same as Selection Sort.
- It has two stages:
    - Stage 1: (**heap construction**). Construct a heap for a given array.
    - Stage 2: (**maximum deletion**). Apply the maximum key deletion n-1 times to the remaining heap
        - Exchange the first and the last element of the heap.
        - Decrease the heap size by 1.
        - Rebuild the heap at the first position.

```
HeapSort(a[], n)
{
   heapConstruct(a, n);
   r = n - 1;
   while (r > 0)
   {
      swap(a[0], a[r]);
      heapRebuild(0, a, r); //heapConstruct(a, r);
      r = r - 1;
   }
}
```

Heap Sort

- An interesting sorting algorithm discovered by J.W.J. Williams (in 1964).
- Idea is same as Selection Sort.
- It has two stages:
  - Stage 1: (**heap construction**). Construct a heap for a given array.
  - Stage 2: (**maximum deletion**). Apply the maximum key deletion n-1 times to the remaining heap
    - Exchange the first and the last element of the heap.
    - Decrease the heap size by 1.
    - Rebuild the heap at the first position.

Heap Sort

Illustrate heap sort to get 3rd max element in the below array:

5, 8, 1, 9, 0, 2, 6, 3, 4

After heap construction, we have max heap: 9 – 8 – 6 – 5 – 0 – 2 – 1 – 3 – 4

4 – 8 – 6 – 5 – 0 – 2 – 1 – 3 – 9 (a0 → a8)

8 – 4 – 6 – 5 – 0 – 2 – 1 – 3 – 9 (a0 → a1)

8 – 5 – 6 – 4 – 0 – 2 – 1 – 3 – 9 (a1 → a3, a3 → x)

3 – 5 – 6 – 4 – 0 – 2 – 1 – 8 – 9 (a0 → a7)

6 – 5 – **3** – 4 – 0 – **2 – 1** – 8 – 9 (a0 → a2, a2 → x)

1 – 5 – **3** – 4 – 0 – **2 – 6** – 8 – 9 (a0 → a6)

5 – 1 – **3** – 4 – 0 – **2 – 6** – 8 – 9 (a0 → a1)

5 – 4 – **3** – 1 – 0 – **2 – 6** – 8 – 9 (a1 → a3)

2 – 4 – **3** – 1 – 0 – **5 – 6** – 8 – 9 (a0 → a5)

Heap Sort

- Best case, Worst case, Average case are the same.
- The order of this algorithm: $O(n\log_2 n)$

1. The heap construction stage: $\boldsymbol{O(n)}$
2. The second stage: the number of key comparisons $\boldsymbol{C(n)}$ needed for eliminating the root keys from the heaps of diminishing sizes from n to 2, we have:

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2\sum_{i=1}^{n-1} \log_2 i$$

$$\le 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n\log_2 n. \in O(n\log_2 n)$$

3. For both stages, we get: $O(n) + O(n\log_2 n) = \boldsymbol{O(n\log_2 n)}$

Heap Sort

- Advantages:
  - The algorithm is also highly consistent with very low memory usage.
  - In-place sort: does not require extra storage
- Disadvantages:
  - unstable
  - expensive
  - not very efficient when working with highly complex data.

Exercise

- Utilize Heap Sort to arrange the following array

4, 1, 3, 14, 16, 9, 10

THANK YOU

for YOUR ATTENTION