Algorithm and Data structure

ا.سعاد محمد عثمان

Definition of algorithms

"Algorithm" refers to a series of specific and logical steps followed to solve a problem or accomplish a particular task within computer programming. These algorithms are used in software development to solve a variety of problems such as sorting data, searching, analysis, and more."

The efficiency and effectiveness of a program depend on the quality of the algorithm used. There's a diverse range of algorithms that vary in complexity and performance. Some famous examples of algorithms include:

- **Search Algorithms:** Such as binary search and linear search.
- **Sorting Algorithms:** Like Insertion Sort, Merge Sort, and Quick Sort.
- **Graph Algorithms:** Such as Depth-First Search and Breadth-First Search.
- **Artificial Intelligence Algorithms:** Including machine learning algorithms and artificial neural networks.

importance of algorithms

1. **Problem-solving:** Algorithmic programming enables solving a variety of complex problems and challenges in various fields such as mathematics, computer science, and engineering.
2. **Increased efficiency:**By using efficient algorithms, software performance can be improved, and resource consumption such as time and memory can be reduced, leading to more efficient and faster programs.
3. **Basis for development:**Algorithmic programming forms the basis for developing new technologies and innovations, such as artificial intelligence applications and machine learning.

Methods of evaluating algorithms

1. **Time Complexity**: This criterion evaluates the amount of time the algorithm needs to complete its task in worst-case, best-case, and average-case scenarios. This is typically determined by reviewing the number of steps the algorithm takes to complete the task based on the input data size.
2. **Space Complexity:**This measure assesses the amount of memory the algorithm requires to perform its task. It measures the temporary memory used by the algorithm during its execution.
3. **Accuracy and Correctness:**This criterion assesses the algorithm's ability to provide accurate and precise results for various scenarios.

Methods of evaluating algorithms

1. **Scalability:** It evaluates the algorithm's ability to handle increasing datasets without negatively impacting its performance.
2. **Mathematical Analysis:**Mathematical Analysis: Mathematics is used to analyze algorithms and estimate their performance theoretically by studying worst-case, best-case, and average-case scenarios.
3. **Empirical Testing:**This type of evaluation involves executing the algorithm on various actual datasets and measuring its performance under real-world conditions.

Time complexity

Time complexity is a term used to evaluate and measure the efficiency of an algorithm in terms of the time it takes to complete its task. It aims to analyze how the execution time of an algorithm grows with respect to the size of the input data.

The time complexity of an algorithm is usually expressed using Big O notation, which provides an upper bound on the growth rate of the algorithm in the worst-case scenario. It describes the relationship between the size of the input data (n) and the number of operations the algorithm performs.

For instance, if an algorithm has a time complexity of O(n), it means that the time taken to complete the algorithm's task grows linearly in direct proportion to the size of the input data. If the input data doubles, the time taken will also roughly double.

Time complexity

O(1) - Constant Time: Indicates that the algorithm's execution time remains constant regardless of the input size. This is the most efficient time complexity.
These algorithms are the most efficient, with execution time remaining constant regardless of the input data size. An example is direct access to an element in an array using its index.

O(log n) - Logarithmic Time: The execution time of these algorithms grows logarithmically as the input data size increases. An example is binary search, which divides the search space by half in each step.

O(n) - Linear Time: The execution time of these algorithms grows linearly with the input data size. An example is linear search, which sequentially searches each element.

O(n^2) - Quadratic Time: The execution time of these algorithms grows quadratically with the input data size and is common in algorithms that use nested loops, like bubble sort.

O(2^n) - Exponential Time: These algorithms have execution time that doubles with each additional input element, making them highly inefficient for large inputs.

important sorting algorithms

1. **Insertion Sort:** Divides the list into two parts: the sorted and unsorted parts, inserting elements from the unsorted part into the sorted part in an ordered manner.
2. **Quick Sort:** Among the most commonly used and efficient sorting algorithms, it employs a repeated partitioning technique and relies on quick partitioning and sorting to organize data.
3. **Merge Sort:** Relies on the merging principle, where it divides the list into sub-lists and then repeatedly merges them to obtain the sorted result.
4. **Heap Sort:** Depends on the data structure called a heap, using this structure to organize data before the sorting process.
5. **Selection Sort:** Chooses the smallest (or largest) element and swaps it with the first element, then the second element, continuing until the list is sorted.
6. **Bubble Sort:** Compares each element with the next element in the list and swaps them if they are in the wrong order. This process repeats until the list becomes sorted.

The problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example:

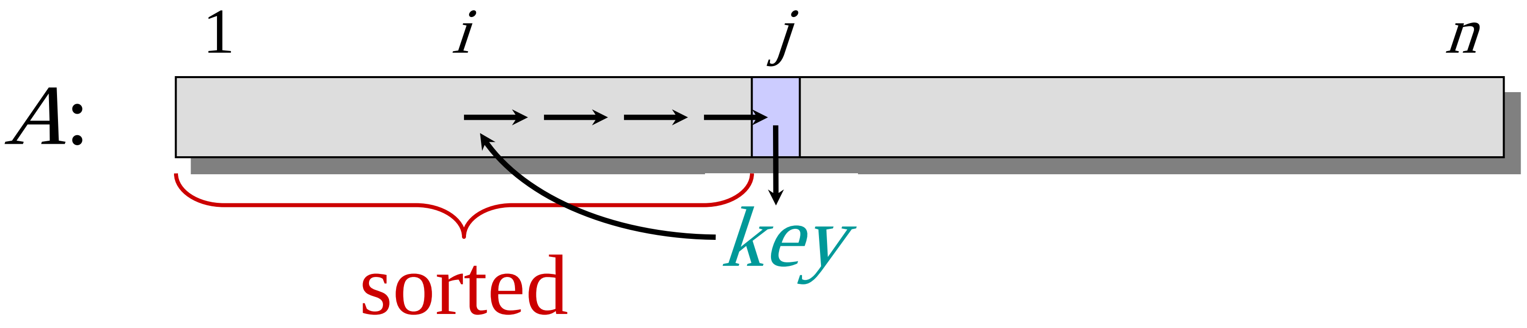
Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Insertion sort

“pseudocode”

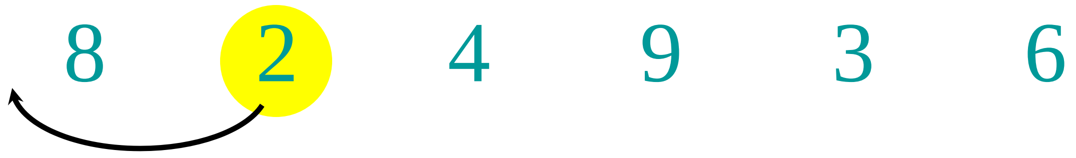
```
INSERTION-SORT $(A, n)$        ⊳ $A[1 \,.\,.\, n]$
    for $j \leftarrow 2$ to $n$
        do $key \leftarrow A[j]$
           $i \leftarrow j - 1$
           while $i > 0$ and $A[i] > key$
               do $A[i+1] \leftarrow A[i]$
                  $i \leftarrow i - 1$
           $A[i+1] = key$
```



Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

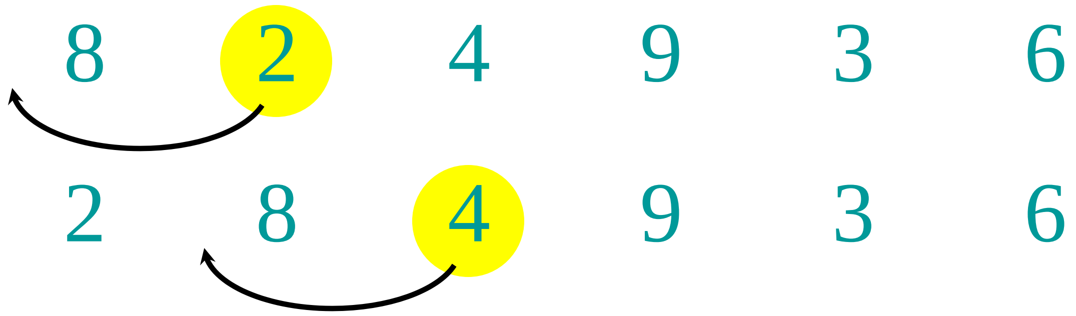


Example of insertion sort

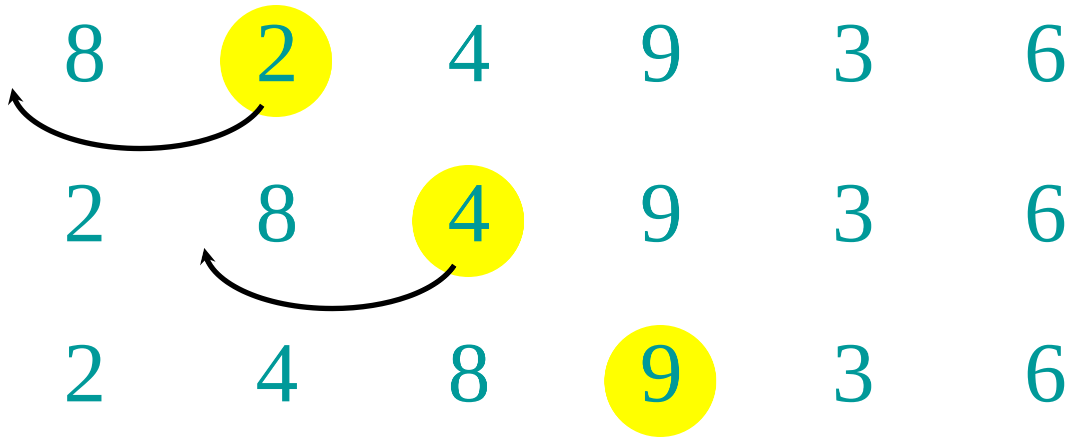
$$\begin{tikzcd} 8 & 2 \arrow[l, bend left] & 4 & 9 & 3 & 6 \end{tikzcd}$$

$$\begin{tikzcd} 2 & 8 & 4 & 9 & 3 & 6 \end{tikzcd}$$

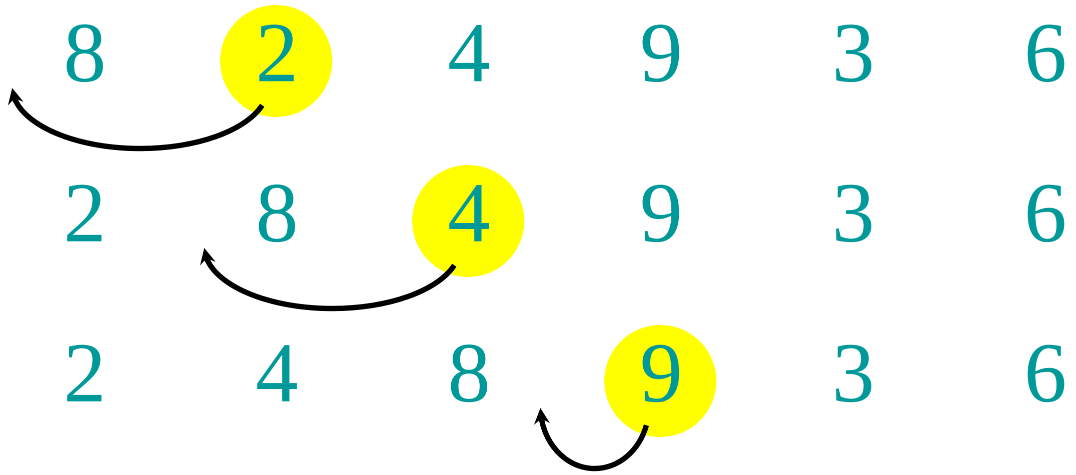
Example of insertion sort



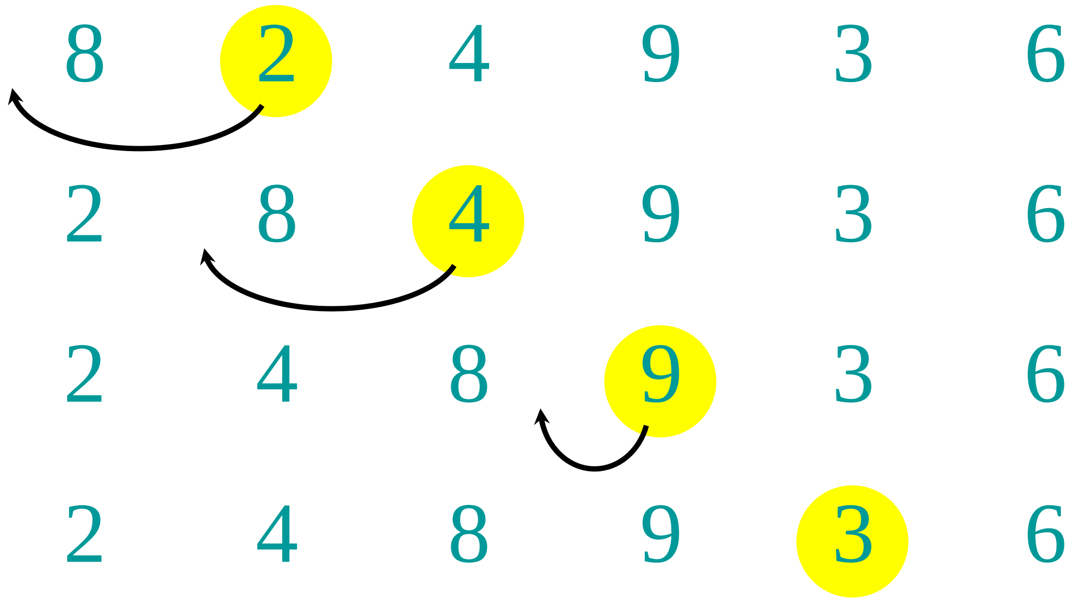
Example of insertion sort



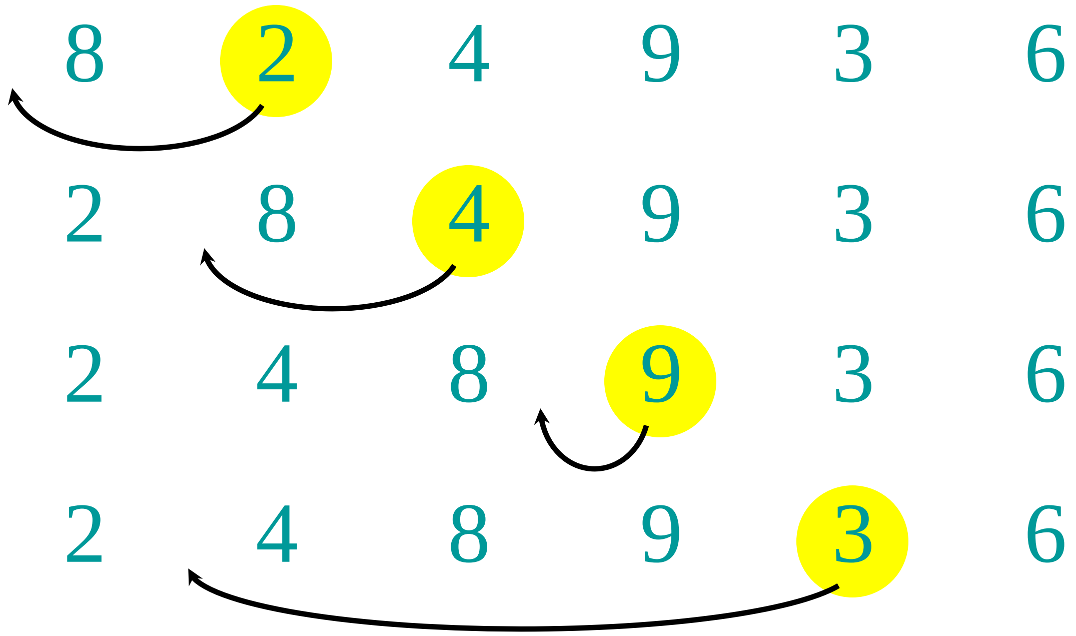
Example of insertion sort



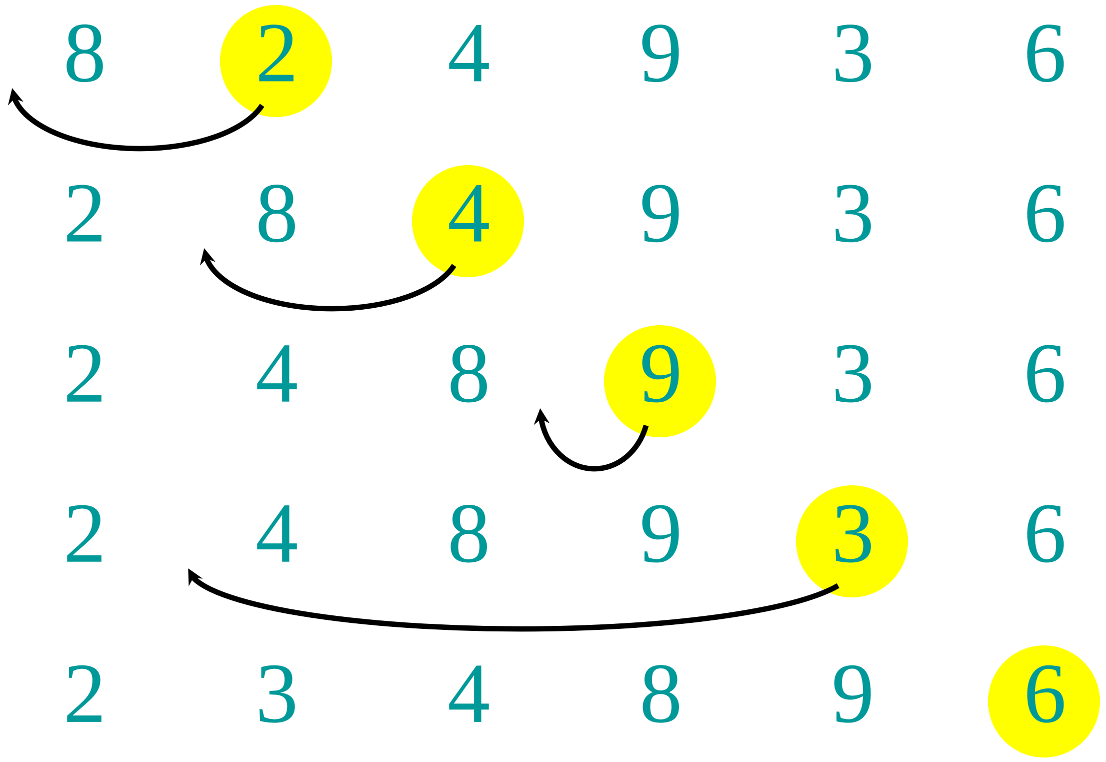
Example of insertion sort



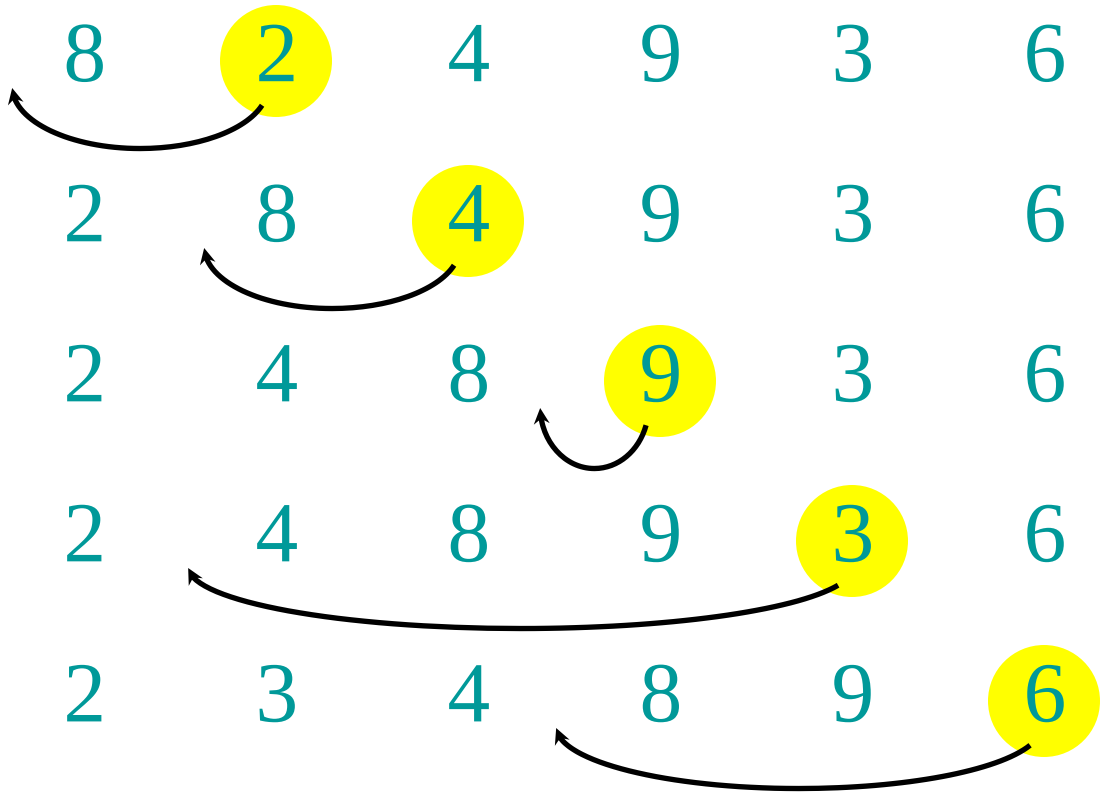
Example of insertion sort



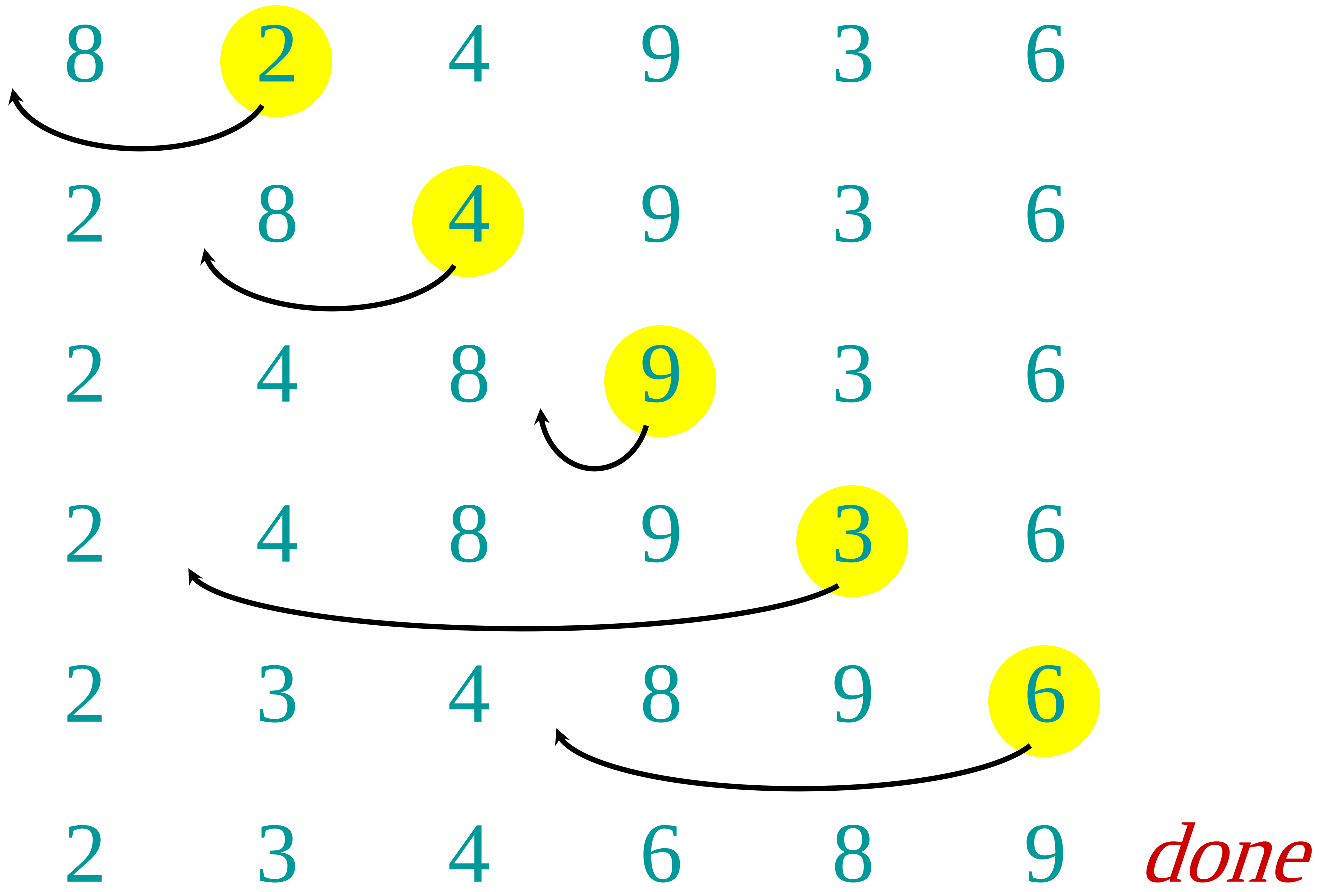
Example of insertion sort



Example of insertion sort



Example of insertion sort



Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Major Simplifying Convention: Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
  - $T_A(n) =$ time of A on length n inputs
- Generally, we seek upper bounds on the running time, to have a guarantee of performance.

Kinds of analyses

Worst-case: (usually)

- $T(n)$ = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (NEVER)

- Cheat with a slow algorithm that works fast on *some* input.

Machine-independent time

What is insertion sort's worst-case time?

BIG IDEAS:

- ***Ignore machine dependent constants,*** otherwise impossible to verify and to compare algorithms
- Look at ***growth*** of $T(n)$ as $n \to \infty$.

"Asymptotic Analysis"

Θ-notation

DEF:

$\Theta(g(n)) = \{ f(n) :$ there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ $\}$

Basic manipulations:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

Insertion sort analysis

Worst case: Input reverse sorted.

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta\left(n^2\right) \qquad \text{[arithmetic series]}$$

Average case: All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta\left(n^2\right)$$

Is insertion sort a fast sorting algorithm?

- Moderately so, for small n.
- Not at all, for large n.

Machine-independent time: An example

A *pseudocode* for insertion sort (INSERTION SORT).

```
INSERTION-SORT(A)
1   for $j \leftarrow 2$ to $length[A]$
2       do $key \leftarrow A[j]$
3       ∇ Insert A[j] into the sortted sequence A[1,..., j-1].
4        $i \leftarrow j - 1$
5        while $i > 0$ and $A[i] > key$
6            do $A[i+1] \leftarrow A[i]$
7                $i \leftarrow i - 1$
8     $A[i+1] \leftarrow key$
```

Analysis of INSERTION-SORT(contd.)

INSERTION - SORT(A)	cost	times
1 **for** $j \leftarrow 2$ **to** $length[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 ∇ Insert $A[j]$ into the sorted sequence $A[1 \cdot\cdot j-1]$	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ *and* $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

Analysis of INSERTION-SORT(contd.)

The total running time is

$$T(n) = c_1 + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1)$$
$$+ c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1).$$

Analysis of INSERTION-SORT(contd.)

The best case: The array is already sorted.
(t_j =1 for j=2,3, ...,n)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

Analysis of INSERTION-SORT(contd.)

- The worst case: The array is reverse sorted

  (t_j =j for j=2,3, ...,n).

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_5(n(n+1)/2 - 1)$$
$$+ c_6(n(n-1)/2) + c_7(n(n-1)/2) + c_8(n-1)$$

$$= (c_5/2 + c_6/2 + c_7/2)n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n$$

$$T(n) = an^2 + bn + c$$