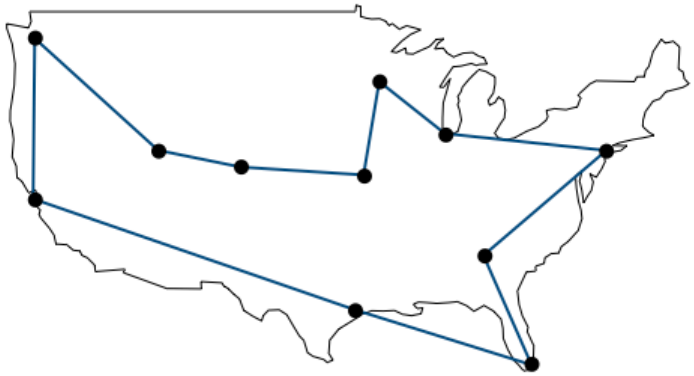
Classic problems

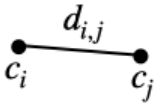
By Afshine Amidi and Shervine Amidi

Traveling salesman

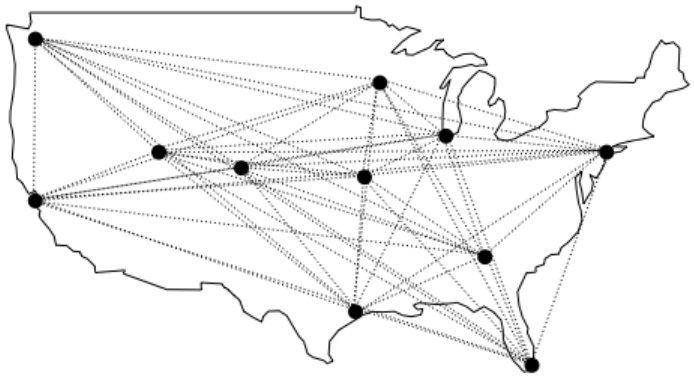
Given n cities $c_1, ..., c_n$, the traveling salesman problem (TSP) is a classic problem that aims at finding the shortest path that visits all cities exactly once and then returns to the starting city.



The distance between each pair of cities (c_i, c_j) is noted $d_{i,j}$ and is known.



A naive approach would consist of enumerating all possible solutions and finding the one that has the minimum cumulative distance. Given a starting city, there are $(n-1)!$ possible paths, so this algorithm would take $\mathcal{O}(n!)$ time. This approach is impracticable even for small values of n.



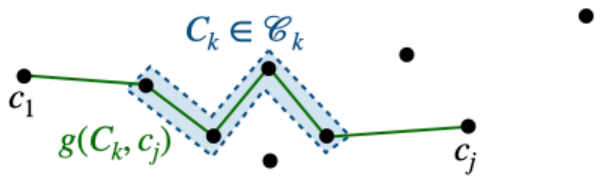
Luckily, the Held-Karp algorithm provides a bottom-up dynamic programming approach that has a time complexity of $\mathcal{O}(n^2 2^n)$ and a space complexity of $\mathcal{O}(n2^n)$.

Suppose c_1 is the starting city. This arbitrary choice does not influence the final result since the resulting path is a cycle. We define the following quantities:

- $\mathscr{C}_k$ contains all sets of k distinct cities in $\{c_2, ..., c_n\}$:

  $$\text{for } k \in [\![0, n-1]\!], \qquad \mathscr{C}_k = \left\{ C_k | C_k \subseteq \{c_2, ..., c_n\}, \#C_k = k \right\}$$

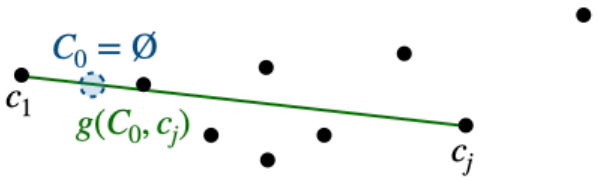
  We note that $\mathscr{C}_k$ has a size of $\binom{n-1}{k}$.
- $g(C_k, c_j)$ is the distance of the shortest path that starts from c_1, goes through each city in $C_k \in \mathscr{C}_k$ exactly once and ends at $c_j \notin C_k$.



The solution is found iteratively:

- *Initialization:* The shortest path between the starting city c_1 and each of the other cities c_j with no intermediary city is directly given by $d_{1,j}$:

  $$\forall j \in [\![2, n]\!], \quad \boxed{g(C_0, c_j) = d_{1,j}}$$

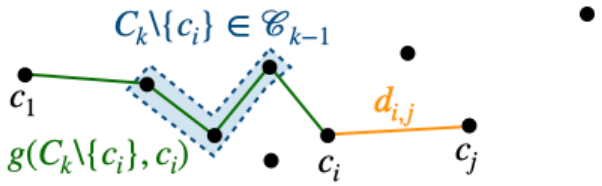


- *Compute step:* Suppose that for all $C_{k-1} \in \mathscr{C}_{k-1}$ and $c_i \notin C_{k-1}$, we know the distance of the shortest path $g(C_{k-1}, c_i)$ between c_1 and c_i via the $k-1$ cities in C_{k-1}.

  Let's take $C_k \in \mathscr{C}_k$. For all $c_i \in C_k$, we note that $C_k \backslash \{c_i\} \in \mathscr{C}_{k-1}$ and that (naturally) $c_i \notin C_k \backslash \{c_i\}$, which means that $g(C_k \backslash \{c_i\}, c_i)$ is known.

We can deduce the distance of the shortest path between c_1 and $c_j \notin C_k$ via k cities by taking the minimum value over all second-to-last cities $c_i \in C_k$:

$$\forall C_k \in \mathscr{C}_k, \forall c_j \notin C_k, \quad \boxed{g(C_k, c_j) = \min_{c_i \in C_k} \Big\{ g(C_k \backslash \{c_i\}, c_i) + d_{i,j} \Big\}}$$



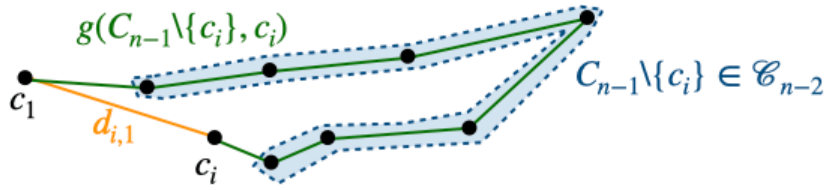
Each step k has the following complexities:

Time	Space
$k \times (n-1-k) \times \binom{n-1}{k}$	$(n-1-k) \times \binom{n-1}{k}$

- *Final step:* The solution is given by $g(\{c_2, ..., c_n\}, c_1)$.

  Since $C_{n-1} = \{c_2, ..., c_n\}$, the last step of the algorithm gives:

$$\boxed{g(C_{n-1}, c_1) = \min_{i \in [\![2,n]\!]} \Big\{ g(C_{n-1} \backslash \{c_i\}, c_i) + d_{i,1} \Big\}}$$



Knapsack

The 0/1 knapsack problem is a classic problem where the goal is to maximize the sum of values of items put in a bag that has a weight limit. Here, "0/1" means that for each item, we want to know whether we should include it (1) or not (0).

More formally, we have a bag of capacity C and n items where each item $i \in [\![1, n]\!]$ has value v_i and weight w_i. We want to find a subset of items $\mathscr{I} \subseteq \{1, ..., n\}$ such that:

$$\boxed{\mathscr{I} = \underset{I \subseteq [\![1,n]\!]}{\operatorname{argmax}} \sum_{i \in I} v_i} \quad \text{with} \sum_{i \in \mathscr{I}} w_i \leqslant C$$

A naive approach would consist of trying out every combination of the n items and take the one with the maximum value which also satisfies the cumulative weight constraint. Such an approach has a time complexity of $\mathcal{O}(2^n)$.

Luckily, this problem can be solved with a bottom-up dynamic programming approach that has a time complexity of $\mathcal{O}(nC)$ and a space complexity of $\mathcal{O}(nC)$.

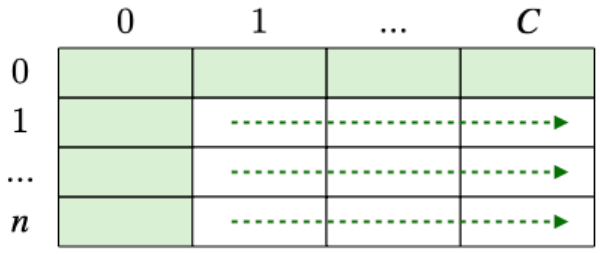
We note $V_{i,j}$ the maximum value of a bag of capacity $j \in [\![0, C]\!]$ that contains items among $\{1, 2, ..., i\}$. Our objective is to get $V_{n,C}$.

	0	...	j	...	C
0					
...					
i			$V_{i,j}$		
...					
n					$V_{n,C}$

- *Initialization:* The maximum values of the following bags are known:
  - Bags of capacity 0: $V_{i,0} = 0$.
  - Bags with 0 item: $V_{0,j} = 0$.

	0	1	...	C
0	0	0	0	0
1	0			
...	0			
n	0			

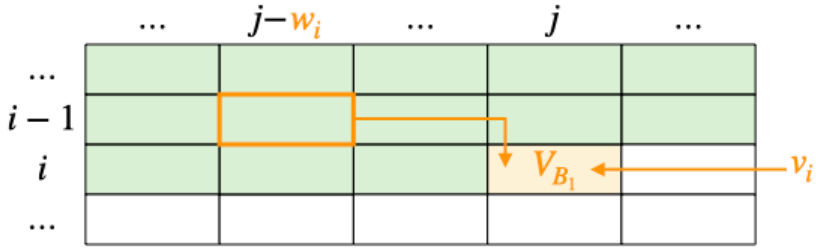
- *Compute maximum value:* Starting from $i = 1$ and $j = 1$, we iteratively fill the 2D array of maximum values from left to right, top to bottom.



In order to determine the maximum value $V_{i,j}$ of a bag of capacity j containing items among $\{1, ..., i\}$, we need to choose between two hypothetical bags:

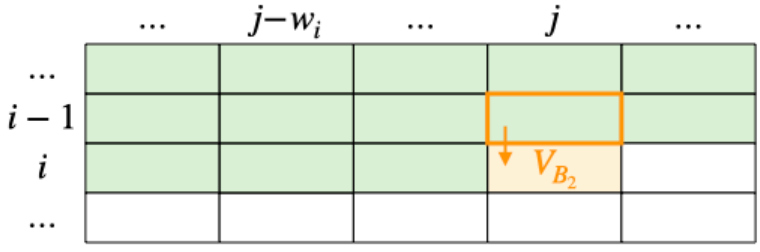
  - *Bag 1: contains item i.* V_{B_1} is found by adding the value v_i of item i to the maximum value of the bag of capacity $j - w_i$ containing items among $\{1, ..., i-1\}$.

$$\boxed{V_{B_1} = V_{i-1,j-w_i} + v_i}$$



  - *Bag 2: does not contain item i.* V_{B_2} is already known: it is the maximum value of the bag of capacity j containing items among $\{1, ..., i-1\}$.

$$\boxed{V_{B_2} = V_{i-1,j}}$$

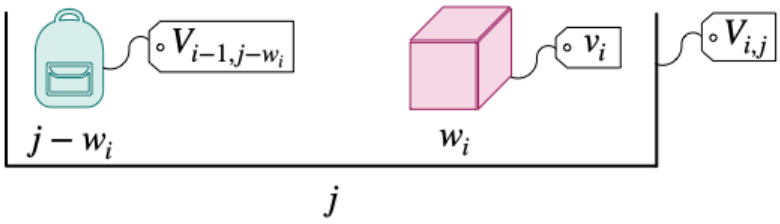


We choose the best potential bag:

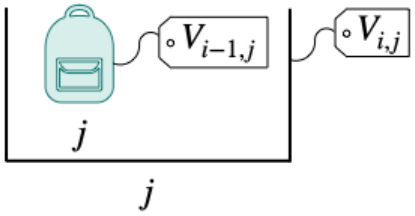
$$\boxed{V_{i,j} = \max(V_{B_1}, V_{B_2})}$$

When the 2D array is filled, the desired value is $V_{n,C}$.

- *Get final items:* In order to know which items were selected, we start from position $(i, j) = (n, C)$ of the 2D array and traverse it iteratively:
  - *Case $V_{i,j} \neq V_{i-1,j}$:* This means that item i was included. We go to position $(i - 1, j - w_i)$ and include item i in the set of included items.



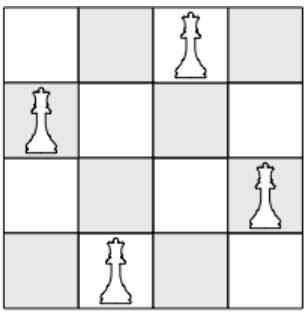
  - *Case $V_{i,j} = V_{i-1,j}$:* This means that item i was not included. We go to position $(i - 1, j)$.

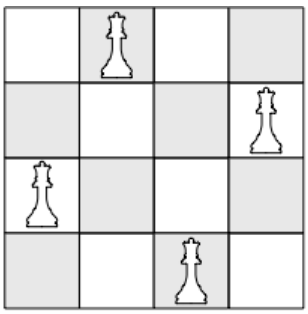


    We repeat this process until retrieving all items.

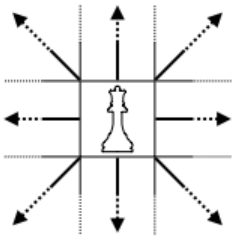
N-Queens

Given a chessboard of size $N \times N$, the N-Queens problem aims at finding a configuration of N queens such that no two queens attack each other.



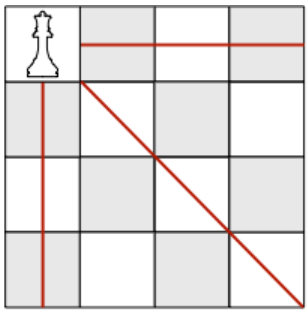


A queen is a chess piece that can move horizontally, vertically or diagonally by any number of steps.

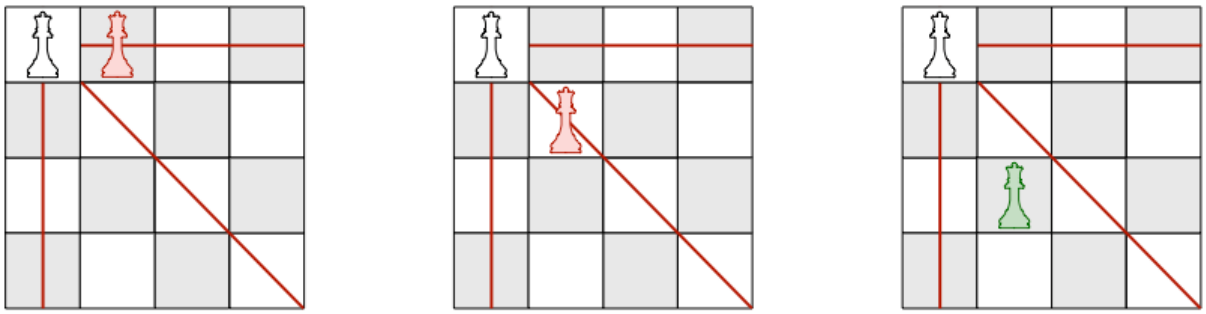


The solution can be found using a backtracking approach:

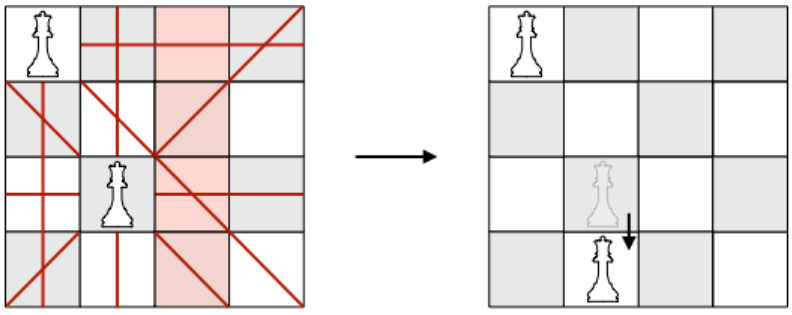
- *Step 1:* Put the 1^{st} queen on the chessboard.



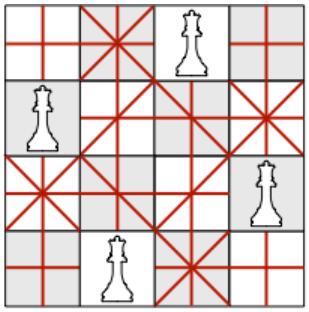
- *Step 2:* Put the 2^{nd} queen in the second column of the chessboard. As long as the partial configuration is invalid, keep changing its position until conditions are satisfied.



- *Step 3:* If there is nowhere to place the next queen, go back one step and try the next partial configuration.

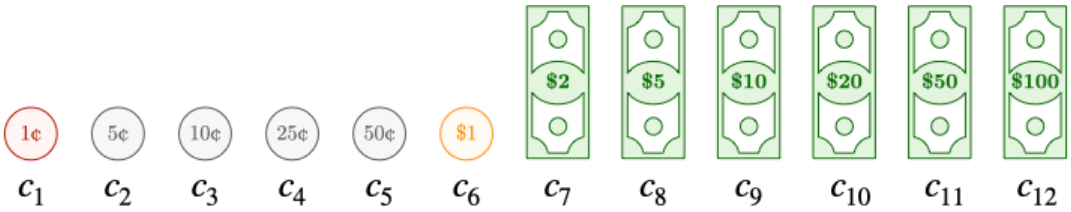


- *Step 4:* Continue this process until finding a valid solution.



Coin change

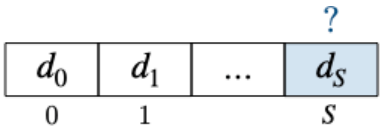
Given an unlimited number of coins of values $\{c_1, .., c_k\}$, the coin change problem aims at finding the minimum number of coins that sum up to a given amount S. By convention, we assume that $c_1 < ... < c_k$.



In other words, we want to find the minimum value of $x = x_1 + ... + x_k$ such that

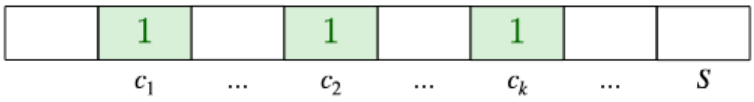
$$S = \sum_{i=1}^{k} x_i c_i$$

The bottom-up dynamic programming solution iteratively computes the minimum amount of coins d_s for each amount $s \in \{1, ..., S\}$.



This approach runs in $\mathcal{O}(S)$ time and takes $\mathcal{O}(S)$ space:

- *Initialization:*
  - Initialize array $D = [d_0, ..., d_S]$ with zeros.
  - For each $i \in [\![1, ..., k]\!]$, set d_{c_i} to 1 since we already know that the corresponding amount only needs 1 coin.

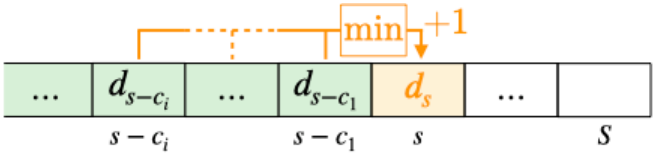


- *Compute step:* Starting from $s = c_1 + 1$, we iteratively fill array D.

In order to obtain amount s, we distinguish two cases:

- *Case* $\exists i, d_{s-c_i} > 0$: We look back at valid values d_{s-c_i} and see which coin c_i minimizes the total number of coins needed to obtain amount s.

$$\boxed{d_s = \min_{\substack{i \in [\![1,k]\!] \\ d_{s-c_i} > 0}} \left\{ d_{s-c_i} + 1 \right\}}$$



- *Case* $\forall i, d_{s-c_i} = 0$: We cannot obtain amount s using coins c_i.

  At the end of this step, amounts $s \in \{1, .., S\}$ with $d_s = 0$ cannot be obtained with the given coins.

The answer to this problem is given by d_S.

Sorting algorithms

By Afshine Amidi and Shervine Amidi

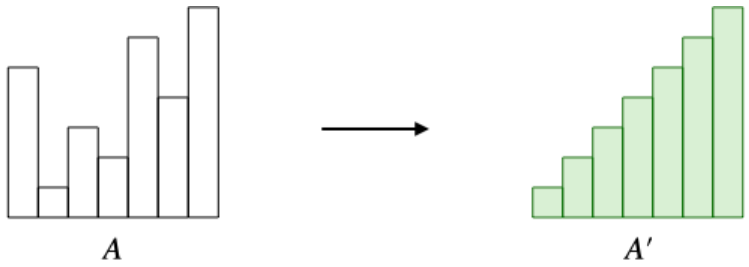
General concepts

In this part, arrays of n elements are visually represented as histograms. The height of each bar represents the value of the associated element in the array.

Sorting algorithm

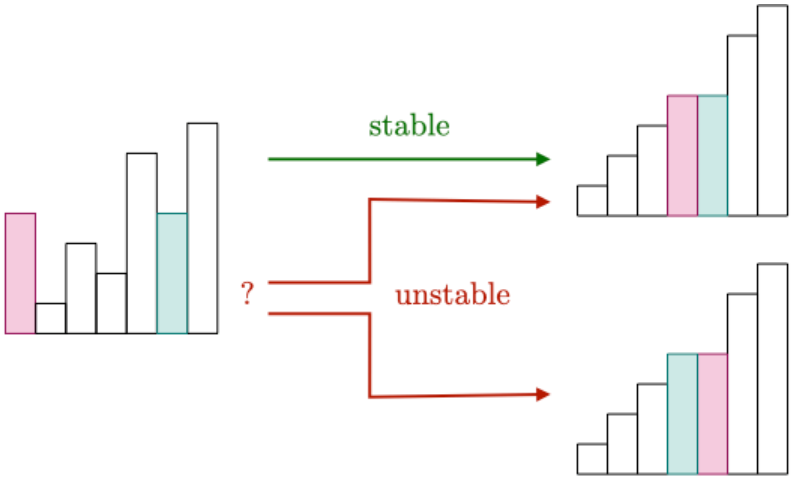
A sorting algorithm takes an unsorted array $A = [a_0, ..., a_{n-1}]$ as input and returns a sorted array $A' = [a'_0, ..., a'_{n-1}]$ as output. A' is a permutation of A such that:

$$\boxed{a'_0 \leqslant a'_1 \leqslant ... \leqslant a'_{n-1}}$$



Stability

A sorting algorithm is said to be stable if the order of tied elements is *guaranteed* to remain the same after sorting the array.



> **REMARK**
>
> Examples of stable sorting algorithms include merge sort and insertion sort.

The next sections will go through each sorting algorithm into more detail

Basic sort

Bubble sort

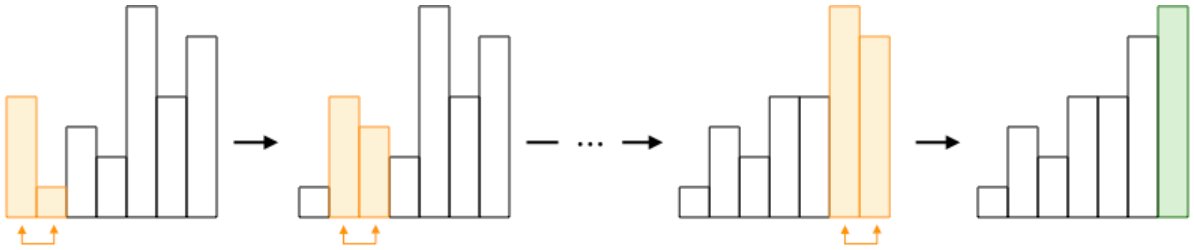
Bubble sort is a stable sorting algorithm that has a time complexity of $\mathcal{O}(n^2)$ and a space complexity of $\mathcal{O}(1)$.

Intuition Compare consecutive elements and swap them if they are not in the correct order.

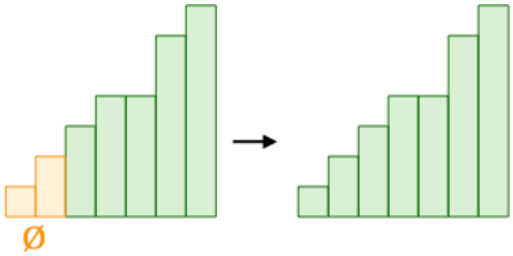
Algorithm

- *Compute step:* Starting from the beginning of the array, compare the element a_l at position i with the element a_r at position $i+1$.
    - *Case $a_l \leqslant a_r$:* They are already in the correct order. There is nothing to do.
    - *Case $a_l > a_r$:* They are not in the correct order. Swap them.

Repeat this process until reaching the end of the array.



- *Repeat step:* Repeat the *compute step* until no swap can be done.



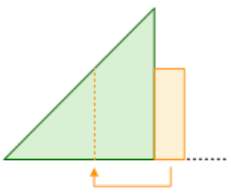
We note that:

- At the end of the k^{th} pass, the last k elements of the array are guaranteed to be in their final positions.
- The algorithm finishes in at most $n-1$ passes. In particular:
  - If the input array is already sorted, then the algorithm finishes in 1 pass.
  - If the input array is reverse sorted, then the algorithm finishes in $n-1$ passes.

Insertion sort

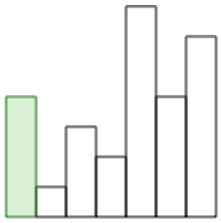
Insertion sort is a stable sorting algorithm that has a time complexity of $\mathcal{O}(n^2)$ and a space complexity of $\mathcal{O}(1)$.

Intuition Incrementally build a sorted subarray by successively inserting the next unsorted element at its correct position.



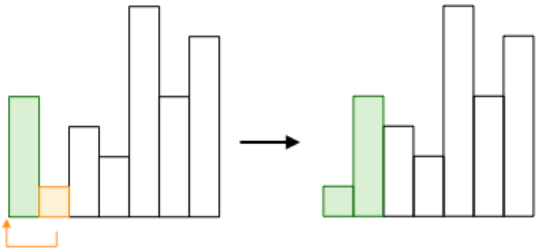
Algorithm

- *Initialization:* The first element of the unsorted array can be interpreted as a subarray of one element that is already sorted.

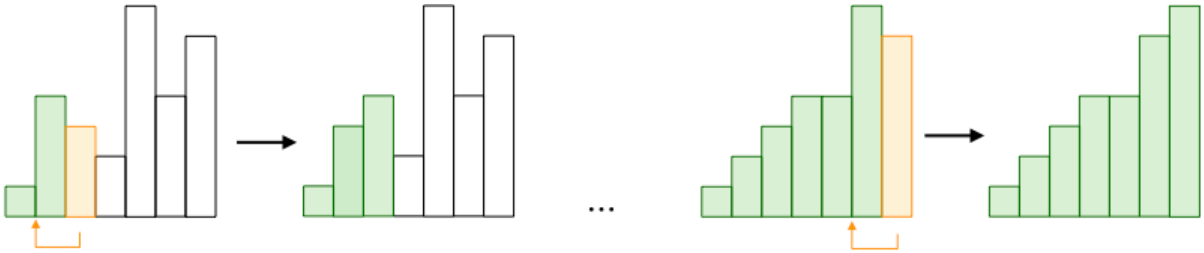


- *Compute step:* Starting from position $i = 1$, we want to insert element a_i into the current sorted subarray of size i. In order to do that, we compare a_i with its preceding element a_p:
  - As long as $a_p > a_i$, we iteratively swap a_p with a_i.
  - This process ends with either a_i verifying $a_p \leqslant a_i$ or a_i being at position 0 of the array.

    At the end of this step, the sorted subarray is of size $i + 1$.



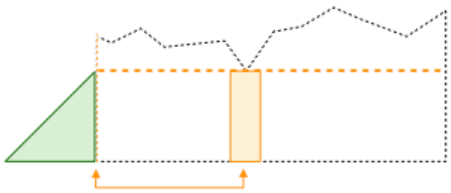
- *Repeat step:* Repeat the *compute step* until the sorted subarray reaches a size of n.



Selection sort

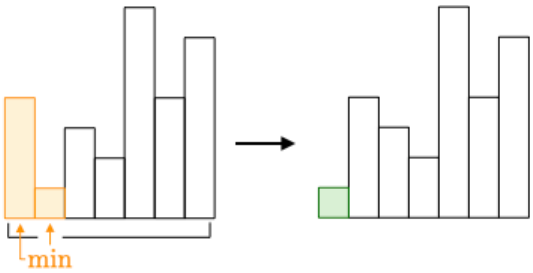
Selection sort is a stable sorting algorithm that has a time complexity of $\mathcal{O}(n^2)$ and a space complexity of $\mathcal{O}(1)$.

Intuition Incrementally build a sorted subarray by successively inserting the minimum value among the remaining elements.



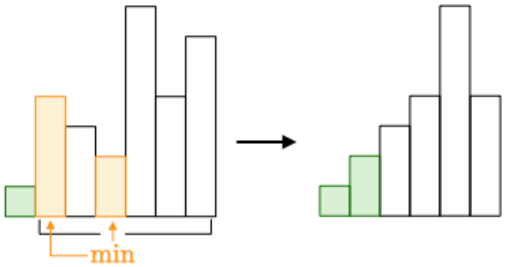
Algorithm

- *Initialization:*
  - Find the minimum value of the unsorted array.
  - Swap it with the element at the beginning of the array. The first element can now be interpreted as a sorted subarray with one element.

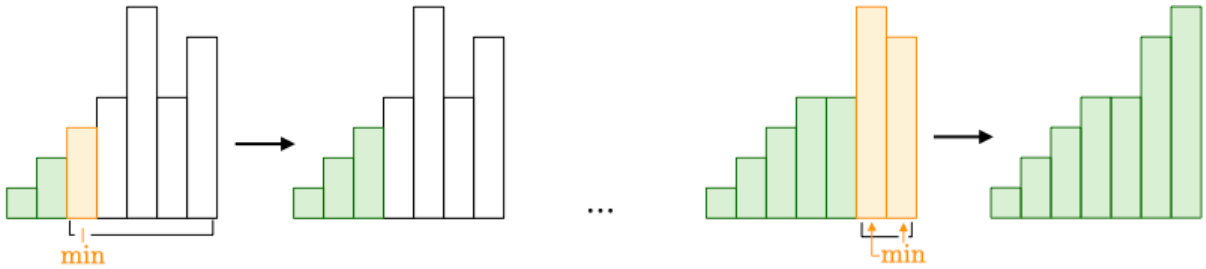


- *Compute step:* Starting from $i = 1$, we want to insert a new element into the sorted subarray of size i.
  - Find the minimum value $a_{\min}$ among the remaining elements at positions $i, \ldots, n-1$ of the array. By construction, $a_{\min}$ is greater or equal than all elements of the current sorted subarray.
  - Swap $a_{\min}$ with the element at position i of the array.

    At the end of this step, the sorted subarray is of size $i + 1$.



- *Repeat step:* Repeat the *compute step* until the sorted subarray reaches a size of n.



> **REMARK**
>
> Insertion and selection sort are very similar in that they build the sorted array from scratch and add elements one at a time.

Cycle sort

Cycle sort is an unstable sorting algorithm that has a time complexity of $\mathcal{O}(n^2)$ and a space complexity of $\mathcal{O}(1)$.

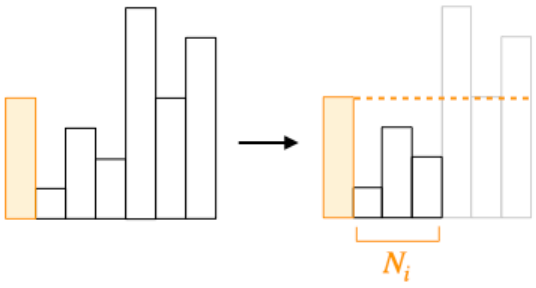
Intuition Determine the index of the final position of each element in the sorted array.

Algorithm

- *Compute step:* Starting from $i = 0$, we want to find the final position of element a_i along with those of the other elements impacted by this move.

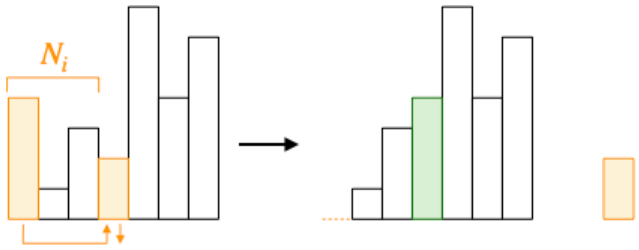
  To do so, we count the number of elements that are smaller than a_i:

$$\boxed{N_i = \#\{k, a_k < a_i\}}$$



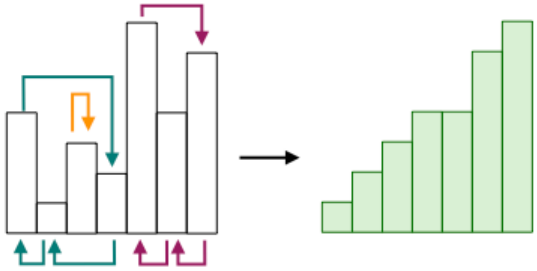
The final position of a_i is at index N_i, since we know that there are N_i smaller elements than a_i in the final sorted array.

- *Case $N_i = i$:* This is a self-cycle, meaning that the element is already at its correct position. There is nothing to do.
- *Case $N_i \neq i$:* This is the start of a cycle of moves. Place a_i at position N_i (or to the right of any duplicates, if applicable) and keep the replaced value in a temporary variable.



Keep moving elements using this logic until getting back to position i.

- *Repeat step:* Repeat the *compute step* until reaching the end of the array. We can see cycles being formed from the way elements are moved.



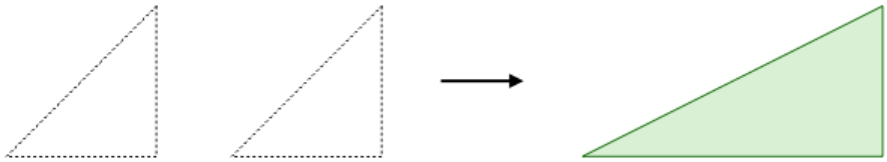
> **REMARK**
>
> This algorithm sorts the array with a minimum amount of rewrites since it only moves elements to their final positions.

Efficient sort

Merge sort

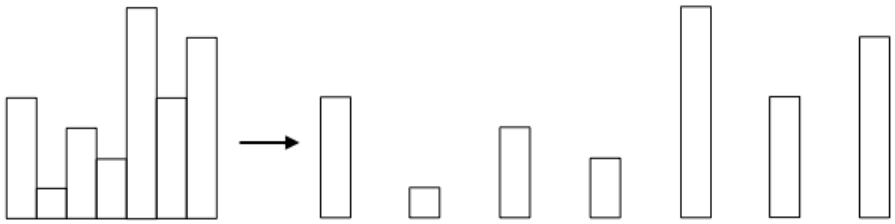
Merge sort is a stable sorting algorithm that has a time complexity of $\mathcal{O}(n \log(n))$ and a space complexity of $\mathcal{O}(n)$.

Intuition Build a sorted array by merging two already-sorted arrays.

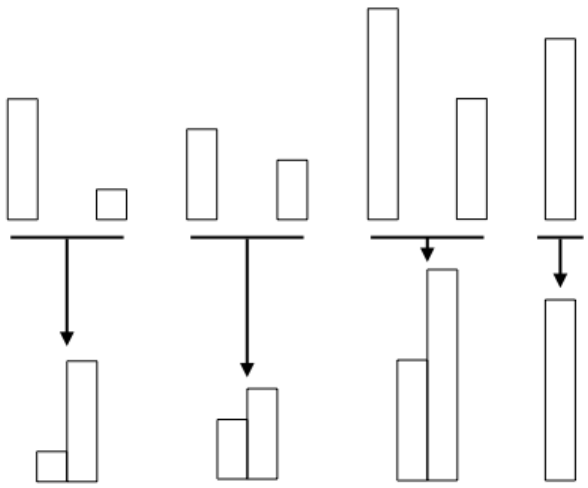


Algorithm

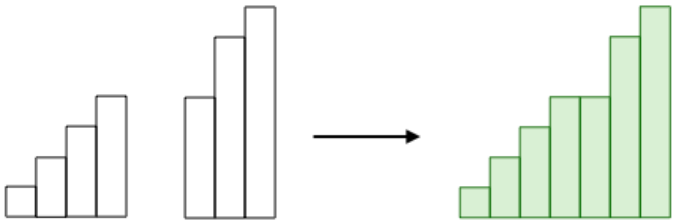
- *Divide step:* Divide the array into as many subarrays as there are elements. Each resulting subarray has only one element and can be considered as sorted.



- *Conquer step:* For each pair of sorted subarrays, build a sorted array by merging them using the two-pointer technique.



  Repeat the process until all subarrays are merged into one final sorted array.

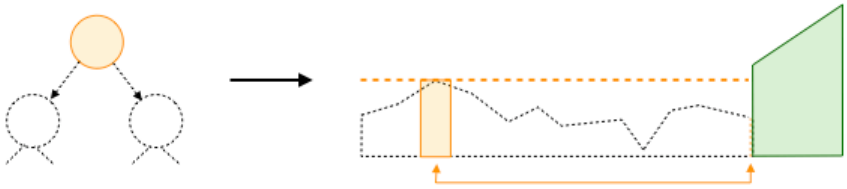


> **REMARK**
>
> This algorithm is usually implemented recursively.

Heap sort

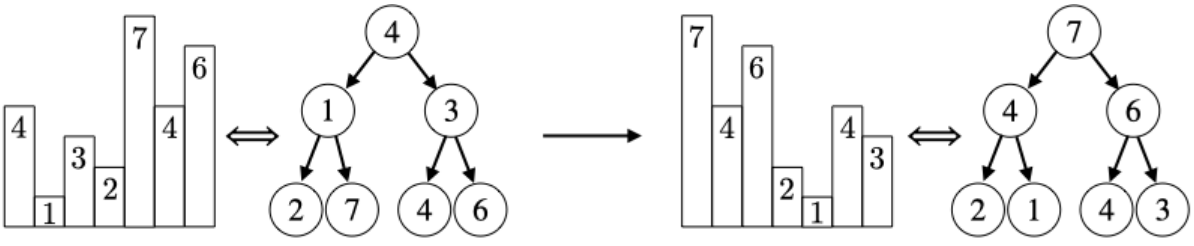
Heap sort is an unstable sorting algorithm that has a time complexity of $\mathcal{O}(n \log(n))$ and a space complexity of $\mathcal{O}(1)$.

Intuition Incrementally build a sorted subarray by successively retrieving the maximum of remaining elements using a max-heap.



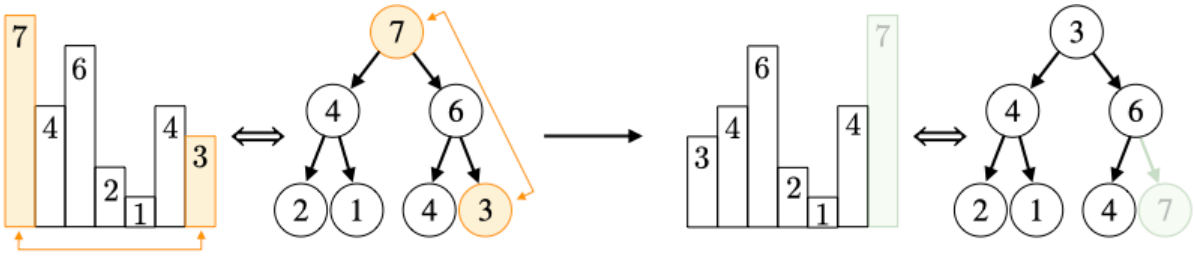
Algorithm

- *Initialization:* Build a max-heap from the unsorted array in $\mathcal{O}(n)$ time. This is done by recursively swapping each parent with their child of highest value.



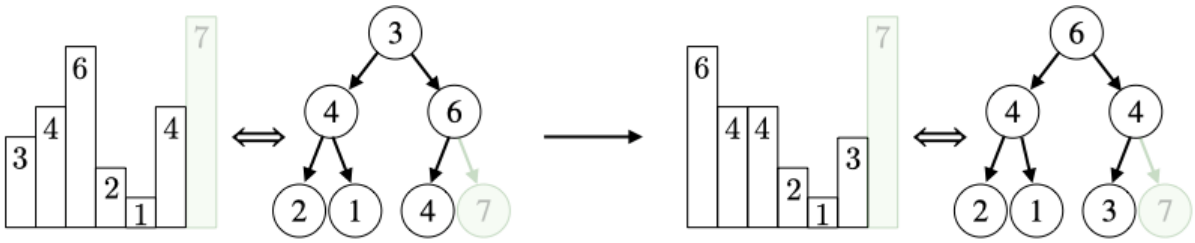
  We can use the same array to represent the max-heap.

- *Compute step:* Starting from $i = 0$, incrementally build a sorted subarray of size $i + 1$ that is placed at the end of the array.
  - Pop an element from the max-heap and place it at position $n - i - 1$ of the array.

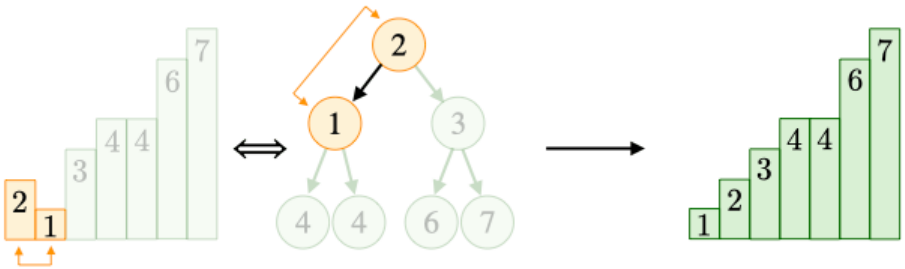


By construction, the popped element is greater or equal than the $n - i - 1$ elements of the heap and smaller or equal than the i previously popped elements.

- Heapify the resulting max-heap of size $n - i - 1$ in $\mathcal{O}(\log(n))$ time.



- *Repeat step:* Repeat the *compute step* until the subarray reaches a size of n.

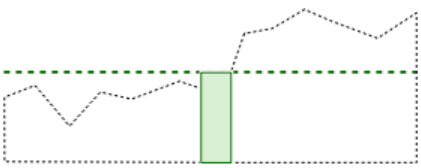


Quick sort

Quick sort is an unstable sorting algorithm that has a time complexity of $\mathcal{O}(n^2)$ and a space complexity of $\mathcal{O}(n)$.

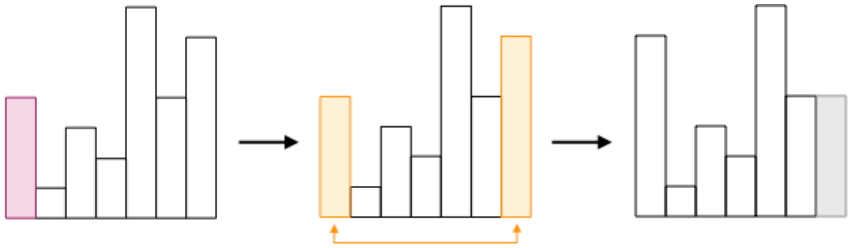
Intuition Recursively choose an element of the array to be the pivot, and put:

- Smaller elements to the left of the pivot.
- Bigger elements to the right of the pivot.



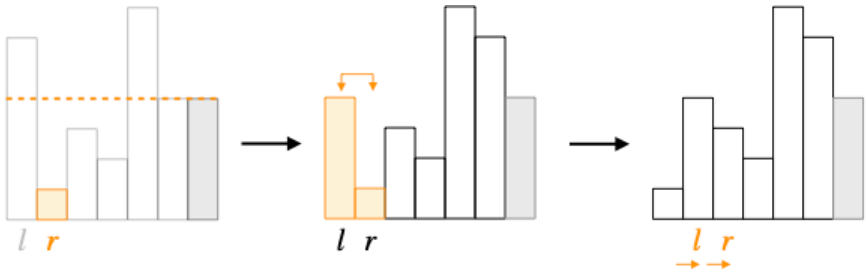
Algorithm

- *Compute step:*
  - *Pivot isolation step:* Choose an element of the array to be the pivot p and swap it with the last element of the array.

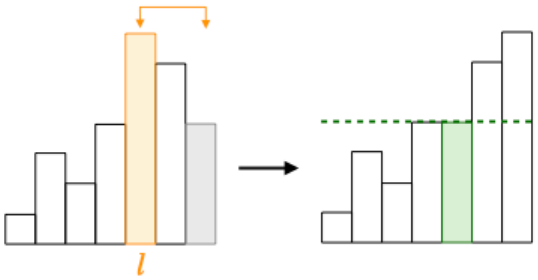


  - *Partition step:* Starting from indices $l = 0$ and $r = 0$, we make sure that elements a_r that are smaller or equal than the pivot p are sent to the left of the array:
    - *Case* $a_r \leqslant p$*:* Swap element a_r at index r with element a_l at index l. Increment l by 1.
    - *Case* $a_r > p$*:* There is nothing to do.

      This process is successively repeated for all indices r until the second-to-last position of the array.

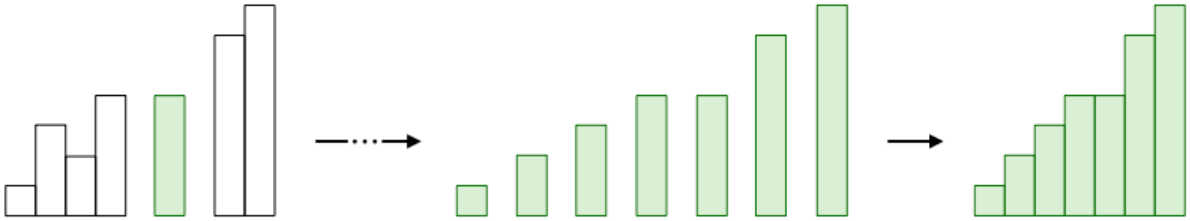


  - *Pivot inclusion step:* The final position of the left pointer l represents the position in the array where its left elements are all array elements that are smaller than or equal to the pivot. We swap the pivot with the element from position l.



At the end of this step, the pivot is at its correct and final position.

- *Recursion step:* The *compute step* is run recursively on the resulting subarrays on the left and right of the pivot. They are then merged back together to form the final sorted array.



We note that the runtime of the algorithm is sensitive to the choice of the pivot and the patterns found in the input array. In general, a time complexity of $\mathcal{O}(n\log(n))$ and a space complexity of $\mathcal{O}(\log(n))$ can be obtained when the pivot is a good approximation of the median of the array.

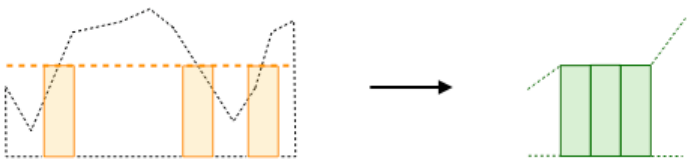
The two most common methods to choose the pivot are:

Special sort

Counting sort

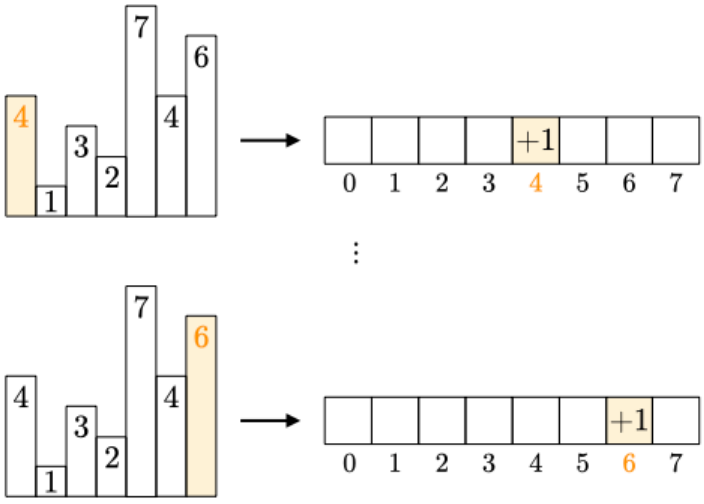
Counting sort is a stable sorting algorithm that is efficient for integer arrays with values within a small range $[\![0, k]\!]$. It has a time complexity of $\mathcal{O}(n+k)$ and a space complexity of $\mathcal{O}(n+k)$.

Intuition Determine the final position of each element by counting the number of times its associated value appears in the array.



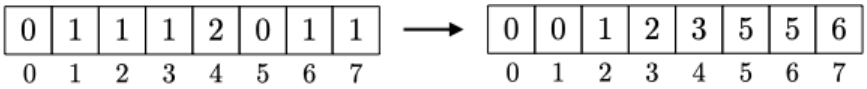
Algorithm

- *Number of occurrences:* This step takes $\mathcal{O}(n+k)$ time and $\mathcal{O}(k)$ space.
  - *Initialization:* Initialize an array C of length $k+1$.
  - *Count step:* Scan array A and increment the counter c_v of each encountered value $v \in \{0, ..., k\}$ by 1.



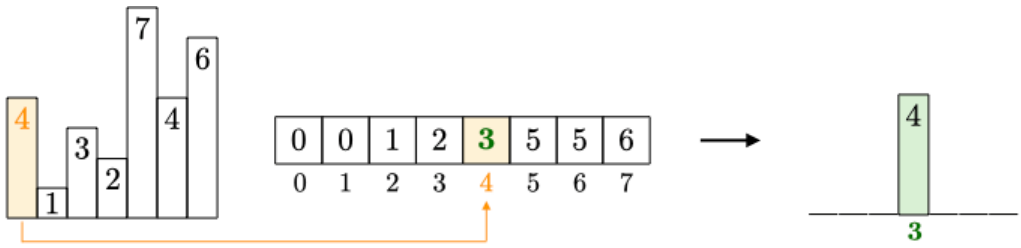
Each count c_v represents the number of times the value $v \in \{0, ..., k\}$ appears in array A.

- *Cumulative step:* Compute the cumulative sum of array $C = [c_0, .., c_k]$ and move each resulting element to the right. This operation is done in-place and takes $\mathcal{O}(k)$ time and $\mathcal{O}(1)$ space.

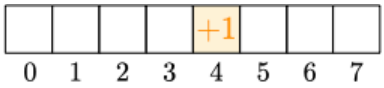


For each value v present in A, the associated element c_v in the resulting array C indicates its starting index i in the final sorted array A'.

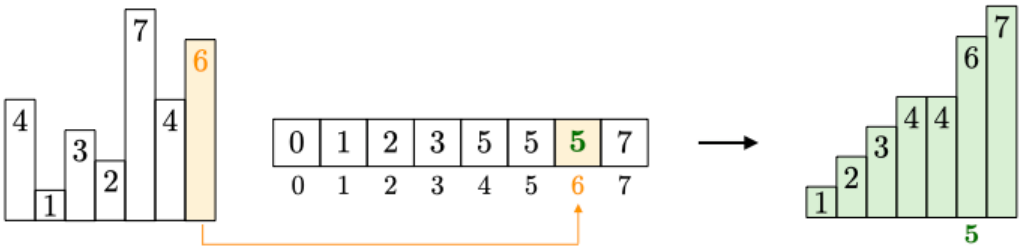
- *Construction of the sorted array:* This step takes $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.
  - *Initialization:* Initialize an array A' of length n that will contain the final sorted array.
  - *Main step:* For each value v of the unsorted array A:
    - Write v to index c_v of A'.



    - Increment c_v by 1 so that any later duplicates can be handled.



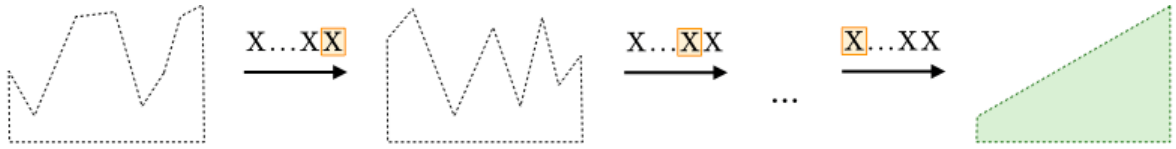
At the end of this process, we obtain the final sorted array A'.



Radix sort

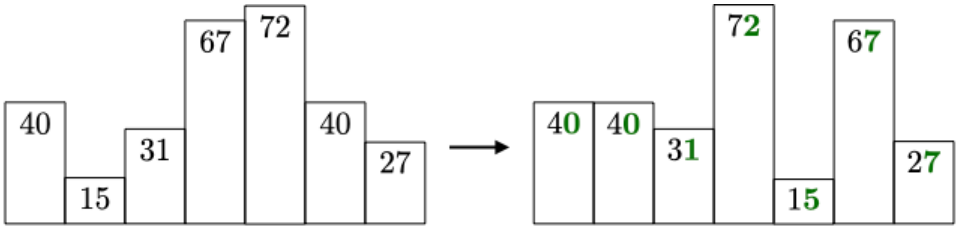
Radix sort is a stable sorting algorithm that is well suited for integer arrays where elements are written with a limited number of digits d, each digit being in the range $[\![0, k]\!]$. This algorithm has a time complexity of $\mathcal{O}(d(n+k))$ and a space complexity of $\mathcal{O}(n+k)$.

Intuition Successively sort elements based on their digits using counting sort.

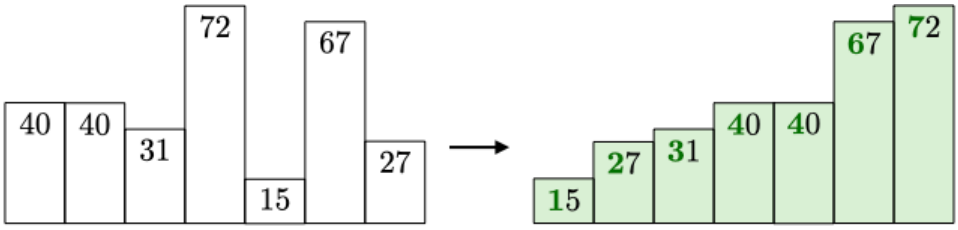


Algorithm

- *Compute step:* Perform counting sort based on the rightmost digit. This step takes $\mathcal{O}(n+k)$ time and $\mathcal{O}(n+k)$ space.



- *Repeat step:* Repeat the *compute step on the remaining digits until reaching the leftmost digit.



At the end of this process, the array is sorted.

The trick of this algorithm lies in the stability property of counting sort: the relative ordering based on a given (weak) digit helps in breaking ties of later (stronger) digits.