Programmation impérative en OCaml

Option Informatique

Anthony Lick
Lycée Janson de Sailly

Analyse d'algorithmes : terminaison, correction, complexité

Analyse d'algorithmes impératifs

Programmation impérative

- En informatique, la programmation **impérative** est un paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.
- Bien que ce ne soit pas la manière de coder la plus utilisée en **OCaml**, c'est tout à fait possible.

Analyse d'algorithmes impératifs

Programmation impérative

- Un objet typique de la programmation impérative est le **tableau**, et une fonction de **tri** qui ne retourne **rien** (le type **unit** en Caml), mais **modifie** le tableau.
- Dans ce chapitre, on va donc voir (ou revoir) les boucles, et la manipulation de tableaux.
- On en profite pour revoir la notion de **terminaison** des fonctions itératives, les preuves de **correction** et l'étude de leur **complexité**.
- Une application de tout ceci se trouve naturellement dans les **tris** : on se limite ici aux tris de base qui de sont pas récursifs.

Terminologie

Pour décrire un algorithme :

- on lui donne en général un nom, on précise quels sont les paramètres qu'il peut prendre en entrée, et le résultat qu'il est sensé renvoyer ;
- on précise aussi de quelle manière il agit sur son **environnement** : modification de la mémoire, affichage éventuel à l'écran, etc.

Tout ceci constitue la **spécification du programme**.

Analyse d'algorithmes impératifs

Blocs simples

Dans nos algorithmes, outre les opérations d'affectations, et de manipulations des variables, on trouve un **découpage en blocs simples** correspondant aux **instructions conditionnelles** et aux **boucles**.

Ce découpage en blocs simples est essentiel : pour montrer que notre algorithme **termine**, qu'il calcule ce qu'il est sensé calculer (**correction**), et pour estimer son coût (**complexité**), il suffit d'analyser chacun de ces blocs.

Terminaison

Terminaison

Pour montrer qu'un algorithme **termine** quel que soit les paramètres passés en entrée (respectant la spécification), il faut montrer que chaque **bloc élémentaire** termine.

Or, les boucles `for` et les **instructions conditionnelles** terminent forcément.

Il reste donc seulement les boucles `while` à analyser.

Terminaison

Terminaison d'une boucle while

En général, pour montrer la terminaison d'une boucle `while`, on exhibe une quantité dépendant des variables, à valeurs dans $\mathbb{N}$, qui **décroit strictement** à chaque passage dans la boucle.

Puisqu'il n'existe pas de suite infinie strictement décroissante dans $\mathbb{N}$, cela prouve que le boucle se termine.

Cette quantité est appelée un **variant de boucle**.

Terminaison

Remarque

Plus généralement, on peut exhiber une quantité à valeurs dans un ensemble $(E, <)$ tel que E ne possède pas de suite strictement décroissante pour $<$ (on dit que $<$ est une relation d'ordre **bien fondée**).

Exemple

C'est le cas de $\mathbb{N}^n$ muni de l'**ordre lexicographique**.

Correction

Correction

Pour montrer qu'un algorithme est **correct**, il faut montrer que quelque soient les paramètres vérifiant sa spécification, l'action de l'algorithme correspond à ce qui est attendu.

Reprenons notre découpage en blocs.

Pour montrer la correction de l'algorithme, il faut montrer que chacun des blocs effectue bien une action précise.

Correction

Pour les blocs **conditionnels**, il n'y a en général pas grand chose à dire de plus que l'algorithme lui-même.

En revanche, analyser les boucles `for` et `while` est essentiel, car l'action de ces boucles n'est pas forcément évidente en première lecture.

Dans ce cas, on se sert d'**invariants de boucles**.

Correction

Définition (Invariant de boucle)

Un **invariant de boucle** est une propriété dépendant des variables de l'algorithme, qui est vérifiée à chaque passage dans la boucle.

Principe

Pour une formulation rigoureuse, on préfère distinguer le cas des boucles `while` et celui des boucles `for`.

Mais dans les deux cas le principe est le même.

Correction

Principe

Une propriété dépendant des paramètres est un **invariant de boucle** si les deux conditions suivantes sont vérifiées :

- la propriété est vérifiée **avant** la boucle ;
- si la propriété est vérifiée en **haut** du corps de boucle, alors elle est vérifiée en **bas** du corps de boucle.

On en conclut (notamment) que la propriété est vérifiée **après** la boucle.

La démonstration qu'une propriété est bien un invariant de boucle est très similaire à une démonstration par **récurrence**.

Boucle `while`

Boucle `while`

Pour les boucles **`while`**, le principe est exactement celui qu'on vient de décrire.

- On exhibe une propriété vraie **avant** la boucle ;
- et qui, si elle est vérifiée en **haut** du corps de la boucle, le sera **en bas**.

On en conclut donc qu'elle est vérifiée **après** la boucle.

```
1  (* Inv *)
2  while condition do
3    (* Inv *)
4    instructions
5    (* Inv *)
6  done ;
7  (* Inv *)
```

Correction des boucles `for`

boucle for

```
1  ...
2  (* Inv(i_d) *)
3  for i = i_d to i_f do
4    (* Inv(i) *)
5    instructions
6    (* Inv(i+1) *)
7  done ;
8  (* Inv(i_f+1) *)
9  ...
```

boucle while équivalente

```
1  let i = ref i_d in
2  (* Inv(i_d) *)
3  while !i <= i_f do
4    (* Inv(i) *)
5    instructions
6    incr i
7    (* Inv(i+1) *)
8  done;
9  (* Inv(i_f+1) *)
```

Principe

La boucle **`for`** de gauche est équivalente à la boucle **`while`** de droite.

L'invariant de la boucle **`while`**, qui dépend à priori de i, est identique dans la boucle **`for`**.

Correction des boucles `for`

boucle for

```
1  ...
2  (* Inv(i_d) *)
3  for i = i_d to i_f do
4    (* Inv(i) *)
5    instructions
6    (* Inv(i+1) *)
7  done ;
8  (* Inv(i_f+1) *)
9  ...
```

boucle while équivalente

```
1  let i = ref i_d in
2  (* Inv(i_d) *)
3  while !i <= i_f do
4    (* Inv(i) *)
5    instructions
6    incr i
7    (* Inv(i+1) *)
8  done;
9  (* Inv(i_f+1) *)
```

Principe

- On montre d'abord que $\mathrm{Inv}(i_d)$ est vrai avant la boucle.
- Ensuite, on montre que si $\mathrm{Inv}(i)$ est vrai **en haut** du corps de la boucle, alors $\mathrm{Inv}(i+1)$ est vrai **en bas** du corps de la boucle.
- Ainsi, une fois la boucle terminée, $\mathrm{Inv}(i_f+1)$ est vrai après la boucle.

Correction des boucles `for`

Remarque

Le principe est le même pour les boucles `for` de la forme :

```
for i=i_d downto i_f do
```

À la différence près qu'il faut montrer que $\mathrm{Inv}(i) \Rightarrow \mathrm{Inv}(i-1)$, et qu'on en déduit que $\mathrm{Inv}(i_f - 1)$ est vrai après la boucle.

Complexité

Complexité

L'étude de la complexité d'une fonction consiste à estimer son coût (temporel ou en mémoire), en fonction des entrées.

Pour différencier deux entrées entre elles, on compare en général leur taille.

Essentiellement pour nous, les entrées seront constituées d'entiers ou de tableaux.

Complexité

Pour les tableaux, la donnée pertinente est la taille.

Pour les entiers, cela dépend du contexte.
Pour un entier n, on peut en effet exprimer la complexité d'une fonction dépendant de n en fonction :

- de l'entier n lui-même ;
- ou de son nombre de chiffres (sa taille), correspondant à $\ln(n)$, à une constante multiplicative près.

En général, on ne tient pas compte des constantes multiplicatives.

Complexité

Exemple

- Pour une fonction calculant $n!$, le nombre d'opérations est clairement linéaire en n.
- Pour exprimer la complexité d'une fonction qui renvoie l'écriture en base 2 d'un nombre exprimé en base 10, on se dirigerait plus naturellement vers $\log_2(n)$.

Complexité

Complexité temporelle

L'exécution d'un algorithme est une séquence d'opérations nécessitant plus ou moins de **temps**.

Pour mesurer ce temps, on considère certaines opérations comme **élémentaires** : par exemple faire une opération arithmétique de base (addition, multiplication, soustraction, division...), lire ou modifier un élément d'un tableau, ajouter un élément à la fin d'un tableau, affecter un entier ou un flottant, etc.

Estimer la **complexité temporelle** d'une fonction sur une entrée de taille donnée signifie estimer le nombre de ces opérations élémentaires effectuées par la fonction sur l'entrée (on suppose que notre ordinateur effectue chaque opération élémentaire en **temps constant**).

Complexité

Complexité spatiale

La **complexité en mémoire** consiste à estimer la mémoire nécessaire à une fonction pour son exécution.

De nos jours, la mémoire disponible sur nos ordinateurs n'est en général pas limitant.

La **complexité spatiale** est donc moins étudiée.

Complexité

Différentes complexités

Les notions de complexité au programme sont les complexités **temporelles** dans le **meilleur** et dans le **pire** cas, c'est à dire le nombre minimal/maximal que requiert l'algorithme pour s'exécuter sur une entrée de taille n.

La complexité dans le **meilleur** des cas n'est en général pas la plus pertinente pour départager deux algorithmes, mais peut aider si les algorithmes en question ont la même complexité dans le pire cas.

La complexité au **pire** est le nombre d'opérations maximales nécessaires pour traiter une entrée de taille donnée.

Complexité

Remarques

Une autre notion de complexité intéressante est la complexité **en moyenne**, qu'on abordera assez peu : elle demande des connaissances en probabilité, et une distribution de probabilités sur les entrées possibles.

En conclusion, lorsqu'on demandera d'exprimer la complexité d'une fonction sans plus de précision, on sous-entendra la complexité dans le **pire cas**.

Complexité asymptotique et notations de Landau

Complexité asymptotique

Supposons pour simplifier que l'algorithme dont on veut calculer la complexité ne prenne qu'un seul paramètre en entrée, de taille n.

Lorsqu'on étudie la complexité $C(n)$ de l'algorithme, c'est bien souvent pour les grandes valeurs de n qu'on s'y intéresse, pour comparer à d'autres algorithmes réalisant la même tâche.

On cherche donc à étudier le comportement asymptotique de $C(n)$, qu'on rapportera aux fonctions usuelles : logarithmes, polynômes, exponentielles.

Complexité asymptotique et notations de Landau

Complexité asymptotique

De plus, on ne cherchera pas systématiquement un équivalent : celui-ci est souvent difficile à obtenir, et n'est pas le plus important.

Exemple

Si deux fonctions nécessitent respectivement $9n\ln(n)$ et $\frac{n^2}{2}$ opérations élémentaires, on retiendra simplement que la première nécessite de l'ordre de $n\ln(n)$ opérations, ce qui est bien meilleur que la seconde qui en nécessite de l'ordre de n^2.

Complexité asymptotique et notations de Landau

Notations de Landau

Soit f et g deux fonctions $\mathbb{N} \to \mathbb{R}_+^*$. On note :

- $f(n) = O(g(n))$ s'il existe $n_0 \in \mathbb{N}$ et $M > 0$ tels que $\forall n \geq n_0,\ f(n) \leq M \cdot g(n)$.
- $f(n) = \Omega(g(n))$ si $g(n) = O(f(n))$.
- $f(n) = \Theta(g(n))$ si $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$.

En pratique

Pour exprimer la complexité $C(n)$, on se contentera bien souvent d'un O, qui est d'ailleurs la seule au programme.

Pour préciser que la borne obtenue est essentiellement optimale, on pourra parler de Θ.

Un exemple important : l'exponentiation

Exemple

Un exemple important est le calcul de x^n pour un entier naturel n donné.

Il n'existe pas de fonction permettant ce calcul sur les entiers en Caml.

On présente ici deux fonctions permettant ce calcul ; l'une est naïve, l'autre est plus élaborée.

Un exemple important : l'exponentiation

```
1  let expo x n=
2    let y=ref 1 in
3    (* Inv(0): y=x^0 *)
4    for i=0 to n-1 do
5      (* Inv(i): y=x^i *)
6      y:=!y*x
7      (* Inv(i+1): y=x^(i+1) *)
8    done ;
9    (* Inv(n): y=x^n *)
10   !y
11 ;;
```

Algorithme naïf

Il consiste à multiplier n fois une variable x, en partant de 1.

Terminaison

La terminaison est évidente : on n'utilise qu'une boucle `for`.

Correction

On peut montrer l'invariant de boucle : $\mathrm{Inv}(i)$: $y = x^i$.

On renvoie donc la valeur de $y = x^n$ après la boucle.

Complexité

La complexité est en $O(n)$ car on effectue n multiplications.

Exponentiation rapide

Exponentiation rapide

Une autre idée consiste à utiliser la décomposition en binaire de l'entier n.

Exemple

Si on souhaite calculer x^{11}, on regarde l'écriture de 11 en binaire : $\overline{1011}^2$.

À partir de x, on peut calculer les x^{2^p} : x, x^2, x^4, x^8.

Comme $11 = \overline{1011}^2$, on a $x^{11} = x^8 \times x^2 \times x$.

Exponentiation rapide

Algorithme d'exponentiation rapide

```
1  let expo_rapide x n =
2    let m = ref n
3    and y = ref x
4    and z = ref 1 in
5    (* Inv: z*y^m = x^n *)
6    while !m > 0 do
7      (* Inv *)
8      if !m mod 2 = 1 then
9        z := !z * !y ;
10     m:=!m/2 ;
11     y:=!y * !y
12     (* Inv *)
13   done ;
14   (* Inv *)
15   !z
16 ;;
```

Exponentiation rapide

L'algorithme de multiplication suivant ce schéma porte le nom d'**algorithme d'exponentiation rapide**. On va voir qu'il est plus efficace que l'algorithme d'exponentiation naïf.

Exponentiation rapide

Terminaison

La terminaison vient du fait que les valeurs prises par m forment une suite dans $\mathbb{N}$ **strictement décroissante**.

```
1  let expo_rapide x n =
2    let m = ref n
3    and y = ref x
4    and z = ref 1 in
5    (* Inv: z*y^m = x^n *)
6    while !m > 0 do
7      (* Inv *)
8      if !m mod 2 = 1 then
9        z := !z * !y ;
10     m:=!m/2 ;
11     y:=!y * !y
12     (* Inv *)
13   done ;
14   (* Inv *)
15   !z
16 ;;
```

Complexité

En terme de **complexité**, on fait un nombre **borné** de multiplications à chaque passage de boucle, et il est facile de voir que le **nombre de passages** dans la boucle est égal au nombre de chiffres dans l'écriture en binaire de n, c'est à dire $O(\log(n))$.

Exponentiation rapide

```
1   let expo_rapide x n =
2     let m = ref n
3     and y = ref x
4     and z = ref 1 in
5     (* Inv: z*y^m = x^n *)
6     while !m > 0 do
7       (* Inv *)
8       if !m mod 2 = 1 then
9         z := !z * !y ;
10      m:=!m/2 ;
11      y:=!y * !y
12      (* Inv *)
13    done ;
14    (* Inv *)
15    !z
16  ;;
```

Correction

L'invariant de boucle est : $zy^m = x^n$.

En effet, c'est vrai pour les valeurs initiales des variables, et pour voir que cette propriété est conservée lors d'un passage dans la boucle, il suffit de distinguer suivant la parité de m.

À la fin, on a $m = 0$ (car on est sorti de la boucle), donc $z = x^n$, et l'algorithme renvoie bien la bonne valeur.

Les tableaux en Caml

Les tableaux en OCaml

Array

On a vu les tableaux (type `array`) dans le premier chapitre.

On va maintenant voir quelques fonctions classiques sur ce type.

Quelques exemples de fonctions basiques

Fonctions de base

Voici quelques fonctions de base qu'il faut savoir recoder vite :

- trouver le minimum d'un tableau ;
- tester l'appartenance à un tableau ;
- compter le nombre d'occurrences d'un élément dans un tableau.

On rappelle que `incr i` est équivalent à `i := !i + 1`.

Quelques exemples de fonctions basiques

```
1  let mini t =
2    let m = ref t.(0) in
3    for i=1 to Array.length t - 1 do
4      m := min !m t.(i)
5    done ;
6    !m
7  ;;
```

Minimum d'un tableau

Pour trouver le minimum de `t`, il faut simplement parcourir tout le tableau, et stocker le minimum courant dans une référence.

Quelques exemples de fonctions basiques

```
1  let appartient t x =
2    let b = ref false in
3    for i = 0 to Array.length t - 1 do
4      b := !b || t.(i) = x
5    done ;
6    !b
7  ;;
```

```
1  let appartient t x =
2    let b = ref false and i = ref 0 in
3    while not !b && !i< Array.length t do
4      b := t.( !i) = x ;
5      incr i
6    done ;
7    !b
8  ;;
```

Appartenance

Remarquez l'utilisation du caractère paresseux de l'opérateur `||` dans la fonction **appartient** : on n'évalue pas `t.(i) = x` si `!b` est égal à `true`.

Pour éviter de parcourir toute le tableau pour vérifier si au moins un élément du tableau vérifie une certaine propriété, on peut utiliser une boucle **`while`** (pensez à **incrémenter** i).

Quelques exemples de fonctions basiques

```
1  exception Trouve ;;
2
3  let appartient t x =
4    try
5      for i=0 to Array.length t - 1 do
6        if t.(i) = x then raise Trouve
7      done ;
8      false
9    with Trouve -> true
10 ;;
```

Appartenance

On peut aussi utiliser une **exception**, comme vu dans le premier chapitre.

Quelques exemples de fonctions basiques

```
1  let occurences t x=
2    let c=ref 0 in
3    for i=0 to Array.length t -1 do
4      c:= !c + (if t.(i) = x then 1 else 0)
5    done ;
6    !c
7  ;;
```

Minimum d'un tableau

Pour compter le nombre d'occurrences, on se sert d'une **référence** comme **compteur**.

Remarquez l'utilisation d'une **instruction conditionnelle** à l'intérieur d'une **expression arithmétique**.

Recherche dichotomique dans un tableau trié

Recherche dans un tableau trié

Si le tableau est trié dans l'ordre croissant, on peut accélérer considérablement la recherche d'un élément.

Si on sait que l'élément à chercher ne peut se trouver qu'entre les indices g inclus et d exclus, on peut discriminer la moitié de la portion en regardant l'élément au milieu de la portion, d'indice $m = \lfloor \frac{g+d}{2} \rfloor$.

- Si l'élément d'indice m est l'élément cherché, on a terminé.
- Si l'élément d'indice m est $<$ à l'élément cherché, celui-ci ne peut se trouver qu'entre $m+1$ inclus et d exclus.
- Si l'élément d'indice m est $>$ à l'élément cherché, celui-ci ne peut se trouver qu'entre g inclus et m exclus.

Recherche dichotomique dans un tableau trié

Initialisation et condition d'arrêt

Initialement, on a $g = 0$ et $d = n$ avec n la taille du tableau.

On peut s'arrêter lorsque $g = d$: dans ce cas, on est sûr que l'élément ne se trouve pas dans le tableau.

Recherche dichotomique dans un tableau trié

```
1   exception Trouve ;;
2
3   let recherche_dicho t x =
4     try
5       let g, d = ref 0, ref (Array.length t) in
6       while !g < !d do
7         let m = (!g + !d) / 2 in
8         if t.(m) = x then raise Trouve ;
9         if t.(m) > x then d := m else g := m+1
10      done ;
11      false
12    with Trouve -> true
13  ;;
```

Terminaison et complexité

La portion $d - g$ sur laquelle on travaille a initialement une taille $t_0 = n$ (la taille du tableau), et diminue au moins de moitié à chaque itération : $t_{n+1} \leq t_n/2$.

Ceci montre que l'algorithme termine en $O(\log n)$ étapes.

Le problème du tri

Problème

Étant donné un tableau t d'éléments d'un ensemble E, et un ordre $\preceq$ sur E, modifier t pour qu'il contienne les mêmes éléments qu'au départ, mais **triés** dans l'ordre croissant par rapport à $\preceq$.

Définition (relation d'ordre)

Soit E un ensemble. Une relation $\preceq$ de $E \times E$ est une **relation d'ordre** sur E si elle vérifie les propriétés suivantes :

- **transitivité** : $\forall x, y, z \in E,\ x \preceq y$ et $y \preceq z \Rightarrow x \preceq z$
- **Réflexivité** : $\forall x \in E,\ x \preceq x$
- **antisymétrie** : $\forall x, y \in E,\ x \preceq y$ et $y \preceq x \Rightarrow x = y$

Ordre total & pré-ordre

Définition (ordre total)

Une relation d'ordre $\preceq$ est dite **totale** si $\forall x, y \in E,\ x \preceq y$ ou $y \preceq x$.

Définition (pré-ordre)

Un **pré-ordre** est une relation transitive et réflexive, mais pas antisymétrique.

Ordre total & pré-ordre

Définition (ordre total)

Une relation d'ordre $\preceq$ est dite **totale** si $\forall x, y \in E,\ x \preceq y$ ou $y \preceq x$.

Définition (pré-ordre)

Un **pré-ordre** est une relation transitive et réflexive, mais pas antisymétrique.

Exemple

Les ensembles $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, ... sont munis de l'ordre **total** $\leq$ standard.

Exemples de relations d'ordre

Exemple

$\mathbb{R}^2$ peut-être muni de l'**ordre lexicographique** $\preceq_{lex}$ défini par :

$$(x, y) \preceq_{lex} (x', y') \iff x < x' \text{ ou } (x = x' \text{ et } y \leq y')$$

C'est un ordre **total**. Il se généralise à $\mathbb{R}^n$, ou a des produits cartésiens $(E_1, \leq_1) \times \cdots \times (E_n, \leq_n)$ d'ensembles ordonnés.

Exemples de relations d'ordre

Exemple

$\mathbb{R}^2$ peut-être muni de l'**ordre lexicographique** $\preceq_{lex}$ défini par :

$$(x, y) \preceq_{lex} (x', y') \iff x < x' \text{ ou } (x = x' \text{ et } y \leq y')$$

C'est un ordre **total**. Il se généralise à $\mathbb{R}^n$, ou a des produits cartésiens $(E_1, \leq_1) \times \cdots \times (E_n, \leq_n)$ d'ensembles ordonnés.

Exemple

$\mathbb{R}^2$ peut aussi être muni du **pré-ordre** $\preceq_1$ défini par :

$$(x, y) \preceq_1 (x', y') \iff x \leq x'$$

Ce n'est pas un ordre car $(0, 0) \preceq_1 (0, 1)$ et $(0, 1) \preceq_1 (0, 0)$.

Exemples de relations d'ordre

Exemple

On peut munir $\mathbb{R}^2$ de l'**ordre produit** $\preceq_\times$ défini par :

$$(x, y) \preceq_\times (x', y') \iff x \leq x' \text{ et } y \leq y'$$

Ce n'est pas un ordre total, car $(0, 1)$ et $(1, 0)$ ne sont pas comparables.

Exemples de relations d'ordre

Exemple

On peut munir $\mathbb{R}^2$ de l'**ordre produit** $\preceq_\times$ défini par :

$$(x, y) \preceq_\times (x', y') \iff x \leq x' \text{ et } y \leq y'$$

Ce n'est pas un ordre total, car $(0, 1)$ et $(1, 0)$ ne sont pas comparables.

Exemple

L'ensemble des chaînes de caractères peut-être muni de l'ordre lexicographique (c'est celui du dictionnaire).

Les algorithmes de tri

Présentation

On détaille ici les algorithmes de tris "**naifs**" les plus classiques.

Ceux-ci sont **quadratiques** (complexité $O(n^2)$ avec n la taille du tableau) et sont donc inefficaces pour de grands tableaux.

On leur préférera l'un des tris **récursifs** (qu'on verra plus tard) dès que le nombre d'éléments à trier dépasse environ 50.

Les algorithmes de tri

Remarque

Les algorithmes de tri qu'on va voir ici auront pour type : `'a array -> unit`.

En effet, ils modifieront directement le tableau donné en entrée, et n'auront donc pas besoin de renvoyer de valeur.
Un tel tri est dit "**en place**".

Lorsqu'un algorithme modifie des variables qui ne lui sont pas internes, on dit qu'il a des **effets de bord**.

Tri par sélection

Remarque

Ce tri particulièrement simple est peut-être celui auquel on pense en premier lorsqu'on écrit un algorithme de tri.

Idée

Supposons qu'un tableau de taille n est déjà en partie trié avec ses i premiers éléments à leur place définitive.

On **sélectionne** le plus petit des $n-i$ éléments restants, qu'on amène en position $i+1$. Le tableau a alors ses $i+1$ premiers éléments à leur place définitive.

Itérer ce procédé $n-1$ fois suffit pour trier le tableau.

Tri par sélection

échange de deux éléments d'un tableau

```
1  let echange t i j=
2    let a=t.(i) in
3    t.(i) <- t.(j) ;
4    t.(j) <- a
5  ;;
```

Échange

On commence par coder la fonction `echange` ci-dessus qui nous sera utile.

Son type est : `'a array -> int -> int -> unit`.

Tri par sélection

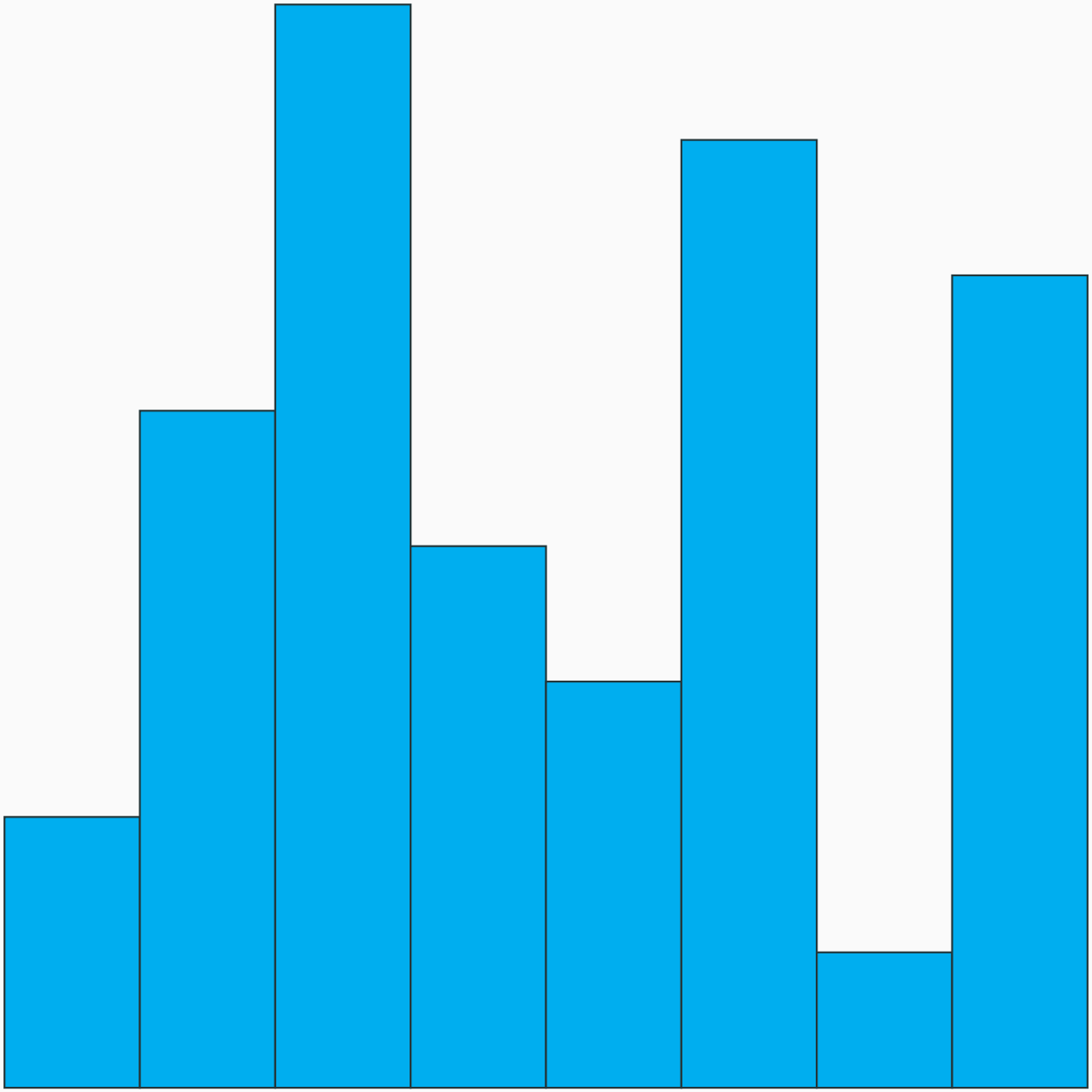
```
1  let tri_selection t =
2    let n = Array.length t in
3    for i=0 to n-2 do
4      (* Inv(i): t.(0),...,t.(i-1) sont triés dans l'ordre croissant
5         et plus petits que les autres éléments de t *)
6      let imin = ref i in
7      for j=i+1 to n-1 do
8        (* Inv2(j): i_min est l'indice du plus petit élément parmi t.(i),...,t.(j-1) *)
9        if t.(j) < t.( !imin) then imin := j
10       (* Inv2(j+1) *)
11     done ;
12     if !imin>i then echange t i !imin
13     (* Inv(i+1) *)
14   done
15 ;;
```

Terminaison et complexité

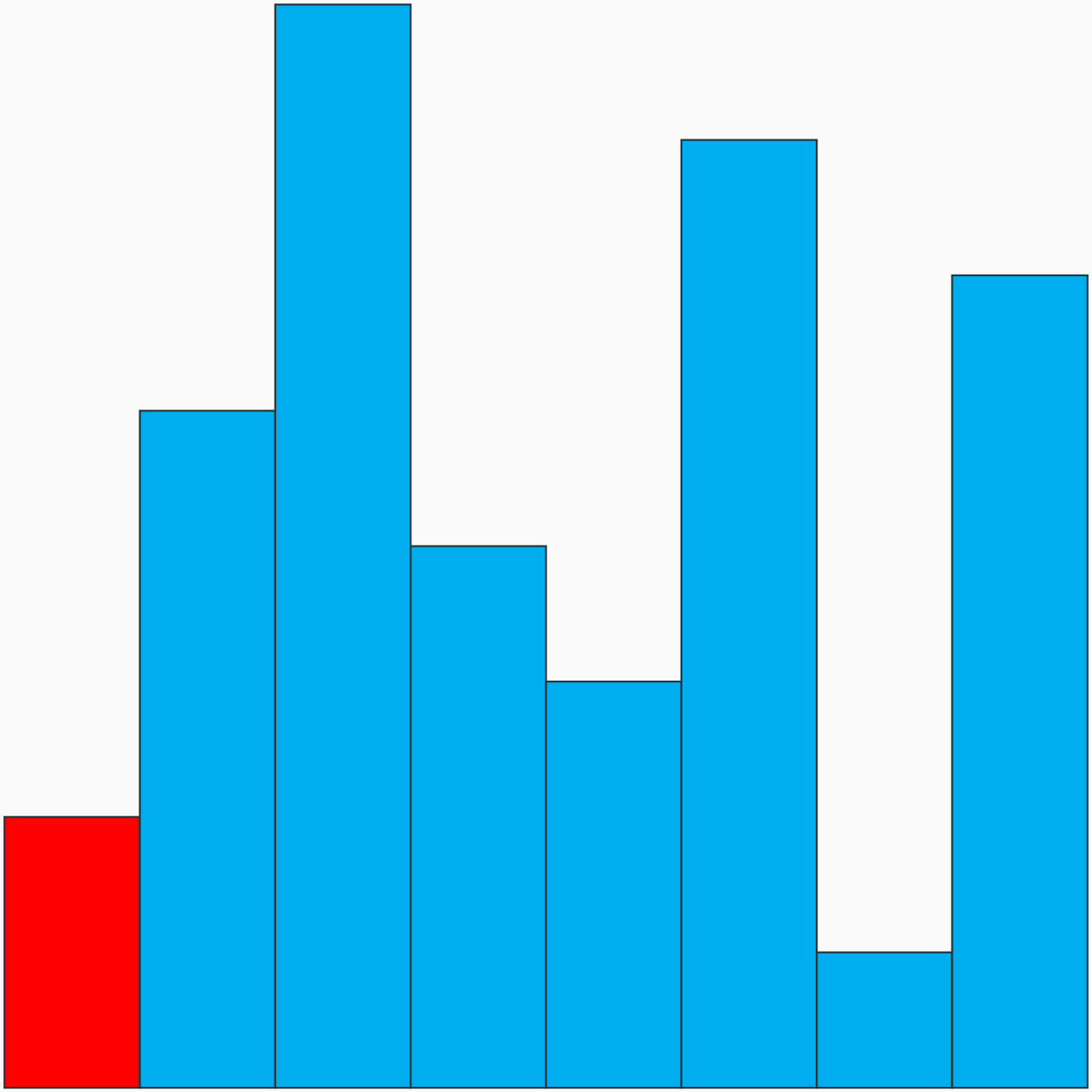
L'algorithme de tri par sélection est constitué de deux boucles imbriquées, donc il termine.

Sa complexité est proportionnelle à $\sum_{i=0}^{n-2}(n-i) = O(n^2)$.

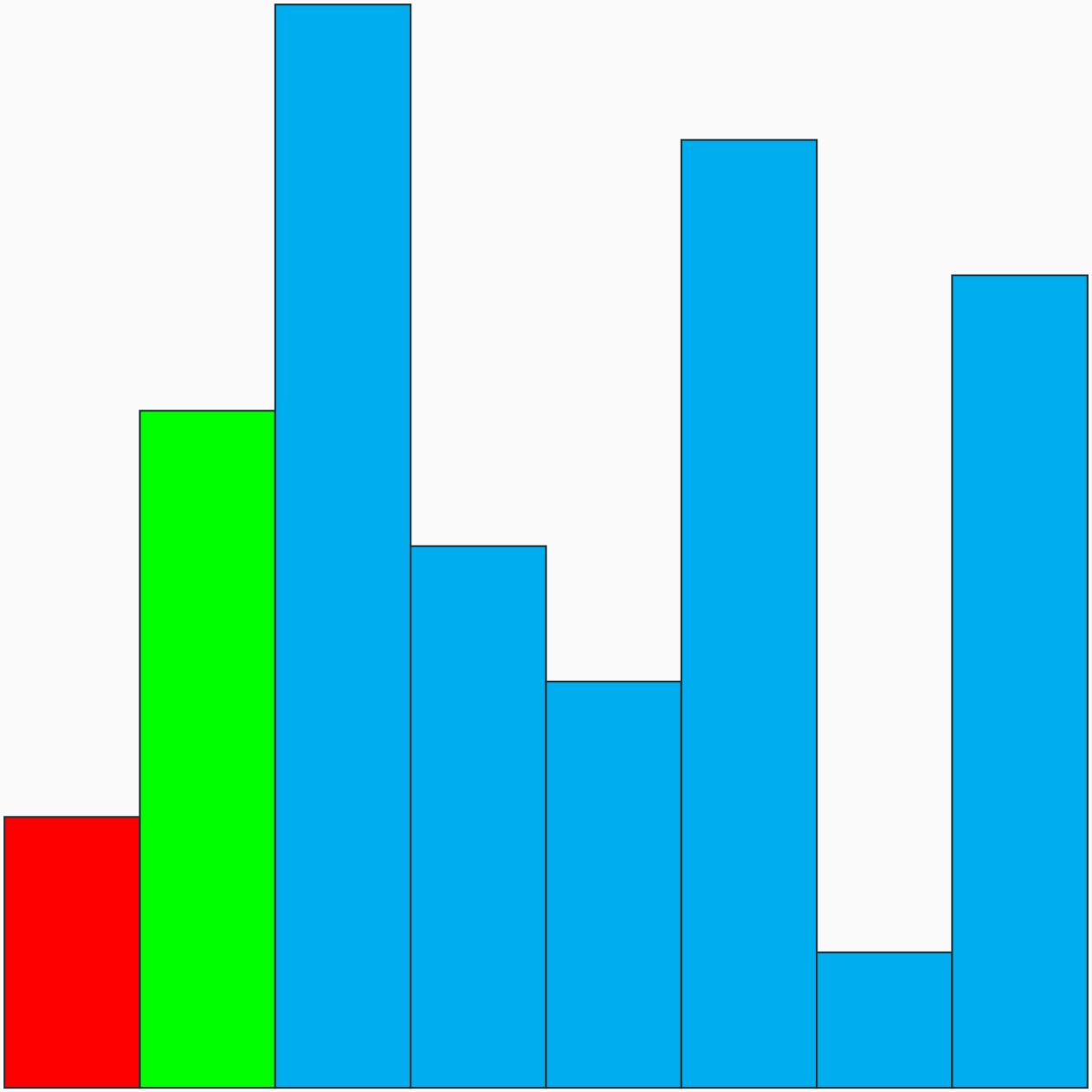
Exemple



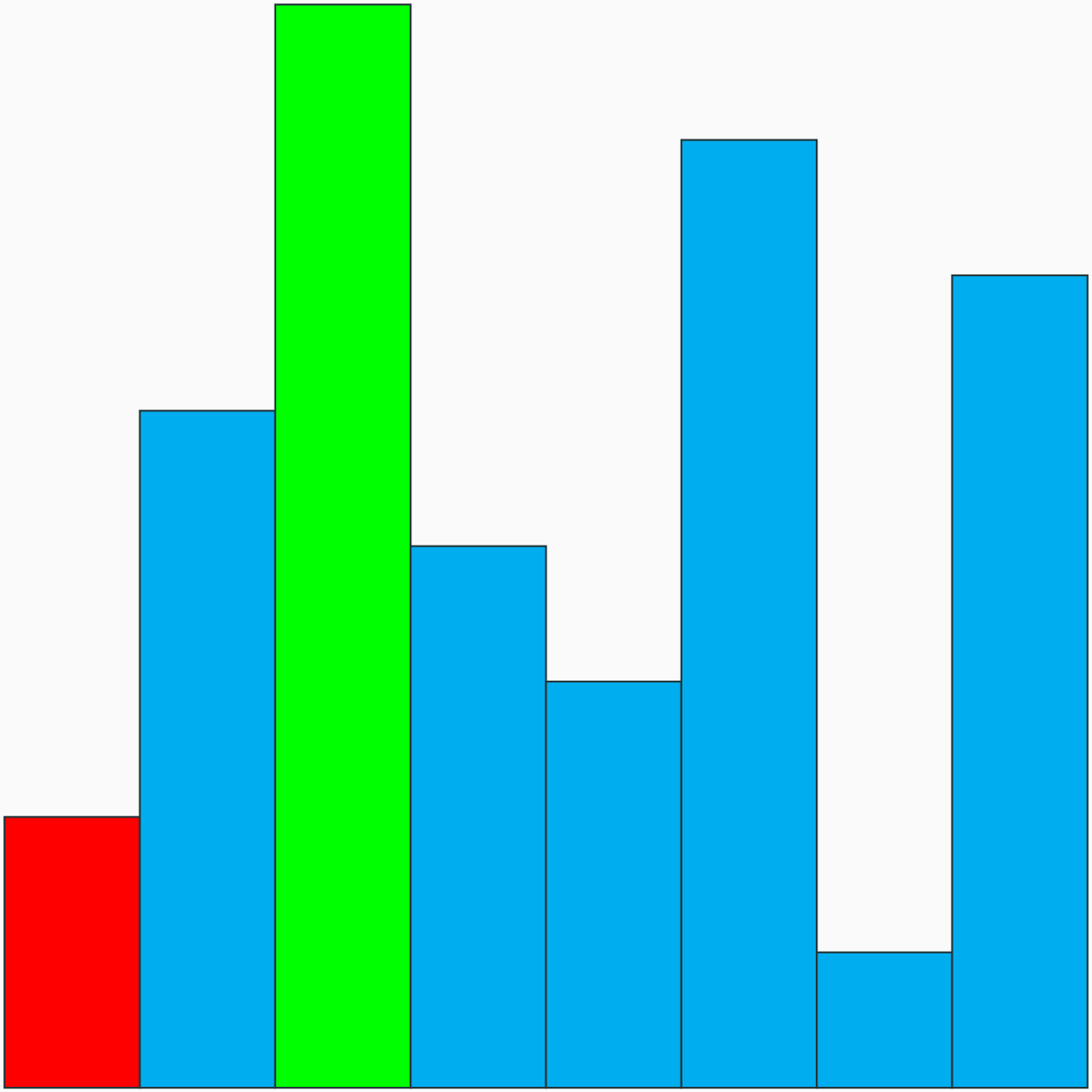
Exemple



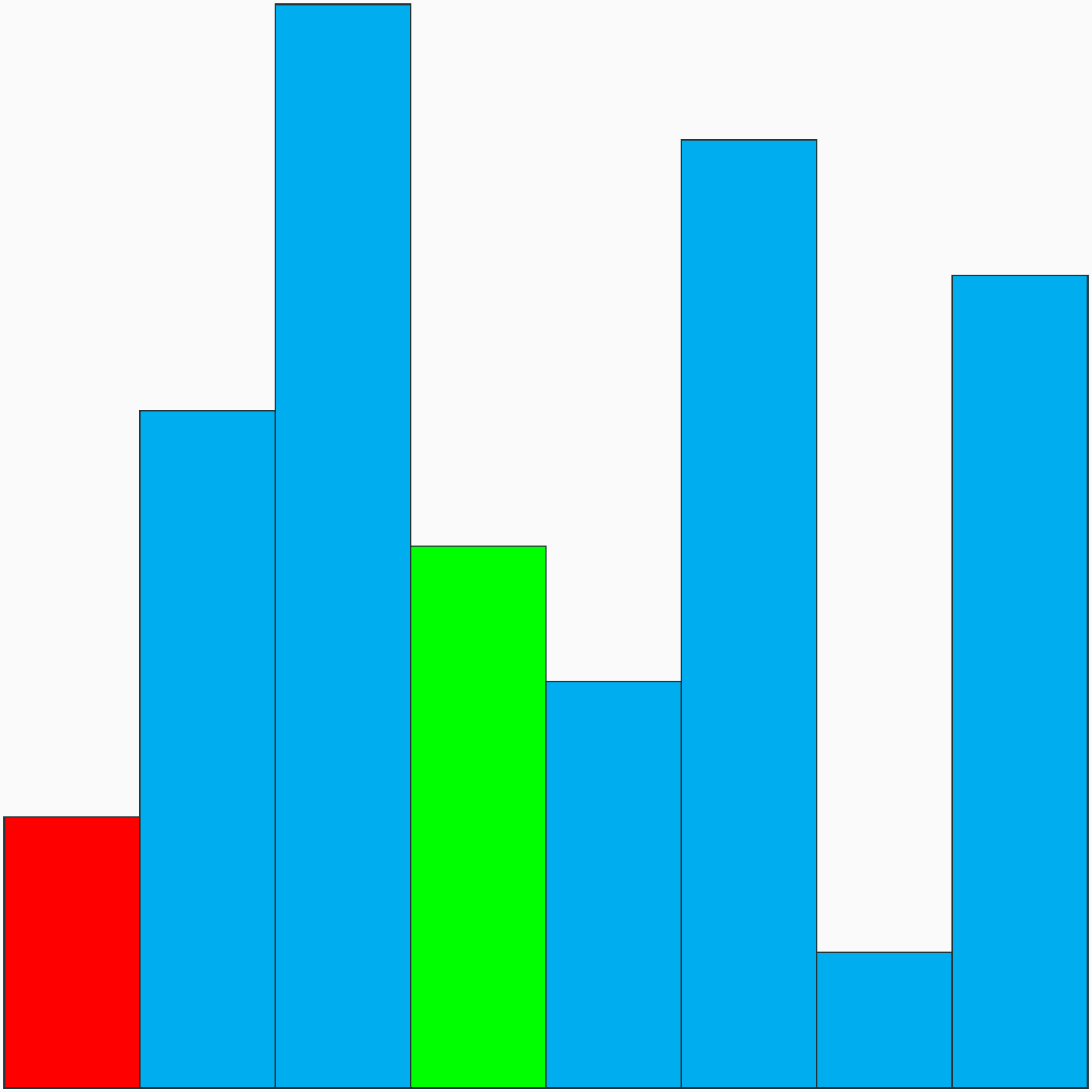
Exemple



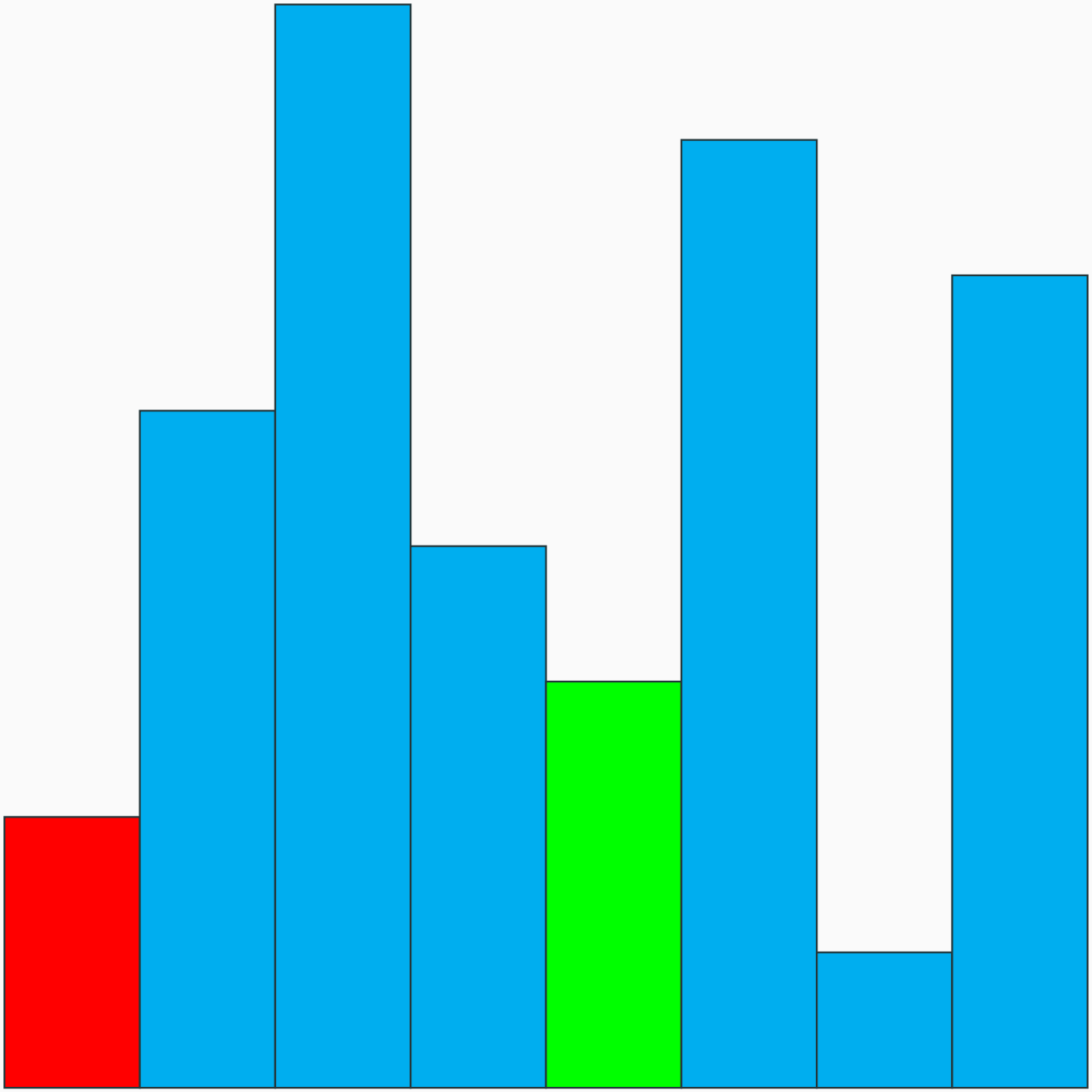
Exemple



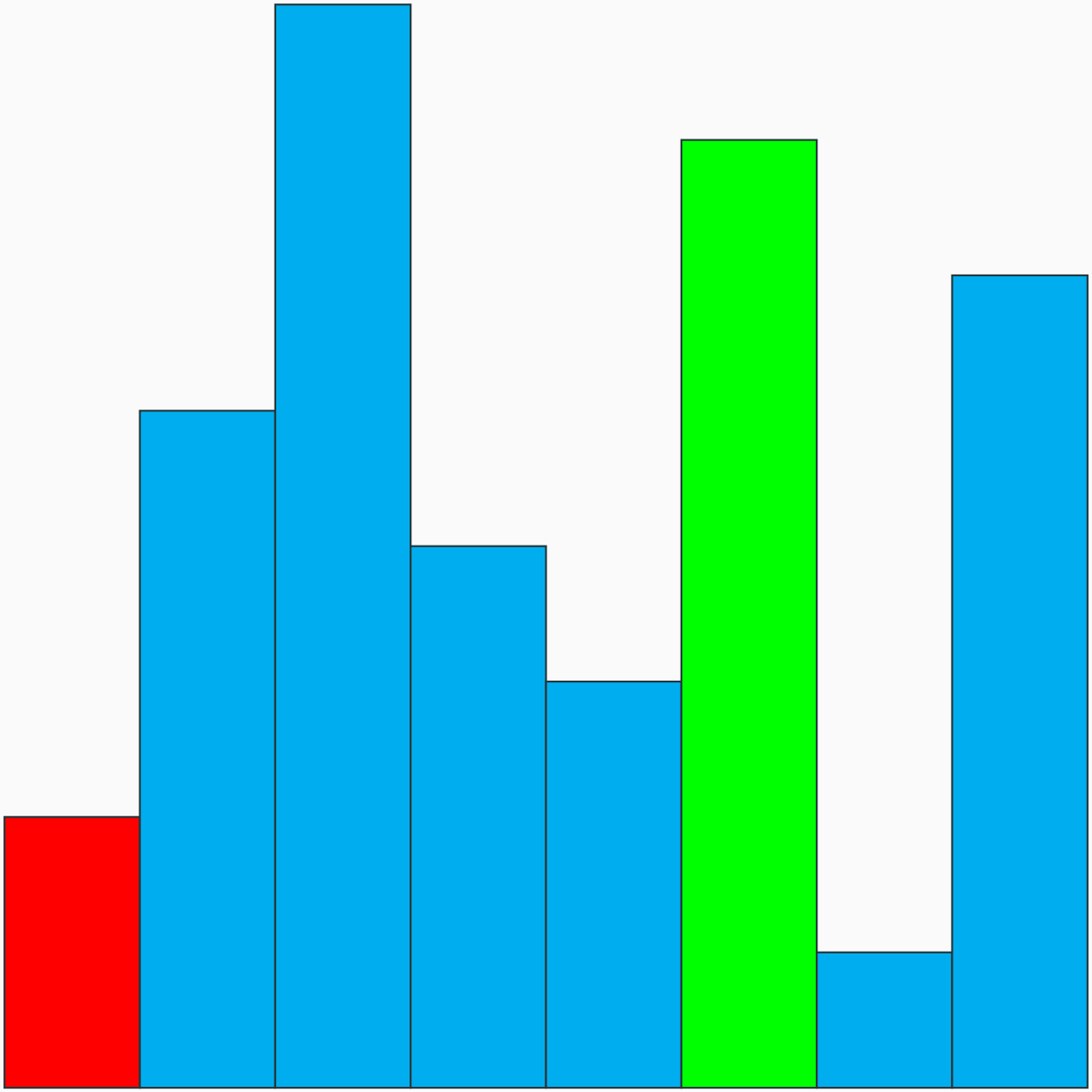
Exemple



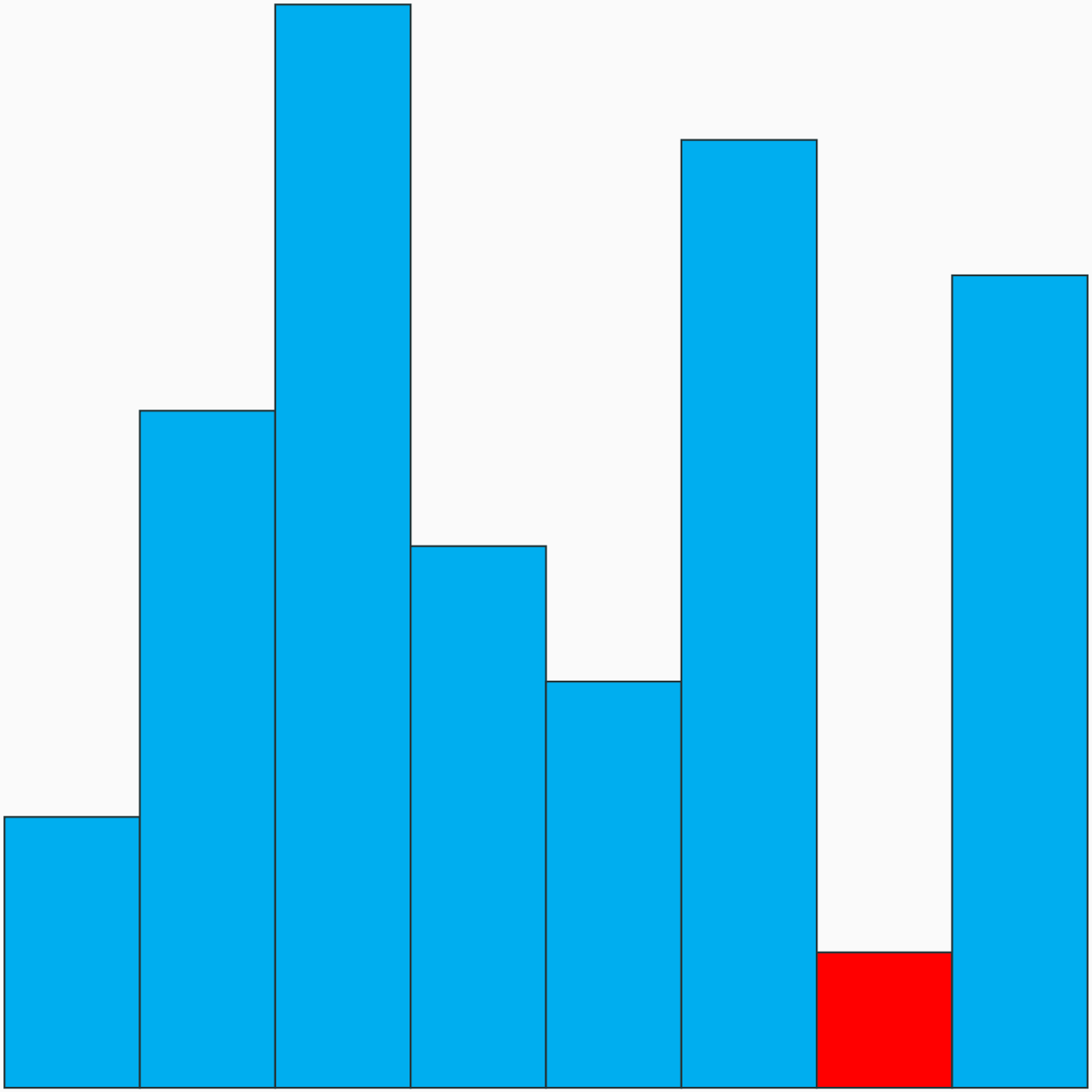
Exemple



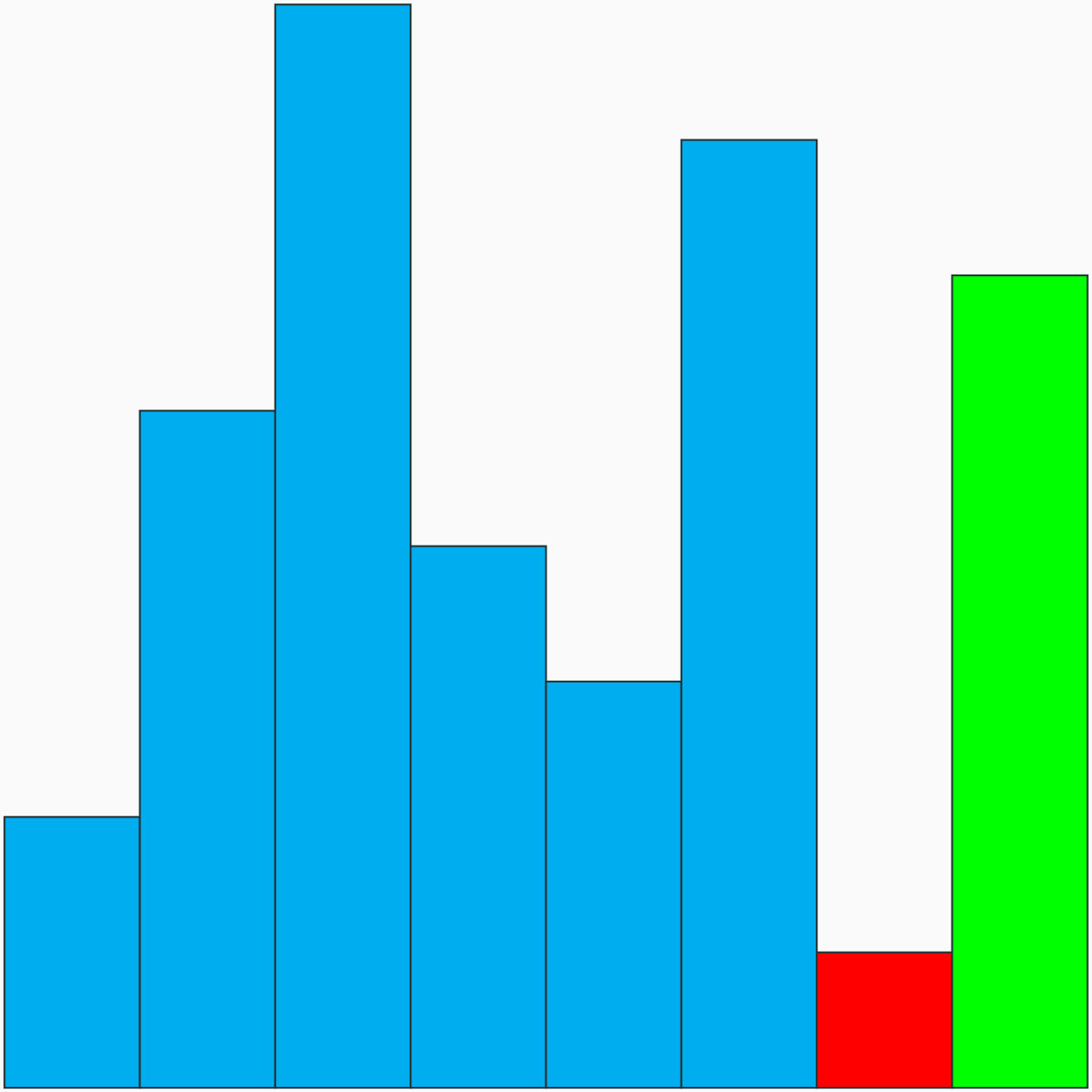
Exemple



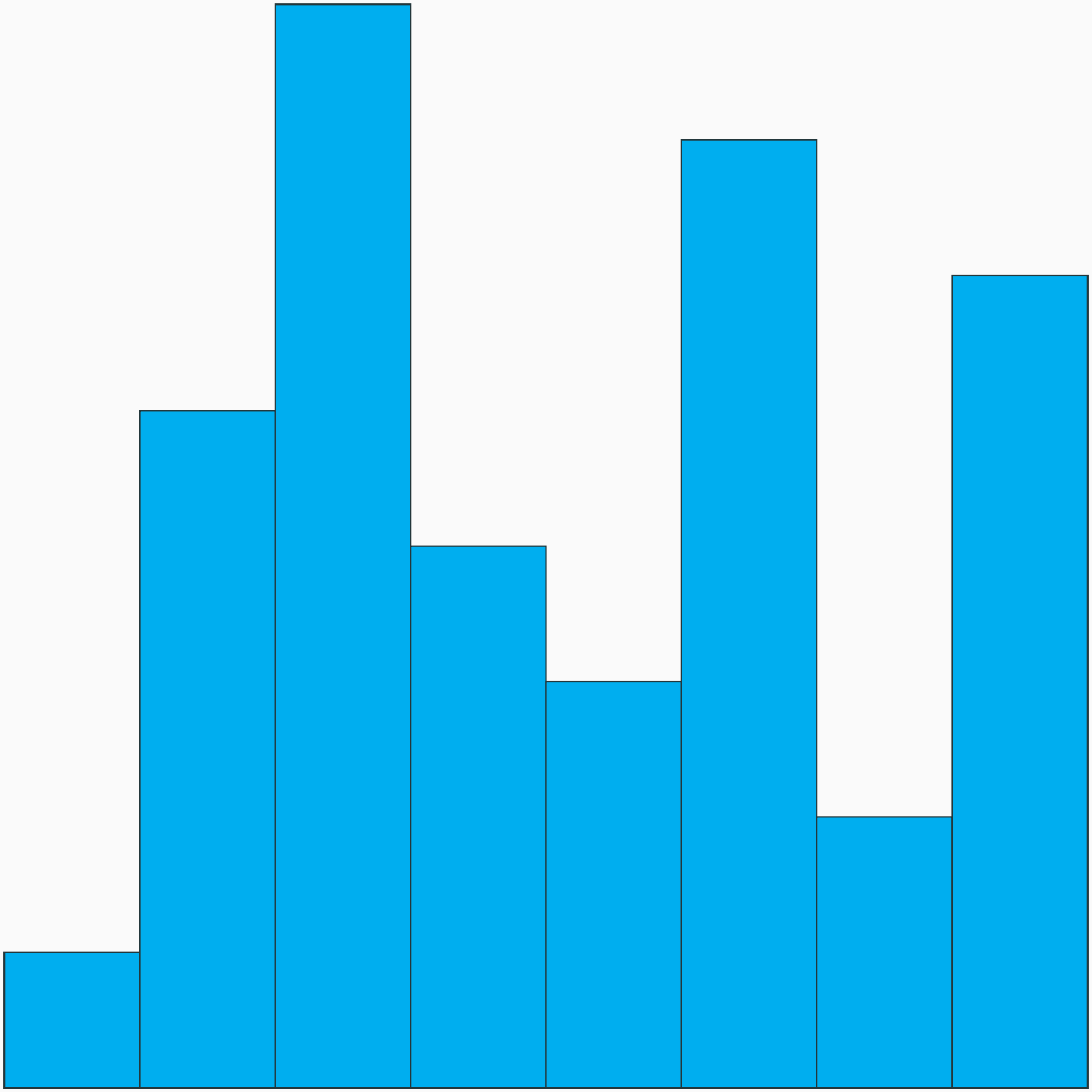
Exemple



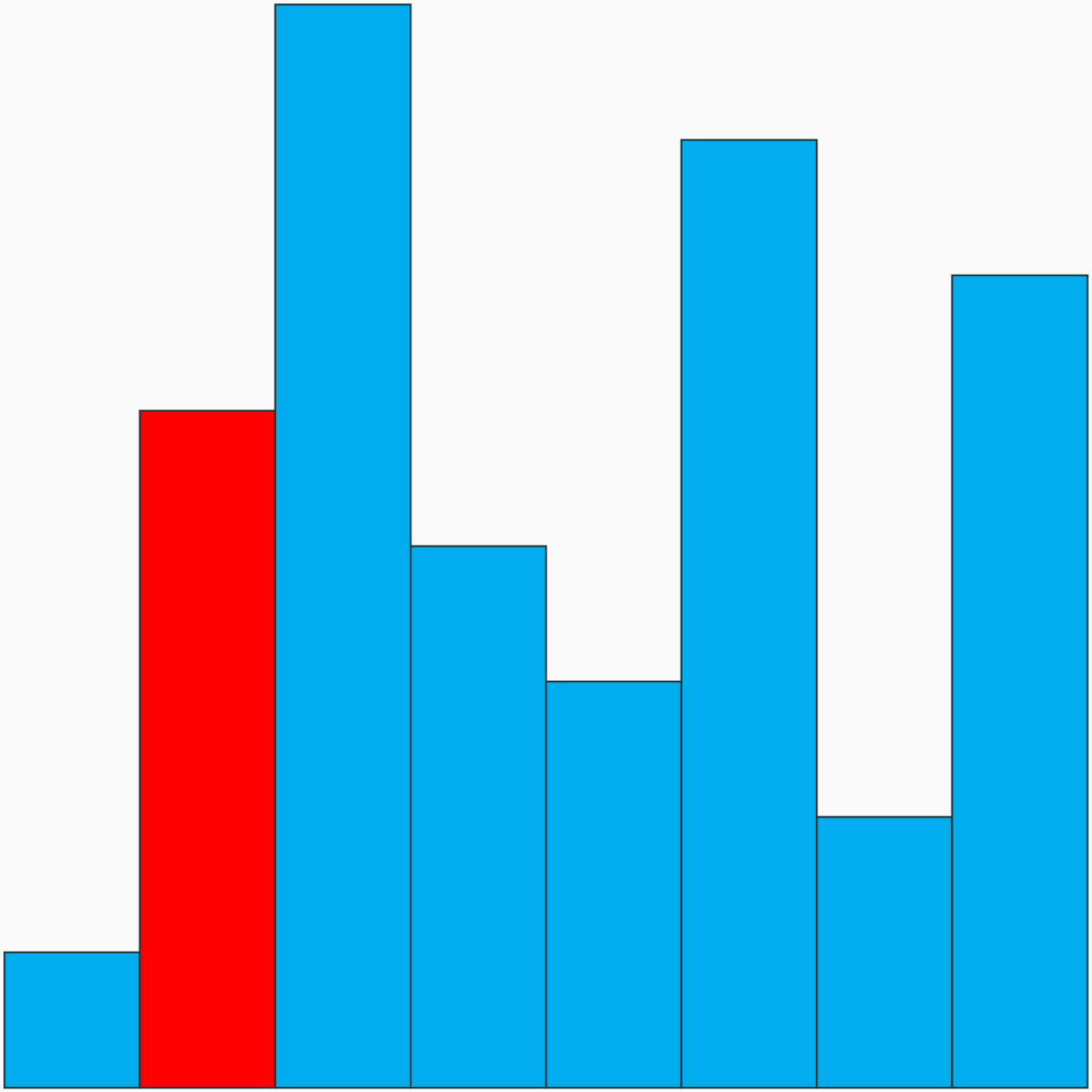
Exemple



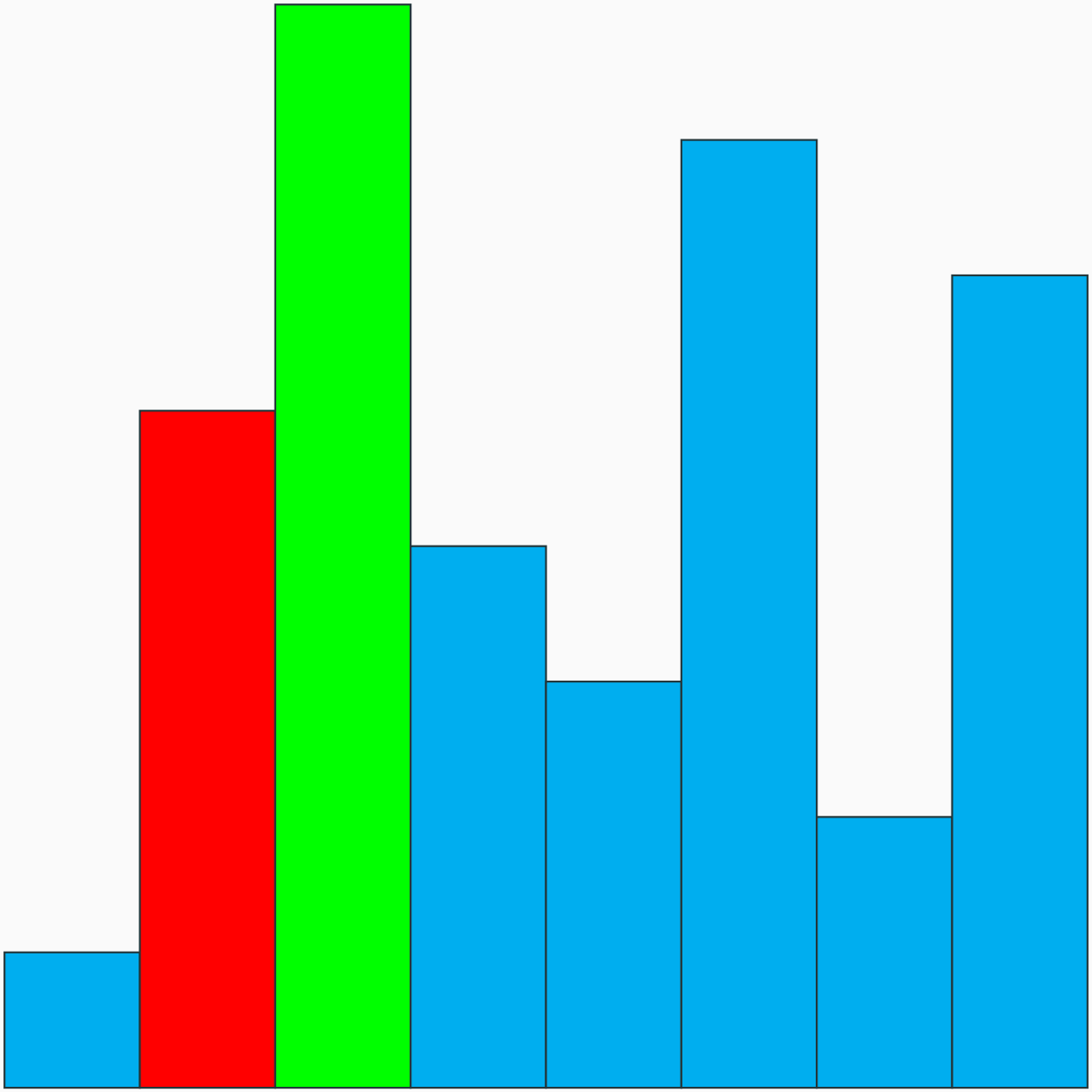
Exemple



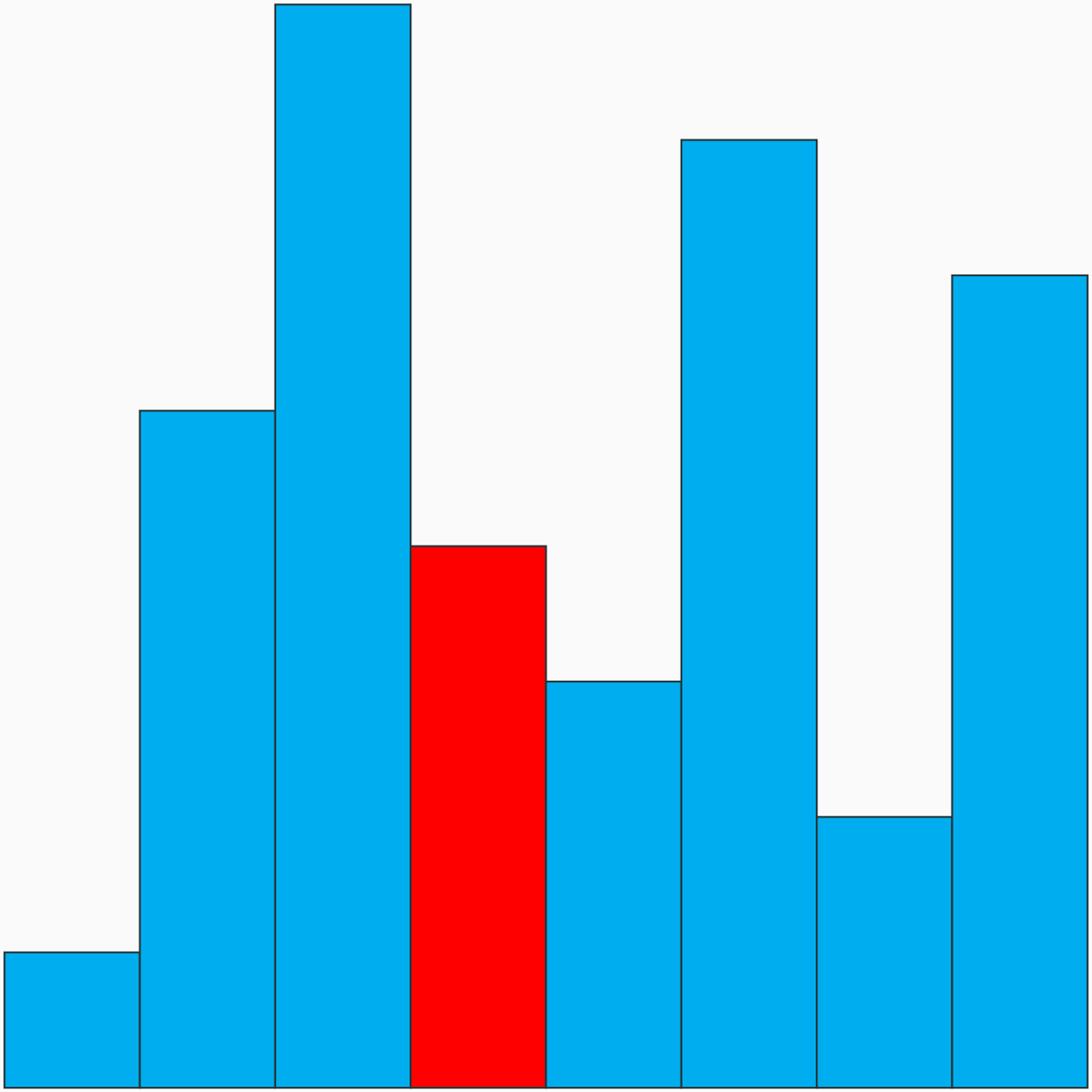
Exemple



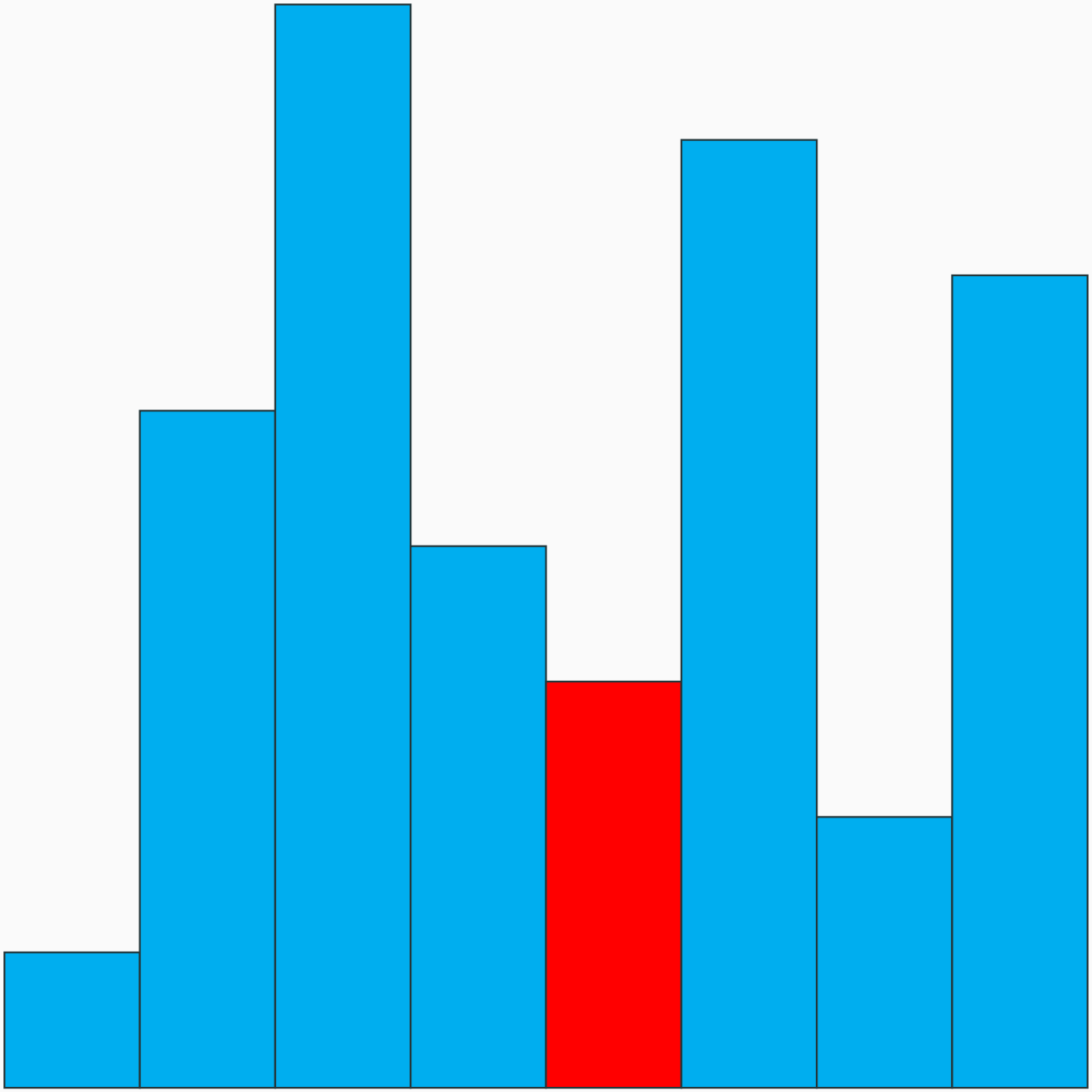
Exemple



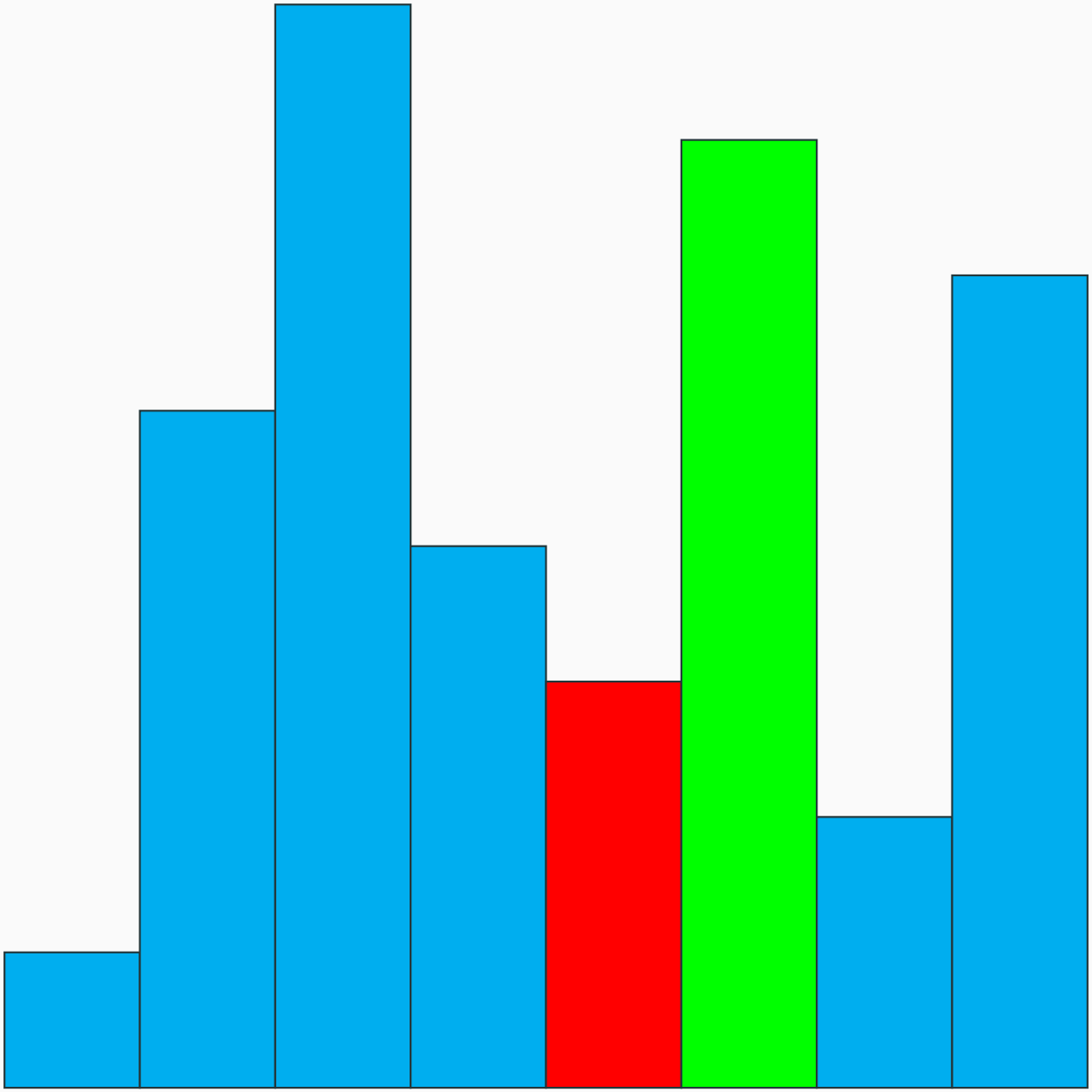
Exemple



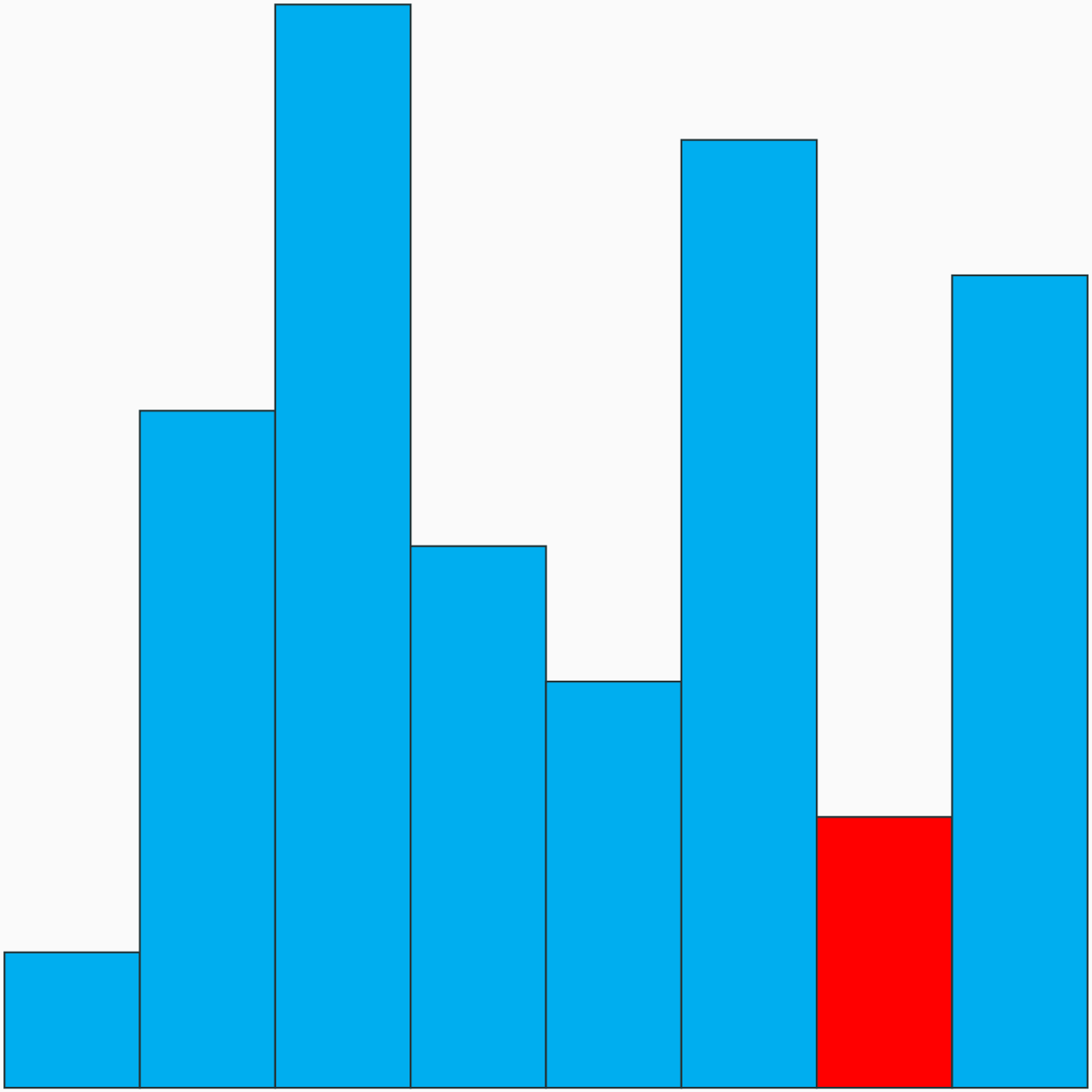
Exemple



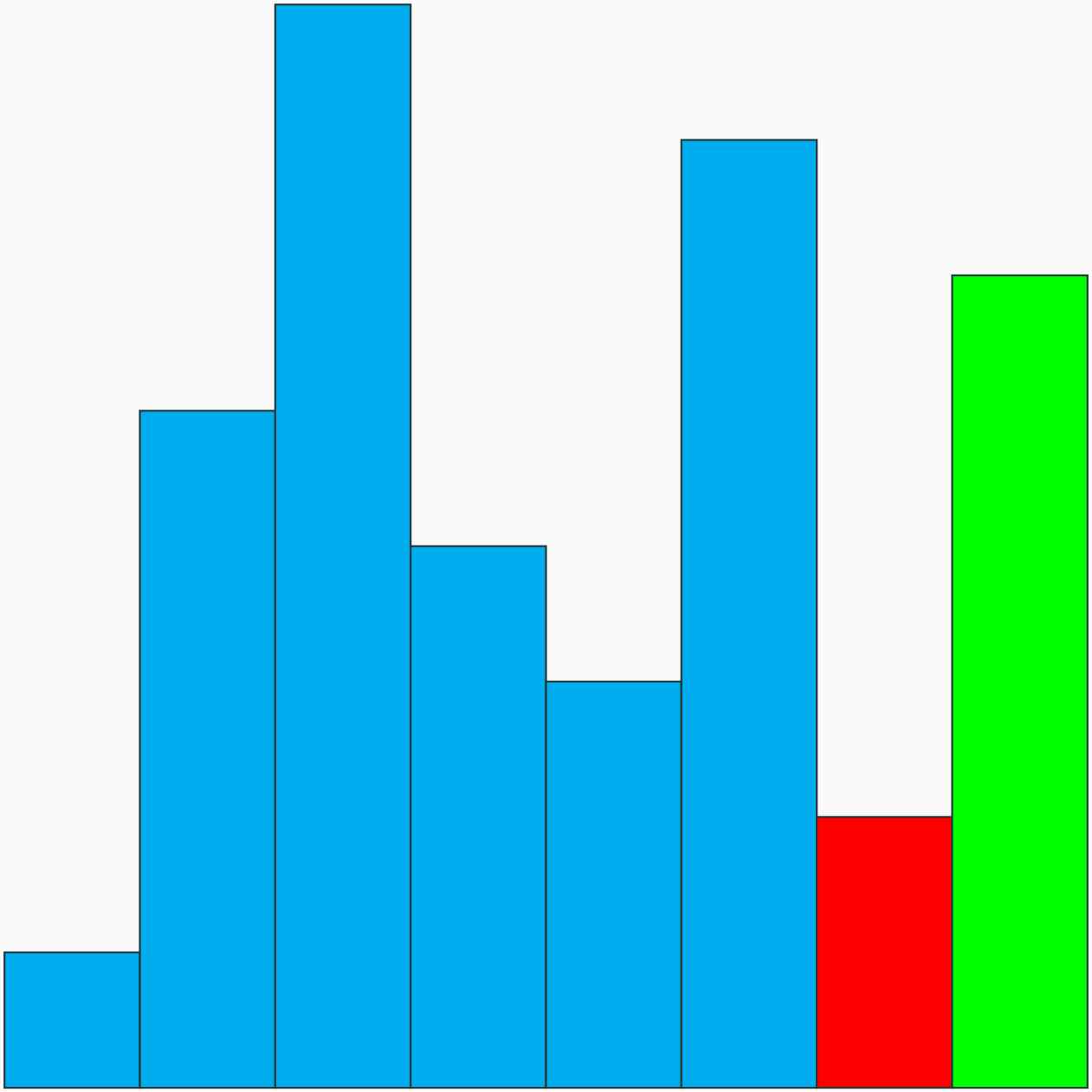
Exemple



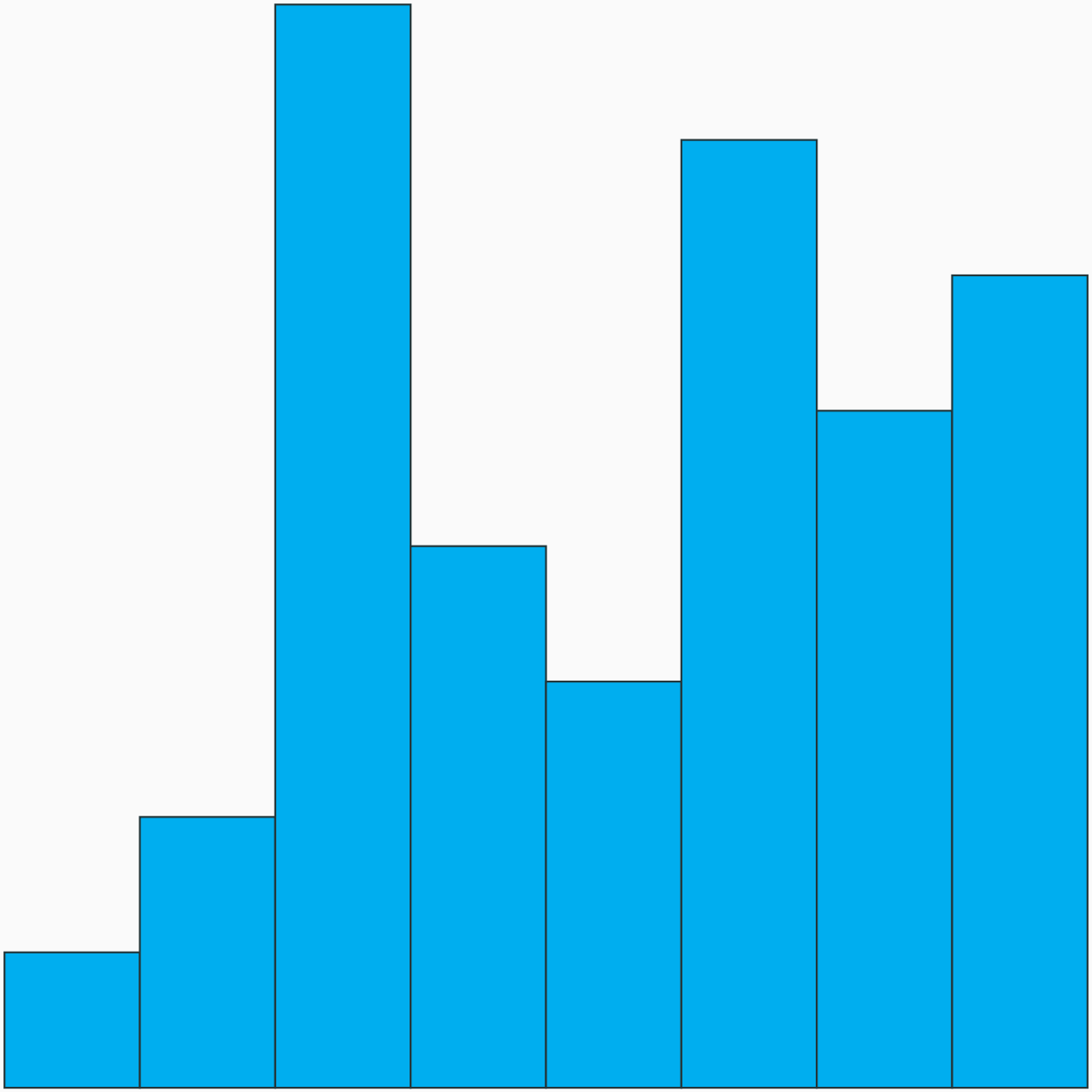
Exemple



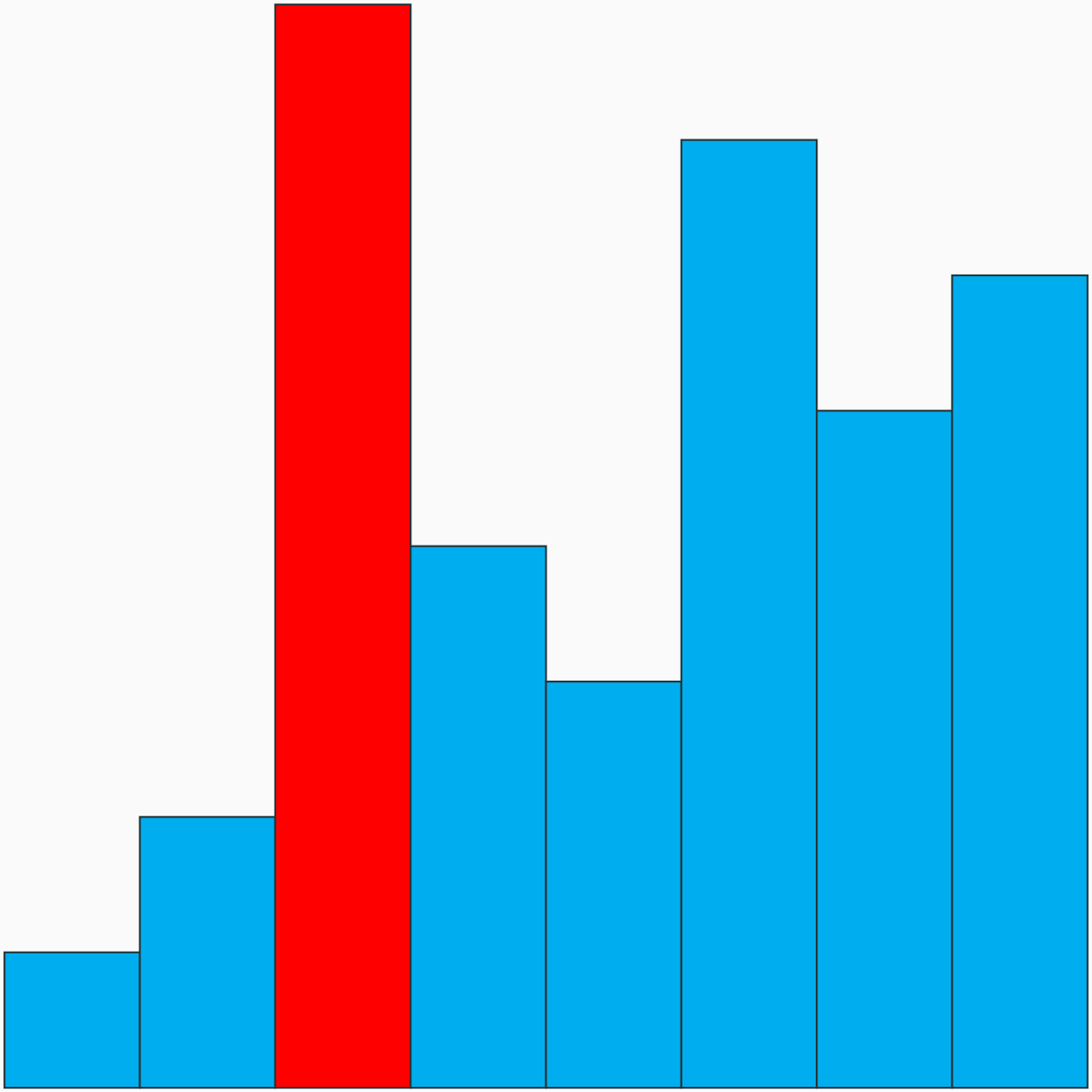
Exemple



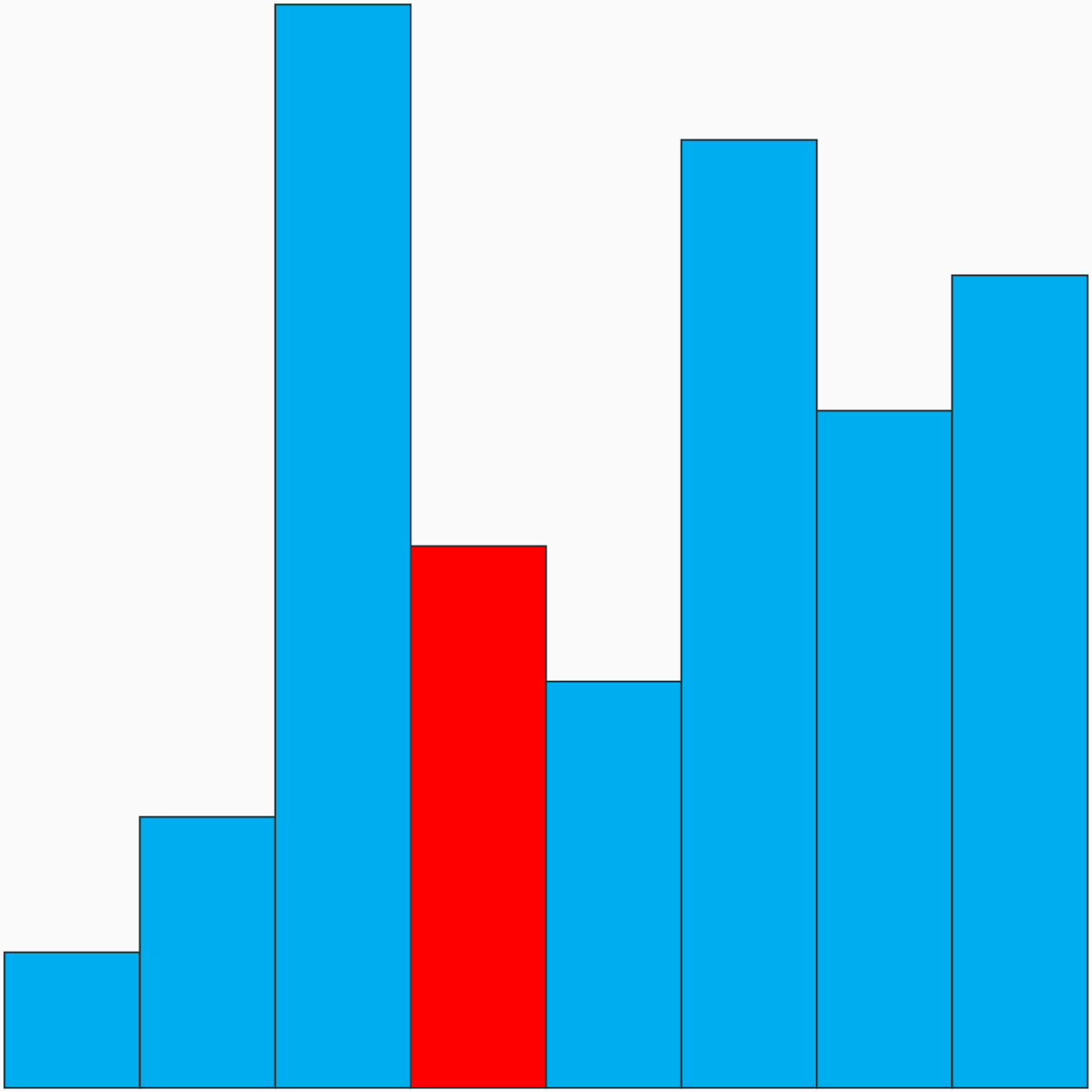
Exemple



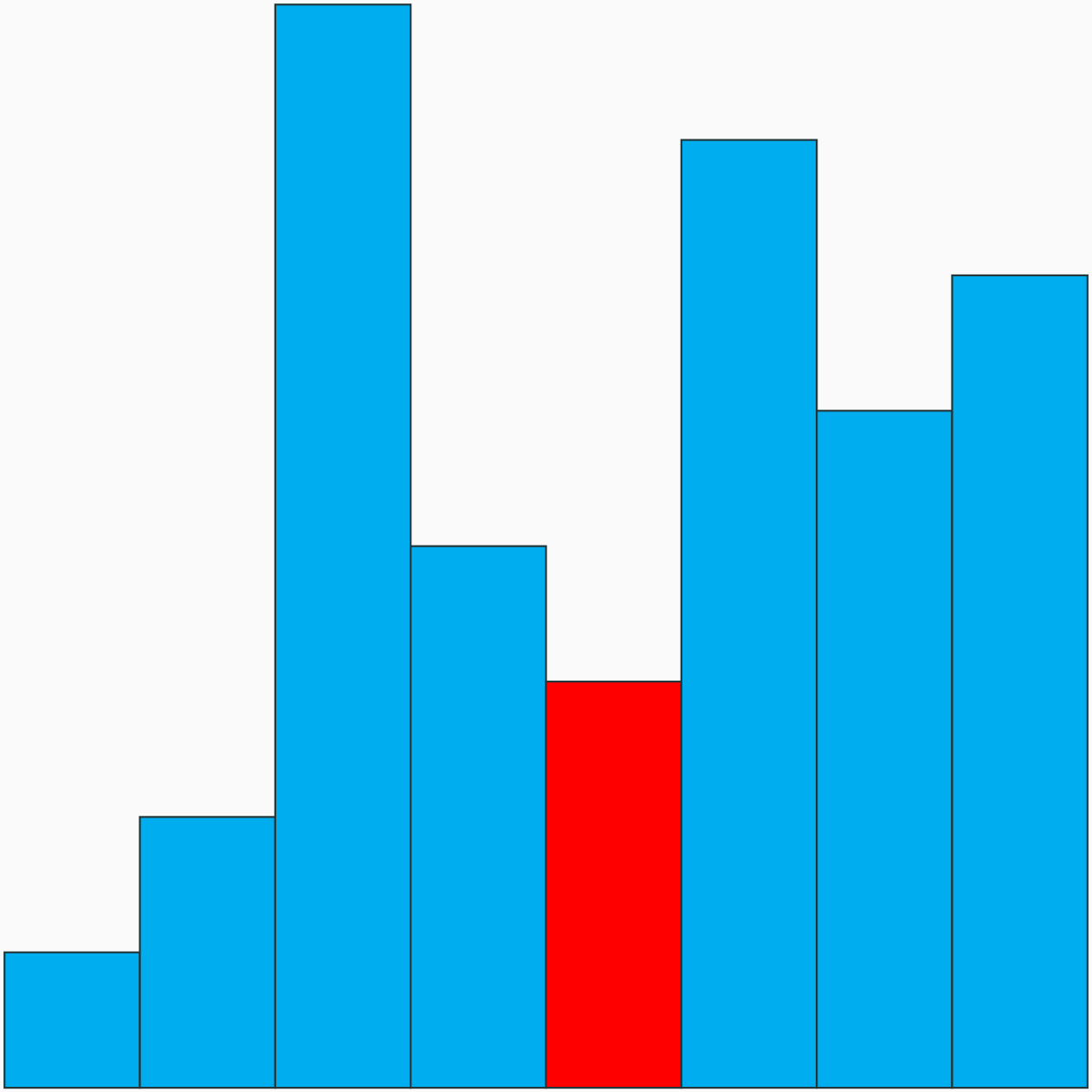
Exemple



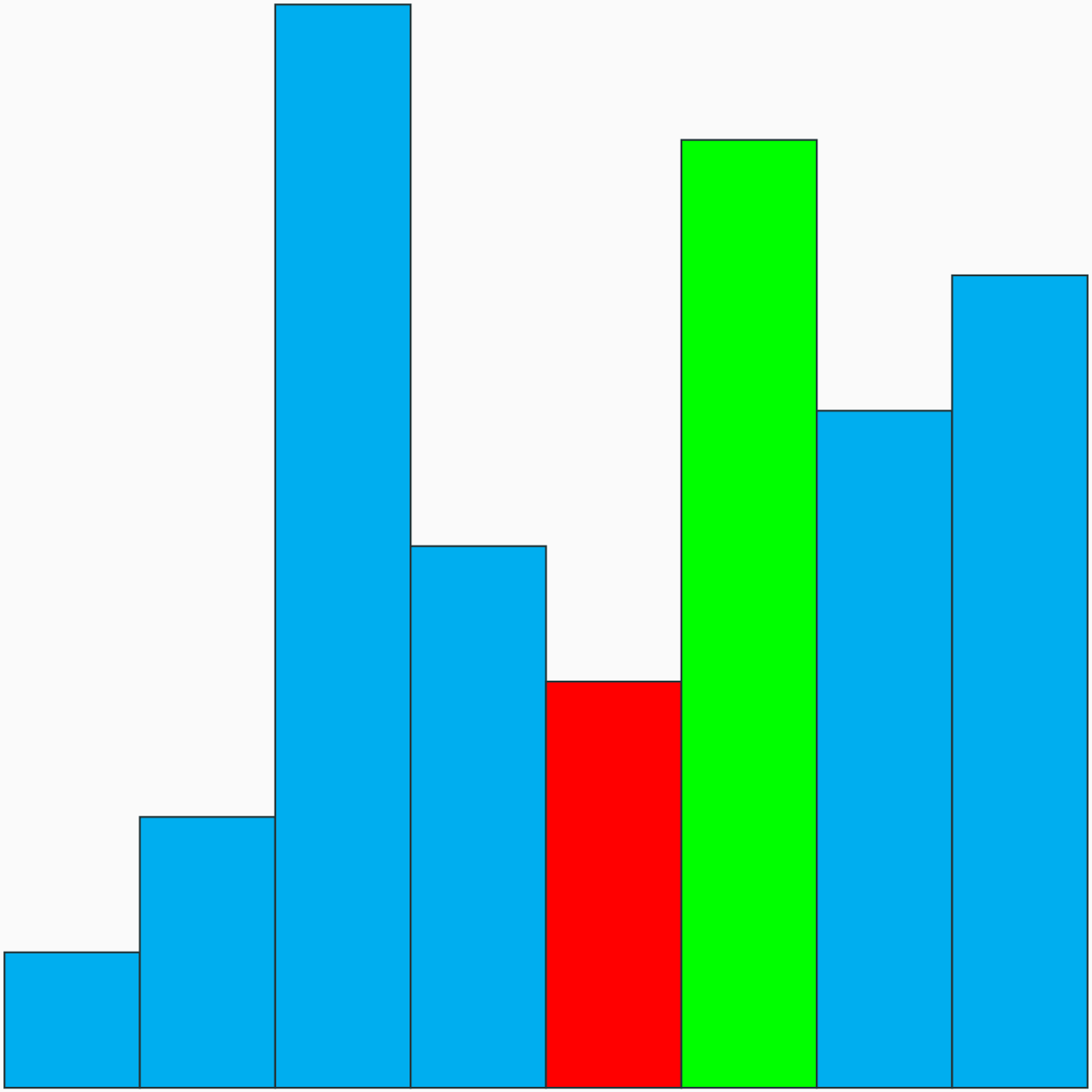
Exemple



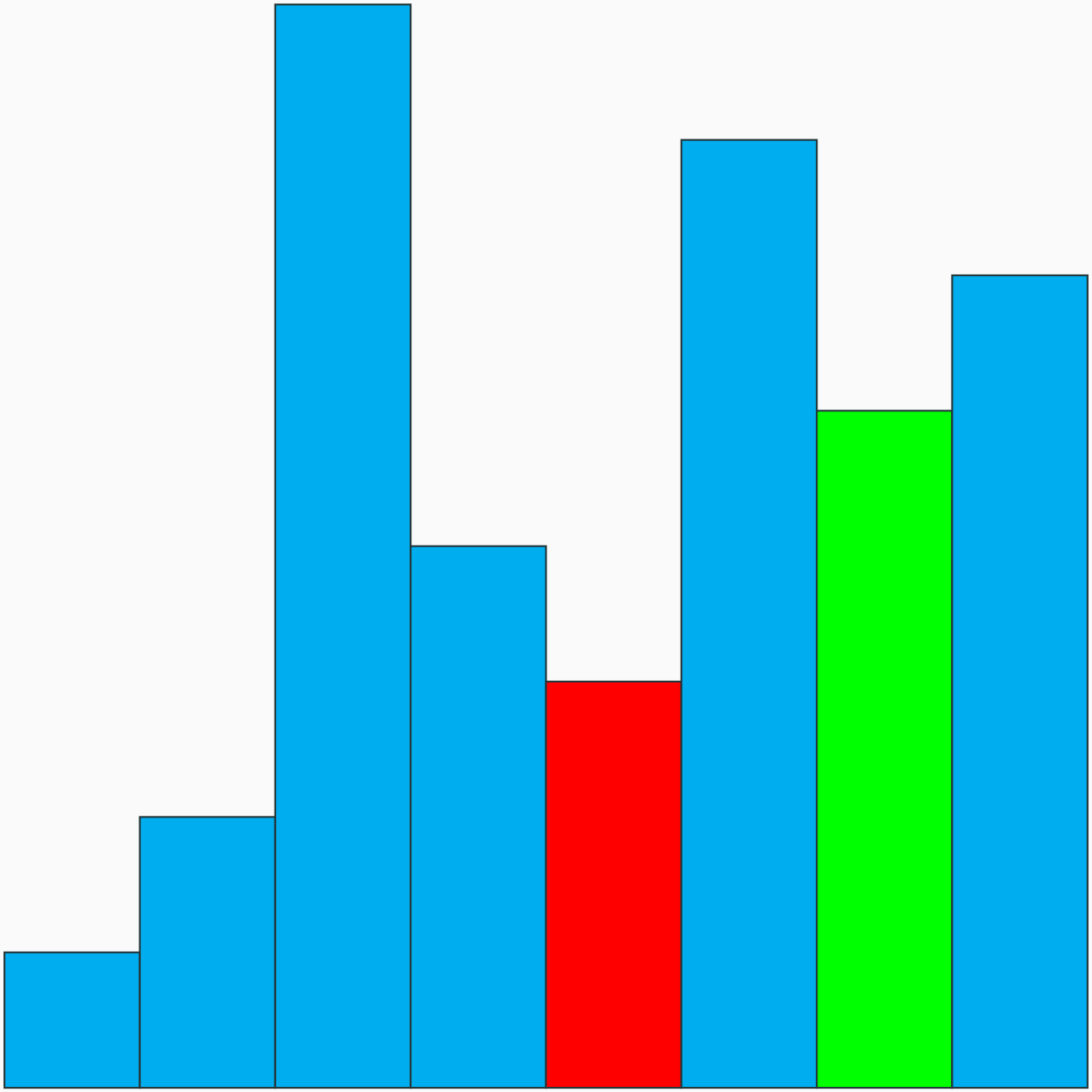
Exemple



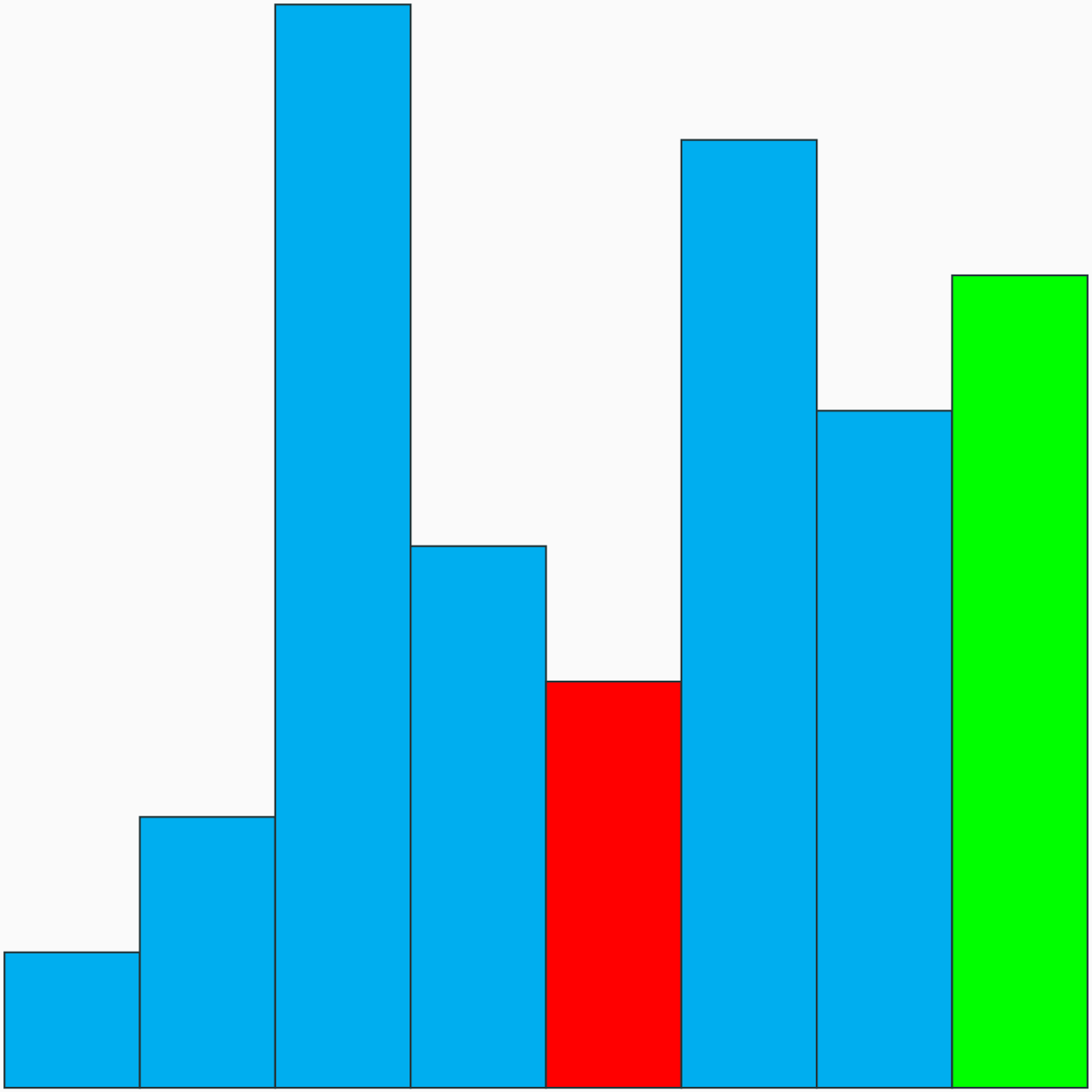
Exemple



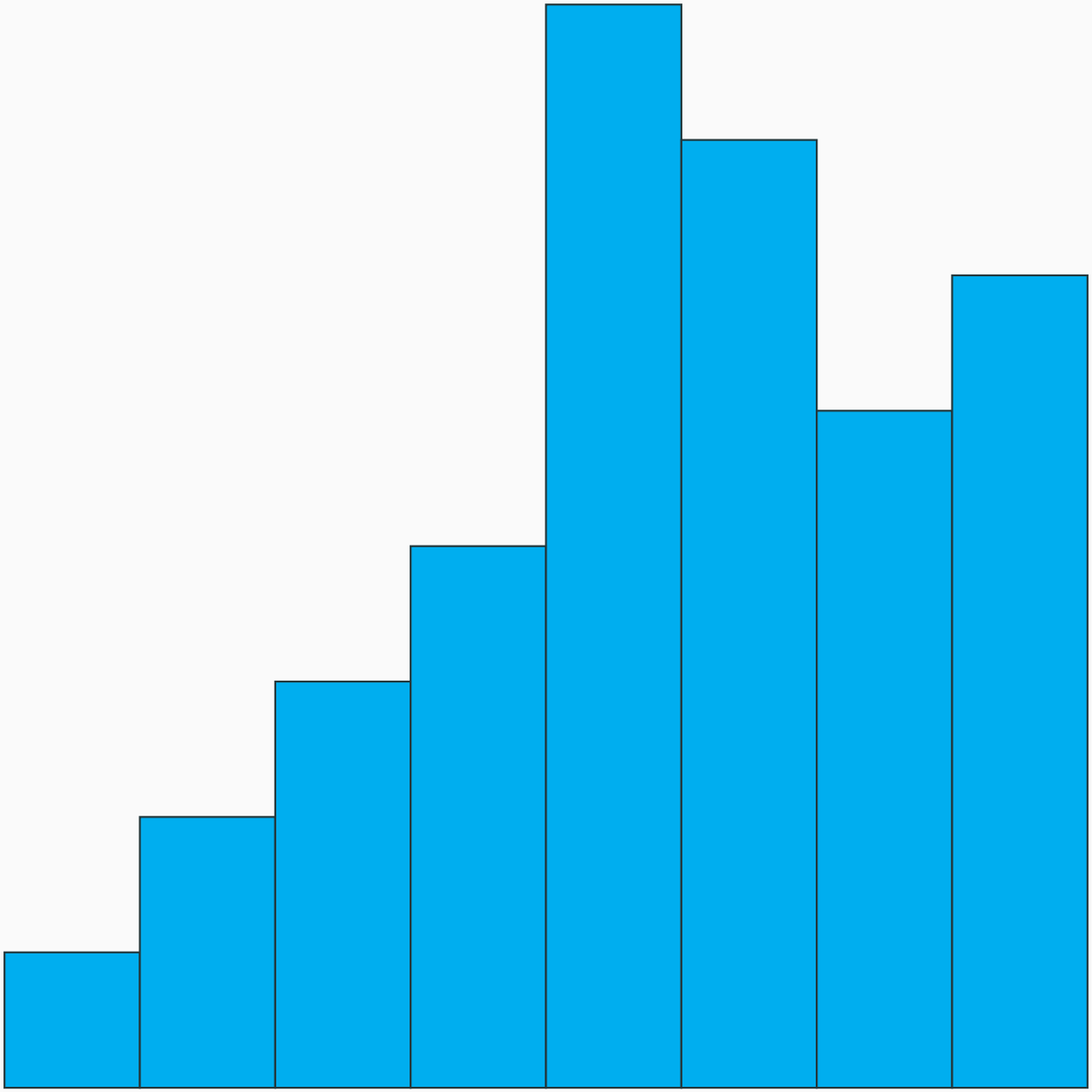
Exemple



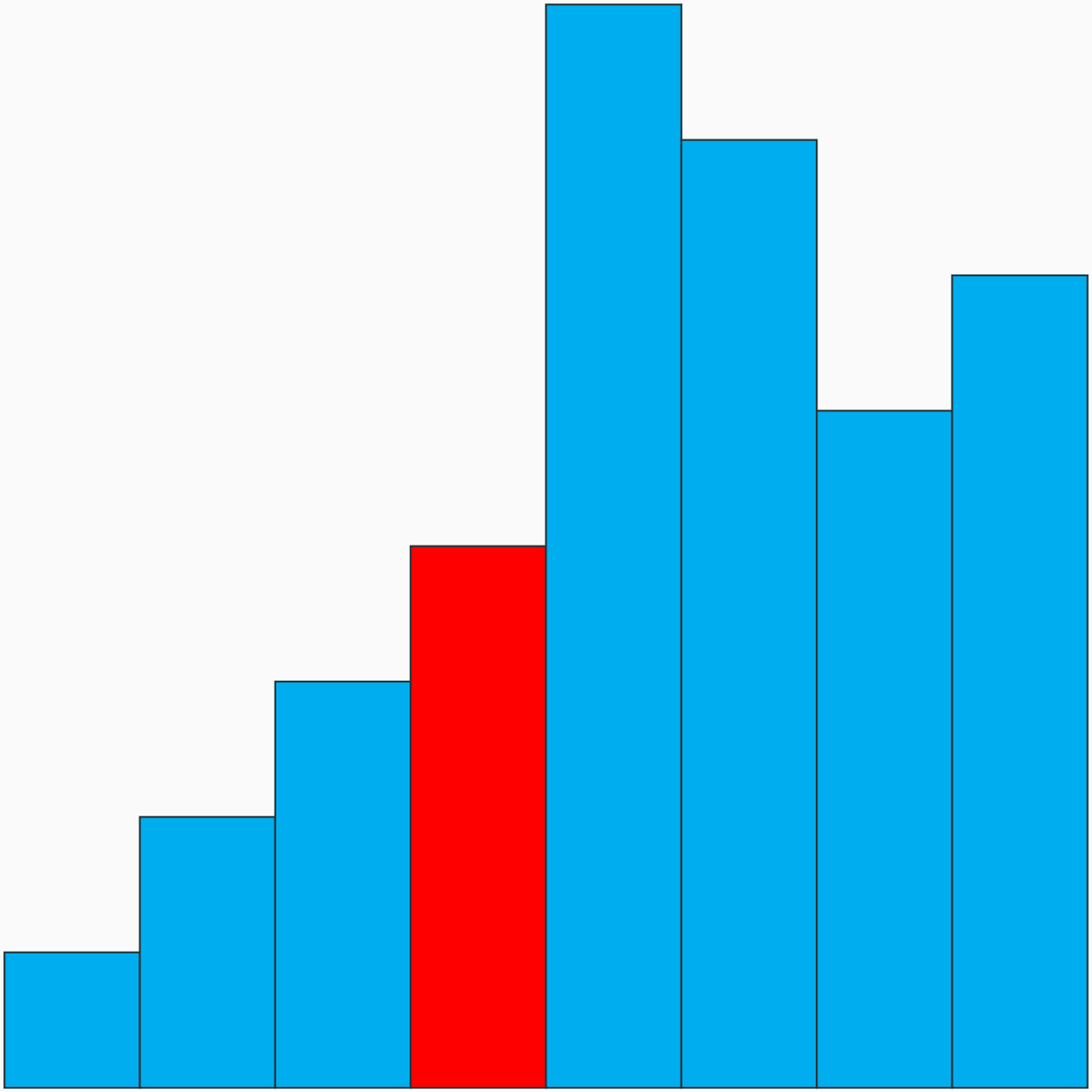
Exemple



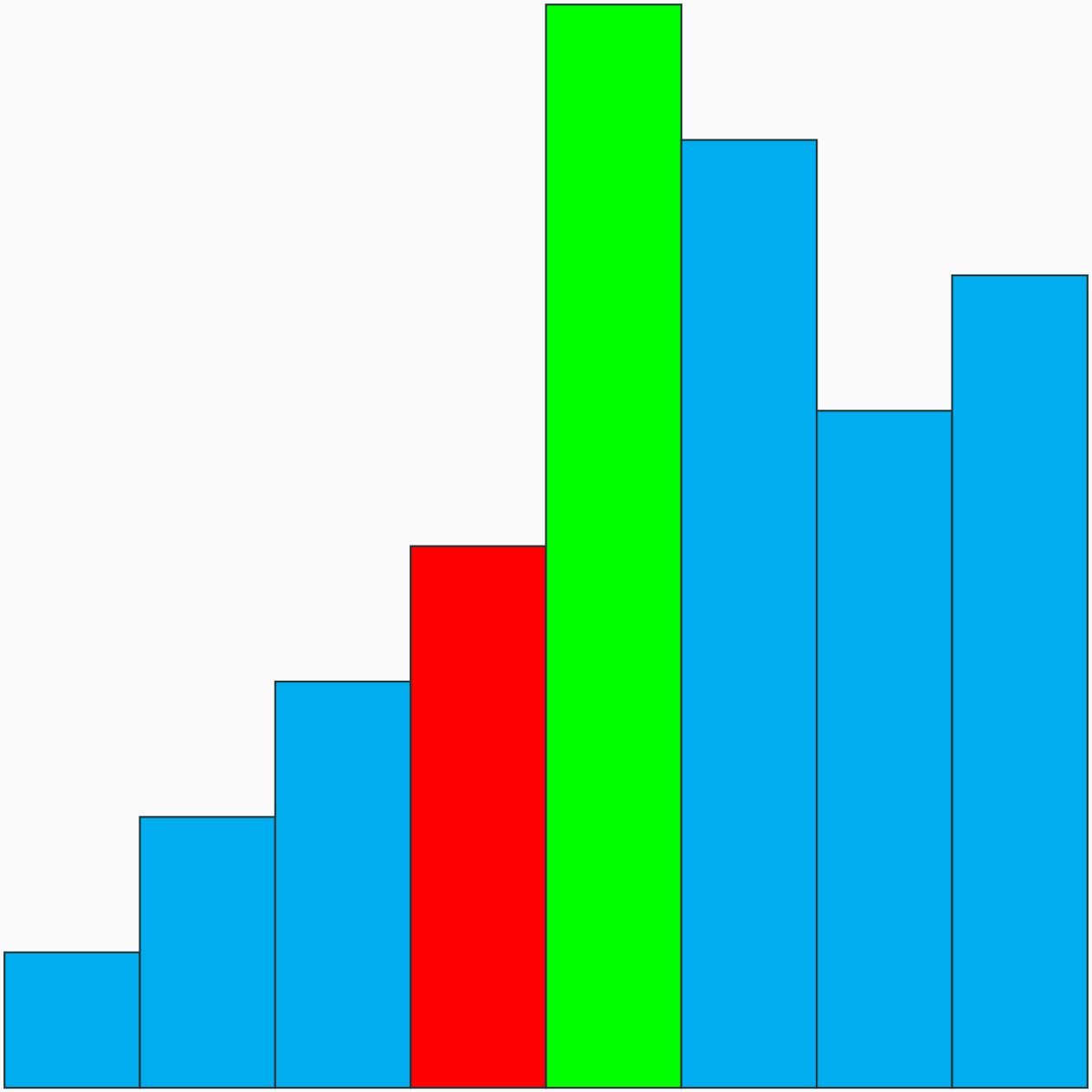
Exemple



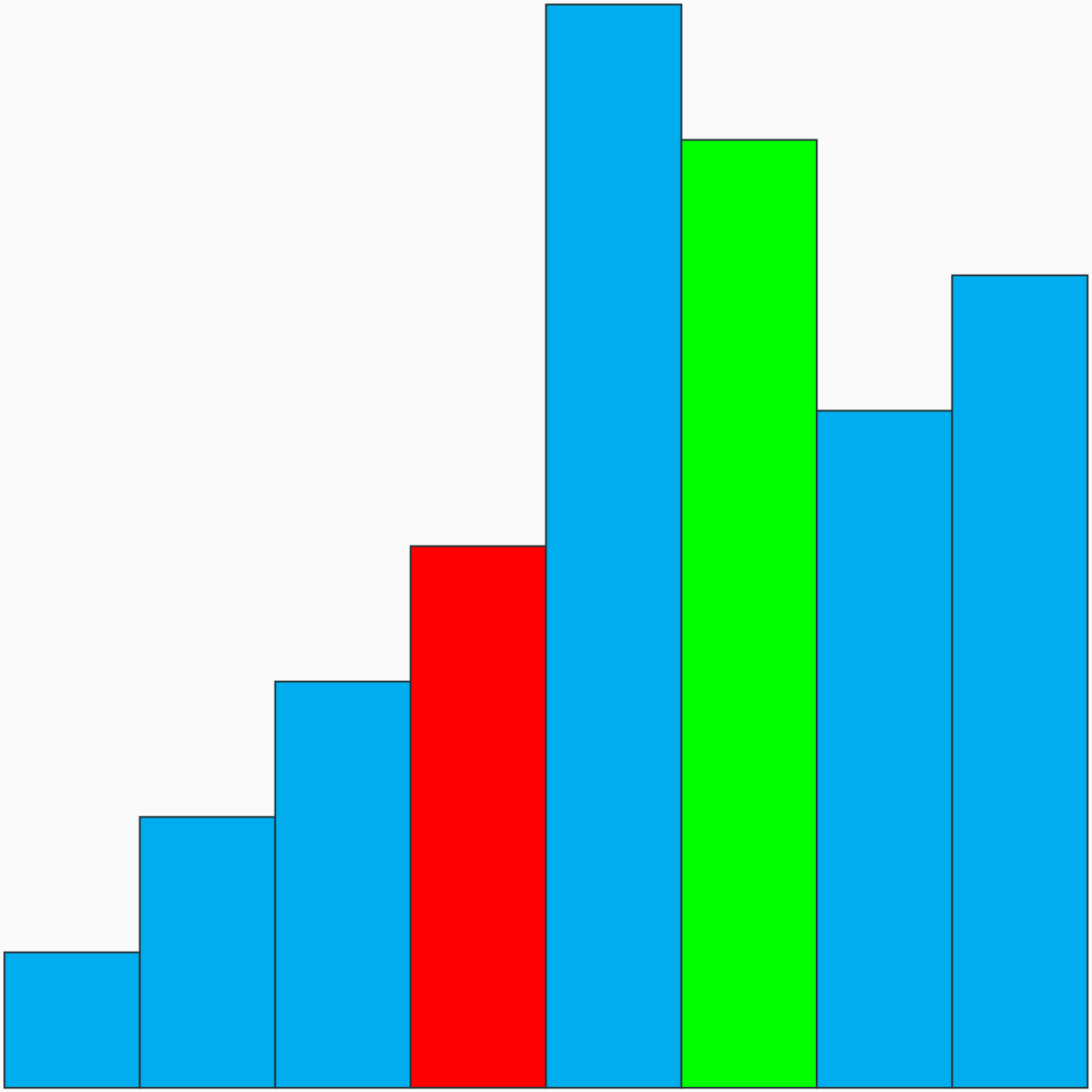
Exemple



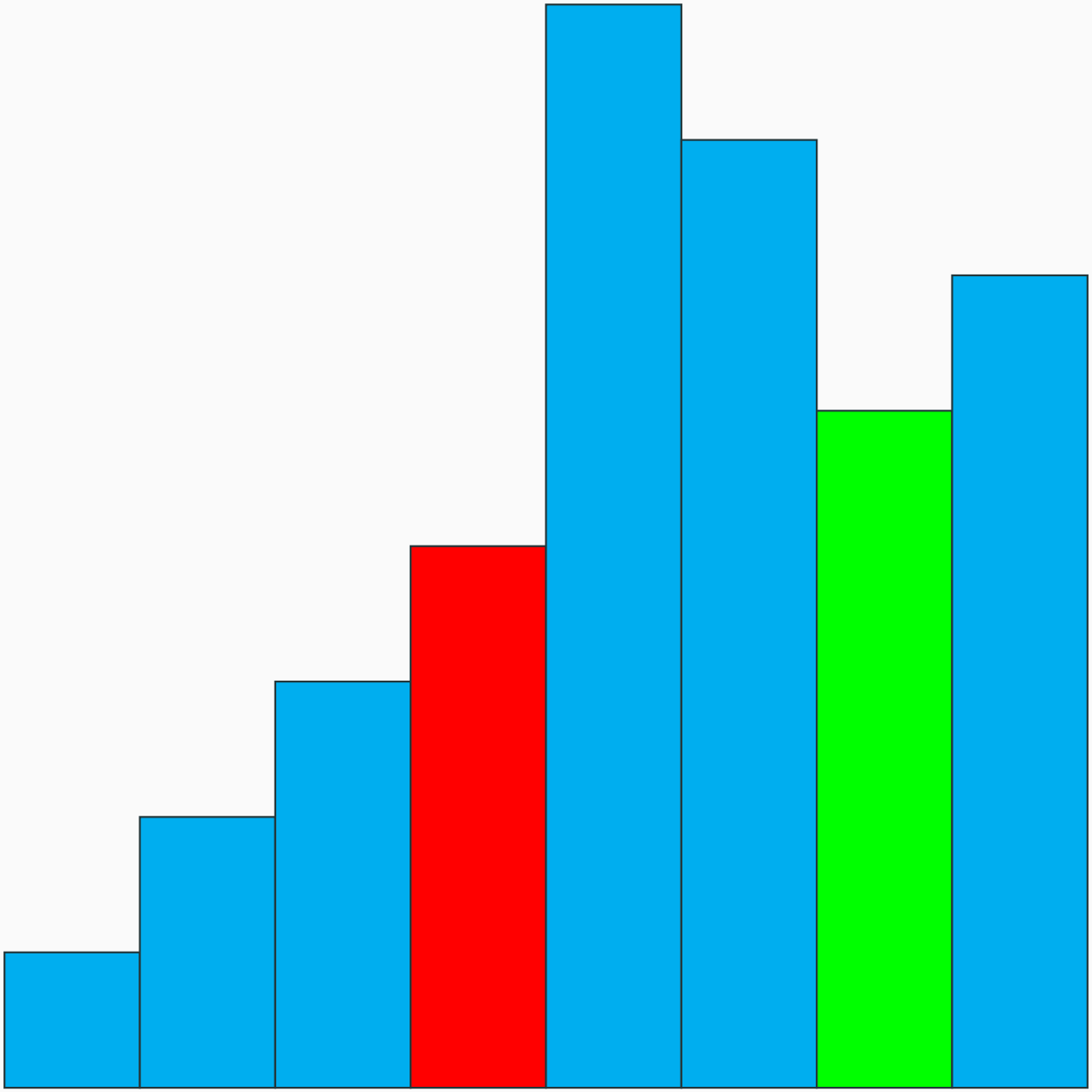
Exemple



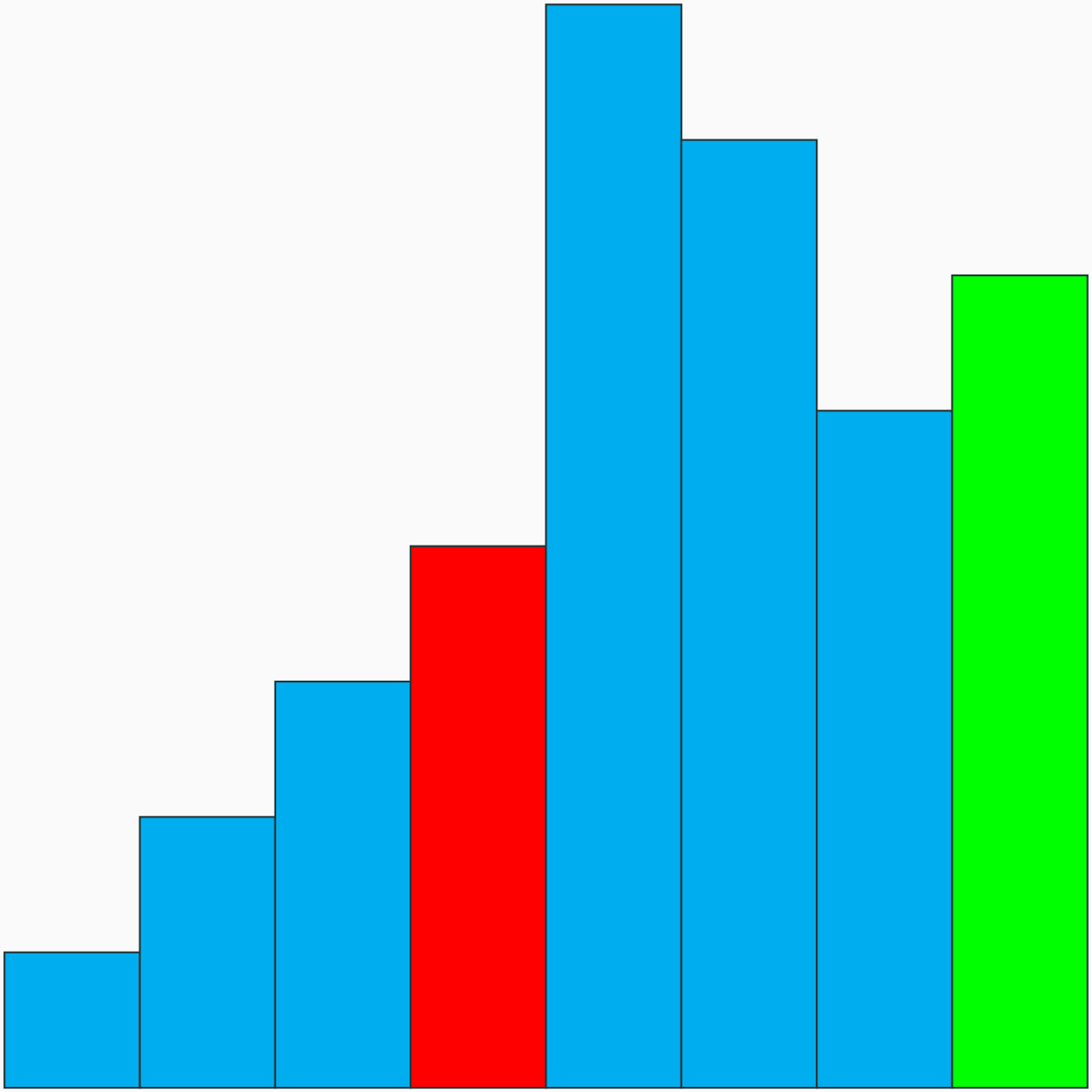
Exemple



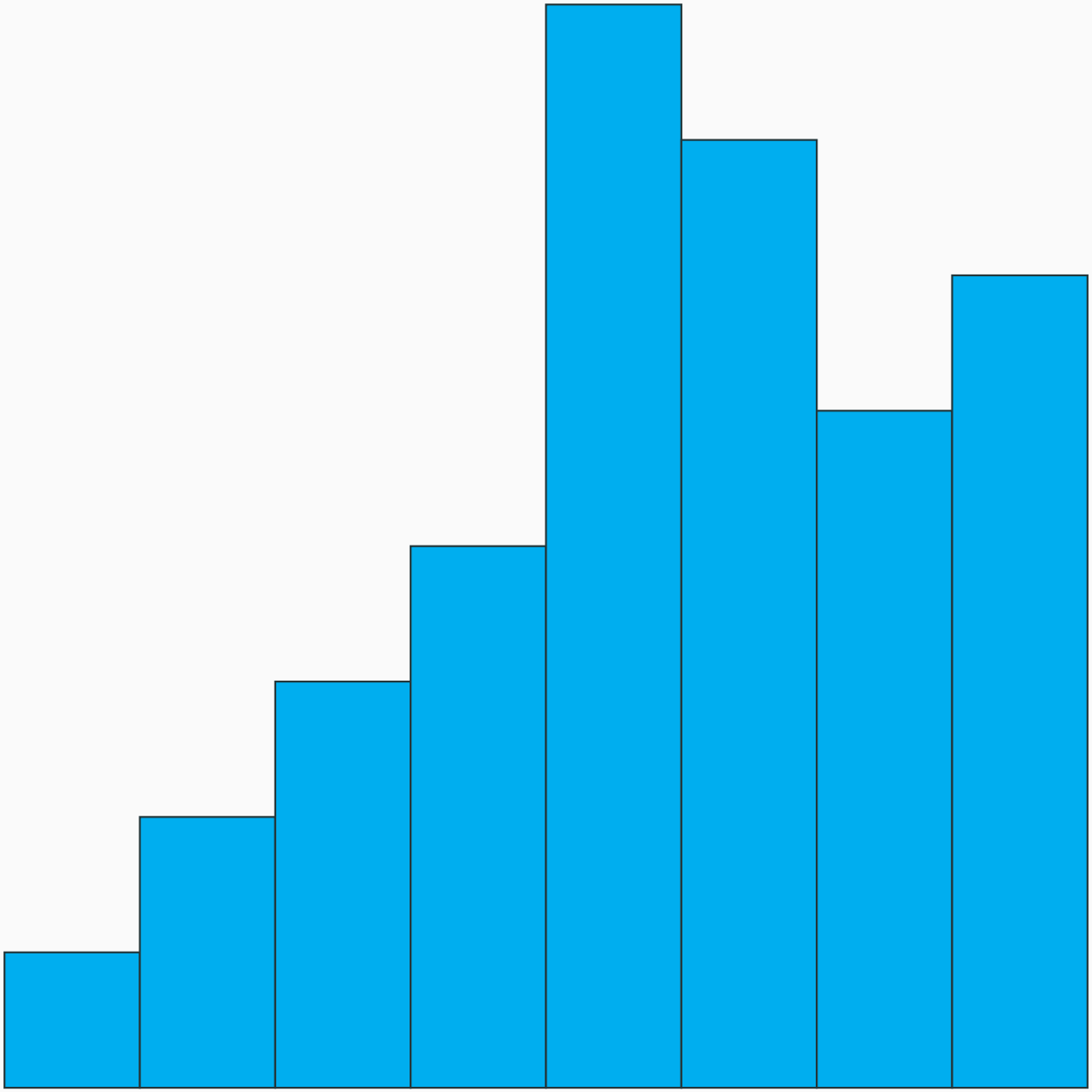
Exemple



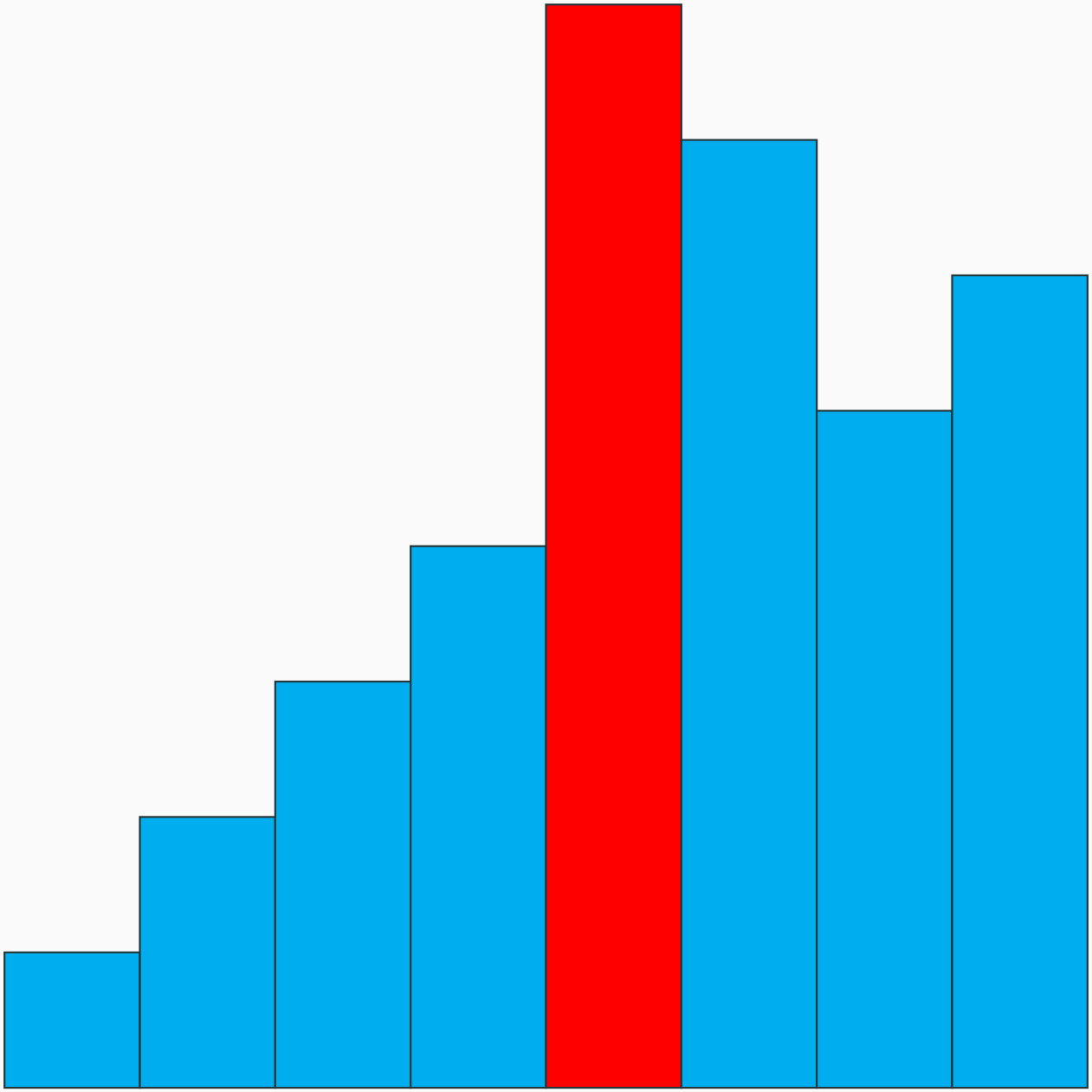
Exemple



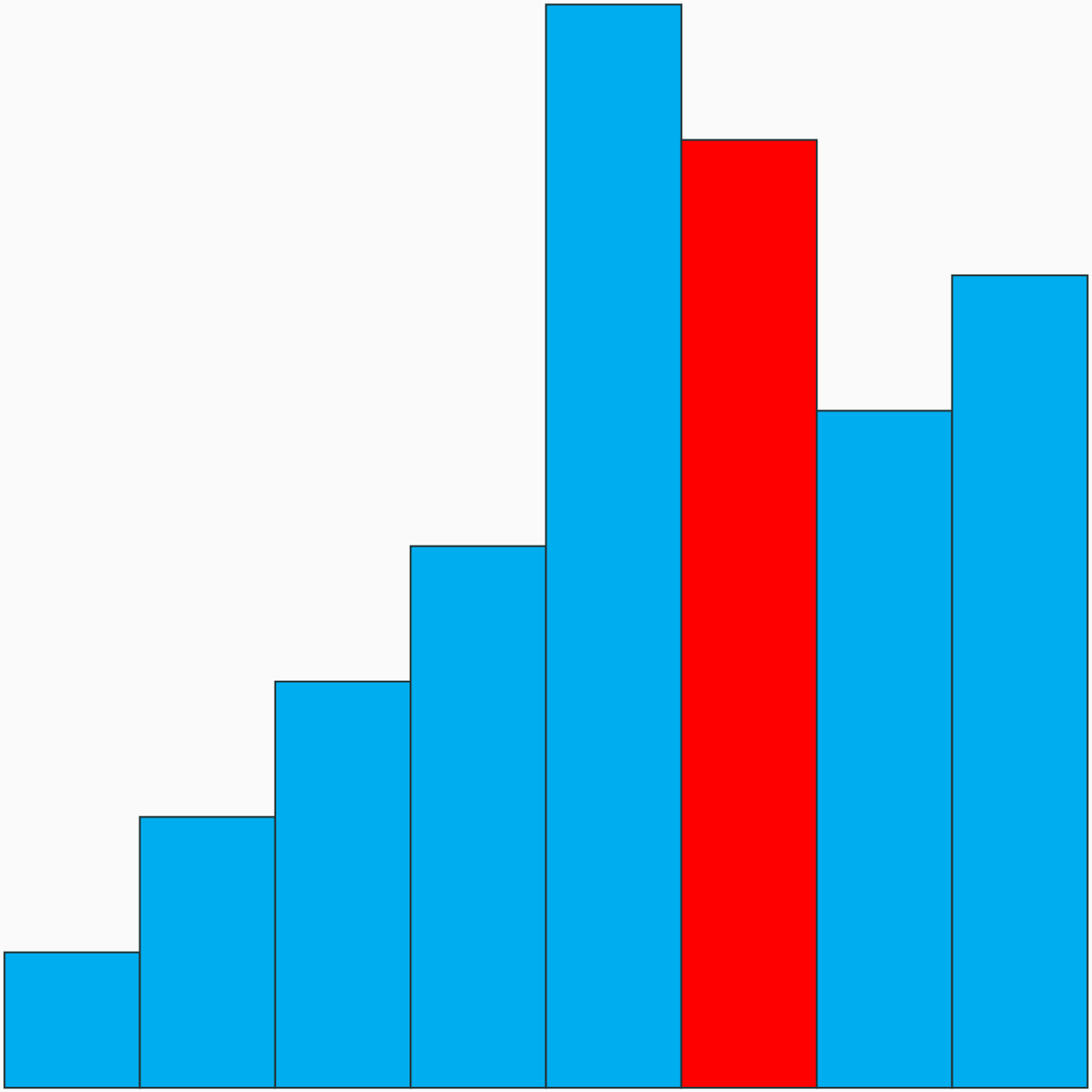
Exemple



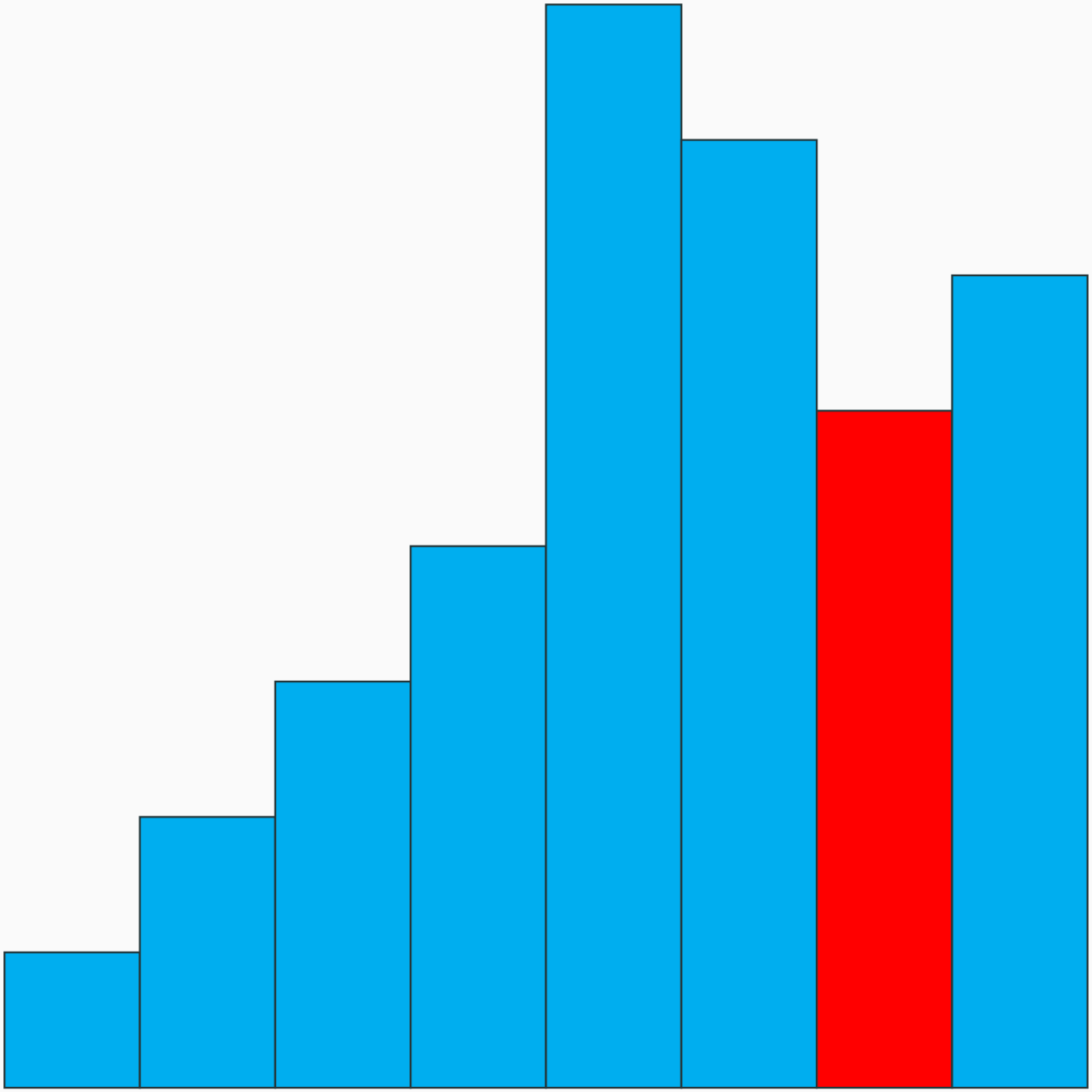
Exemple



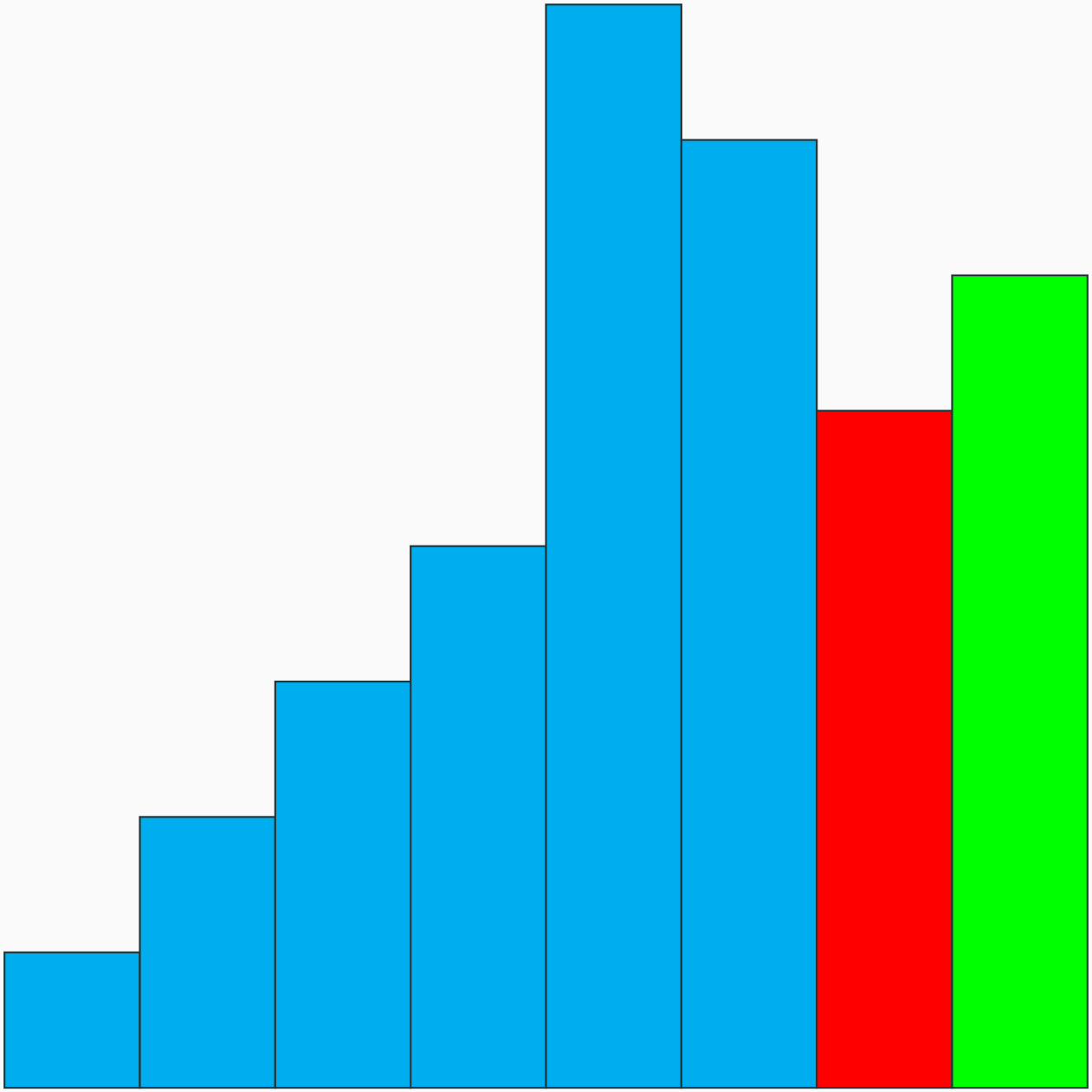
Exemple



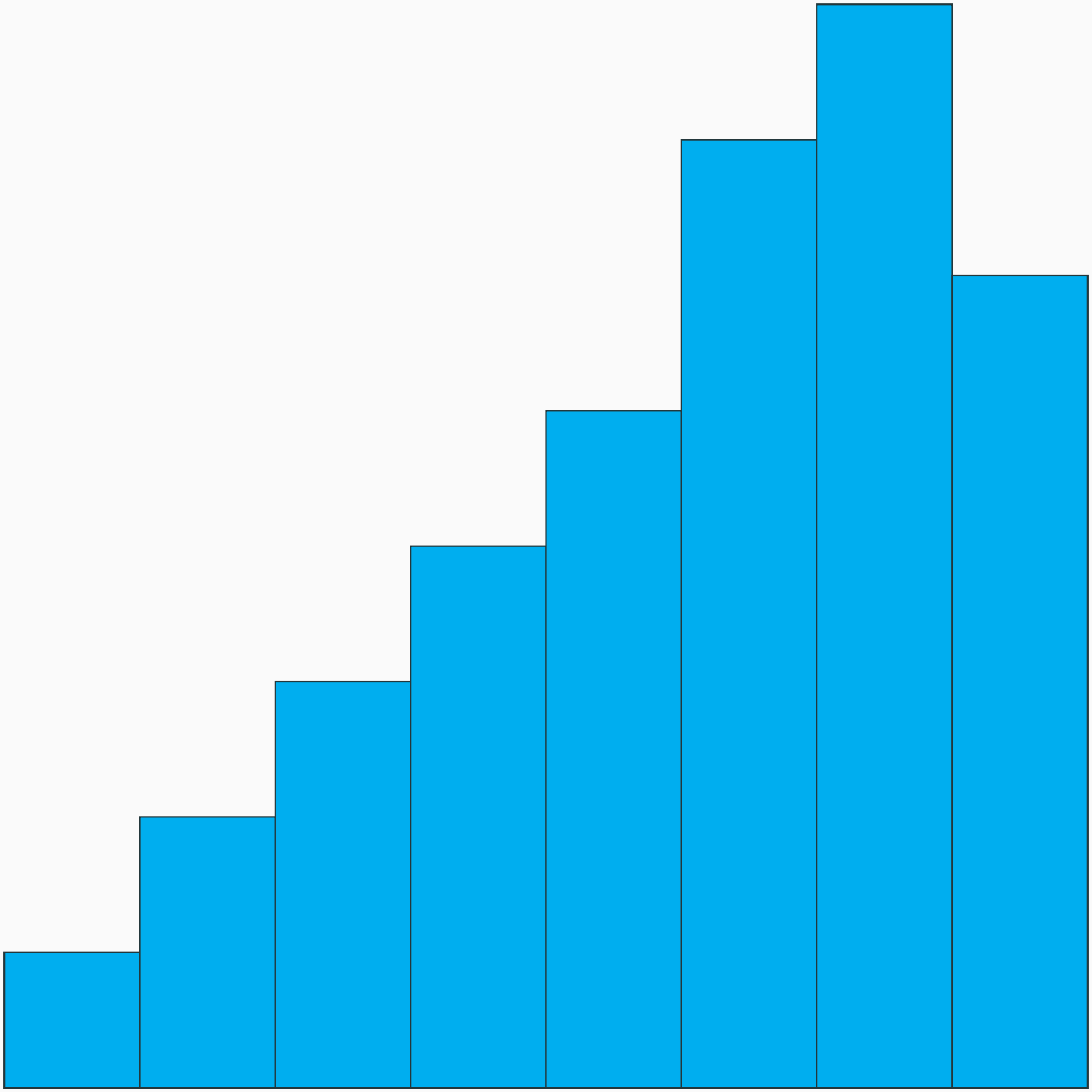
Exemple



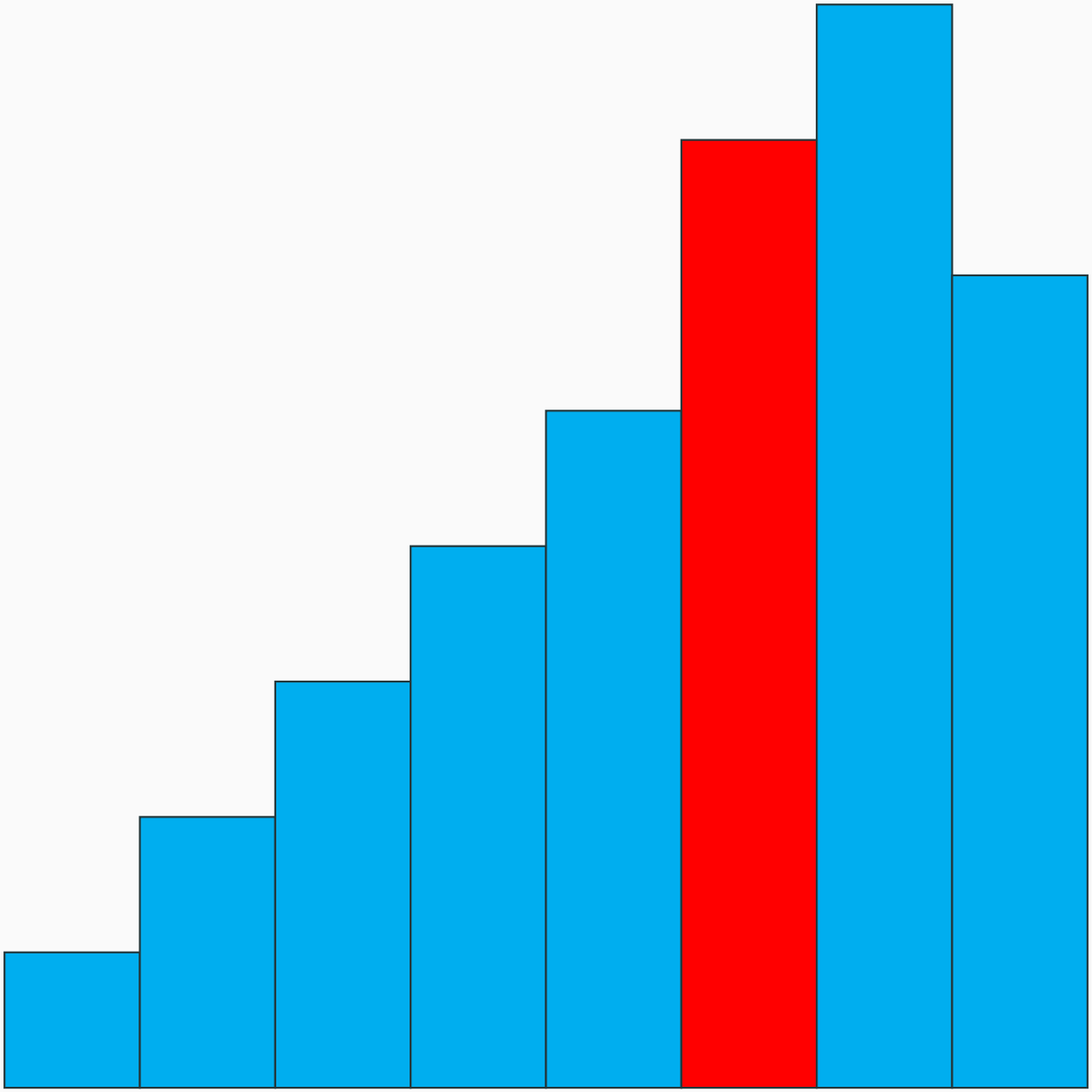
Exemple



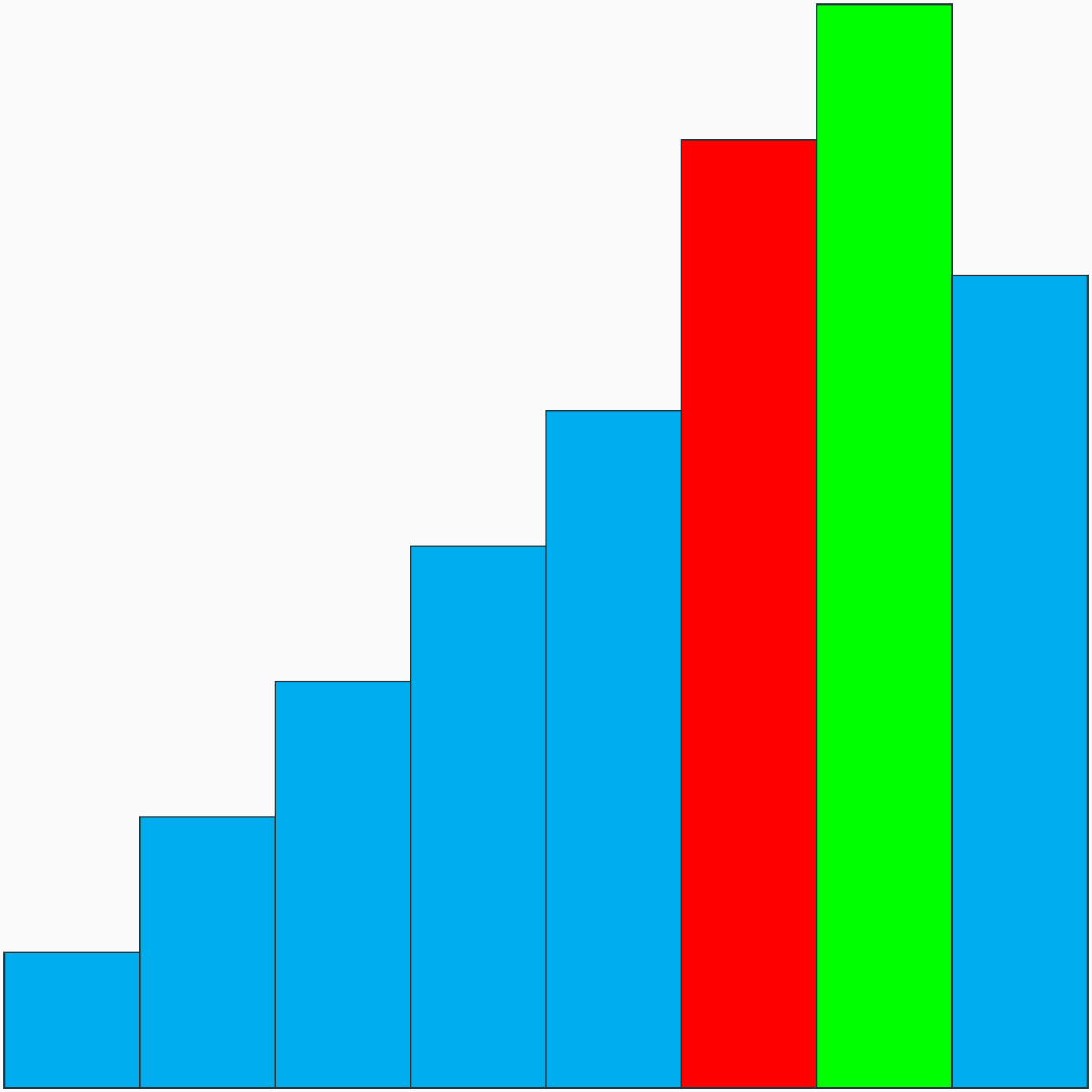
Exemple



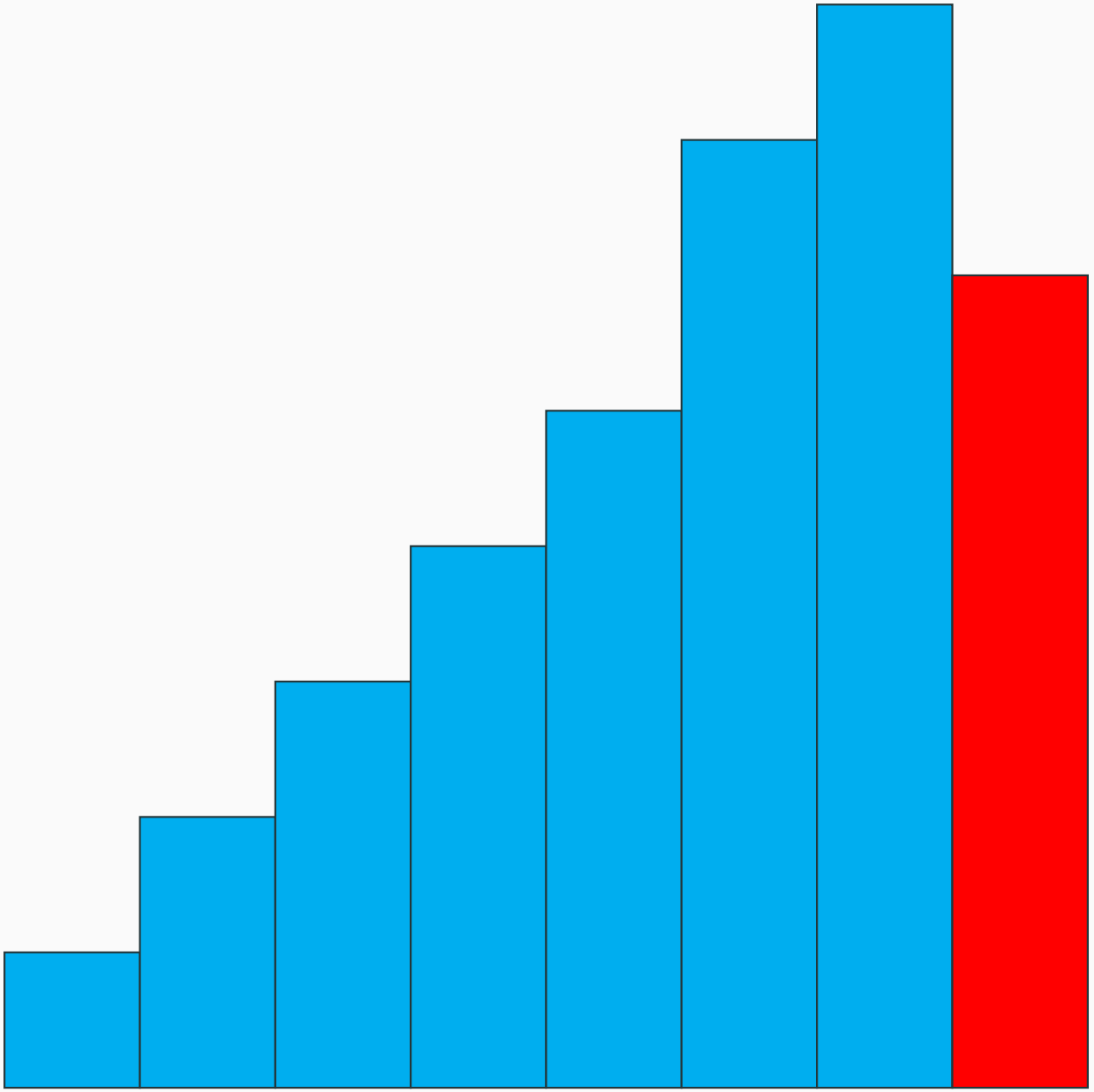
Exemple



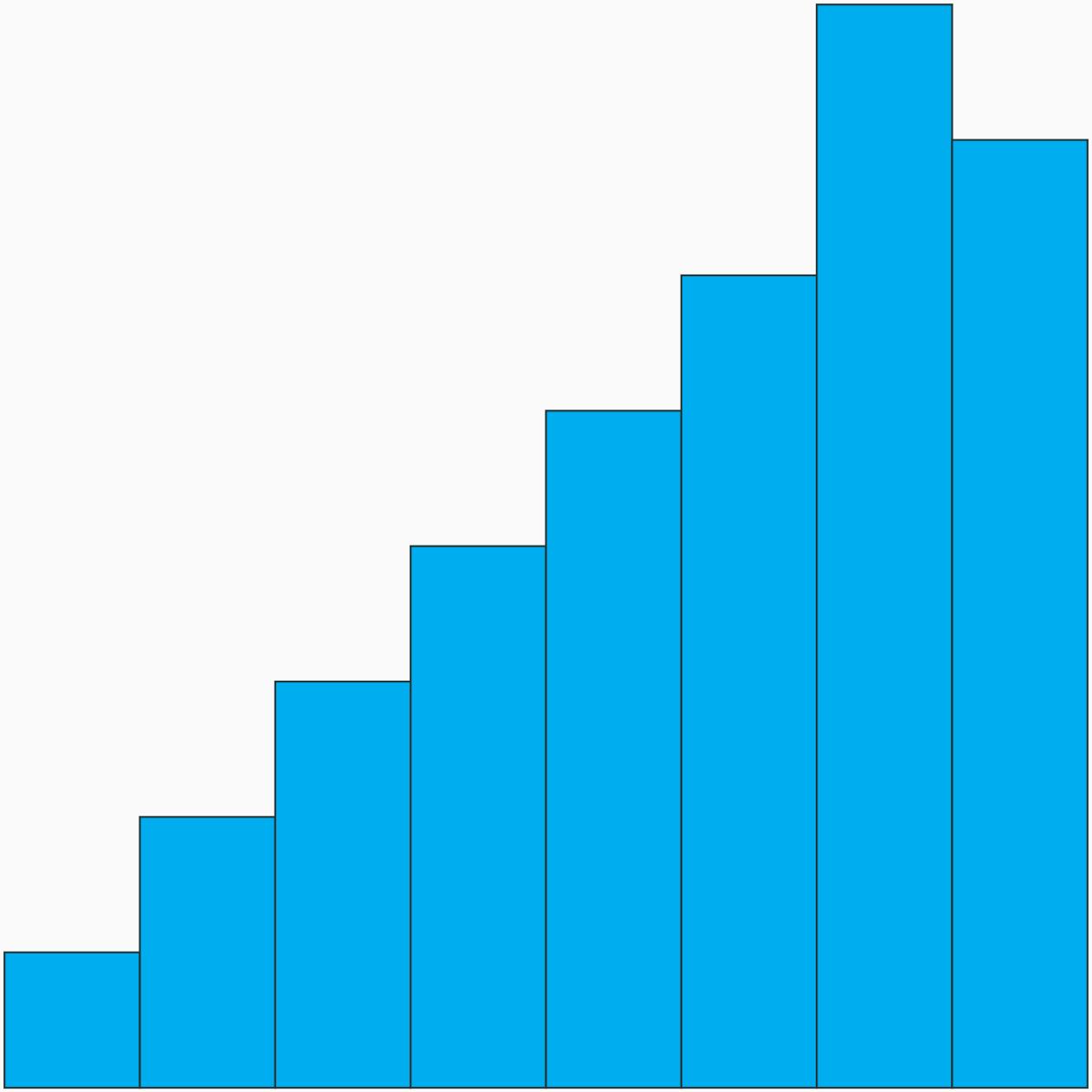
Exemple



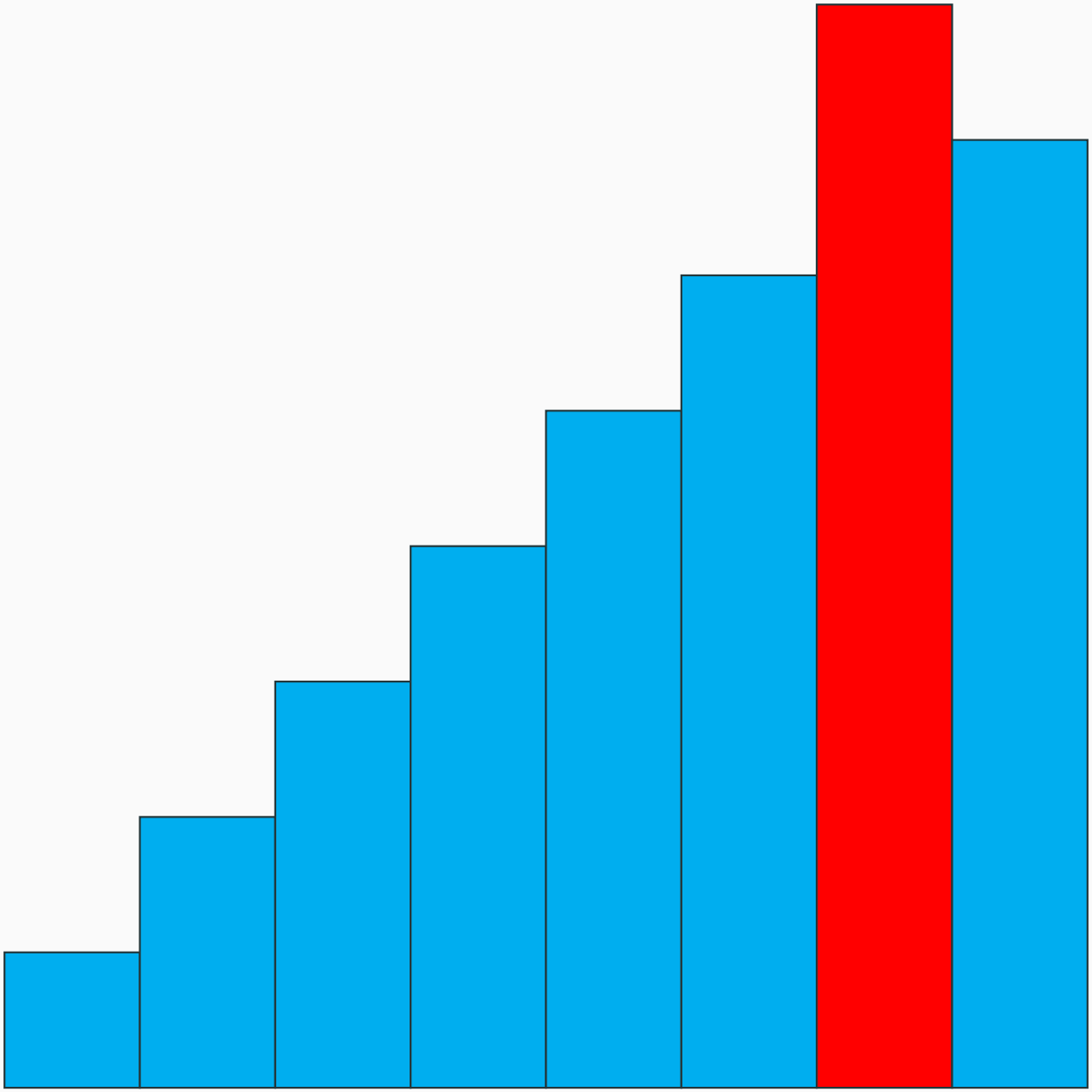
Exemple



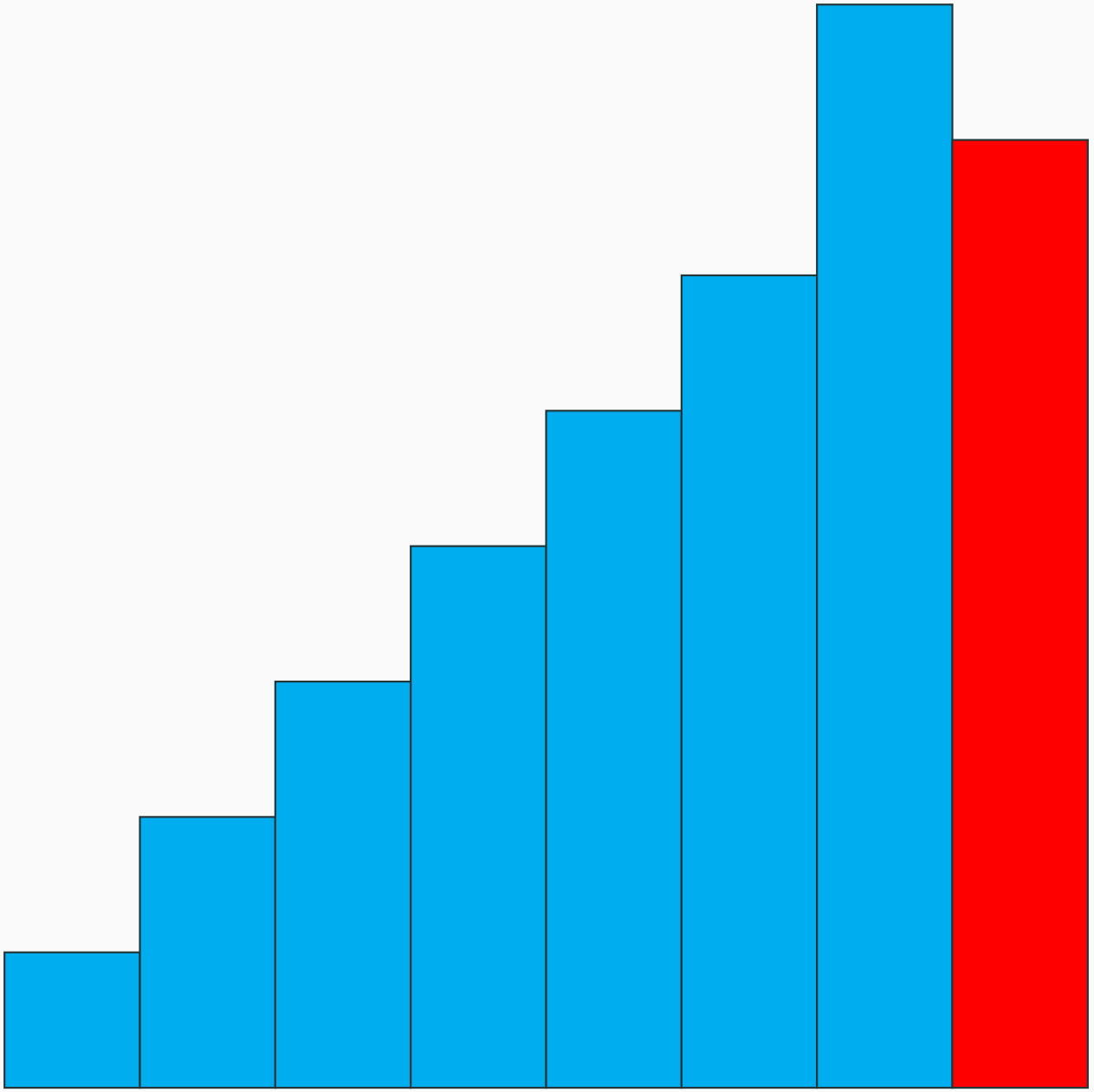
Exemple



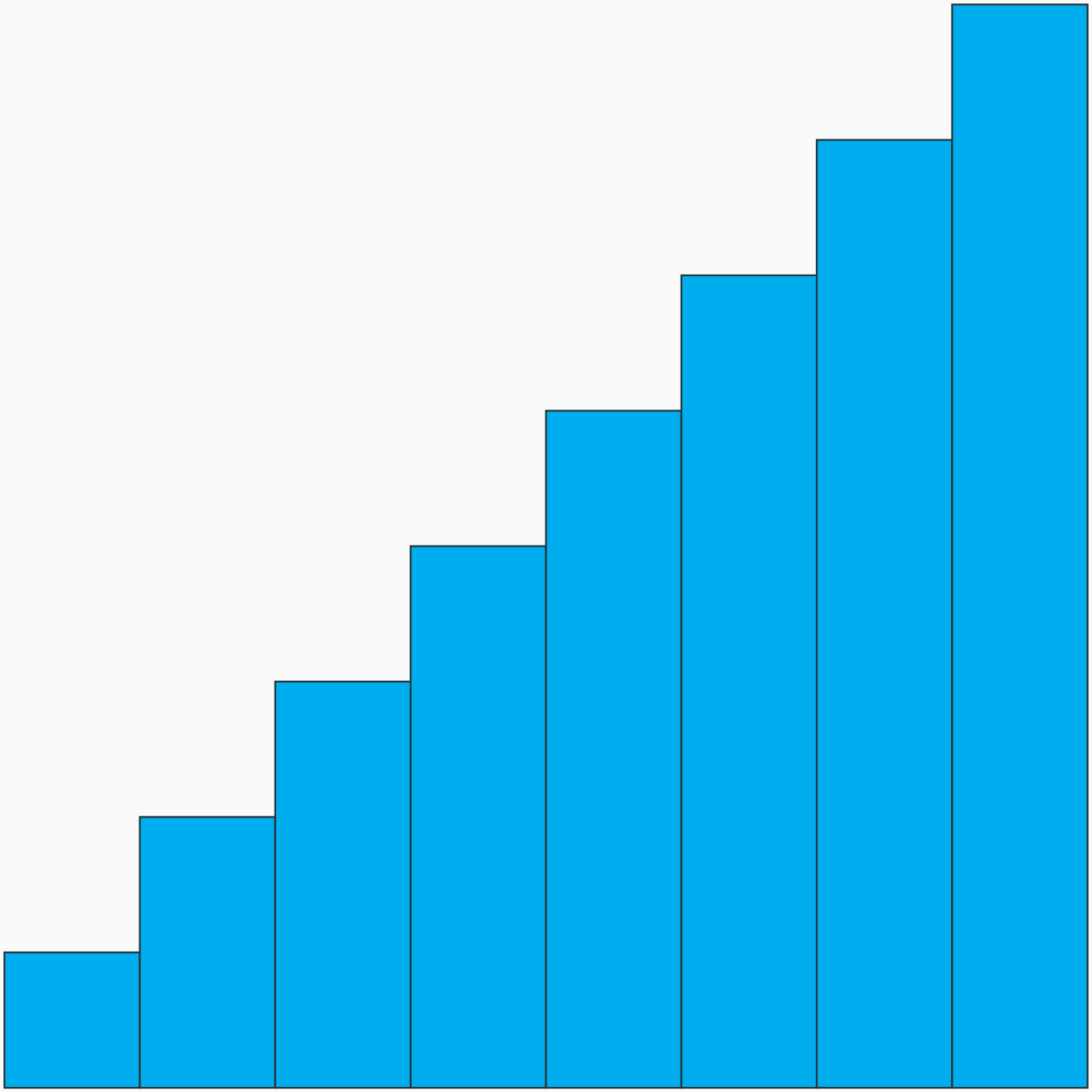
Exemple



Exemple



Exemple



Preuve de correction du tri par sélection

La boucle `for` **interne** a pour effet de positionner la variable `i_min` à l'indice de l'élément minimal du tableau entre les indices i et $n-1$.

Ainsi, un passage dans la boucle `for` **externe** positionne l'élément minimal du tableau entre les indices i et $n-1$ en position i.

Cette boucle `for` principale possède l'invariant suivant :

$\mathrm{Inv}(i)$: Les éléments du tableau entre les indices 0 et $i-1$ sont triés dans l'ordre croissant, et sont plus petits que les autres.

Preuve de correction du tri par sélection

$\mathrm{Inv}(i)$: Les éléments du tableau entre les indices 0 et $i-1$ sont triés dans l'ordre croissant, et sont plus petits que les autres.

- Tout d'abord, $\mathrm{Inv}(0)$ est vrai : en effet, le sous tableau `t[0:0]` est vide.
- Clairement, si $\mathrm{Inv}(i)$ est vrai en haut du corps de la boucle, $Inv(i+1)$ est vrai en bas du corps de boucle : en effet, on positionne le plus petit élément de parmi `t.(i),...,t.(n-1)` en position i.

Preuve de correction du tri par sélection

Le compteur de boucle i prend toutes les valeurs de $[\![0, n-2]\!]$.

Par suite, l'invariant $\mathrm{Inv}(n-2+1) = \mathrm{Inv}(n-1)$ est vérifié en sortie de boucle, ce qui implique que les $n-1$ premiers éléments du tableau sont triés en sortie de boucle, et plus petits que l'autre élément du tableau, à savoir `t.(n-1)`.

Ainsi, le tableau est entièrement trié en sortie de boucle, donc en sortie de fonction, et le tri est **correct**.

Remarque

En toute rigueur, il faudrait exhiber un invariant de boucle pour la boucle `for` **interne**, celui-ci est plutôt évident et est marqué dans le code.

Tri par insertion

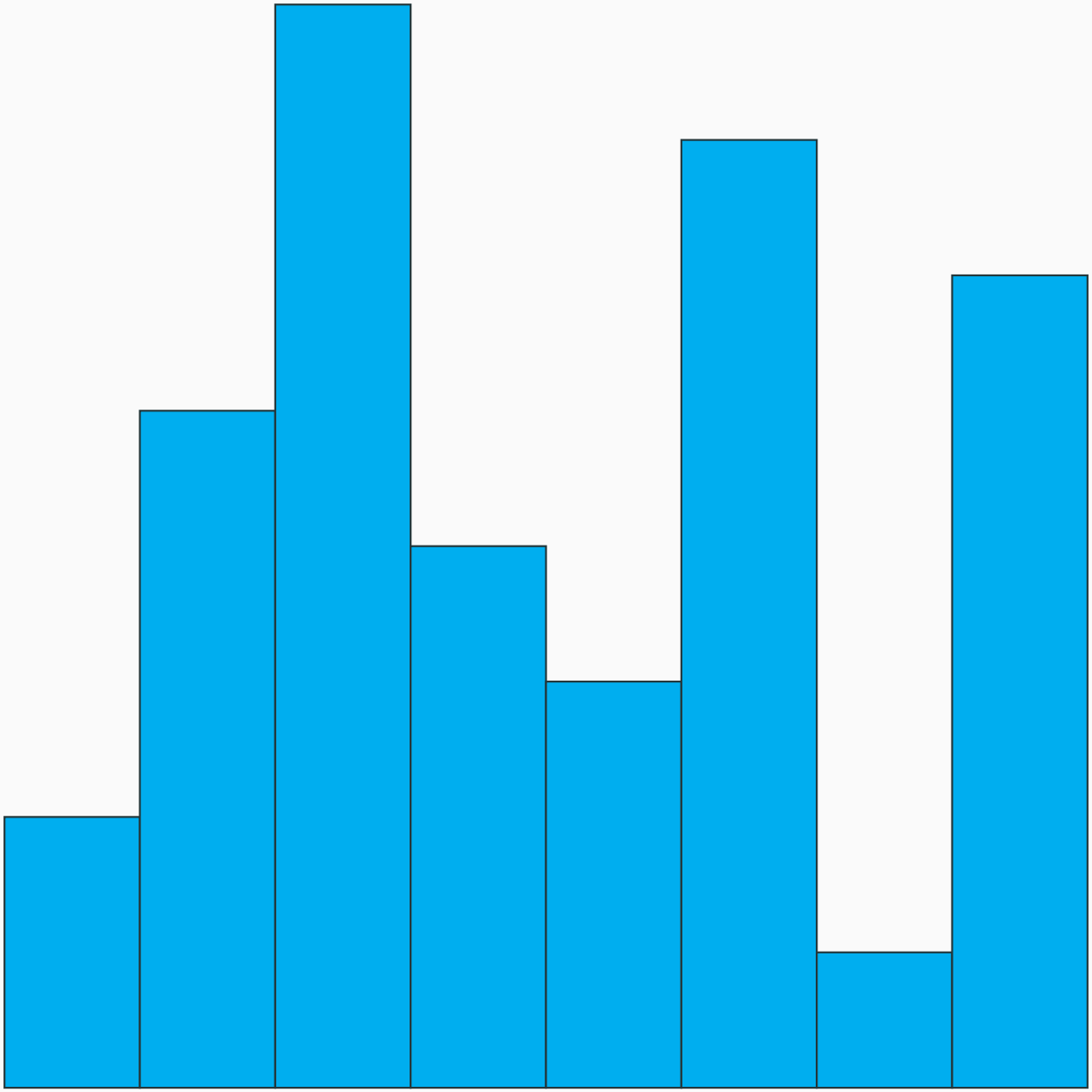
Idée

- On maintient constamment la partie gauche du tableau trié.
- Lorsqu'on considère un nouvel élément x (celui juste à droite de la partie triée), il faut l'**insérer** dans la portion de gauche de façon à ce que cette portion augmentée de 1 élément soit triée.
- Pour ce faire, plutôt que de procéder par échanges, on sauvegarde la valeur de x dans une variable.
- Il suffit ensuite de faire monter un à un les éléments du tableau tant qu'ils sont plus grands que x.
- Une fois ceci effectué, on peut positionner x.

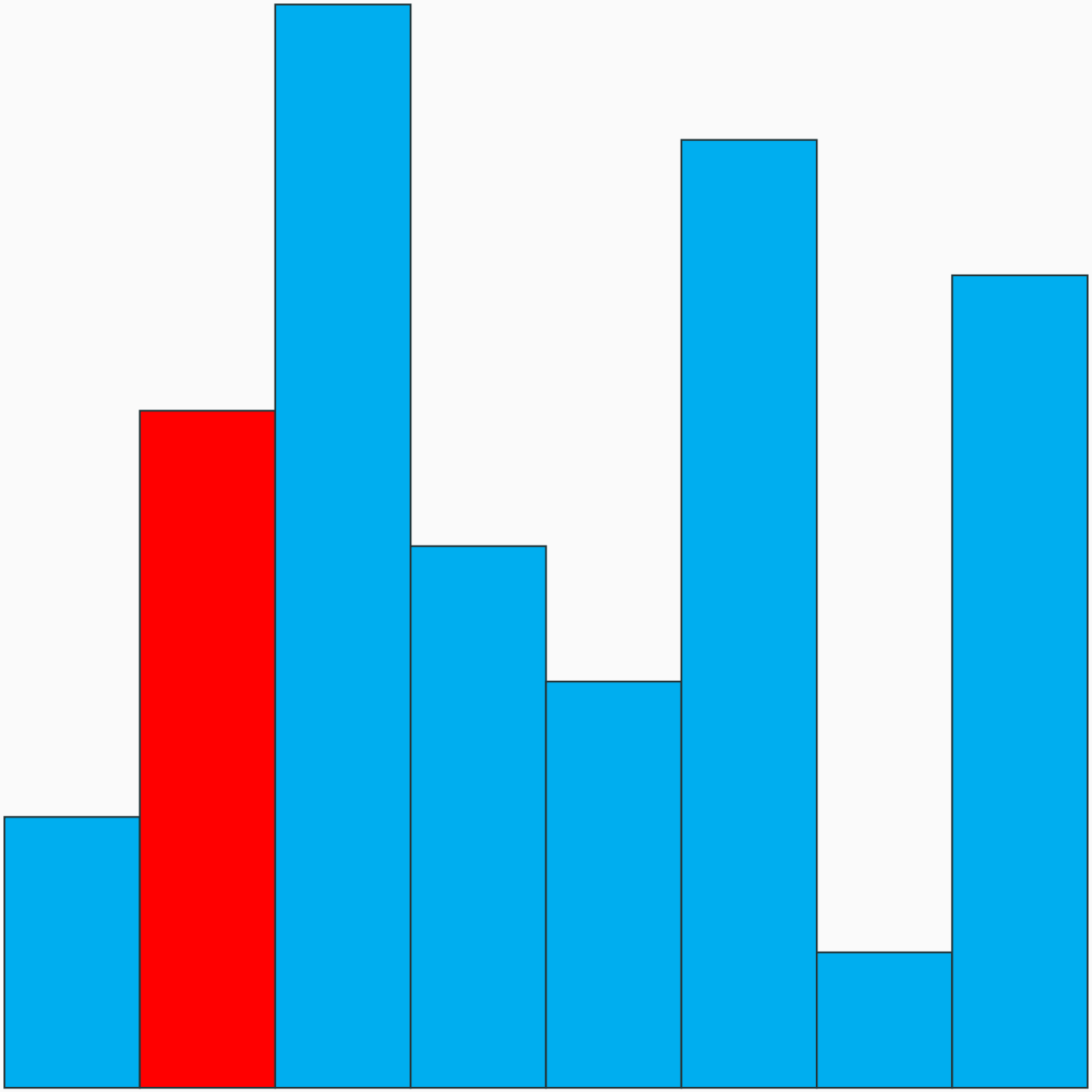
Tri par insertion

```
1  let tri_insertion t =
2    let n = Array.length t in
3    for i=1 to n-1 do
4    (* Inv'(i): t.(0),..,t.(i-1) triés *)
5      let x = t.(i) and j = ref i in
6      while !j>0 && t.(!j-1) > x do
7        (* Inv: Pour tout k vérifiant j<k<=i, L[k]>x *)
8        t.(!j) <- t.(!j-1) ;
9        decr j
10       (* Inv *)
11     done ;
12     t.(!j) <- x
13     (* Inv'(i+1) *)
14   done
15 ;;
```

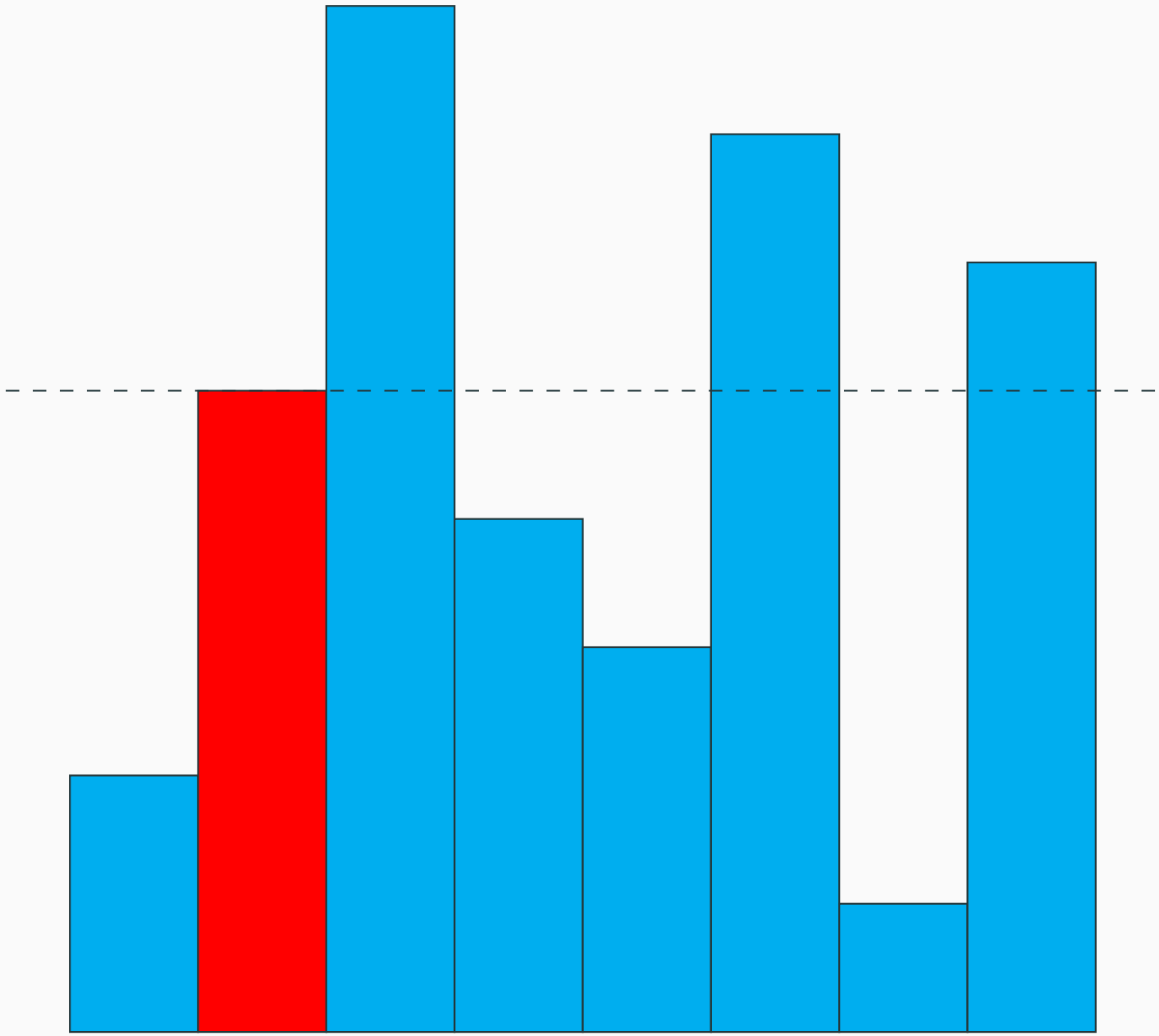
Exemple



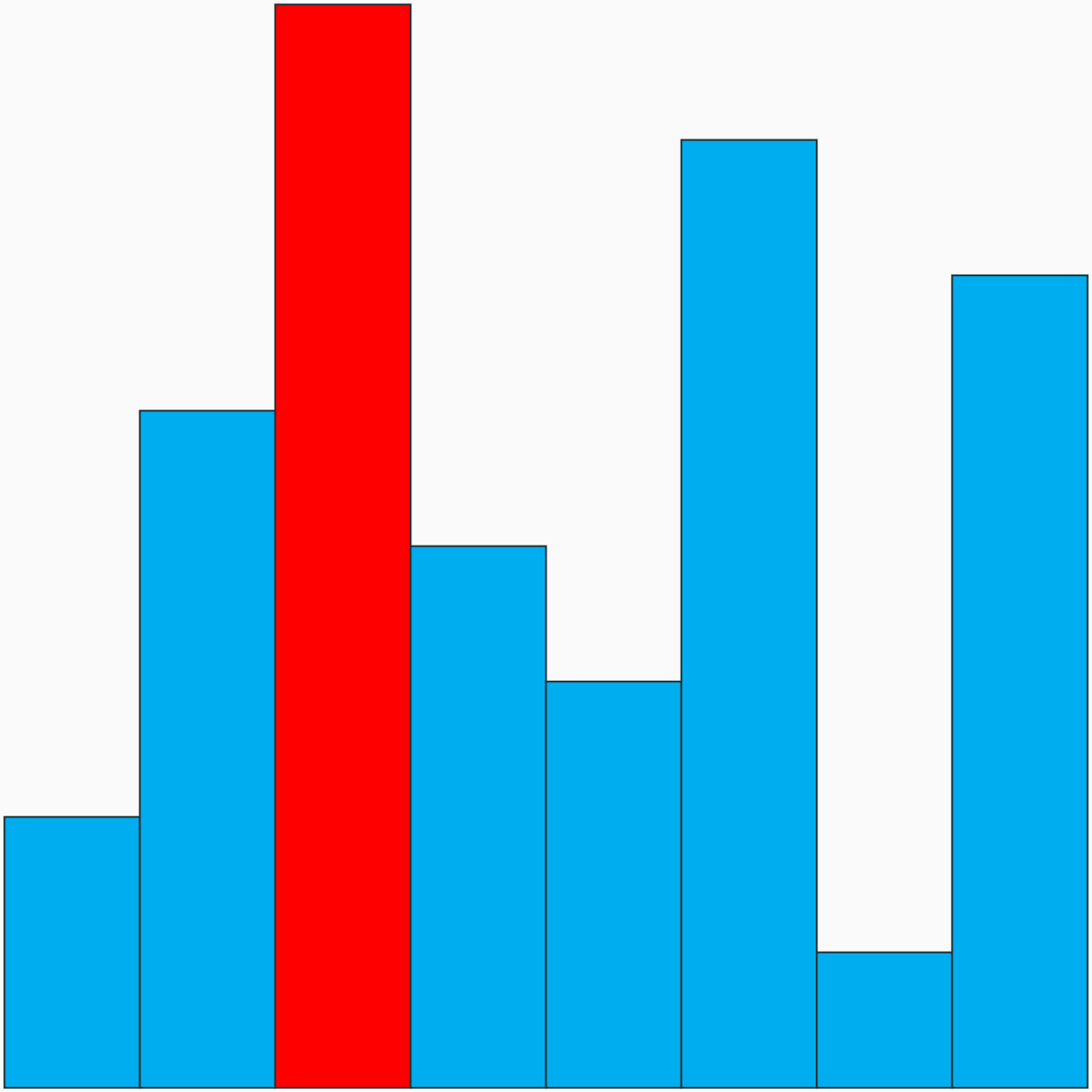
Exemple



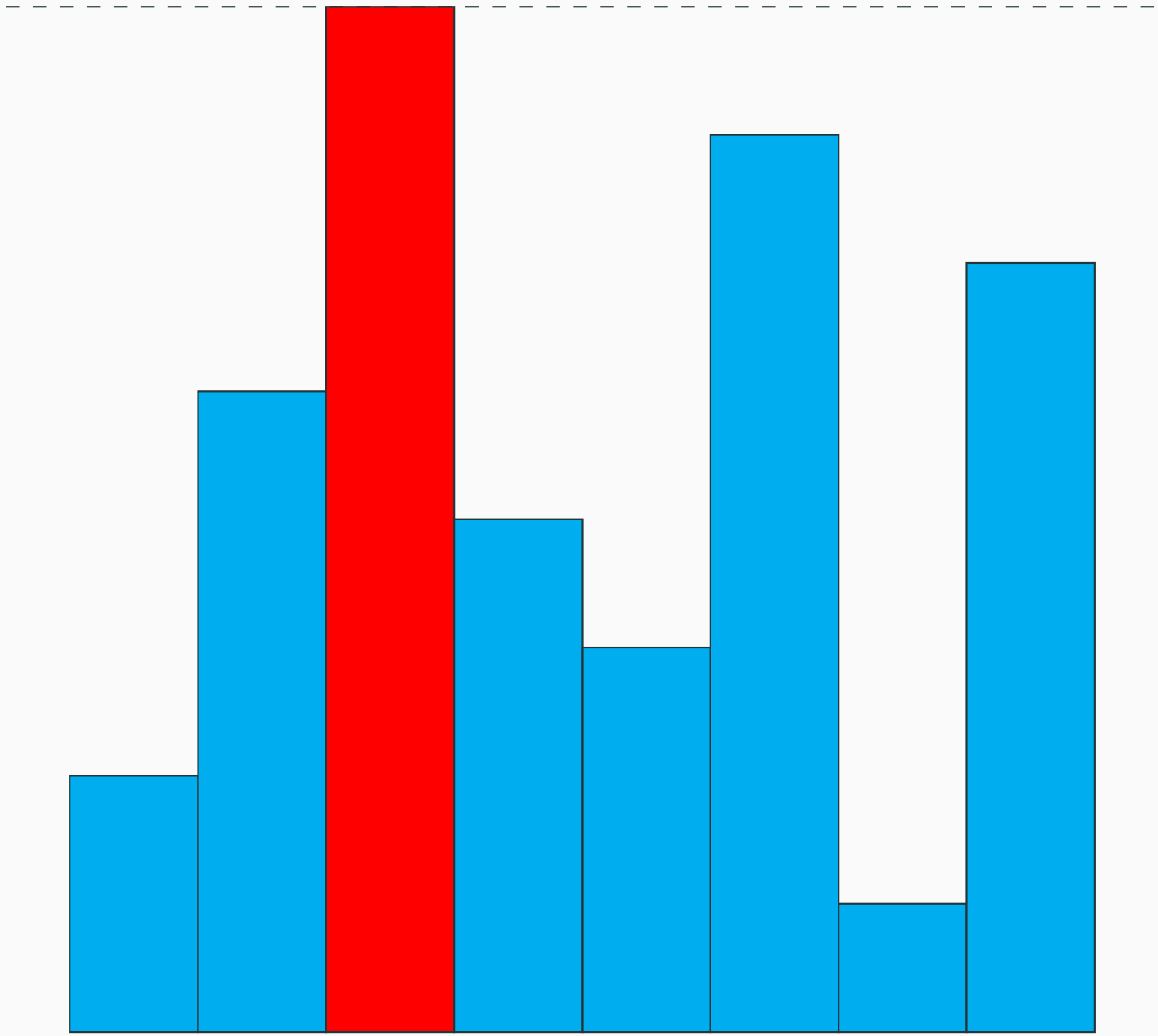
Exemple



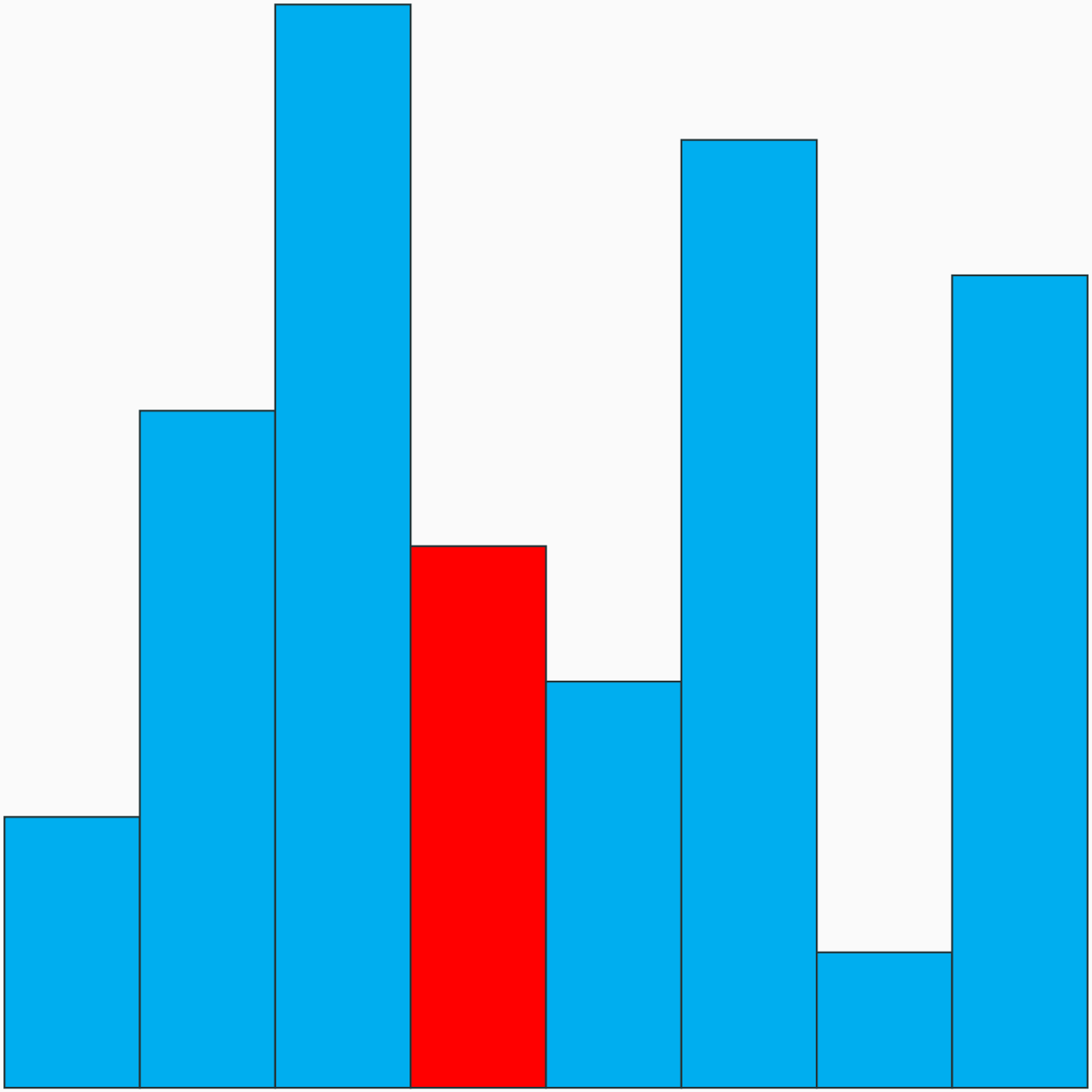
Exemple



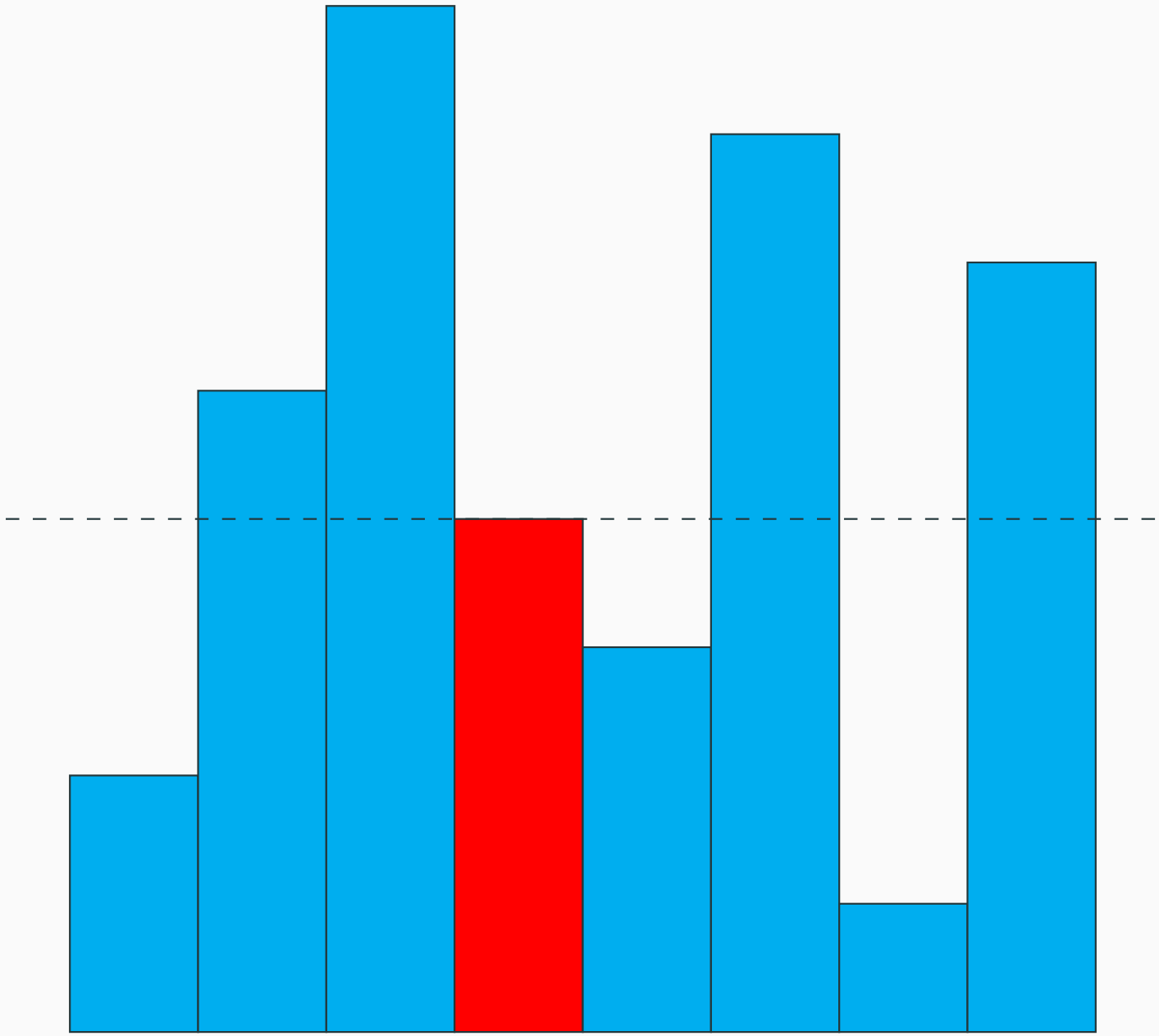
Exemple



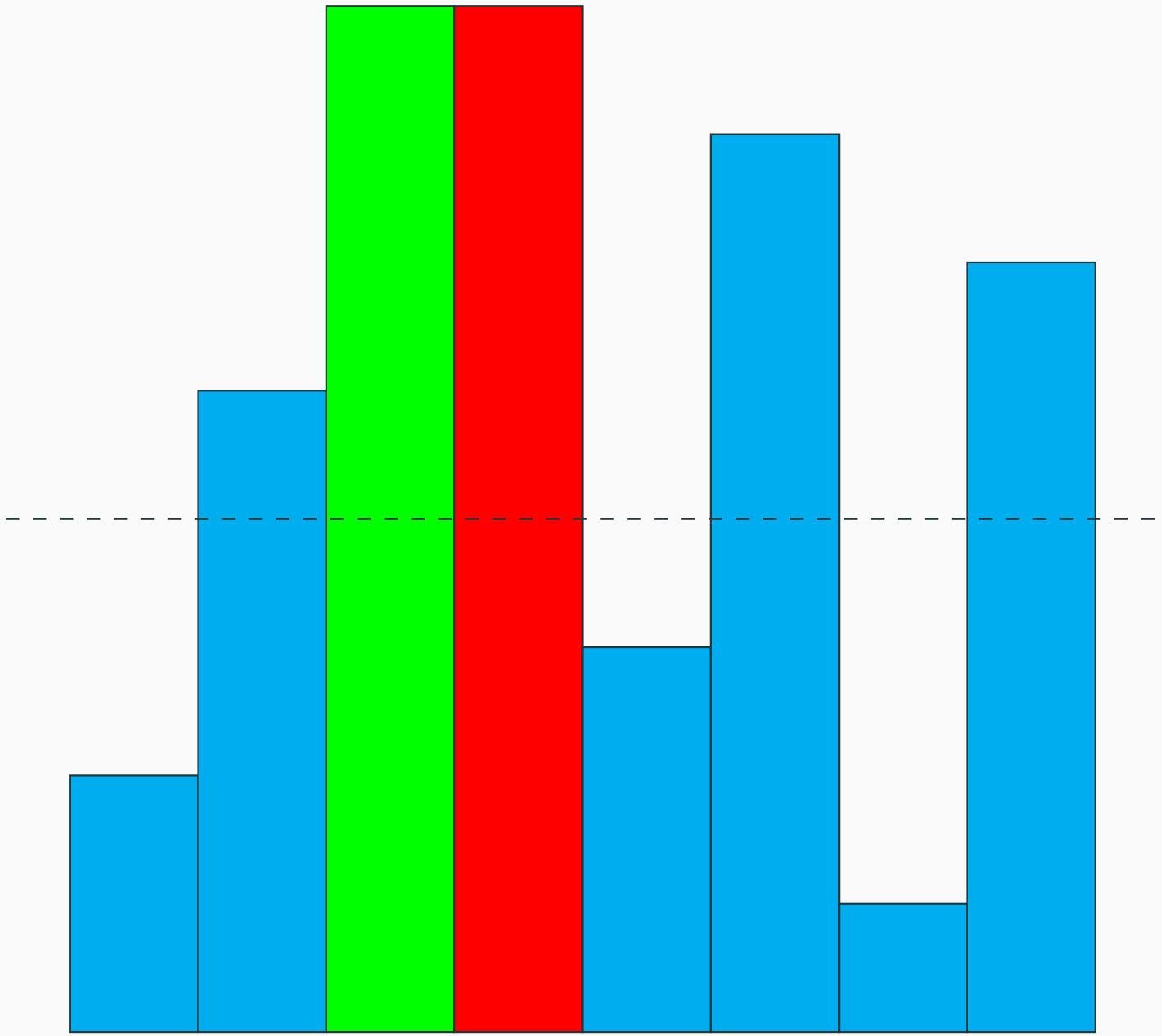
Exemple



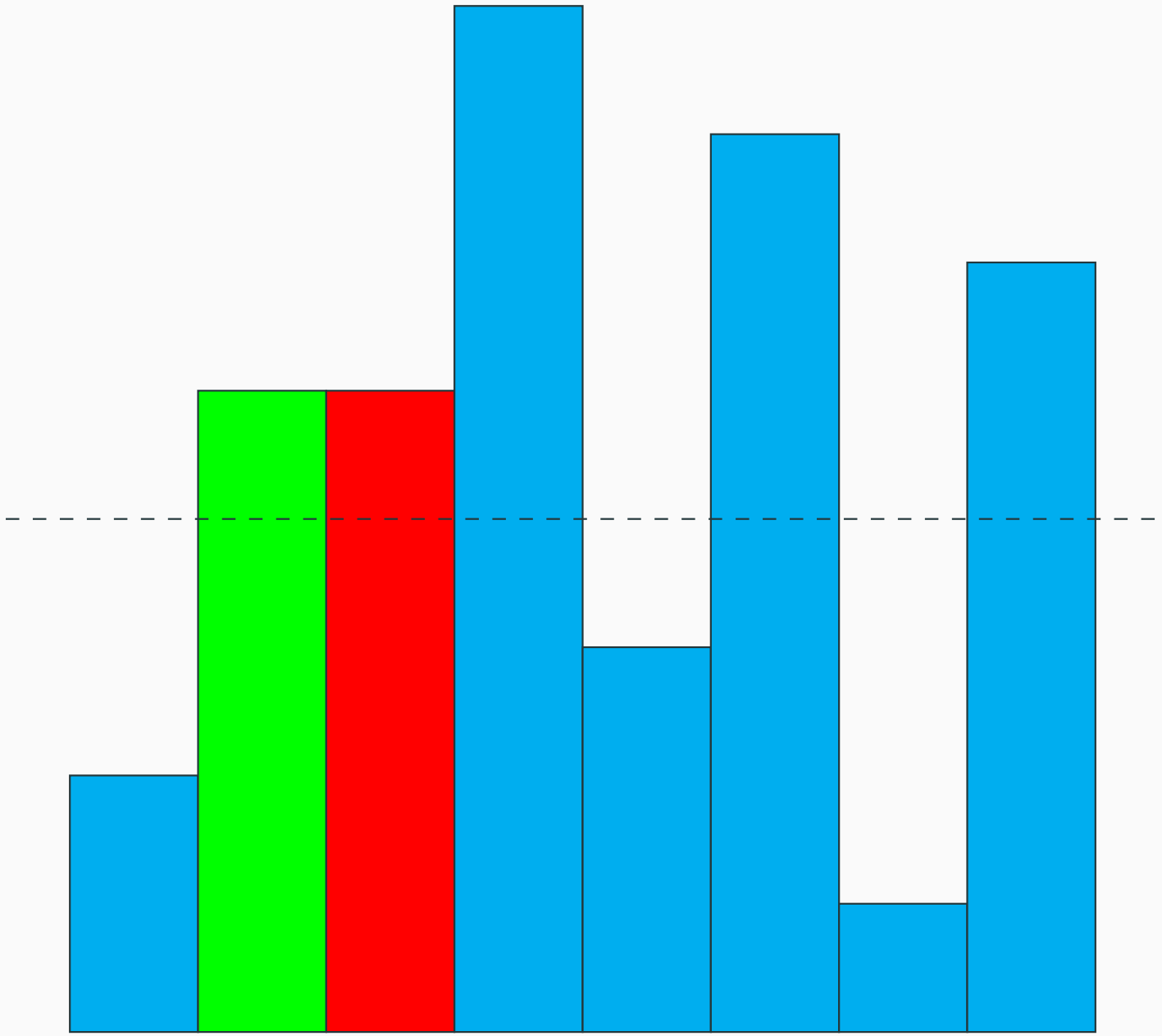
Exemple



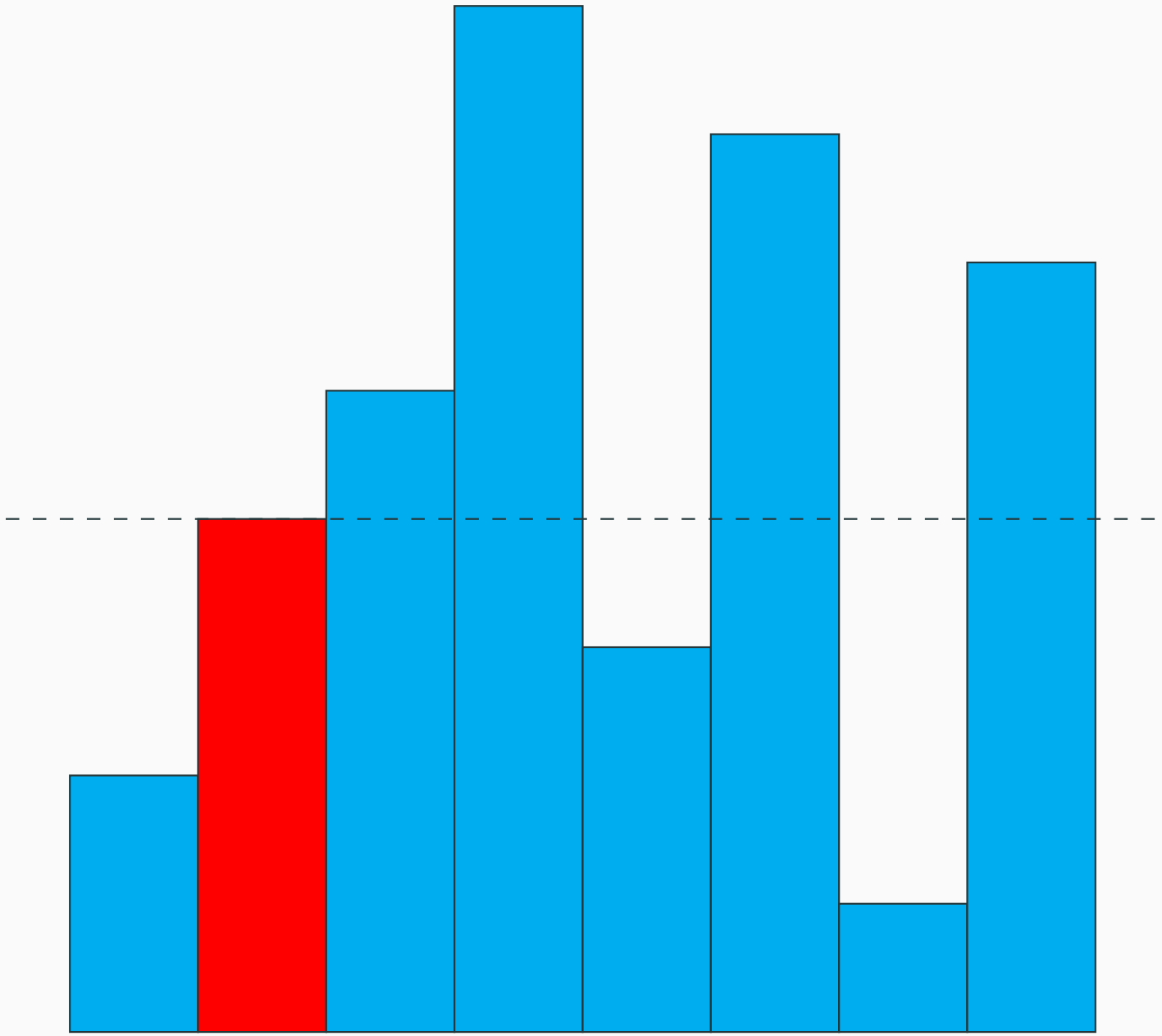
Exemple



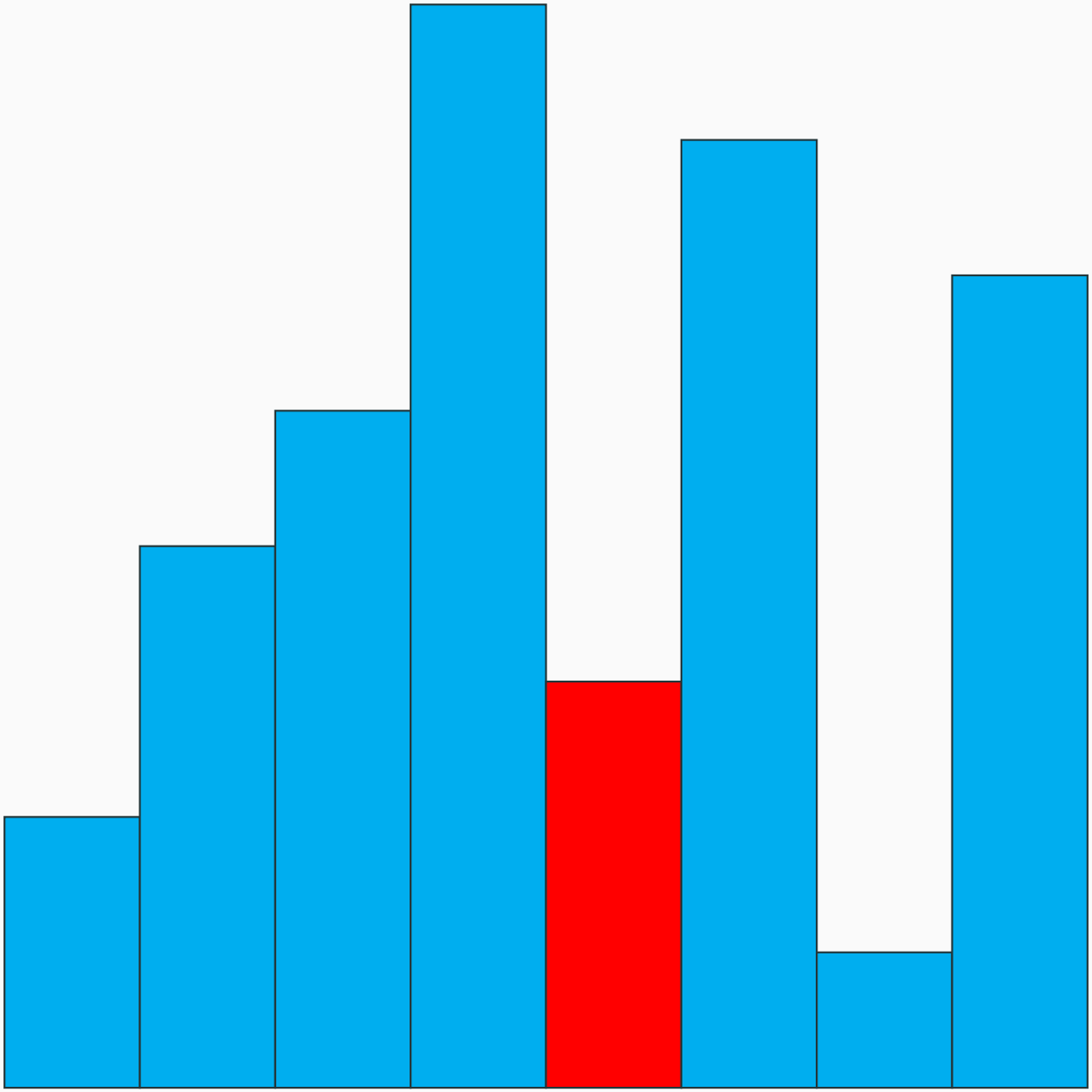
Exemple



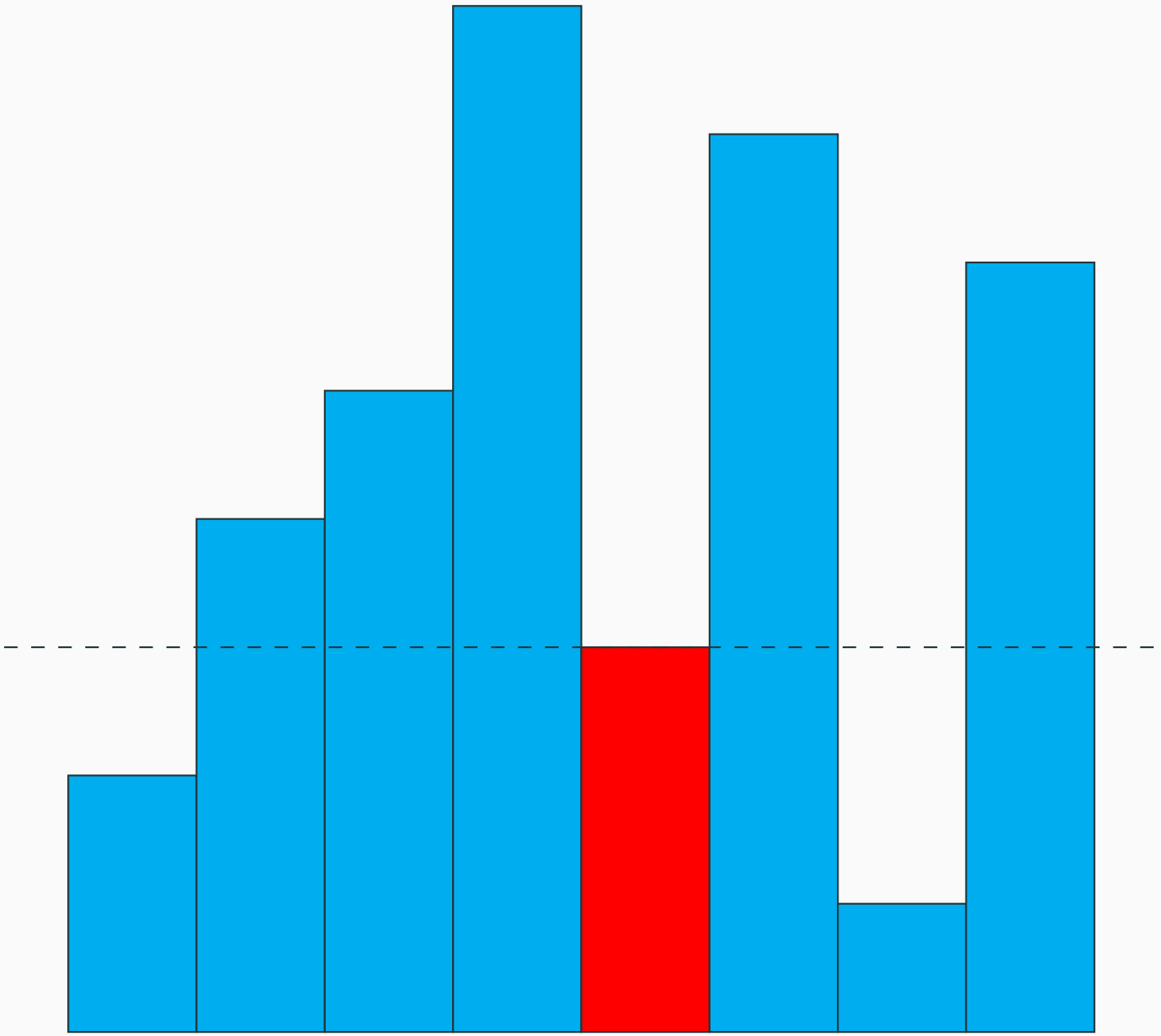
Exemple



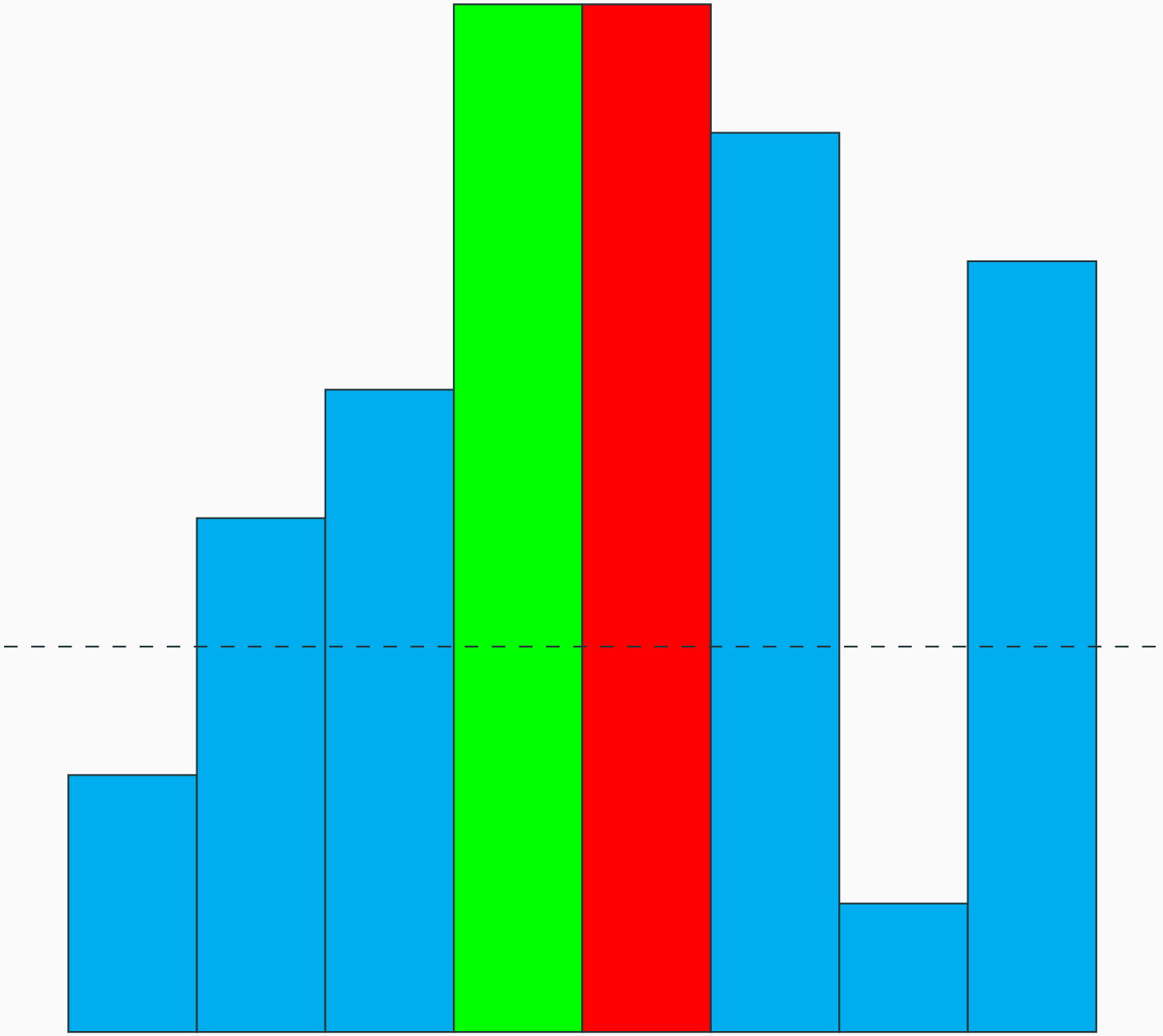
Exemple



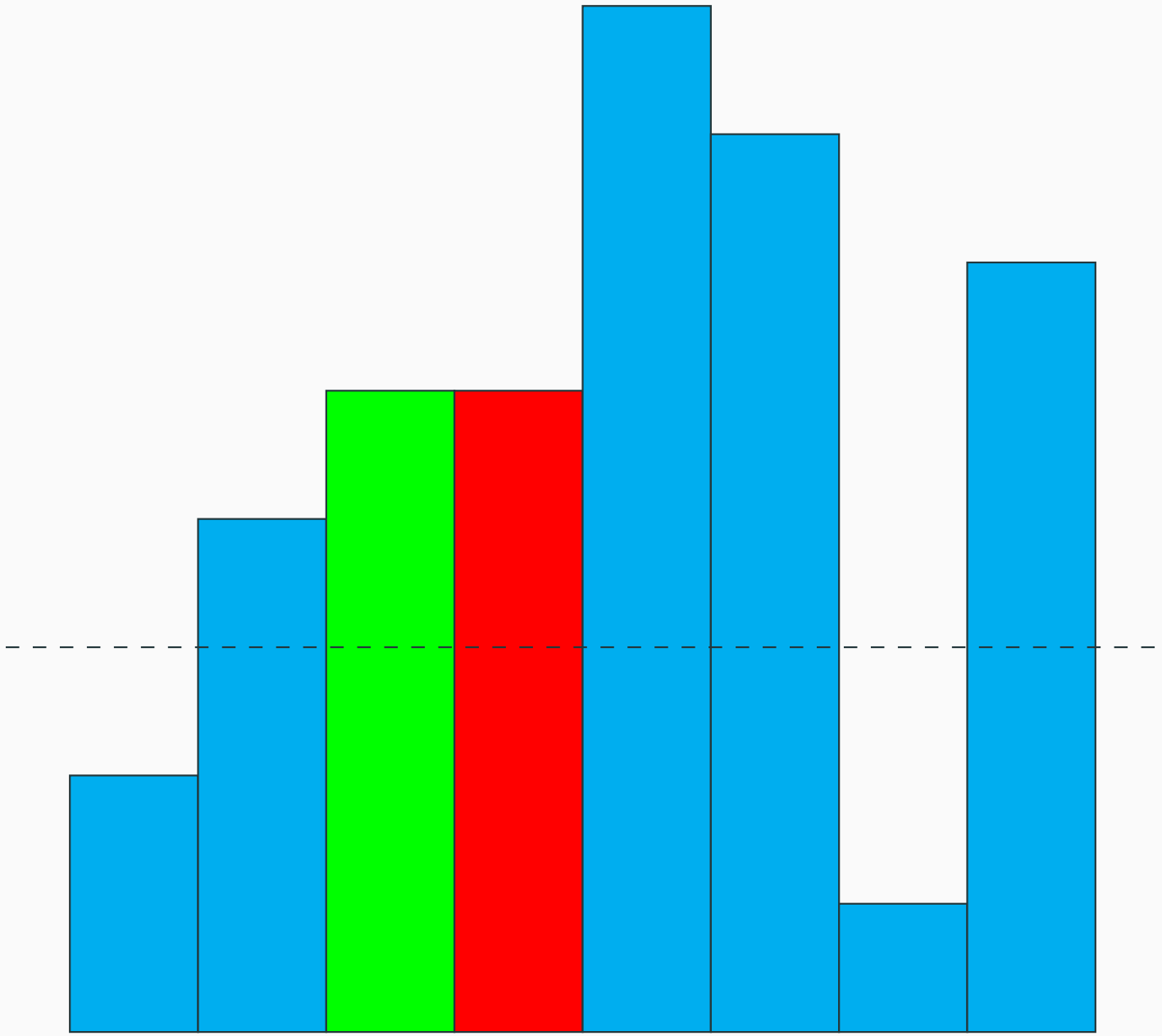
Exemple



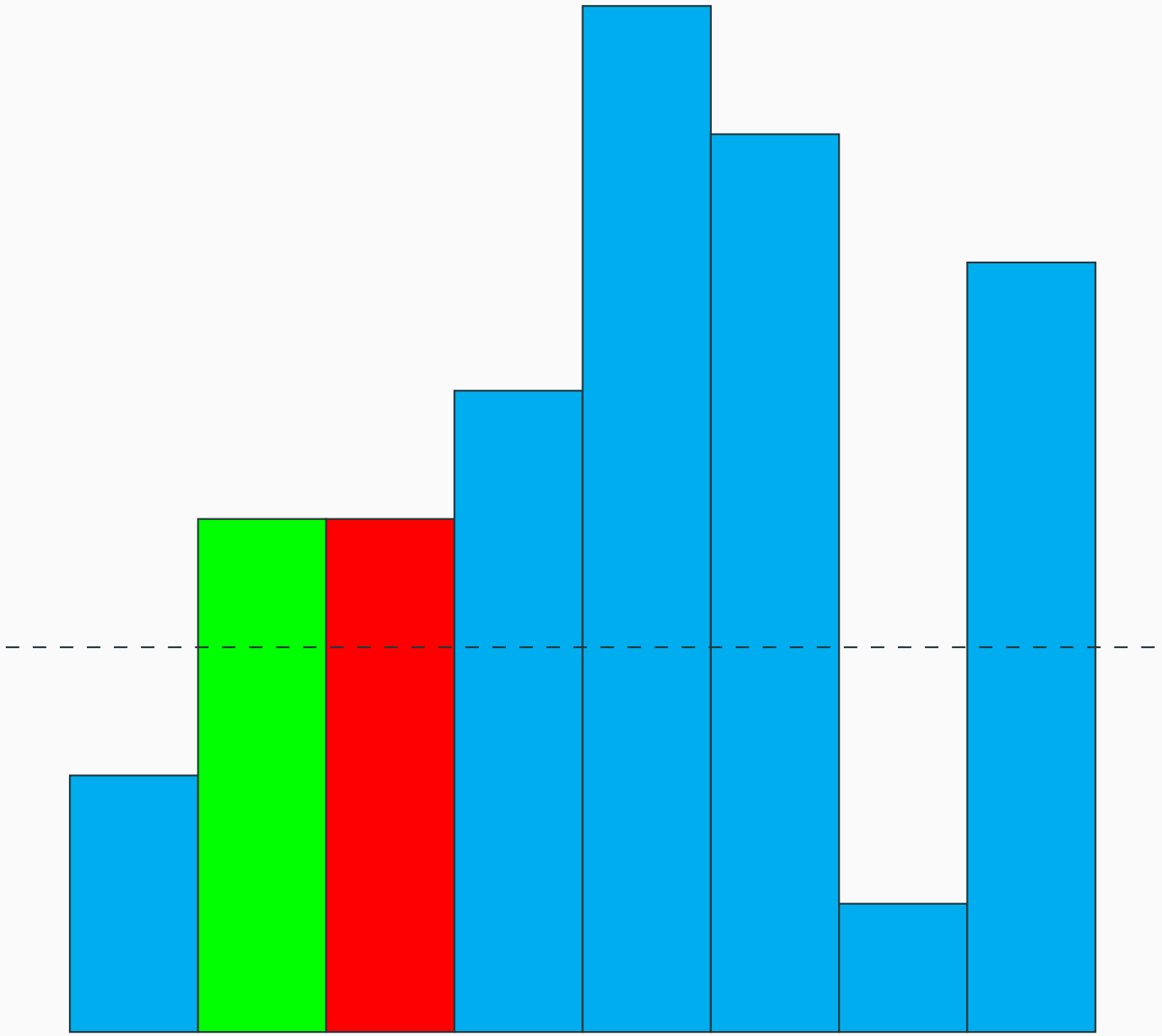
Exemple



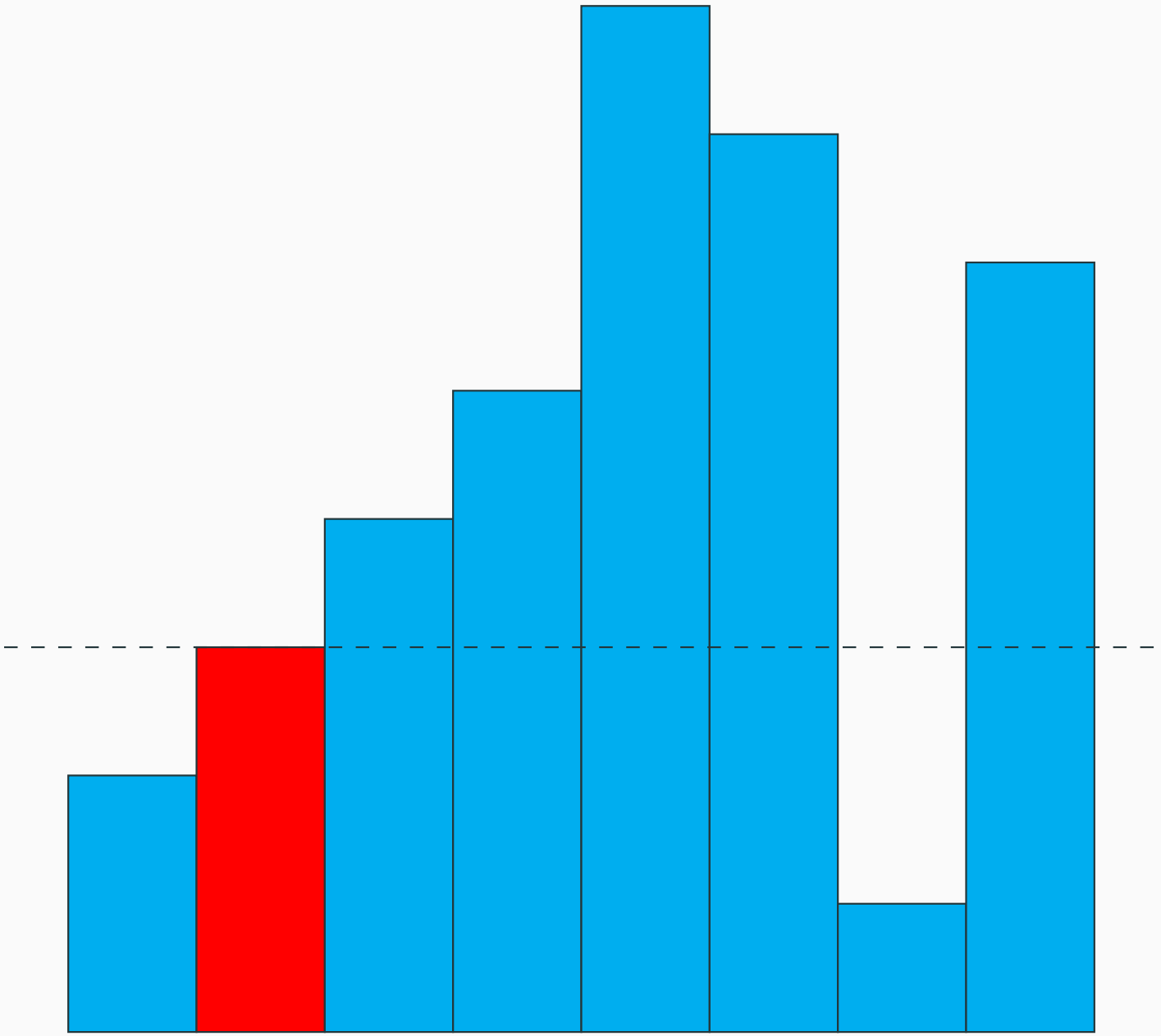
Exemple



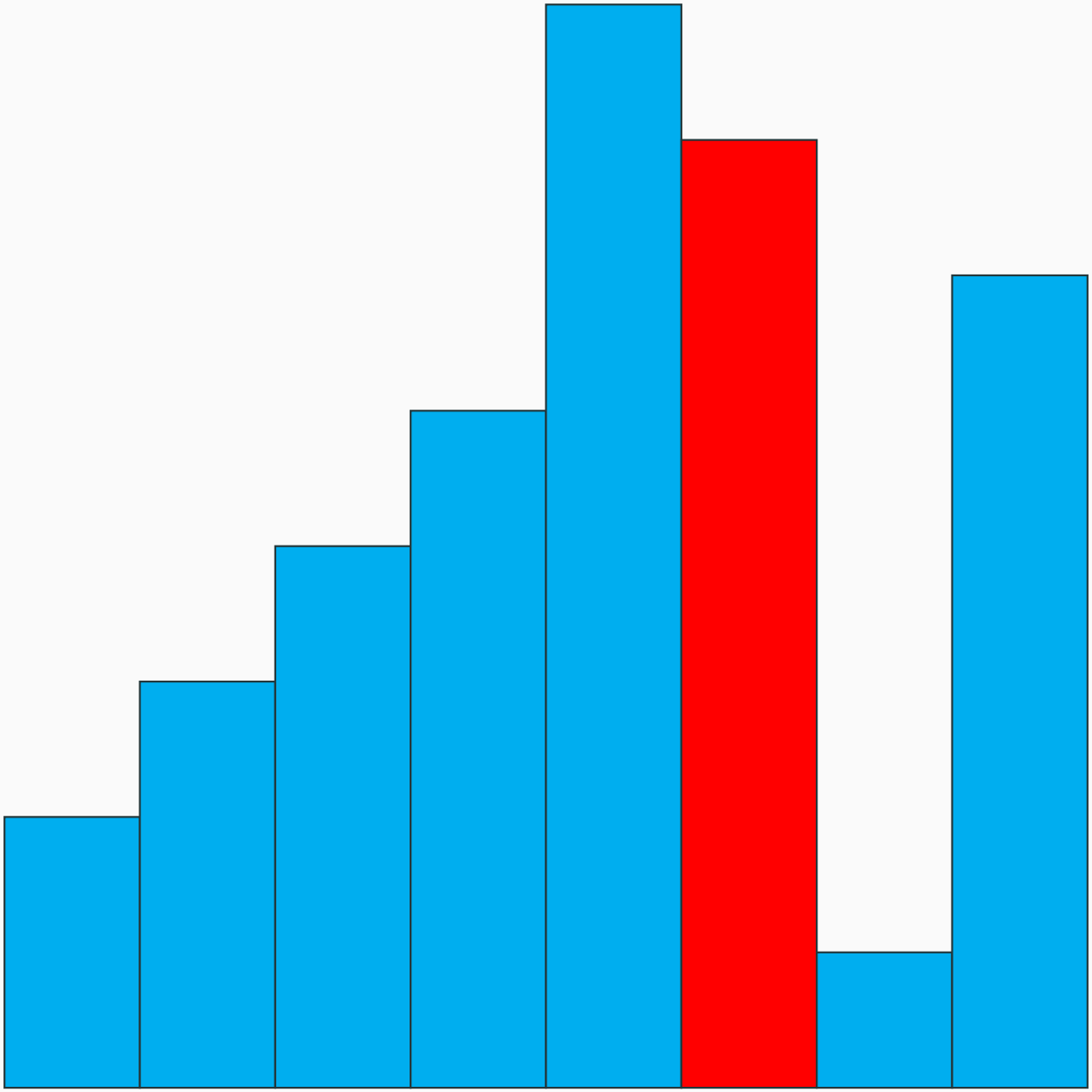
Exemple



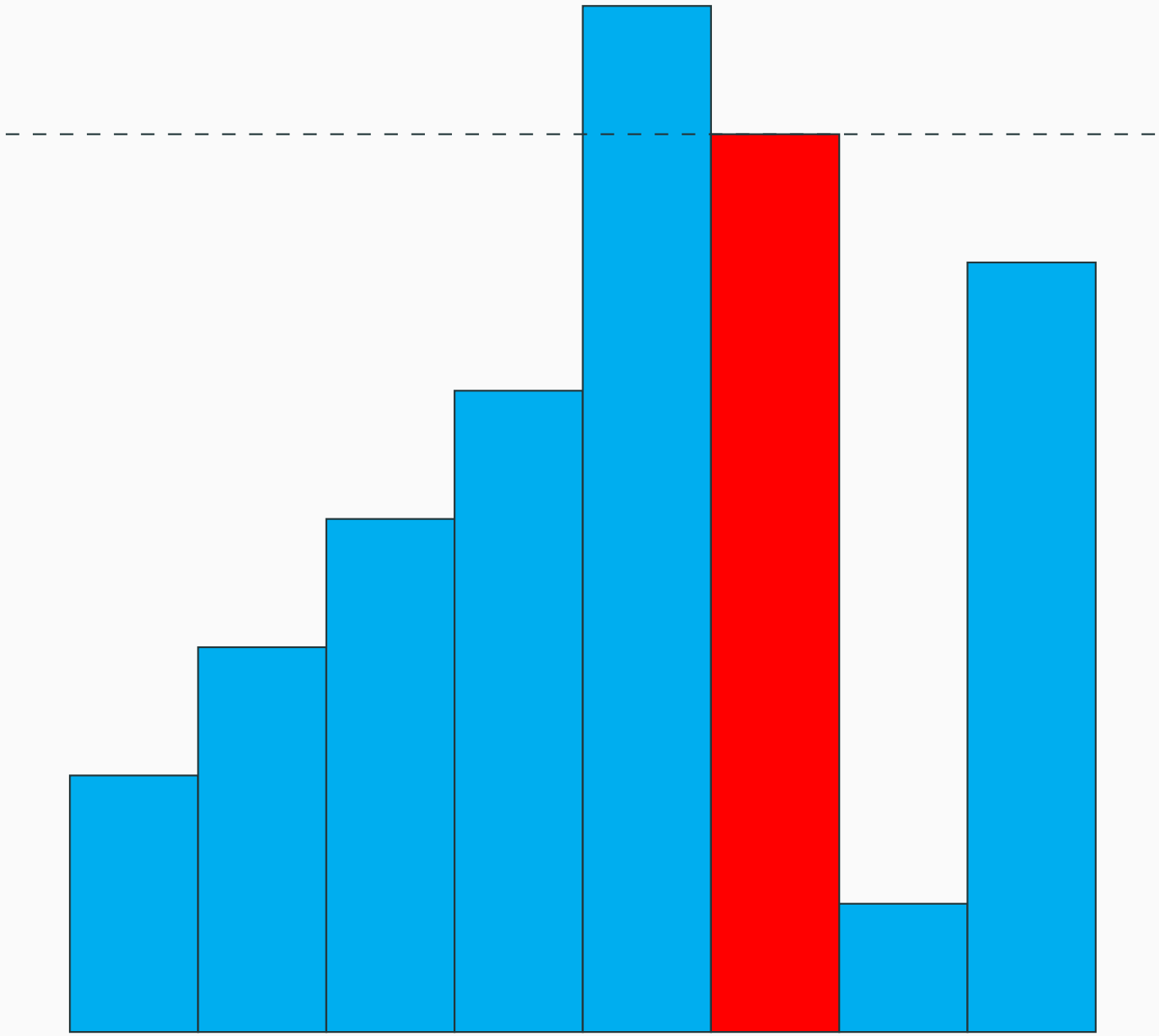
Exemple



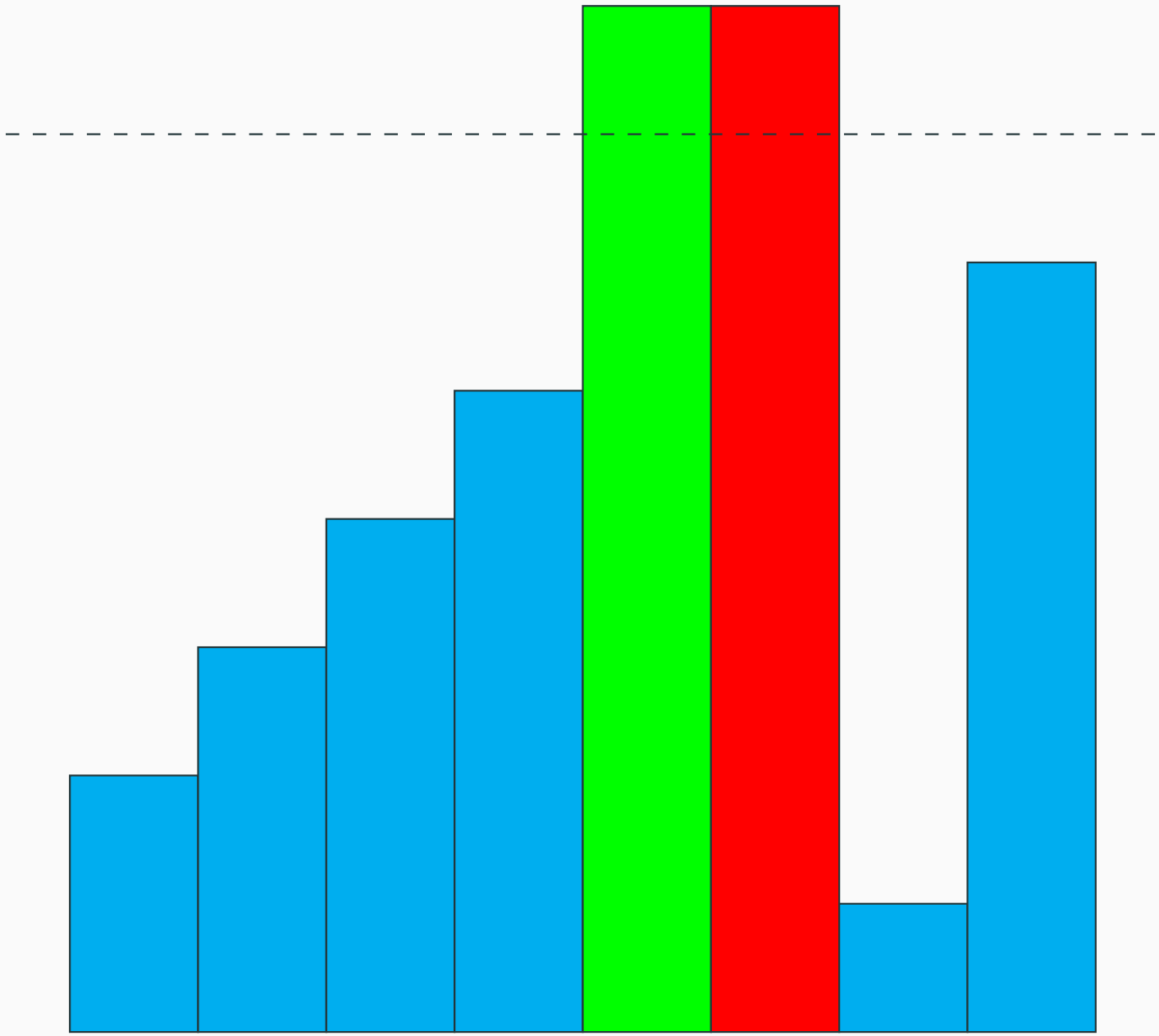
Exemple



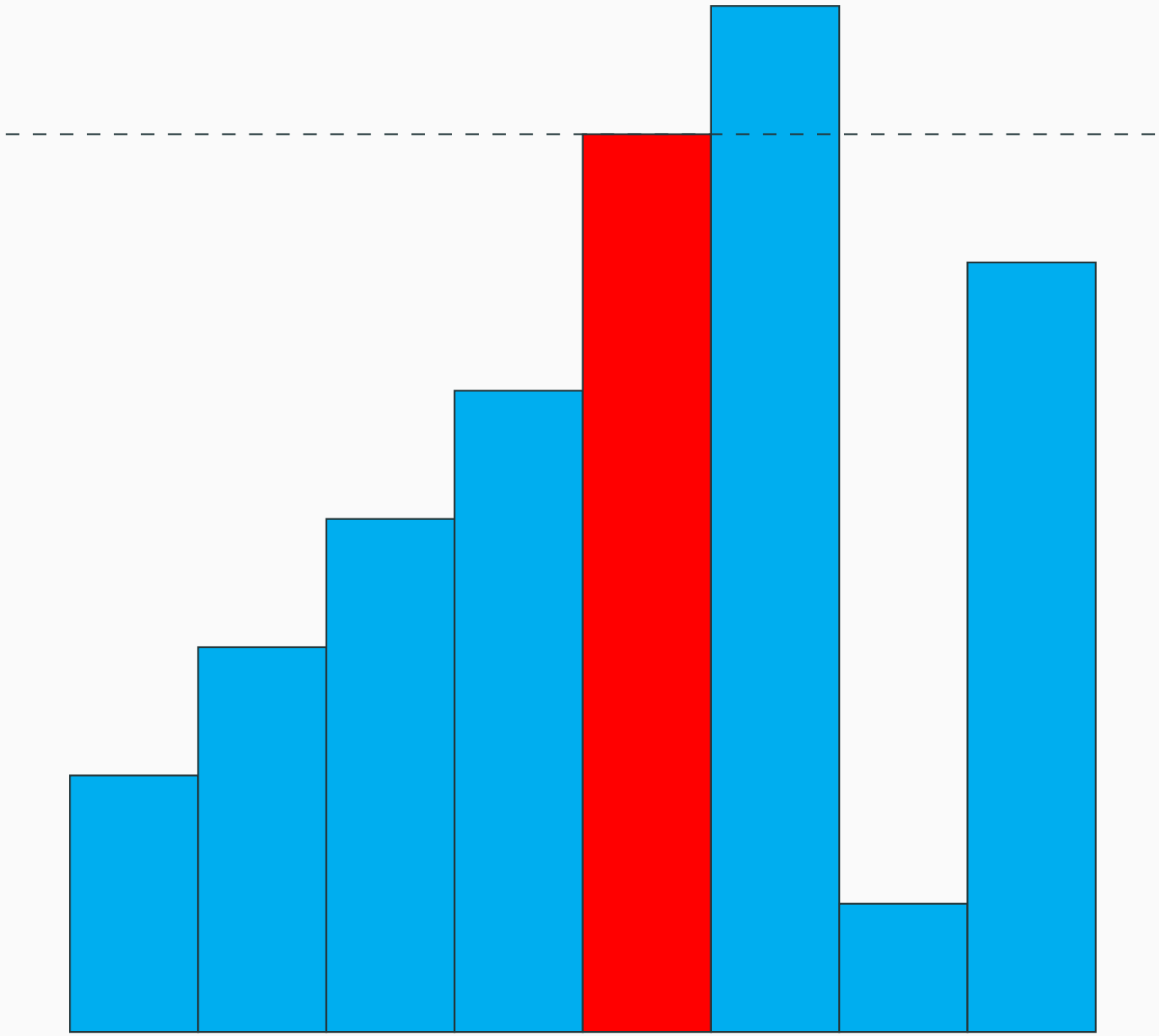
Exemple



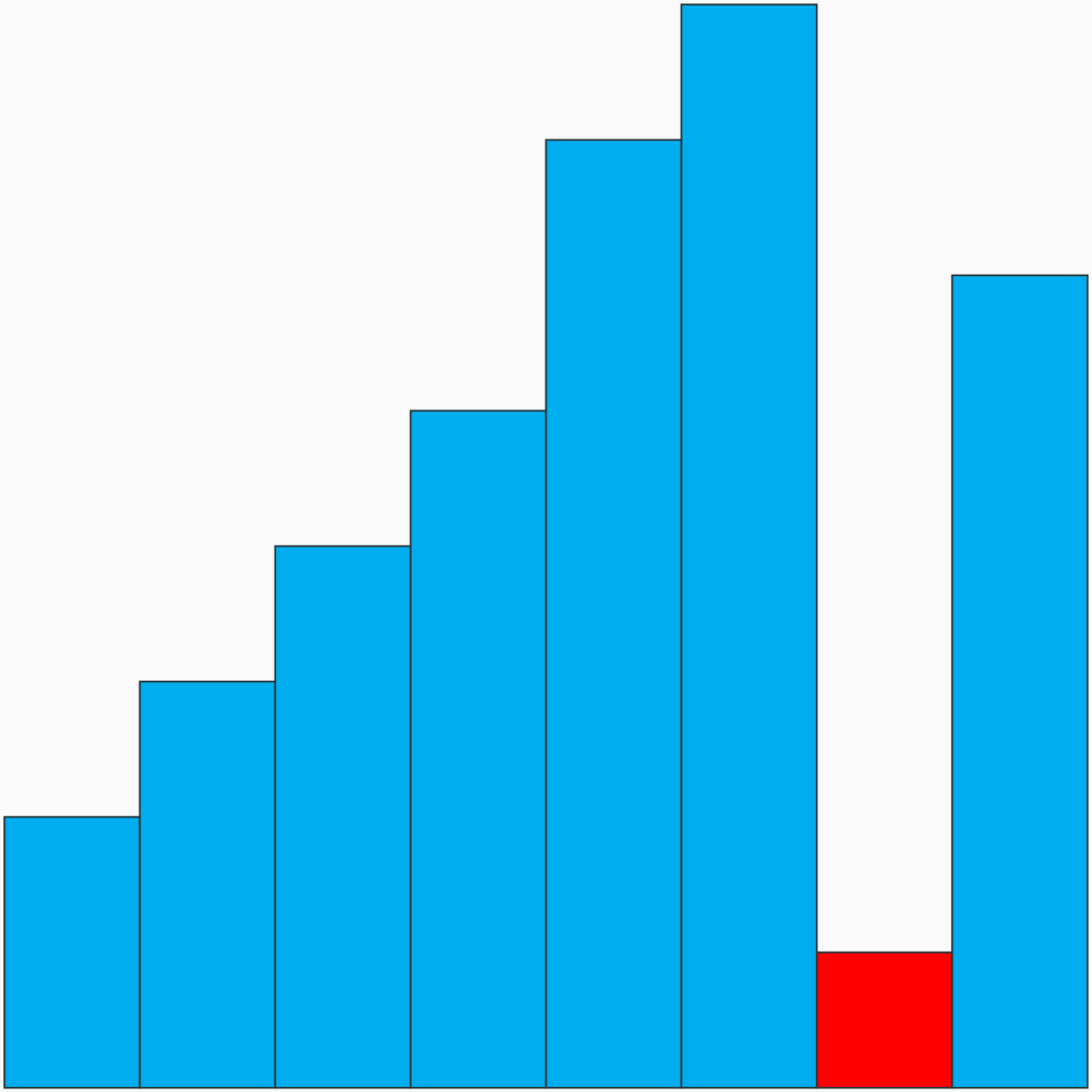
Exemple



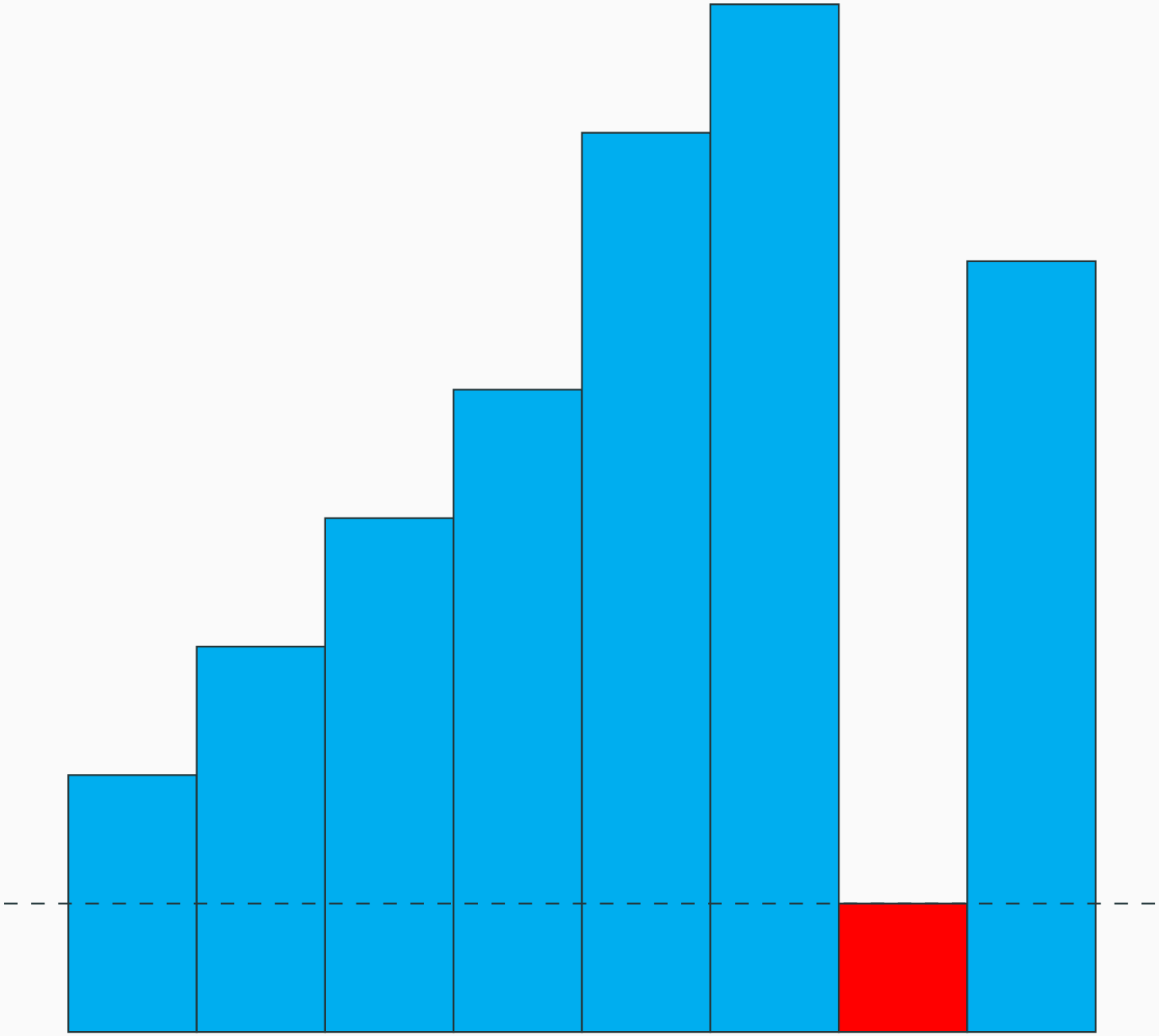
Exemple



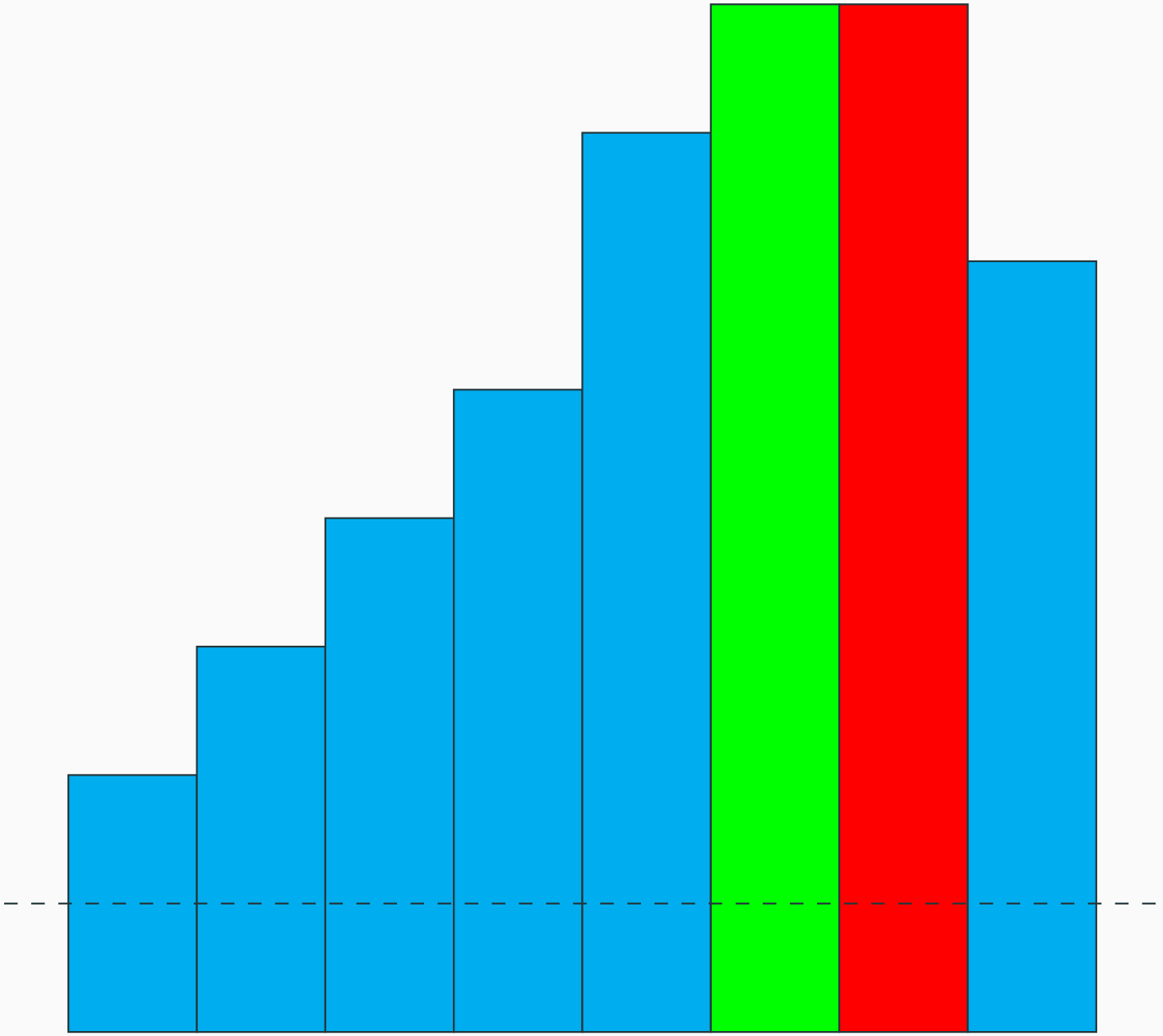
Exemple



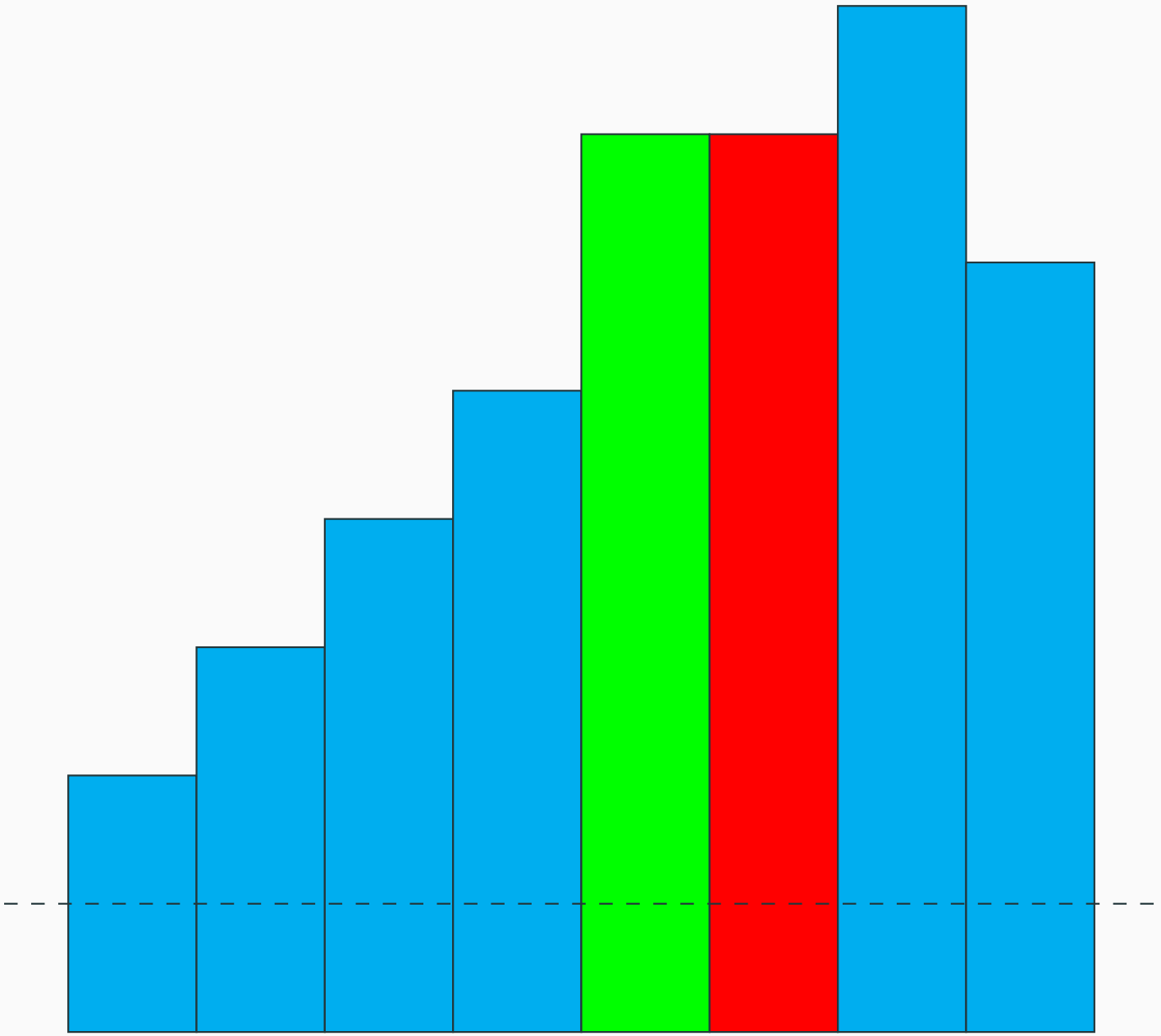
Exemple



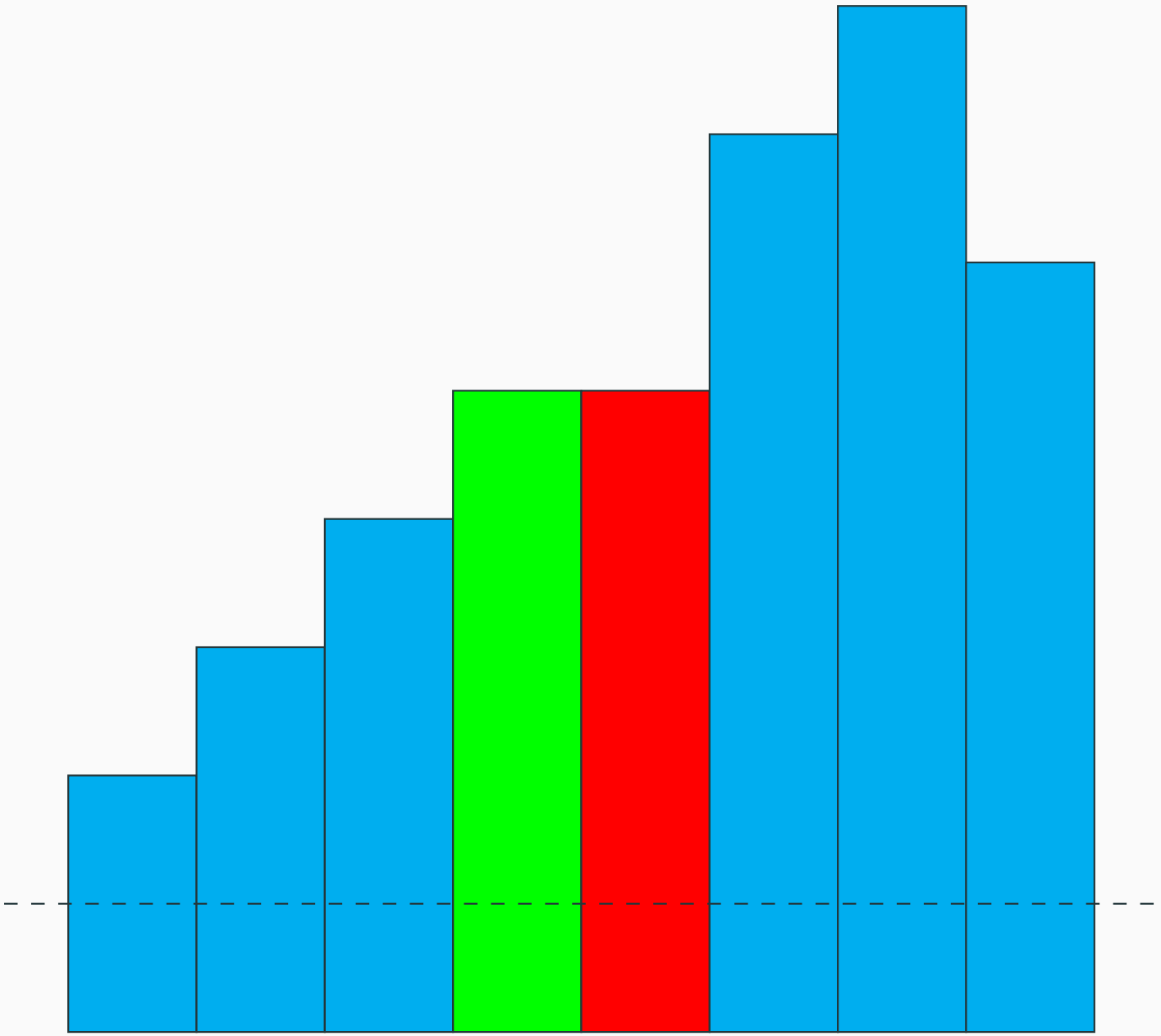
Exemple



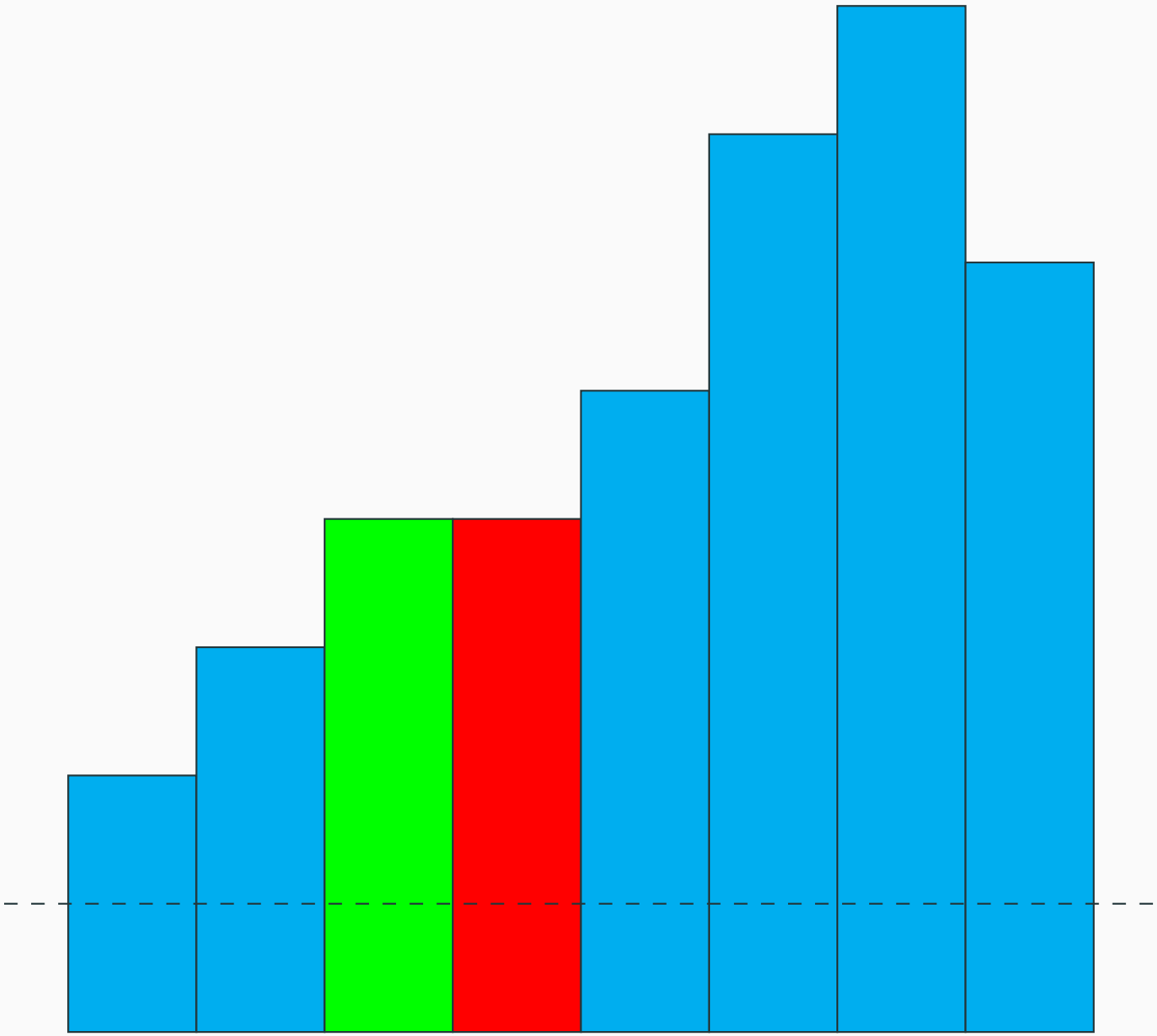
Exemple



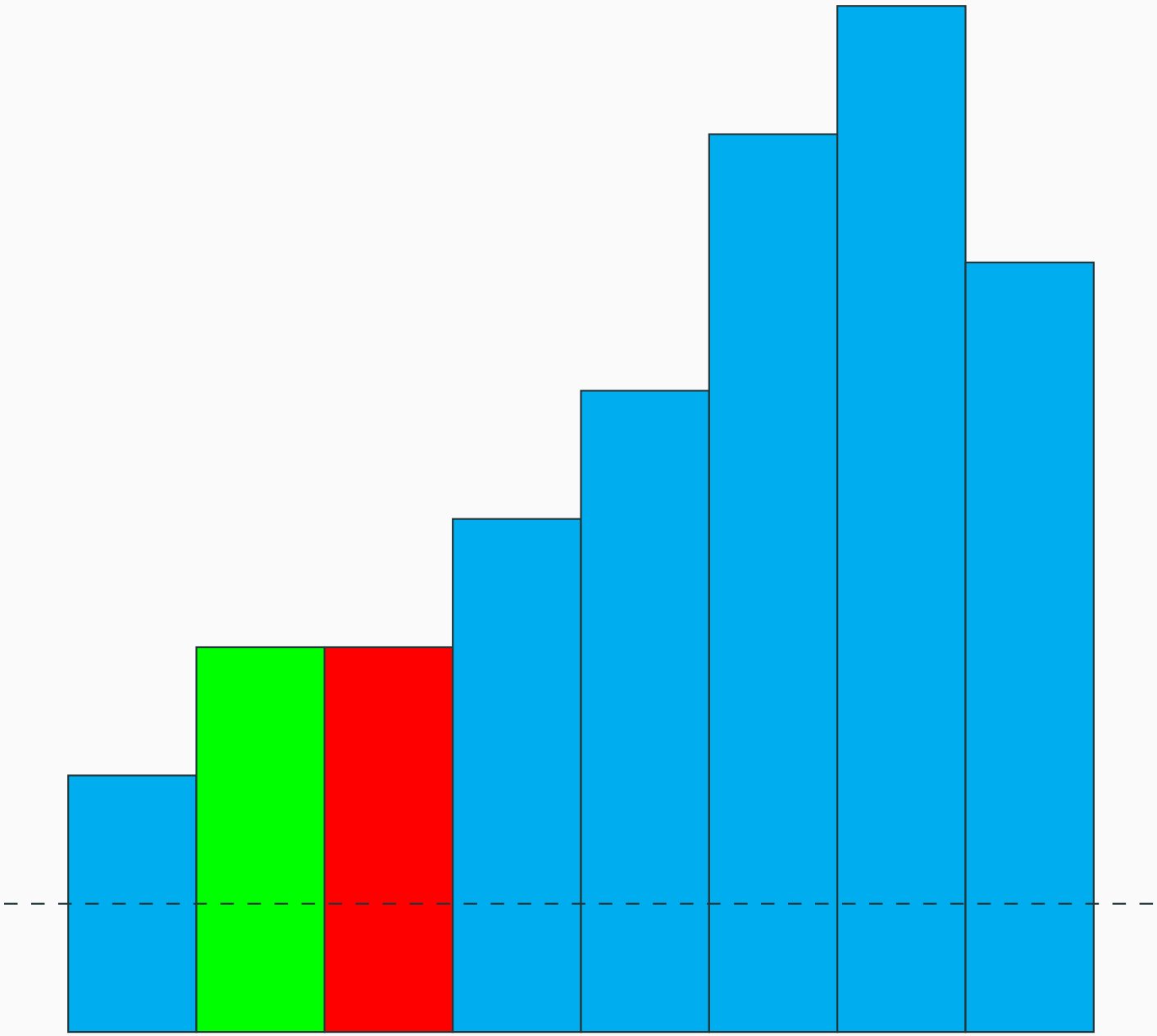
Exemple



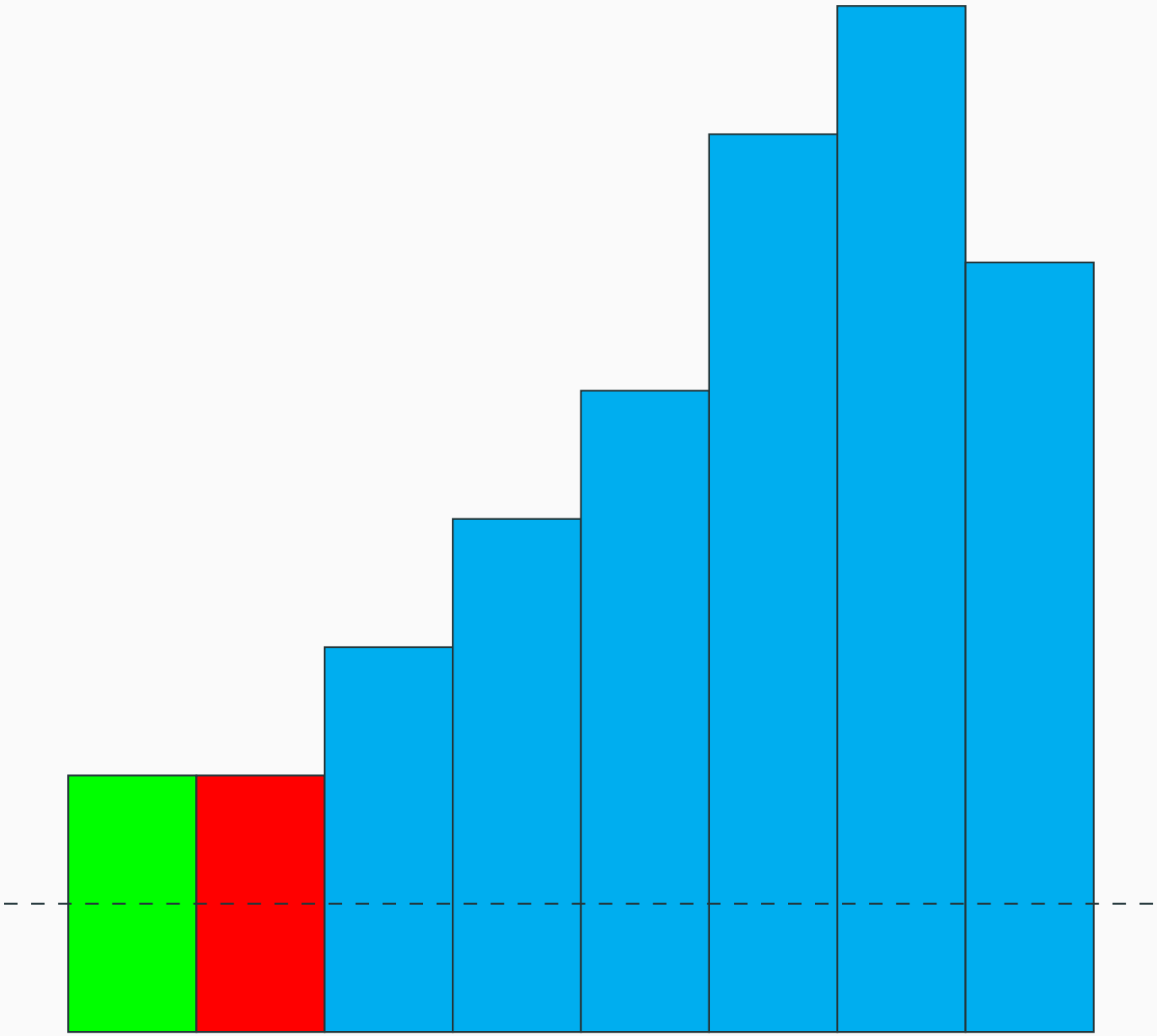
Exemple



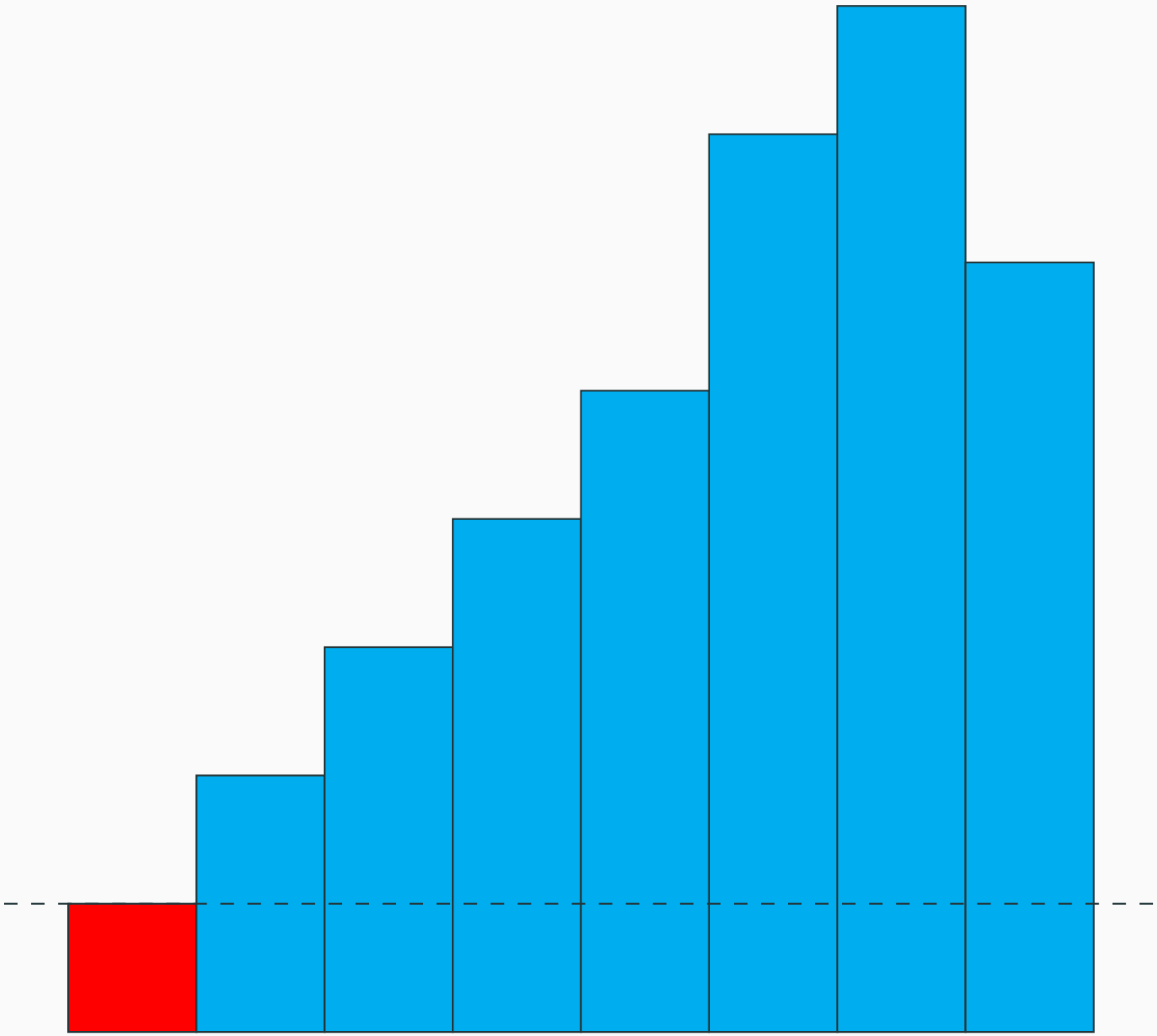
Exemple



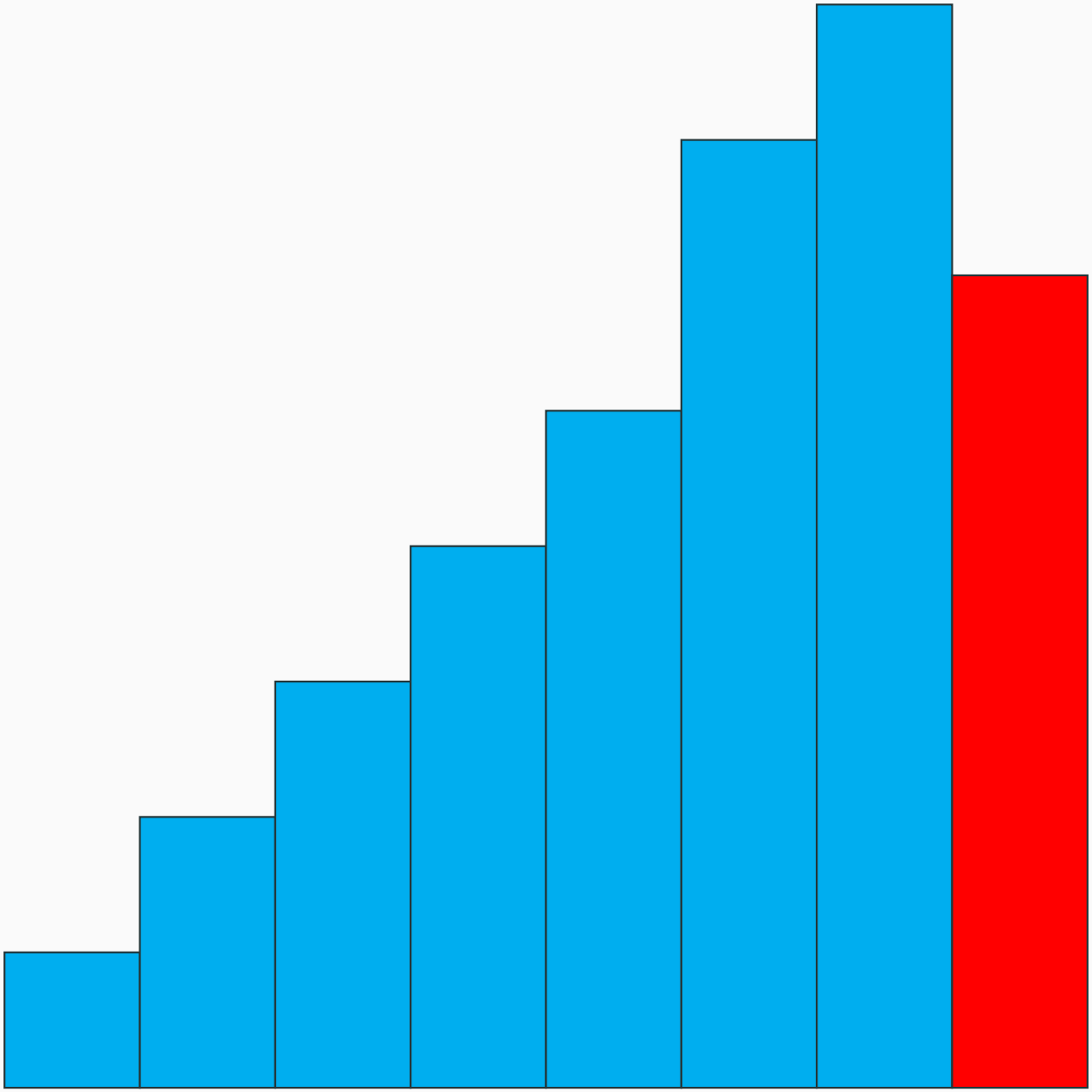
Exemple



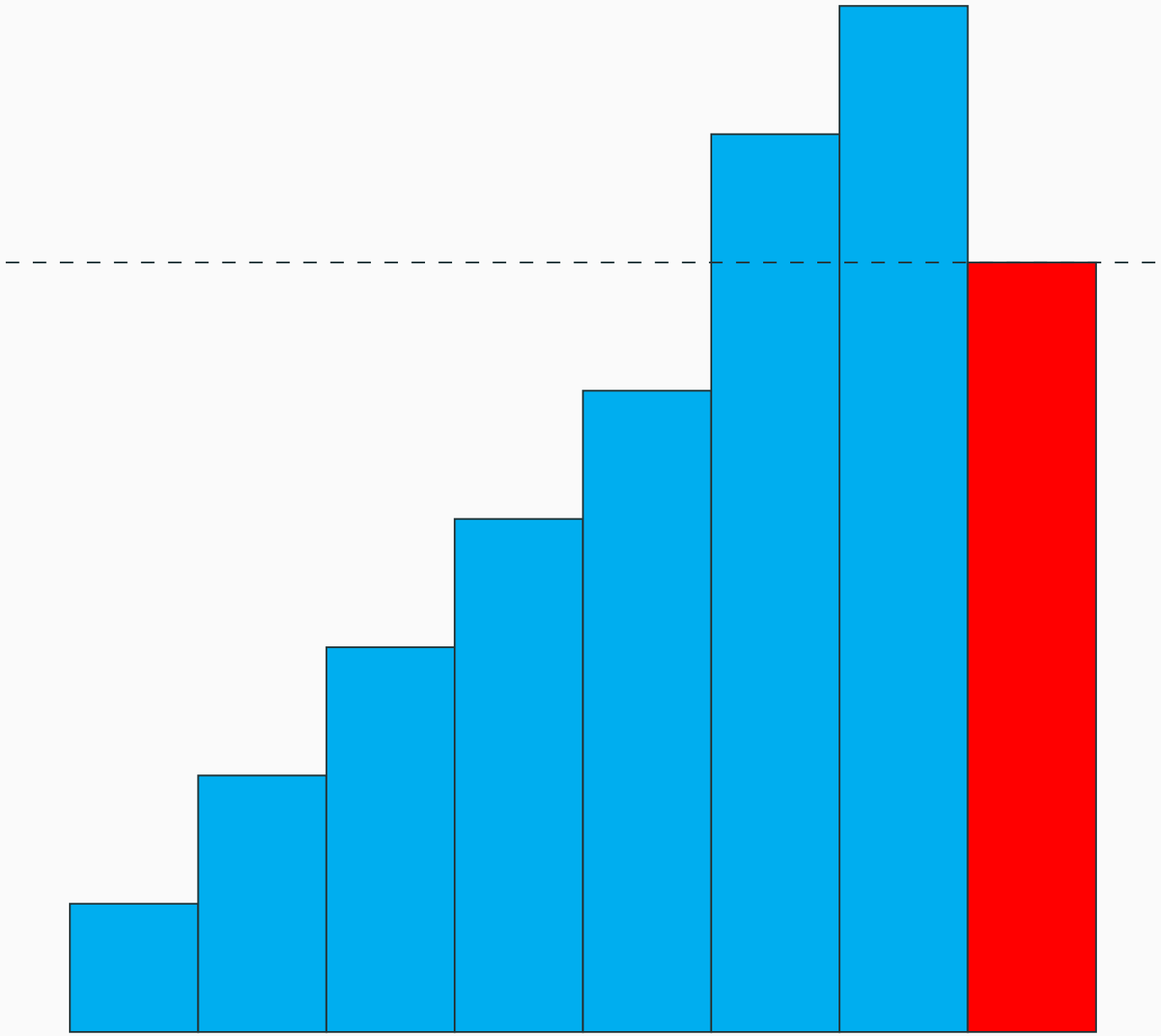
Exemple



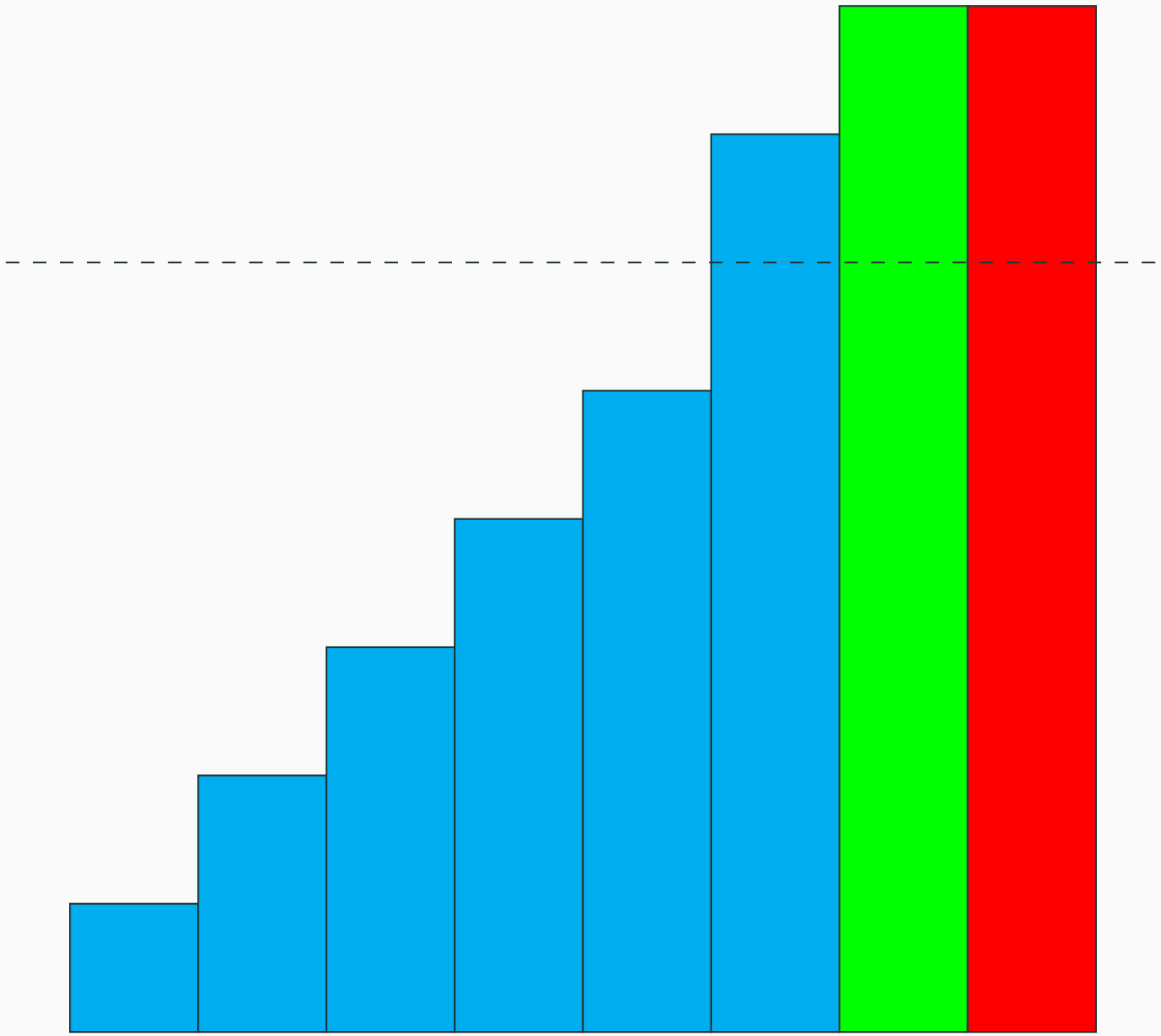
Exemple



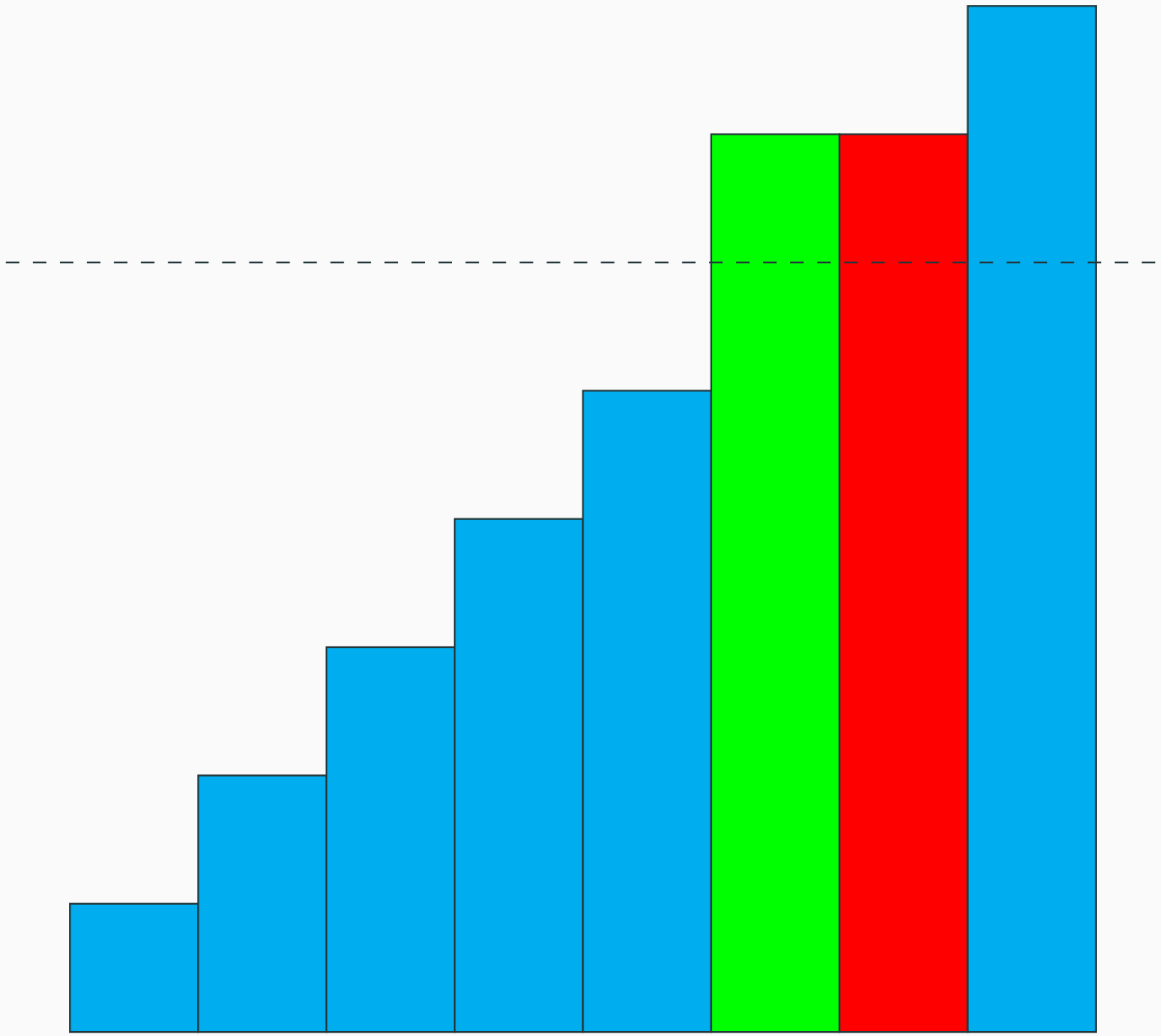
Exemple



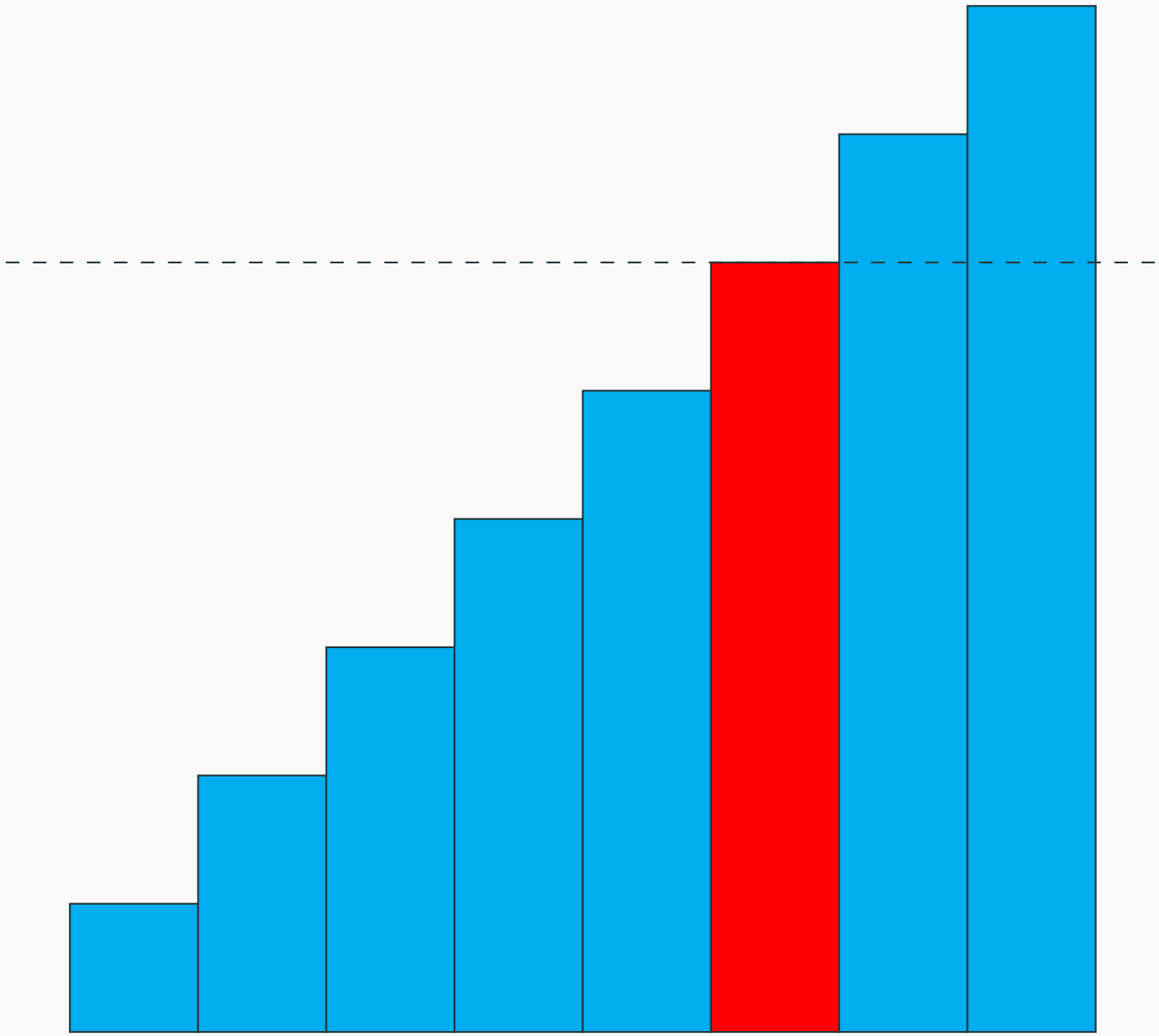
Exemple



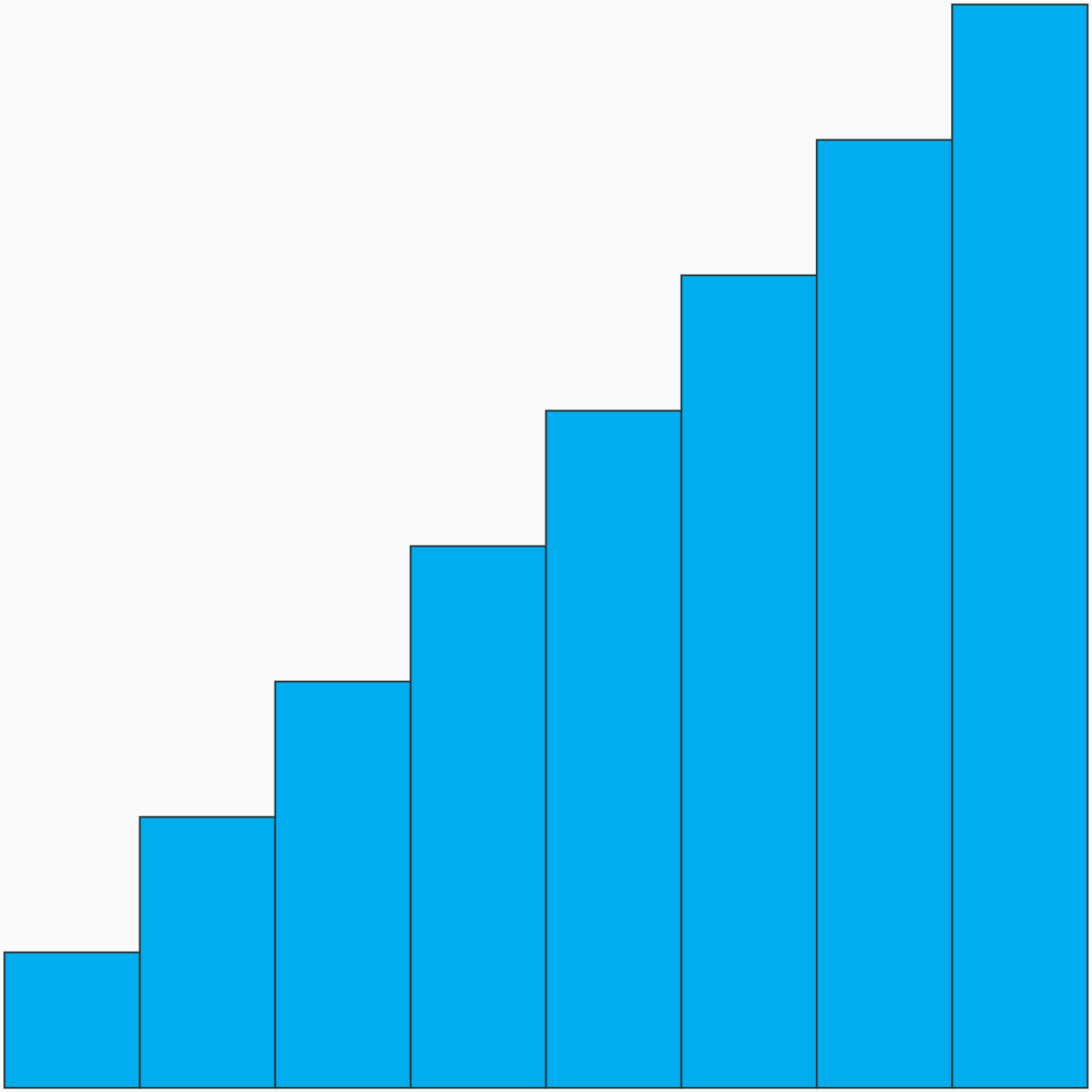
Exemple



Exemple



Exemple



Tri par insertion

Terminaison

L'algorithme de **tri par insertion** est constitué d'une boucle `while` dans une boucle `for`.

Il faut donc montrer que pour tout $i \in [\![1, n-1]\!]$, la boucle `while` termine, ce qui est à peu près évident.

La variable j (on identifie encore référence et valeur) est initialisée à i juste avant la boucle, la condition de continuation du `while` comporte notamment la condition $j > 0$ et j est **décrémenté** à chaque tour de boucle.

Tri par insertion

Remarque

Notons que les indices du tableau considérés ne produisent jamais d'erreurs (d'accès en dehors du tableau).

Remarquez que si $j = 0$ au niveau de la condition du `while`, alors la condition `!j>0` n'est pas vérifiée et on n'a pas besoin d'évaluer `t.(!j-1) > x` (qui produirait un dépassement d'indice) pour s'apercevoir que la condition

```
!j>0 && t.(!j-1) > x
```

est fausse.

Ceci est dû au comportement **paresseux** de l'opérateur `&&`.

Tri par insertion

Complexité

Dans le **meilleur** des cas, on ne passe jamais dans la boucle `while`, et la complexité est en $O(n)$.
C'est le cas si le tableau est déjà triée.

Dans le **pire** des cas, on passera toujours j fois dans la boucle `while`, est la complexité est en $O(n^2)$.
C'est le cas si le tableau est triée dans l'ordre décroissant au départ.

Preuve de correction du tri par insertion

La boucle `while` admet pour invariant :

$$\text{Inv} \ : \ \forall k \text{ tel que } j < k \leq i, \texttt{t.(k)>x}$$

En sortie de boucle `while`, la condition `!j>0 && t.(!j-1) > x` est fausse, ainsi la boucle `for` possède l'invariant suivant :

$\text{Inv}'(i)$: Les éléments `t.(0),...,t.(i-1)` sont triés dans l'ordre croissant.

Preuve de correction du tri par insertion

$\mathrm{Inv}'(i)$: Les éléments `t.(0),...,t.(i-1)` sont triés dans l'ordre croissant.

En effet :

- Si $i = 1$, `t.(0)` tout seul forme bien un ensemble trié dans l'ordre croissant, donc $\mathrm{Inv}'(1)$ est vrai avant la boucle.
- Si, pour $i \in [\![1, n-1]\!]$, $\mathrm{Inv}'(i)$ est vrai en haut du corps de la boucle, alors $\mathrm{Inv}'(i+1)$ est vrai en bas du corps de la boucle. En effet, après l'exécution de la boucle `while`, les éléments `t.(k)` pour $j < k \leq i$ sont $>$ à x, et ceux avant l'indice j (exclus) sont $\leq$ (avec j éventuellement nul). Ainsi, placer x en position j mène à la portion `t.(0),...,t.(i)` triée.

Preuve de correction du tri par insertion

Correction

Au final, après la boucle `for`, les éléments `t.(0)`,...,`t.(n-1)` sont triés dans l'ordre croissant, et la fonction est **correcte**.

Tri à bulles

Idée

L'algorithme du **tri à bulles** parcourt le tableau, et compare les couples d'éléments **successifs**.

Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont **échangés**.

Si au moins un échange a eu lieu pendant le parcours, l'algorithme procède à un nouveau parcours.

S'il n'y a pas eu d'échange pendant un parcours, cela signifie que le tableau est trié et l'algorithme s'**arrête**.

Remarque

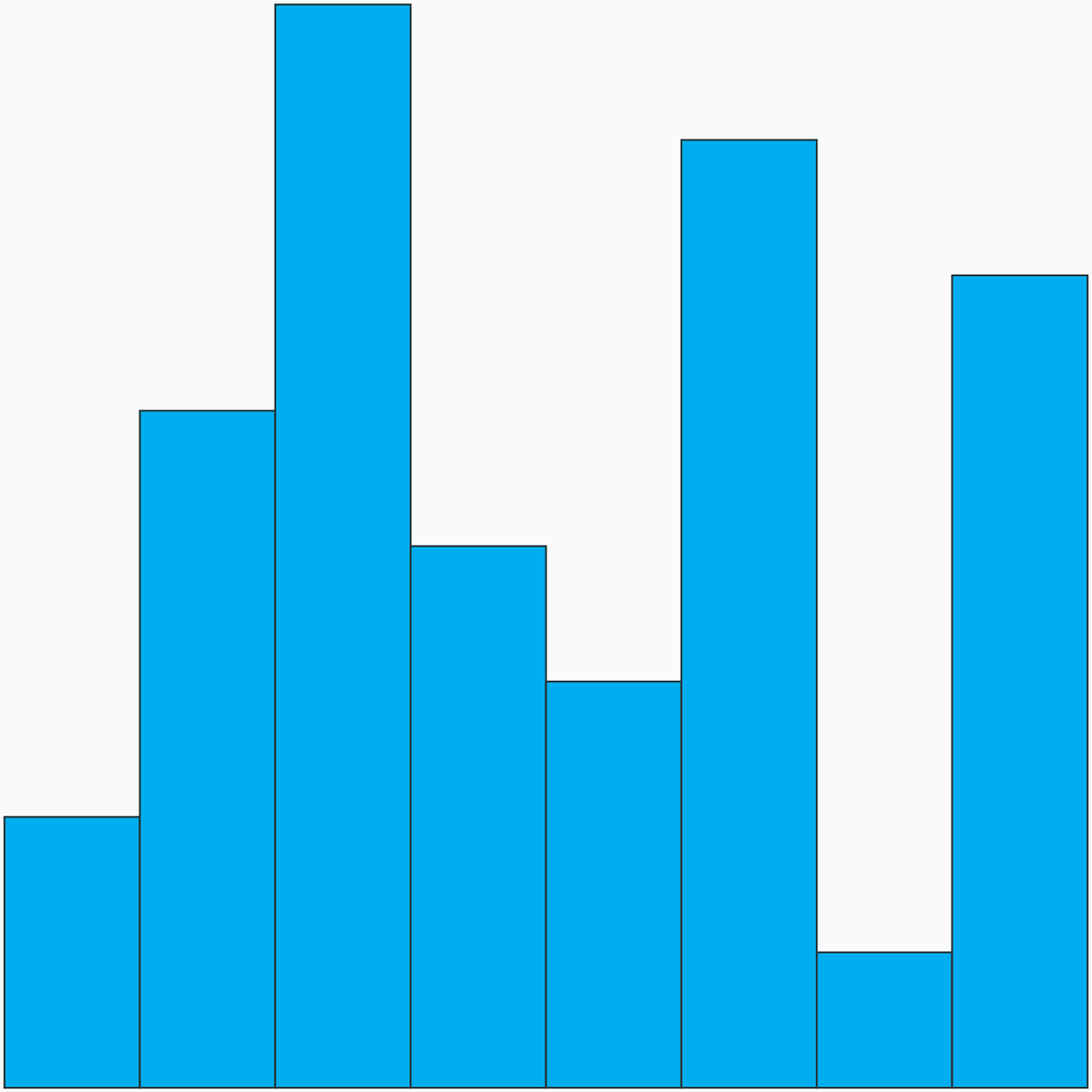
Ce tri est l'un des plus populaire, mais aussi l'un des moins efficaces.

Tri à bulles

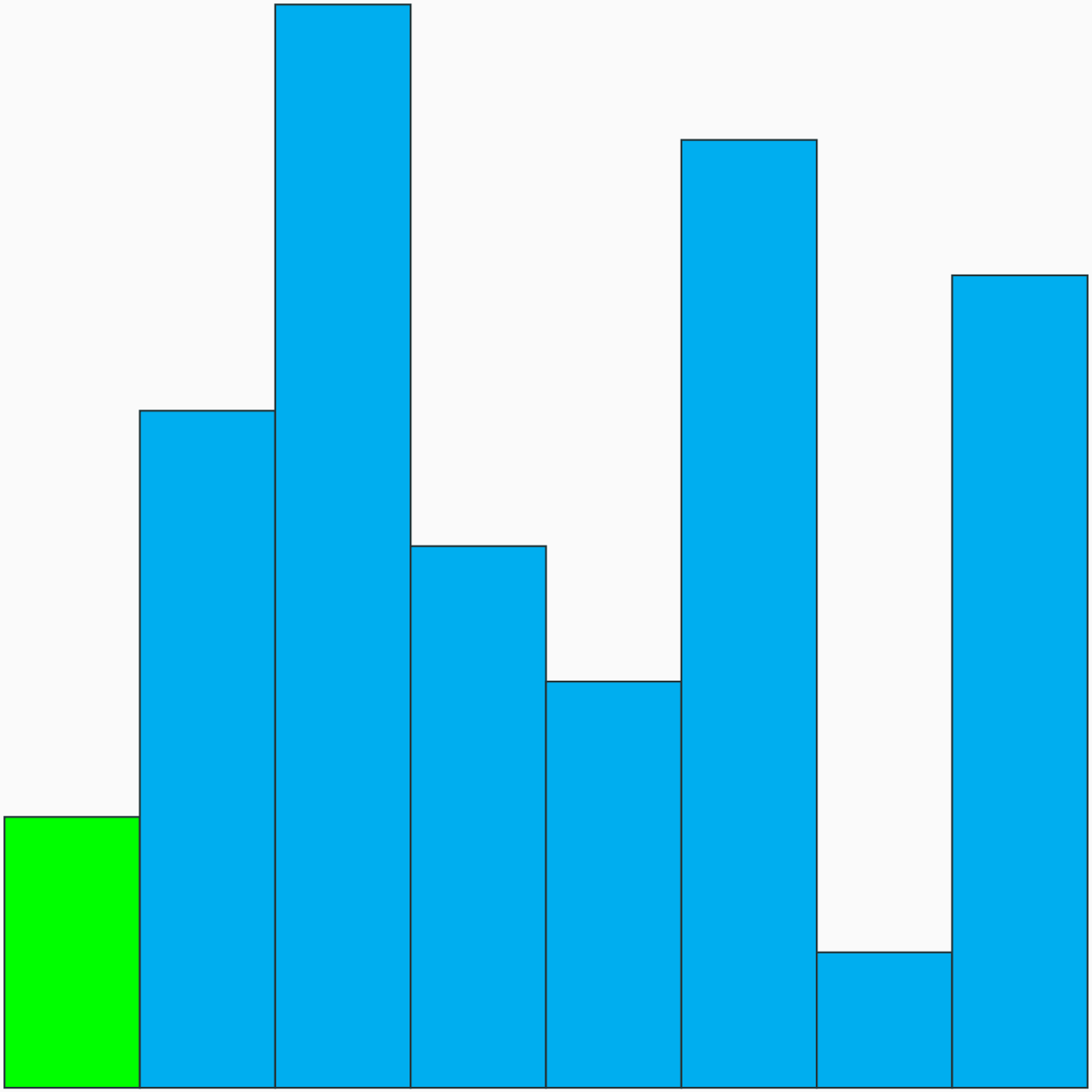
Le tri à bulles

```
let tri_bulles t =
  let n = Array.length t in
  let p = ref n and pasfini = ref true in
  while !pasfini do
    (* Inv: les éléments t.(p),...t.(n-1) sont triés,
     * et sont plus grands que les autres éléments de le tableau.
     * De plus, pasfini = false ou les éléments t.(0),...,t.(p-1) sont triés.
     *)
    pasfini := false ;
    for i=0 to !p-2 do
      (* Inv'(i): t.(i) est plus grand que tous les éléments à sa gauche. *)
      if t.(i) > t.(i+1) then
        begin
          echange t i (i+1) ;
          pasfini := true
        end
      (* Inv'(i+1) *)
    done ;
    decr p (* À chaque nouveau parcours, on peut en fait s'arrêter un élément plus tôt. *)
    (* Inv *)
  done
;;
```

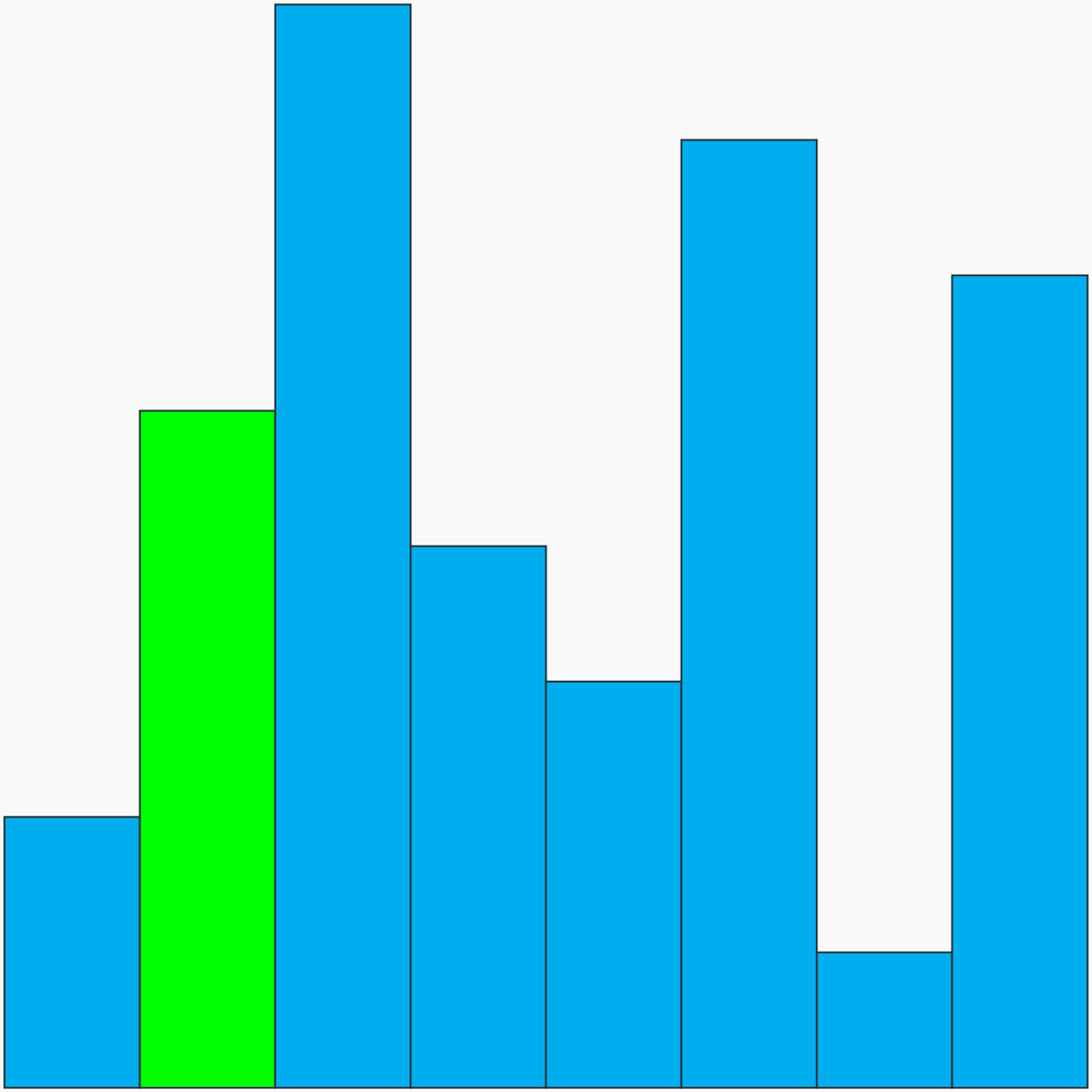
Exemple



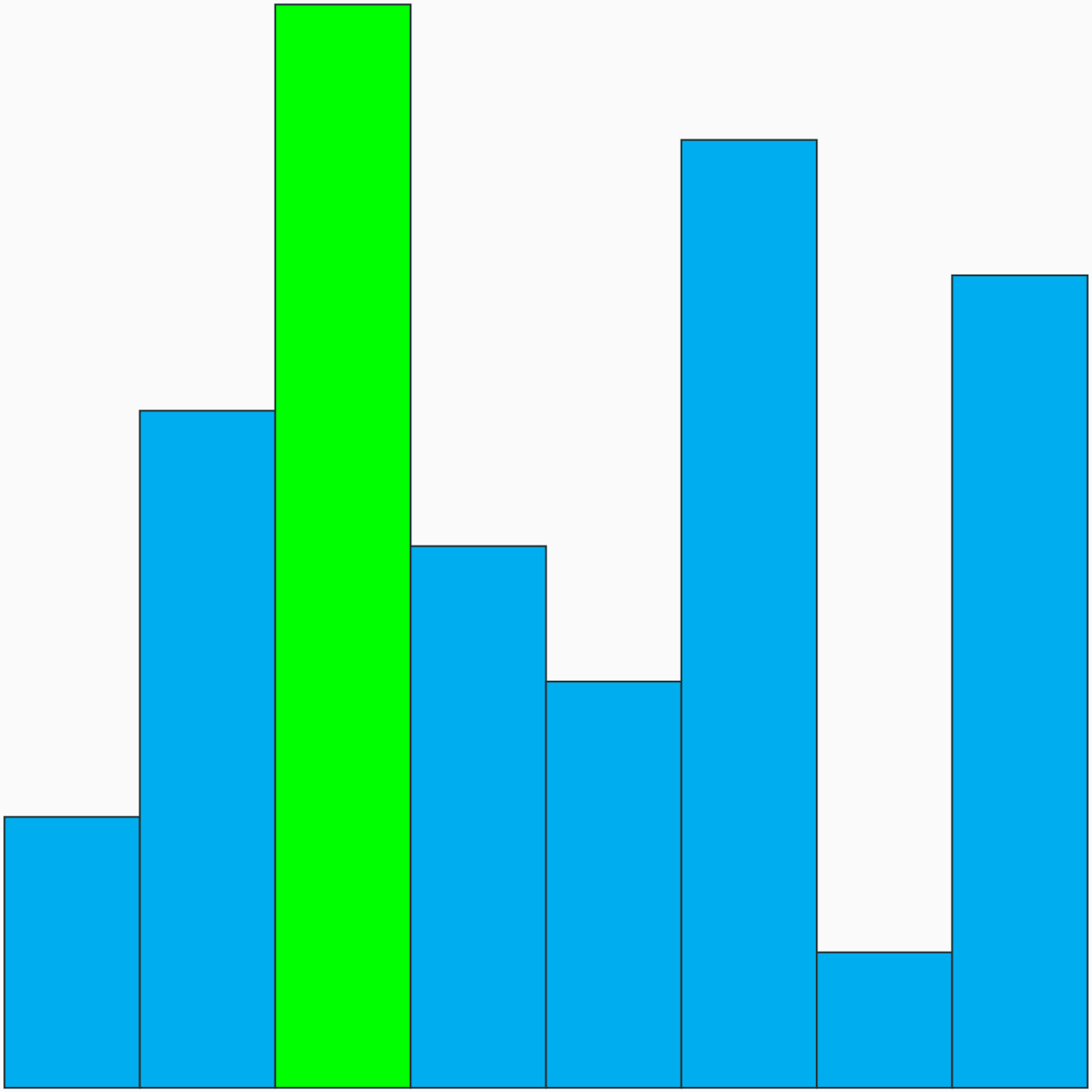
Exemple



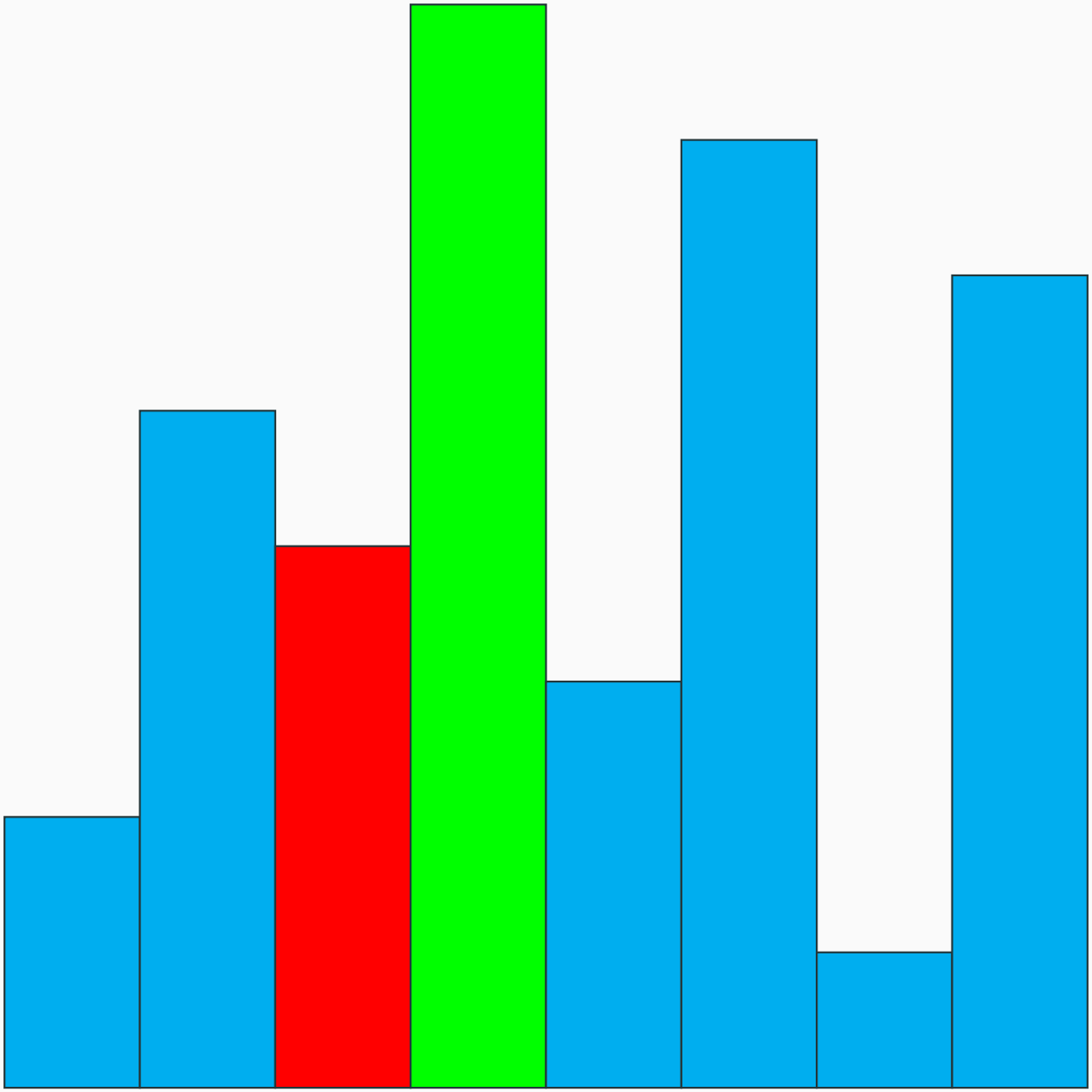
Exemple



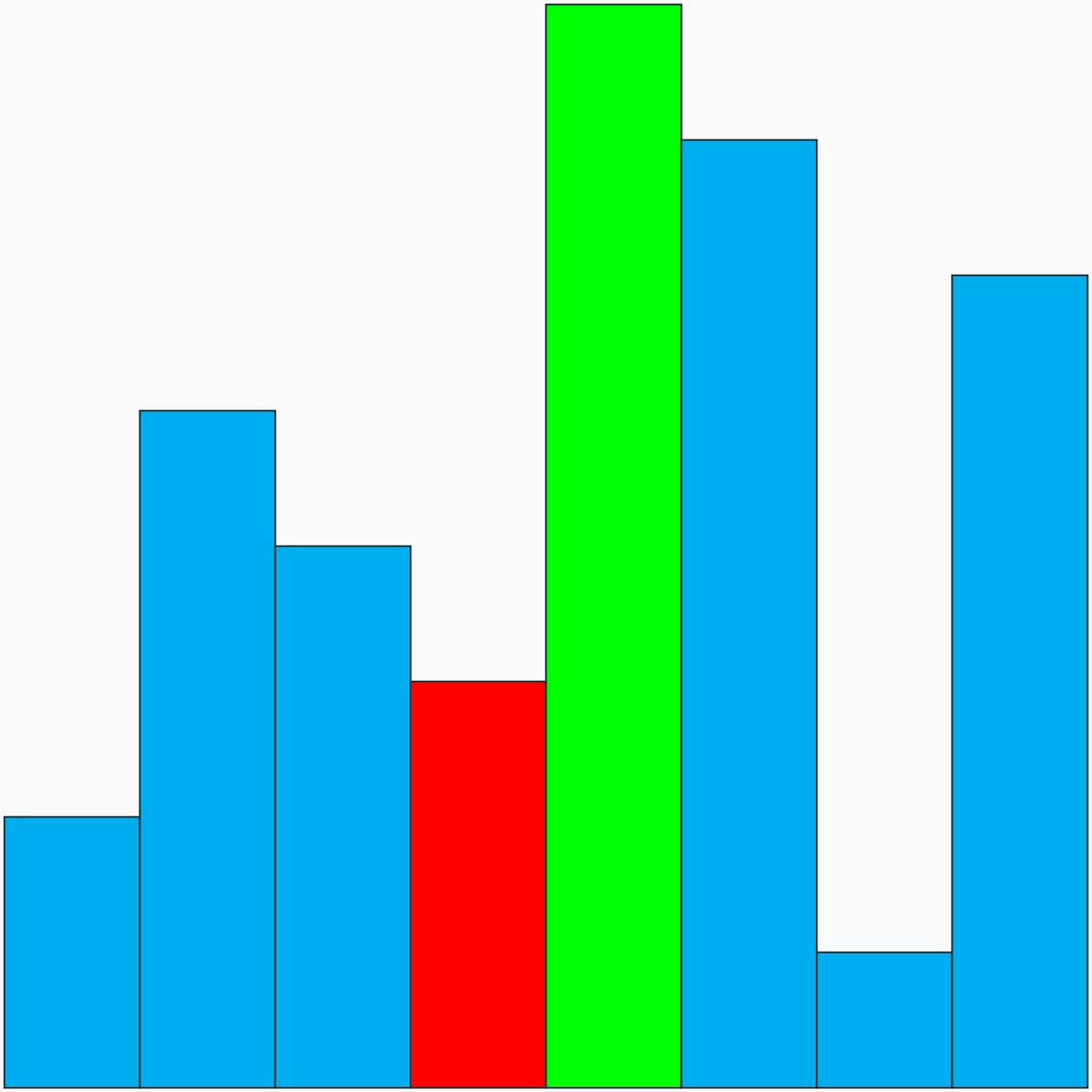
Exemple



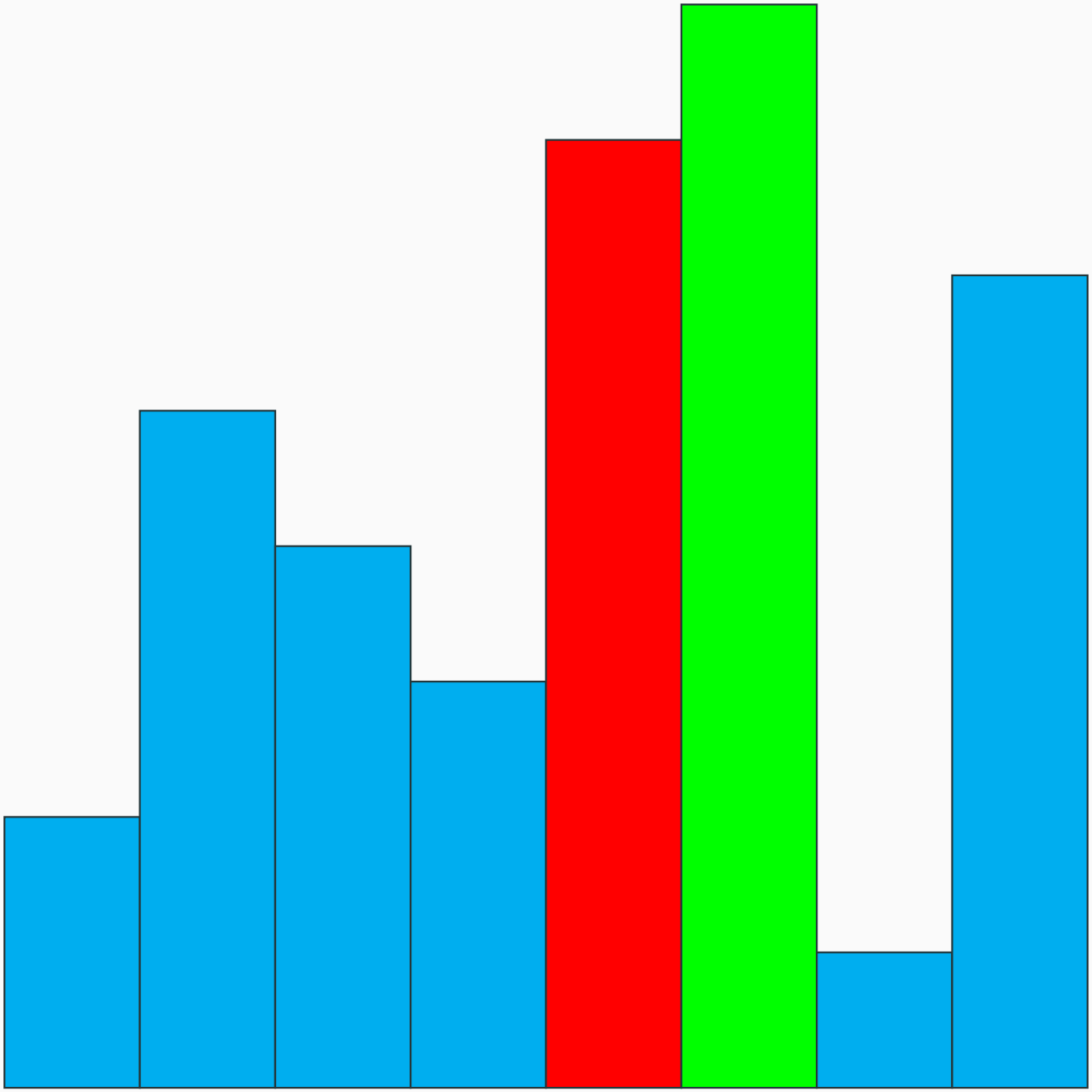
Exemple



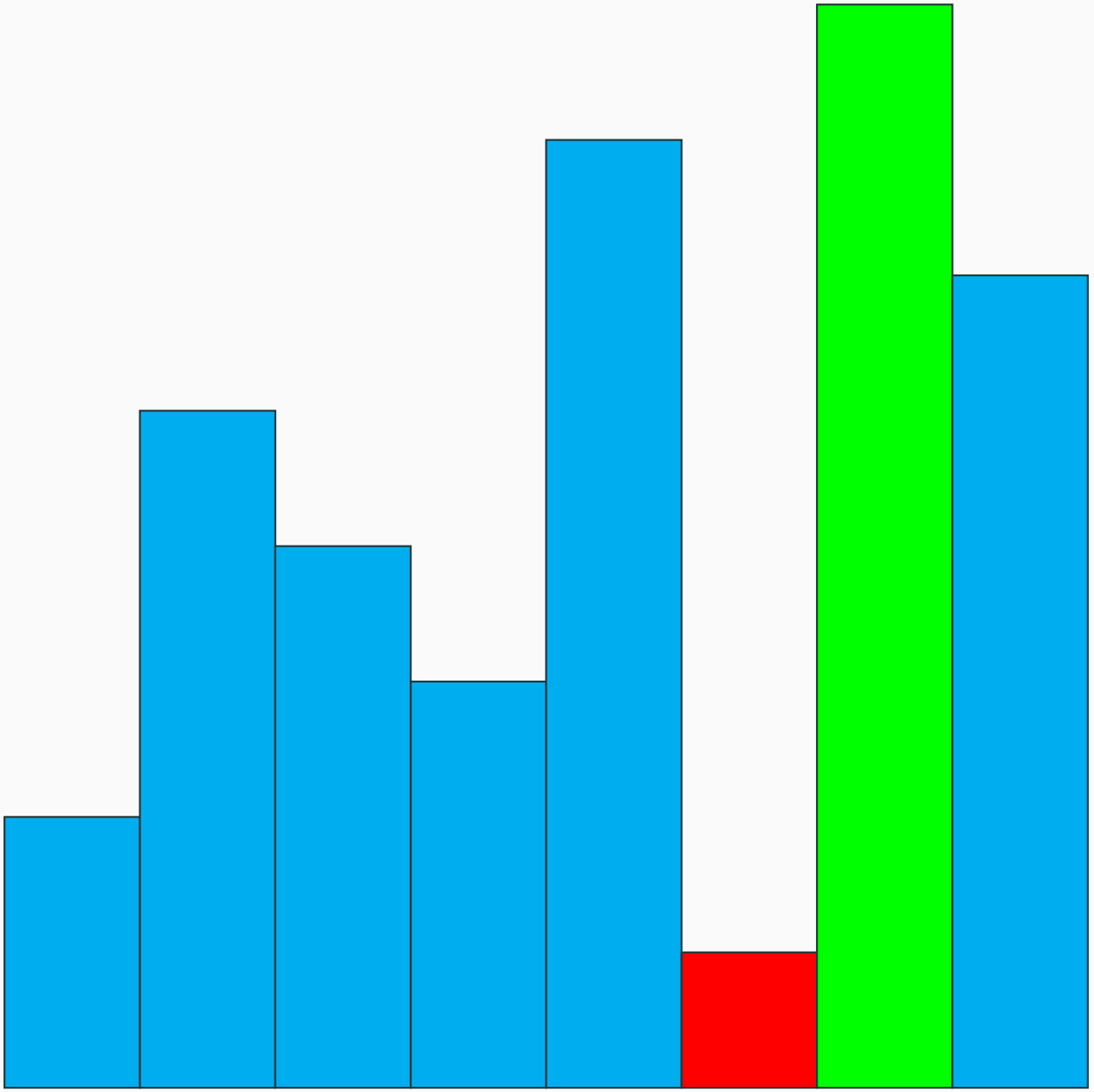
Exemple



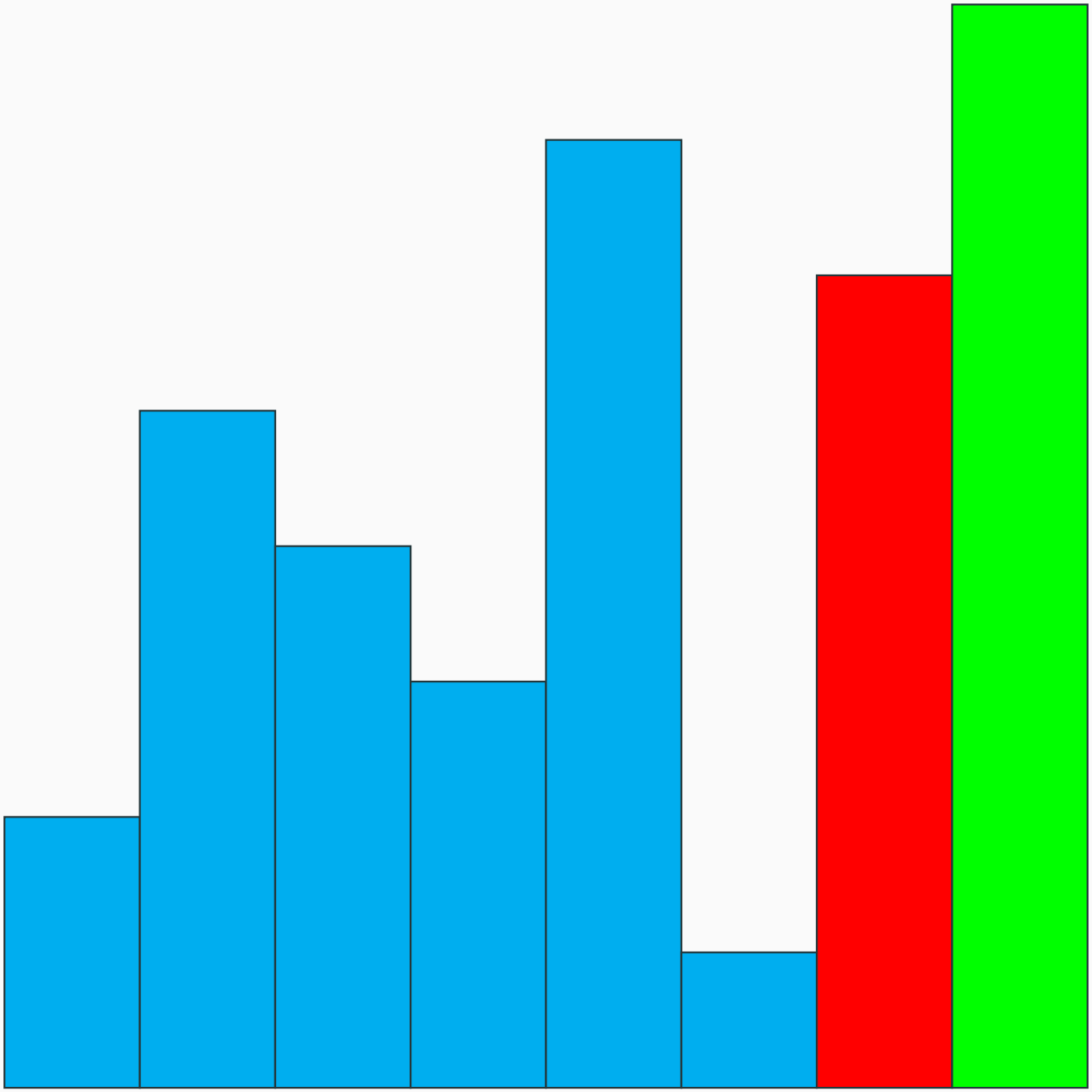
Exemple



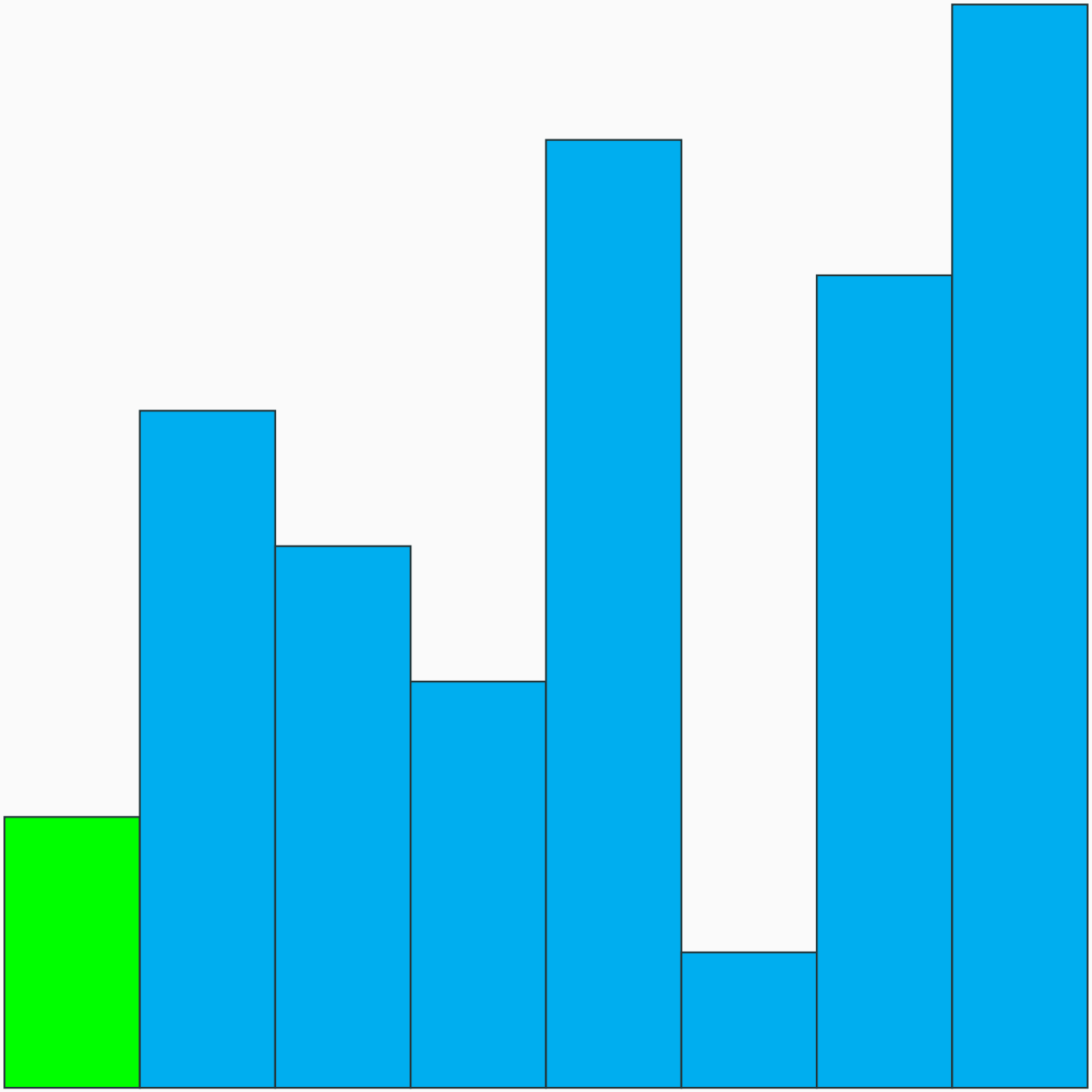
Exemple



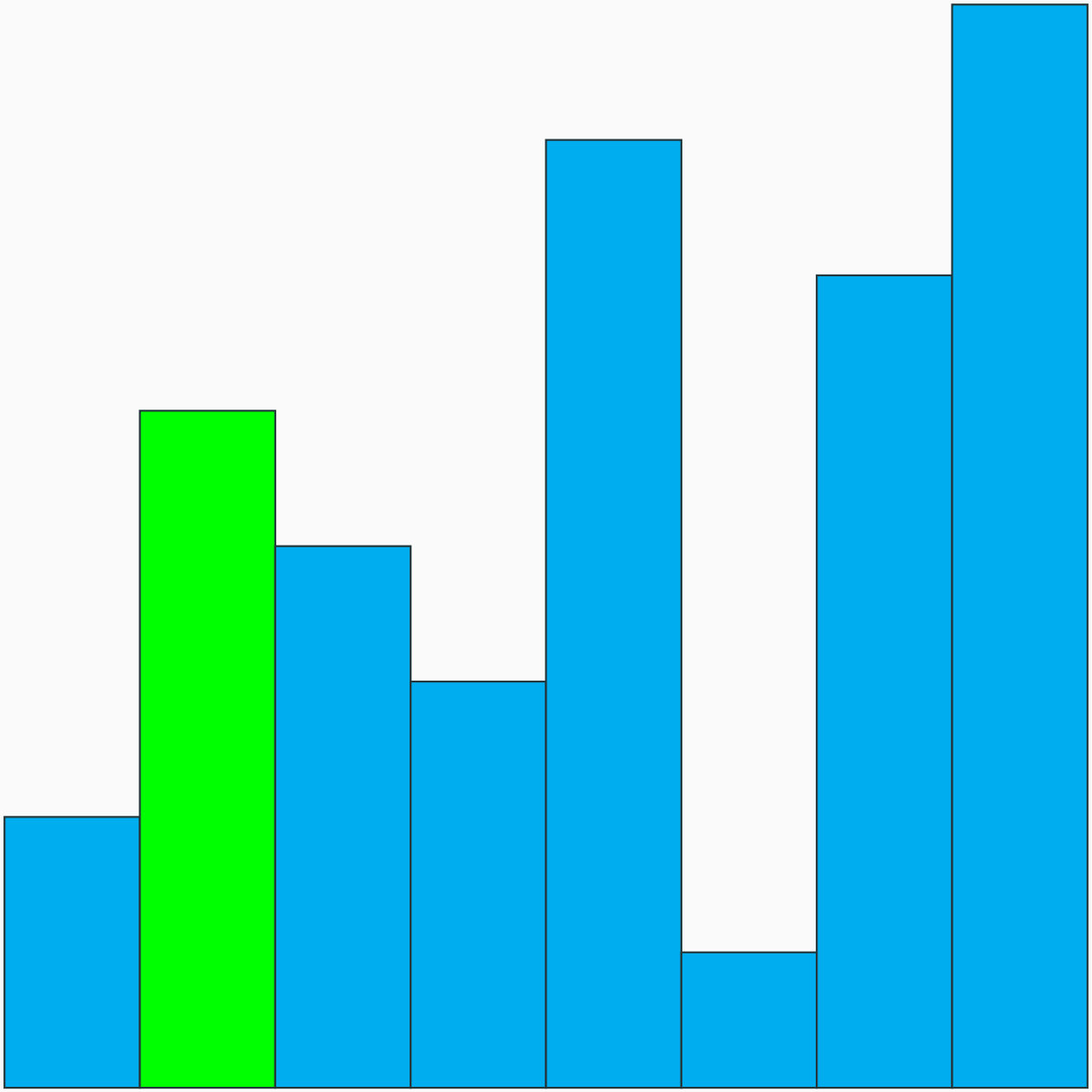
Exemple



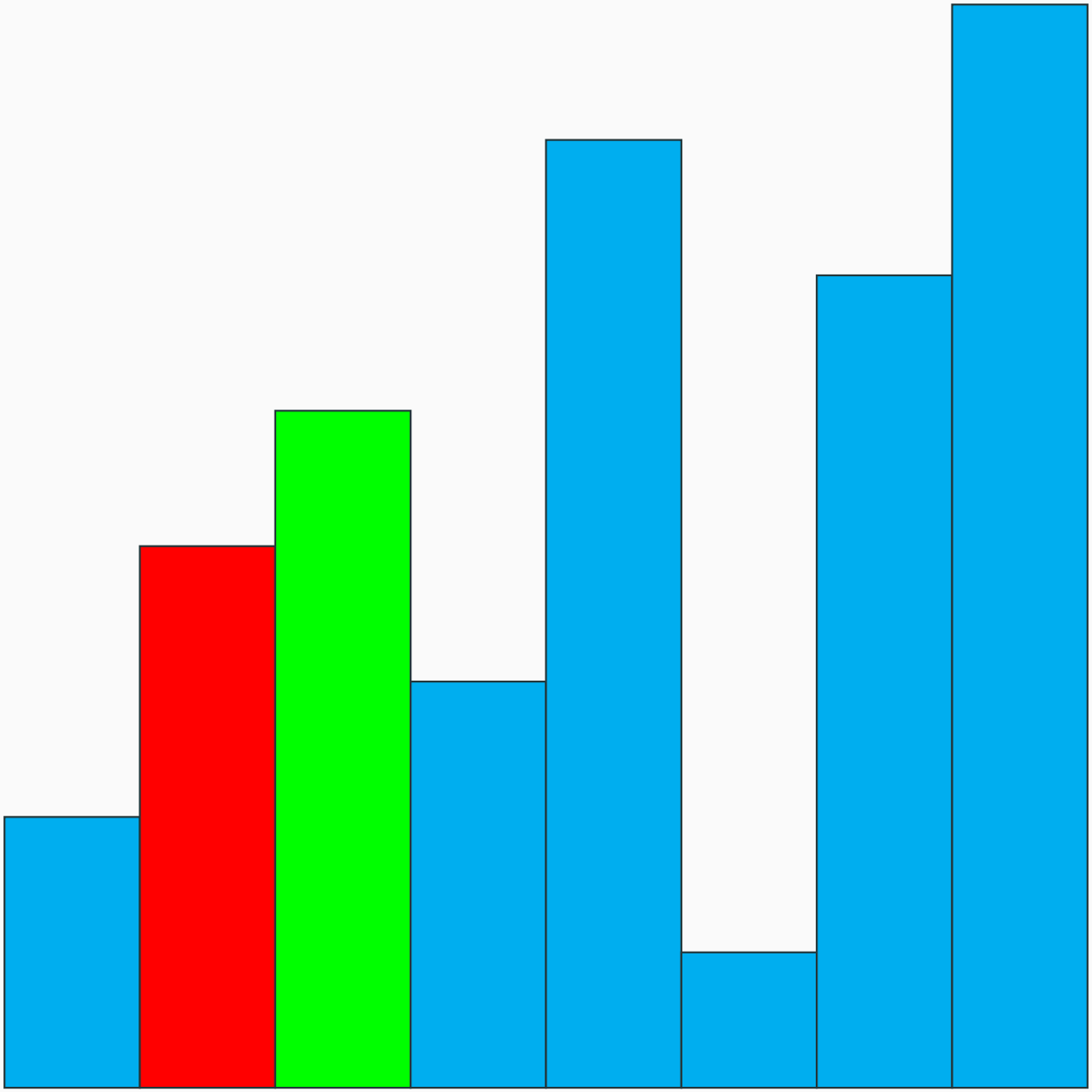
Exemple



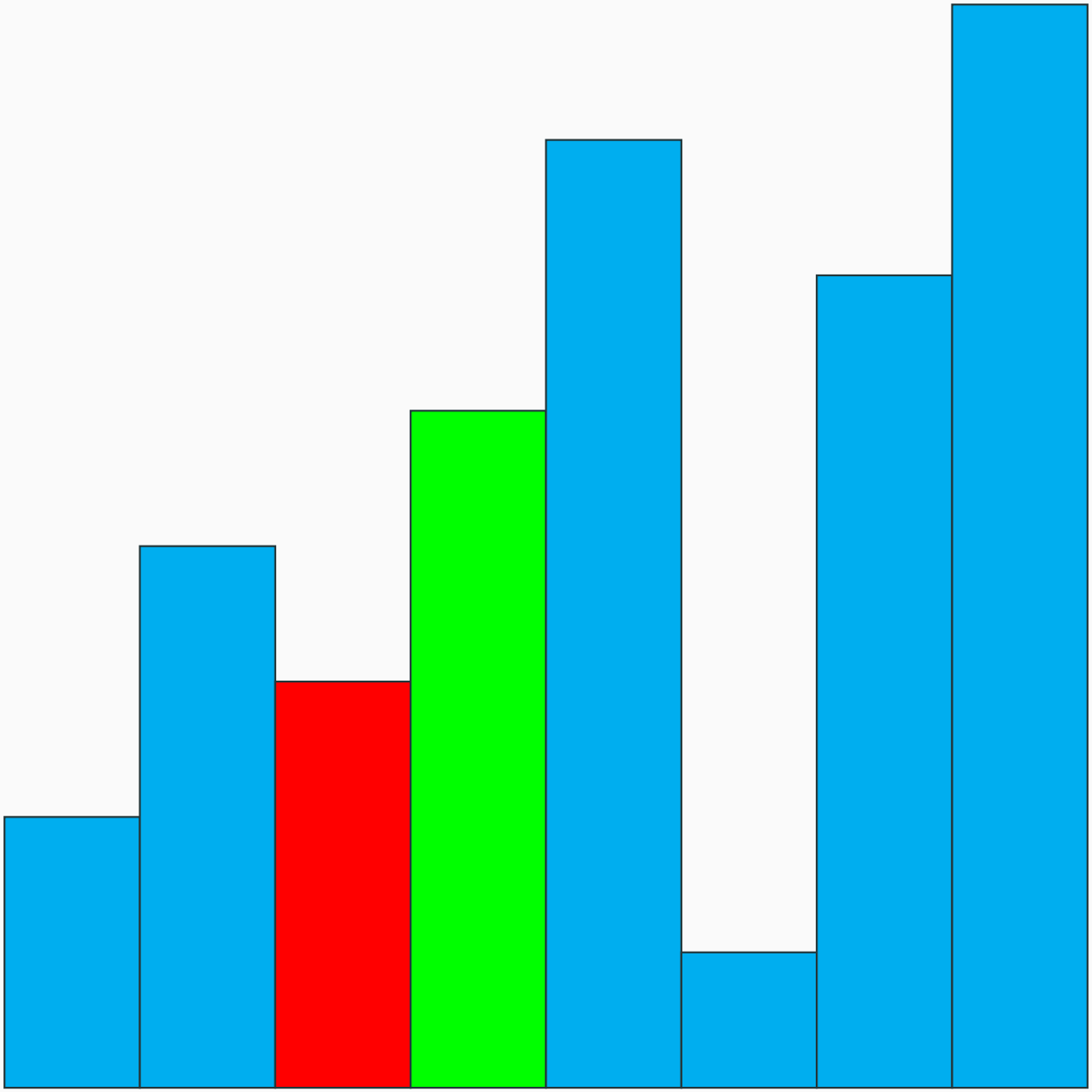
Exemple



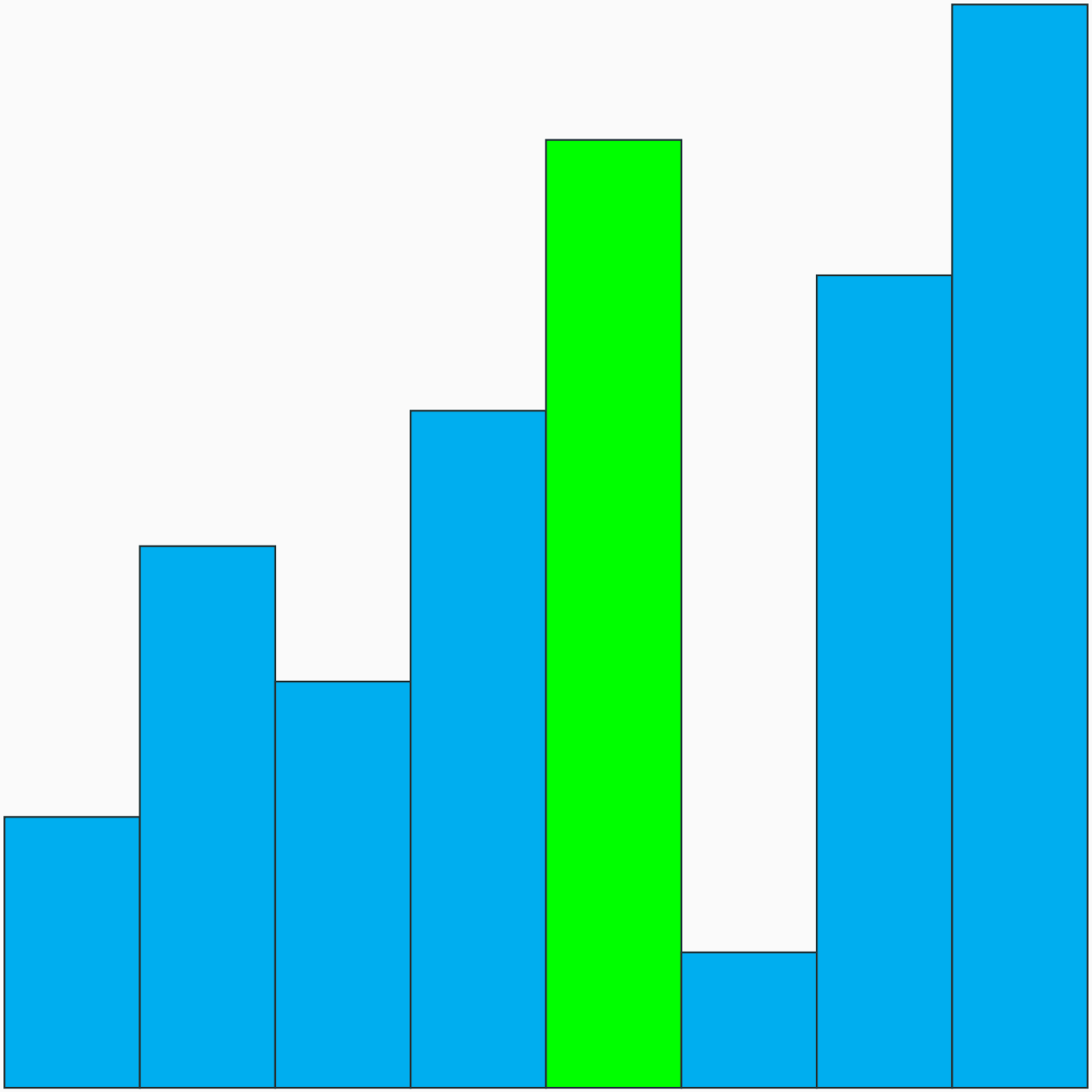
Exemple



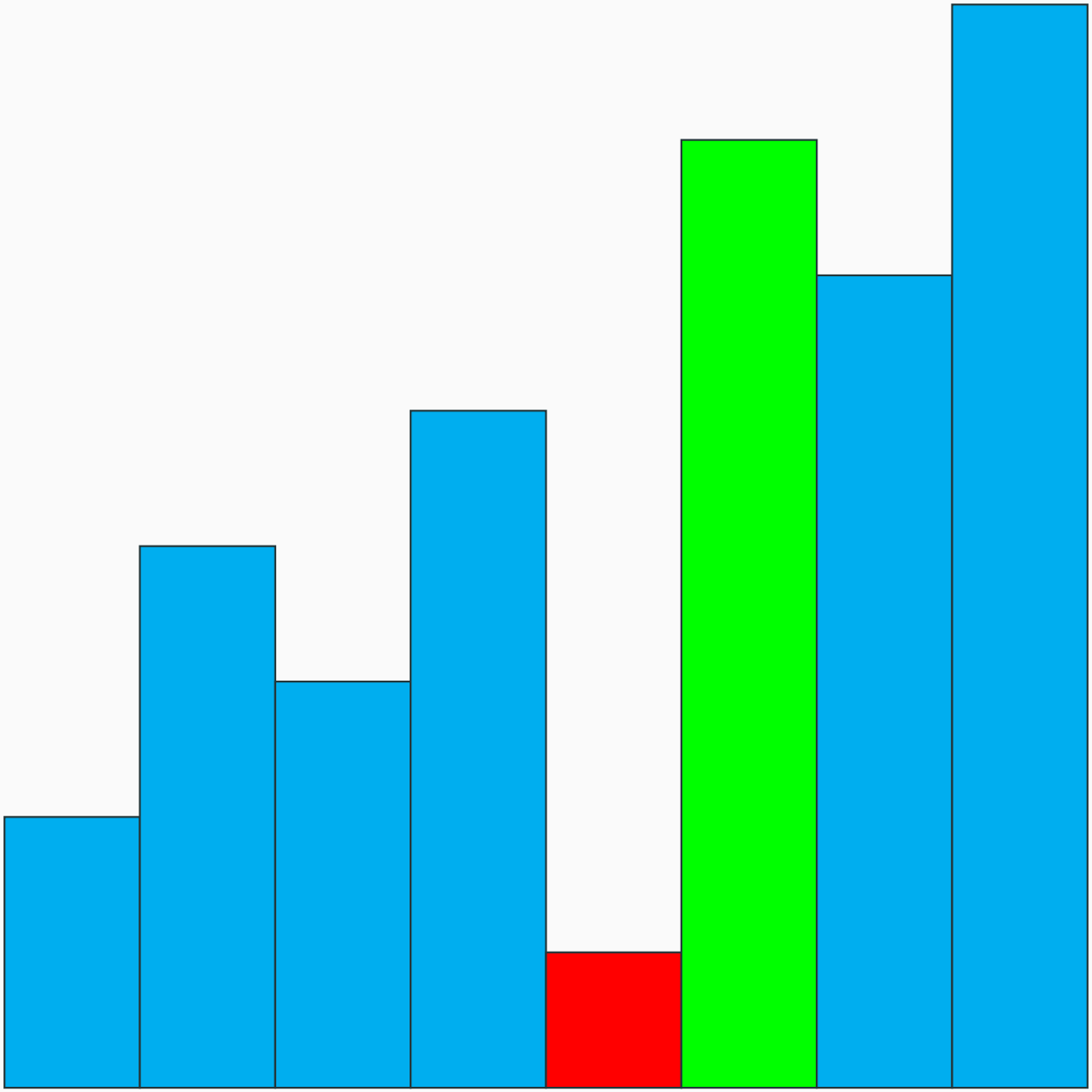
Exemple



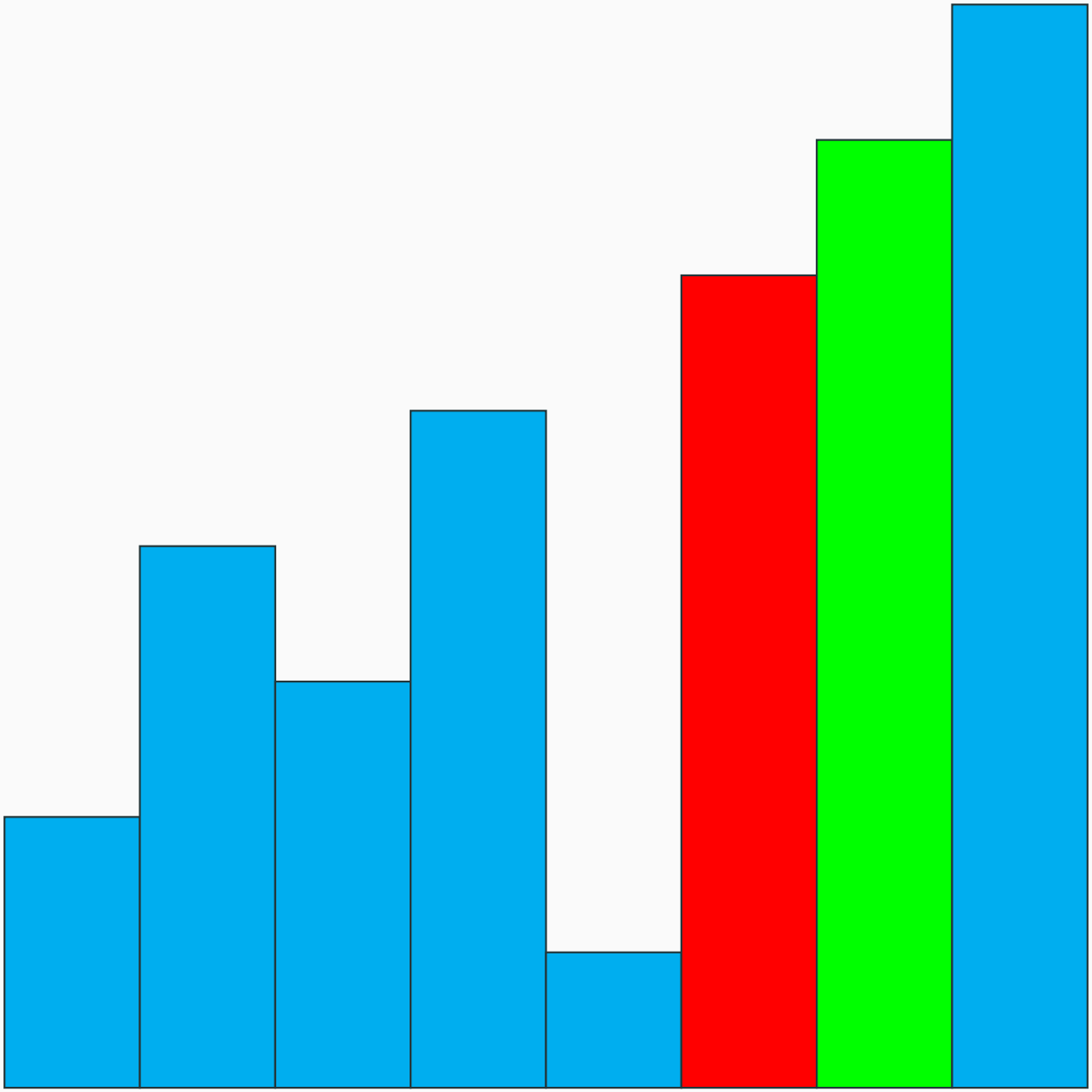
Exemple



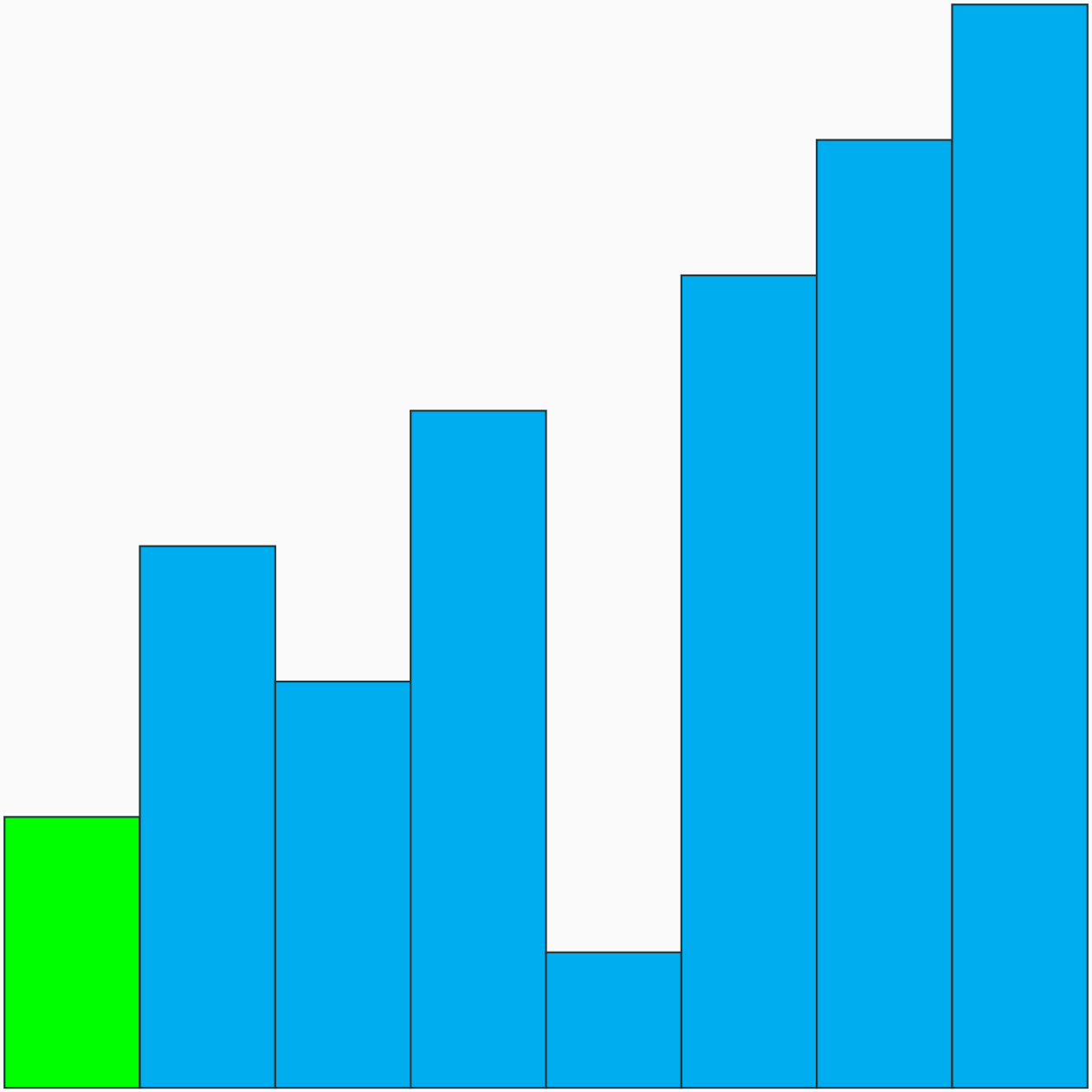
Exemple



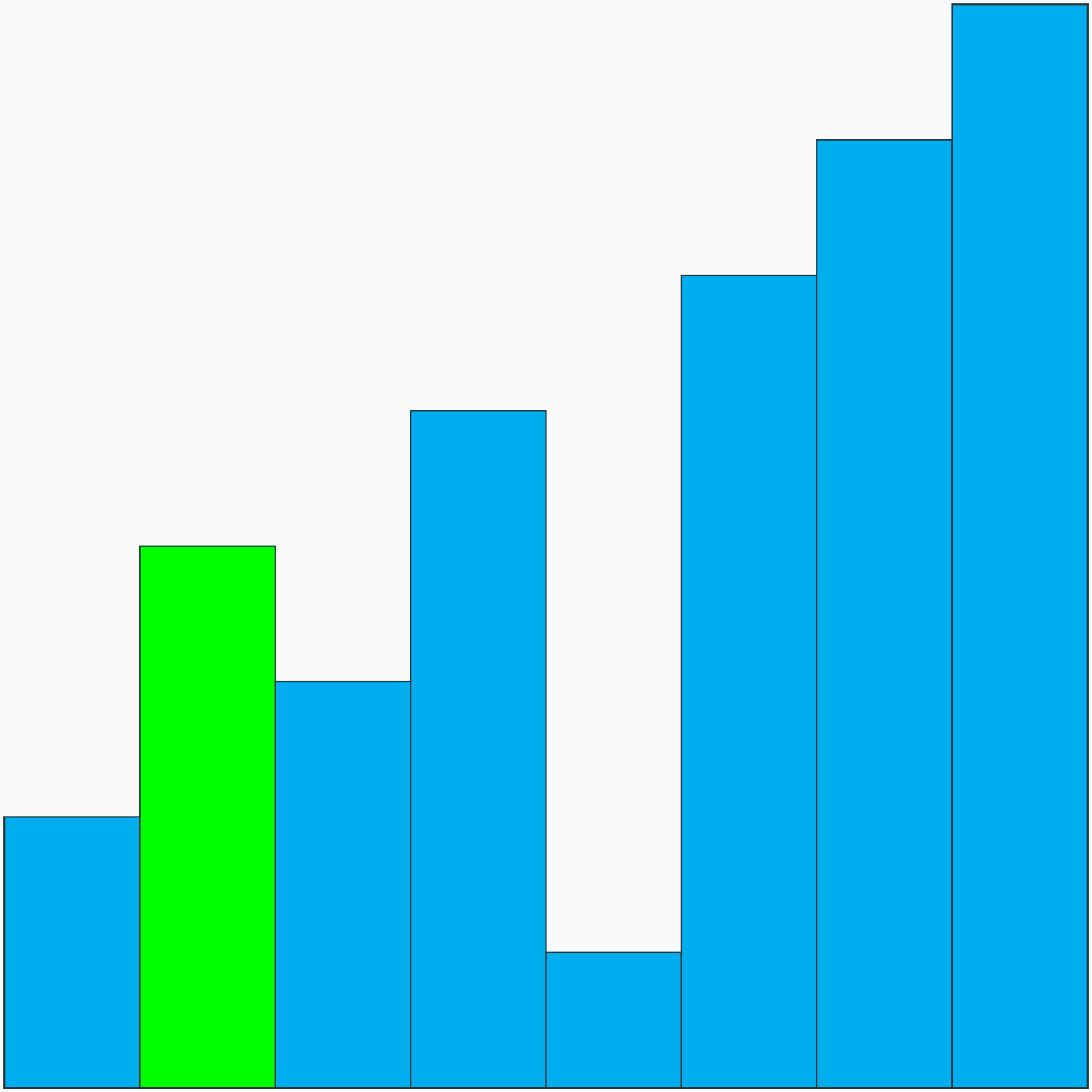
Exemple



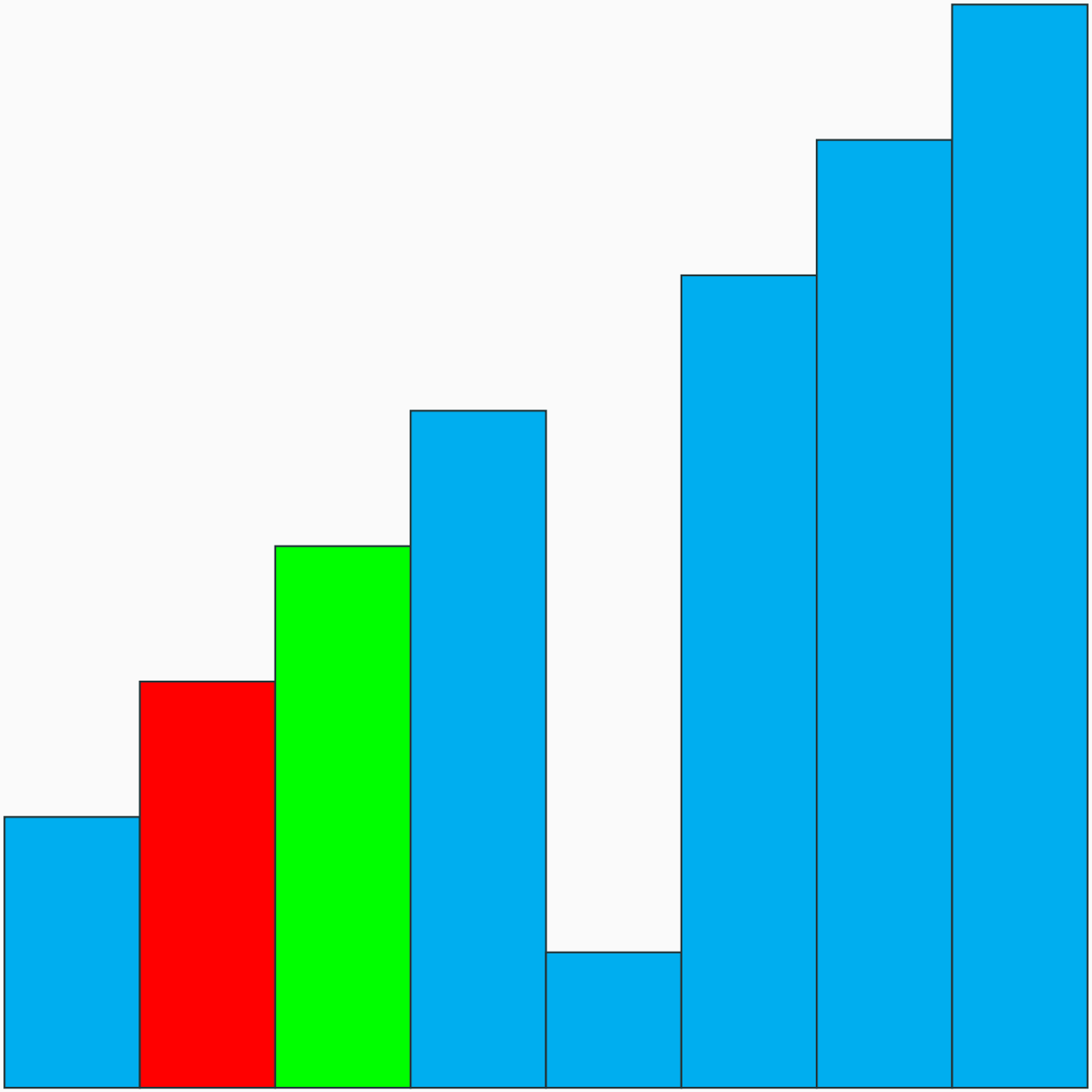
Exemple



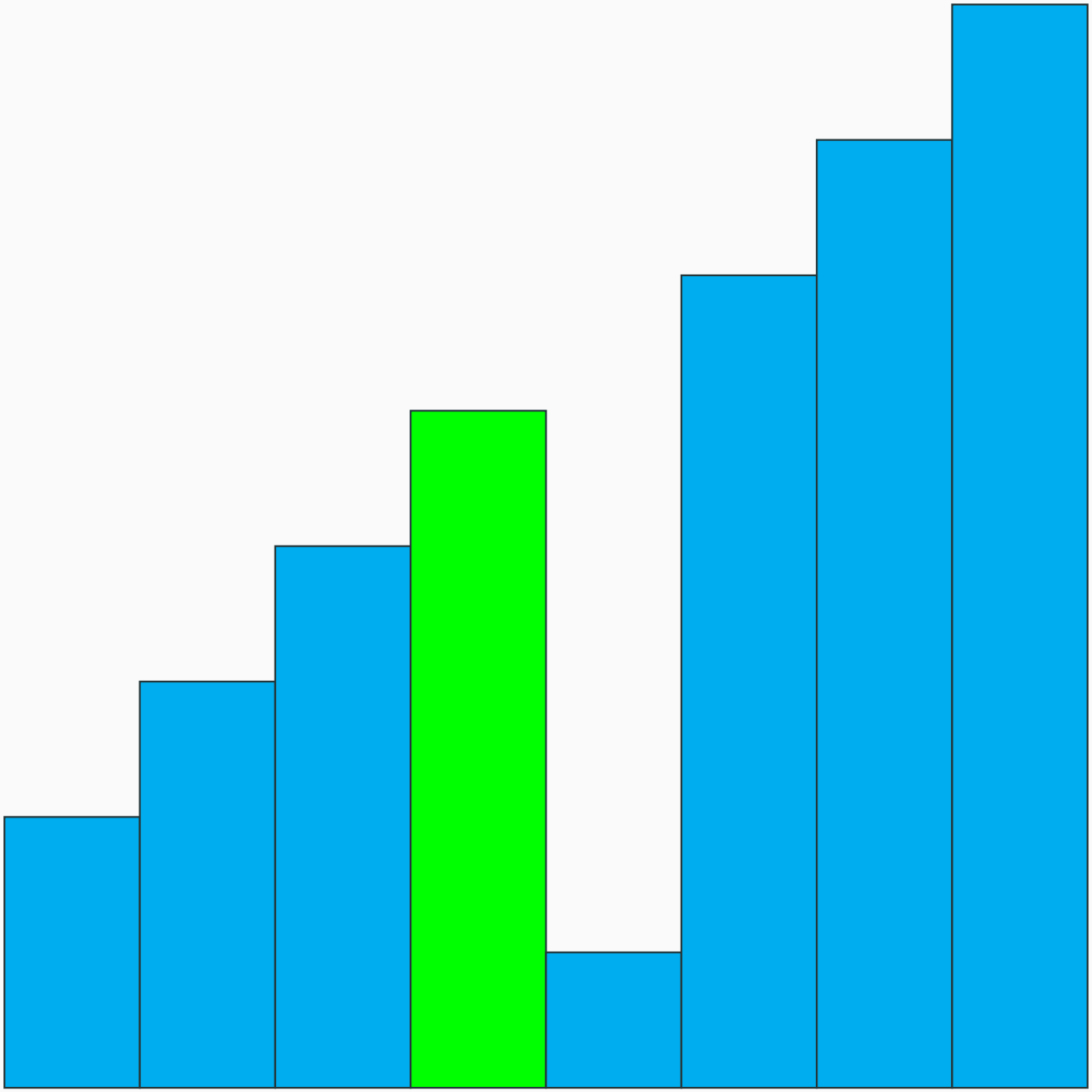
Exemple



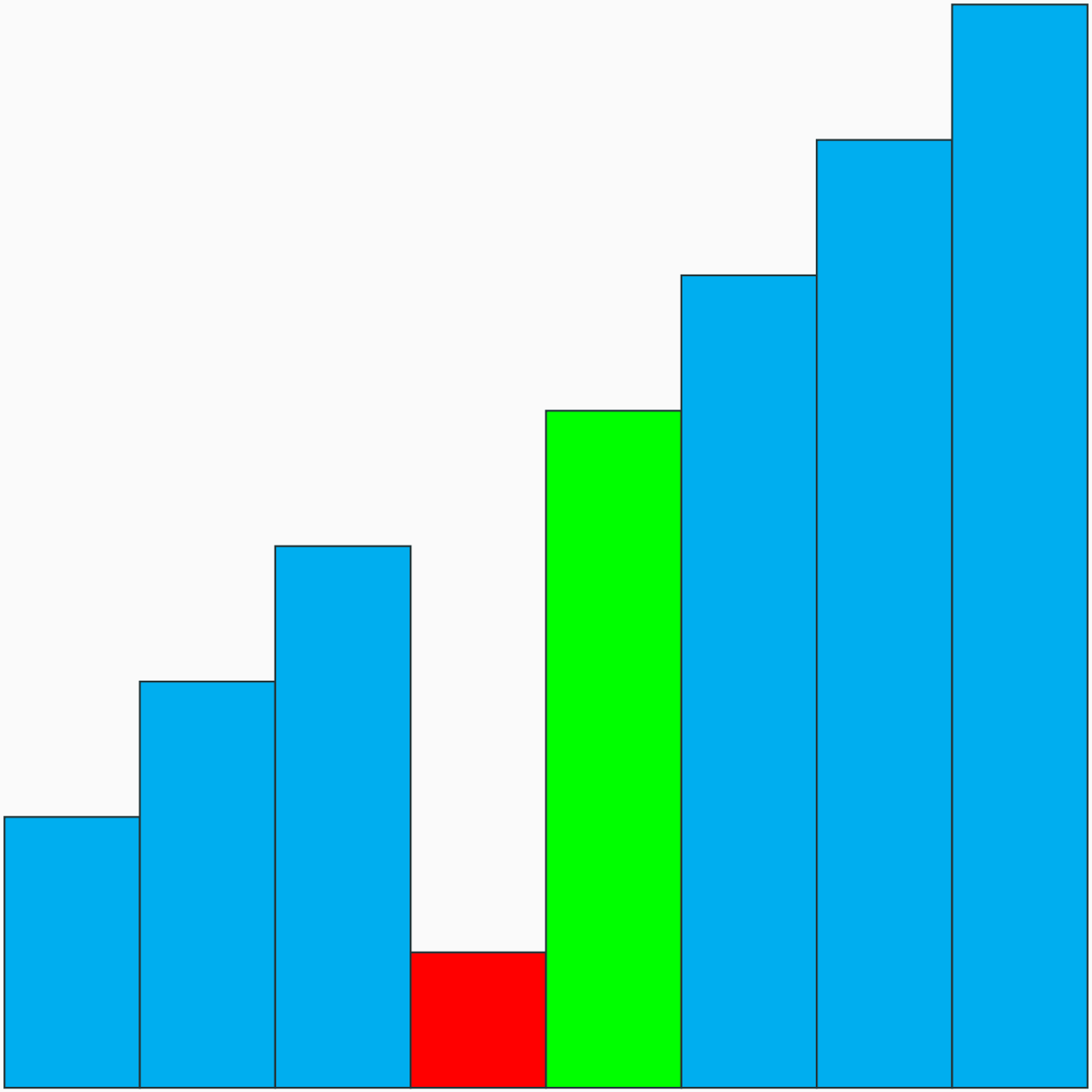
Exemple



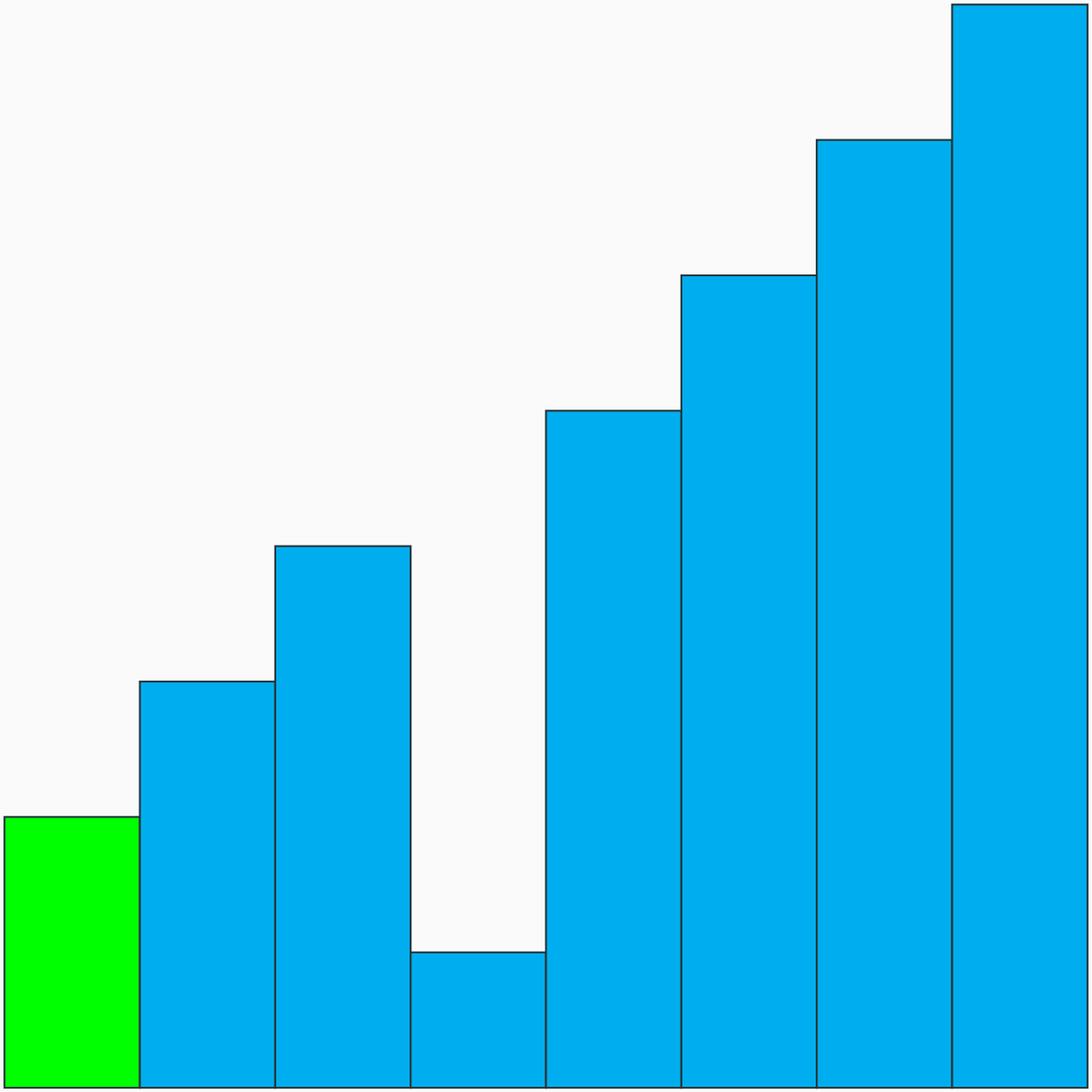
Exemple



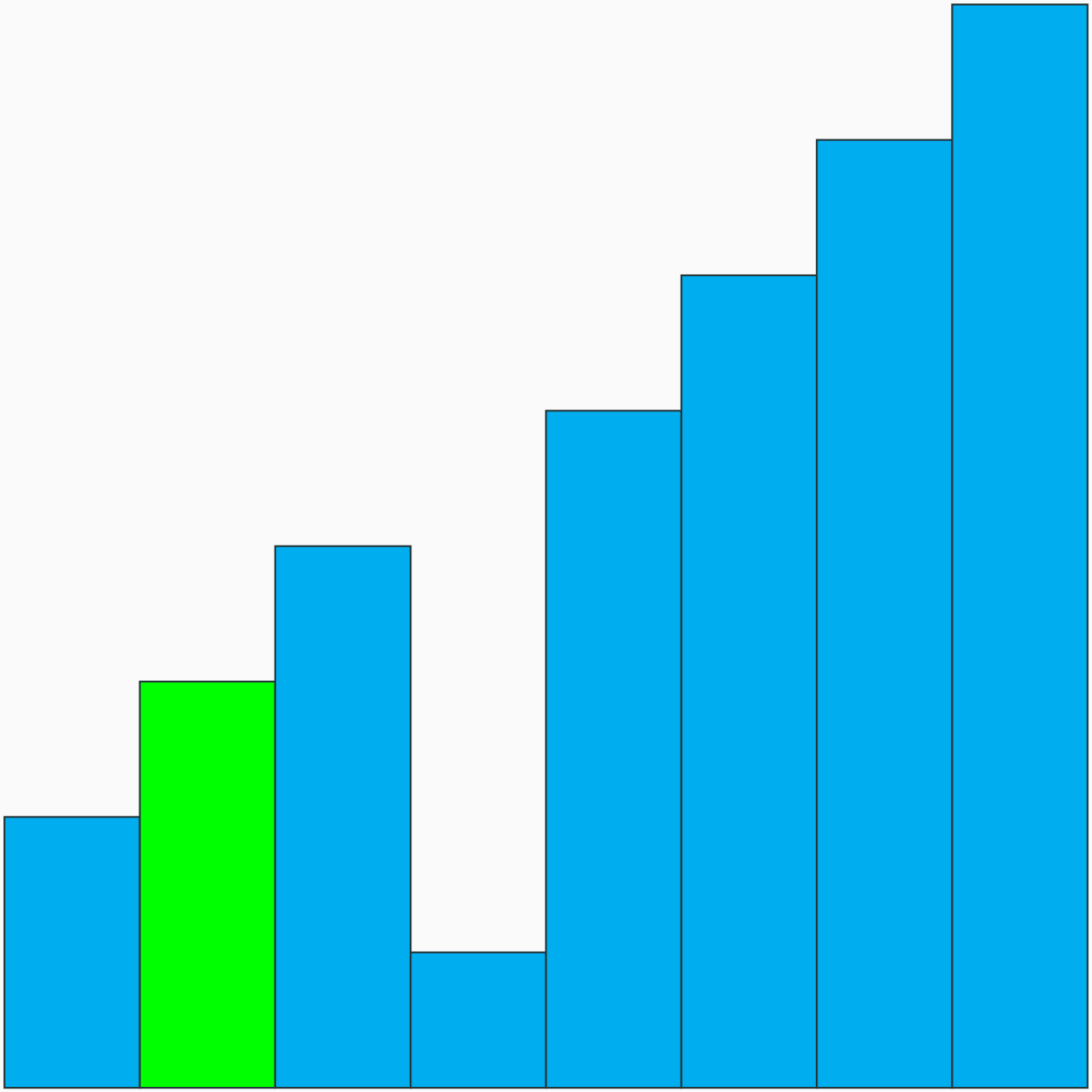
Exemple



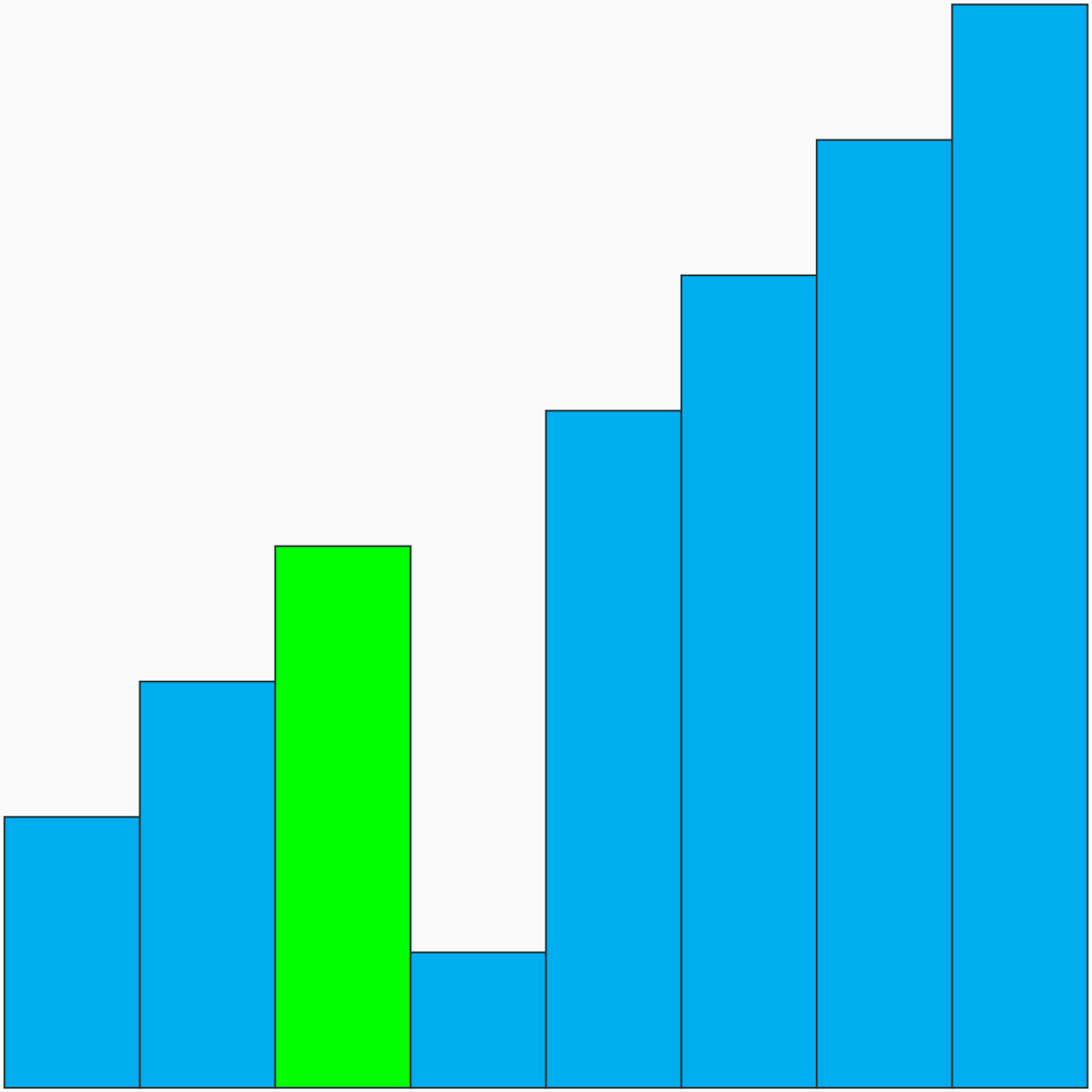
Exemple



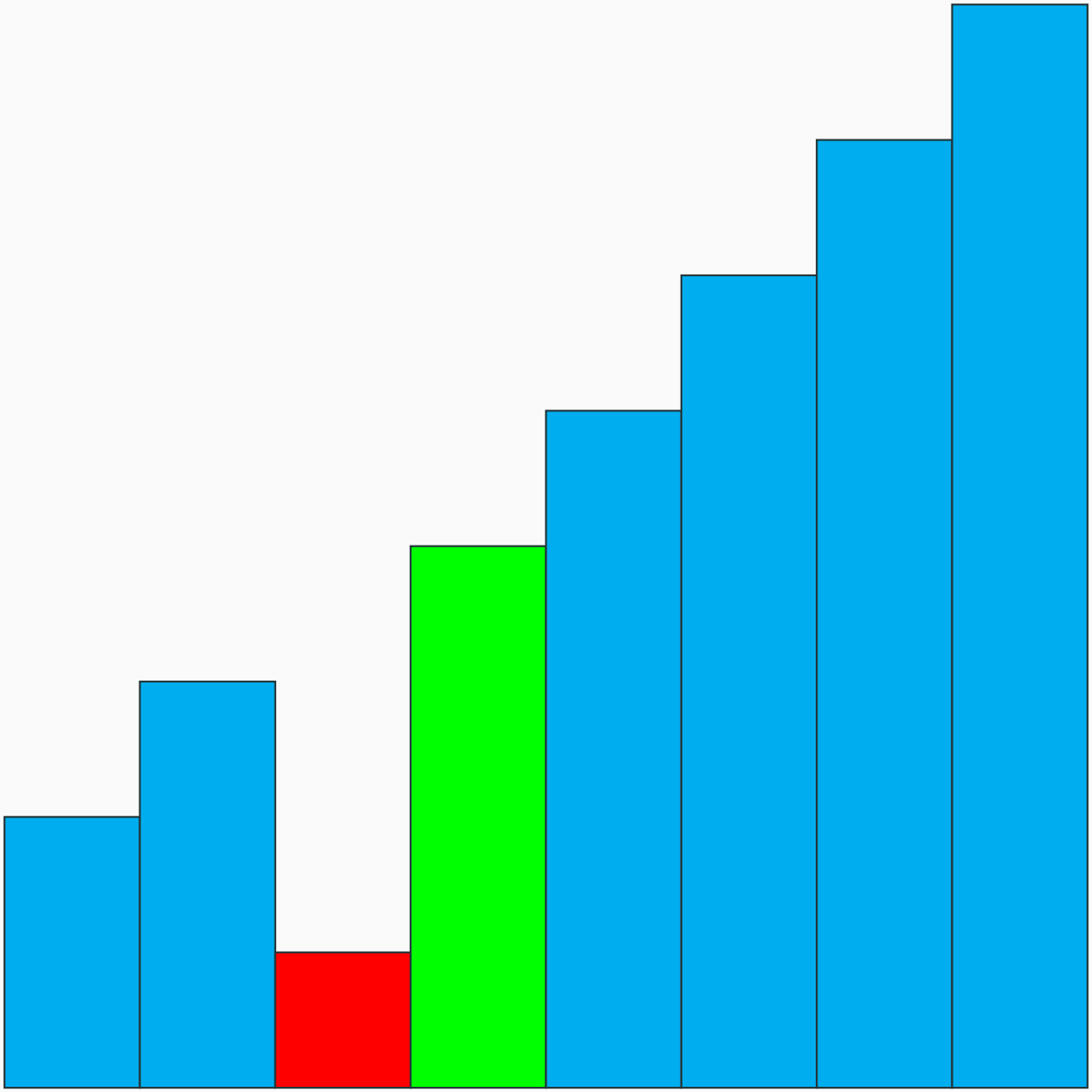
Exemple



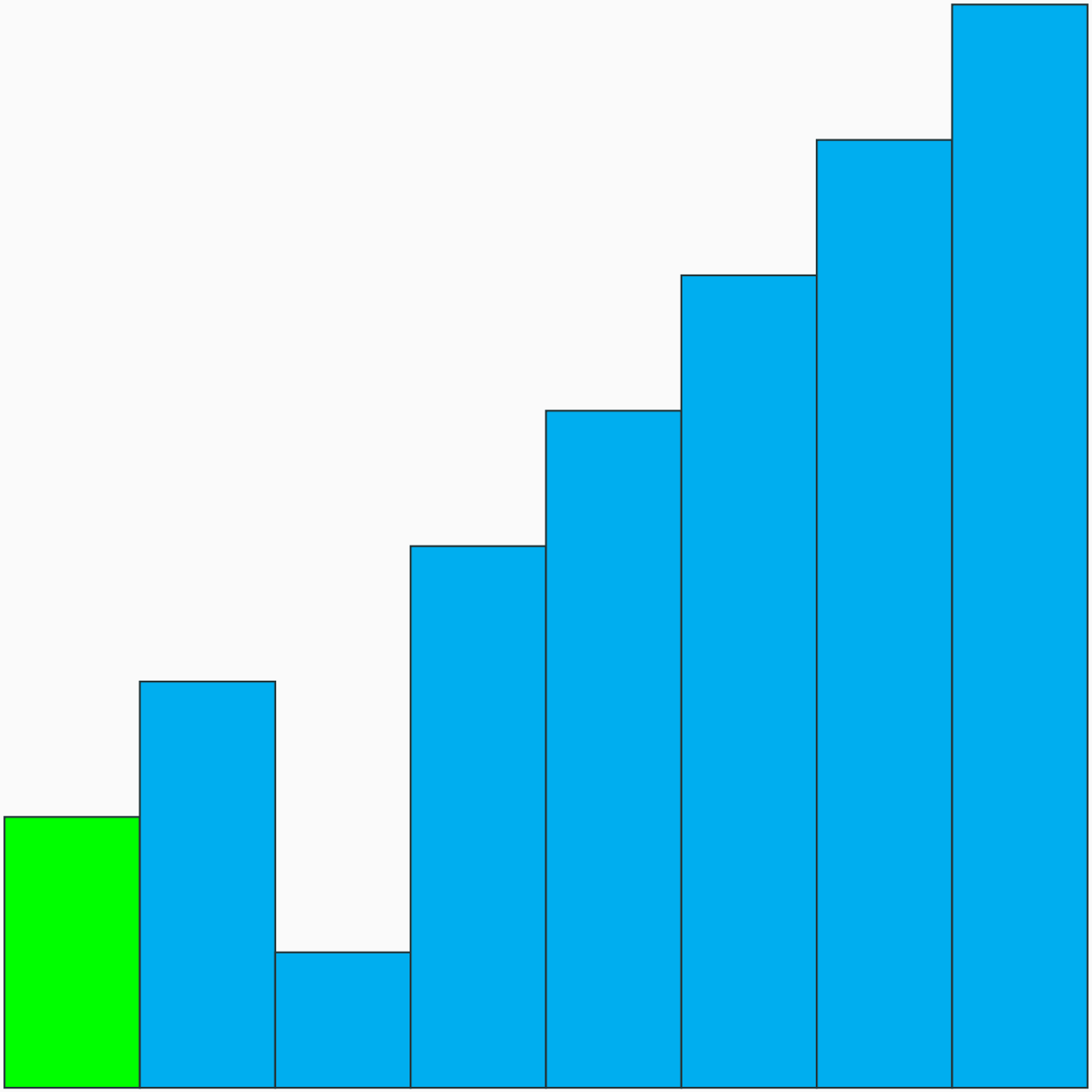
Exemple



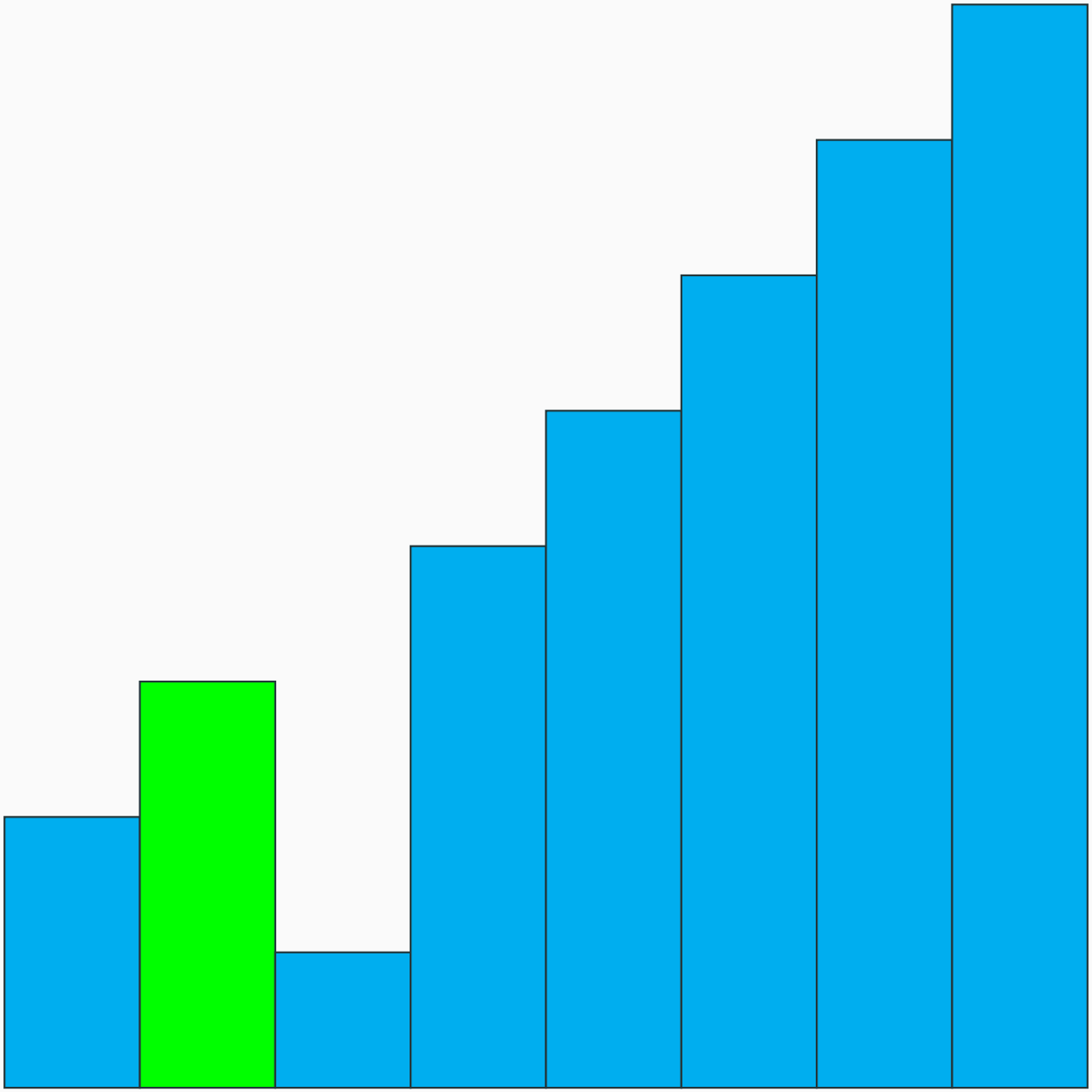
Exemple



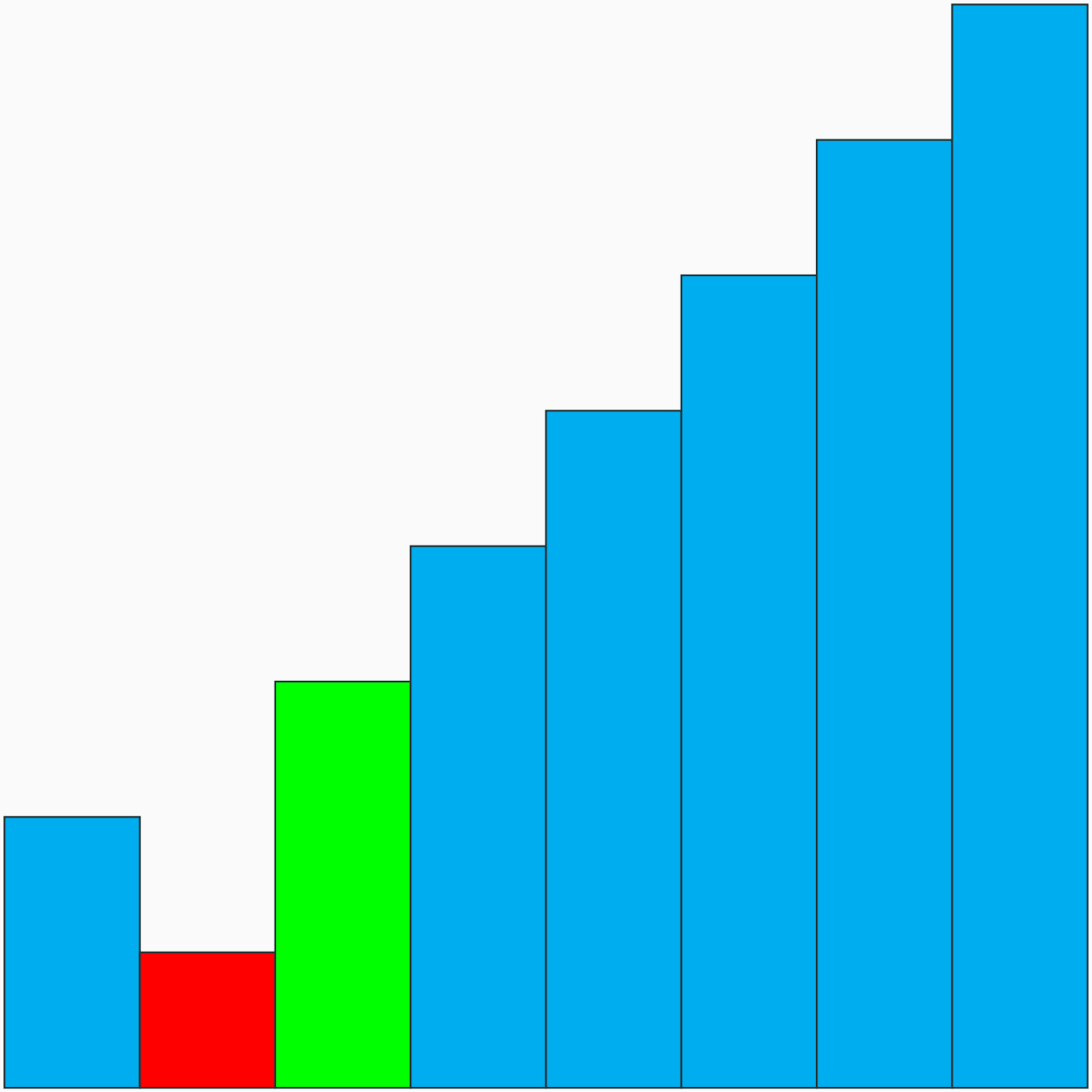
Exemple



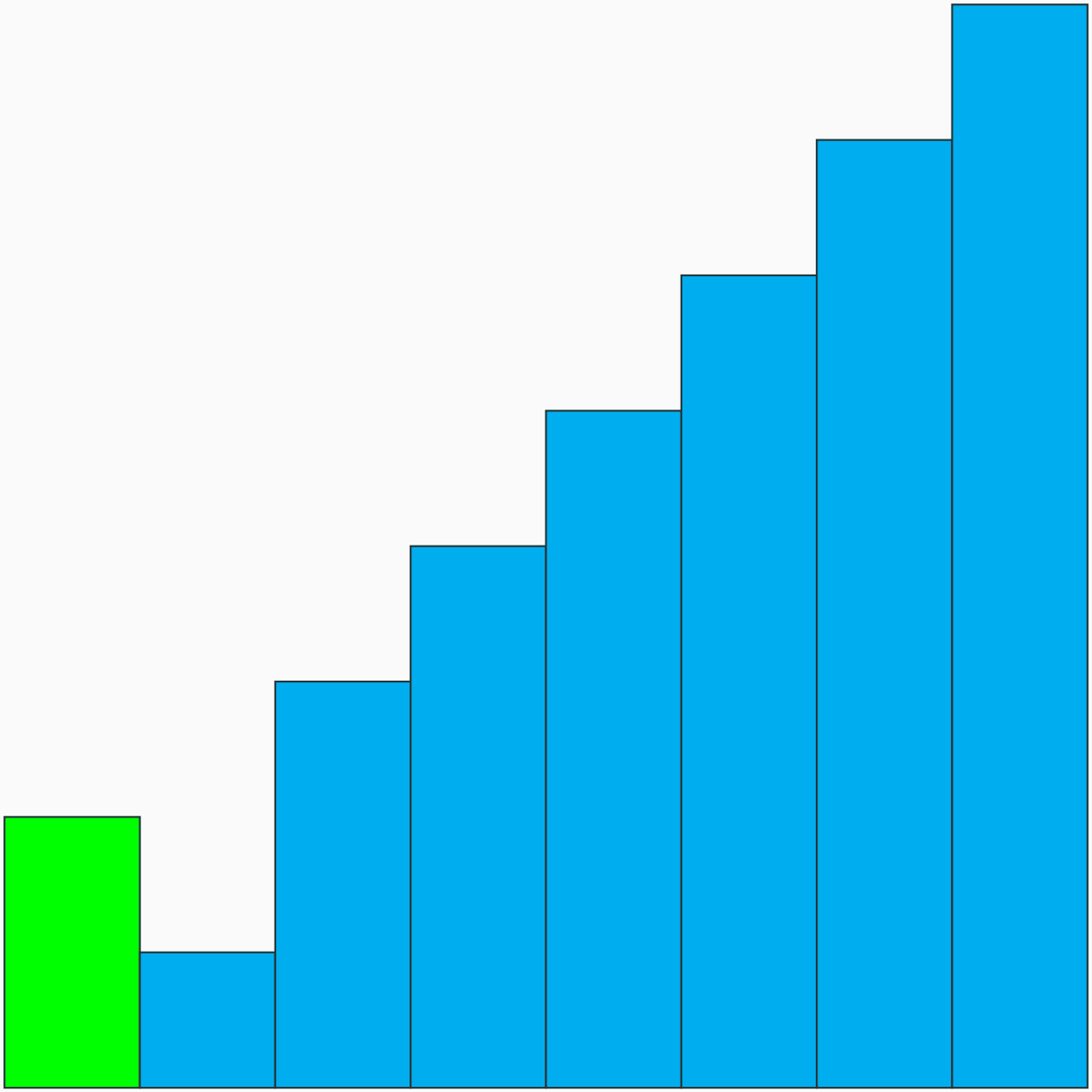
Exemple



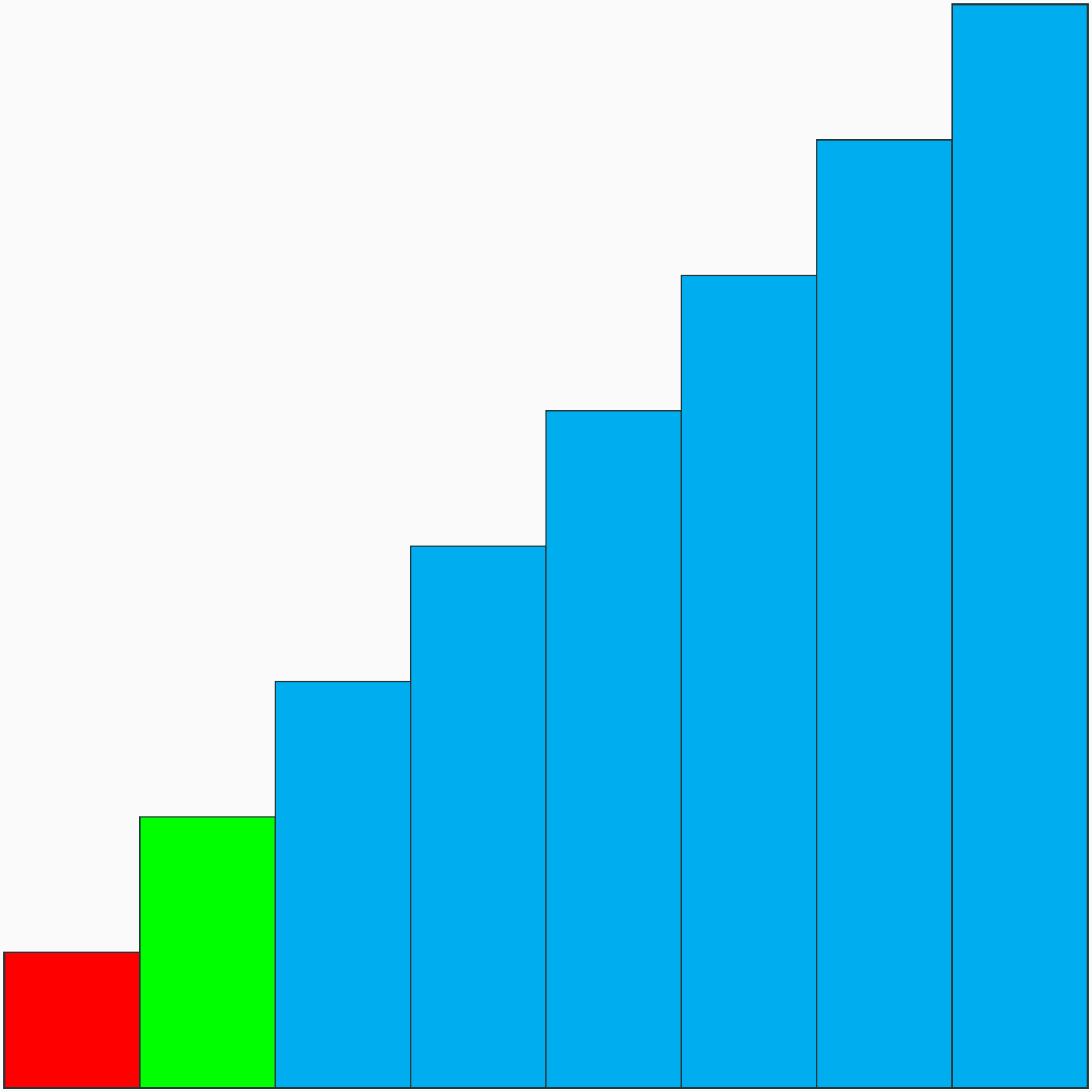
Exemple



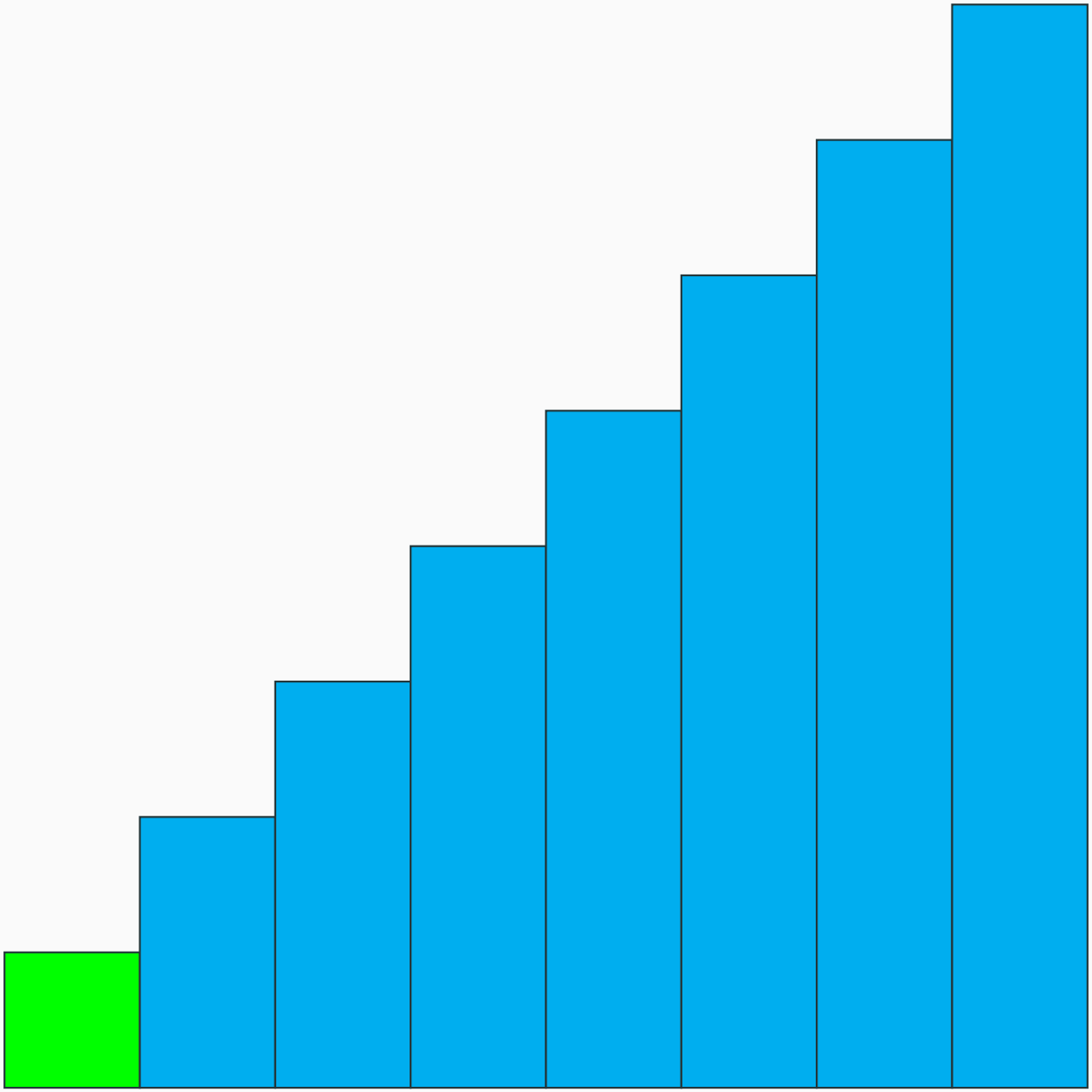
Exemple



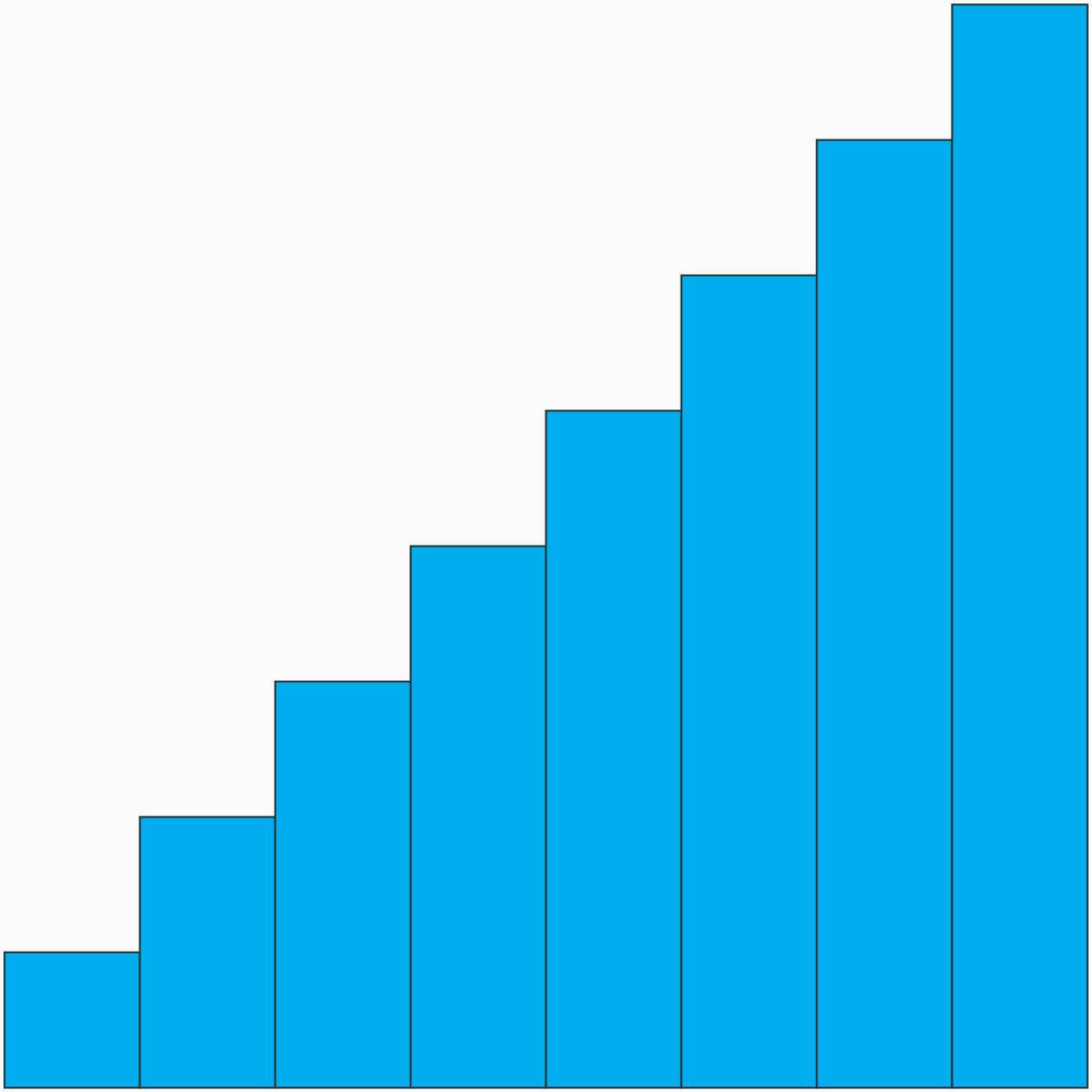
Exemple



Exemple



Exemple



Terminaison du tri à bulles

Terminaison

L'algorithme du tri à bulles termine.

Preuve

p est strictement décroissant à chaque passage dans la boucle `while`. De plus, si $p \leq 1$, la boucle `for` ne fait rien, car l'ensemble des entiers à parcourir est vide. Ainsi, `pasfini` reste à `false` et la boucle `while` termine.

Correction du tri à bulles

Correction

L'algorithme du tri à bulles est correct : à la fin de l'algorithme, le tableau est triée.

Invariant de la boucle `while`

Inv : Les éléments `t.(p),...,t.(n-1)` sont triés, et sont plus grands que les autres éléments de le tableau. De plus, `pasfini` $=$ `false` ou les éléments `t.(0),...,t.(p-1)` sont triés.

Invariant de la boucle `for`

$\mathrm{Inv}'(i)$: `t.(i)` est plus grand que tous les éléments à sa gauche.

Tri à bulles

Complexité dans le meilleur cas

La complexité du tri à bulles dans le **meilleur** des cas est **linéaire** : en effet, si le tableau est déjà trié on effectue seulement un parcours, soit $n - 1$ comparaisons et aucune affectation.

Complexité dans le pire cas

Dans le **pire** des cas, la complexité est **quadratique** : c'est le cas si le tableau est initialement trié dans l'ordre décroissant.

Dans ce cas, on effectue n parcours de la boucle `while`, et à chaque fois $p - 1$ échanges et comparaisons. Le nombre total d'échanges et de comparaisons est donc de

$$\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2}$$

Implémentation d'une pile et d'une file dans un tableau

Implémentation d'une file et d'une pile dans un tableau

Présentation

Dans cette partie, on décrit comment implémenter une **pile** ou une **file** à l'aide d'un **tableau**.

Remarque : structures de tailles bornées ou non

De par la finitude de la mémoire qu'utilise une structure de pile ou de file, le nombre d'éléments qu'on peut mettre dans ce type de structure est nécessairement **borné**.

Toutefois, cette borne est en général conséquente, si bien que l'on peut considérer que la **capacité** (nombre d'éléments que l'on peut stocker) est **infinie** : on parle de pile ou de file **non bornée**.

Implémentation d'une file et d'une pile dans un tableau

Remarque : structures de tailles bornées ou non

On va implémenter ici des structures dans lesquelles la capacité est bornée une fois pour toute à la création de l'objet : la pile ou la file est **bornée**.

La fonction de création prendra donc en **paramètre** la capacité.

L'explication est simple : on va utiliser des **tableaux**, dont la taille est elle-même fixée une fois pour toute : la taille du tableau sera égale à la capacité choisie.

Piles

Rappel (Piles)

On rappelle qu'une **pile** suit le principe "**LIFO**" : dernier arrivé, premier sorti. Les opérations à écrire pour implémenter une pile sont les suivantes :

- **création** d'une pile **vide** ;
- **test d'égalité** au vide ;
- **accès au sommet** d'une pile non vide ;
- **retrait** de l'élément au sommet d'une pile non vide ;
- **ajout** d'un élément au sommet de la pile (non pleine ici).

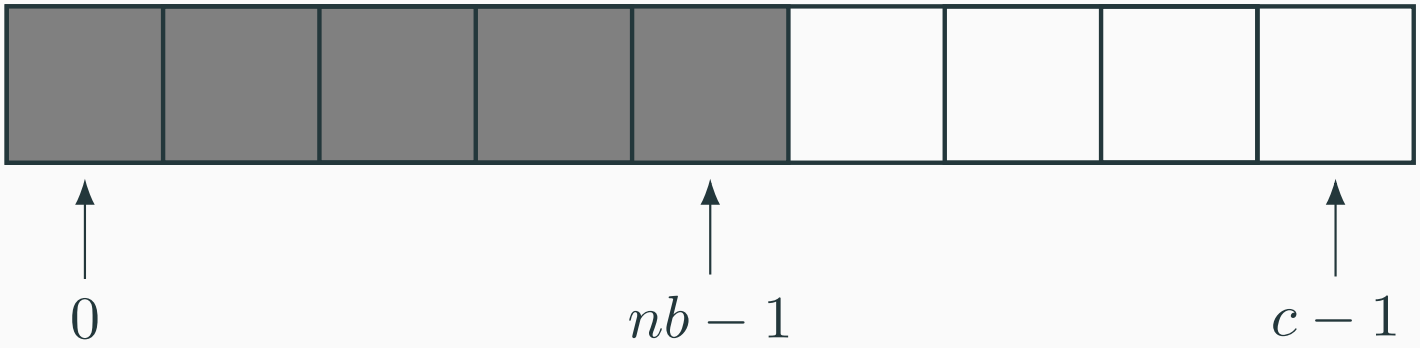
Représentation dans un tableau

Implémentation

La représentation choisie est celle d'un **type enregistrement** dans lequel sont stockés :

- la capacité `capacite` : nombre maximal d'éléments que l'on peut stocker dans la pile ;
- le nombre d'éléments `nb` présents dans la pile ;
- un tableau `contenu` de taille `capacite`, dont les `nb` premiers éléments sont les éléments présents dans la pile, les autres éléments du tableaux ne sont pas des éléments de la pile. Le sommet est l'élément d'indice `nb-1`.

Représentation dans un tableau



Figure

Sur la figure ci-dessus, le fond de la pile est à l'indice 0, et le sommet est à l'indice $nb-1$.

Les éléments grisés sont ceux de la pile, les éléments blancs sont quelconques.

L'**ajout** d'un élément dans la pile se fait en position nb (puis il faut incrémenter nb).

Pour **retirer** le sommet de la pile, il suffit de décrémenter nb.

Implémentation concrète

Polymorphisme

On décide de donner un type pile **polymorphe**, le type est fixé à la création et la fonction de création prend donc deux paramètres :

- la capacité ;
- et un élément permettant de créer le tableau (néanmoins la pile est vide au départ).

Remarque

Le champ `capacite` est superflu, car la capacité d'une telle pile peut s'obtenir comme la longueur du tableau `contenu`.

Néanmoins, l'implémentation est plus claire avec ce champ supplémentaire.

Implémentation concrète

```
1  type 'a pile = {capacite: int ; mutable nb: int ; contenu: 'a array} ;;
2
3  let creer_pile c x = {capacite = c ; nb = 0 ; contenu = Array.make c x} ;;
4
5  let pile_vide p = p.nb = 0 ;;
6
7  let pile_pleine p = p.nb = p.capacite
8
9  let empiler p x = match pile_pleine p with
10   | true -> failwith "pile pleine"
11   | false -> p.contenu.(p.nb) <- x ; p.nb <- p.nb + 1
12 ;;
13
14 let sommet p = match pile_vide p with
15   | true -> failwith "pile vide"
16   | false -> p.contenu.(p.nb - 1)
17 ;;
18
19 let depiler p = match pile_vide p with
20   | true -> failwith "pile vide"
21   | false -> p.nb <- p.nb - 1 ; p.contenu.(p.nb)
22 ;;
```

Exemple d'utilisation

Exemple d'utilisation

```
# let p=creer_pile 5 0 ;;
val p : int pile = {capacite = 5; nb = 0; contenu = [|0; 0; 0; 0; 0|]}
# empiler p 1 ;;
- : unit = ()
# empiler p 2 ;;
- : unit = ()
# empiler p 3 ;;
- : unit = ()
# depiler p ;;
- : int = 3
# p ;;
- : int pile = {capacite = 5; nb = 2; contenu = [|1; 2; 3; 0; 0|]}
```

Remarque

Attention, la pile **p** ne possède que deux éléments à la fin de cet exemple : les deux premiers du tableau **contenu**.

Les trois suivants ne font pas partie de la pile.

Complexité des opérations

Complexité

La création est en $O(c)$, où c est la capacité de la pile.

Toutes les autres opérations se font en temps constant ($O(1)$).

Files

Rappel (Files)

On rappelle qu'une **file** suit le principe "**FIFO**" : premier arrivé, premier sorti. Les opérations à écrire pour implémenter une file sont les suivantes :

- **création** d'une file **vide** ;
- **test d'égalité** au vide ;
- **retrait** de l'élément en tête de file non vide ;
- **ajout** d'un élément en queue d'une file (non pleine ici).

Représentation à l'aide d'un tableau

Implémentation

On va donner une réalisation d'une **file bornée à** partir d'un **tableau**, semblable à celle d'une pile.

C'est un peu plus complexe pour une file, parce qu'on ne réalise pas l'ajout et la suppression d'éléments au même endroit.

L'astuce est de considérer le tableau comme "**circulaire**".

On utilise donc deux indices supplémentaires (mutables), qui indiquent les positions de la **tête** de file (le premier à avoir été inséré dans la file, donc le prochain à sortir), et de la **queue** (position du prochain élément qui va être inséré).

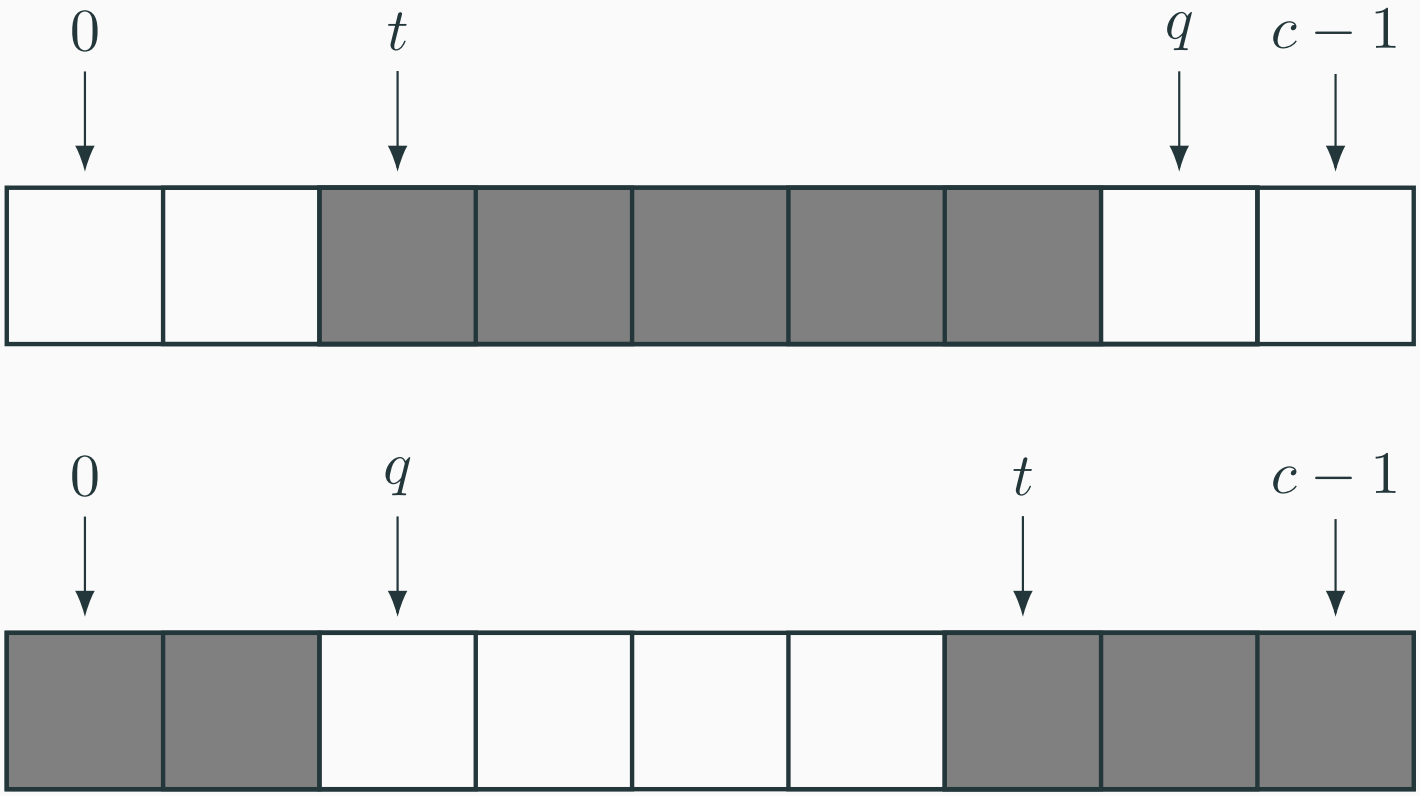
Représentation à l'aide d'un tableau

Implémentation

La représentation choisie est donc celle d'un **type enregistrement** dans lequel sont stockés :

- la capacité `capacite` ;
- le nombre `nb` d'éléments dans la file ;
- les indices `tete` et `queue` ;
- un tableau `contenu` dont les éléments entre les indices `tete` (inclus) et `queue` (exclu) sont présents dans la file, et les autres éléments du tableau ne sont pas présents dans la file.

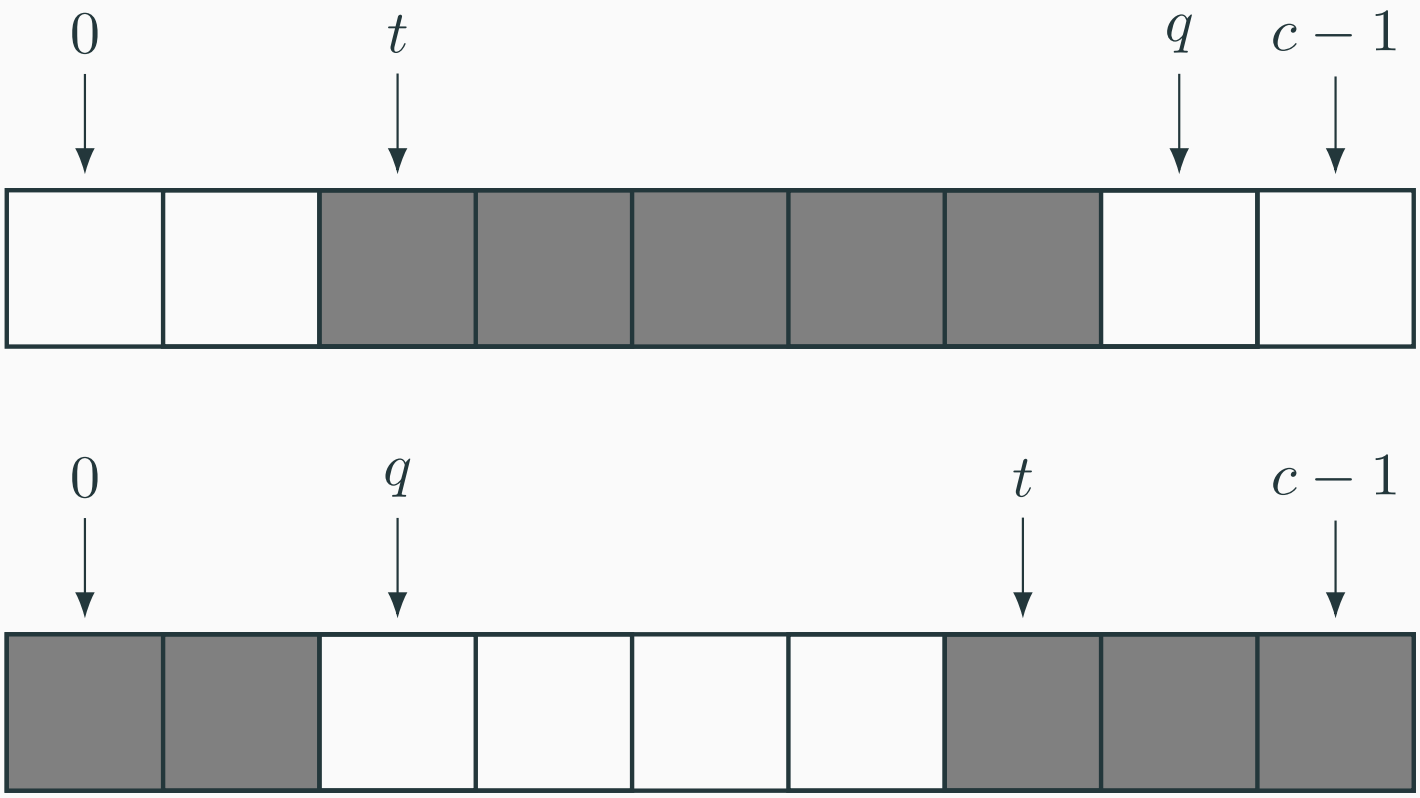
Représentation dans un tableau



Figure

Sur les deux figures ci-dessus, il y a 5 éléments dans la file : les éléments grisés, entre t (inclus) et q (exclu).

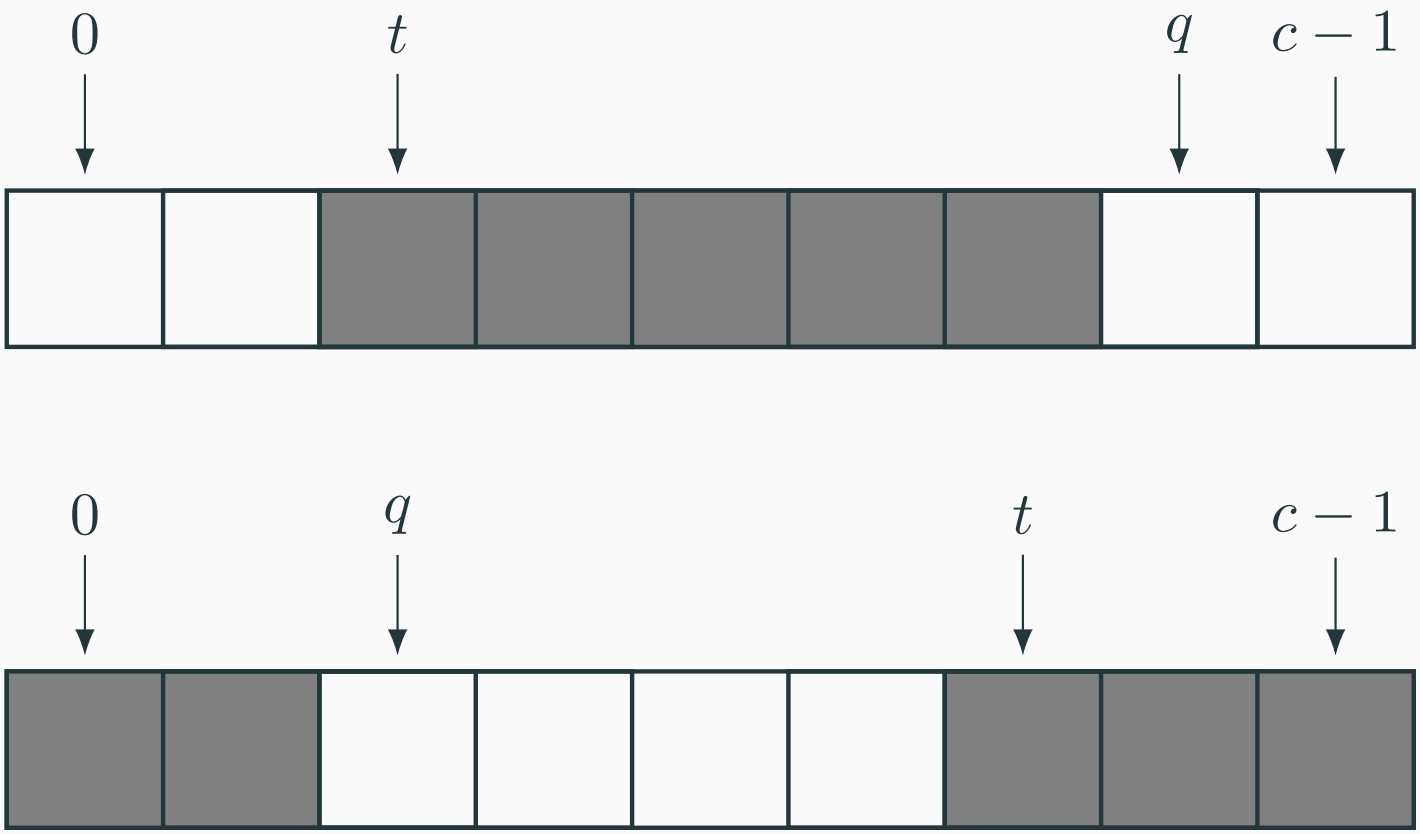
Représentation dans un tableau



Figure

L'**ajout** d'un élément se fait en position q (puis on augmente q de 1 modulo c).

Représentation dans un tableau



Figure

Pour faire **défiler** la file, il suffit d'augmenter t de 1 modulo c.

Représentation dans un tableau

Remarques

- Là encore, le champ **`capacite`** est superflu.
- Il en va de même pour l'un des deux champs **`tete`** ou **`queue`** car la relation suivante est toujours vérifiée :

$$\texttt{queue} - \texttt{tete} \equiv \texttt{nb} \mod \texttt{capacite}$$

- En revanche, le champ **`nb`** ne peut s'obtenir à partir de **`tete`** et **`queue`**, car lorsque $\texttt{tete} = \texttt{queue}$, c'est **`nb`** qui permet de faire la distinction entre une file **vide** et une file **pleine**.

Implémentation concrète

```
1  type 'a file = {capacite: int; mutable nb: int; mutable tete: int;
2                  mutable queue: int; contenu: 'a array} ;;
3
4  let creer_file c x = {capacite = c ; nb = 0 ; tete = 0 ; queue = 0 ; contenu=Array.make c x} ;;
5
6  let file_vide f = f.nb = 0 ;;
7
8  let file_pleine f = f.nb = f.capacite ;;
9
10 let enfiler f x = match file_pleine f with
11   | true -> failwith "file pleine"
12   | false -> f.contenu.(f.queue) <- x ;
13              f.queue <- (f.queue + 1) mod f.capacite ;
14              f.nb <- f.nb + 1
15 ;;
16
17 let defiler f = match file_vide f with
18   | true -> failwith "file vide"
19   | false -> let x = f.contenu.(f.tete) in
20                f.tete <- (f.tete + 1) mod f.capacite ;
21                f.nb <- f.nb - 1 ;
22                x
23 ;;
```

Exemple d'utilisation

Exemple d'utilisation

```
# let f=creer_file 5 0 ;;
val f : int file =
{capacite = 5; nb = 0; tete = 0; queue = 0; contenu = [|0; 0; 0; 0; 0|]}
# for i=1 to 3 do enfiler f i done ;;
- : unit = ()
# defiler f ;;
- : int = 1
# f;;
- : int file =
{capacite = 5; nb = 2; tete = 1; queue = 3; contenu = [|1; 2; 3; 0; 0|]}
```

Remarque

Les éléments présents dans la file (2 et 3), sont bien entre l'indice de tête inclus et l'indice de queue exclu.

Complexité des opérations

Complexité

Là encore, la création est en $O(c)$, où c est la capacité.

Et toutes les autres opérations se font en temps constant ($O(1)$).