Algorithmes “Diviser pour régner”

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Introduction

Paradigme “Diviser pour régner”

Diviser pour régner

Dans le chapitre précédent, nous avons vu un exemple d'algorithme “diviser pour régner” : le **tri fusion**.

Nous allons maintenant voir d'autres exemples de résolution de problèmes via des algorithmes de ce type, dont on rappelle le fonctionnement :

- **découper** l'instance à résoudre en instances **plus petites** ;
- résoudre **récursivement** le problème sur les instances plus petites ;
- **reconstituer** la solution du problème sur l'instance **initiale** à partir des solutions précédemment obtenues.

Tri fusion

Analyse du tri fusion

Principe du tri fusion

Pour trier une liste de taille $n > 1$, on avait suivi le principe suivant :

- découper la liste à trier en deux listes de tailles égales (à un près) via une fonction **`fission`** ;
- appeler récursivement le tri sur les deux portions ;
- fusionner ces deux listes triées via une fonction **`fusion`**, qui renvoyait alors une liste triée.

Analyse des fonctions auxiliaires

```
1  let rec fission l = match l with
2    | [] | [_] -> l, []
3    | x::y::q -> let a,b=fission q in x::a, y::b
4  ;;
5
6  let rec fusion l1 l2 = match l1, l2 with
7    | [],_ -> l2
8    | _, [] -> l1
9    | x::q1, y::_ when x<=y -> x::(fusion q1 l2)
10   | _,x::q2 -> x::(fusion l1 q2)
11 ;;
```

Terminaison

La fonction **`fission`** termine sur les listes de taille 0 ou 1, et l'appel récursif se fait sur une liste de taille $n-2$. Donc elle termine.

Analyse des fonctions auxiliaires

```
1  let rec fission l = match l with
2    | [] | [_] -> l, []
3    | x::y::q -> let a,b=fission q in x::a, y::b
4  ;;
5
6  let rec fusion l1 l2 = match l1, l2 with
7    | [],_ -> l2
8    | _, [] -> l1
9    | x::q1, y::_ when x<=y -> x::(fusion q1 l2)
10   | _,x::q2 -> x::(fusion l1 q2)
11 ;;
```

Terminaison

De même, en notant n_1 et n_2 les taille des listes en paramètre de la fonction **`fusion`**, celle-ci termine si $n_1 = 0$ ou $n_2 = 0$, et sinon l'appel récursif se fait sur deux listes de tailles $(n_1', n_2') = (n_1 - 1, n_2)$ ou $(n_1, n_2 - 1)$.

Dans tous les cas, $(n_1', n_2') < (n-1, n_2)$ pour l'**ordre produit**.

Donc la fonction termine.

Analyse des fonctions auxiliaires

```
1  let rec fission l = match l with
2    | [] | [_] -> l, []
3    | x::y::q -> let a,b=fission q in x::a, y::b
4  ;;
5
6  let rec fusion l1 l2 = match l1, l2 with
7    | [],_ -> l2
8    | _, [] -> l1
9    | x::q1, y::_ when x<=y -> x::(fusion q1 l2)
10   | _,x::q2 -> x::(fusion l1 q2)
11 ;;
```

Correction

On montre facilement par induction sur les quantités évoquées précédemment les propriétés suivantes :

"**`fission l`** renvoie un couple de listes **`(l1,l2)`** de tailles égales à un près, dont les éléments sont ceux de **`l`**."

Analyse des fonctions auxiliaires

```
1  let rec fission l = match l with
2    | [] | [_] -> l, []
3    | x::y::q -> let a,b=fission q in x::a, y::b
4  ;;
5
6  let rec fusion l1 l2 = match l1, l2 with
7    | [],_ -> l2
8    | _, [] -> l1
9    | x::q1, y::_ when x<=y -> x::(fusion q1 l2)
10   | _,x::q2 -> x::(fusion l1 q2)
11 ;;
```

Correction

On montre facilement par induction sur les quantités évoquées précédemment les propriétés suivantes :

"**`fusion l1 l2`** renvoie une liste triée dont les éléments sont ceux de **`l1`** et **`l2`** si ces listes sont triées."

Analyse des fonctions auxiliaires

```
1  let rec fission l = match l with
2    | [] | [_] -> l, []
3    | x::y::q -> let a,b=fission q in x::a, y::b
4  ;;
5
6  let rec fusion l1 l2 = match l1, l2 with
7    | [],_ -> l2
8    | _, [] -> l1
9    | x::q1, y::_ when x<=y -> x::(fusion q1 l2)
10   | _,x::q2 -> x::(fusion l1 q2)
11 ;;
```

Complexité

La complexité de la fonction **fission** vérifie l'équation $C(n) = C(n-2) + O(1)$, dont la solution est $C(n) = O(n)$. De même, avec $n = n_1 + n_2$, la complexité de **fusion** est solution de $C(n) = C(n-1) + O(1)$, d'où également une complexité en $O(n)$ pour cette fonction.

Analyse de la fonction de tri

```
1  let rec tri_fusion l = match l with
2    | [] | [_] -> l
3    | _ -> let a,b=fission l in fusion (tri_fusion a) (tri_fusion b)
4  ;;
```

Terminaison et correction

La fonction **`tri_fusion`** est correcte sur les listes de taille 0 ou 1, et pour $n > 1$ elle se rappelle récursivement sur des listes de tailles $\lfloor \frac{n}{2} \rfloor < n$ et $\lceil \frac{n}{2} \rceil < n$. Donc elle termine.

Elle est correcte par récurrence sur n et correction des fonctions **`fission`** et **`fusion`**.

Analyse de la fonction de tri

```
1  let rec tri_fusion l = match l with
2    | [] | [_] -> l
3    | _ -> let a,b=fission l in fusion (tri_fusion a) (tri_fusion b)
4  ;;
```

Complexité

La complexité de la fonction **`tri_fusion`** vérifie l'équation :

$$C(n) = 2 \cdot C(n/2) + O(n)$$

D'après le **théorème maître** du chapitre précédent, on a donc $C(n) = O(n \log n)$.

Tri rapide

Tri rapide (QuickSort)

QuickSort

Le **tri rapide** a été inventé par Hoare en 1961.

Bien implémenté, c'est probablement le tri le plus efficace en pratique.

Bien que sa complexité soit quadratique, il possède une très bonne complexité en moyenne.

Il a de plus l'avantage de s'effectuer en place, contrairement au tri fusion.

Tri rapide (QuickSort)

Idée

- Contrairement au tri fusion, on ne coupe pas arbitrairement en deux.
- On choisit un élément p qu'on appelle le **pivot**, et on sépare les éléments en deux parties :
  - d'un côté, les éléments $\geq p$;
  - de l'autre, les éléments $< p$.
- Pour choisir le pivot, il y a plusieurs possibilités.
  $\hookrightarrow$ On peut par exemple choisir le premier élément du tableau.
- Ensuite, on trie récursivement les deux sous-ensembles, et il ne reste plus qu'à les concaténer.

Fonction de partition

Tri en place

On pourrait écrire une fonction `partition l x` partitionnant les éléments de `l` en deux listes formées des éléments plus petits et plus grands que `x`.

On en déduirait alors un tri récursif similaire au tri fusion, qui renverrait une nouvelle liste.

Mais l'avantage du tri rapide sur le tri fusion est qu'il peut s'écrire facilement **en place**, sur les tableaux.

Partition d'une portion du tableau

La fonction à écrire prend en entrée un tableau `t`, et deux indices g et d délimitant le sous-tableau de `t` à partitionner, qui est `t[g:d]`.

On suppose que $g < d - 1$, de sorte qu'il y ait au moins deux éléments dans la portion. L'idée est alors la suivante :

- On choisit (arbitrairement) le pivot comme étant le premier élément de la portion, à savoir `t.(g)`.
- On parcourt le reste de la portion (de l'indice $g + 1$ inclus à l'indice d exclu) en gardant une séparation de la portion déjà parcourue en deux morceaux :
  - celui des éléments strictement inférieurs au pivot,
  - et celui des morceaux supérieurs ou égaux.

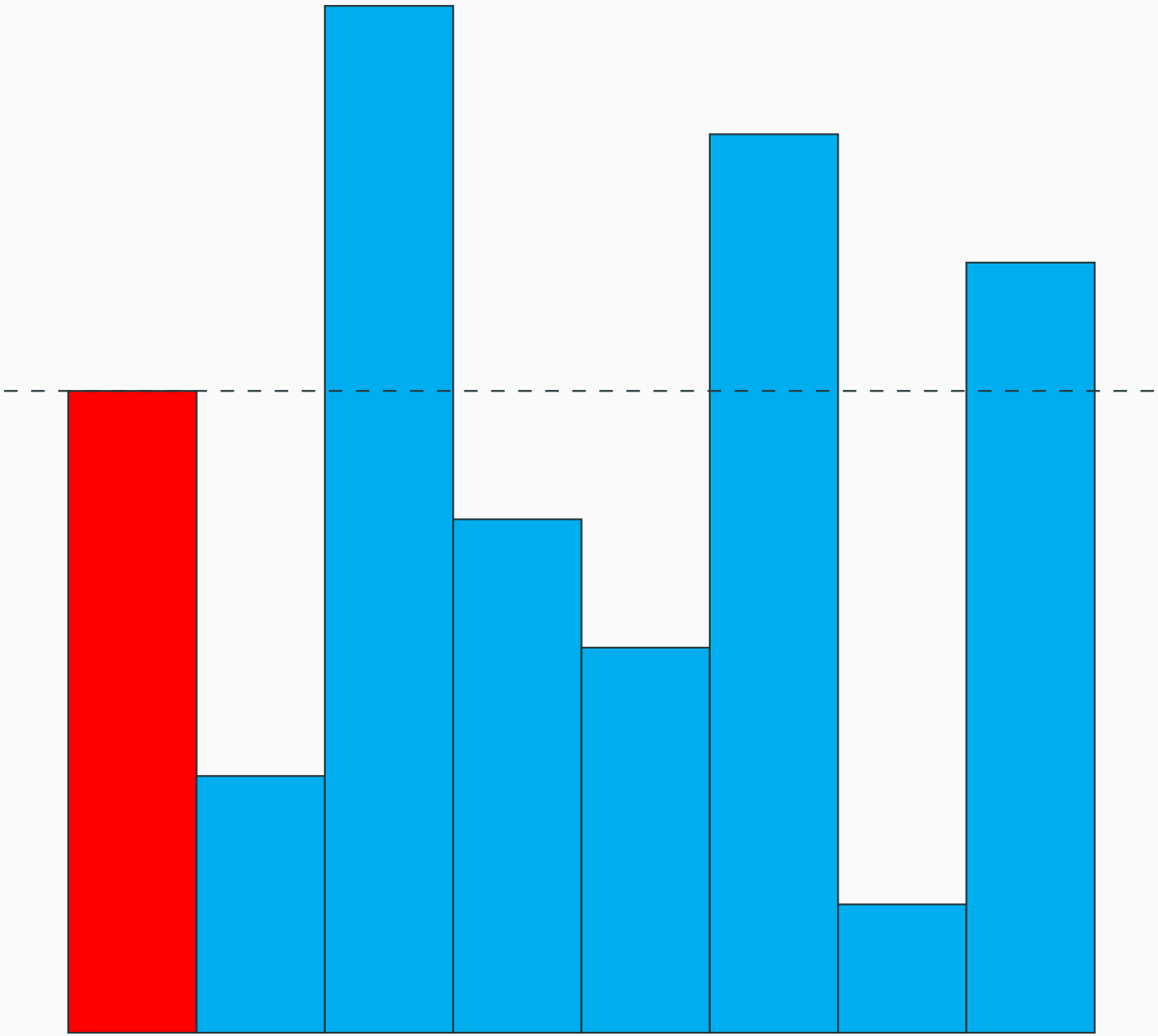
Partition d'une portion du tableau

- Une fois la portion parcourue entièrement, il suffit d'amener le pivot à la jonction des deux morceaux, pour avoir partitionné la portion en trois zones :
  - les éléments strictement inférieurs au pivot,
  - le pivot,
  - et les éléments supérieurs.
- Dans l'optique de trier le tableau, il faudra faire des appels récursifs sur les deux zones à gauche et à droite du pivot : on renvoie donc l'indice où se trouve le pivot à la fin de la partition.

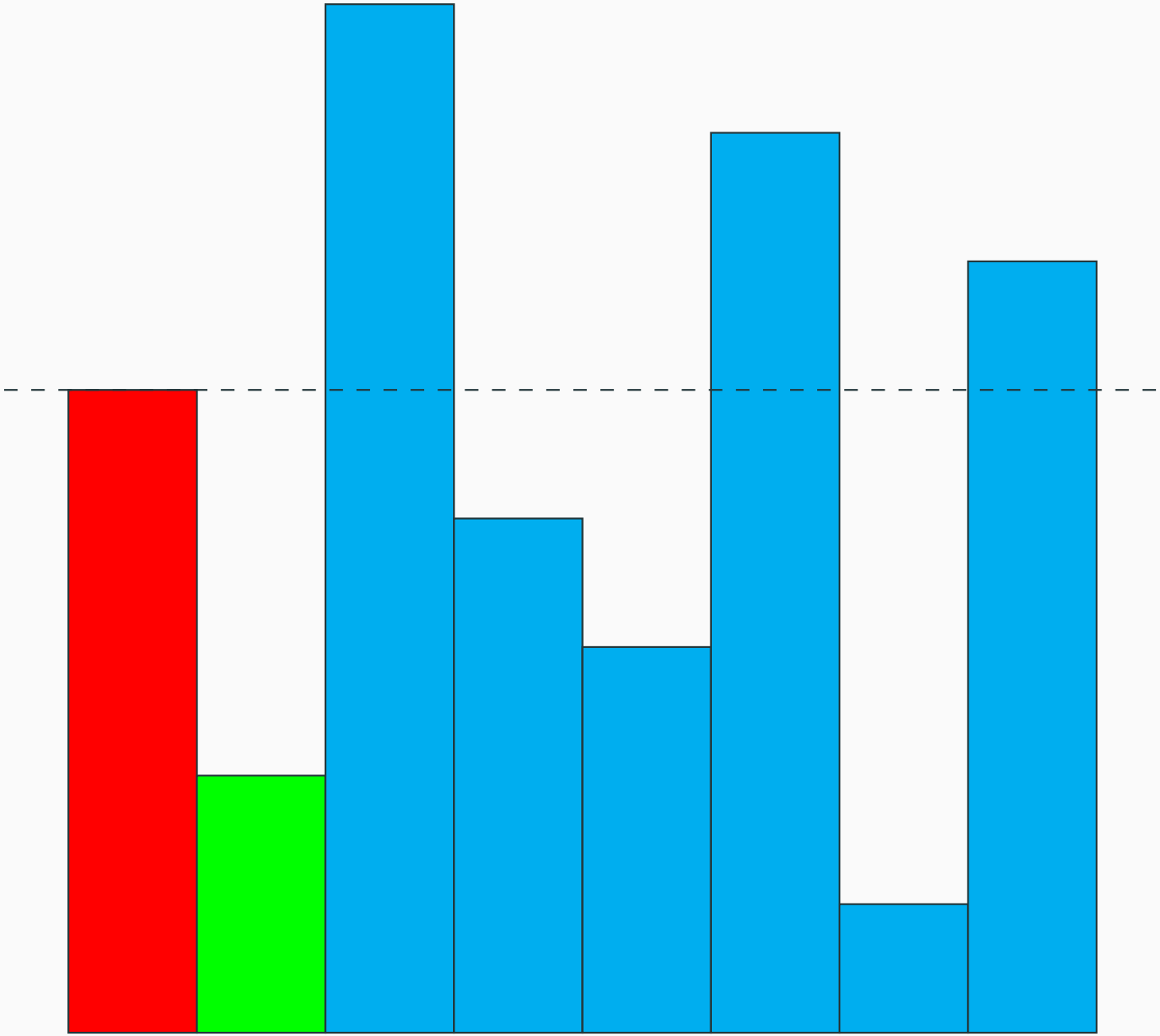
Ocaml

```
1  let partition t g d =
2    let p = t.(g) in (* valeur du pivot *)
3    let ip = ref g in (* position du pivot *)
4    for i=g+1 to d-1 do
5    (* Inv(i): les éléments de t[g+1:ip+1] sont strictement inférieurs au pivot t.(0),
6       les éléments de t[ip+1:i] sont supérieurs, avec i>=ip+1 *)
7      if t.(i)<p then
8        begin
9          incr ip ;
10         if i > !ip then (* on échange les éléments i et ip *)
11           begin
12             let x = t.(i) in
13             t.(i) <- t.(!ip) ;
14             t.(!ip) <- x
15           end
16       end
17   done ;
18   if !ip > g then (* on met le pivot à sa place finale *)
19     begin
20       let x = t.(g) in
21       t.(g) <- t.(!ip) ;
22       t.(!ip) <- x
23     end ;
24   !ip
25 ;;
```

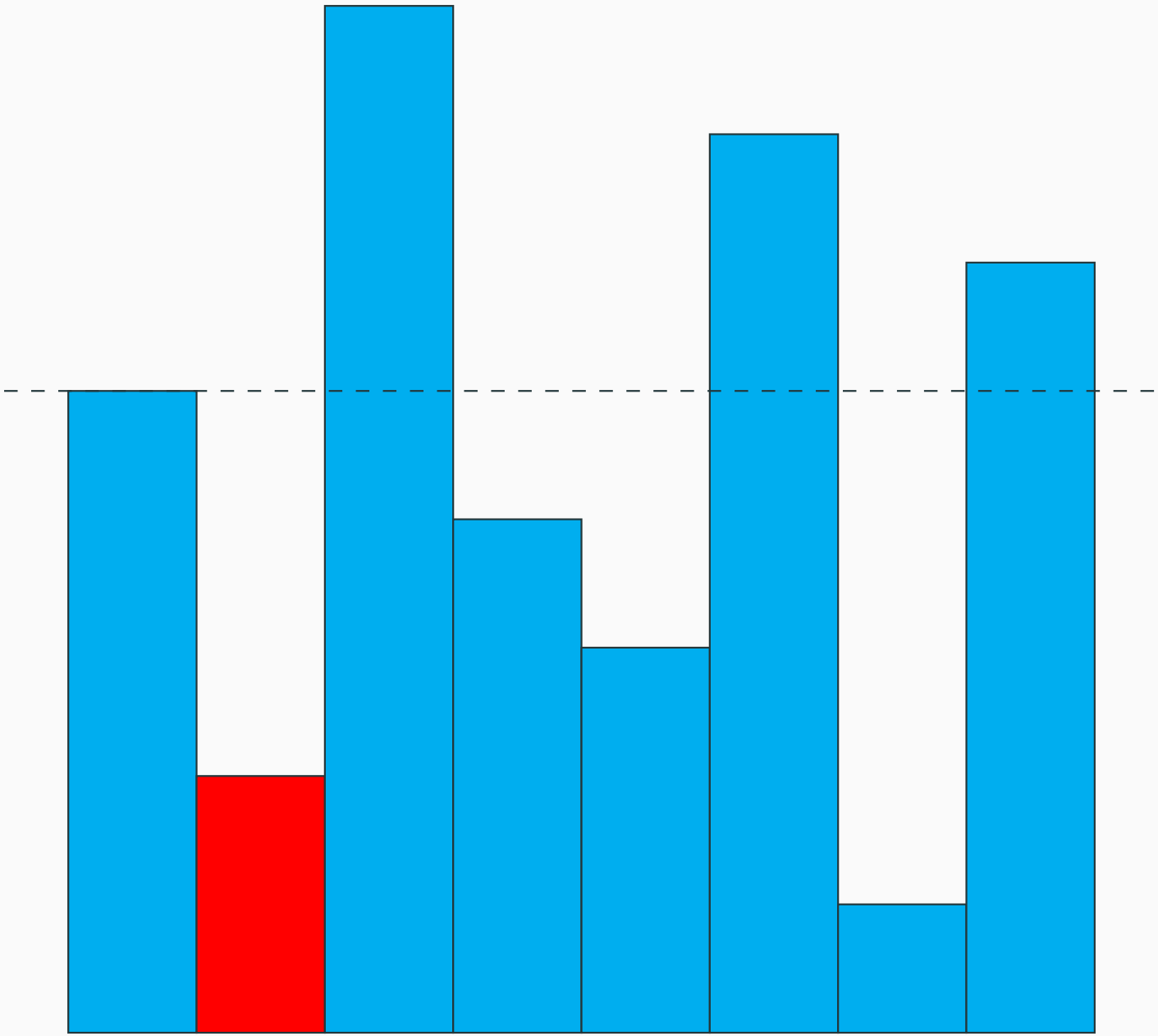
Exemple



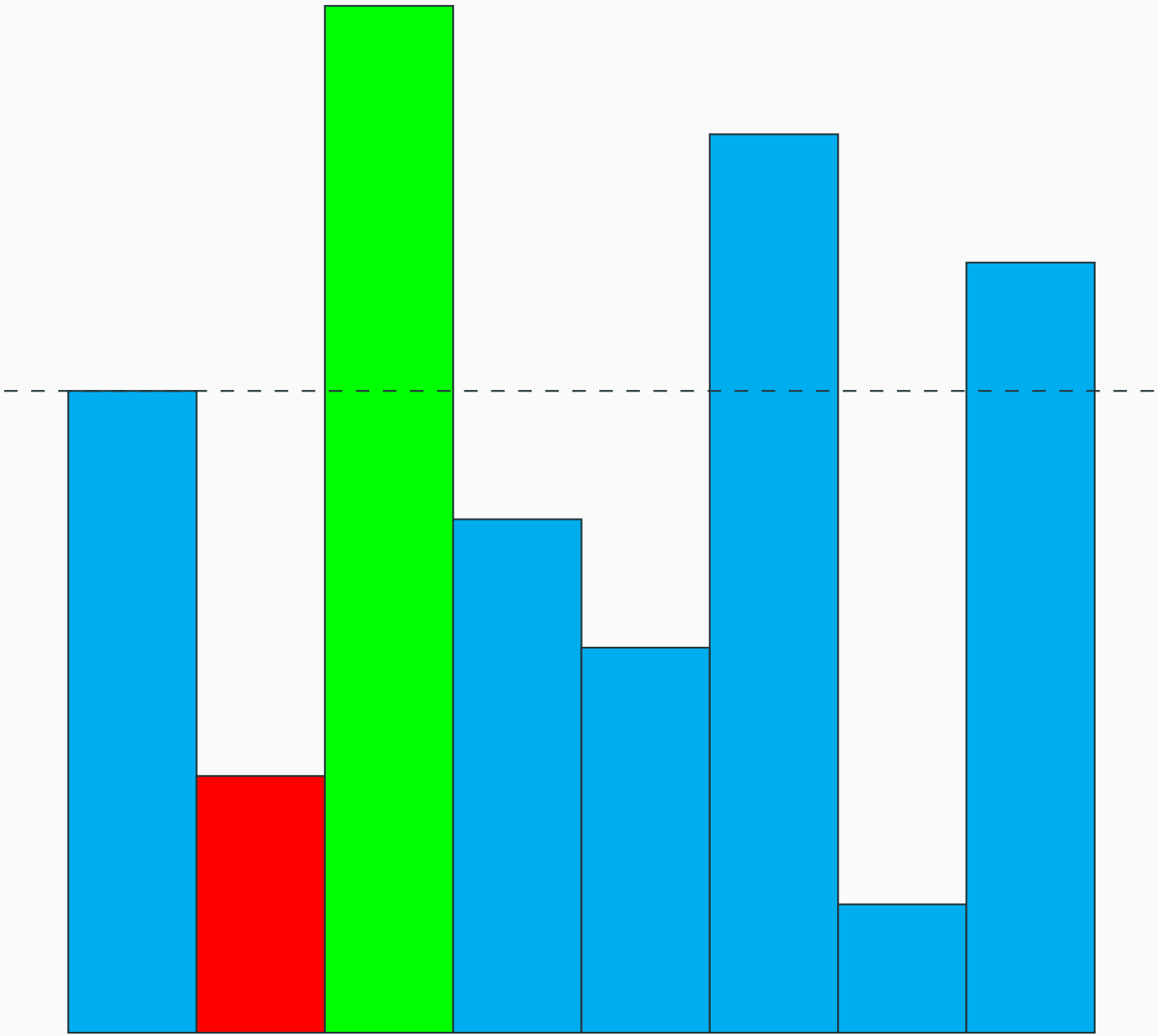
Exemple



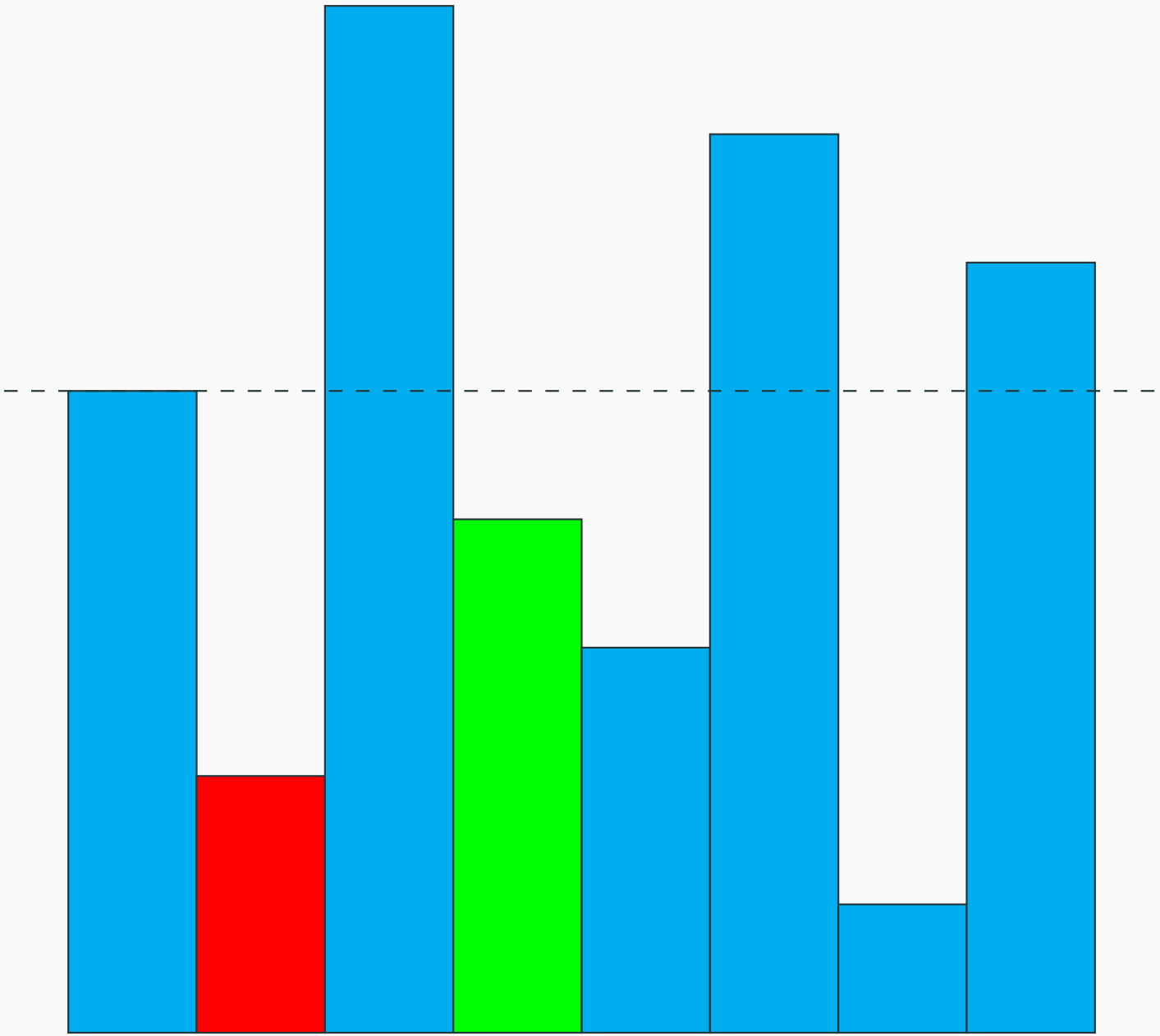
Exemple



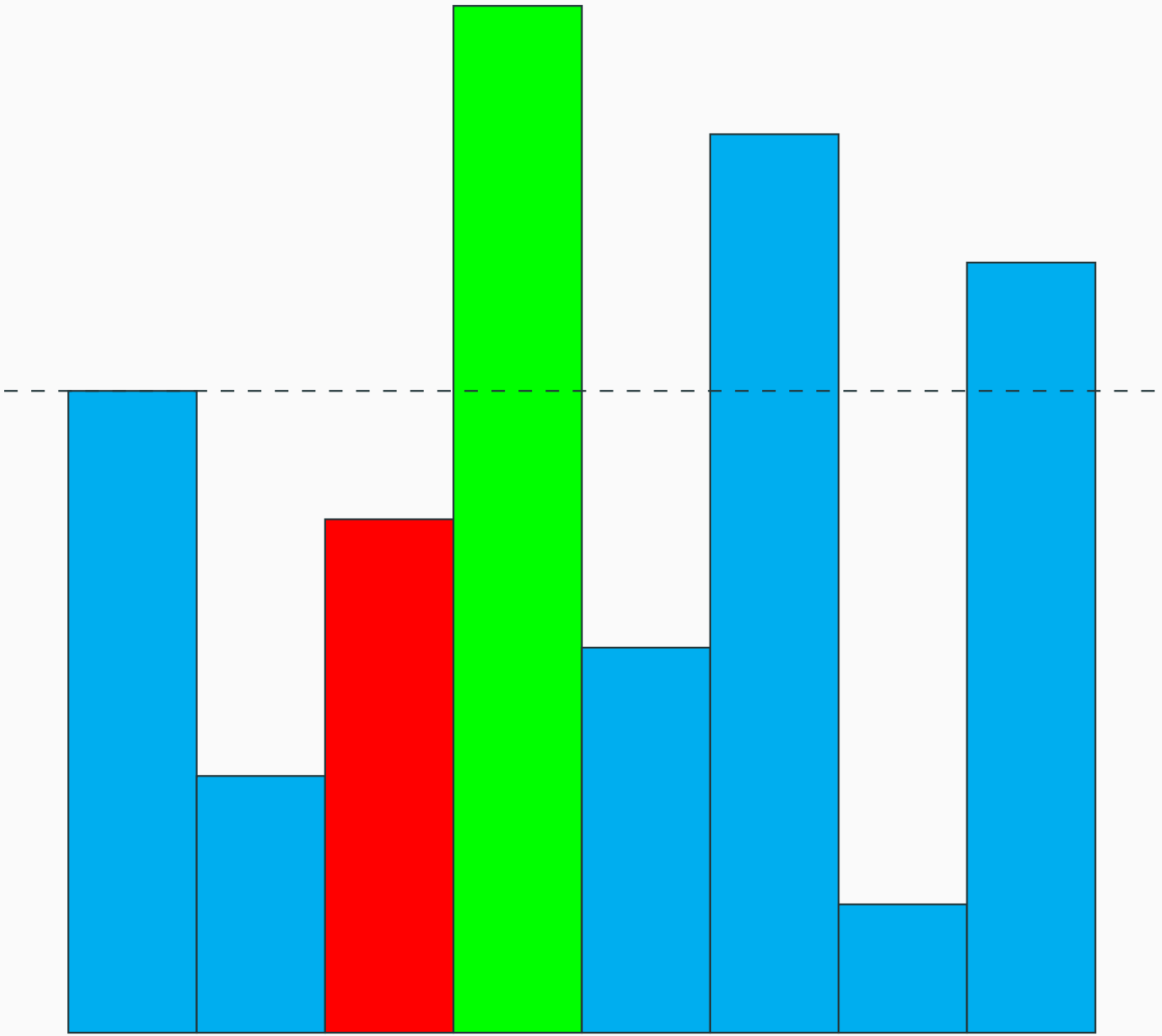
Exemple



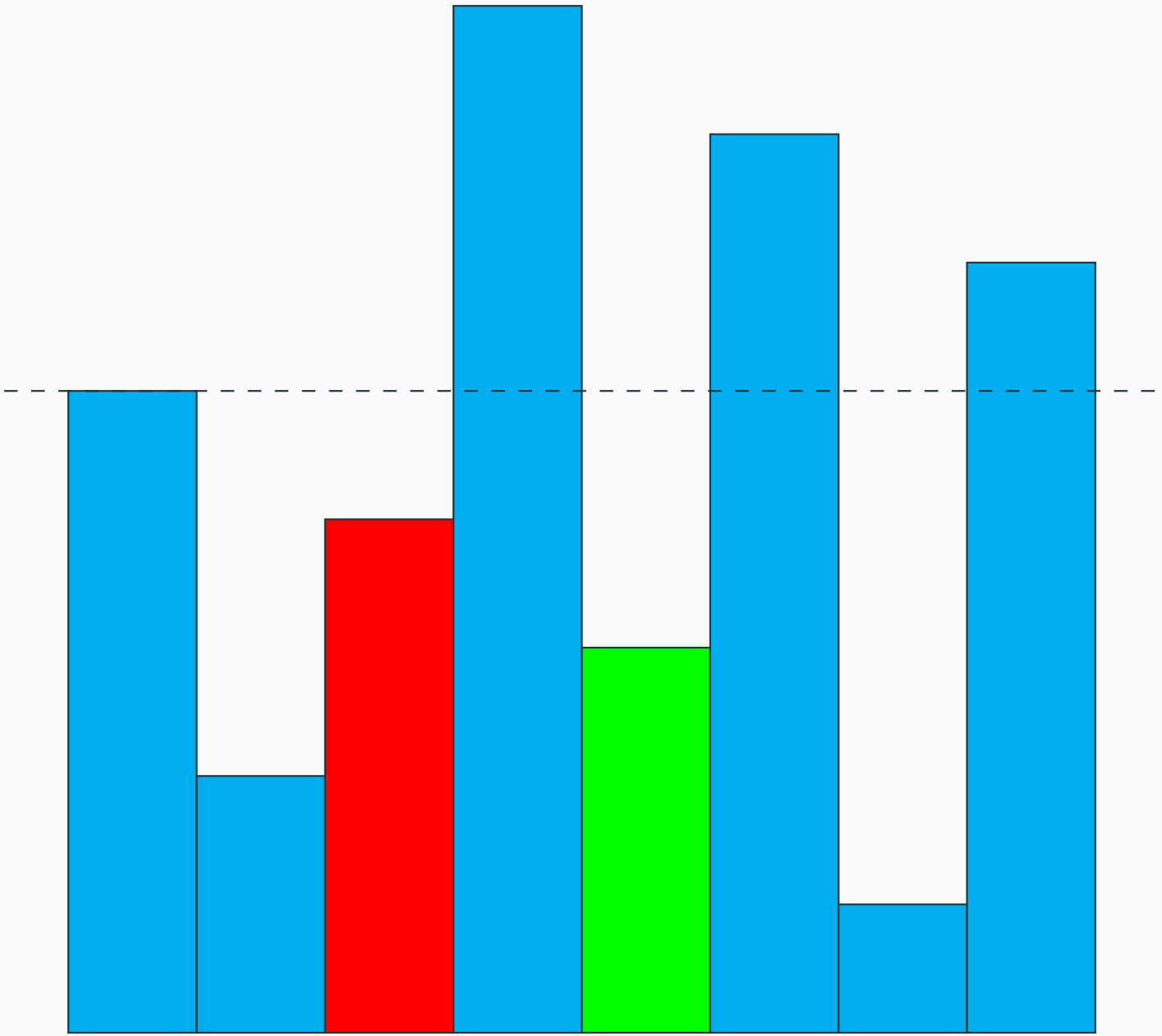
Exemple



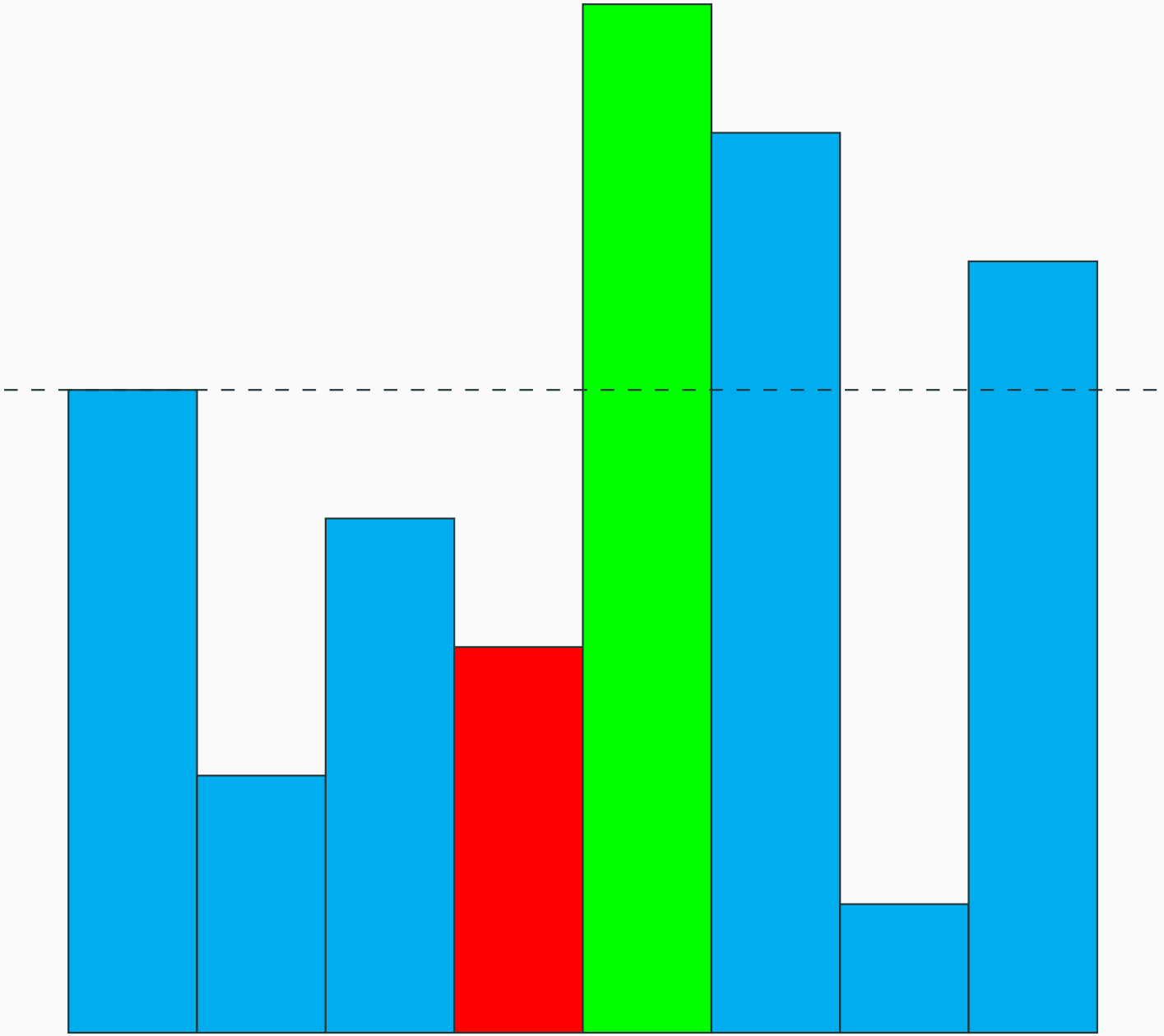
Exemple



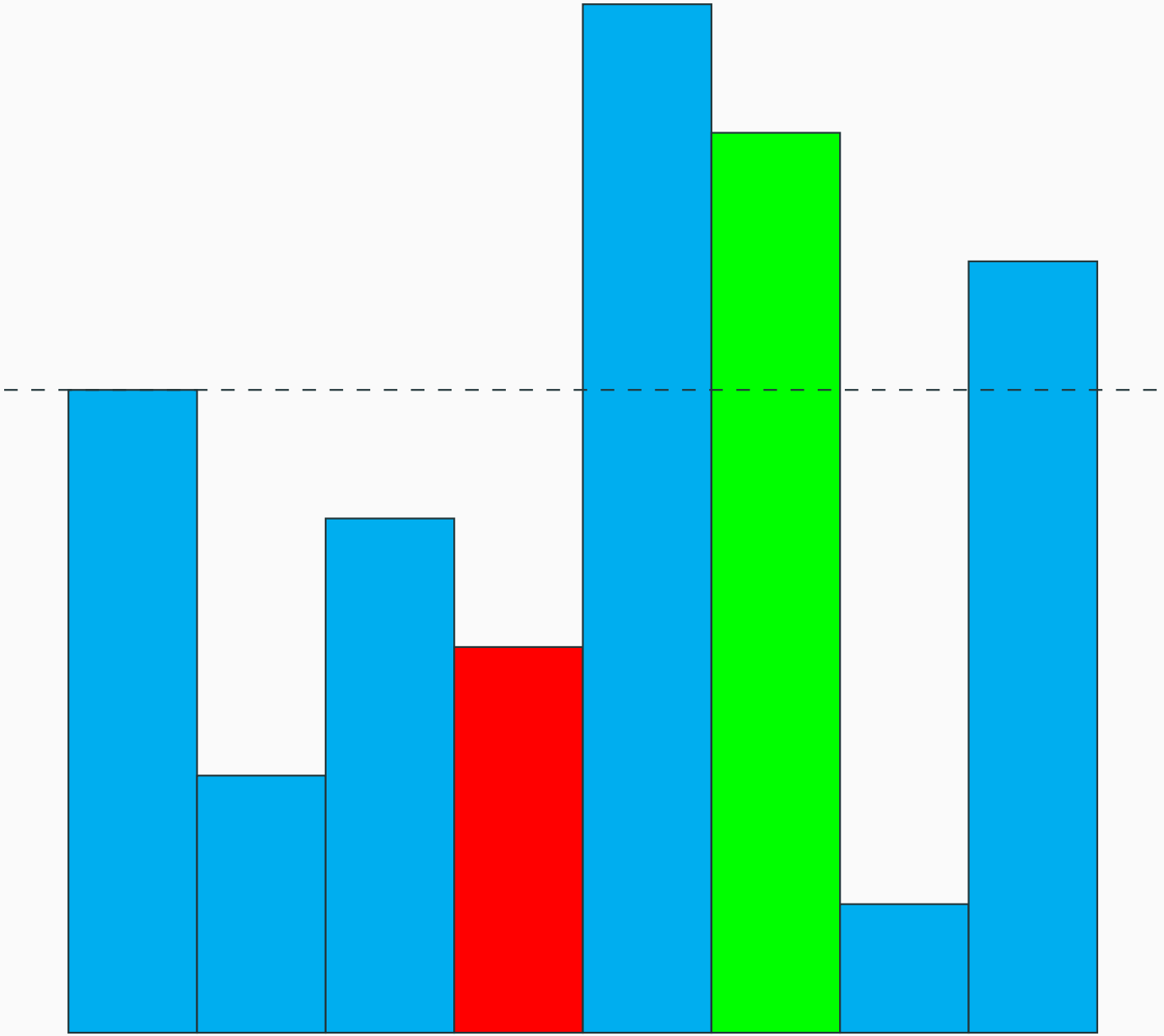
Exemple



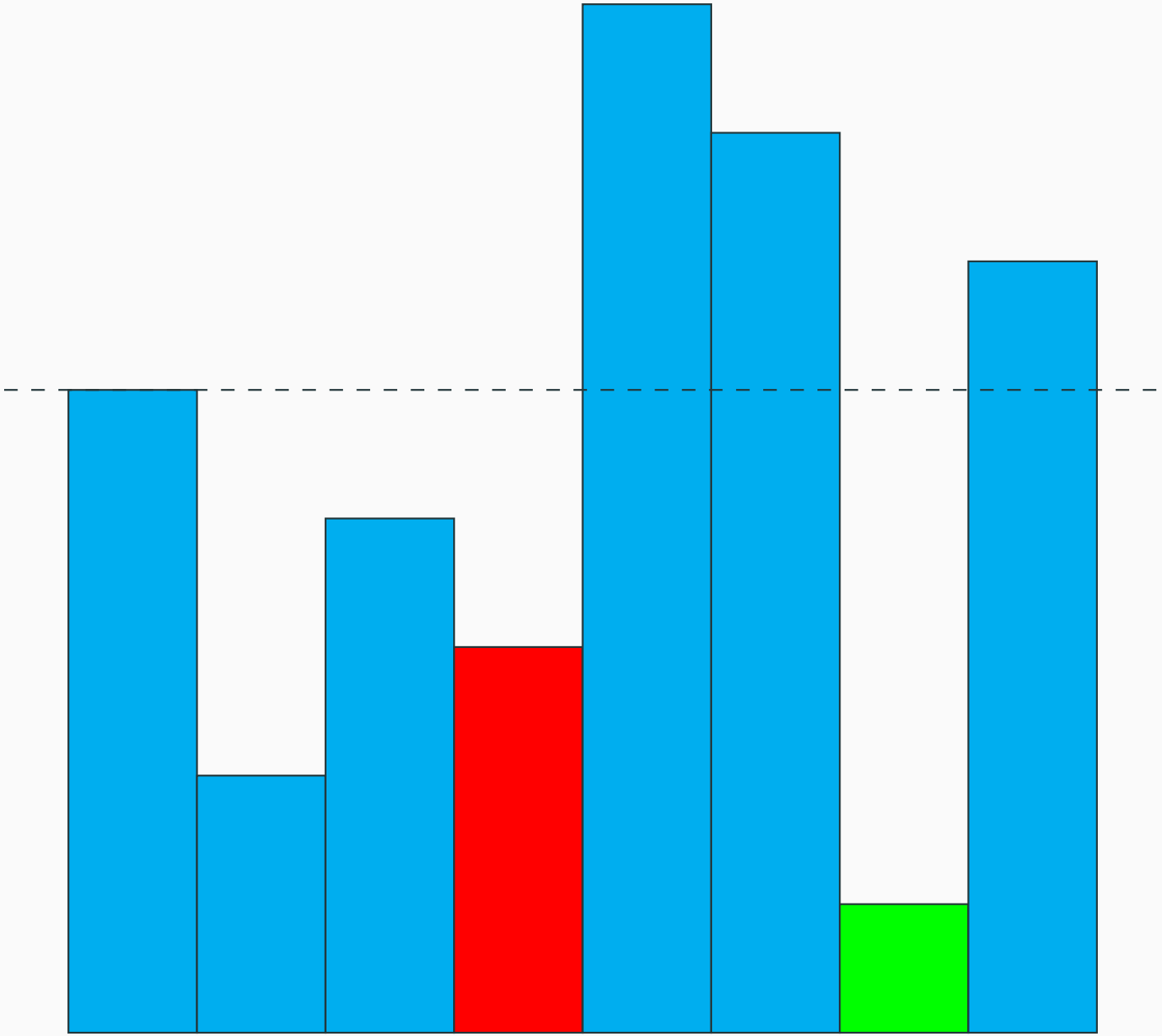
Exemple



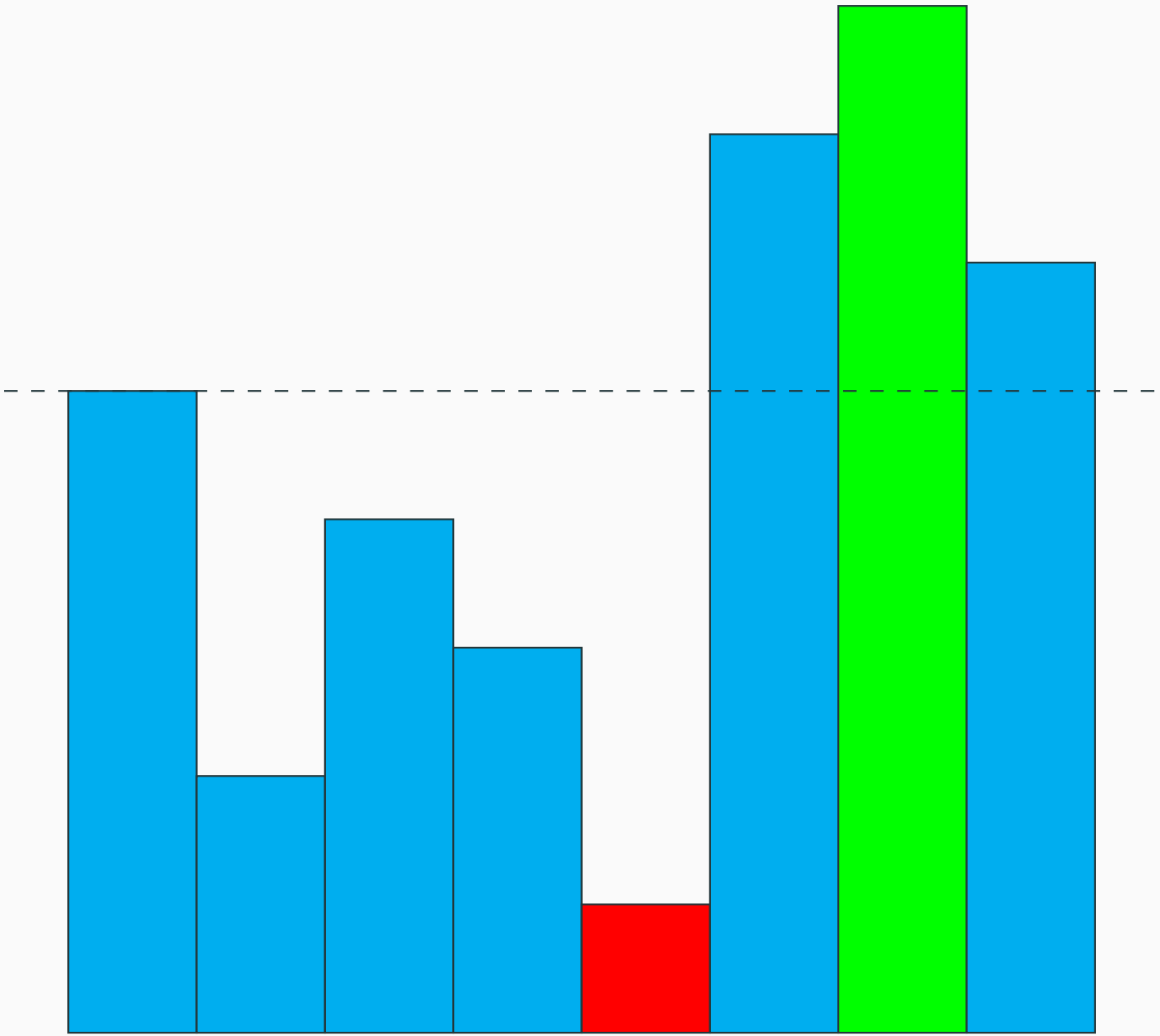
Exemple



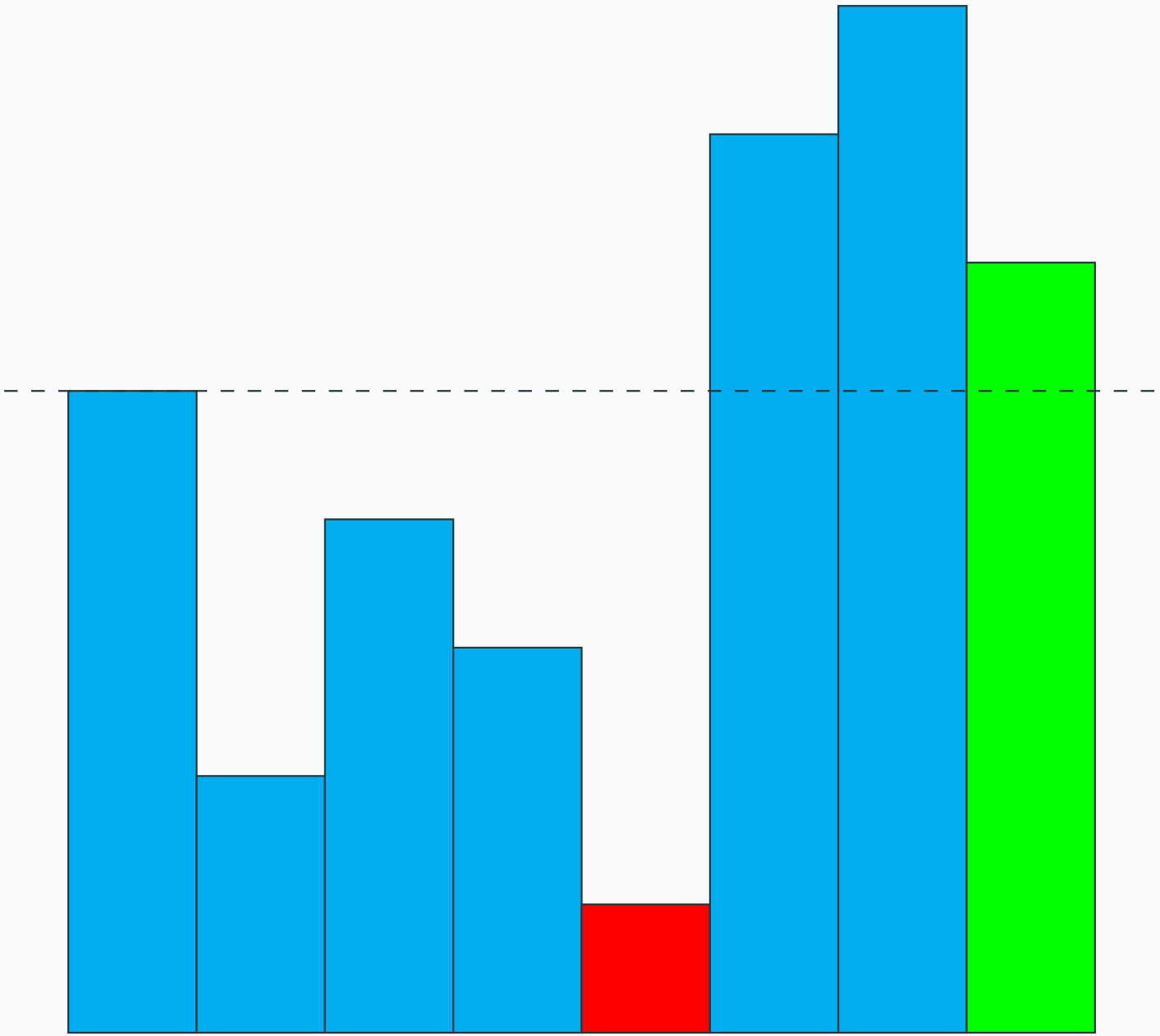
Exemple



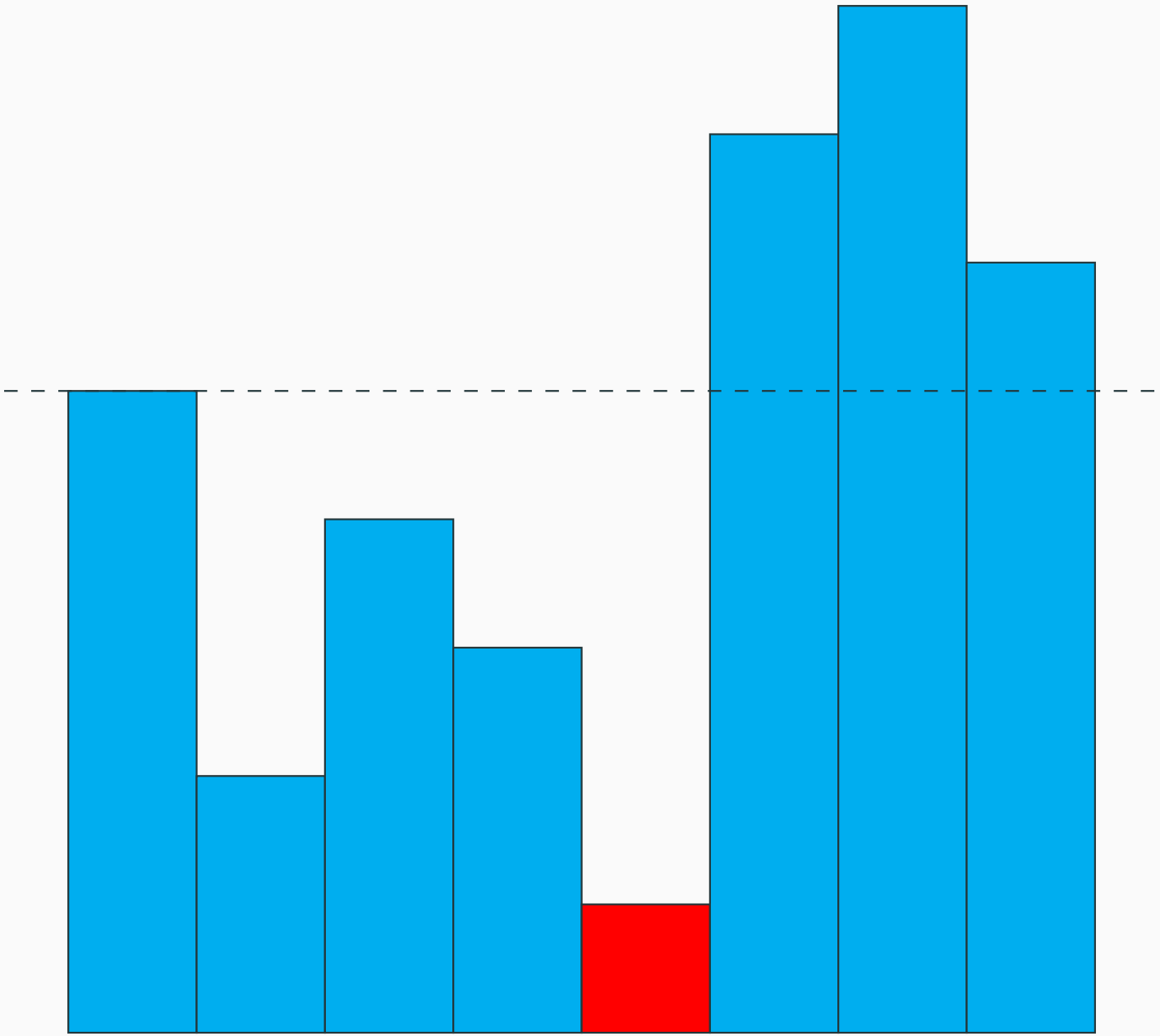
Exemple



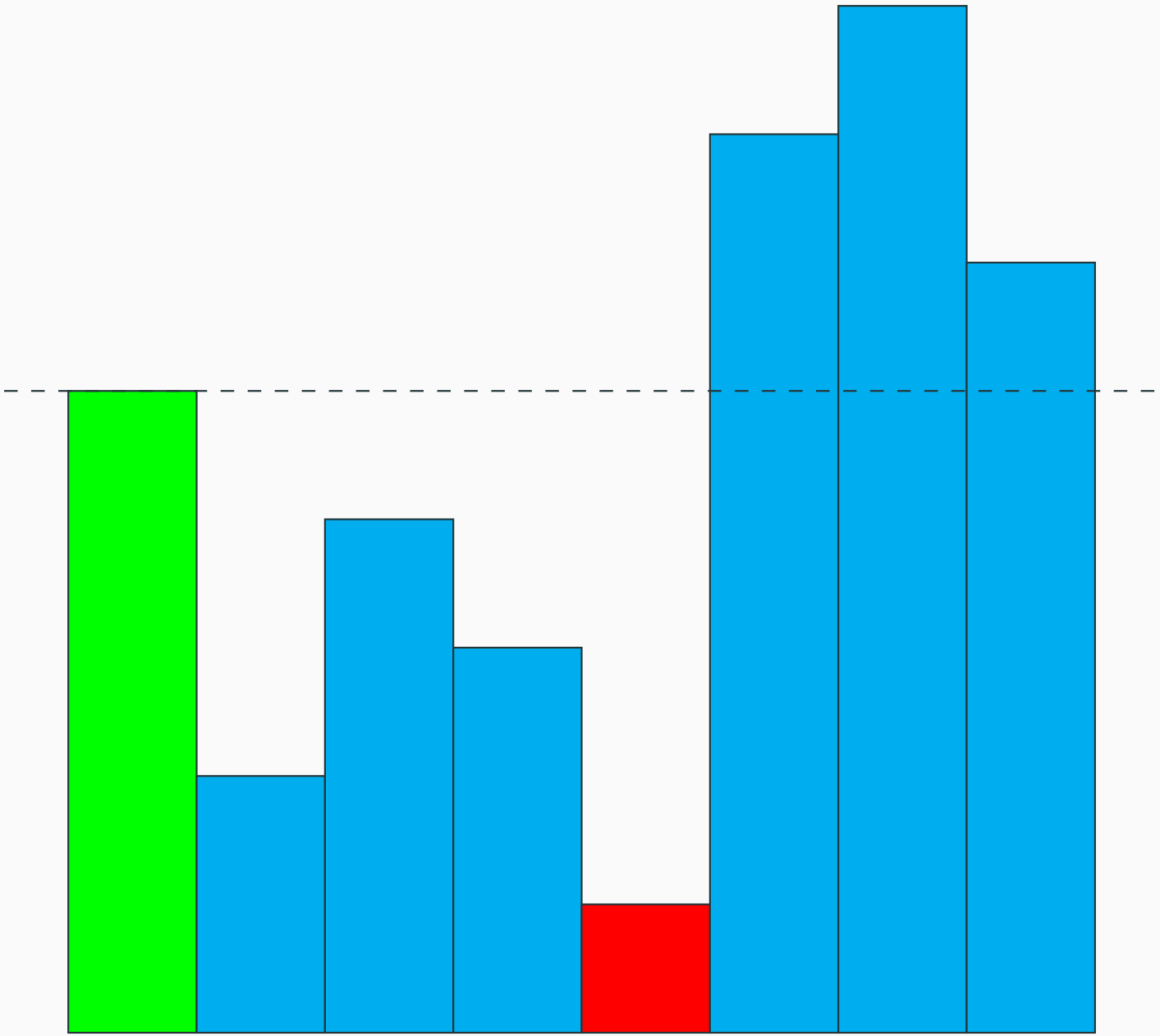
Exemple



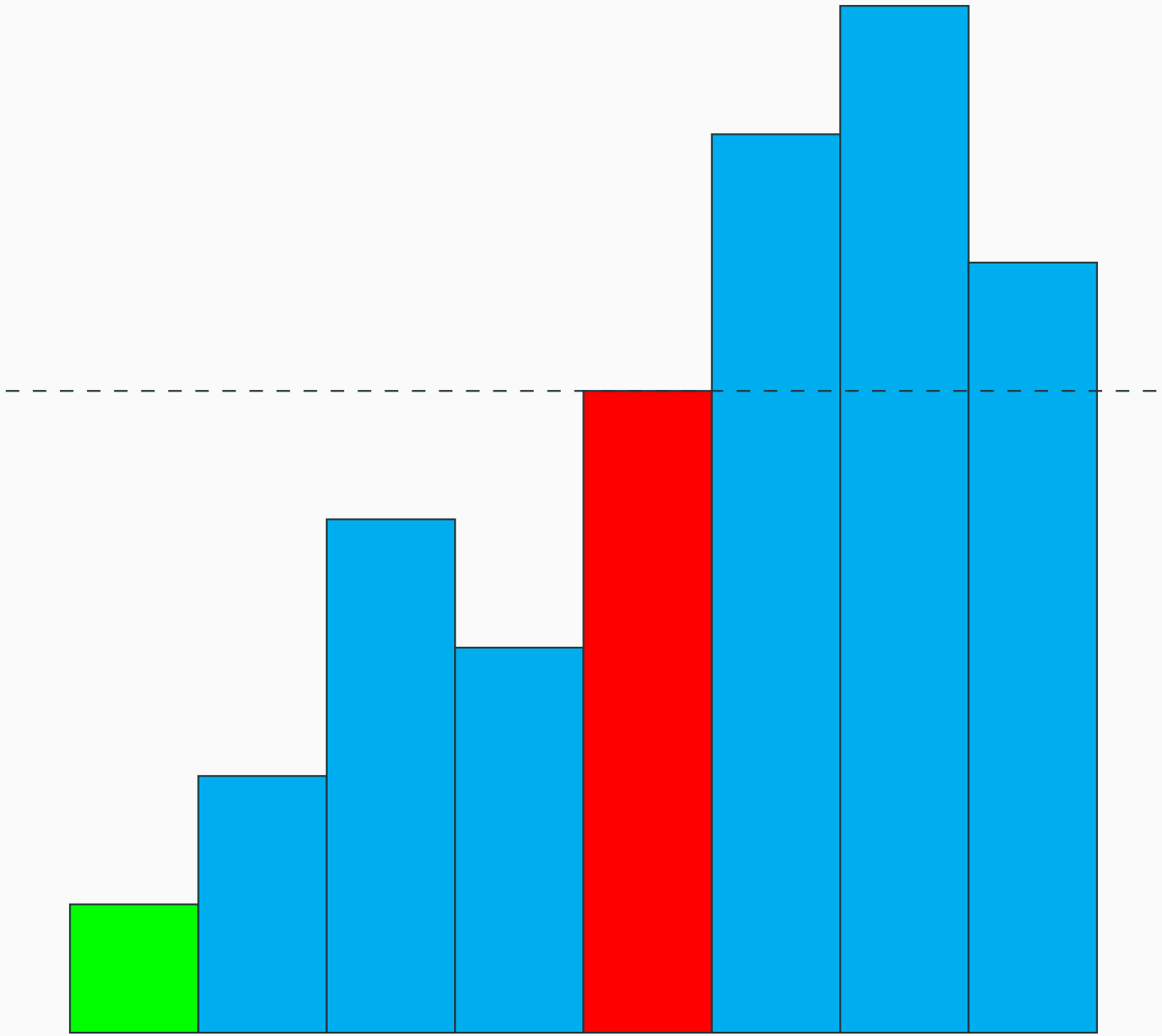
Exemple



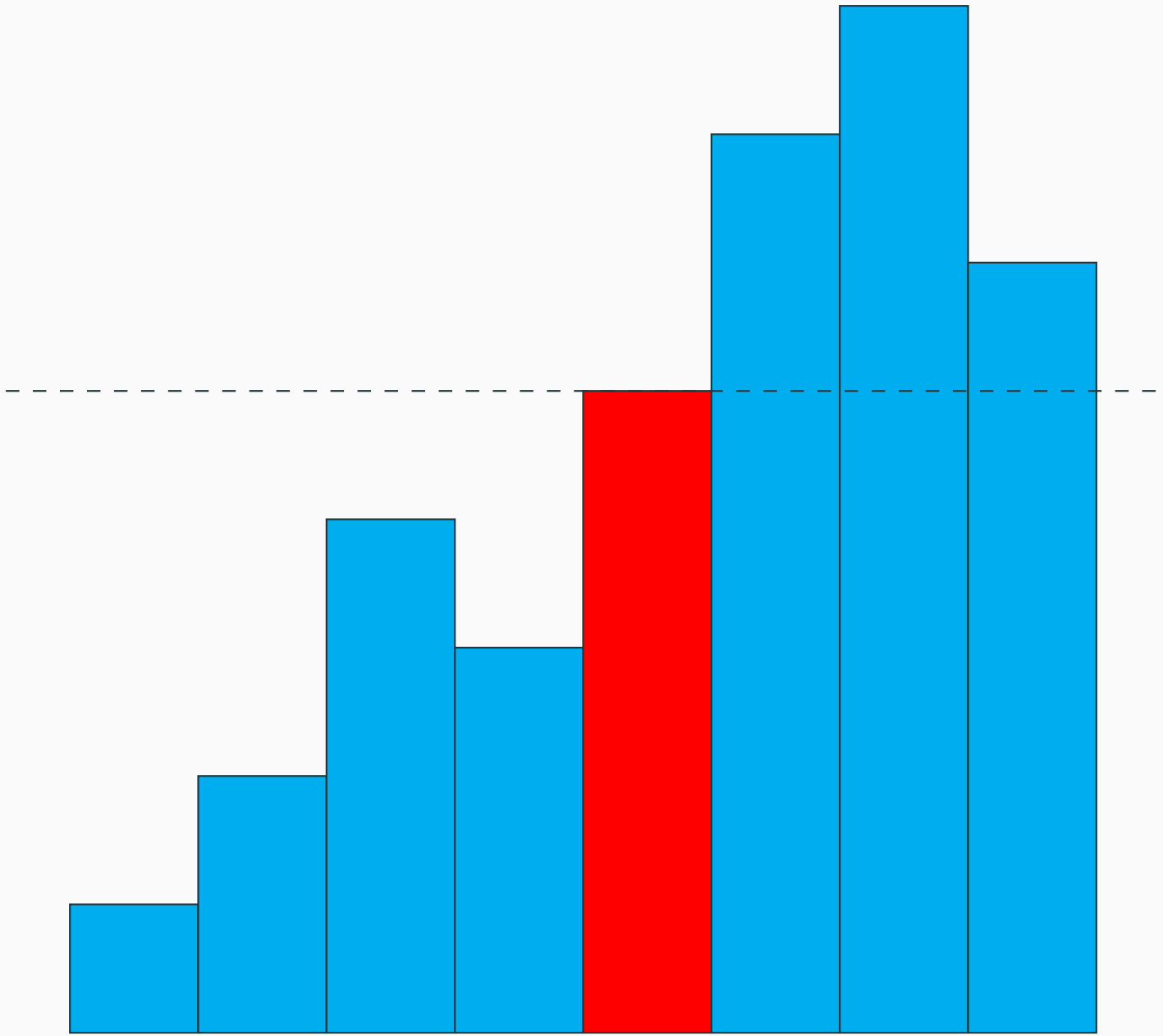
Exemple



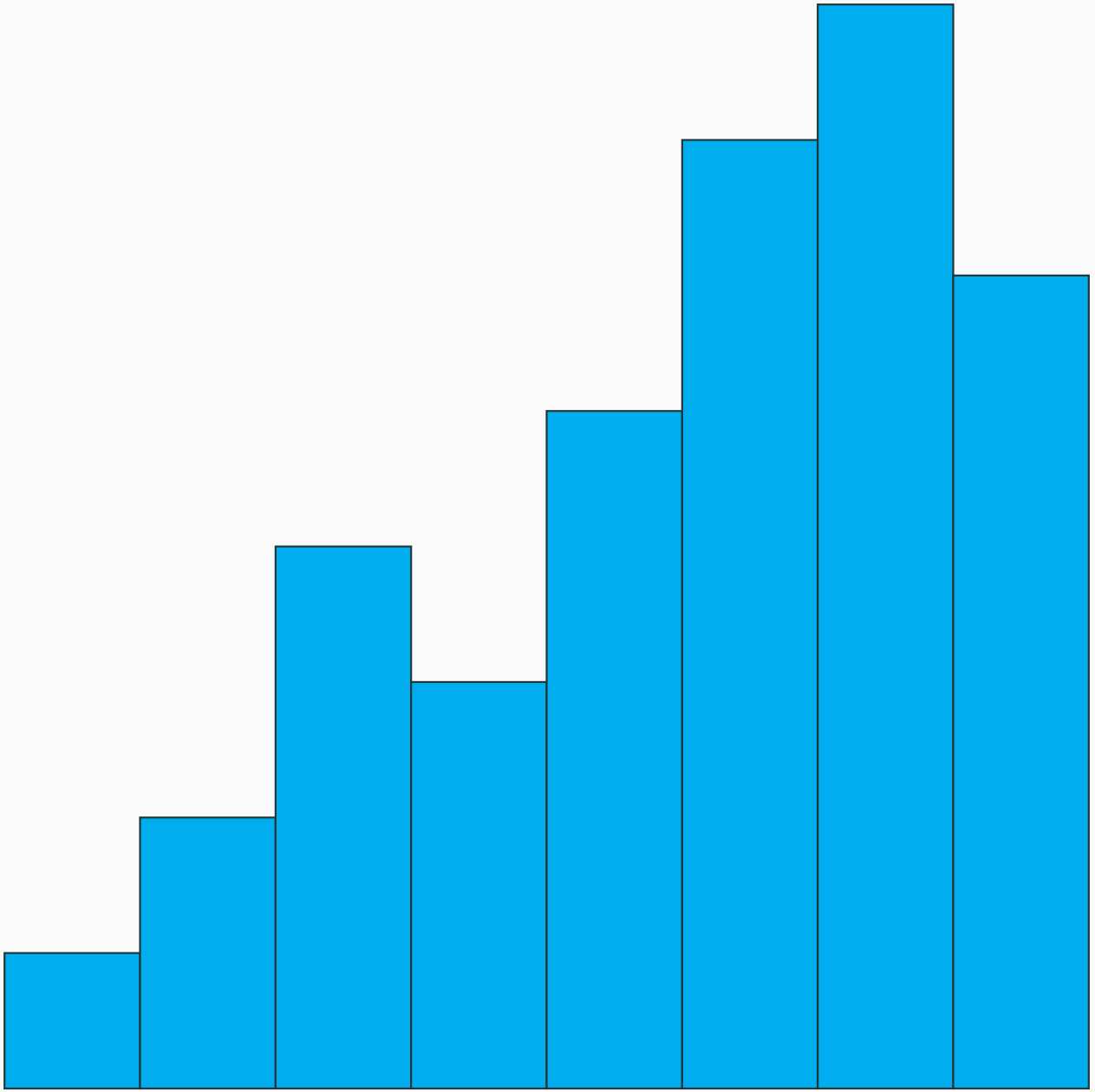
Exemple



Exemple



Exemple



Terminaison et correction

Terminaison

La fonction `partition` termine, car il n'y a pas de boucle `while` ni d'appels récursifs.

Correction

Pour montrer la correction, on va utiliser l'invariant de boucle suivant :

Inv_i : Les éléments du tableau `t[g+1:ip+1]` sont strictement inférieurs au pivot `t.(0)`, les éléments du tableau `t[ip+1:i]` sont supérieurs, avec $i \geq i_p + 1$.

Preuve de correction

Inv_i : Les éléments du tableau `t[g+1:ip+1]` sont strictement inférieurs au pivot `t.(0)`, les éléments du tableau `t[ip+1:i]` sont supérieurs, avec $i \geq i_p + 1$.

- $\mathrm{Inv}_{\mathrm{g+1}}$ est vrai en début de boucle, car dans ce cas $g + 1 = i_p + 1$, donc les deux tableaux mentionnées dans l'invariant sont vides.
- Pour $i \in [\![g + 1, d - 1]\!]$, si Inv_i est vrai en haut de la boucle, alors $\mathrm{Inv}_{\mathrm{i+1}}$ est vrai en bas de la boucle.
  - En effet, si `t.(i)>=pivot`, on ne fait rien.
  - Si `t.(i)<pivot`, on incrémente i_p puis on échange `t.(i)` et `t.(!ip)` (seulement si $i \neq i_p$, sinon il n'y a rien à faire).

Preuve de correction

Inv_i : Les éléments du tableau `t[g+1:ip+1]` sont strictement inférieurs au pivot `t.(0)`, les éléments du tableau `t[ip+1:i]` sont supérieurs, avec $i \geq i_p + 1$.

- Au final, $\mathrm{Inv}_{d-1+1} = \mathrm{Inv}_d$ est vrai après la boucle : les éléments de `t[g+1:ip+1]` sont $<$ au pivot, et les éléments de `t[ip+1:d]` sont $\geq$ au pivot.
- Comme le dernier `if` échange `t.(g)` et `t.(!ip)` (ce qui est nécessaire seulement si $i_p > g$), la fonction se termine avec `t` partitionné au niveau du pivot.
- On renvoie i_p, qui est maintenant l'indice du pivot.

Complexité

Complexité

Dans tous les cas, la fonction `partition t g d` fait exactement $d - g - 1$ **comparaisons**.

Dans le meilleur cas, elle ne fait aucun **échange** (lorsque les éléments sont tous $<$ ou tous $\geq$ au pivot).

Dans le pire des cas, elle fait $d - g - 1$ **échanges**.

En pratique, on espère que la partition soit équilibrée au maximum, et avoir un nombre d'**échanges** proche de $\frac{d-g}{2}$.

Le tri rapide

Idée

Si on a une portion délimitée par les indices g (inclus) et d (exclu) à trier, alors :

- si $g \geq d - 1$, la portion possède 0 ou 1 élément, donc est triée : il n'y a rien à faire.
- sinon, on réalise une partition entre ces indices.
  On obtient alors l'indice i_p où l'on a mis le pivot, où :
  - les éléments de `t[g:ip]` sont $<$ `t.(g)`,
  - ceux de `t[ip+1:d]` sont $\geq$ `t.(g)`.

  Il suffit de recommencer sur ces portions du tableau.

OCaml

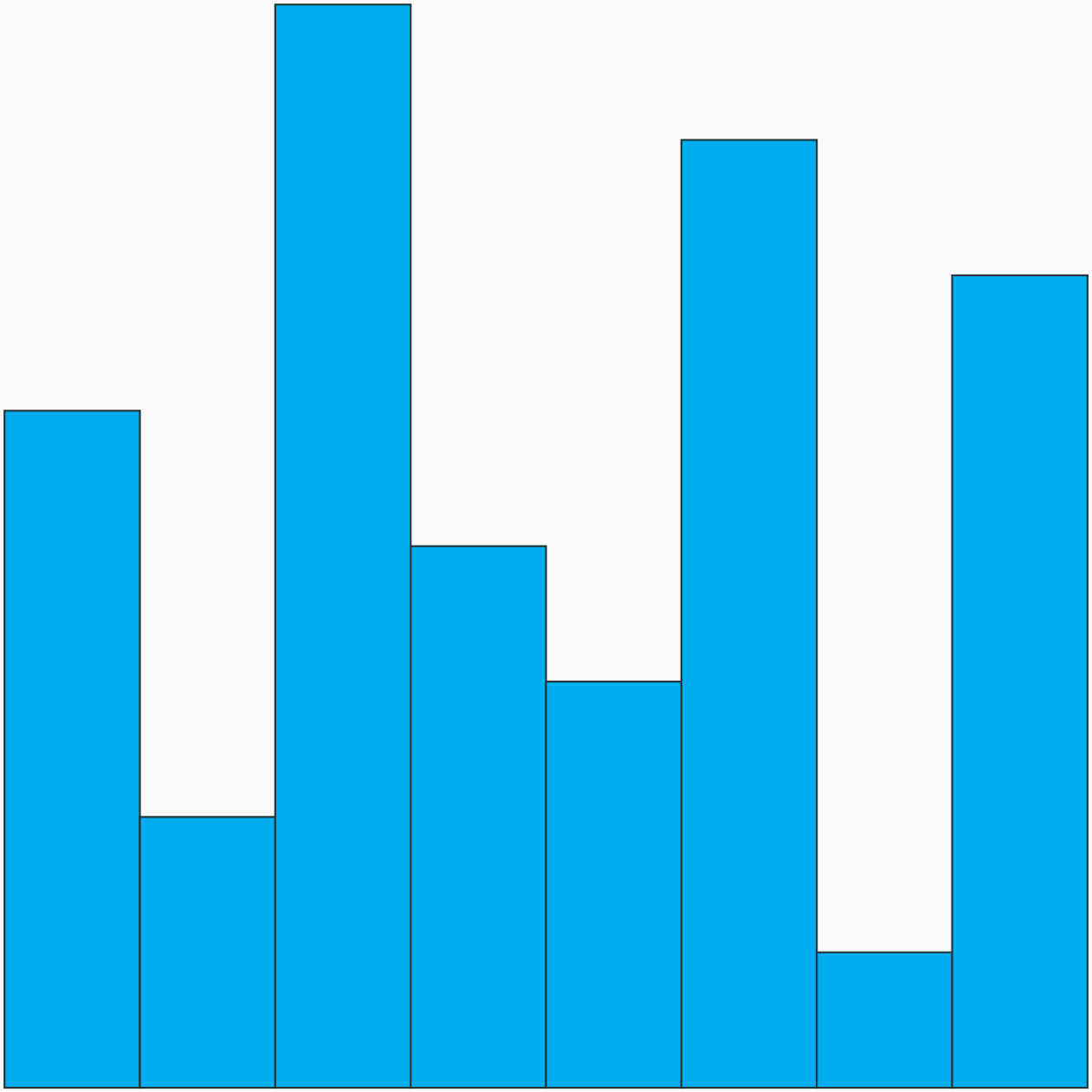
```
1  let tri_rapide t =
2    let rec aux g d = match g<d-1 with
3      | false -> ()
4      | true -> let ip = partition t g d in
5        aux g ip ;
6        aux (ip+1) d
7    in
8    aux 0 (Array.length t)
9  ;;
```

Fonction auxiliaire

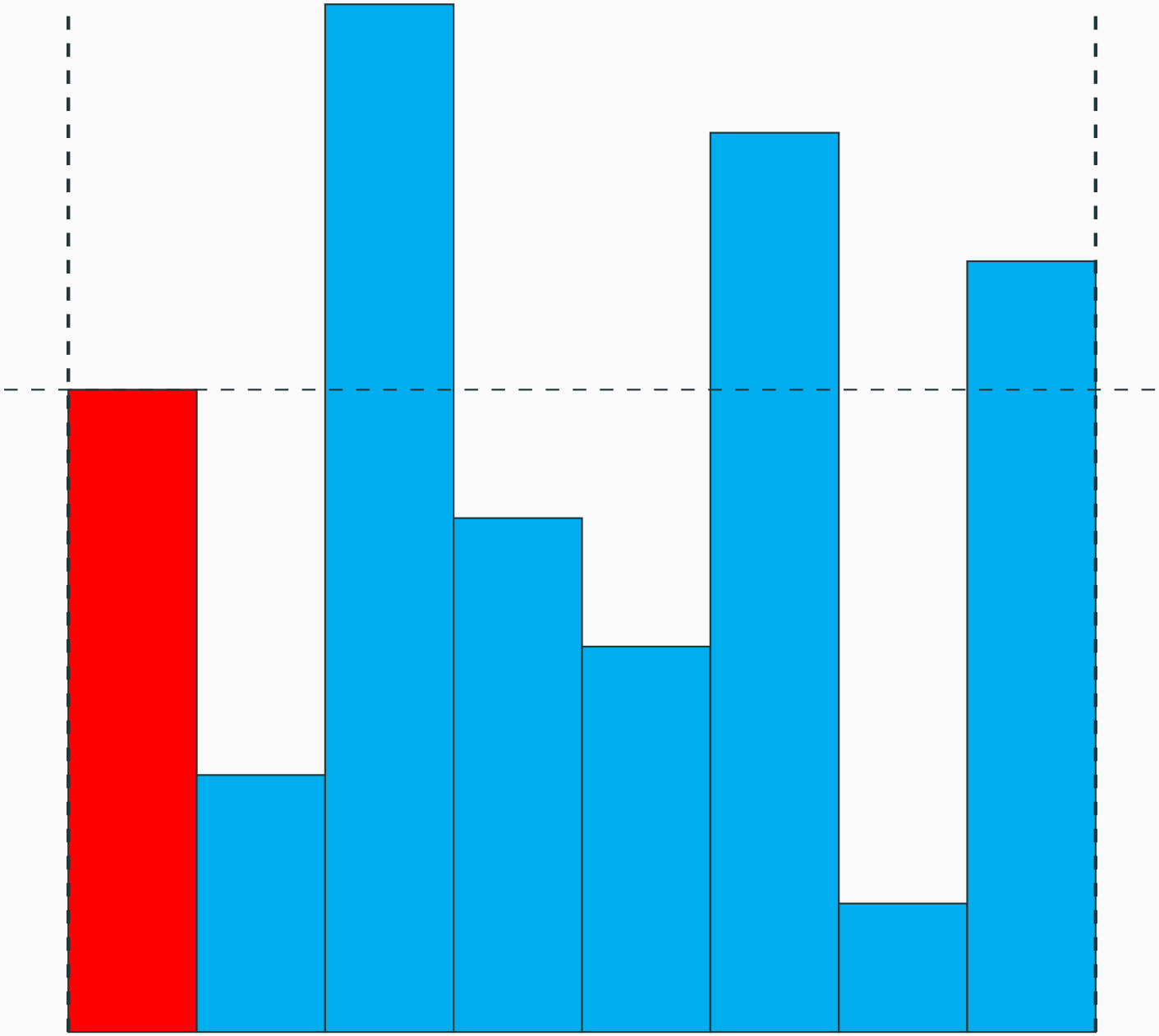
On utilise une fonction auxiliaire locale qui effectue le tri rapide (récursif) entre les indices g et d.

La fonction principale appelle cette fonction aux sur toute le tableau.

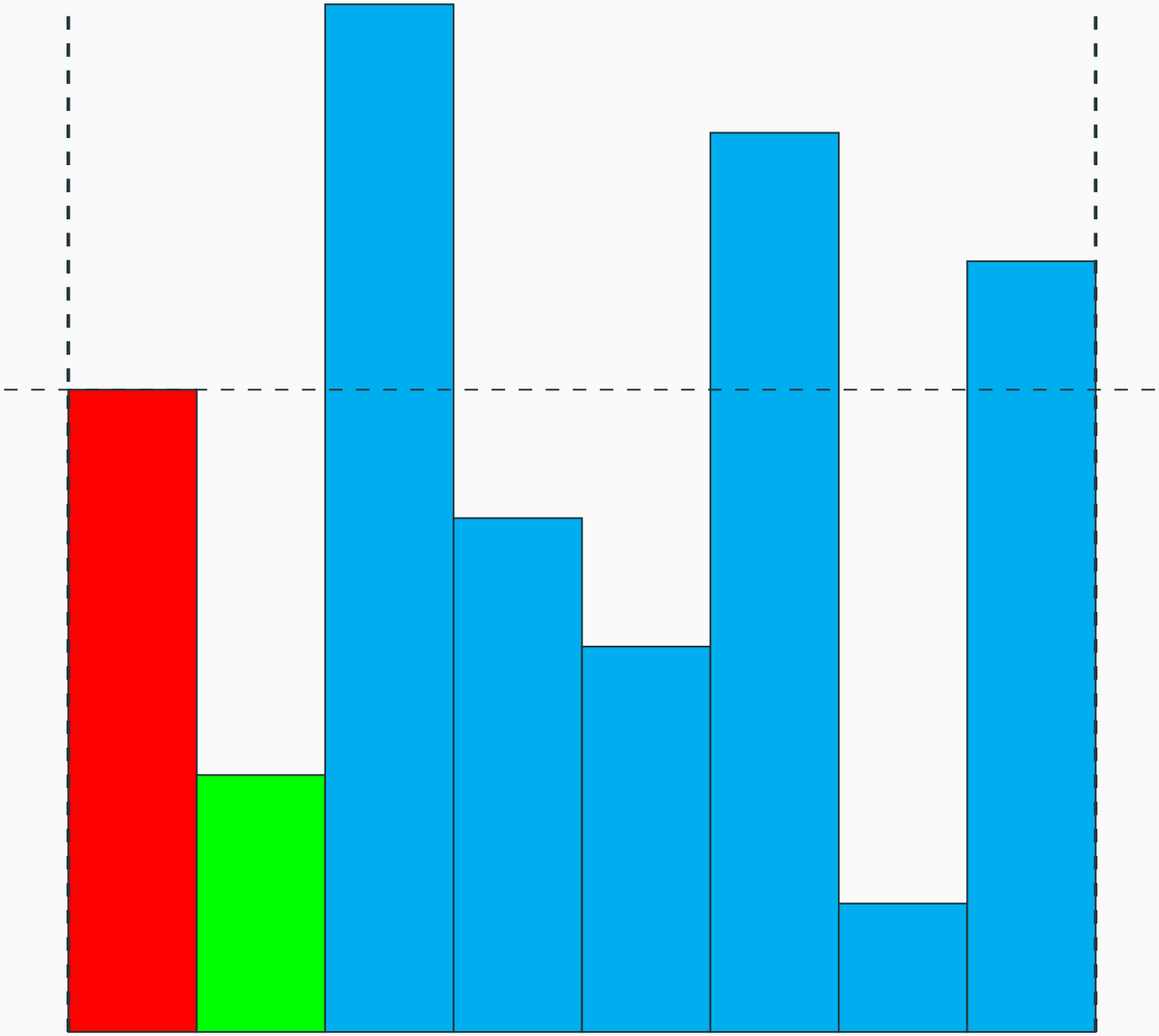
Exemple



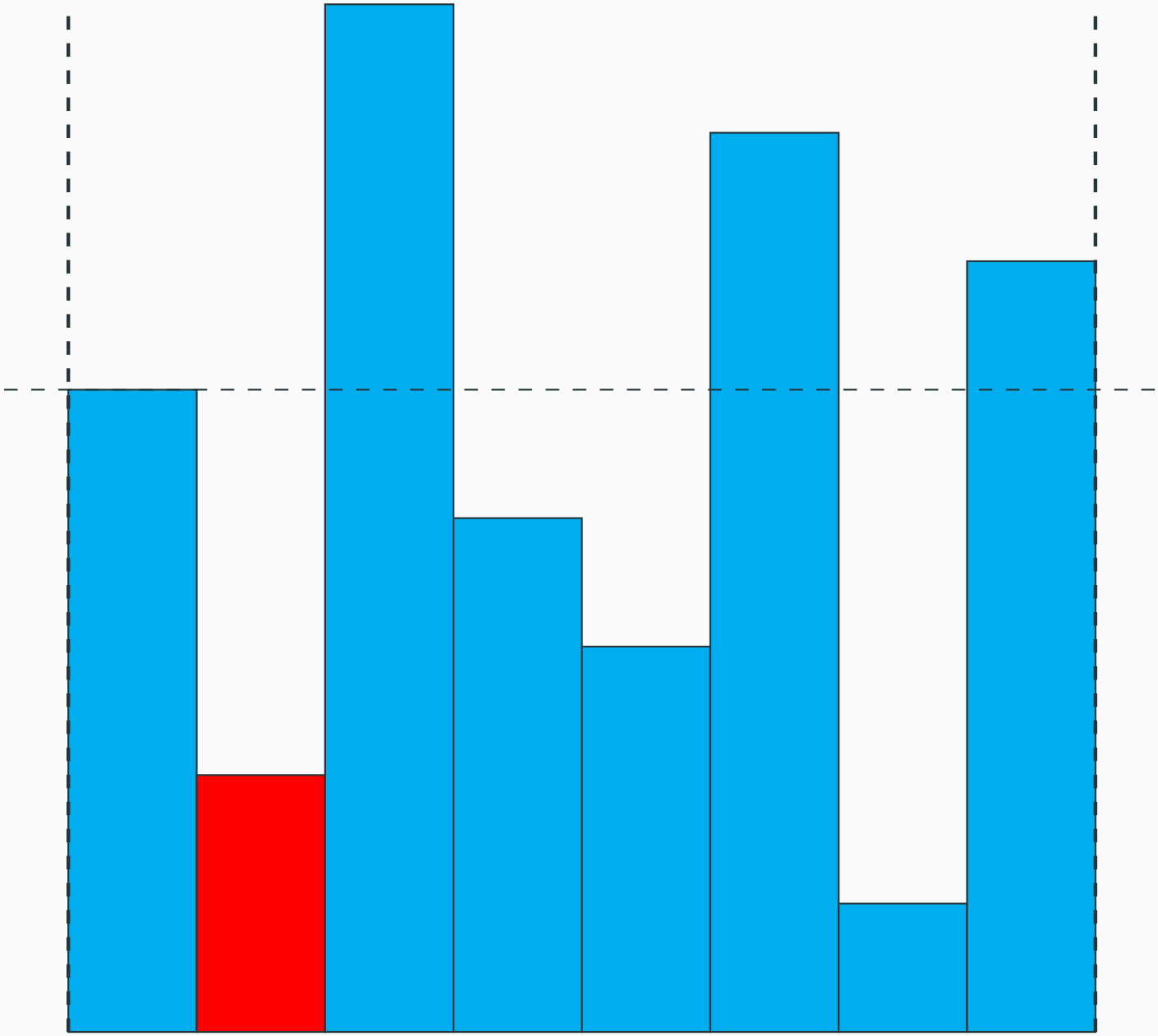
Exemple



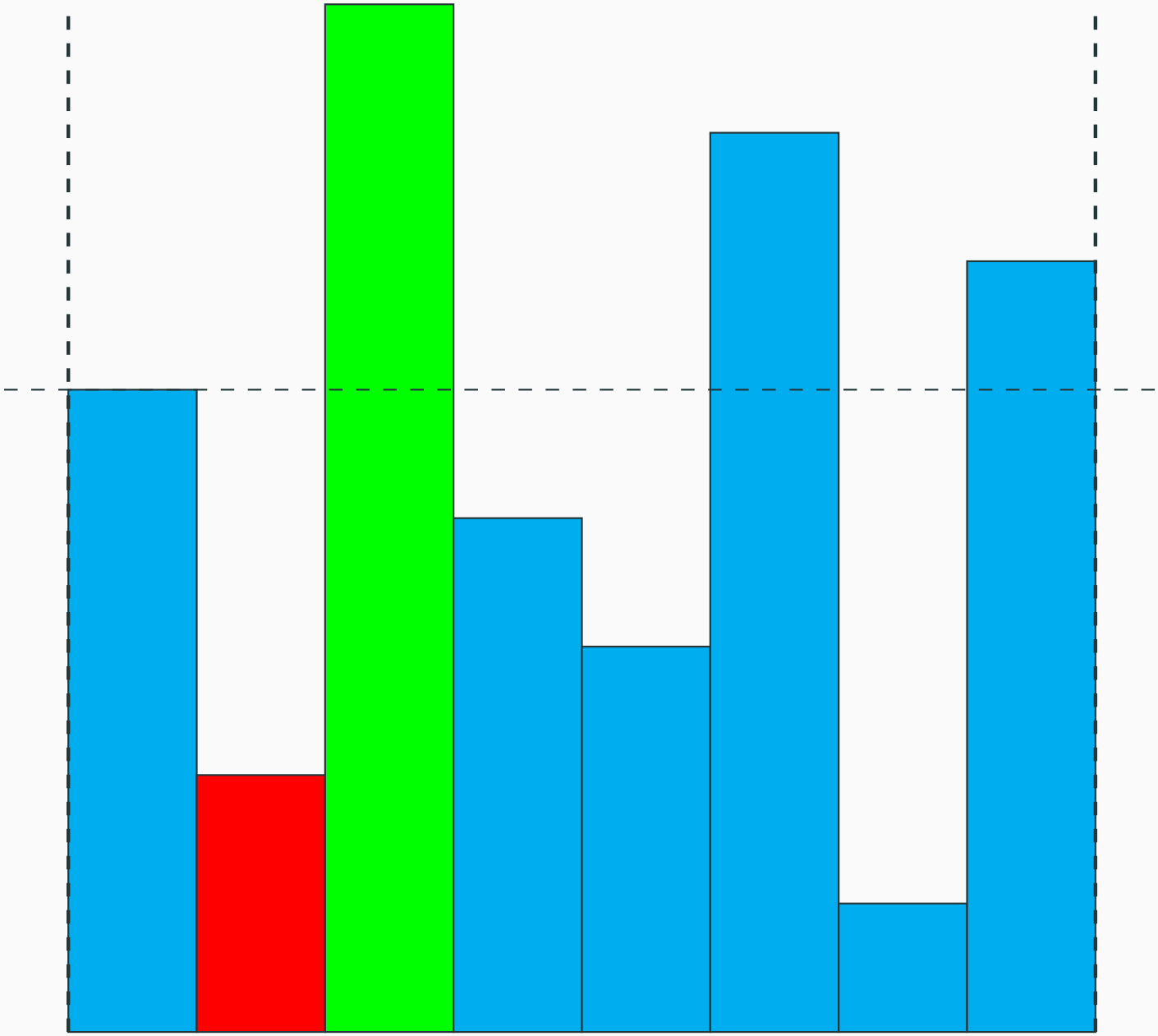
Exemple



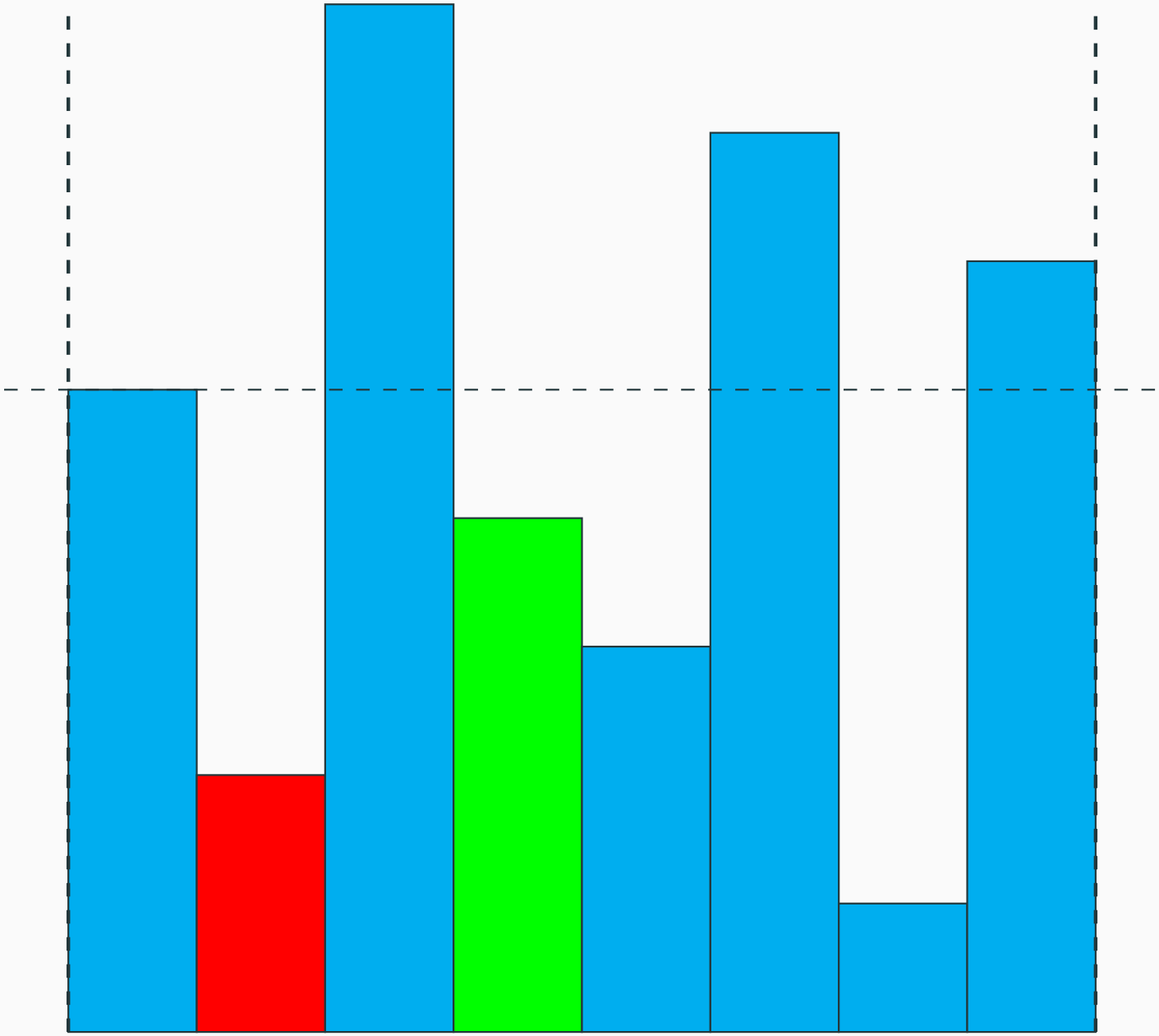
Exemple



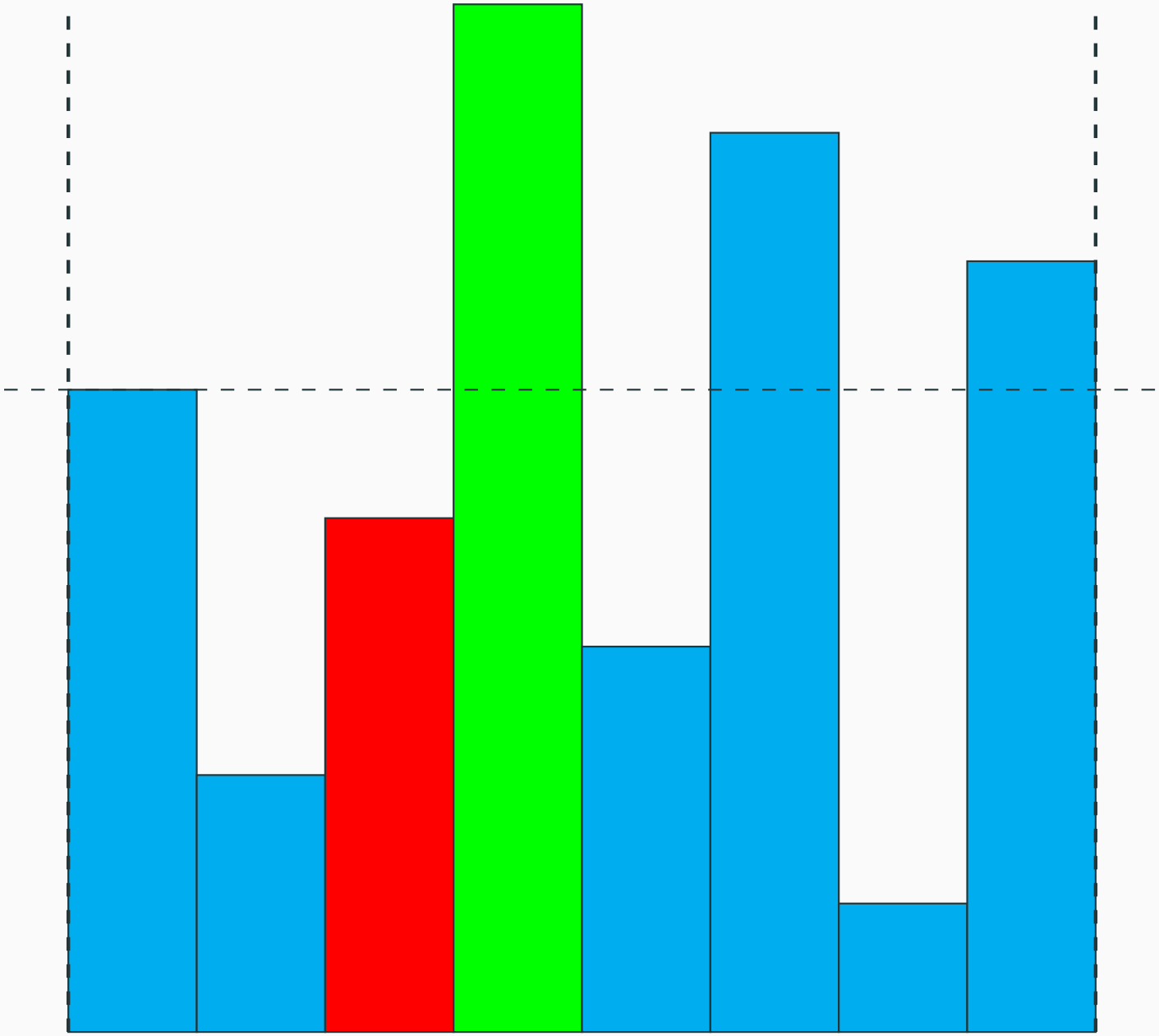
Exemple



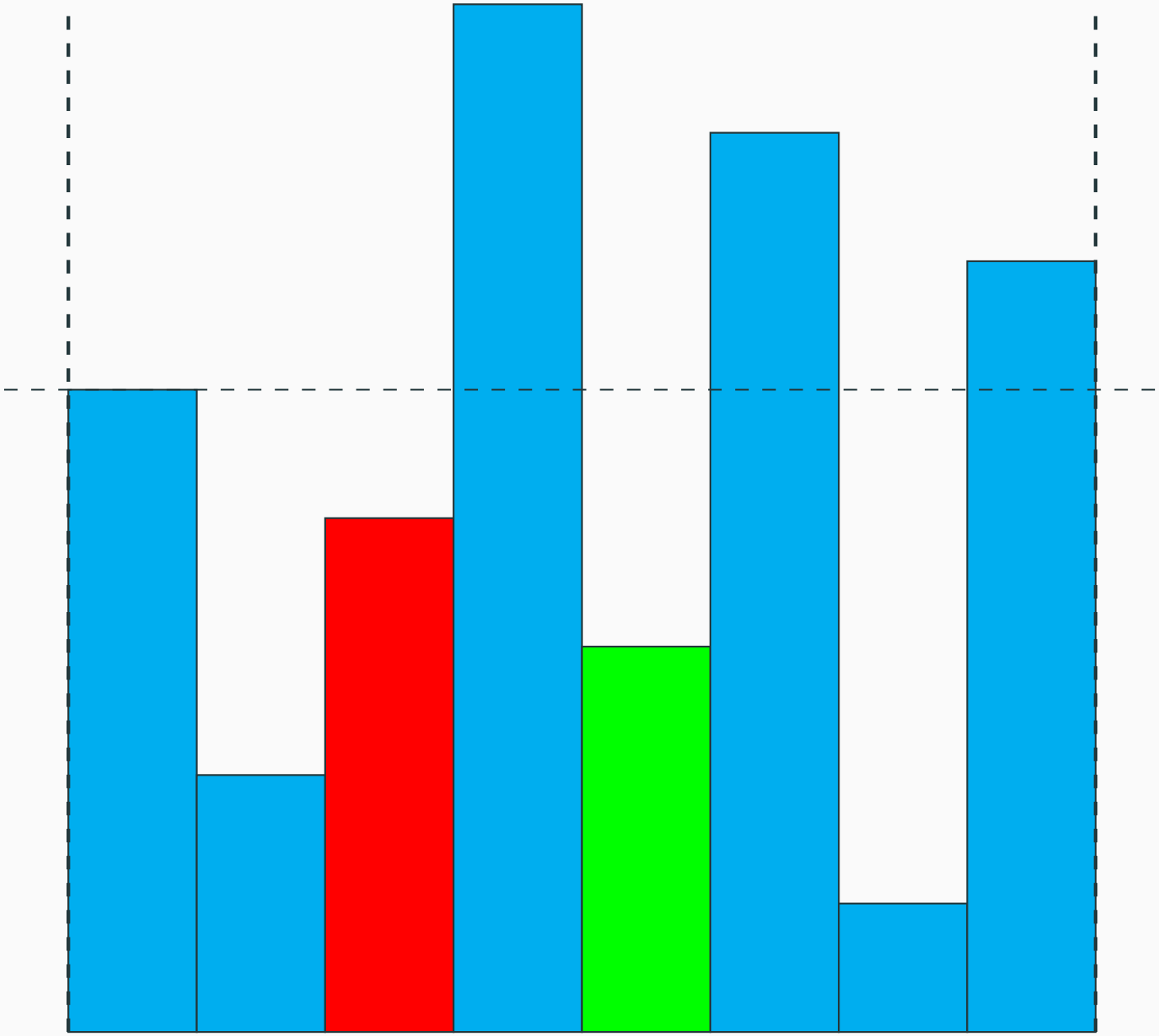
Exemple



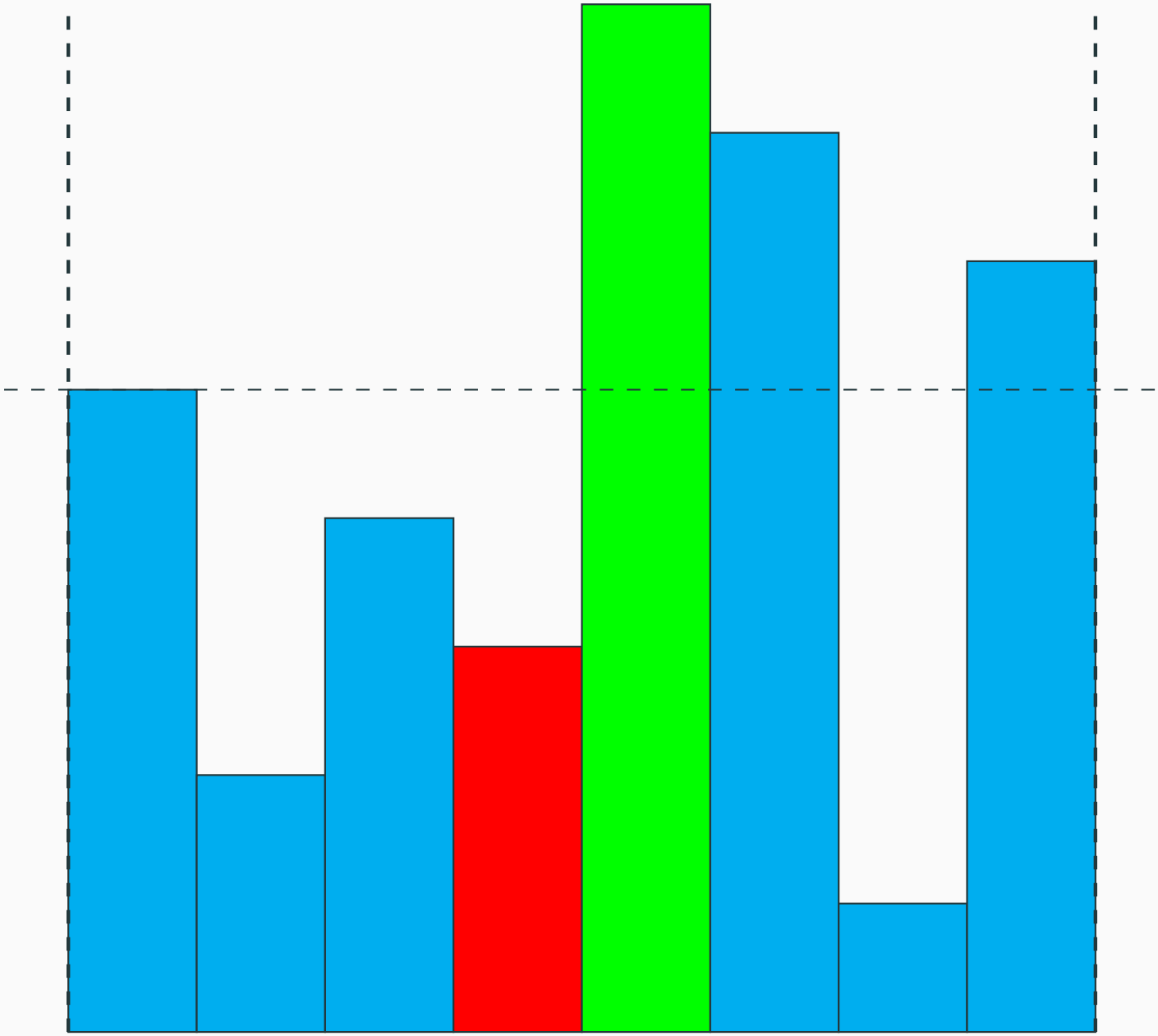
Exemple



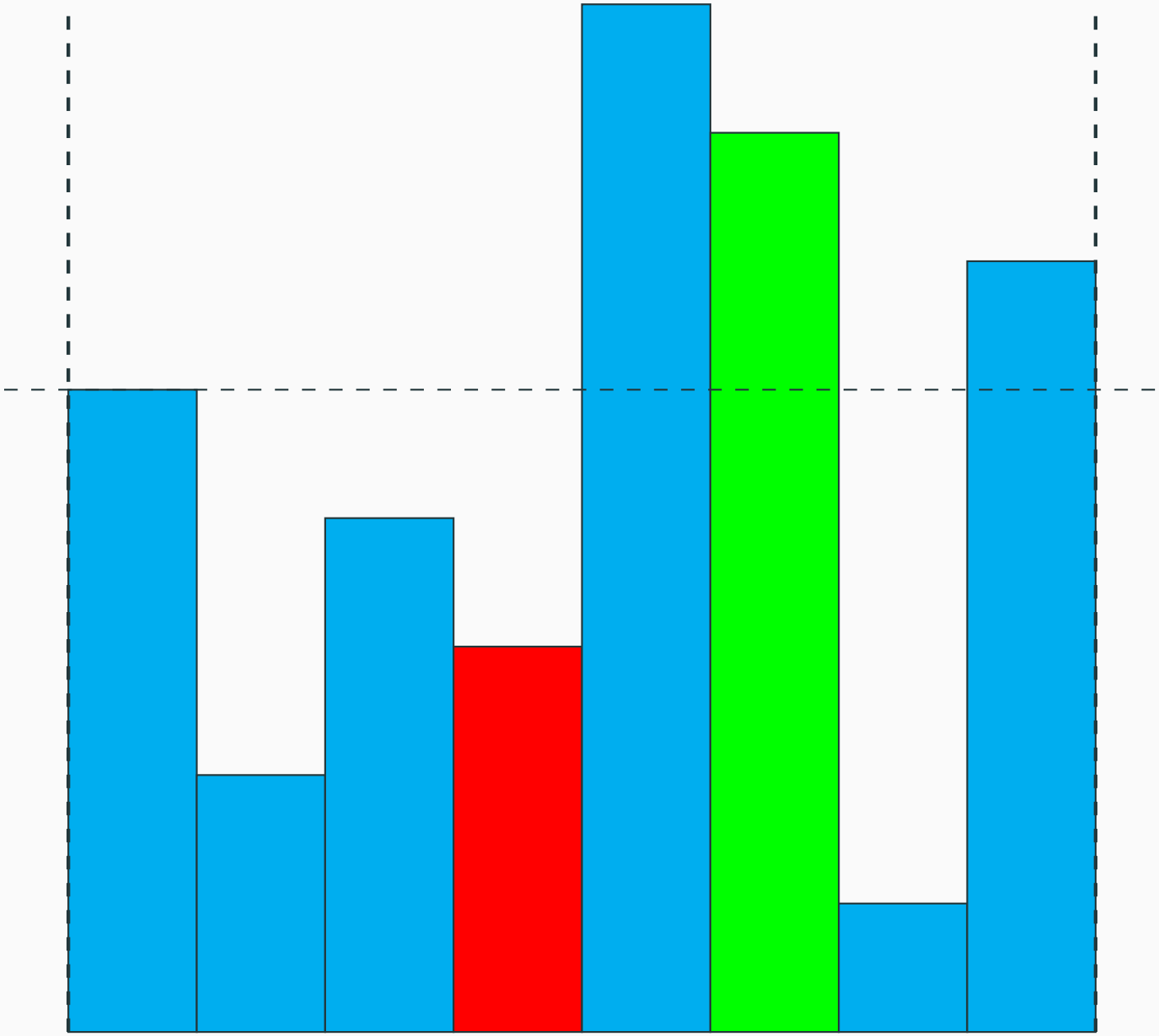
Exemple



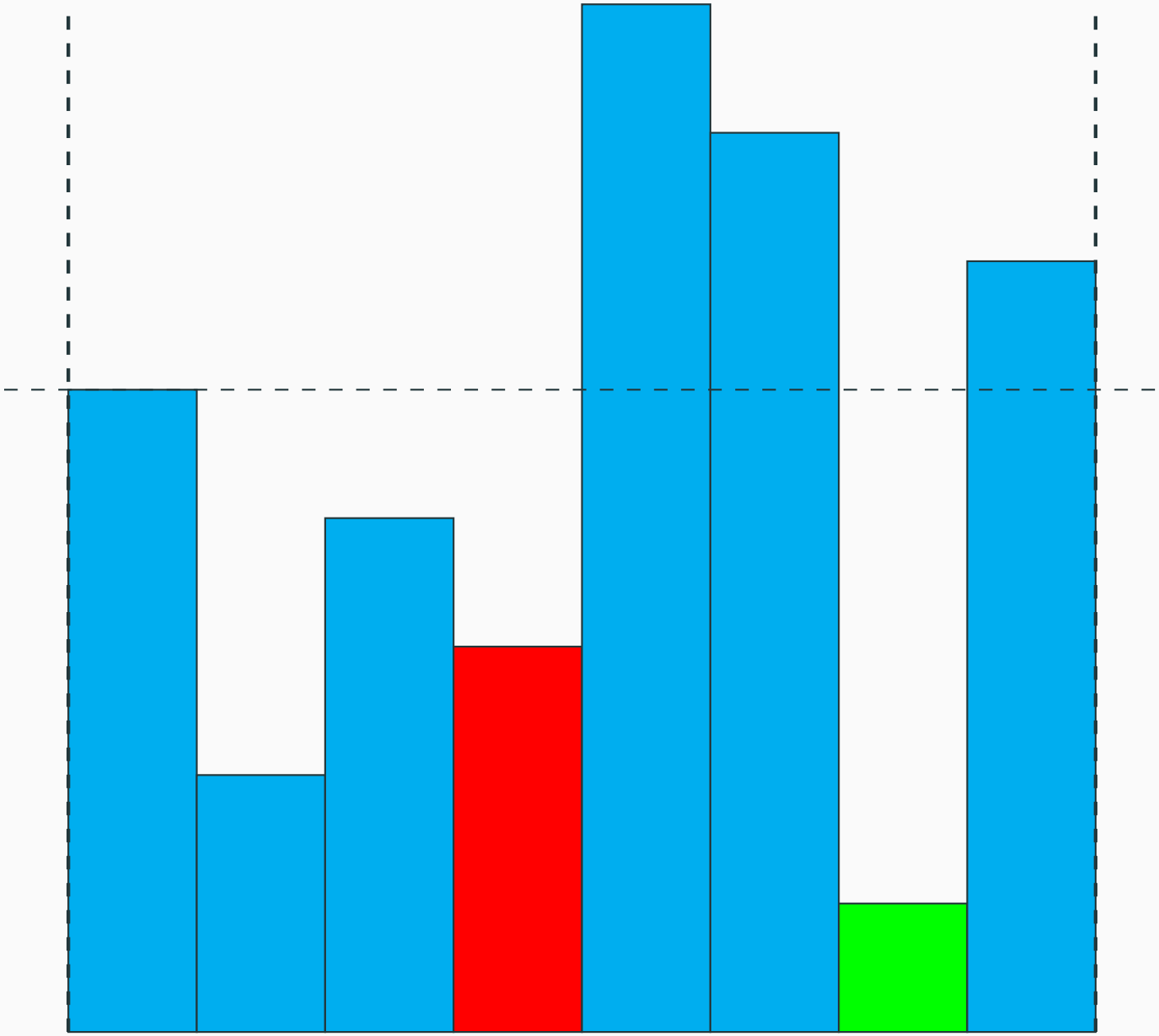
Exemple



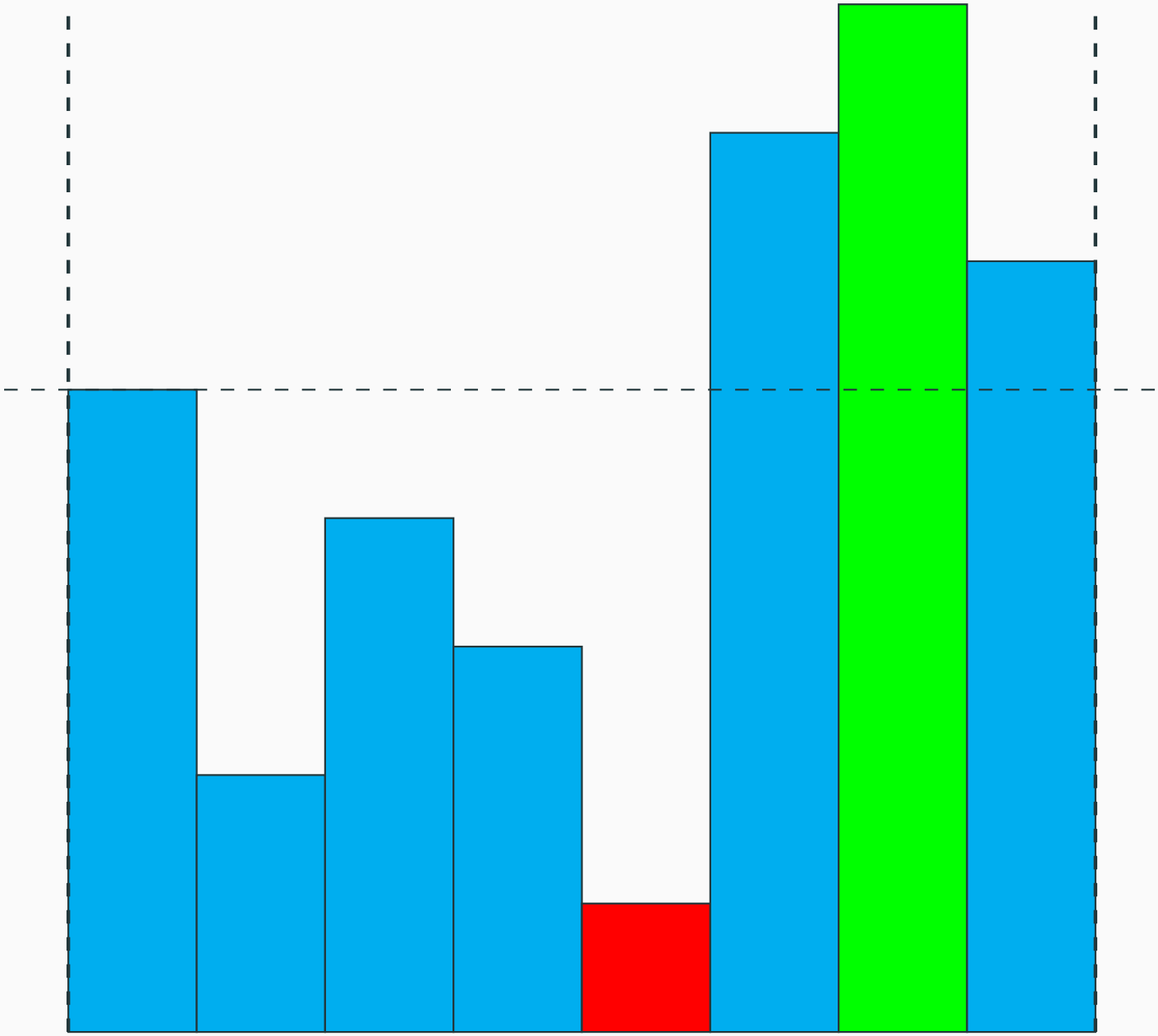
Exemple



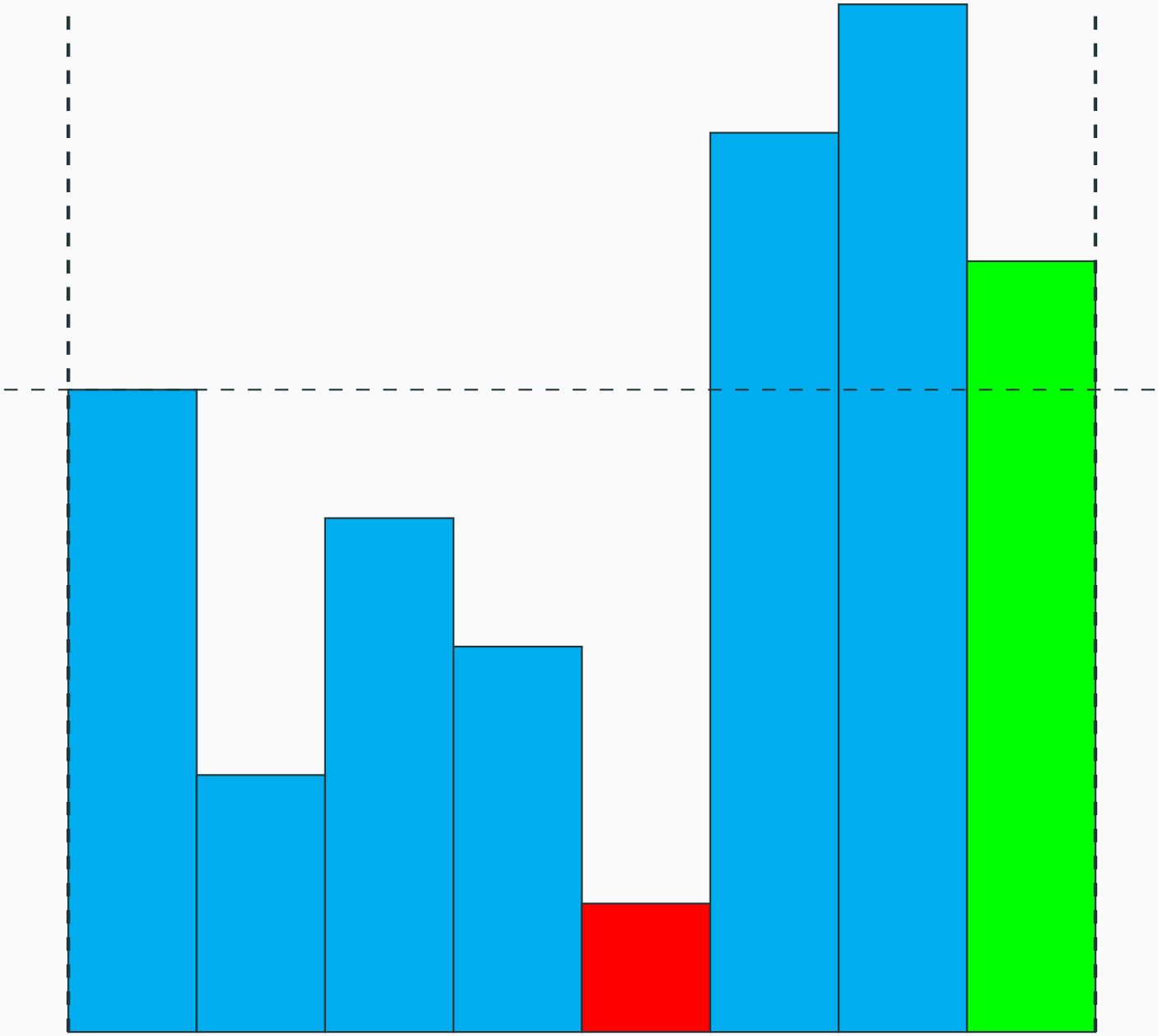
Exemple



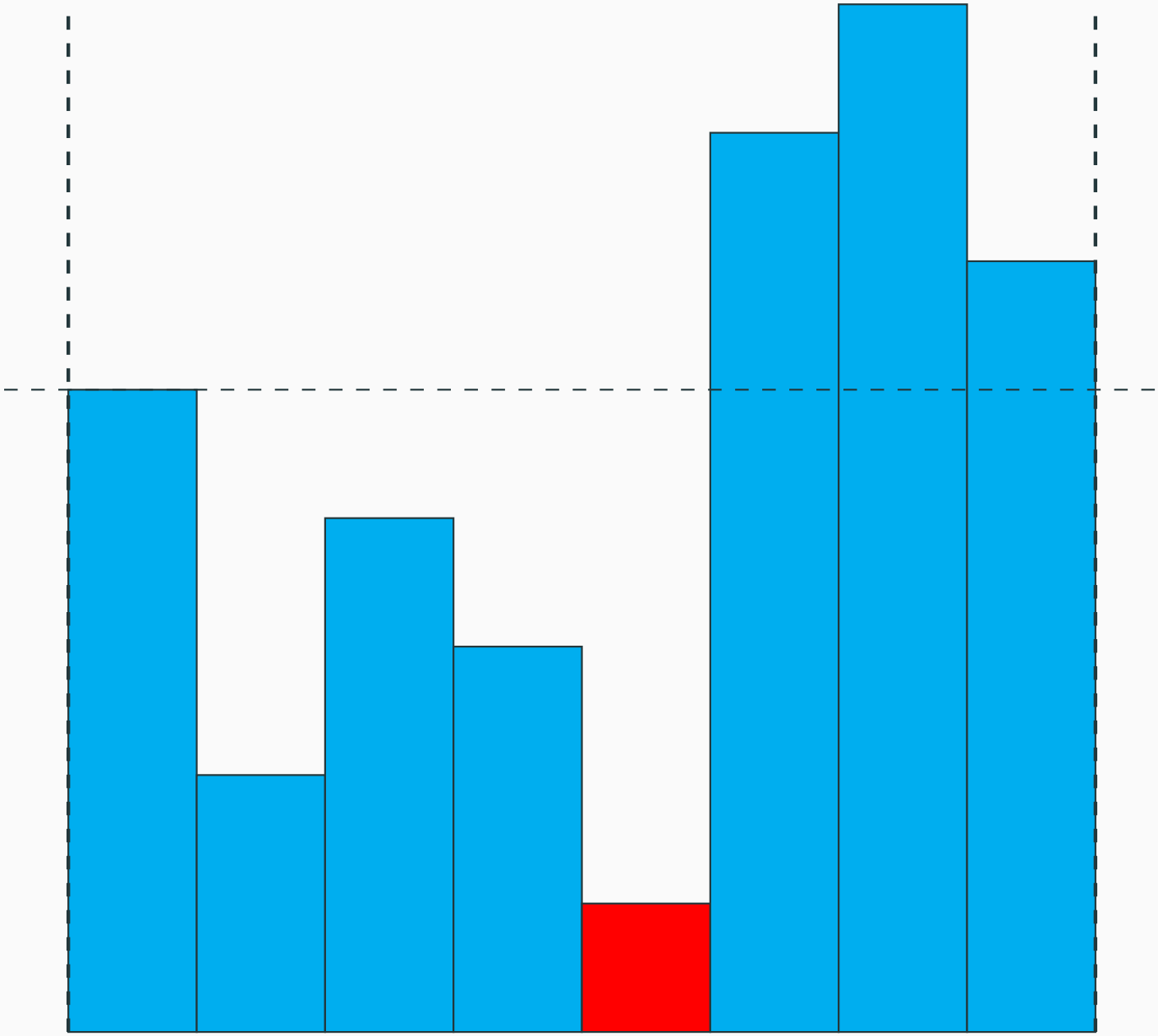
Exemple



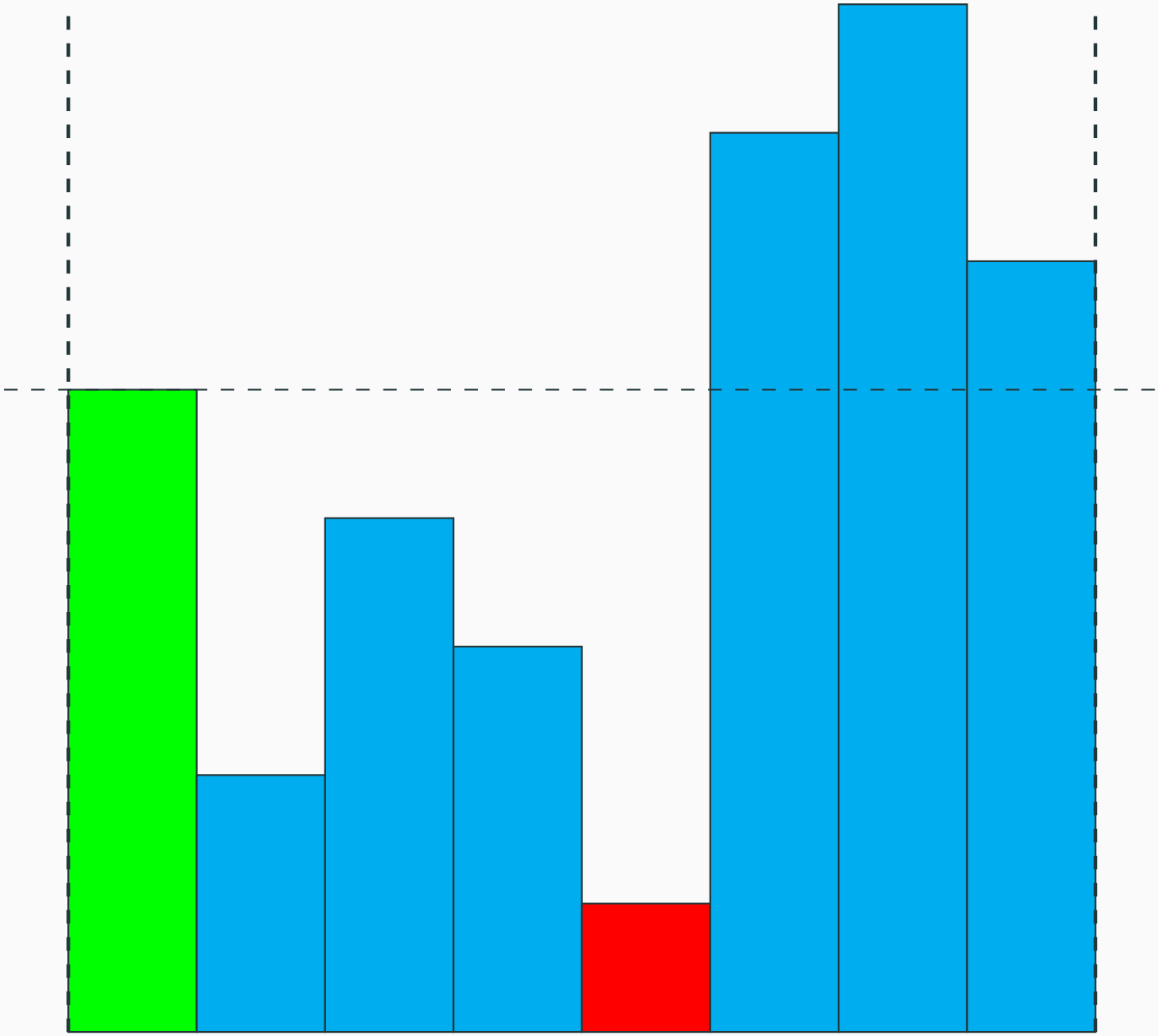
Exemple



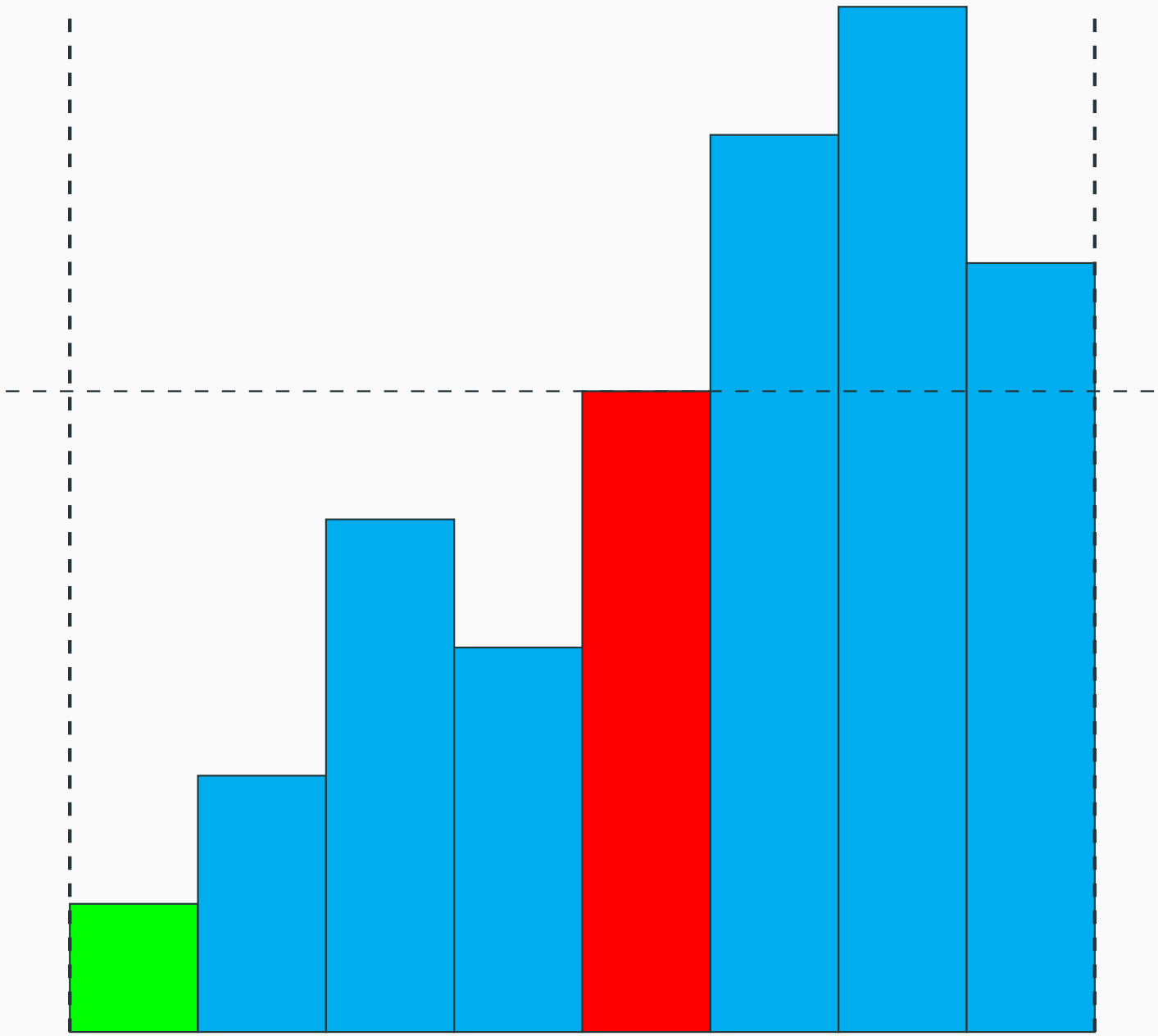
Exemple



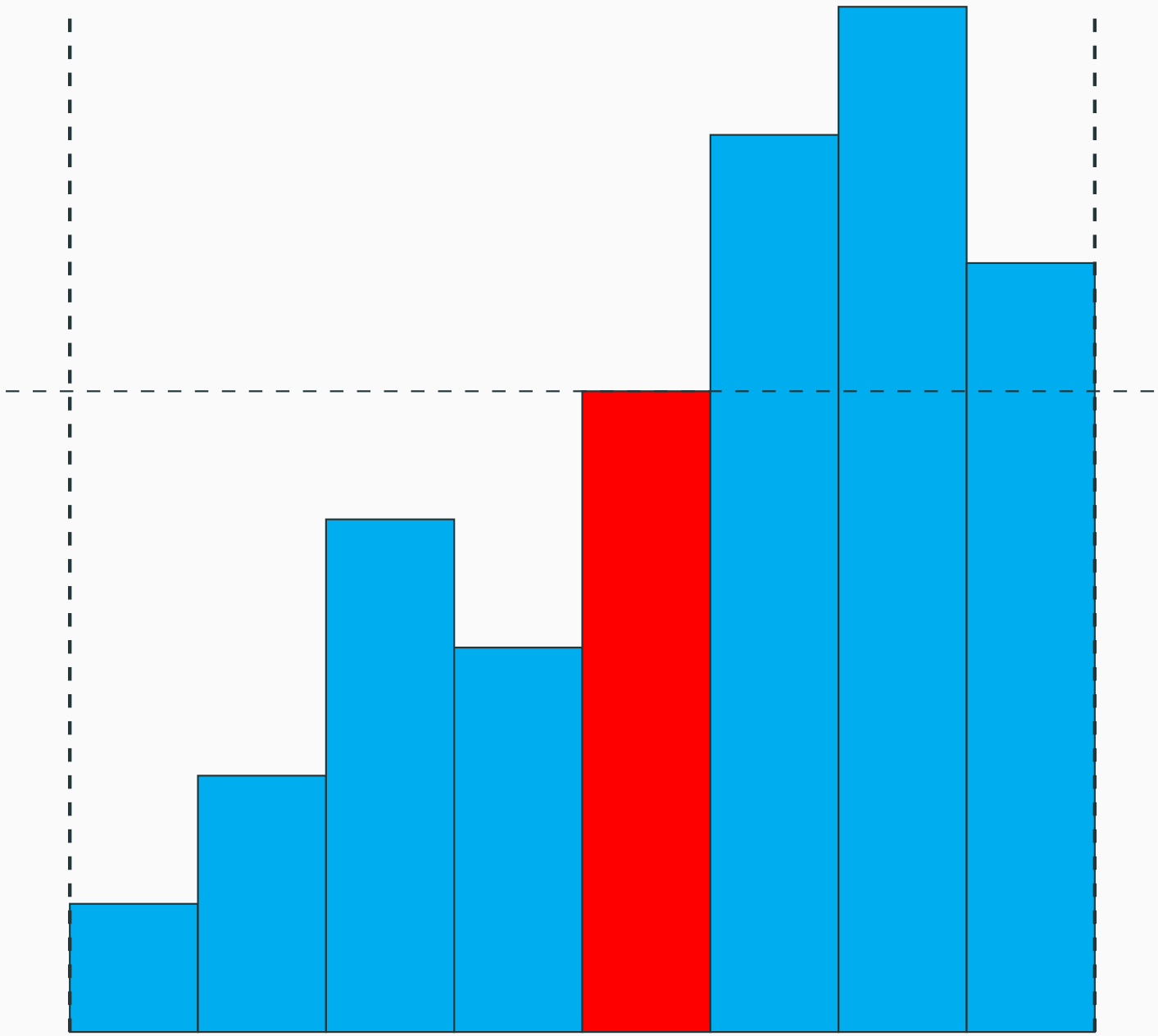
Exemple



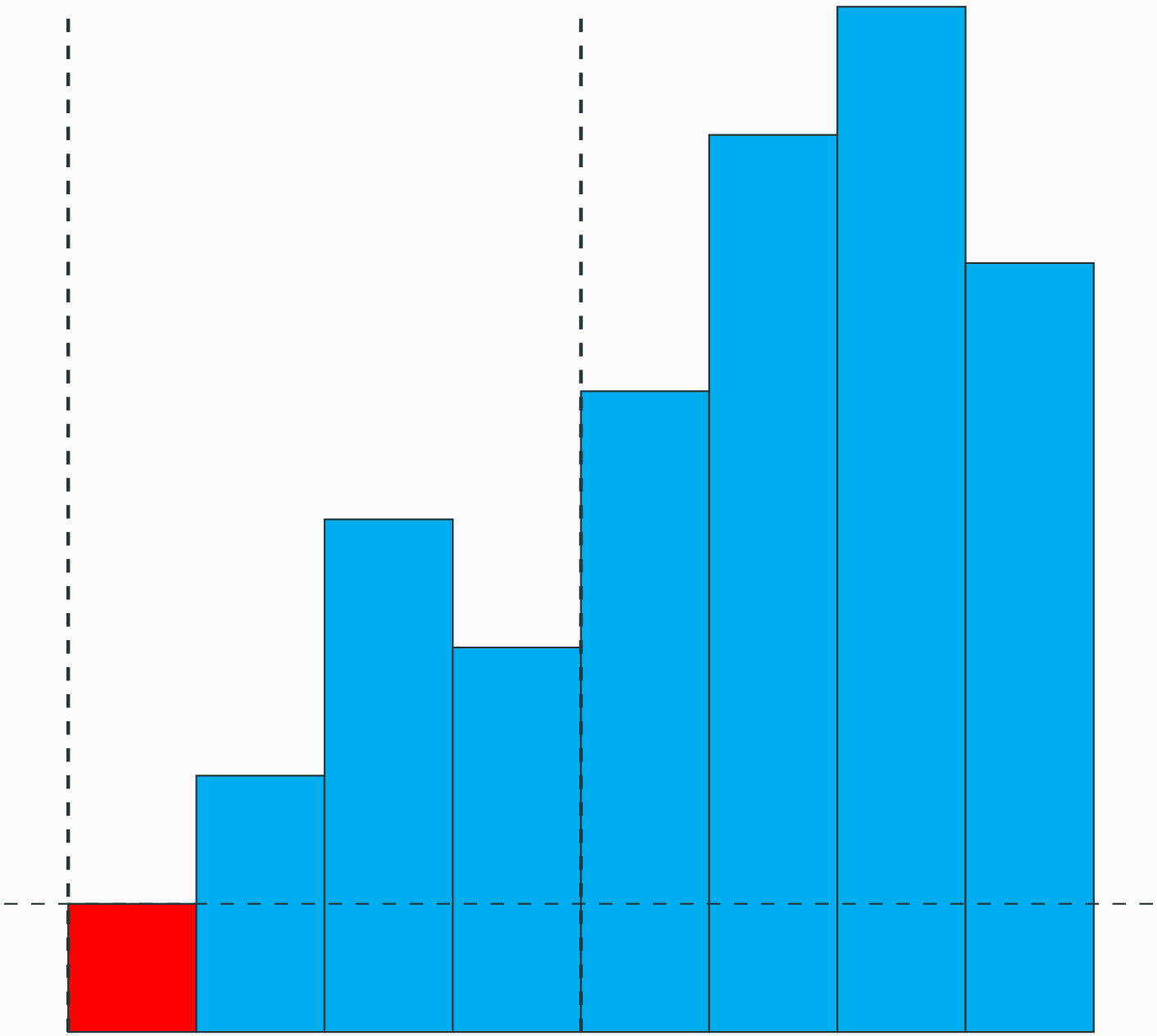
Exemple



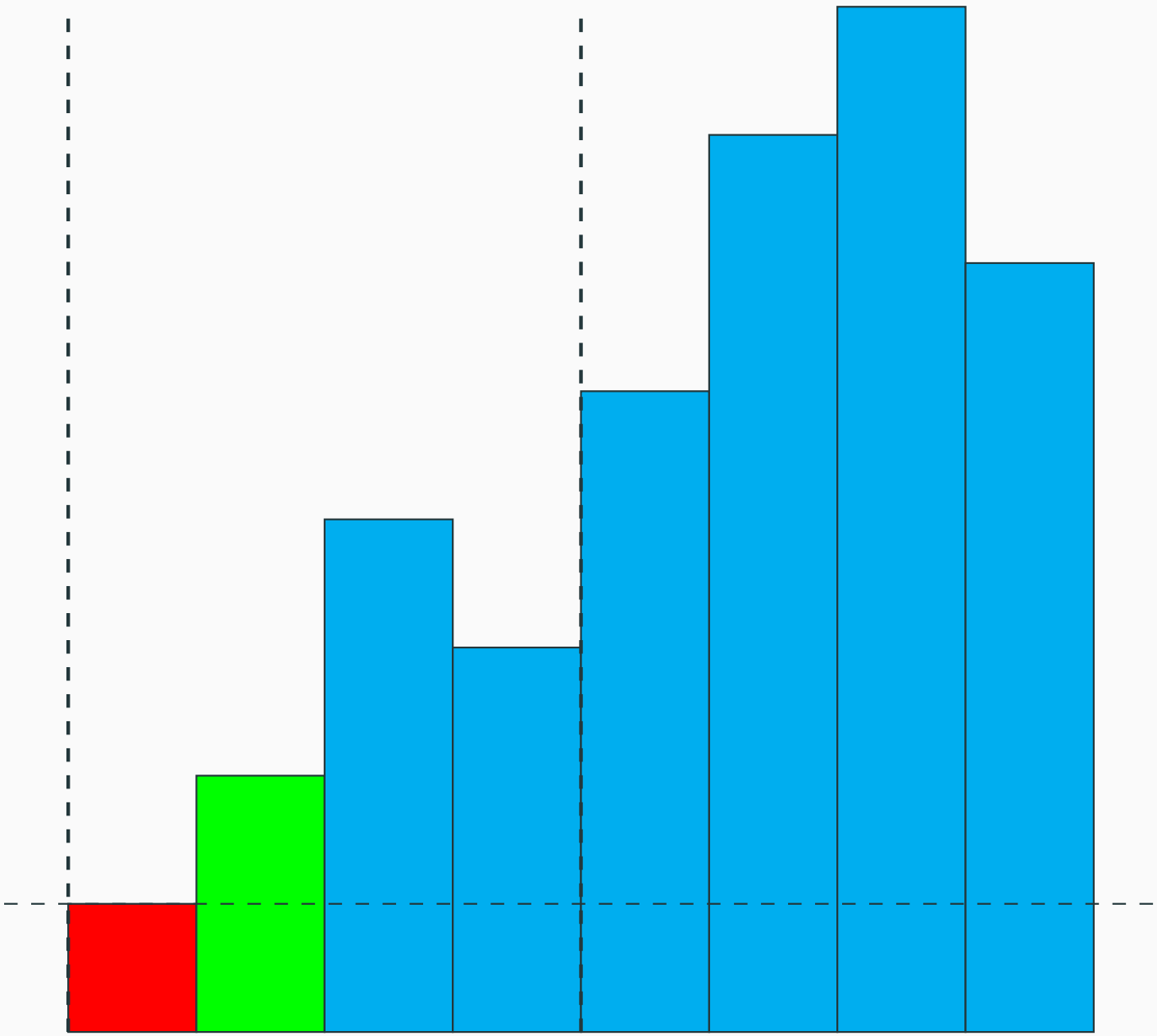
Exemple



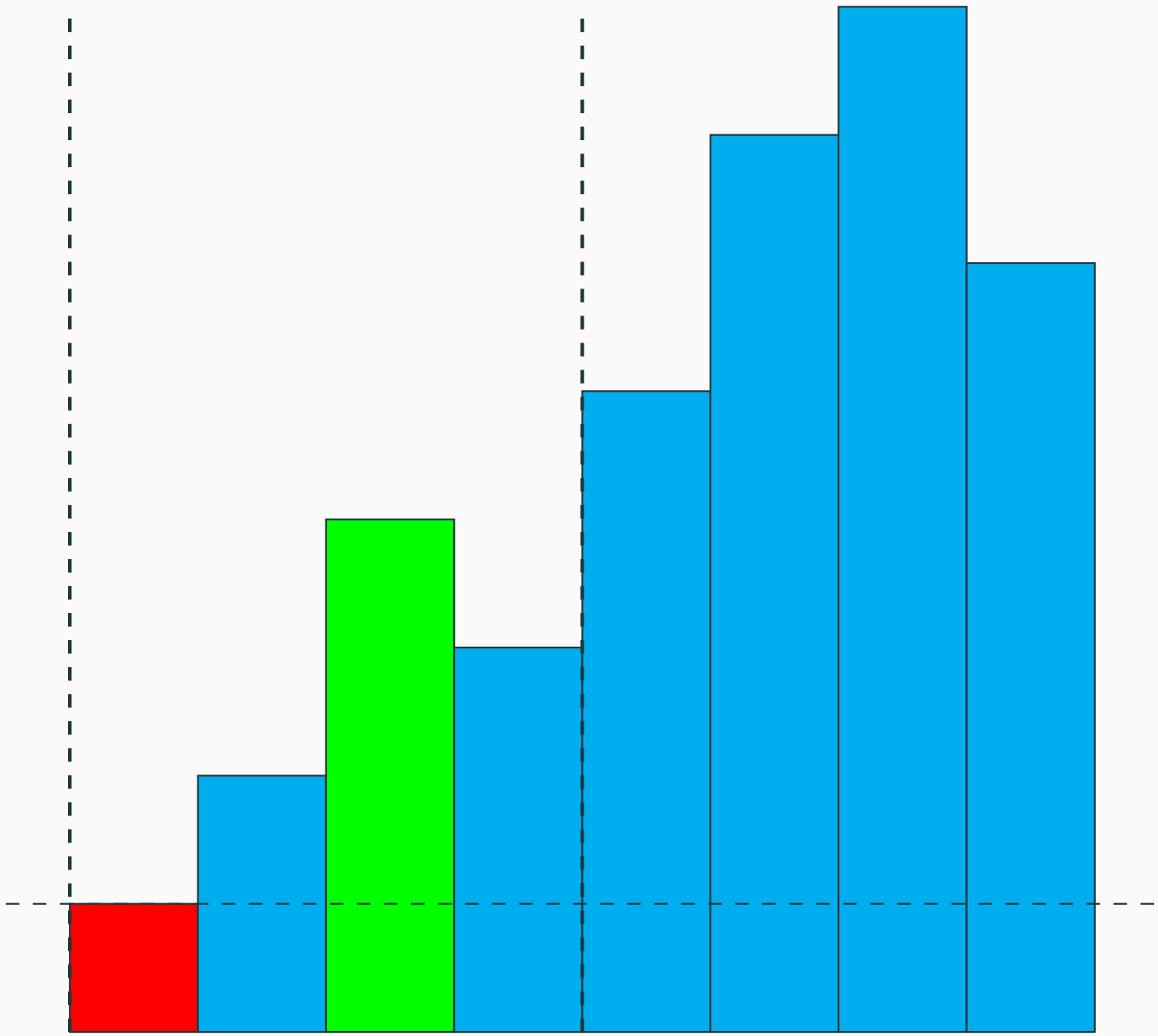
Exemple



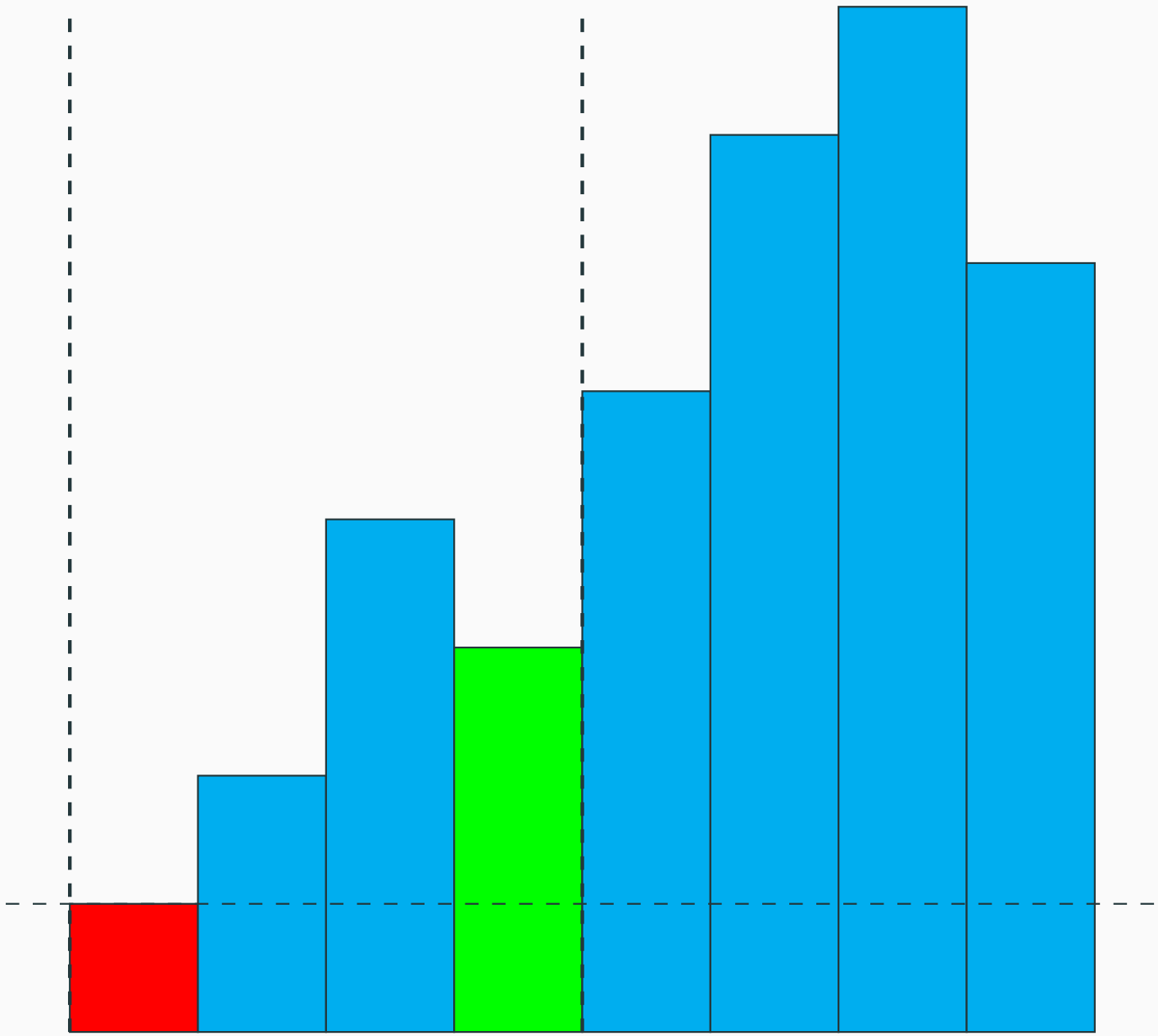
Exemple



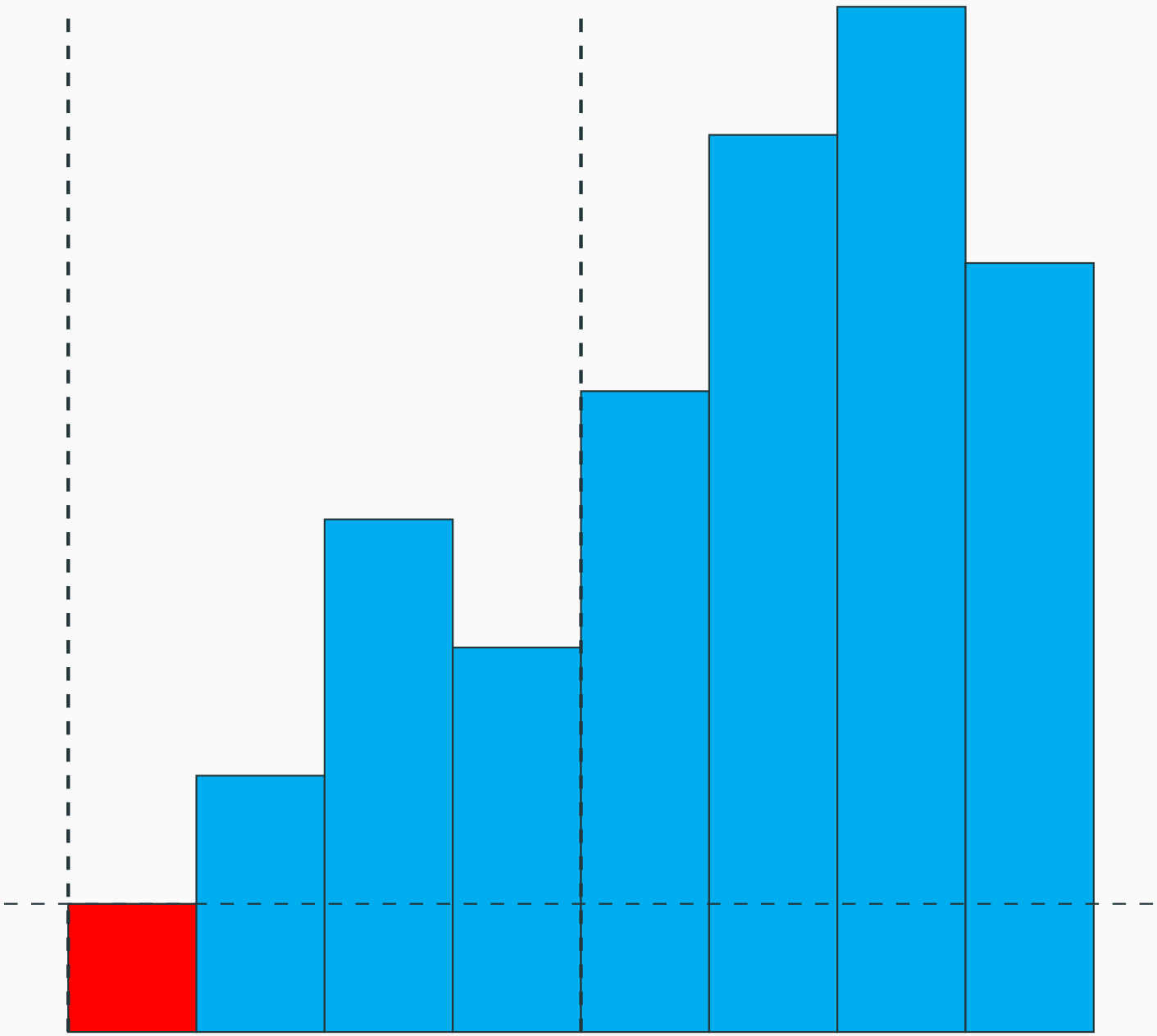
Exemple



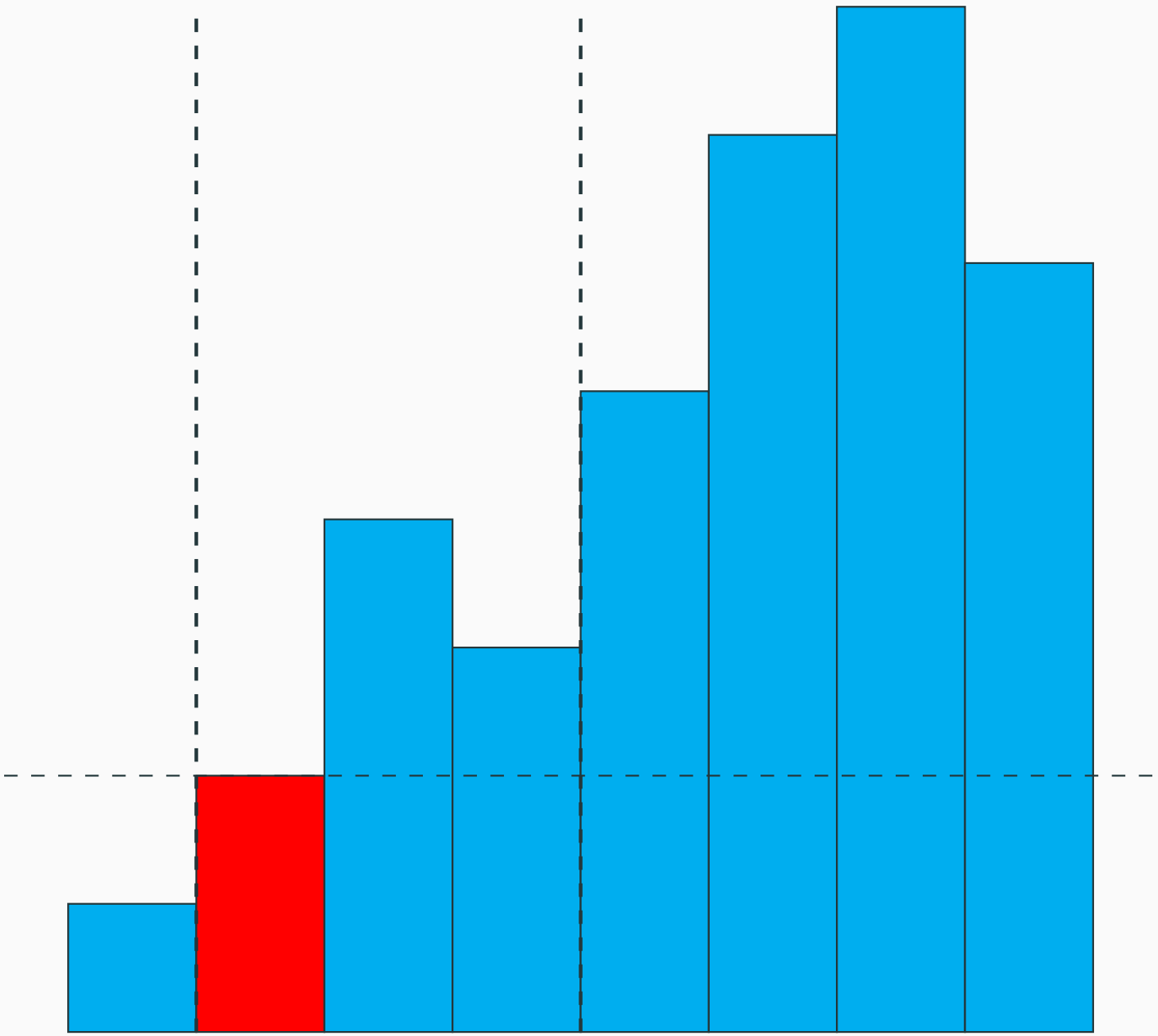
Exemple



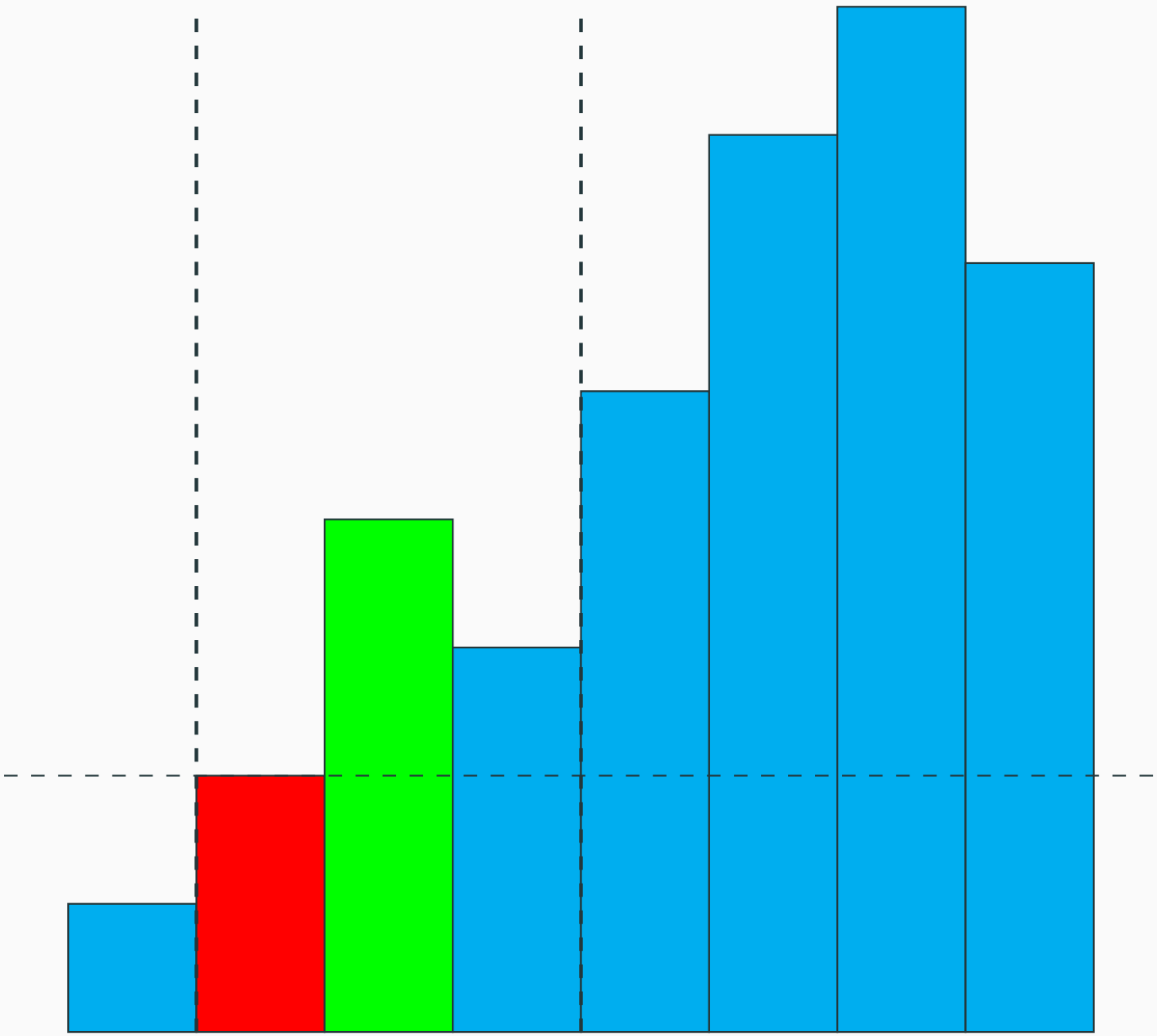
Exemple



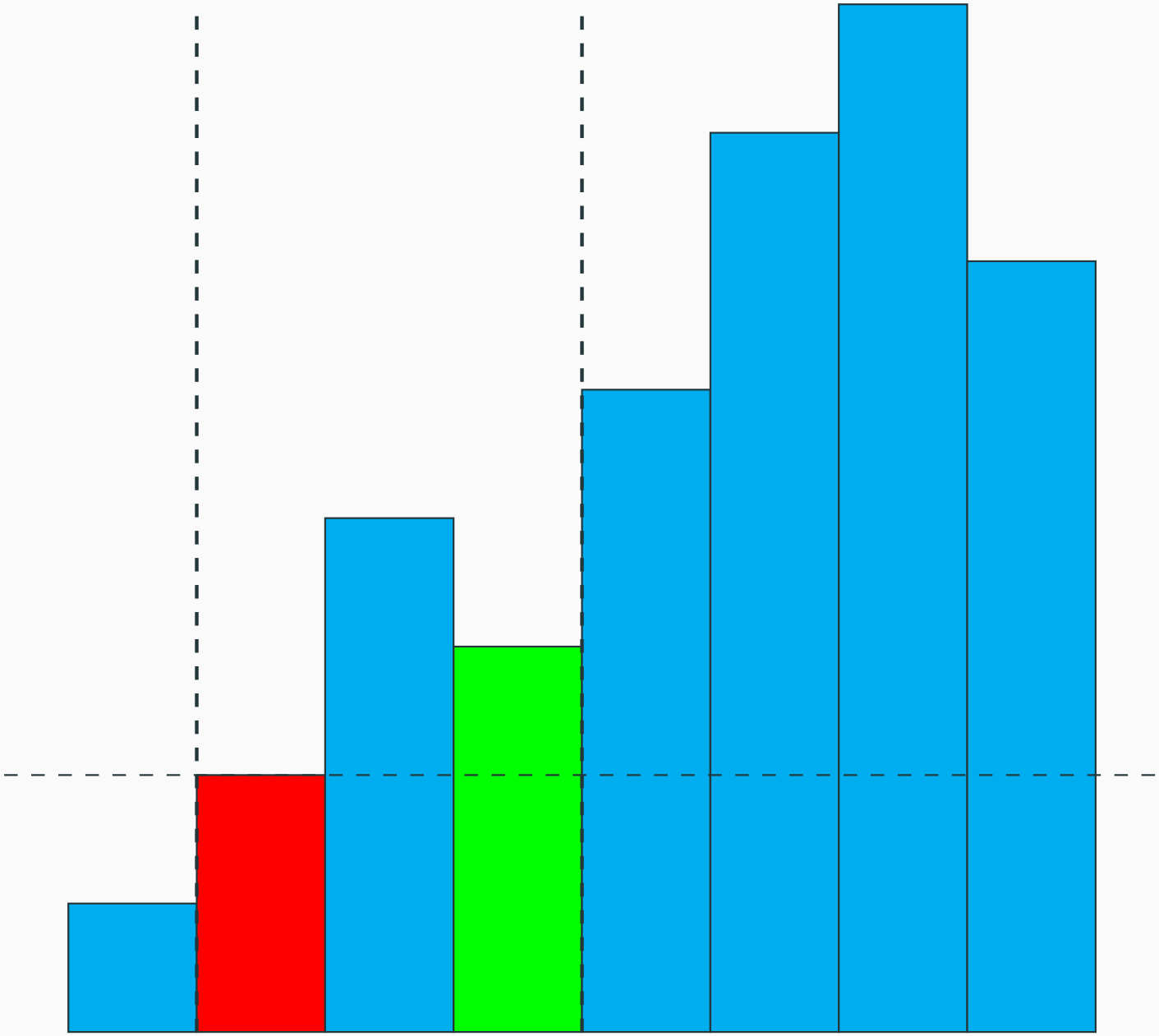
Exemple



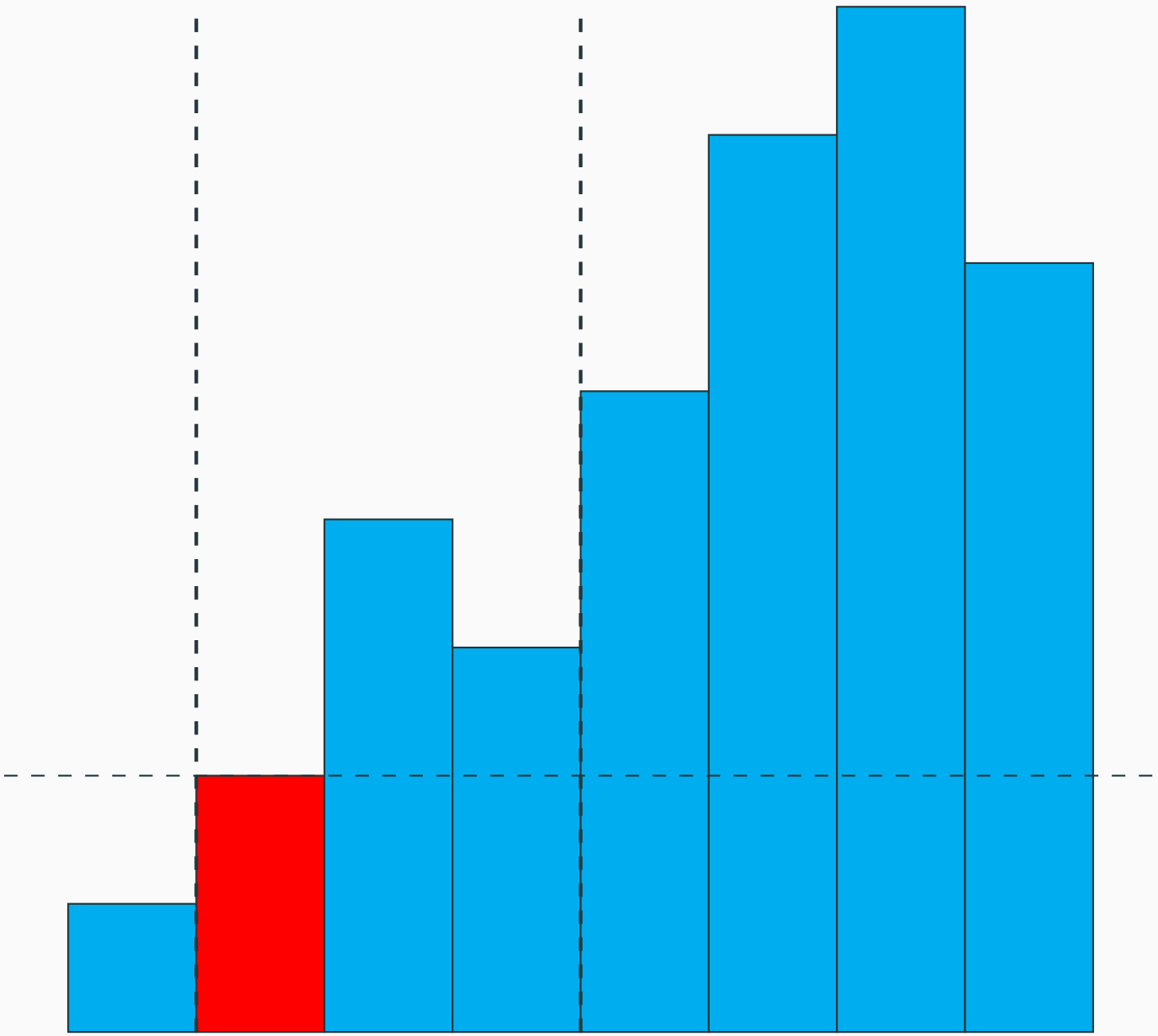
Exemple



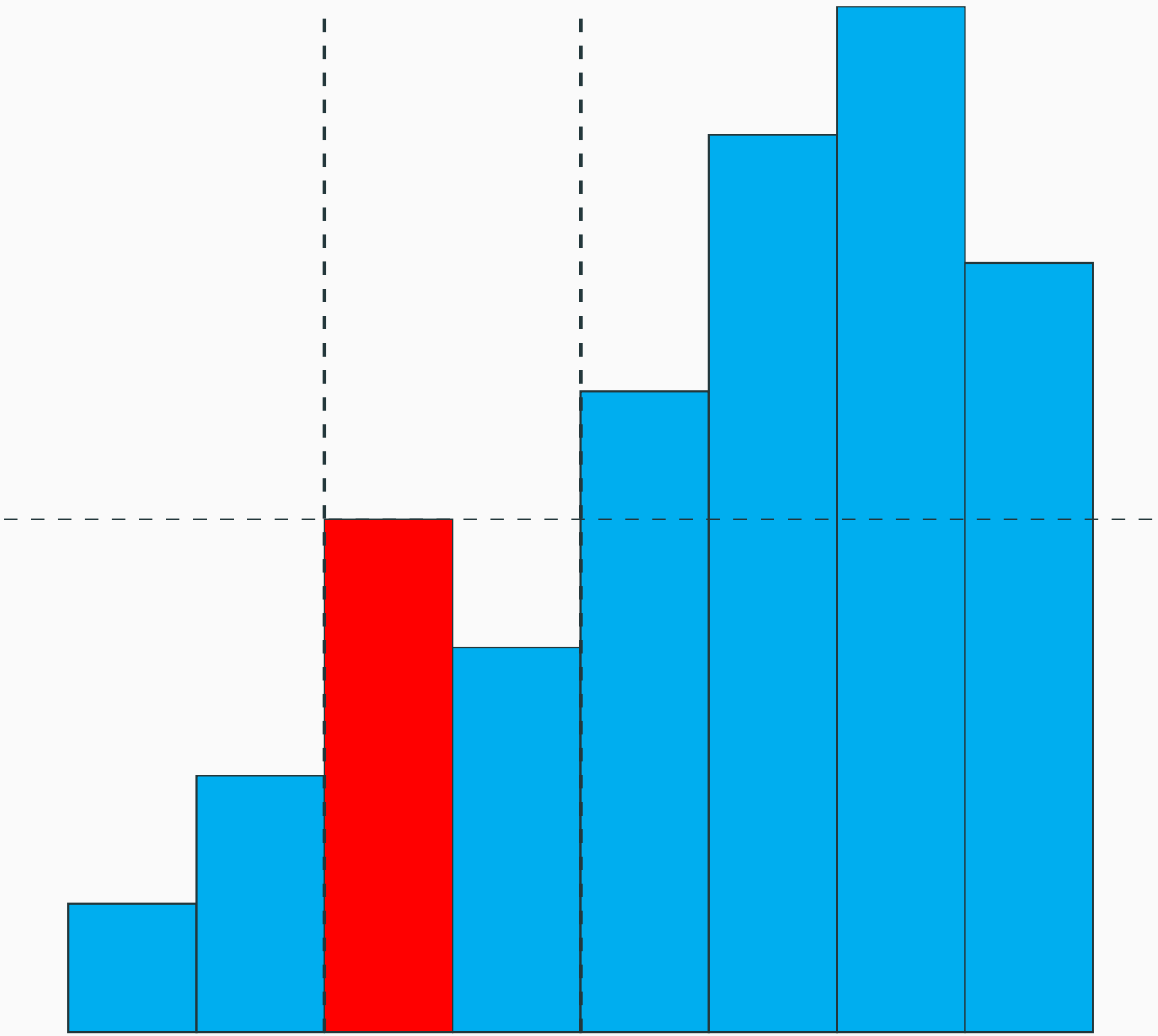
Exemple



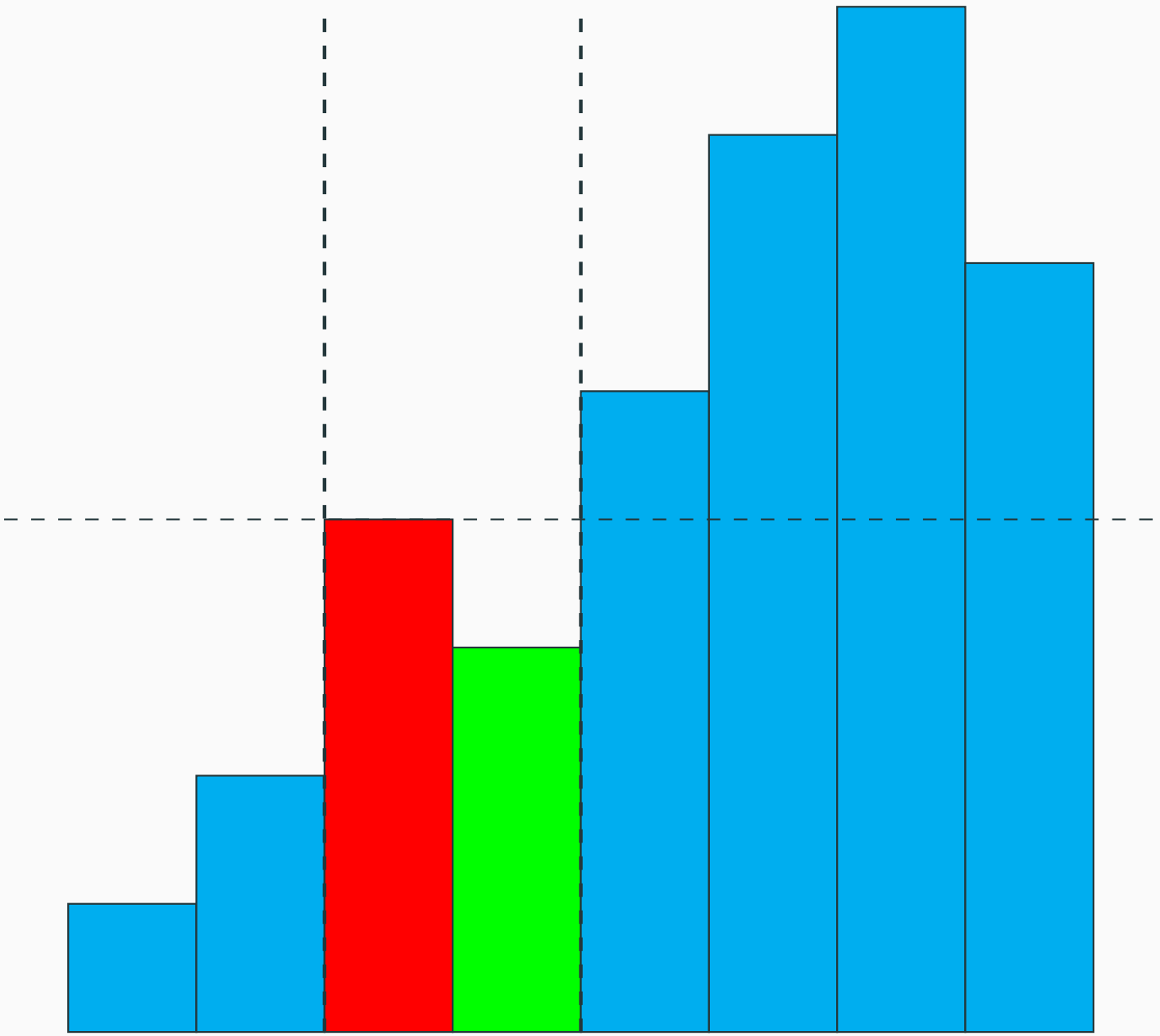
Exemple



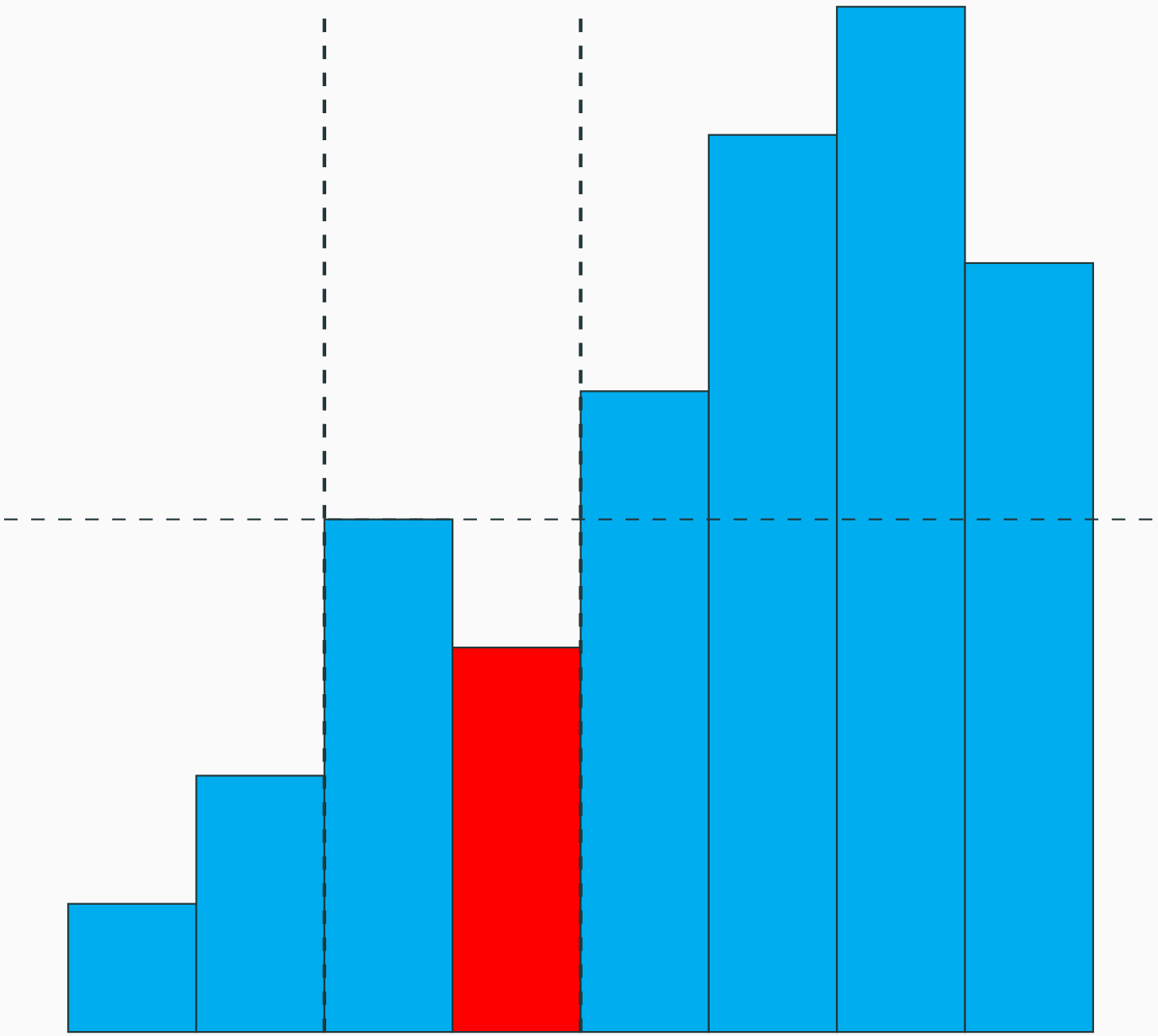
Exemple



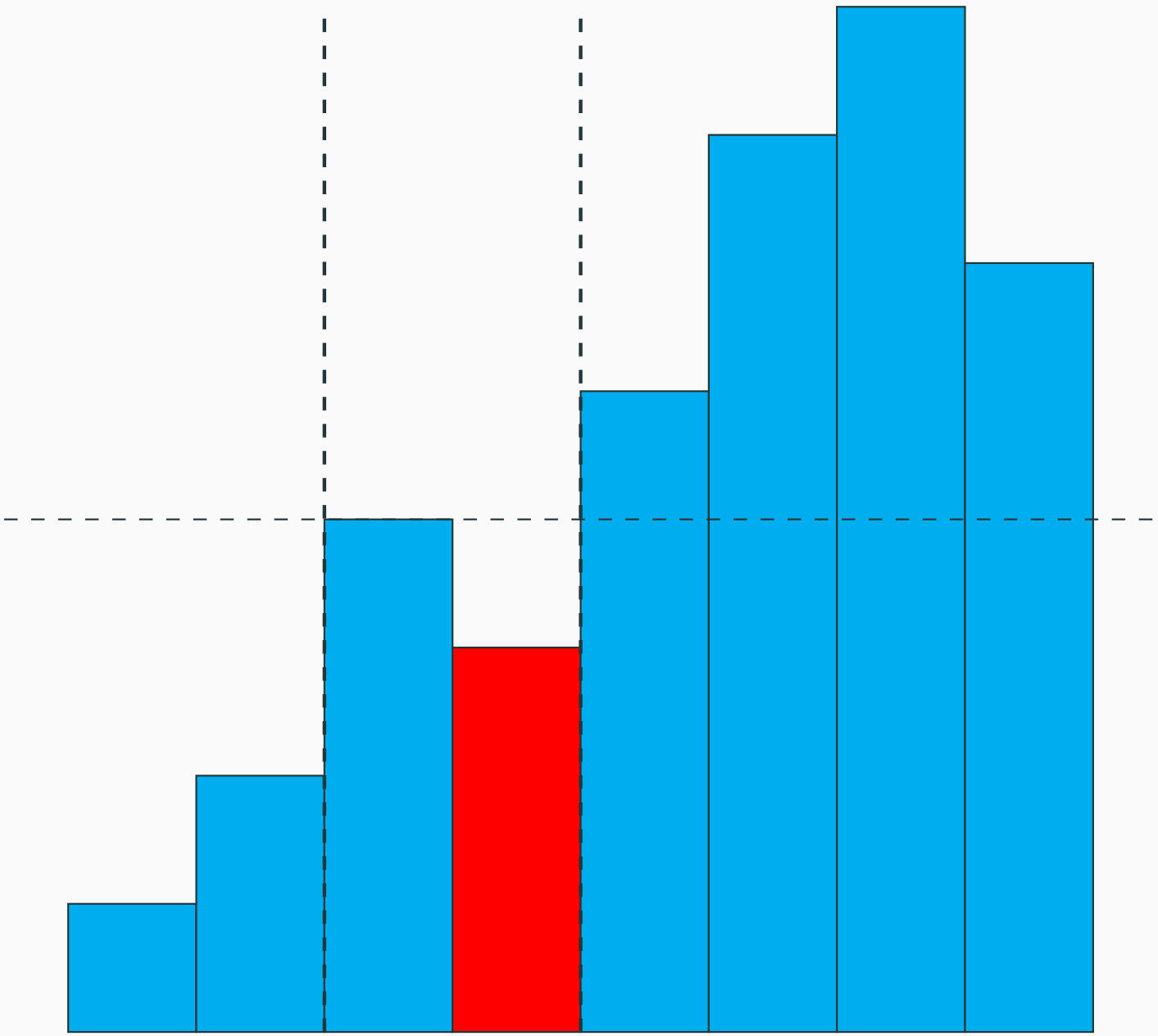
Exemple



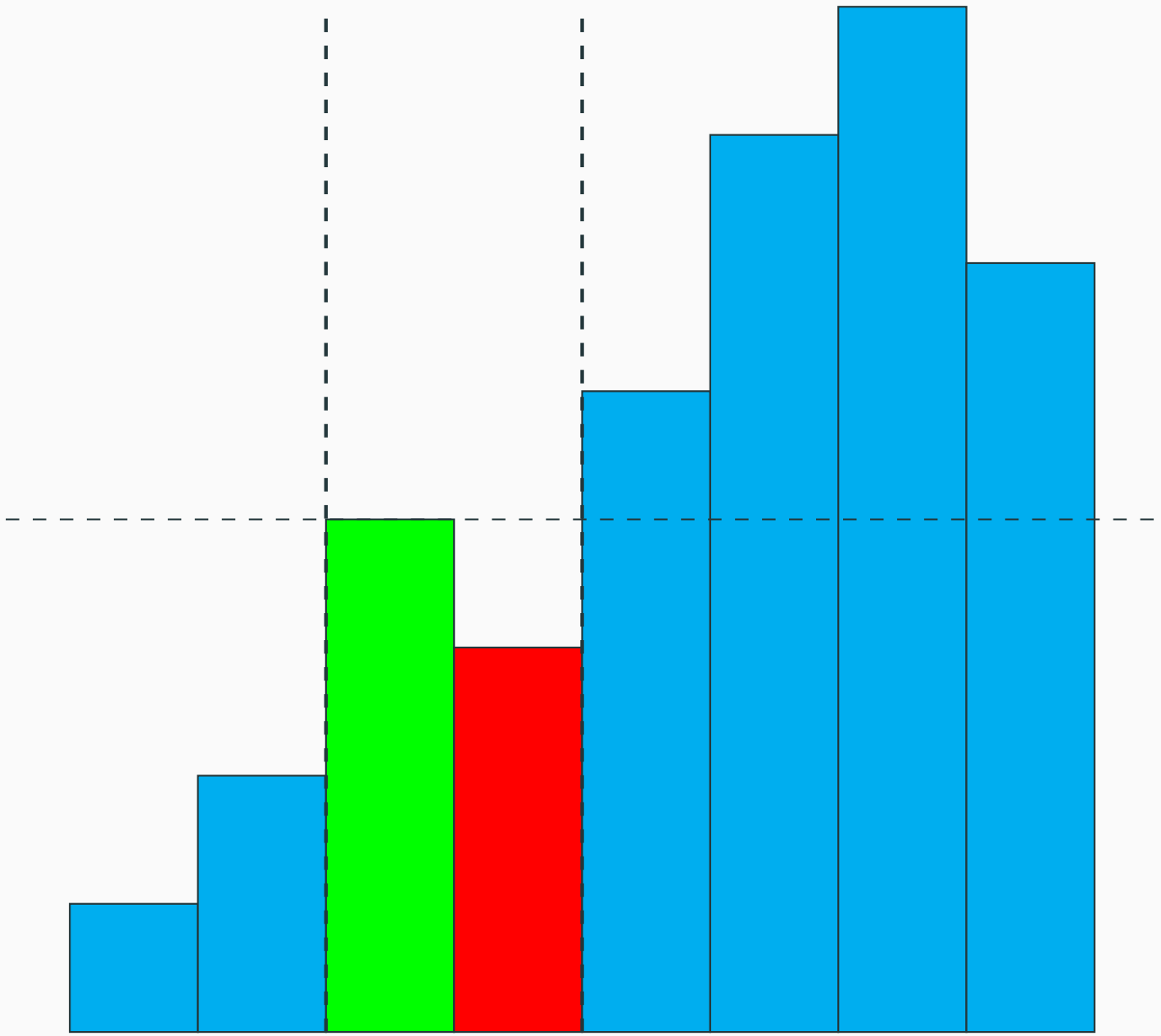
Exemple



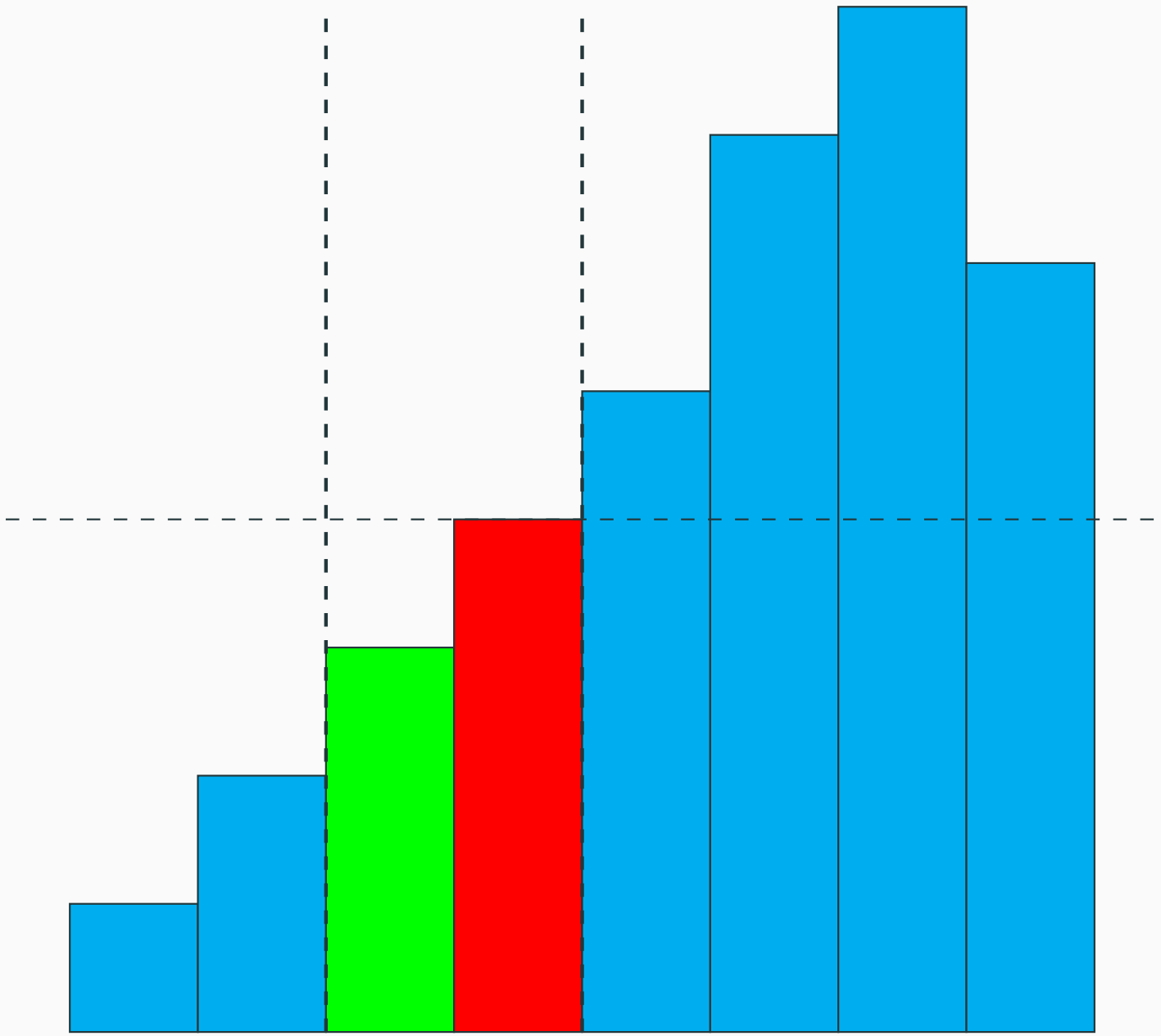
Exemple



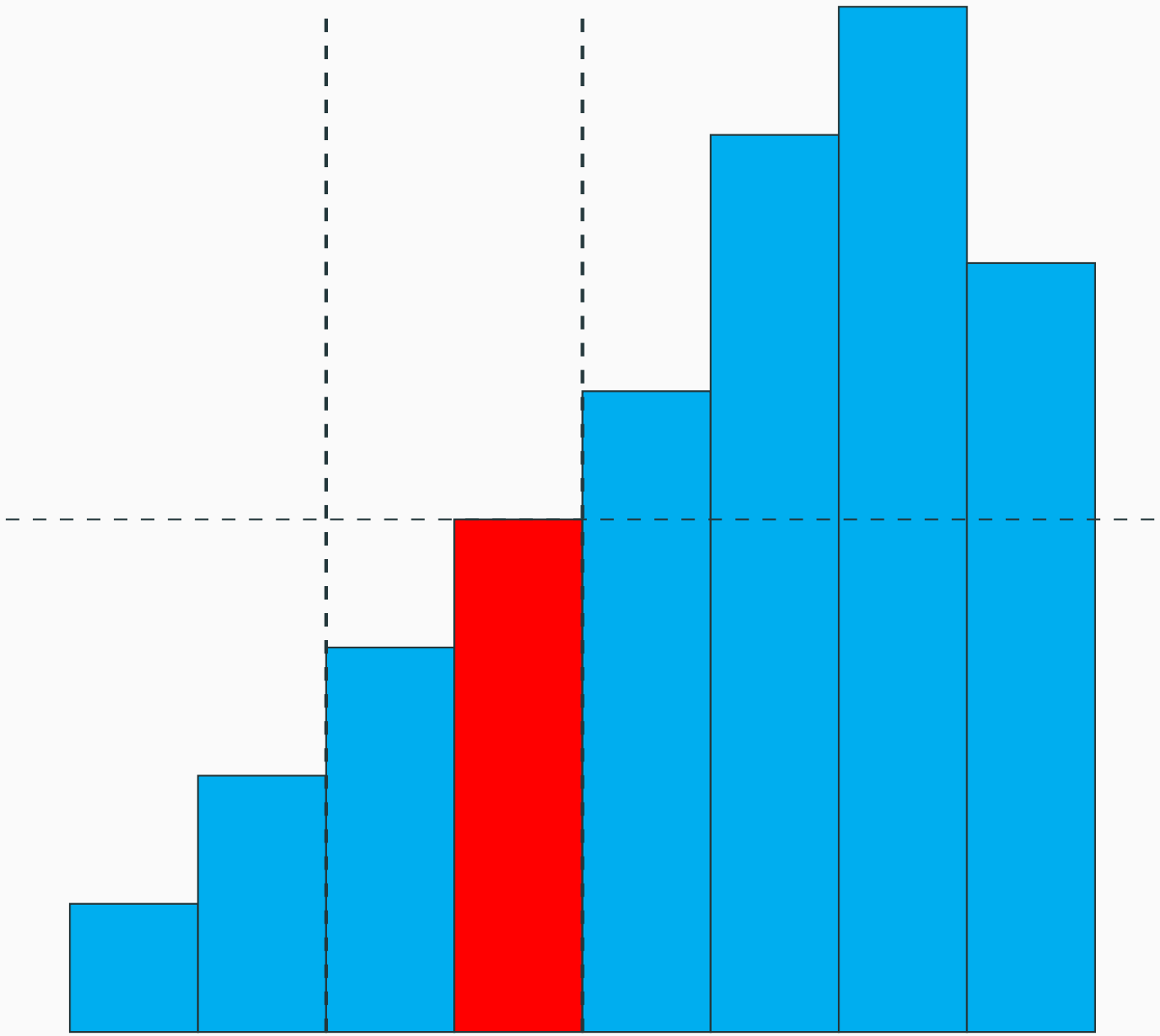
Exemple



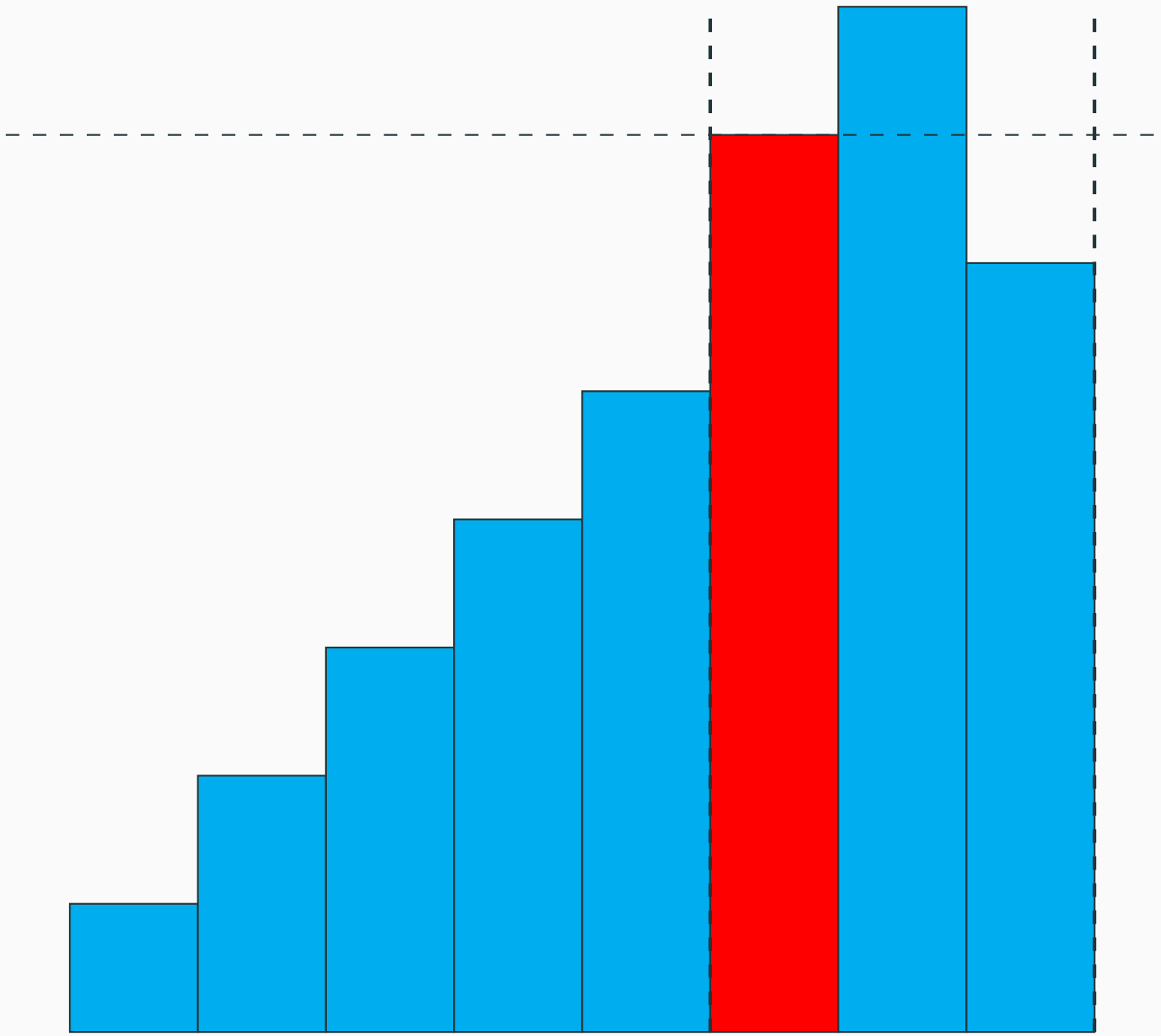
Exemple



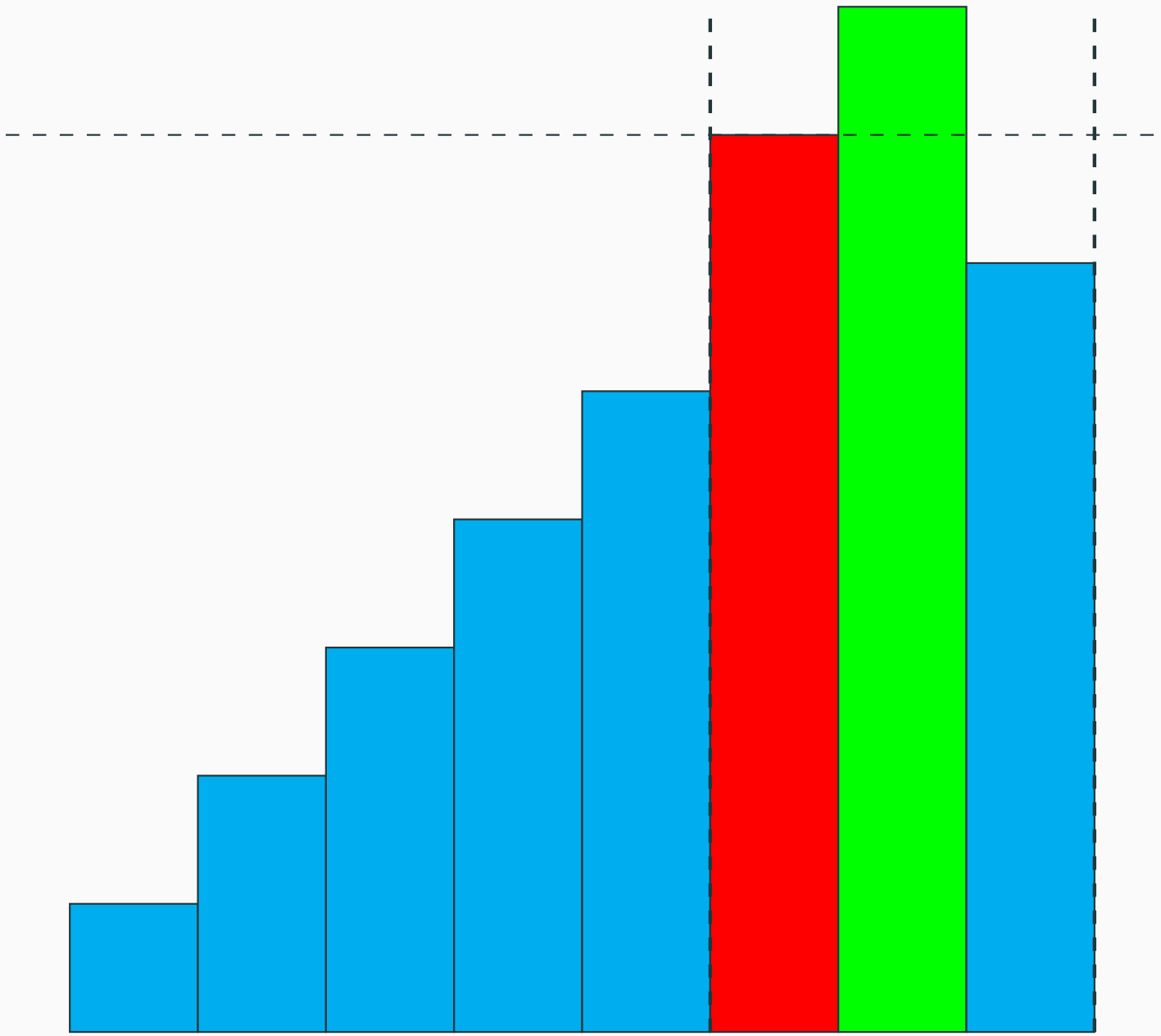
Exemple



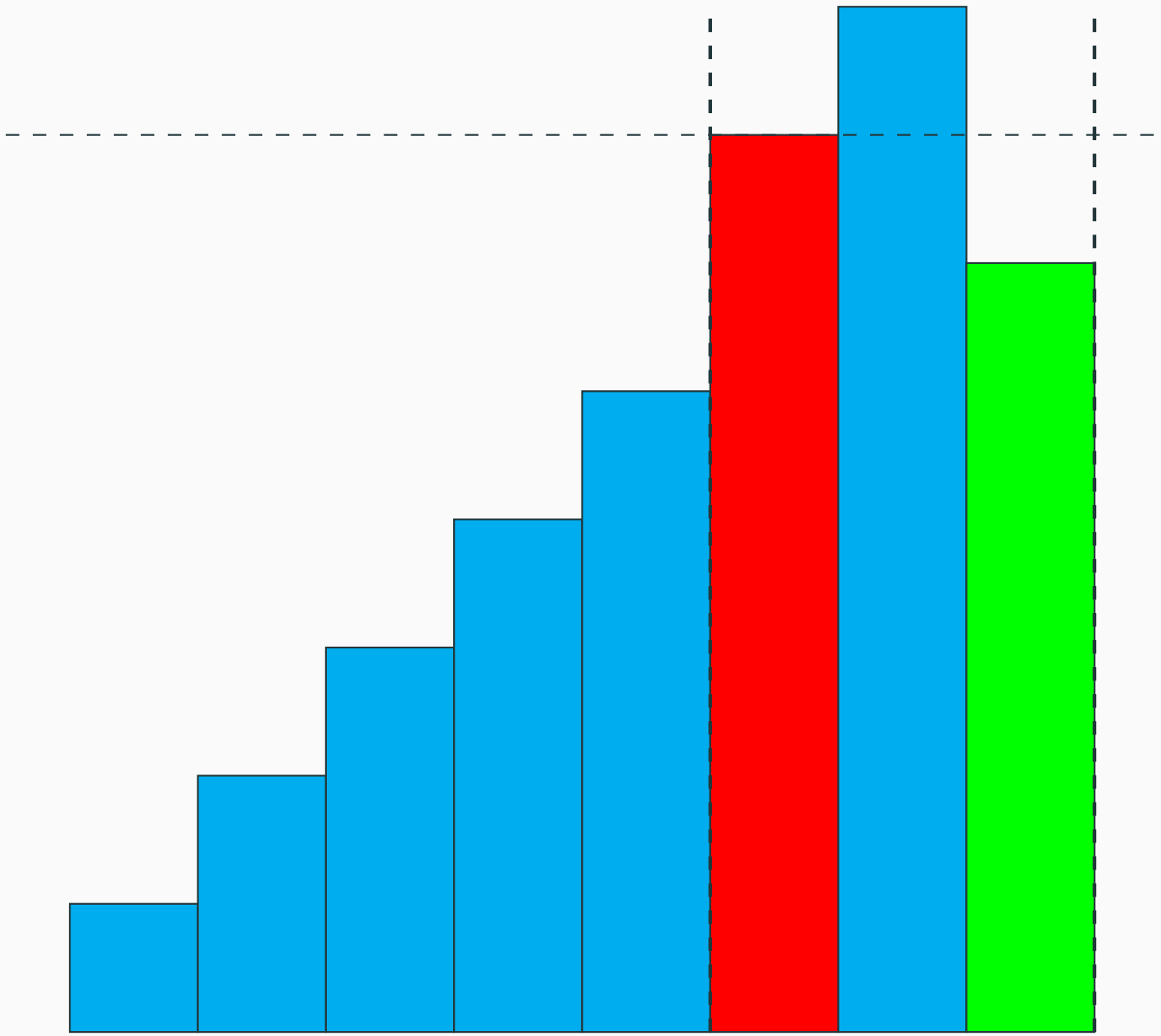
Exemple



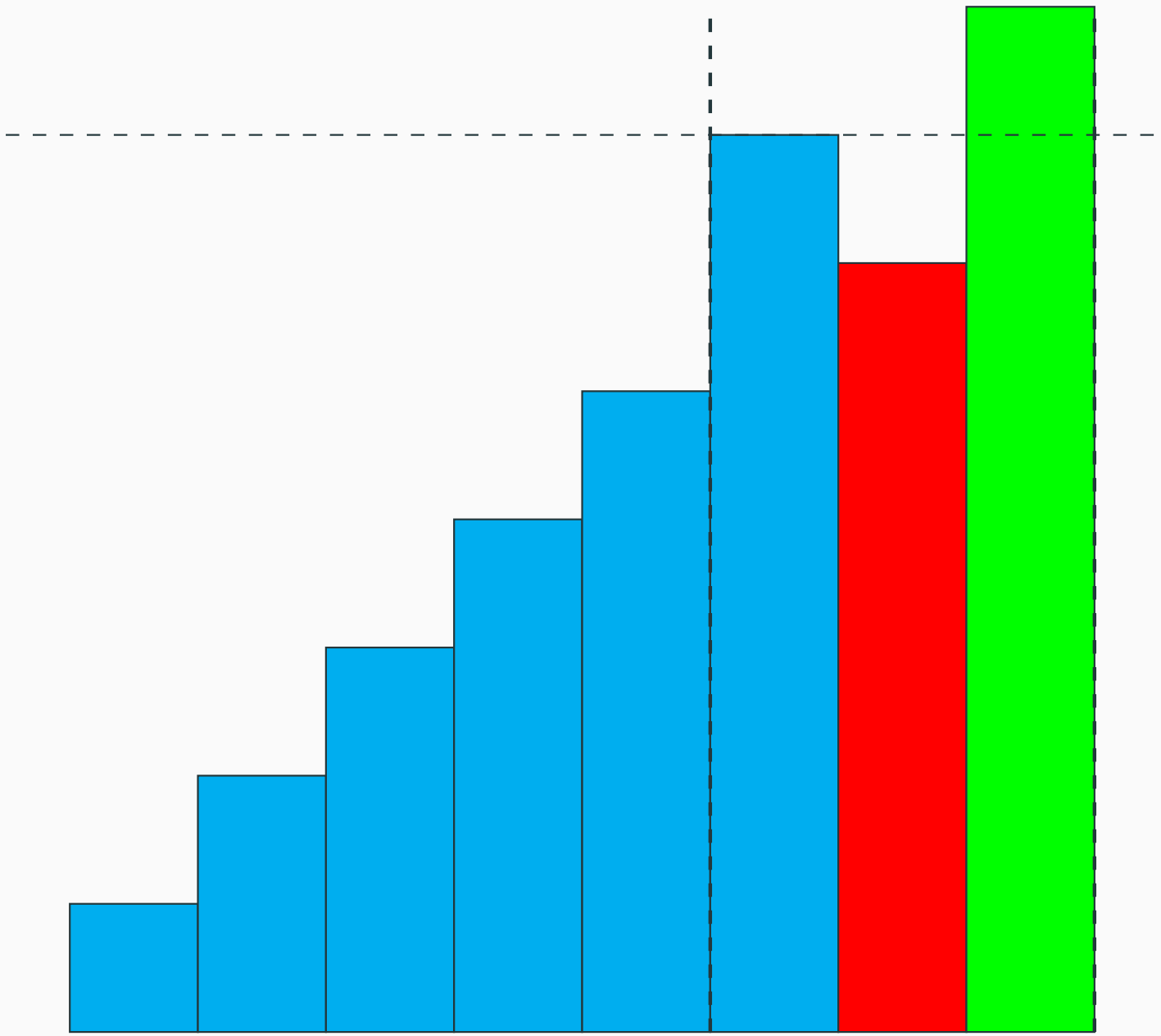
Exemple



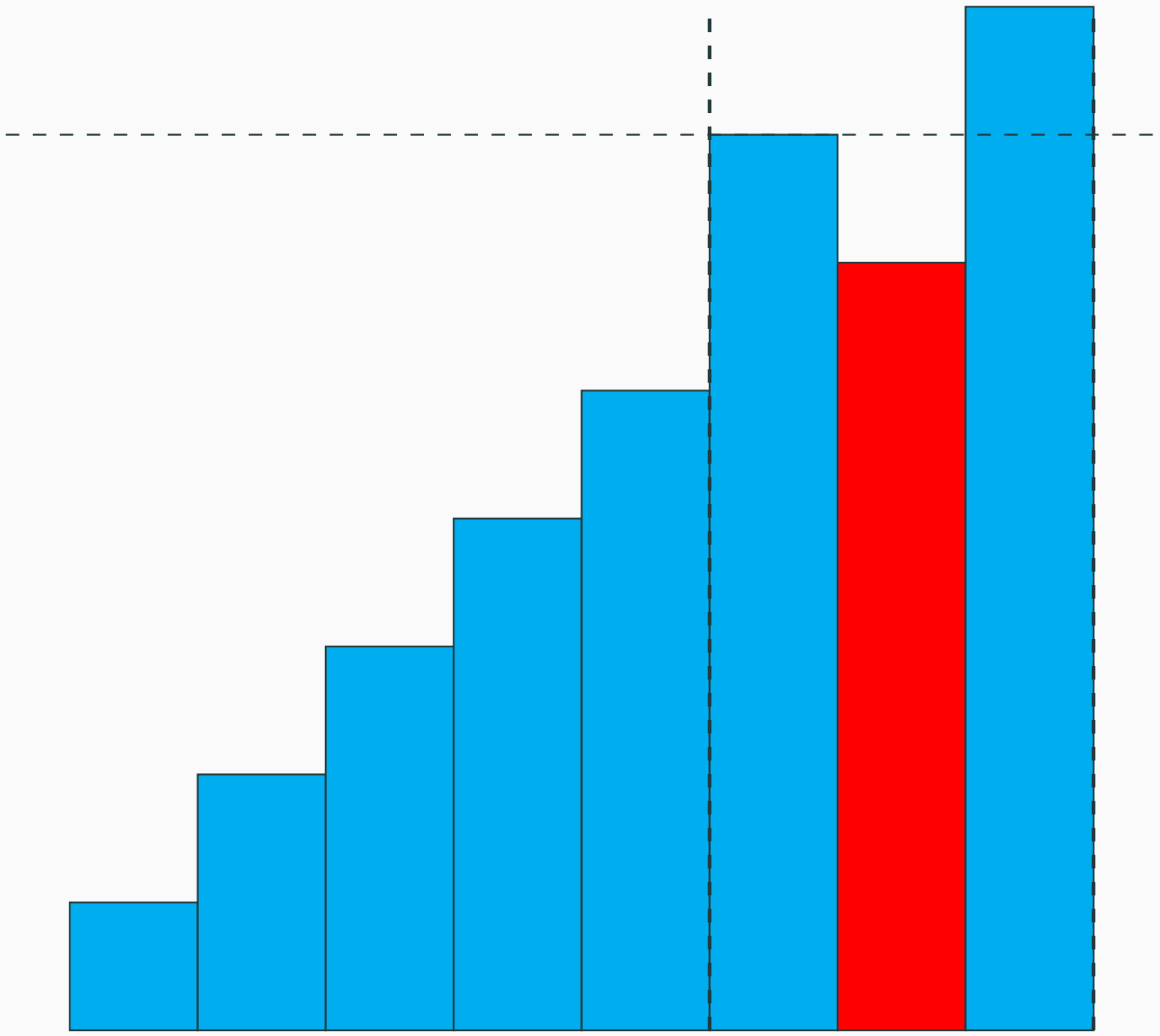
Exemple



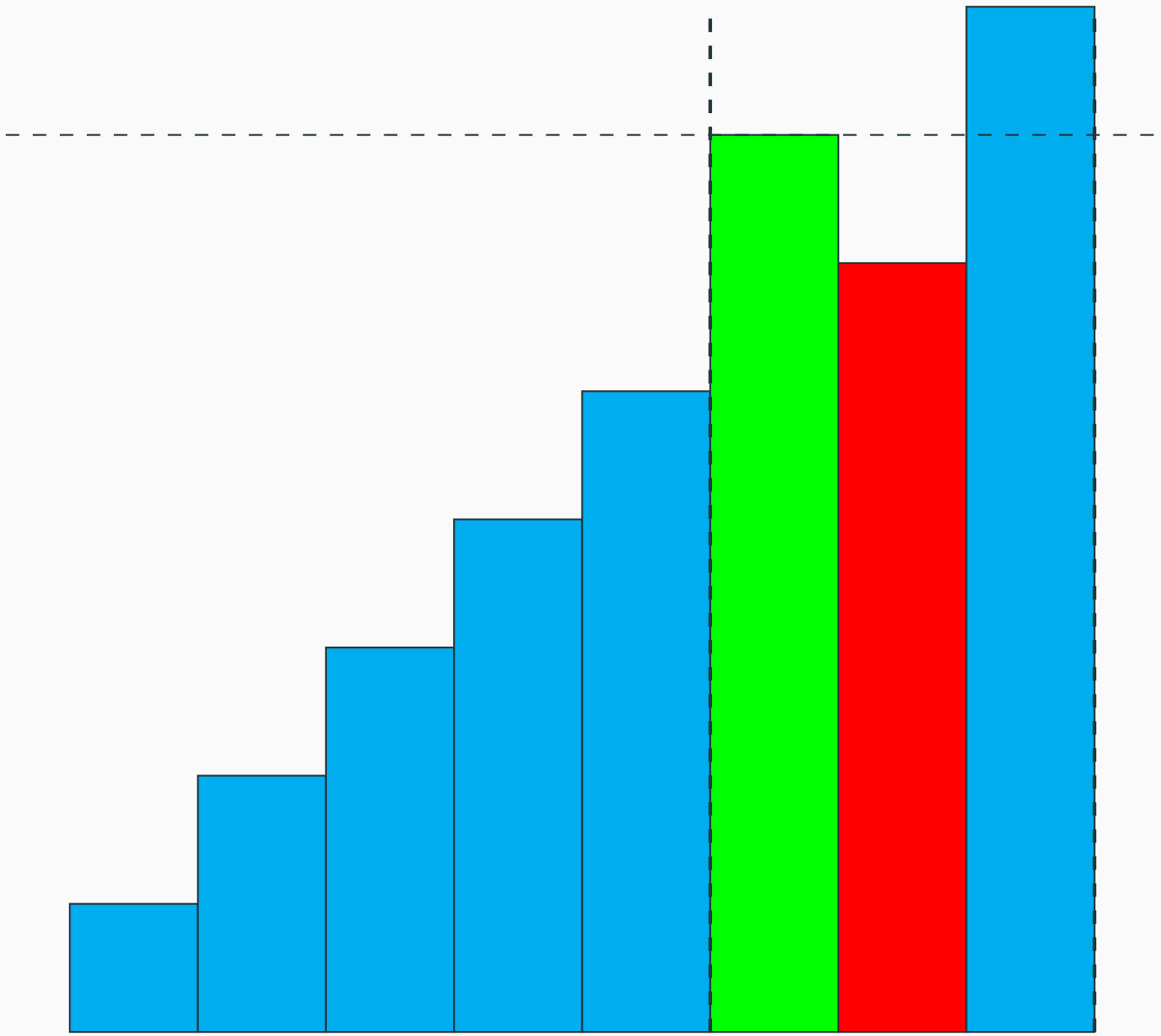
Exemple



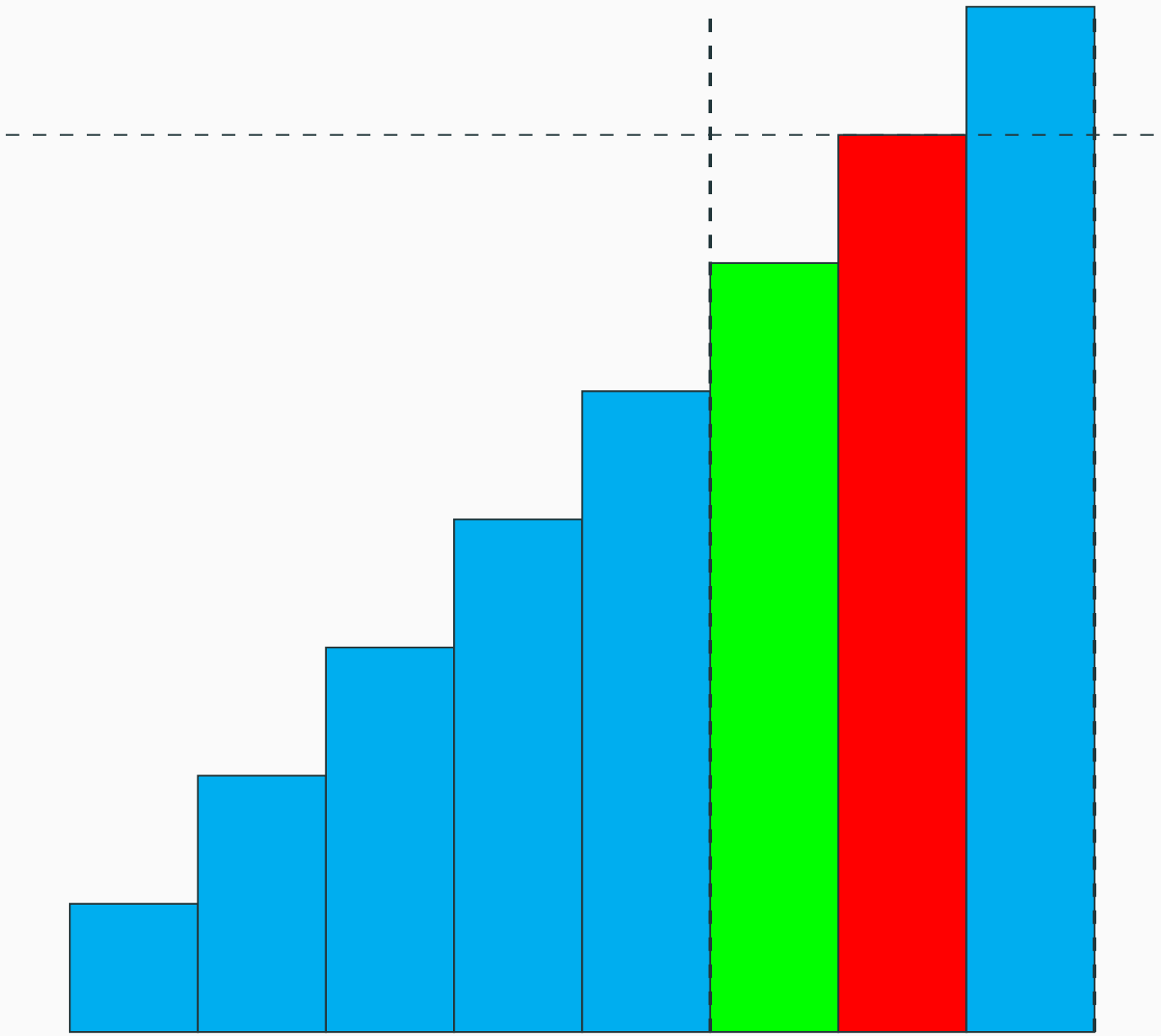
Exemple



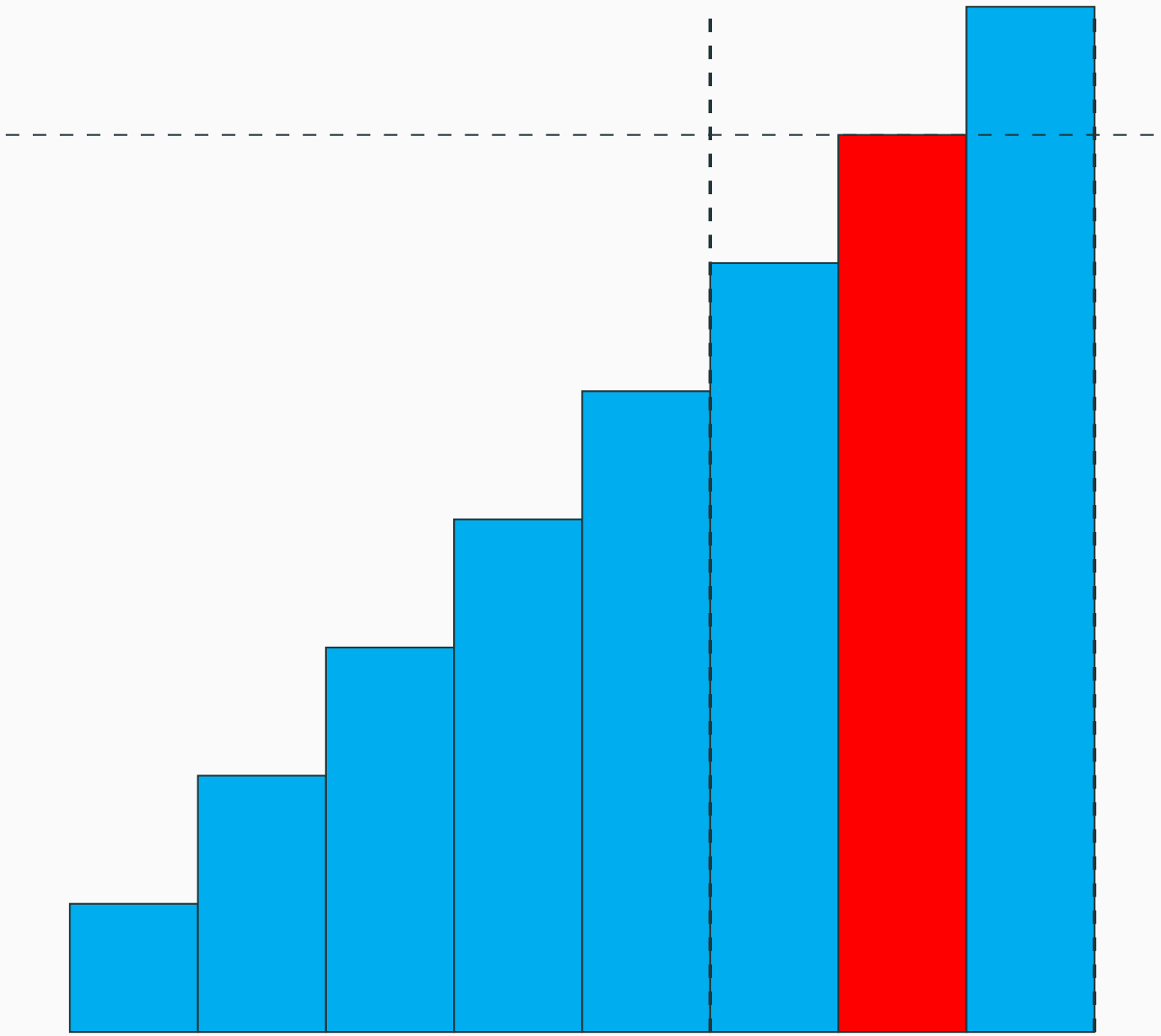
Exemple



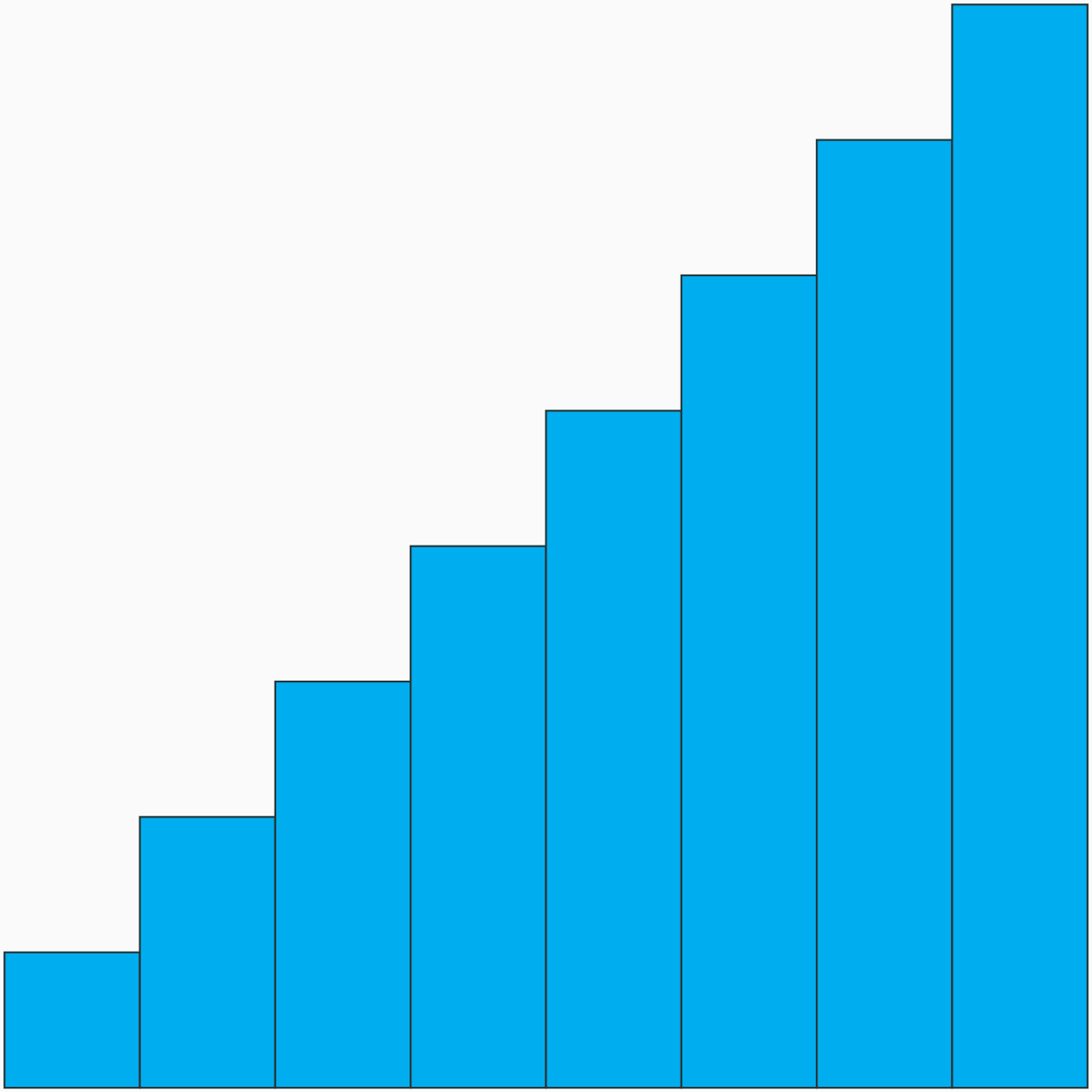
Exemple



Exemple



Exemple



Terminaison et correction

Terminaison

les appels récursifs de la fonction aux `g d` se font toujours sur des valeurs de $d - g$ strictement décroissantes, et la fonction `partition` termine, donc l'algorithme termine.

Correction

Comme la fonction `partition` est correcte, on montre facilement par récurrence que aux `g d` trie le tableau `t[g:d]`.

Ainsi, `tri_rapide t` trie bien le tableau `t`.

Complexité

Pire cas

Supposons que le tableau soit déjà triée dans l'ordre croissant.

Alors, le premier appel à `partition` réalise une très mauvaise partition : le tableau est laissée dans le même état, et le pivot renvoyé est celui en position 0.

Le premier appel récursif est alors aux `0 0` qui ne fait rien, et le second est aux `1 (Array.length t)` qui va trier `t[1:]` en suivant le même procédé.

Dans ce cas, on effectue au total un nombre quadratique de comparaisons.

Complexité

Meilleur cas

En revanche, si chaque appel à `partition` sépare les éléments en deux parties égales à un élément près, la complexité de l'algorithme vérifie la relation de récurrence suivante :

$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1$$

Ce qui, d'après le théorème maître, nous donne une complexité en $O(n \log(n))$.

Complexité en moyenne

Complexité en moyenne

Pour l'étude de la complexité en moyenne, on suppose que les éléments de L sont tous distincts et que leurs positions relatives sont équiprobables.

Ainsi, on suppose que L est constitué des éléments de $[\![0, n-1]\!]$, et on va compter le nombre de comparaisons effectuées en moyenne :

$$C_{\text{moy}}(n) = \frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} C([\sigma(0), \ldots, \sigma(n-1)])$$

Complexité en moyenne

Complexité en moyenne

On pourrait montrer que la complexité en moyenne du tri rapide est $\Theta(n \ln n)$.

C'est l'un des tris les plus rapides en pratique, d'où son nom.

C'est le tri utilisé par Python pour trier une liste `L` lorsqu'on fait `L.sort()`.

Propriétés et optimisations

Propriétés

L'avantage du tri rapide sur le tri fusion est qu'il s'effectue **en place**.

Par contre, il n'est **pas stable**.

Propriétés et optimisations

```
1  let r = g + Random.int (d-g) in (* entier au hasard entre g et d-1 *)
2  if r > g then
3    begin
4      let x = t.(g) in
5      t.(g) <- t.(r)
6      t.(r) <- x
7    end
```

Optimisation

Un tableau déjà triée est un pire cas pour le tri rapide.

Un moyen d'y remédier est de modifier `partition` pour que le pivot ne soit pas systématiquement `t.(g)`, mais un élément de `t[g:d]` pris au hasard.

On peut ainsi rajouter les lignes ci-dessus au début de l'algorithme `partition`.

Multiplication rapide de polynômes

Multiplication rapide

Exemples historiques

On va maintenant voir deux autres exemples, historiquement très importants, d'algorithmes "diviser pour régner".

Le but est de multiplier rapidement deux polynômes (ou deux matrices) de même taille.

Multiplication rapide de polynômes

Problème

On souhaite effectuer la multiplication de deux polynômes de même degré $n-1$.

La méthode naïve a une complexité en $O(n^2)$, et il a longtemps été conjecturé qu'il n'existait pas de meilleur algorithme.

Multiplication rapide de polynômes

Karatsuba

Anatoly Karatsuba découvrit la méthode qu'on va voir dans ce cours en 1960 : celle-ci a une complexité en $O(n^{1.59})$.

Histoire

Historiquement, la question que s'était posée Karatsuba portait sur la multiplication d'entiers représentés en binaire.

Mais un entier représenté dans une base b comme $N = \sum a_k b^k$ et le polynôme $\sum a_k X^k$ sont deux objets très proches.

Néanmoins, la multiplication de polynômes est légèrement plus facile car il n'y a pas de retenue à gérer.

Multiplication rapide de polynômes

Impact et recherche

Cet algorithme a eu une importance considérable, et le problème de multiplication de polynômes a fait l'objet d'intenses recherches depuis.

La meilleure borne connue pour ce problème est quasi-optimale : $O(n \log n \log \log n)$.

Multiplication rapide de polynômes

Problème

Soit $n \in \mathbb{N}$. Soit $P = \sum_{k=0}^{n-1} p_k X^k$ et $Q = \sum_{k=0}^{n-1} q_k X^k$ deux polynômes de même degré, où les coefficients sont dans un anneau commutatif $\mathbb{A}$ quelconque. Une opération dans $\mathbb{A}$ (multiplication, addition, soustraction) est considérée comme **élémentaire**.

On représente P et Q par les tableaux $[|p_0; p_1; \ldots; p_{n-1}|]$ et $[|q_0; q_1; \ldots; q_{n-1}|]$.

Le produit $P \times Q$ est un polynôme de degré $2n - 2$.

Comment obtenir efficacement le tableau de ses $2n - 1$ coefficients ?

Méthode naïve

Méthode naïve

On utilise simplement la relation $PQ = \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} p_k q_j X^{k+j}$, ce qui nous donne un algorithme en $O(n^2)$.

```
1   let produit p q =
2     let n = Array.length p in
3     assert (Array.length q = n) ; (* lève l'exception Assert_Failure si la condition est fausse *)
4     let t = Array.make (2*n-1) 0 in
5     for k=0 to n-1 do
6       for j=0 to n-1 do
7         t.(j+k) <- t.(j+k) + p.(k)*q.(j)
8       done
9     done ;
10    t
11  ;;
```

Paradigme “diviser pour régner” : l’algorithme de Karatsuba

Idée

On va utiliser le principe “diviser pour régner” pour essayer de résoudre le problème plus efficacement.

Cas $n = 2$

Si P et Q sont tous les deux de degré 1, on a :

$$PQ = p_0q_0 + (p_0q_1 + p_1q_0)X + p_1q_1X^2$$

On a *a priori* 4 multiplications.

Paradigme “diviser pour régner” : l’algorithme de Karatsuba

Cas $n = 2$

Si P et Q sont tous les deux de degré 1, on a :

$$PQ = p_0q_0 + (p_0q_1 + p_1q_0)X + p_1q_1X^2$$

Si on calcule les 3 produits suivants :

$$\begin{cases} t_0 = p_0q_0 \\ t_1 = p_1q_1 \\ t_2 = (p_0 + p_1) \times (q_0 + q_1) \end{cases}$$

On a alors $p_0q_1 + p_1q_0 = t_2 - t_1 - t_0$.

On a donc que 3 produits à calculer.

Paradigme “diviser pour régner” : l’algorithme de Karatsuba

Algorithme de Karatsuba

- Si $n = 1$, P et Q sont constants, et le produit est simplement le produit des deux constantes.
- Sinon, posons $m = \lfloor \frac{n}{2} \rfloor$, et découpons nos polynômes en 2. On écrit donc $P_0 = \sum_{k=0}^{m-1} p_k X^k$ et $P_1 = \sum_{k=m}^{n-1} p_k X^{k-m}$ de sorte que $P = P_0 + X^m P_1$.
  De même, $Q = Q_0 + X^m Q_1$.
- Posons ensuite

$$T_0 = P_0 Q_0 \ ; \ T_1 = P_1 Q_1 \ ; \ T_2 = (P_0 + P_1) \times (Q_0 + Q_1)$$

Comme pour le cas $n = 2$, on a :

$$P \times Q = T_0 + X^m (T_2 - T_1 - T_0) + X^{2m} T_1$$

Paradigme “diviser pour régner” : l’algorithme de Karatsuba

$$T_0 = P_0 Q_0 \ ; \ T_1 = P_1 Q_1 \ ; \ T_2 = (P_0 + P_1) \times (Q_0 + Q_1)$$

$$P \times Q = T_0 + X^m (T_2 - T_1 - T_0) + X^{2m} T_1$$

```
let rec karatsuba p q =
  let n = Array.length p in
  if n=1 then [|p.(0)*q.(0)|] else
    begin
      let k = n/2 in
      let p0, p1 = Array.sub p 0 k, Array.sub p k (n-k)
      and q0, q1 = Array.sub q 0 k, Array.sub q k (n-k) in
      let t0, t1 = karatsuba p0 q0, karatsuba p1 q1 in
      for i=0 to k-1 do
        p1.(i) <- p0.(i) + p1.(i) ;
        q1.(i) <- q0.(i) + q1.(i)
      done ;
      let t2 = karatsuba p1 q1 in
      let t = Array.make (2*n-1) 0 in
      for i=0 to Array.length t0 - 1 do
        t.(i) <- t.(i) + t0.(i) ;
        t.(i+k) <- t.(i+k) - t0.(i)
      done ;
      for i=0 to Array.length t1 - 1 do
        t.(i+k) <- t.(i+k) - t1.(i) + t2.(i) ;
        t.(i+2*k) <- t.(i+2*k) + t1.(i)
      done ;
      t
    end
;;
```

Complexité

Pour déterminer la complexité globale, il faut estimer la complexité des deux étapes diviser et régner.

Ici, on compte les opérations arithmétiques dans l'anneau $\mathbb{A}$ (additions, multiplications, soustractions).

- Pour diviser, il faut créer les tableaux associés aux polynômes P_0 et P_1, et calculer les polynômes $P_0 + P_1$ (et idem avec Q_0 et Q_1) ce qui se fait en $O(n)$.
- Pour régner, il faut créer une liste T de taille $(2n-1)$, et combiner les trois listes associées à T_0, T_1 et T_2 pour obtenir la liste T. Ceci se fait également en $O(n)$.

Complexité

On fait 3 appels récursifs sur des polynômes de taille $n/2$, donc

$$C(n) = 3 \cdot C(n/2) + O(n)$$

On peut donc appliquer le théorème maître.

Complexité

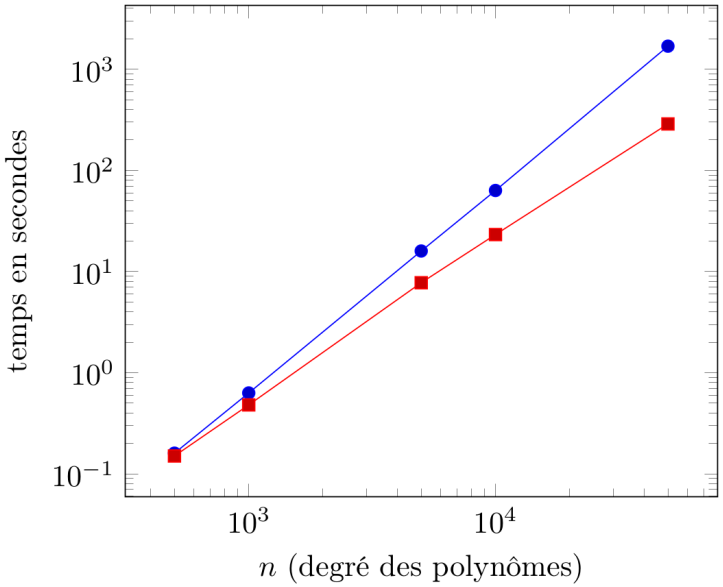
L'algorithme de Karatsuba a une complexité en $O(n^{\log_2(3)})$, avec $\log_2(3) \approx 1.58 < 2$.

En pratique

Test

On teste nos algorithmes sur des polynômes de divers degrés, avec des coefficients tirés aléatoirement sur l'intervalle $[\![-1000, 1000]\!]$.

$\deg(P) = \deg(Q) = n$	100	500	1000	5000	10000	50000
multiplication naïve	0.006	0.16	0.63	15.96	63.3	1692
Karatsuba	0.07	0.15	0.48	7.75	23.2	288



Multiplication rapide de matrices

Multiplication rapide de matrices

Problème

On souhaite effectuer la multiplication de deux matrices de taille $n \times n$.

La méthode naïve a une complexité en $O(n^3)$, et il a longtemps été conjecturé qu'il n'existait pas de meilleur algorithme.

Multiplication rapide de matrices

Strassen

Volker Strassen découvrit la méthode qu'on va voir dans ce cours en 1969 : celle-ci a une complexité en $O(n^{2.81})$.

Impact et recherche

Aujourd'hui, le problème de la multiplication matricielle résiste toujours. En 2014, François Le Gall a publié un algorithme en $O(n^{2.3728639})$, mais la constante cachée dans le O est tellement énorme que l'algorithme n'est pas utilisable en pratique.

Il est conjecturé qu'il existe un algorithme de complexité $O(n^{2+\varepsilon})$ pour tout $\varepsilon > 0$, mais la recherche en est encore très loin !

Multiplication rapide de matrices

Problème

Soit $n \in \mathbb{N}^*$. On se donne $A = (a_{i,j})$ et $B = (b_{i,j})$ deux matrices carrées de même taille $n \times n$, à coefficients dans un anneau commutatif $\mathbb{A}$.

Le produit $C = A \times B$ est une matrice de taille $n \times n$, le but est de calculer ses n^2 coefficients $(c_{i,j})$ efficacement.

Algorithme naïf

La formule suivante mène à un algorithme en $O(n^3)$:

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

Algorithme de Strassen

Formules de Strassen

Supposons que n soit pair. Alors A, B et C se décomposent chacune en 4 blocs de taille $\frac{n}{2} \times \frac{n}{2}$:

$$A = \left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right), \; B = \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right), \; C = \left(\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array}\right)$$

On considère alors les 7 produits P_i suivants, à partir desquels on peut obtenir les $C_{i,j}$:

$$\begin{cases} P_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\ P_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\ P_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\ P_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\ P_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\ P_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ P_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{cases}$$

$$\begin{cases} C_{1,1} &= P_1 + P_4 - P_5 + P_7 \\ C_{1,2} &= P_3 + P_5 \\ C_{2,1} &= P_2 + P_4 \\ C_{2,2} &= P_1 - P_2 + P_3 + P_6 \end{cases}$$

Algorithme de Strassen

Analyse

La multiplication naïve par blocs requiert 8 multiplications, alors que cette méthode n'en requiert que 7 : c'est là que se situe le gain de l'algorithme de Strassen.

Complexité

L'algorithme de Strassen utilise les formules précédentes, récursivement. Dans le cas $n = 1$, on effectue simplement le produit de deux coefficients.

Les additions et soustractions nécessaires se font en $O(n^2)$, on a donc une complexité vérifiant : $C(n) = 7 \cdot C(n/2) + O(n^2)$. D'après le théorème maître, on a donc $C(n) = O(n^{\log_2(7)})$, avec $\log_2(7) \approx 2.81 < 3$.

Algorithme de Strassen

Implémentation pratique

La constante cachée dans le $O(n^{\log_2(7)})$ étant assez élevée, il faut implémenter finement l'algorithme pour obtenir un gain par rapport à l'algorithme naïf, et ce gain ne s'observe que pour des matrices de tailles déjà conséquentes.

En pratique, on n'utilise pas les formules de Strassen pour de petites matrices : on se contente de l'algorithme naïf.

Calcul de la paire de points la plus proche dans un nuage de points

Calcul de la paire de points les plus proches dans un nuage de points

Problème

Pour clore ce chapitre, on conclut par un problème géométrique : la recherche du couples de points le plus proche dans un nuage de points.

Le problème est le suivant : on se donne un tableau de taille n contenant des couples de flottants, représentant un nuage de points, et on veut identifier les deux points les plus proches, naturellement avec la meilleure complexité possible.

Calcul de la paire de points les plus proches dans un nuage de points

Algorithme naïf

Il y a $\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$ couples de points distincts dans le nuage : on peut donc tous les examiner en conservant le couple le plus proche.

On obtient donc la solution au problème avec une complexité en $O(n^2)$.

Approche “diviser pour régner”

Dimension 1

Avant de discuter d’un algorithme efficace, regardons déjà ce qu’on peut faire avec comme nuage de points un sous-ensemble de la droite réelle.

Approche “diviser pour régner”

Solution en dimension 1

- On peut trier les n points d'un nuage réel en temps $O(n \log n)$ avec le tri fusion.
- Trouver le couple de points les plus proches est alors très facile car ils seront forcément côte à côte dans le tableau : $\hookrightarrow$ un parcours en $O(n)$ suffit.

En dimension 1, ce problème a donc une solution en $O(n \log n)$.

De manière assez surprenante, on peut obtenir la même complexité en dimension 2.

Approche “diviser pour régner”

Dimension 2

On va appliquer une stratégie “diviser pour régner”.

- Si le nuage a peu de points (disons ≤ 4), on applique l'algorithme naïf.
- Sinon :
  - On peut séparer le nuage de points en deux parties (presque) égales, autour d'un axe vertical.
    Remarquons tout de suite qu'il est commode que les points soient triés par abscisse croissante.
  - On calcule ensuite (récursivement) la distance minimale et le couple de points correspondant dans les parties gauches et droites.
  - Il faut ensuite calculer la distance entre les points situés de part et d'autre de la droite verticale.

Approche “diviser pour régner”

Étape de règne

Il faut faire attention à ce que cette dernière étape (de **règne**) ne soit pas trop coûteuse : si on le fait naïvement, on va tester toutes les combinaisons possibles entre un point parmi $\frac{n}{2}$ à gauche, et un point parmi $\frac{n}{2}$ à droite, ce qui va coûter $O(n^2)$.

On peut optimiser cette étape, pour qu'elle coûte $O(n)$.

Notations

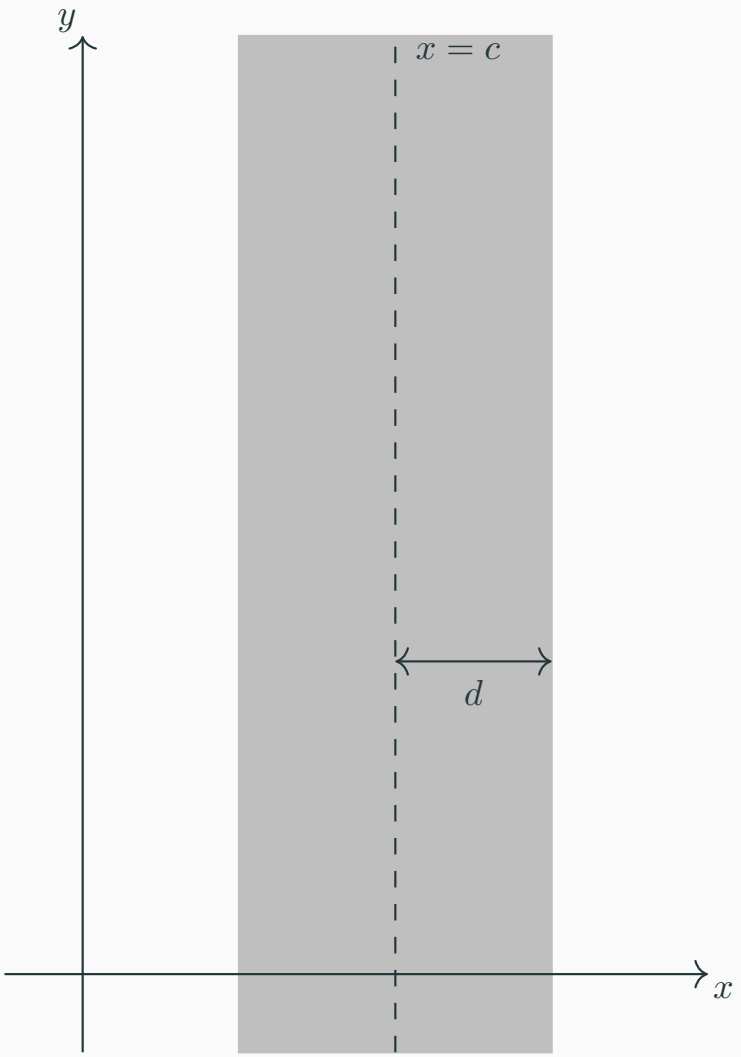
Notons c l'abscisse au niveau de laquelle on a divisé notre nuage de points en 2.

Soit d_1 et d_2 les distances minimales trouvées par les appels récursifs sur les parties gauche et droite respectivement.
Notons $d = \min(d_1, d_2)$.

Approche “diviser pour régner”

Étape de règne

On cherche un point de la partie gauche et un point de la partie droite dont la distance est $< d$.
S'ils ne sont pas dans la bande d'abscisses $[c - d, c + d]$ (en gris sur la figure), ce n'est pas la peine de les tester. On va parcourir les points de cette bande par ordonnées croissantes.

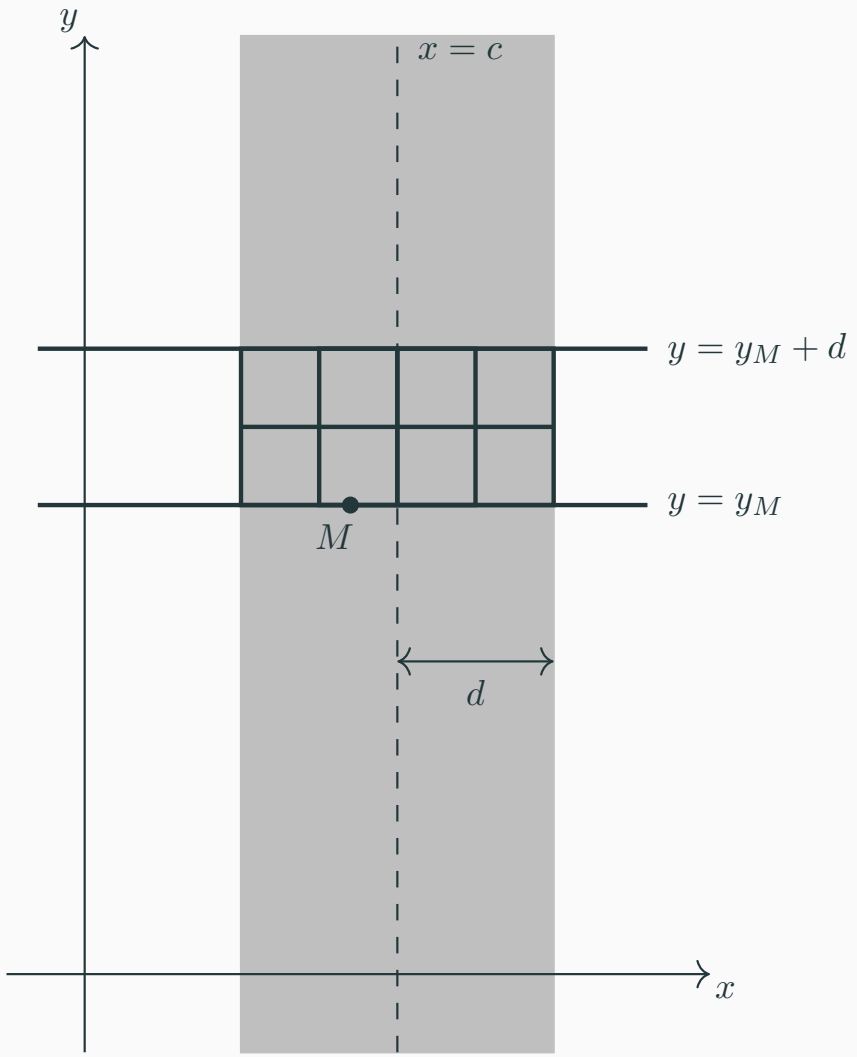


Approche “diviser pour régner”

Étape de règne

Soit $M(x_M, y_M)$ un point de cette bande. Un point d'ordonnée $\geq y_M$ est à distance $< d$ de M si et seulement s'il est dans le rectangle $[c-d, c+d] \times [y_M, y_M + d]$.

Ce rectangle se découpe alors en 8 carrés de côtés $\frac{d}{2}$ (cf. figure).



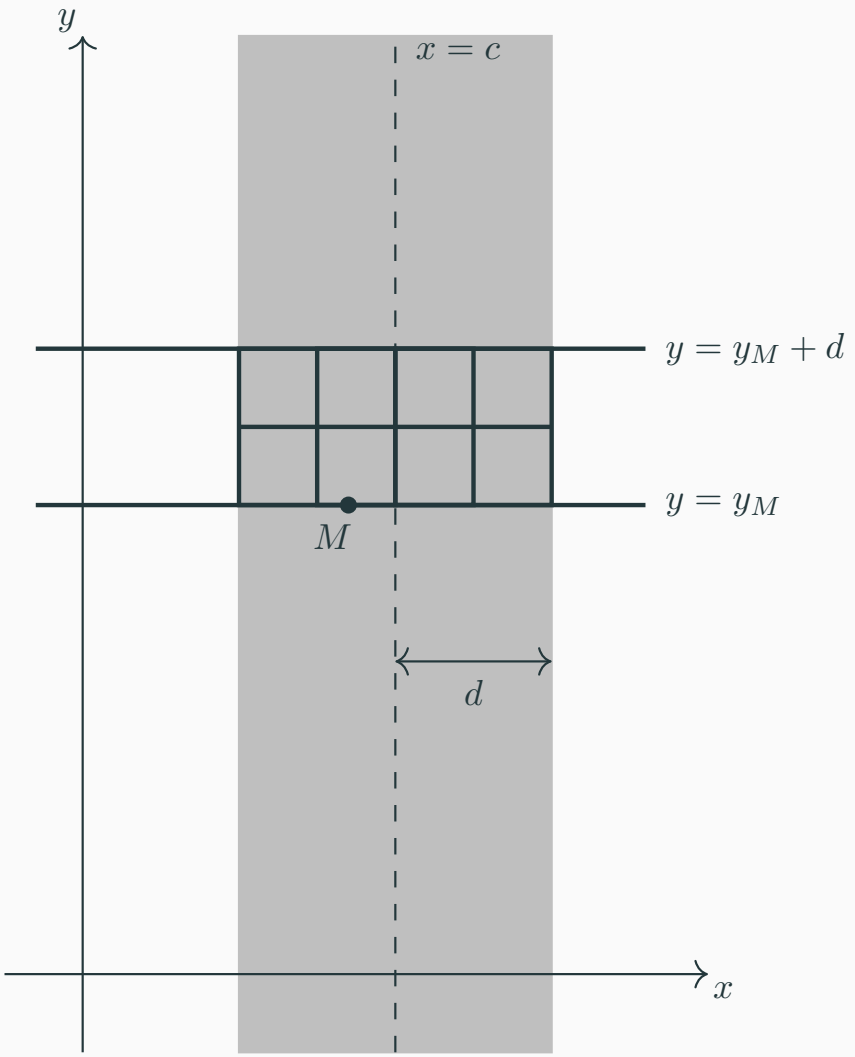
Approche “diviser pour régner”

Étape de règne

Si deux points du nuage appartenaient au même carré, leur distance serait inférieure à la diagonale : $\frac{d}{\sqrt{2}}$.
Absurde par définition de d.

Il n'y a donc qu'un seul point du nuage au plus par carré, donc 7 points en plus de M dans le rectangle.

Et ce sont les seuls points potentiellement à distance $< d$ de M.

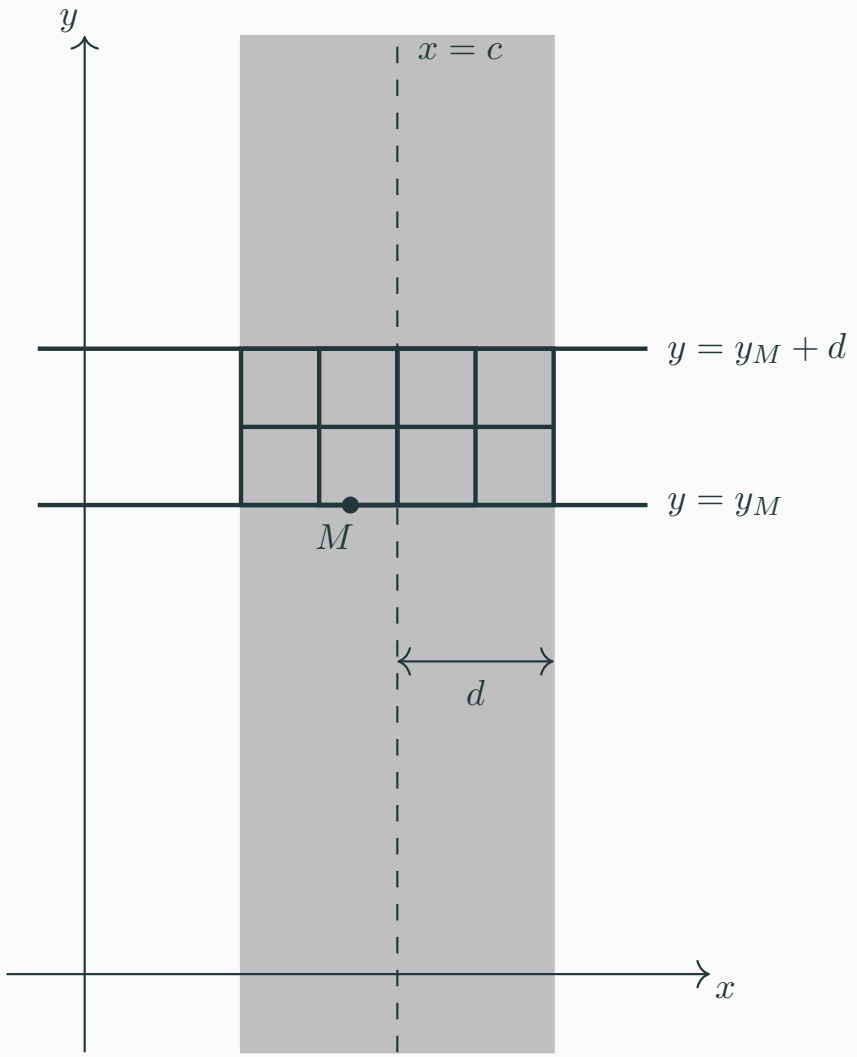


Approche “diviser pour régner”

Étape de règne

Si on parcourt les points M de la bande par ordonnée croissante, il suffit donc de calculer la distance entre M et les 7 points suivants.

Ainsi, l'analyse de la bande se fait en $O(n)$.



Approche “diviser pour régner”

Complexité

Dans ce cas, la complexité de notre algorithme vérifiera :

$$C(n) = 2 \cdot C(n/2) + O(n)$$

Donc, d’après le théorème maître, $C(n) = O(n \log n)$.

Approche “diviser pour régner”

Implémentation pratique

Résumons les idées de l'algorithme :

- Pour séparer les points du nuage autour d'un axe vertical, il est pratique que les points soient triés par **abscisse** croissante.
- Pour examiner les points de la bande, il est pratique que les points soient triés par **ordonnée** croissante.

Si l'on retrie notre ensemble selon les abscisses ou les ordonnées à chaque appel récursif, on n'obtiendra pas un algorithme en $O(n \log n)$.

Il vaut mieux introduire de la redondance dans la manière de stocker les points du nuage.

Approche “diviser pour régner”

Implémentation pratique

On duplique initialement le nuage sous forme de deux tableaux :

- t_1 contient les points triés par **abscisse** croissante ;
- t_2 contient les points triés par **ordonnée** croissante ;

Ce **précalcul** coûte $O(n \log n)$ si on utilise le tri fusion, qui ne sera appelé que 2 fois en tout.

Notre fonction récursive prend donc en entrée ces deux tableaux t_1 et t_2.

Approche “diviser pour régner”

Implémentation pratique

À chaque étape, avant les appels récursifs, il faut :

- couper t_1 en deux au milieu (notons c l'abscisse du milieu) ;
- répartir les éléments de t_2 selon c (en conservant l'ordre de t_2), ce qui se fait en $O(n)$.

Une fois les appels récursifs effectués, et la distance d calculée :

- on extrait les points de la bande $[c-d, c+d]$ à partir de t_2 (en conservant l'ordre de t_2) ;
- on parcourt la bande en $O(n)$.

Approche “diviser pour régner”

Remarque

Pour gérer les cas “pathologiques”, par exemple si tous les points se retrouvent sur une même droite $x = c$, il vaut mieux trier les tableaux selon les ordres suivants :

- t_1 : trié selon l'ordre lexicographique où l'on regarde **d'abord les abscisses**, puis les ordonnées ;
- t_2 : trié selon l'ordre lexicographique où l'on regarde **d'abord les ordonnées**, puis les abscisses.