Analyse d'algorithmes

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Introduction

Introduction

But

Le but de ce chapitre est d'étudier de manière théorique les algorithmes. On va donner les outils permettant de répondre aux trois questions suivantes :

- **Terminaison** : l'algorithme s'arrête-t-il un jour ?
- **Correction** : l'algorithme fait-il bien ce qu'il est sensé faire ? Autrement dit, est-il correct ?
- **Complexité** : combien de temps met-il à s'exécuter ?

Introduction

Algorithme vs Programme

Tout ce dont nous allons discuter dans ce chapitre ne dépend pas du langage dans lequel on a écrit notre **programme**.

La discussion porte en fait sur l'**algorithme** utilisé, peu importe dans quel langage on va l'implémenter.

Les algorithmes

Définition (algorithme)

Un algorithme est une fonction qui prend des données en arguments, effectue une séquence finie non ambigüe d'instructions, et renvoie un résultat.

Donald Knuth

Étendons un peu cette définition en donnant une liste de points caractérisant un algorithme, par **Donald Knuth**.

Les algorithmes

Prérequis d'un algorithme (Donald Knuth)

- **finitude** : un algorithme doit toujours se terminer après un nombre fini détapes.
- **définition précise** : chaque étape d'un algorithme doit être définie précisément et sans ambiguïté.
- **entrées** : des quantités (prises dans un ensemble spécifié) qui lui sont données avant qu'un algorithme ne commence.
- **sorties** : des quantités ayant une relation spécifiée avec les entrées.
- **rendement** : toutes les opérations que l'algorithme doit accomplir doivent être suffisamment basiques pour pouvoir être en principe réalisées dans une durée finie par un humain utilisant un papier et un crayon.

Description d'un algorithme

Spécification

Pour décrire un algorithme, on lui donne en général un nom adéquat, on précise quels sont ses paramètres (entrées) et le résultat (sorties) qu'il est sensé renvoyer.

On précise également de quelle manière il agit sur son **environnement** :

- modification de la mémoire,
- affichage éventuel à l'écran,
- ...

Tout ceci constitue la **spécification de l'algorithme**.

Analyse d'algorithme

Blocs

Dans nos algorithmes, outre les opérations d'affectations, d'entrée/sortie, et de manipulations de variables, on peut trouver des blocs simples :

- boucles `for` ;
- boucles `while` ;
- blocs conditionnels `if`, `else if`, . . . , `else`.

Ce découpage en blocs simples est essentiel.

Terminaison

Terminaison

Terminaison

Pour montrer qu'un algorithme termine quel que soit le jeu de paramètres passé en entrée respectant la spécification, il faut montrer que chaque bloc élémentaire décrit précédemment termine.

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

Le seul soucis pourrait venir d'une boucle `while`.

Terminaison d'une boucle `while`

```
1  while (n != 0)
2  {
3    n--;
4  }
```

Exemple

Considérons le code ci-dessus.

- Si, avant la boucle **`while`**, la variable n contient un entier positif, cette boucle s'arrêtera au bout de n étapes.
- Par contre, si elle contient un entier strictement négatif, c'est la catastrophe : n prendra une infinité de valeurs, toutes strictement négatives.

Exponentiation

Algorithme d'exponentiation

```
1  int expo(int x,int n)
2  {
3    int res = 1;
4    for (int i = 0; i<n; i++)
5    {
6      res *= x;
7    }
8    return res;
9  }
```

Exemple

Prenons un exemple plus concret : le calcul d'une puissance.

Pour x un entier (ou un flottant), et n un entier naturel, on peut partir de $y = 1$ et multiplier n fois y par x.

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  int expo_rapide(int x, int n)
2  {
3    int y = x, z = 1, m = n;
4    while (m>0)
5    {
6      int q = m / 2, r = m%2;
7      if (r == 1)
8      {
9        z *= y;
10     }
11     y *= y;
12     m = q;
13   }
14   return z;
15 }
```

Exponentiation rapide

L'algorithme de ci-dessus porte le nom d'**algorithme d'exponentiation rapide**.

On montrera par la suite qu'il est bien plus efficace que l'algorithme d'exponentiation naïf vu précédemment.

Exponentiation rapide

```
1  int m = n;
2  while (m>0)
3  {
4      int q = m / 2;
5      m = q;
6  }
```

Terminaison

La fonction suppose que l'entier n est positif dans sa spécification.

Observons maintenant le code : la condition du while porte sur m, qui doit être strictement positif pour qu'un tour de boucle s'effectue.

Si on supprime tout ce qui n'a pas trait à la modification de la variable m dans le code, on le code ci-dessus.

Exponentiation rapide

```
1  int m = n;
2  while (m>0)
3  {
4      int q = m / 2;
5      m = q;
6  }
```

Terminaison

Ainsi, les valeurs prises par m sont positives et strictement décroissantes à chaque itération de la boucle, car $\lfloor \frac{m}{2} \rfloor < m$ pour tout entier $m > 0$.

Donc m fini par être nul, et la boucle termine.

Variant de boucle

Méthode générale

En général, pour montrer la terminaison d'une boucle, on procède ainsi : on exhibe une quantité, dépendant des paramètres, 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 la boucle termine.

Définition

Un **variant de boucle** est une quantité positive, à valeurs dans $\mathbb{N}$, dépendant des variables de la boucle, qui décroit strictement à chaque passage dans la boucle.

Variant de boucle

En pratique

Dans l'exemple précédent, le variant de boucle est à peu près évident, et ce sera en général le cas pour nos algorithmes.

Difficulté

Mais cela ne veut pas dire que prouver la terminaison d'un programme est toujours facile !

Parfois, il n'est pas du tout évident de montrer (ou d'infirmer) qu'une boucle termine.

variant de boucle

Fonction de Syracuse

```
1  int syracuse(int n)
2  {
3    int m = n;
4    while (m != 1)
5    {
6      if (m%2 == 0)
7      {
8        m = m / 2;
9      }
10     else
11     {
12       m = 3*m + 1;
13     }
14   }
15   return 1;
16 }
```

Exemple

Il est conjecturé que la fonction ci-dessus termine quel que soit l'entier $n > 0$ passé en argument, mais personne n'a été capable de le prouver.

Correction

Correction

Blocs

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

Pour cela, on va étudier les blocs qui constituent l'algorithme, et montrer que chacun de ces blocs effectue une action bien précise.

- Pour les blocs **conditionnels** (`if`, `else if`, ..., `else`), il n'y a en général pas de difficulté.
- En revanche, analyser les boucles `for` et `while` est essentiel, car l'action de ces boucles n'est pas forcément évidente en première lecture.

Invariant de boucle

Définition

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

Correction des boucles `while`

Algorithme d'exponentiation rapide

```
1  int expo_rapide(int x, int n)
2  {
3    int y = x, z = 1, m = n;
4    /* Inv: z * y^m = x^n */
5    while (m>0)
6    {
7      // Inv
8      int q = m / 2, r = m%2;
9      if (r == 1)
10     {
11       z *= y;
12     }
13     y *= y;
14     m = q;
15     // Inv
16   }
17   // Inv
18   return z;
19 }
```

Exemple

Reprenons l'algorithme d'exponentiation rapide, qui consiste principalement en une boucle **`while`**.

Correction des boucles `while`

Invariant de boucle

Juste avant la boucle, on a $y = x$, $z = 1$, et $m = n$, donc $z \times y^m = x^n$.

On va montrer que si cette propriété est vraie **en haut du corps de la boucle**, alors elle est vraie **en bas du corps de la boucle**.

Correction des boucles `while`

Invariant de boucle

$$z \times y^m = x^n$$

```
1   while (m>0)
2   {
3     // Inv
4     int q = m / 2, r = m%2;
5     if (r == 1)
6     {
7       z *= y;
8     }
9     y *= y;
10    m = q;
11    // Inv
12  }
```

Preuve

Si m est pair, alors $q = \frac{m}{2}$, $r = 0$, $y' = y^2$, $m' = q = \frac{m}{2}$, et $z' = z$.

On a alors $z' \times y'^{m'} = z \times (y^2)^{\frac{m}{2}} = z \times y^m = x^n$.

Correction des boucles `while`

Invariant de boucle

$$z \times y^m = x^n$$

```
1   while (m>0)
2   {
3     // Inv
4     int q = m / 2, r = m%2;
5     if (r == 1)
6     {
7       z *= y;
8     }
9     y *= y;
10    m = q;
11    // Inv
12  }
```

Preuve

Si m est impair, alors $q = \frac{m-1}{2}$, $r = 1$, $y' = y^2$, $m' = q = \frac{m-1}{2}$, et $z' = z \times y$.

On a alors $z' \times y'^{m'} = z \times y \times (y^2)^{\frac{m-1}{2}} = z \times y^m = x^n$.

Correction des boucles `while`

Conclusion

Ainsi,

- la propriété $z \times y^m = x^n$ est vraie avant la boucle,
- et elle est maintenue à chaque passage dans la boucle (c'est un **invariant de boucle**),

donc cette propriété est vérifiée **après** la boucle.

Or, en sortie de boucle, $m = 0$. Donc la propriété signifie que $z = x^n$.

Comme on renvoie z, l'algorithme renvoie bien la valeur de x^n : il est donc **correct**.

Correction des boucles `for`

Simplification

Pour faciliter la compréhension, on ne traitera que des boucles de la forme :

```
for (int i = m; i < n; i += p)
{
  /* instructions qui ne modifient ni i, ni m, ni n, ni p */
}
```

Ainsi, l'invariant de boucle doit être vérifié pour $i = m$ au départ, et il faut montrer que si $\mathrm{Inv}(i)$ est vrai en haut du corps de la boucle, alors $\mathrm{Inv}(i+p)$ est vrai en bas du corps de la boucle.

Dans ce cas, l'invariant est vérifié après la boucle pour la dernière valeur de i.

Correction des boucles for

boucle while

```
1  int i = 0;
2  // Inv(0)
3  while (i < n)
4  {
5    // Inv(i)
6    /* instructions qui ne modifient
7    ni i, ni n */
8    i++;
9    // Inv(i)
10 }
11 // Inv(n)
```

boucle for équivalente

```
1  int i;
2  // Inv(0)
3  for (i = 0; i < n; i++)
4  {
5    // Inv(i)
6    /* instructions qui ne modifient
7    ni i, ni n */
8    /* les mêmes que dans le while */
9    // Inv(i+1)
10 }
11 // Inv(n)
```

Principe

La boucle **while** de gauche est équivalente à la boucle **for** 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 while

```
1  int i = 0;
2  // Inv(0)
3  while (i < n)
4  {
5    // Inv(i)
6    /* instructions qui ne modifient
7    ni i, ni n */
8    i++;
9    // Inv(i)
10 }
11 // Inv(n)
```

boucle for équivalente

```
1  int i;
2  // Inv(0)
3  for (i = 0; i < n; i++)
4  {
5    // Inv(i)
6    /* instructions qui ne modifient
7    ni i, ni n */
8    /* les mêmes que dans le while */
9    // Inv(i+1)
10 }
11 // Inv(n)
```

Principe

- On montre d'abord que $\mathrm{Inv}(0)$ 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, $\mathrm{Inv}(n)$ est vrai après la boucle.

Exemple : somme des éléments d'un tableau

Somme des éléments d'un tableau

```c
int somme(int *tab, int n)
{
  int s = 0;
  for (int i = 0; i < n; i++)
  {
    // Inv(i): s est la somme des i premiers éléments de tab.
    s += tab[i];
    // Inv(i+1): s est la somme des i+1 premiers éléments de tab.
  }
  return s;
}
```

Exemple

L'algorithme ci-dessus calcule la somme des éléments d'un tableau **tab** de taille n à l'aide d'une boucle **for**.

L'invariant est tout simplement :

$\mathrm{Inv}(i)$: "**s** est la somme des i premiers éléments de **tab**."

Exemple : moyenne des éléments d'un tableau

Moyenne des éléments d'un tableau

```
1  int moyenne(int *tab, int n)
2  {
3    return somme(tab, n) / n;
4  }
```

Exemple

Comme la fonction `somme` précédente est correcte, on en déduit immédiatement la correction de la fonction `moyenne` ci-dessus.

Exemple : recherche du maximum d'un tableau

```
1   int maximum(int *tab, int n)
2   {
3     int m = tab[0];
4     for (int i = 1; i < n; i++)
5     {
6       // Inv(i): m est le plus grand élément parmi les i premiers elements de tab.
7       if (tab[i] > m)
8       {
9         m = tab[i];
10      }
11      // Inv(i+1): m est le plus grand élément parmi les i+1 premiers elements de tab.
12    }
13    return m;
14  }
```

Exemple

L'algorithme ci-dessus recherche le maximum d'un tableau.

Ici aussi, l'invariant est immédiat :

$\mathrm{Inv}(i)$: "`m` est le plus grand élément parmi les i premiers éléments de **`tab`**."

Cas d'une instruction de sortie dans la boucle

Instruction de sortie

Si dans la boucle se trouve une instruction de sortie (`return` par exemple), elle sera en général à l'intérieur d'une instruction conditionnelle.

Dans ce cas :

- Tant qu'on ne suit pas cette instruction de sortie, on prouve l'invariant de boucle comme précédemment.
- Lorsque l'instruction de sortie est suivie, on ne prouve plus que l'invariant est toujours vrai à la fin de la boucle, mais on regarde quelle propriété est vraie en sortie de boucle.

Exemple : recherche d'un élément dans un tableau

Recherche dans un tableau

```
1  bool recherche(int *tab,int n, int x)
2  {
3    for (int i = 0; i < n; i++)
4    {
5      if (tab[i] == x) { return true; }
6    }
7    return false;
8  }
```

Exemple

L'algorithme ci-dessus cherche si **x** se trouve dans **tab**.

Si on trouve un indice i tel que **tab[i] == x**, on sort immédiatement de la fonction en renvoyant **true**, ce qui est correct.

Sinon, l'invariant est vérifié en bas de la boucle.

Ainsi, si l'on ne passe jamais par l'instruction **return true**, $\mathrm{Inv}(n)$ est vrai, ce qui veut dire que **x** n'est pas dans **tab**, et il est correct de renvoyer **false**.

Recherche dichotomique dans un tableau trié

Recherche dichotomique

```
1  bool recherche_dicho(int *tab, int n, int x)
2  {
3    int g = 0, d = n;
4    while (g < d)
5    {
6      // Inv: x ne se trouve ni dans tab[0:g] ni dans tab[d:n].
7      int m = (g + d) / 2;
8      if (tab[m] == x) { return true; }
9      else if (tab[m] < x) { g = m + 1; }
10     else { d = m; }
11     // Inv: x ne se trouve ni dans tab[0:g] ni dans tab[d:n].
12   }
13   return false;
14 }
```

Recherche dichotomique

Si l'on suppose que le tableau est trié, l'algorithme ci-dessus est beaucoup plus efficace pour trouver un élément dans le tableau.

Recherche dichotomique dans un tableau trié

Recherche dichotomique

```
1  bool recherche_dicho(int *tab, int n, int x)
2  {
3    int g = 0, d = n;
4    while (g < d)
5    {
6      // Inv: x ne se trouve ni dans tab[0:g] ni dans tab[d:n].
7      int m = (g + d) / 2;
8      if (tab[m] == x) { return true; }
9      else if (tab[m] < x) { g = m + 1; }
10     else { d = m; }
11     // Inv: x ne se trouve ni dans tab[0:g] ni dans tab[d:n].
12   }
13   return false;
14 }
```

Terminaison

La quantité $d - g$ est à valeurs dans $\mathbb{N}$ et décroît strictement à chaque itération de la boucle **`while`**, donc l'algorithme termine.

Recherche dichotomique dans un tableau trié

Recherche dichotomique

```
1  bool recherche_dicho(int *tab, int n, int x)
2  {
3    int g = 0, d = n;
4    while (g < d)
5    {
6      /* Inv: x ne se trouve ni dans tab[0:g]
7         ni dans tab[d:n]. */
8      int m = (g + d) / 2;
9      if (tab[m] == x) { return true; }
10     else if (tab[m] < x) { g = m + 1; }
11     else { d = m; }
12     /* Inv: x ne se trouve ni dans tab[0:g]
13        ni dans tab[d:n]. */
14   }
15   return false;
16 }
```

Correction

Si **`tab[m] == x`**, on renvoie simplement **`true`** et la fonction est correcte.

Recherche dichotomique dans un tableau trié

Recherche dichotomique

```
1  bool recherche_dicho(int *tab, int n, int x)
2  {
3    int g = 0, d = n;
4    while (g < d)
5    {
6      /* Inv: x ne se trouve ni dans tab[0:g]
7         ni dans tab[d:n]. */
8      int m = (g + d) / 2;
9      if (tab[m] == x) { return true; }
10     else if (tab[m] < x) { g = m + 1; }
11     else { d = m; }
12     /* Inv: x ne se trouve ni dans tab[0:g]
13        ni dans tab[d:n]. */
14   }
15   return false;
16 }
```

Correction

Sinon :

- si **tab[m] < x**, comme le tableau est triée cela signifie que **x** ne peut se trouver qu'à un indice $> m$,
- et de même, si **tab[m] > x**, **x** ne peut se trouver qu'à un indice $< m$.

Dans tous les cas, l'invariant de boucle est vrai en fin de boucle s'il était vrai en début de boucle.

Recherche dichotomique dans un tableau trié

Recherche dichotomique

```
1  bool recherche_dicho(int *tab, int n, int x)
2  {
3    int g = 0, d = n;
4    while (g < d)
5    {
6      /* Inv: x ne se trouve ni dans tab[0:g]
7         ni dans tab[d:n]. */
8      int m = (g + d) / 2;
9      if (tab[m] == x) { return true; }
10     else if (tab[m] < x) { g = m + 1; }
11     else { d = m; }
12     /* Inv: x ne se trouve ni dans tab[0:g]
13        ni dans tab[d:n]. */
14   }
15   return false;
16 }
```

Correction

De plus, l'invariant est vrai avant la boucle, car les deux tableaux de l'invariant sont alors vides.

Ainsi, après la boucle, comme $g = d$, l'invariant assure que **x** n'est pas dans **tab**, et il est donc correct de renvoyer **false**.

Exemple : le tri par sélection

Tri d'un tableau

On cherche un algorithme qui prend en entrée un tableau **tab**, et qui trie ses éléments dans l'ordre croissant.

Il existe différentes méthodes pour trier un tableau, mais on va étudier ici l'un des plus simples à titre d'exemple : le **tri par sélection**.

Exemple : le tri par sélection

Tri par sélection

Le principe du tri par sélection est le suivant :

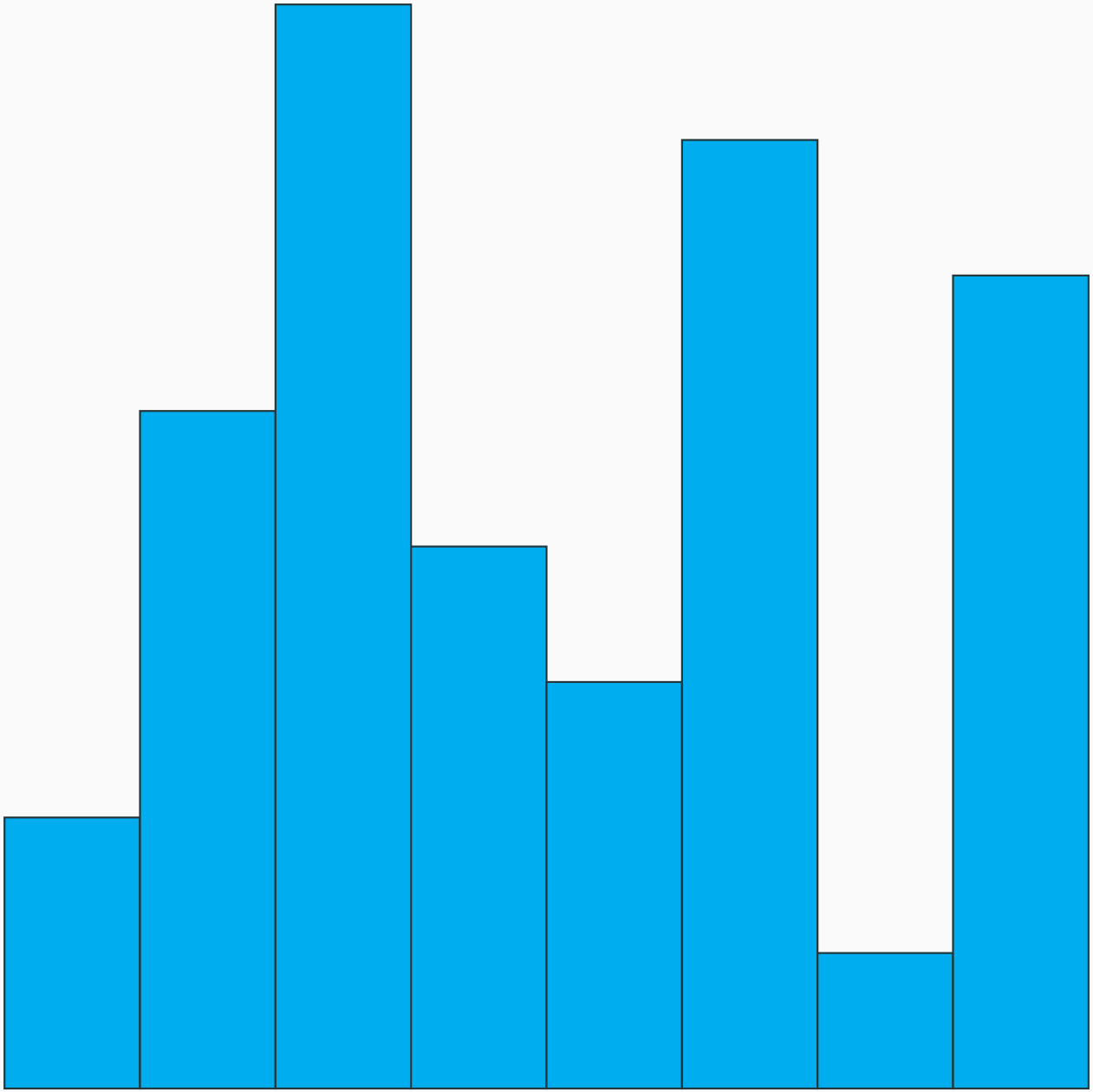
- On commence par chercher le minimum du tableau, puis on le place en première position.
- On recommence ensuite le procédé pour chercher le minimum du reste du tableau, et le placer en deuxième position.
- On réitère ce processus jusqu'à avoir trier le tableau.

Exemple : le tri par sélection

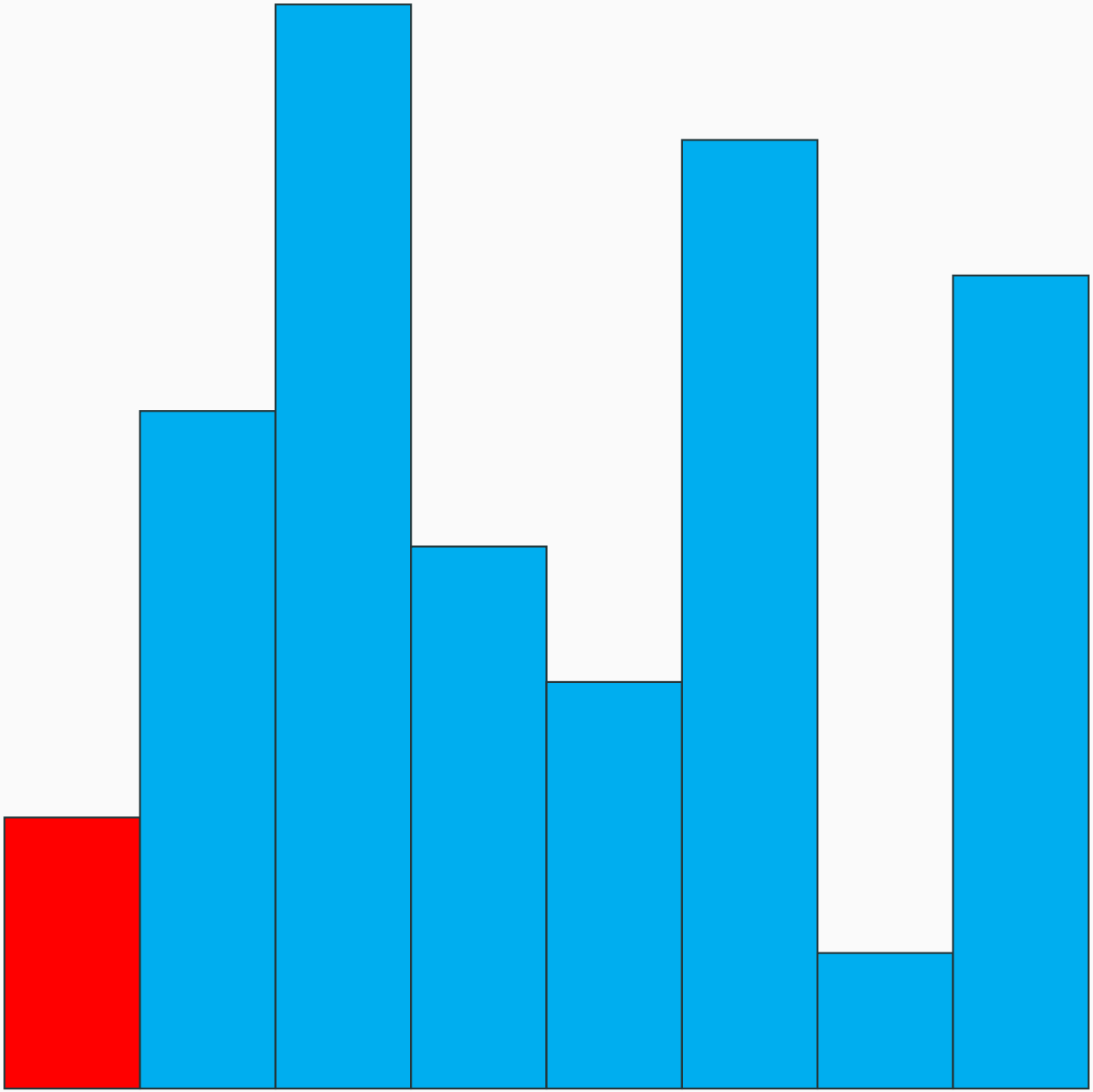
Tri par sélection

```c
void echange(int *tab, int i, int j)
{
  int temp = tab[i];
  tab[i] = tab[j];
  tab[j] = temp;
}

int indice_min(int *tab, int g, int d)
{
  int i_min = g;
  for (int j = g + 1; j <= d; j++)
  {
    if (tab[j] < tab[i_min])
      i_min = j;
  }
  return i_min;
}

void tri_par_selection(int *tab, int n)
{
  for (int i = 0; i < n; i++)
  {
    int i_min = indice_min(tab, i, n-1);
    echange(tab, i, i_min);
  }
}
```

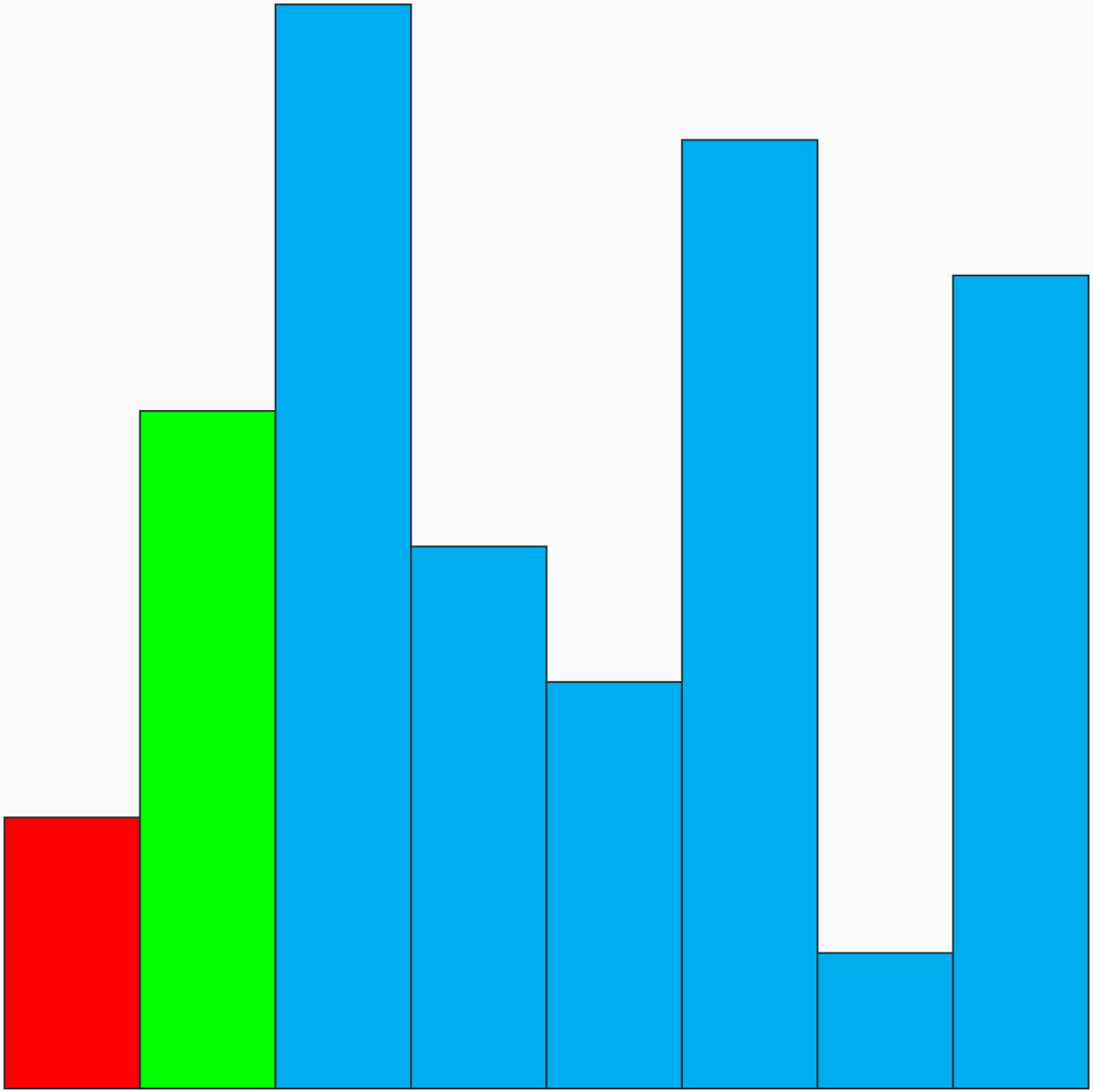
Exemple



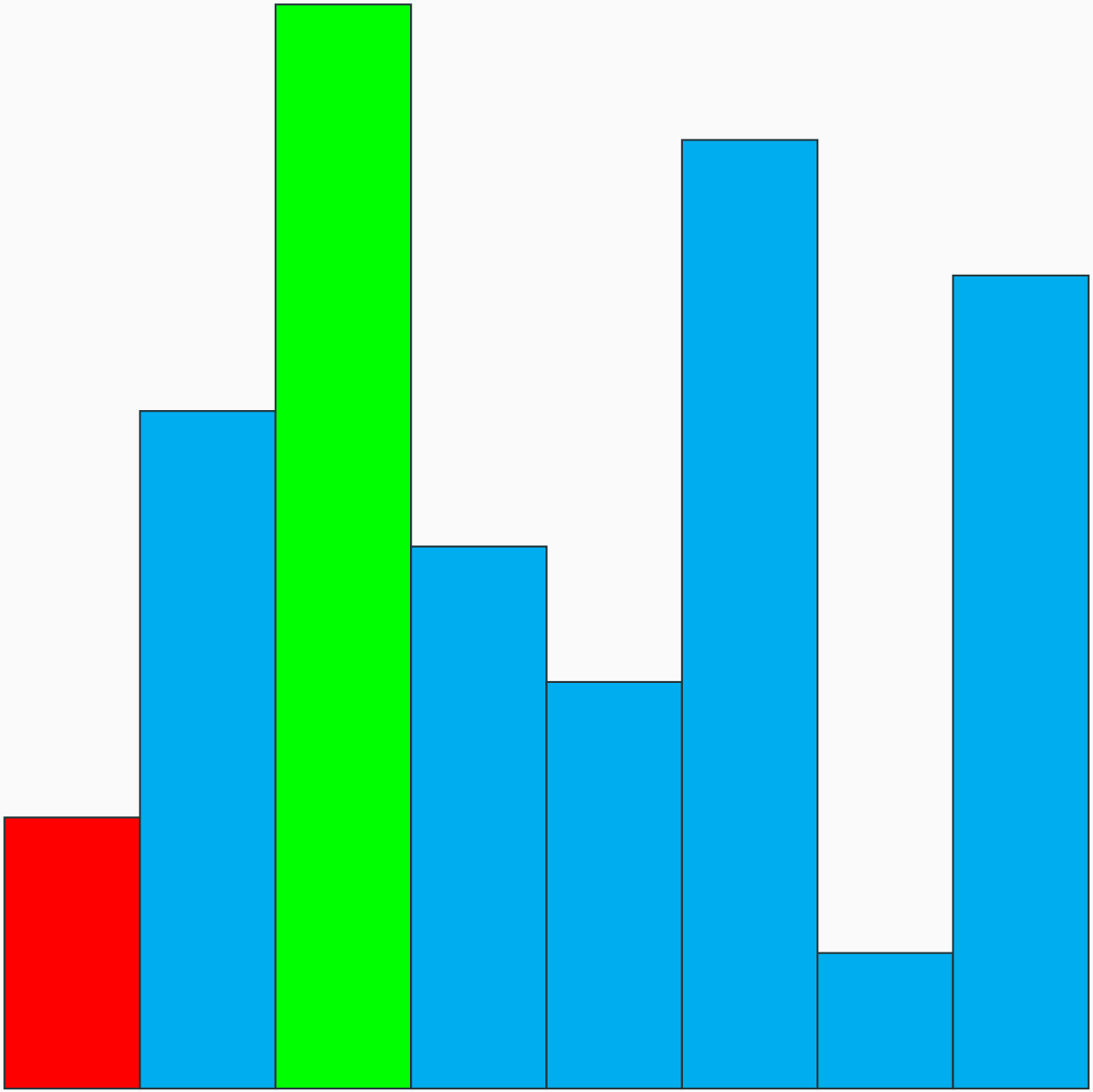
Exemple



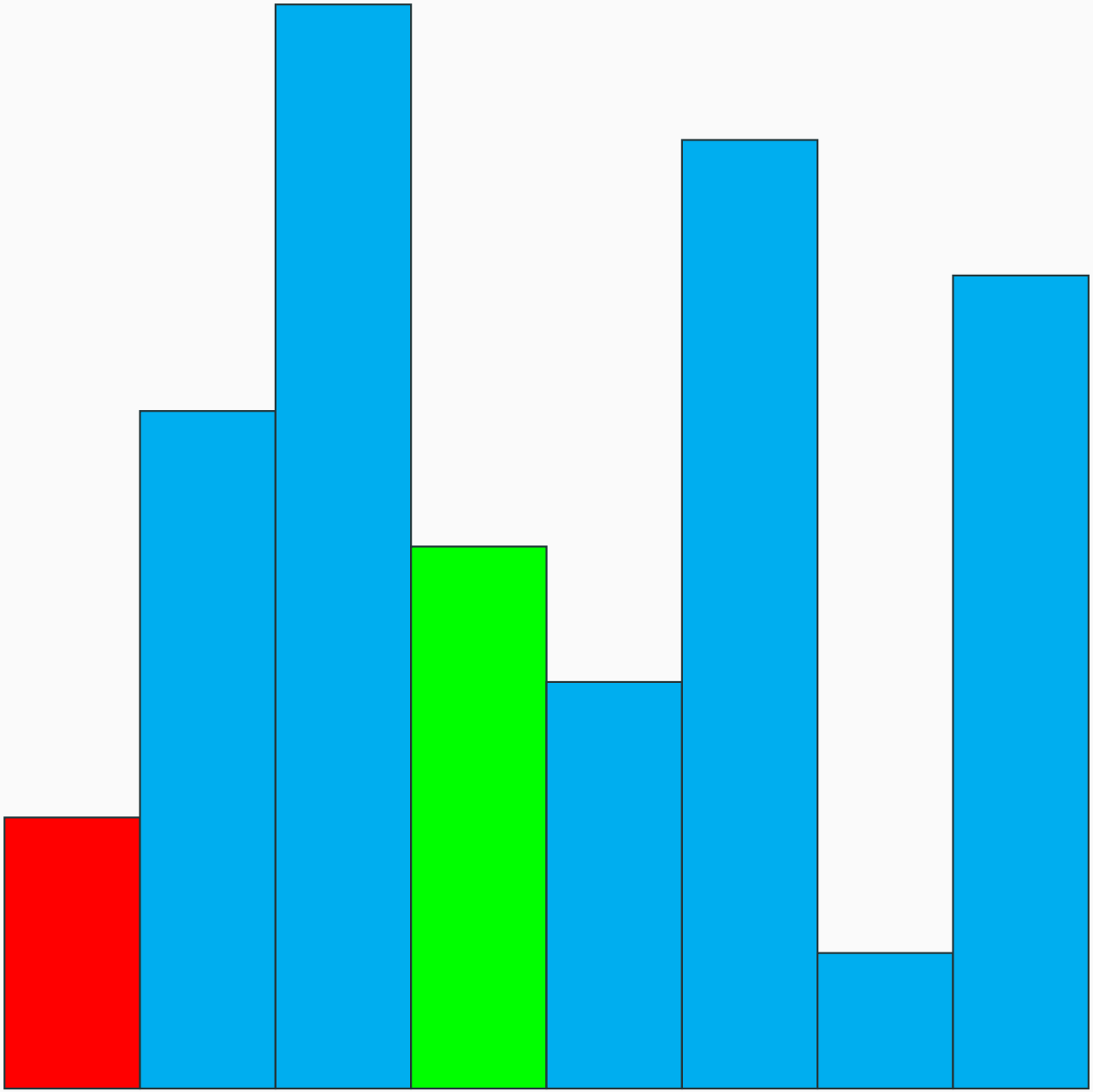
Exemple



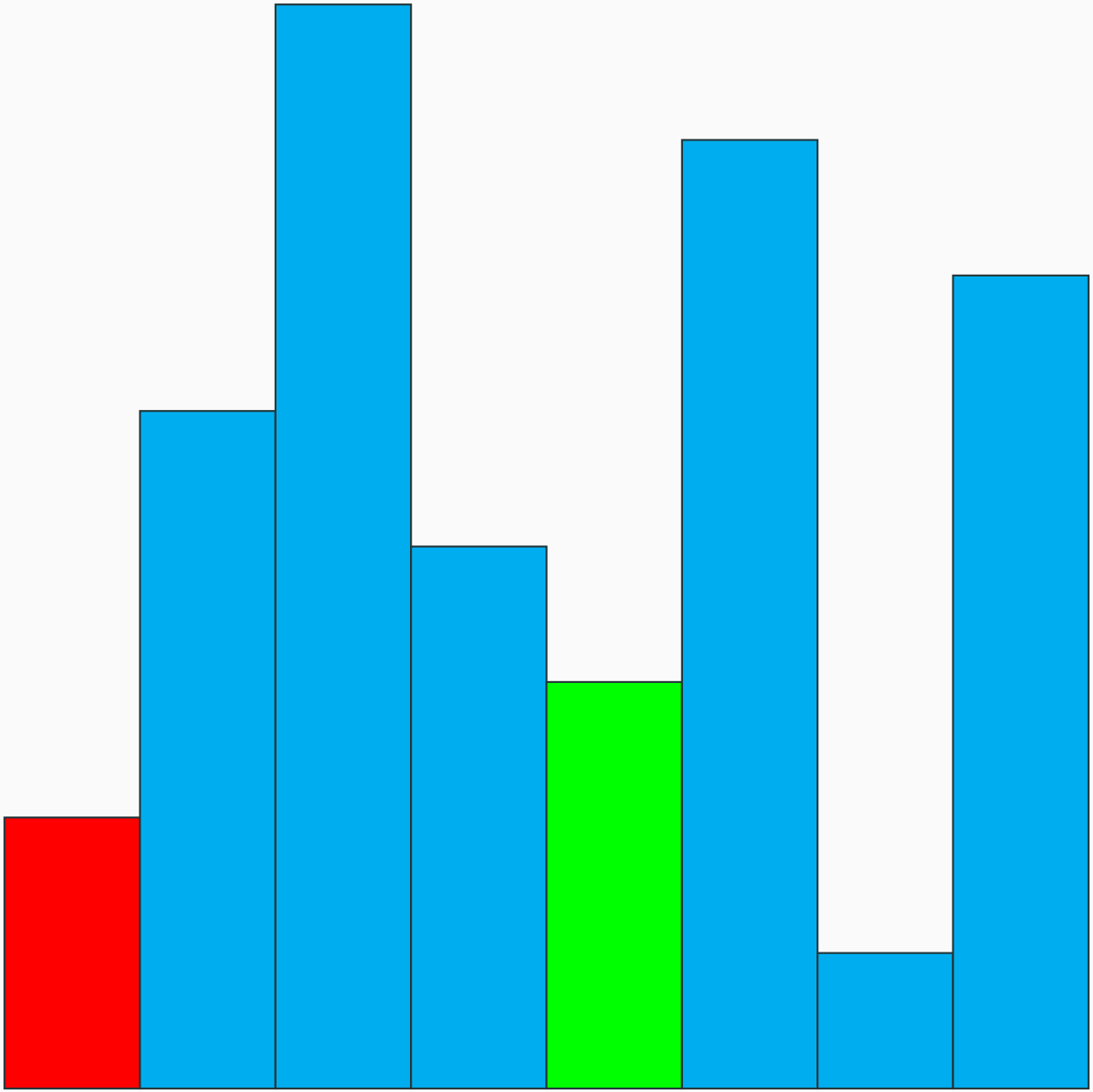
Exemple



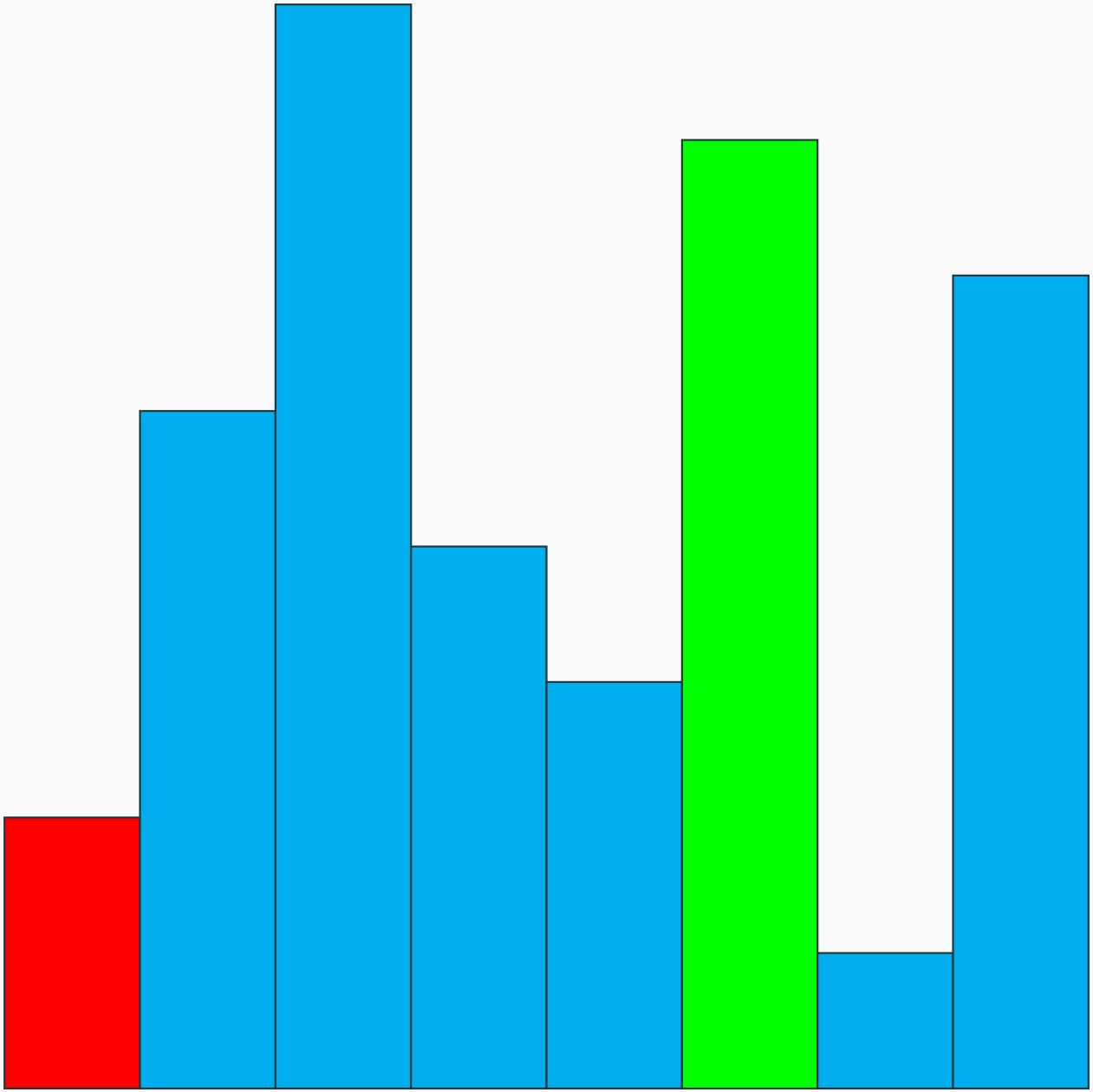
Exemple



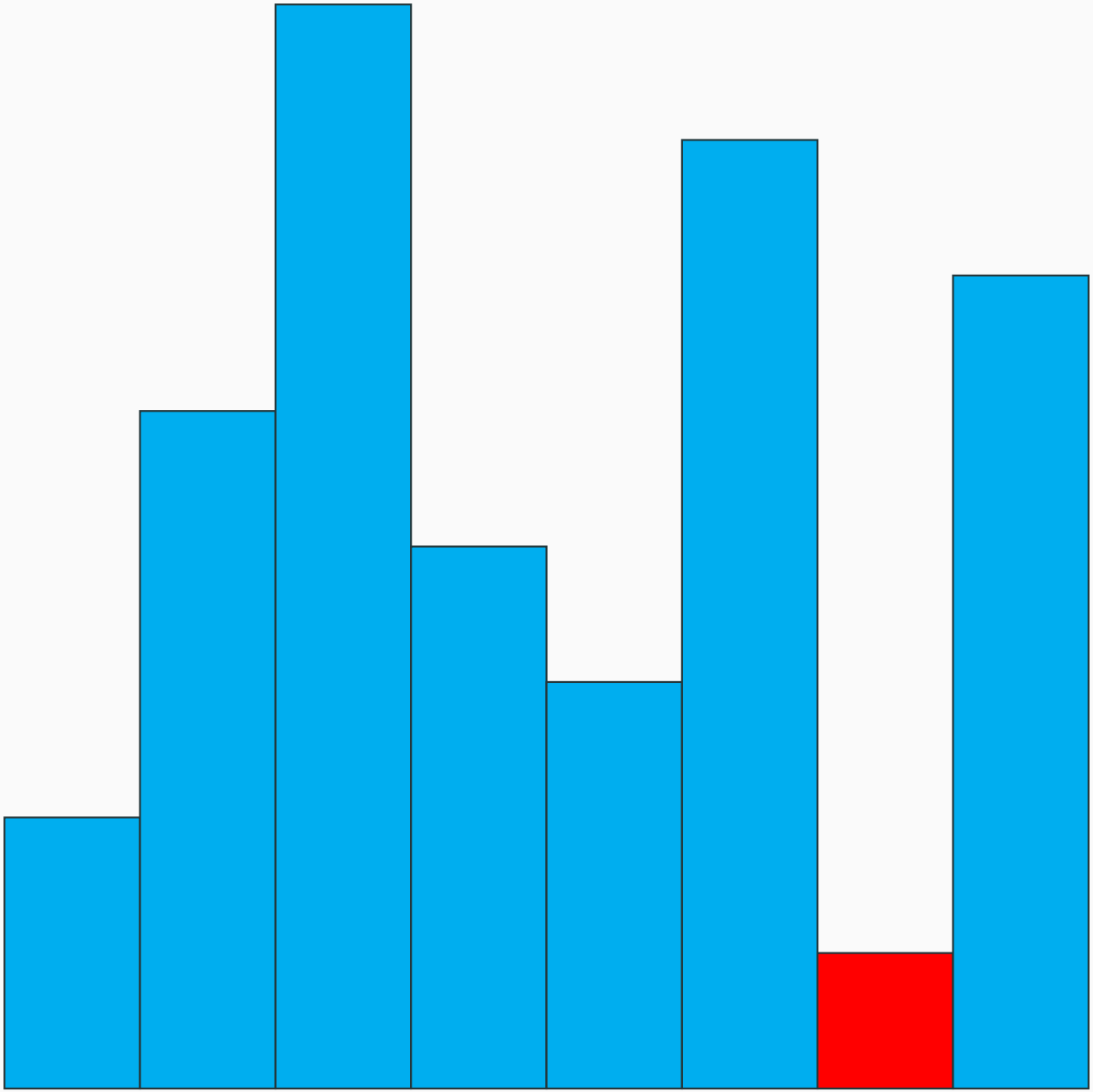
Exemple



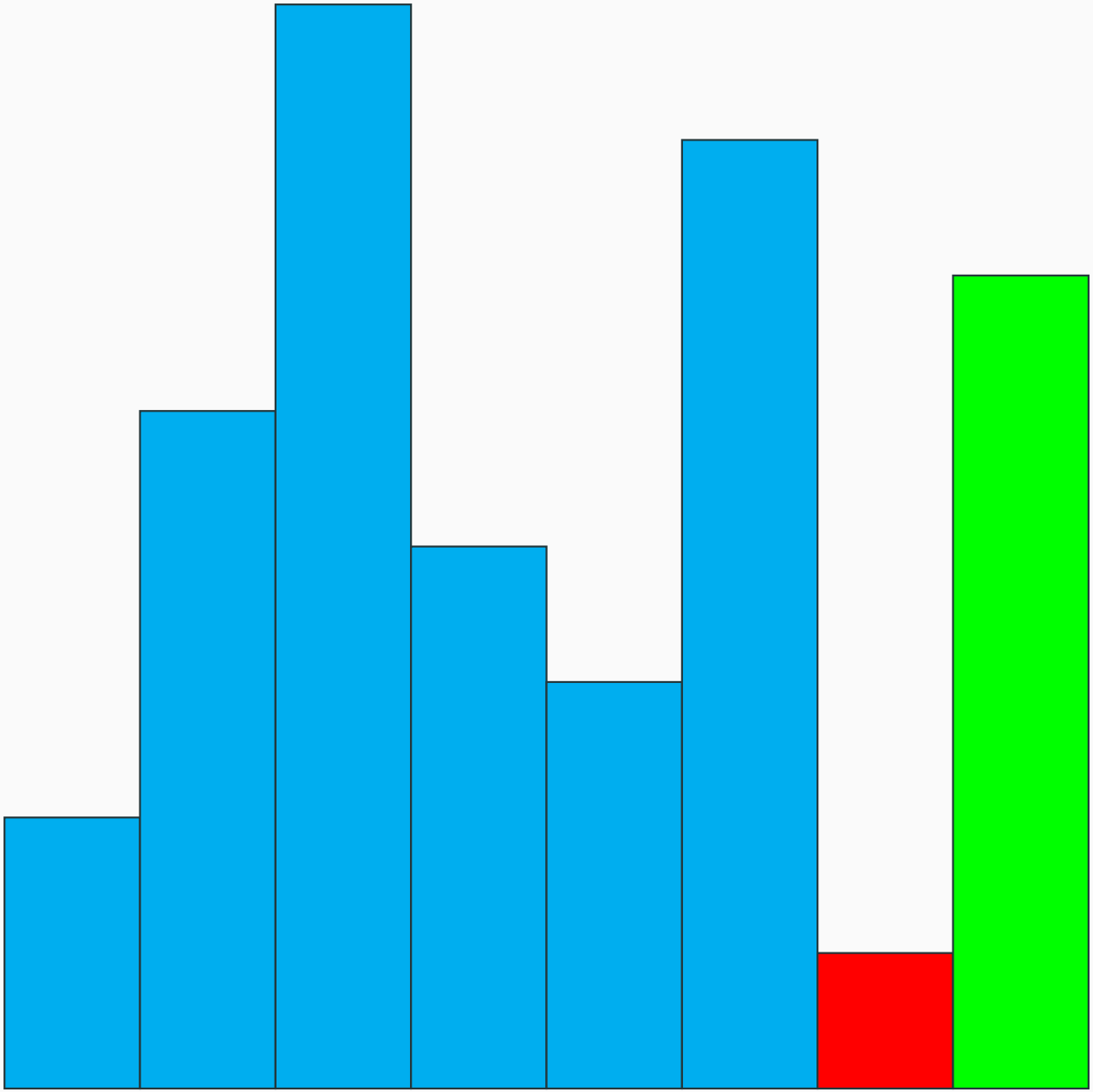
Exemple



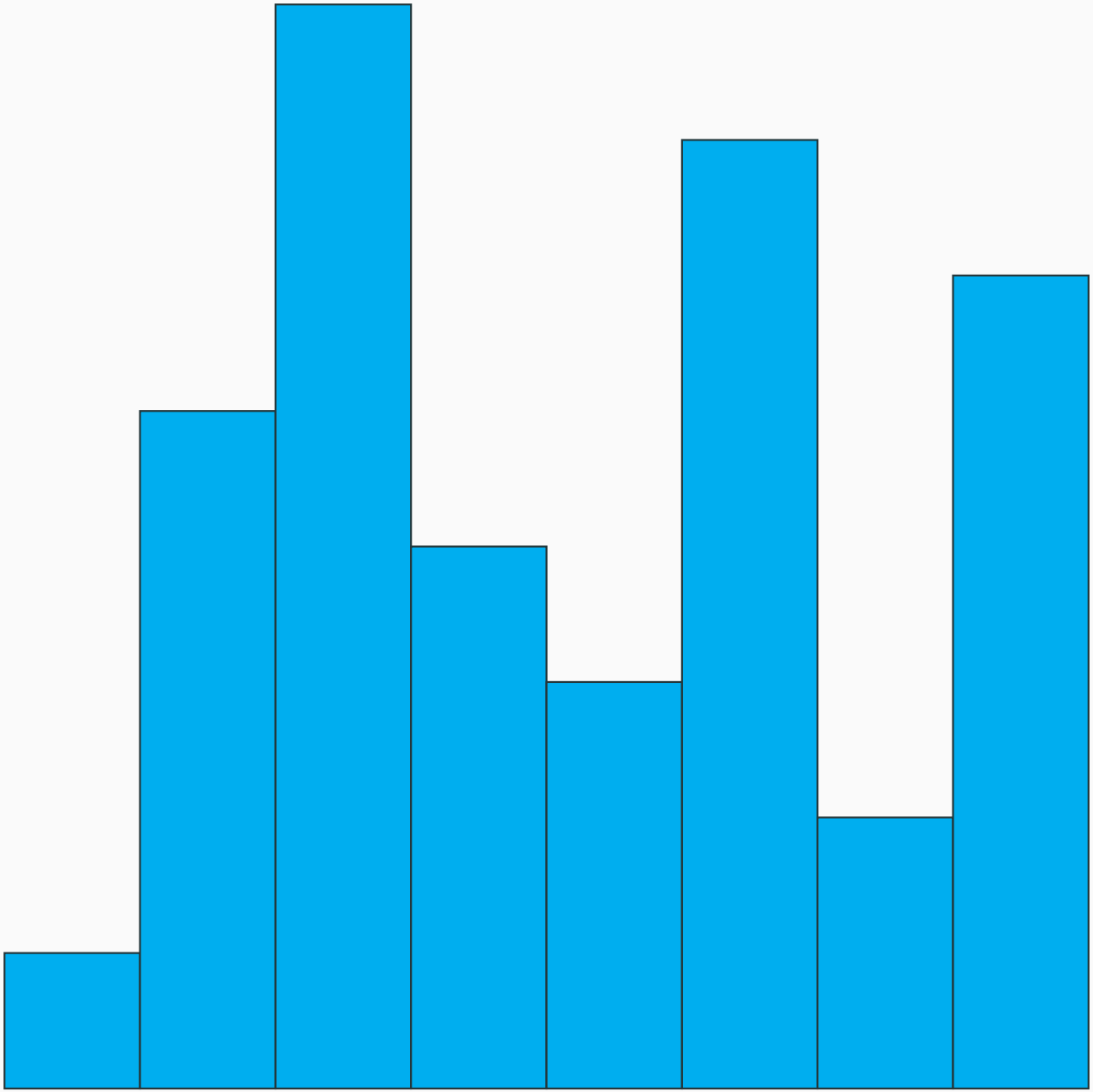
Exemple



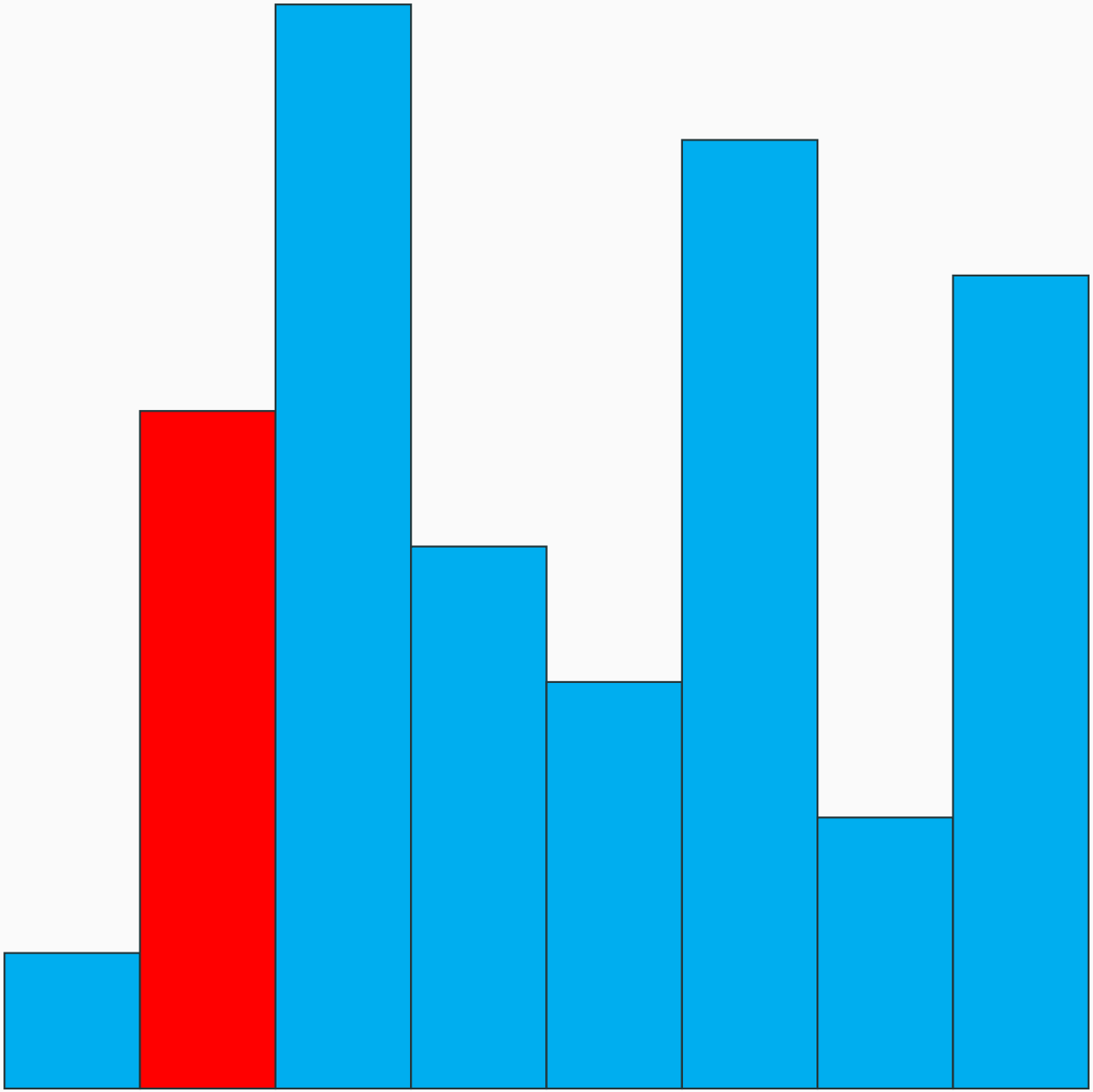
Exemple



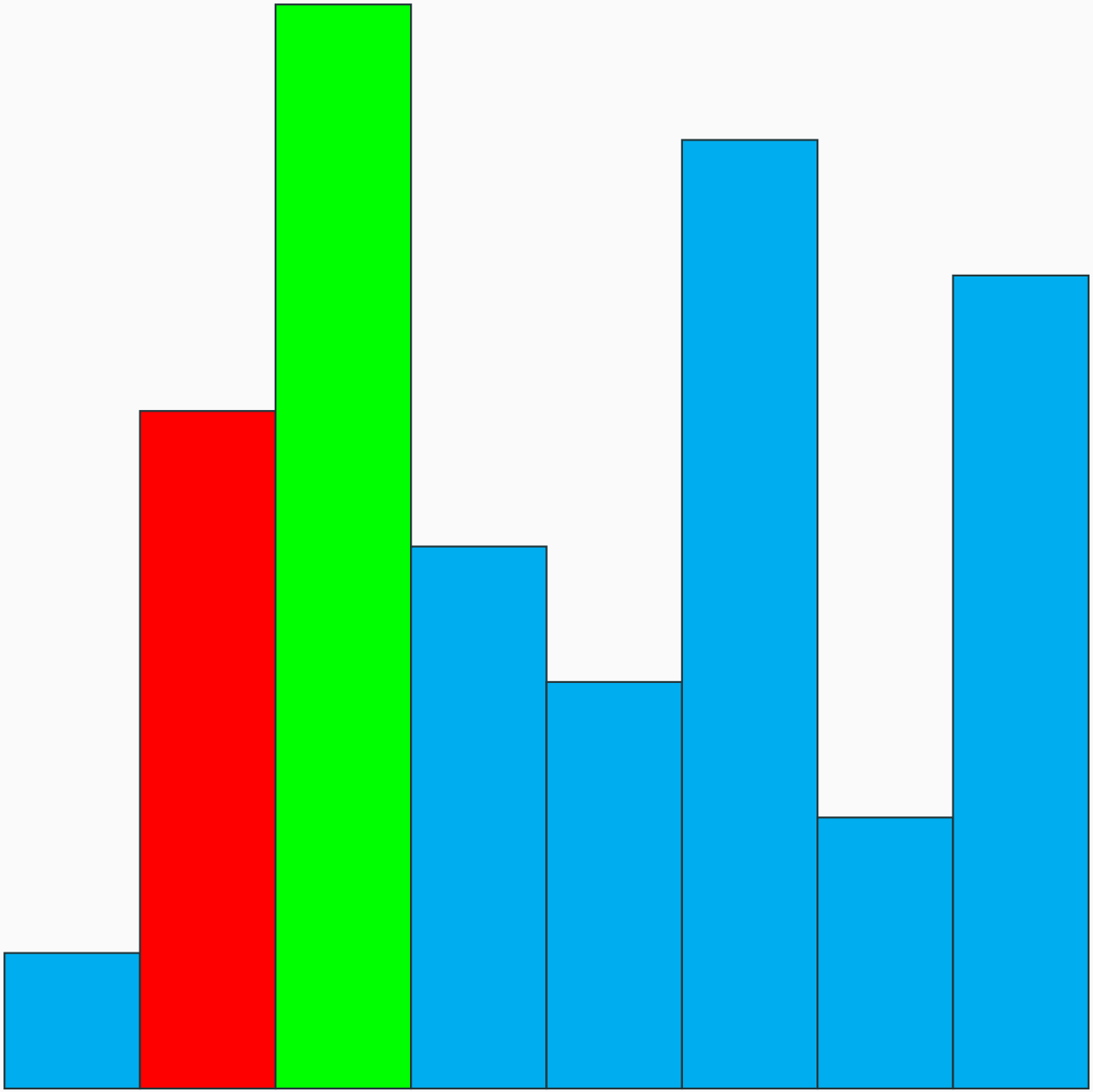
Exemple



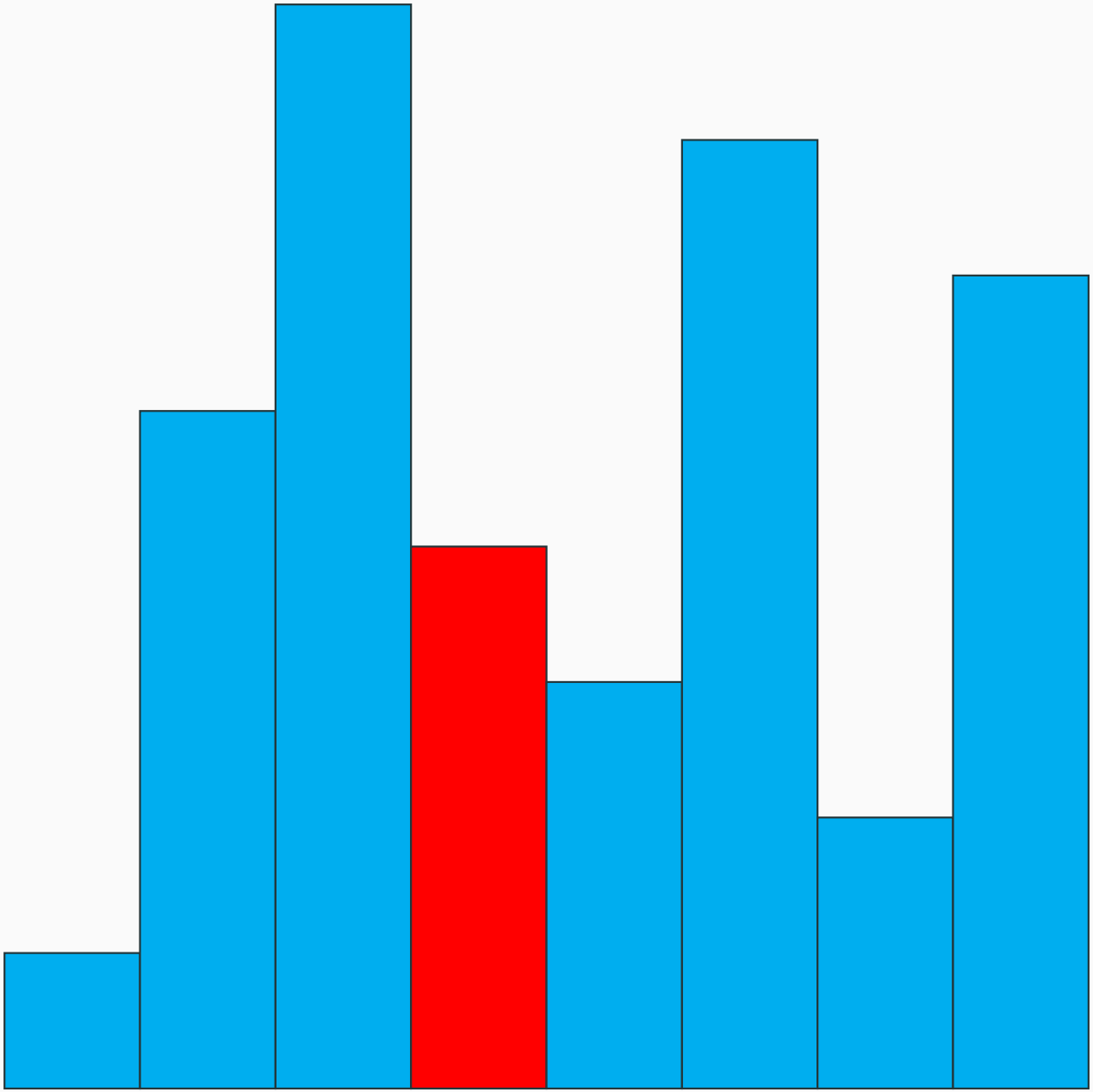
Exemple



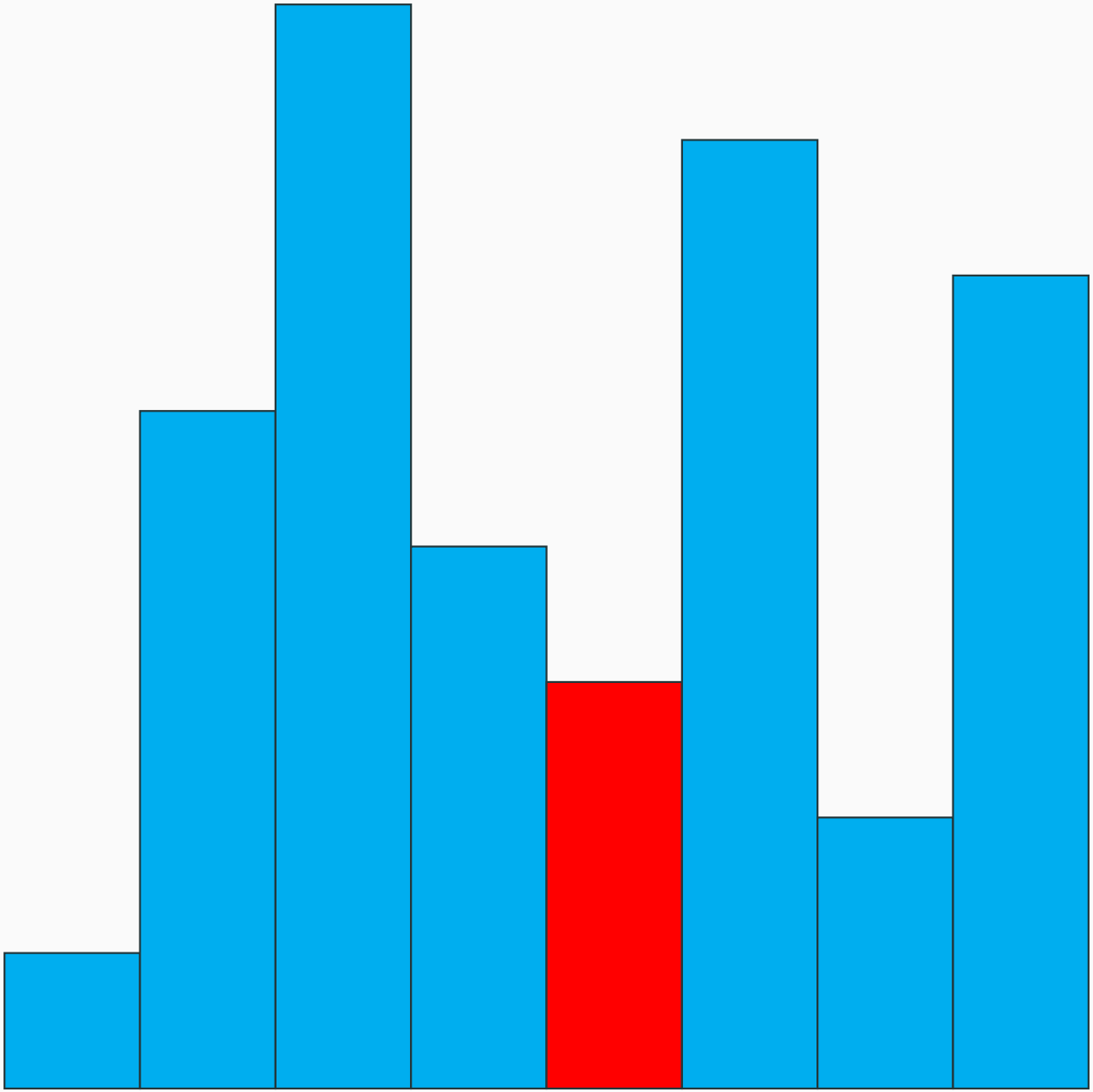
Exemple



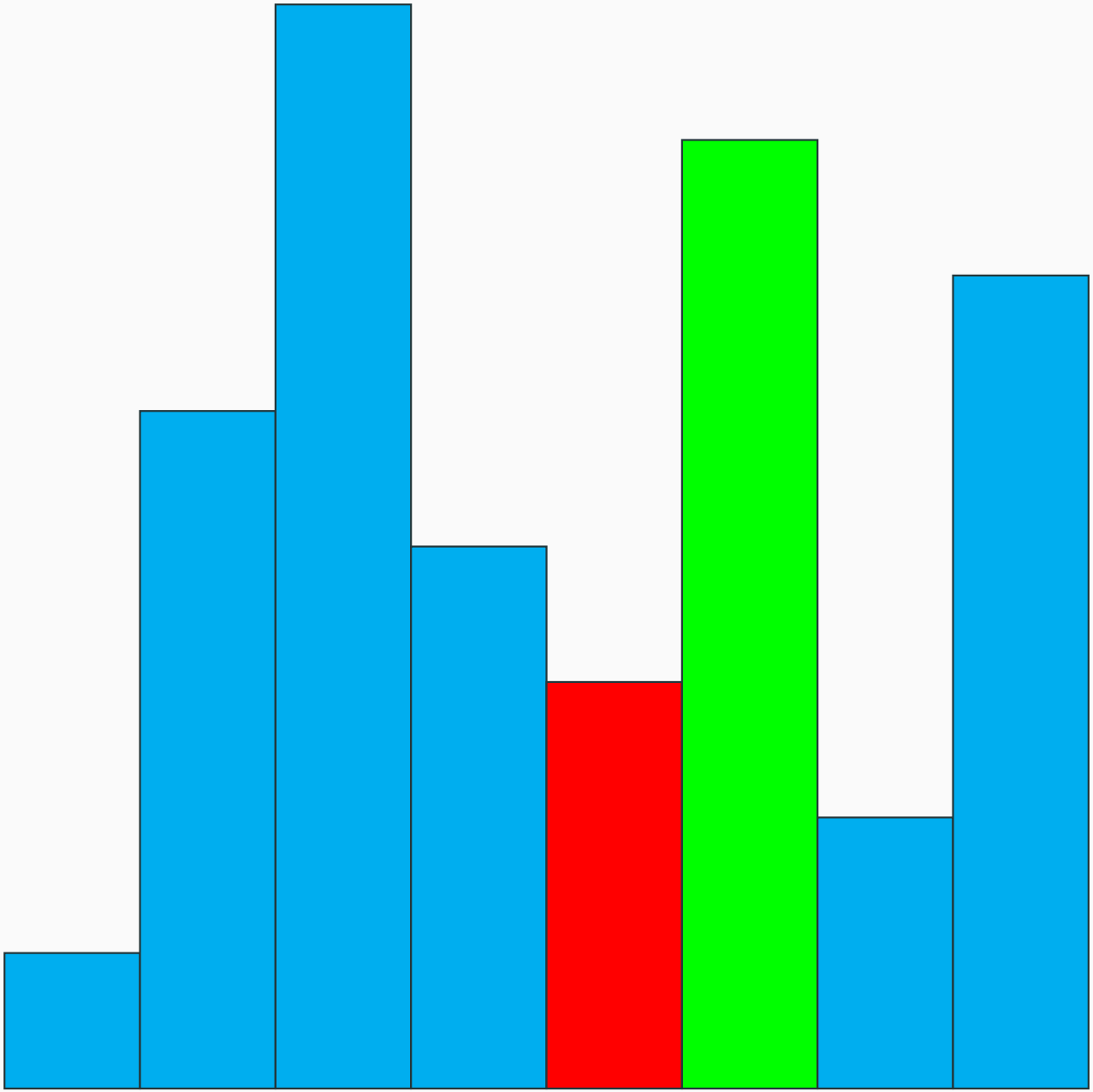
Exemple



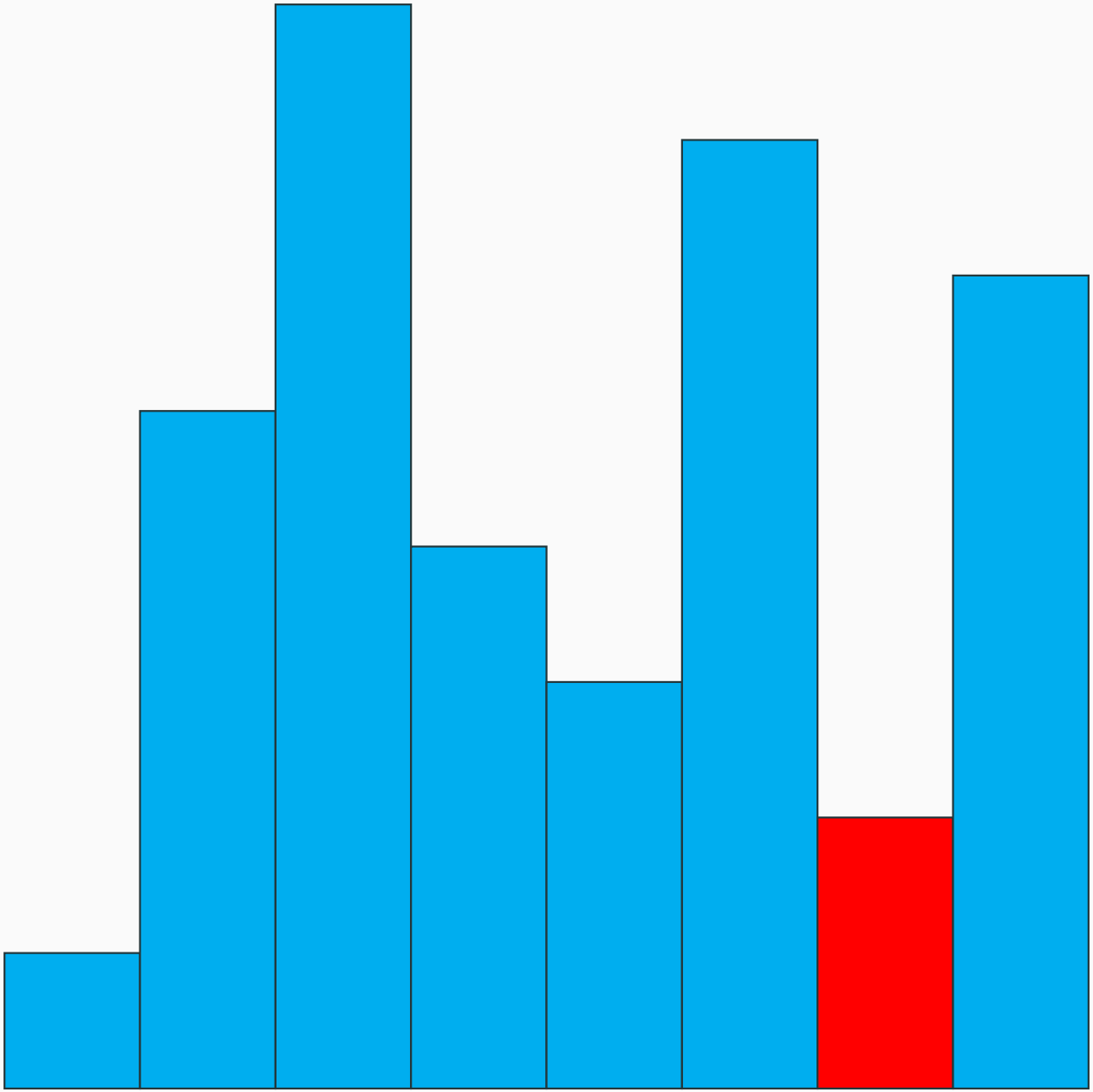
Exemple



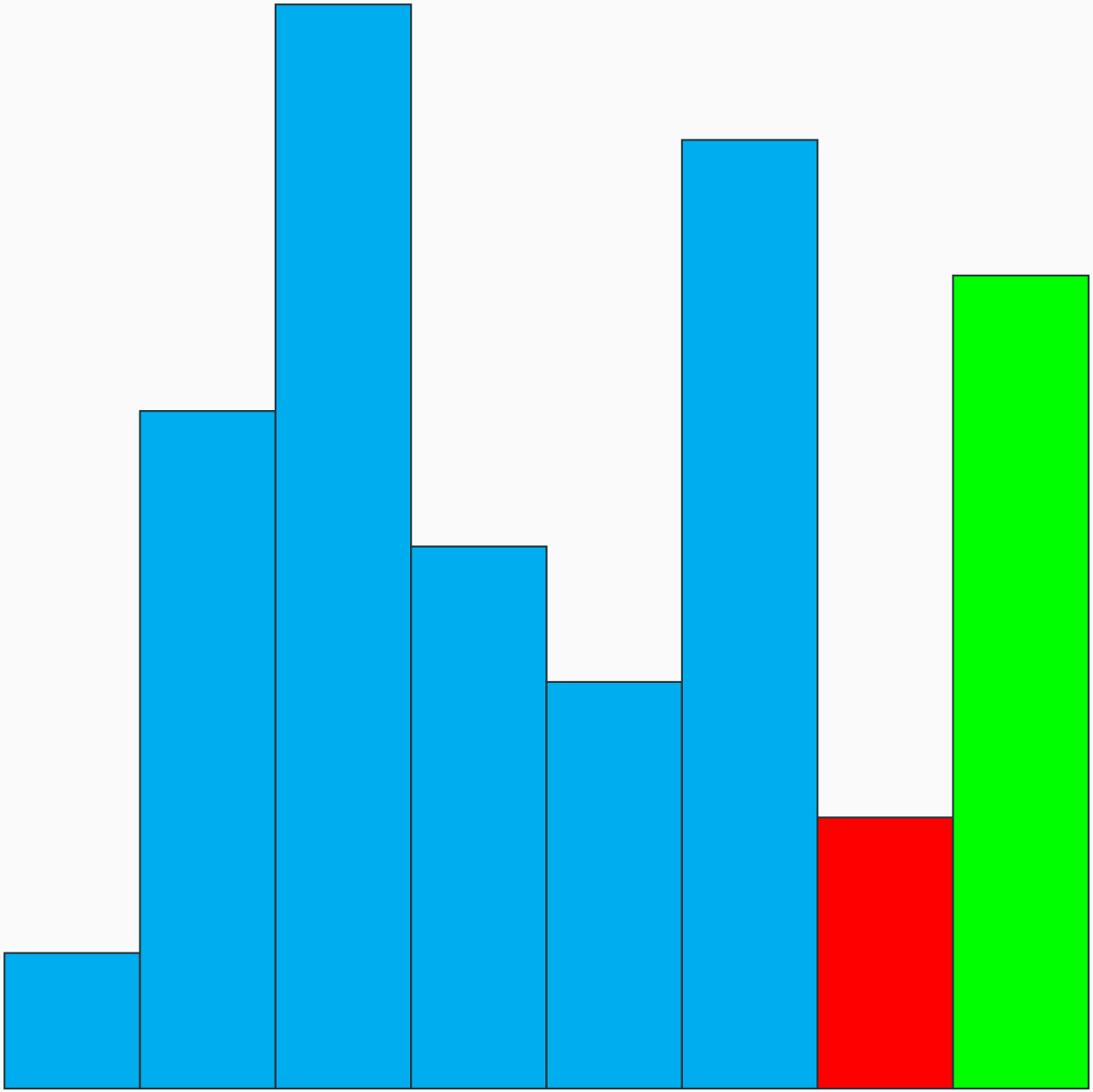
Exemple



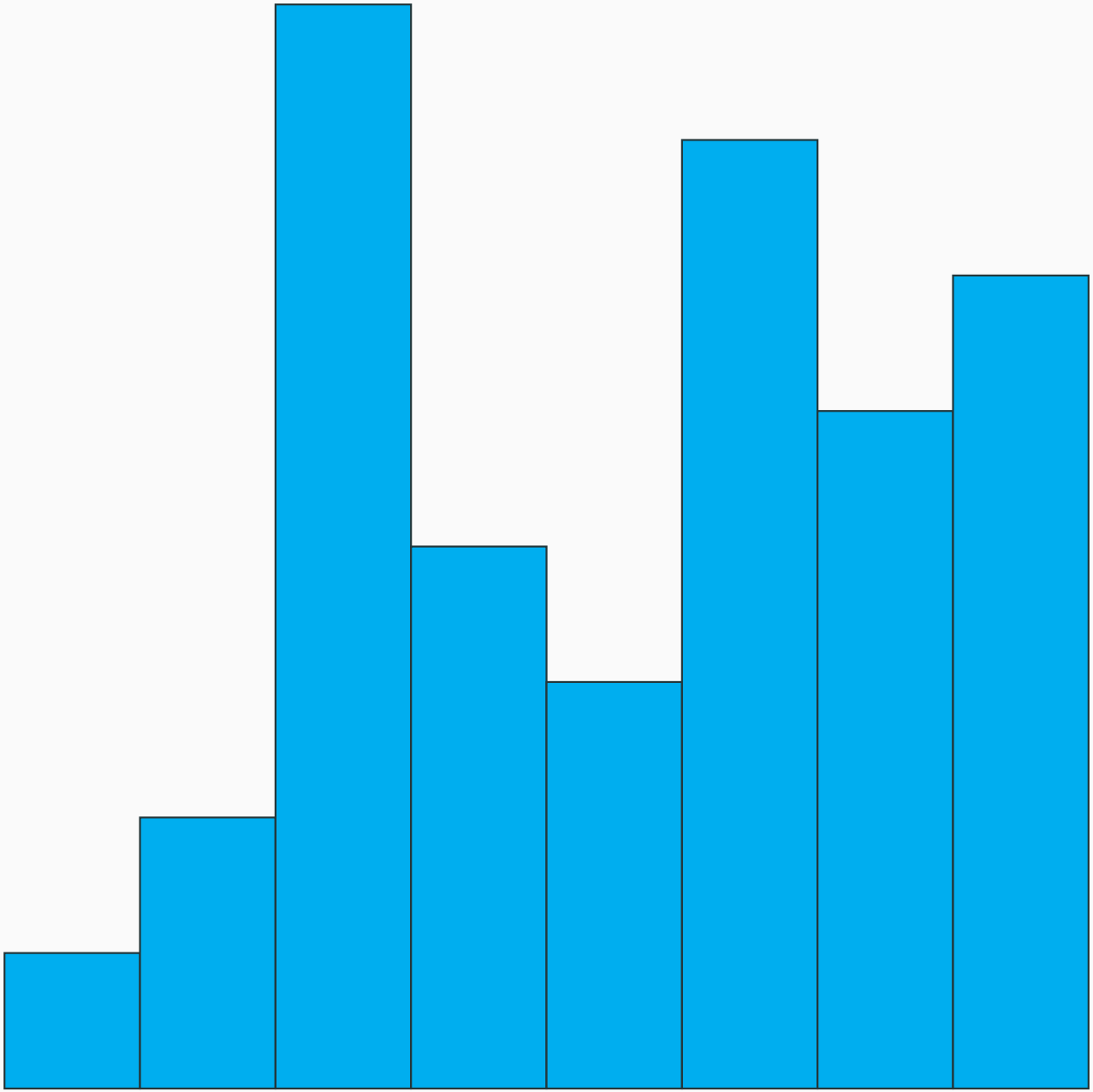
Exemple



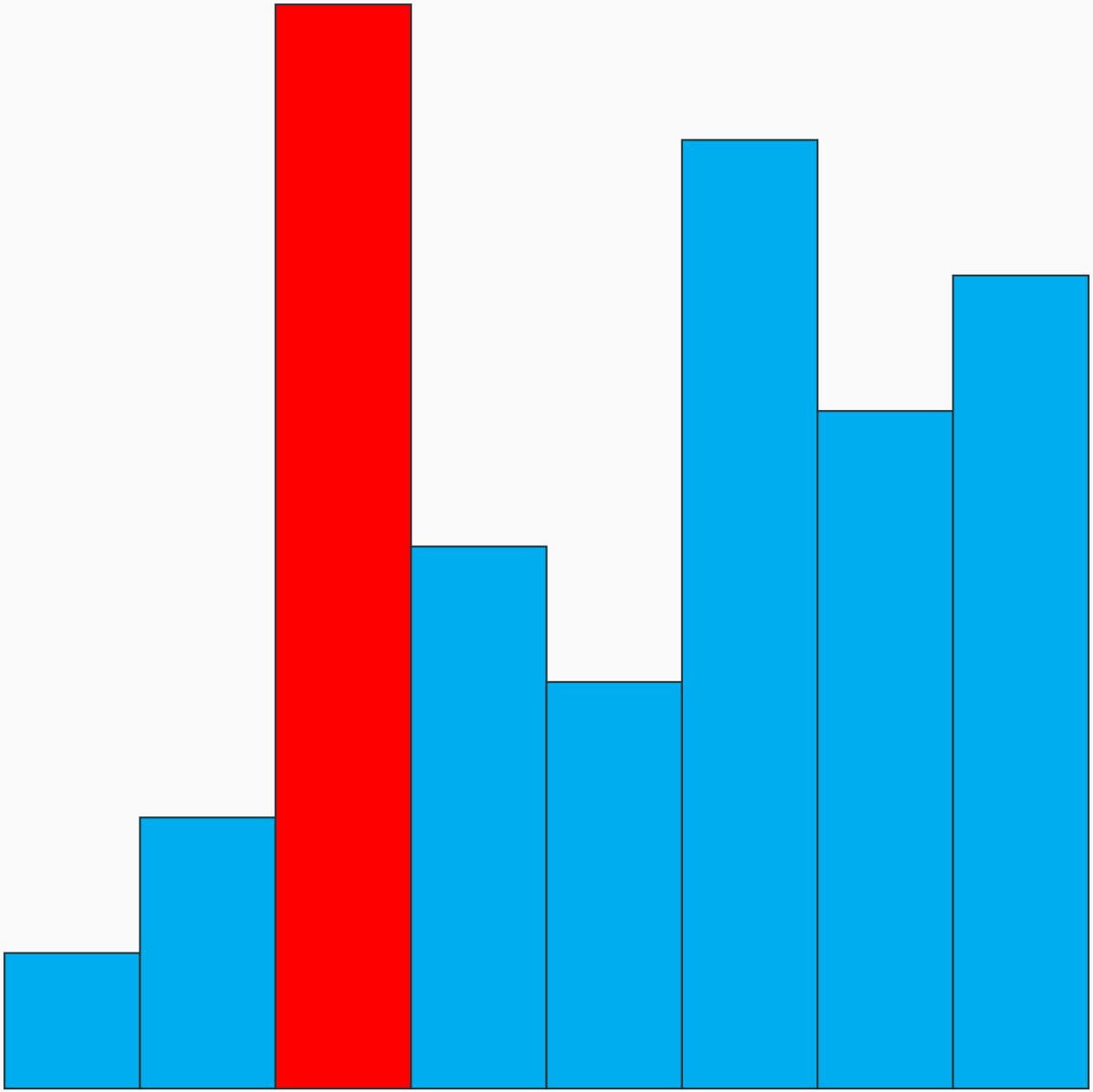
Exemple



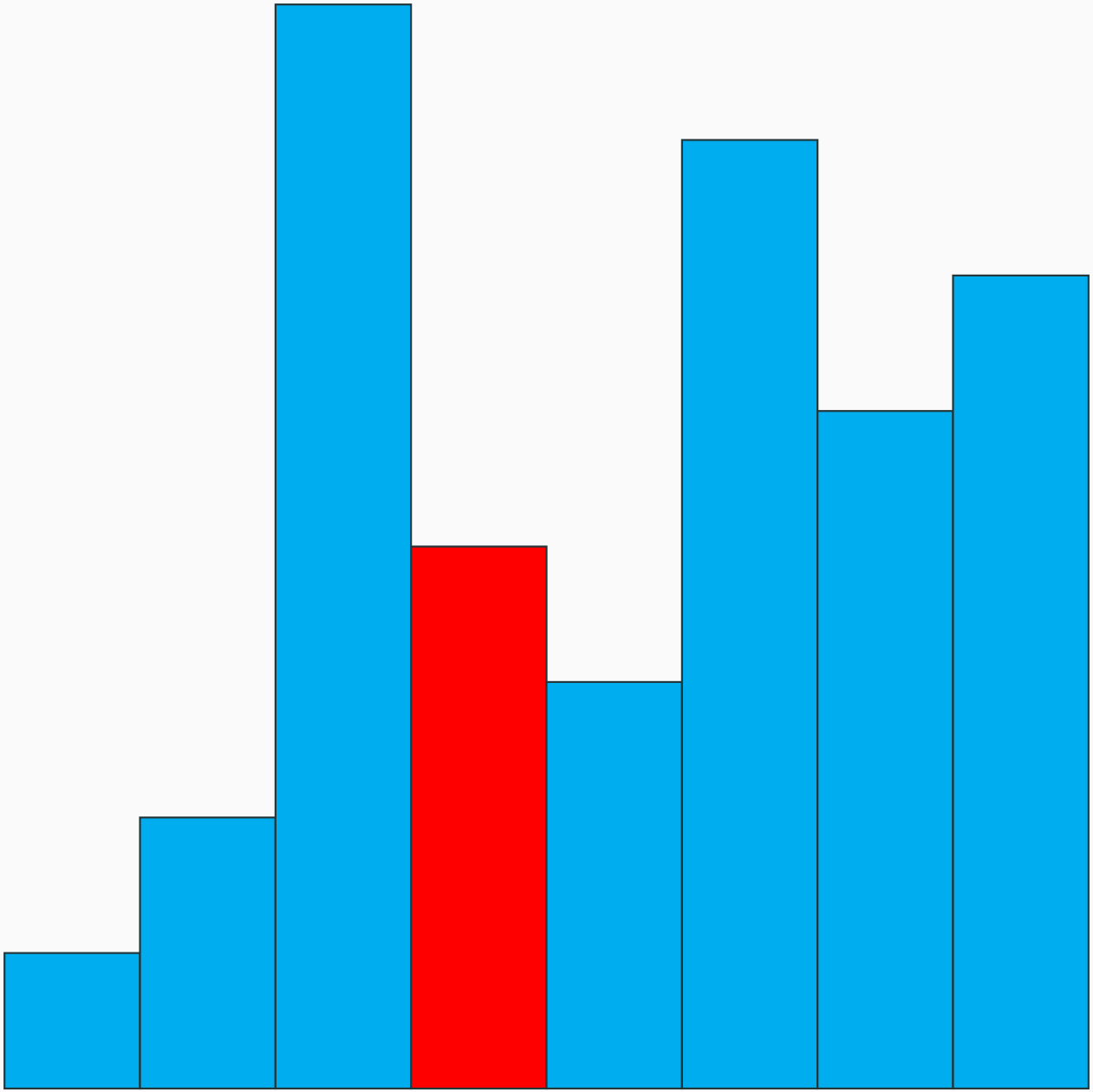
Exemple



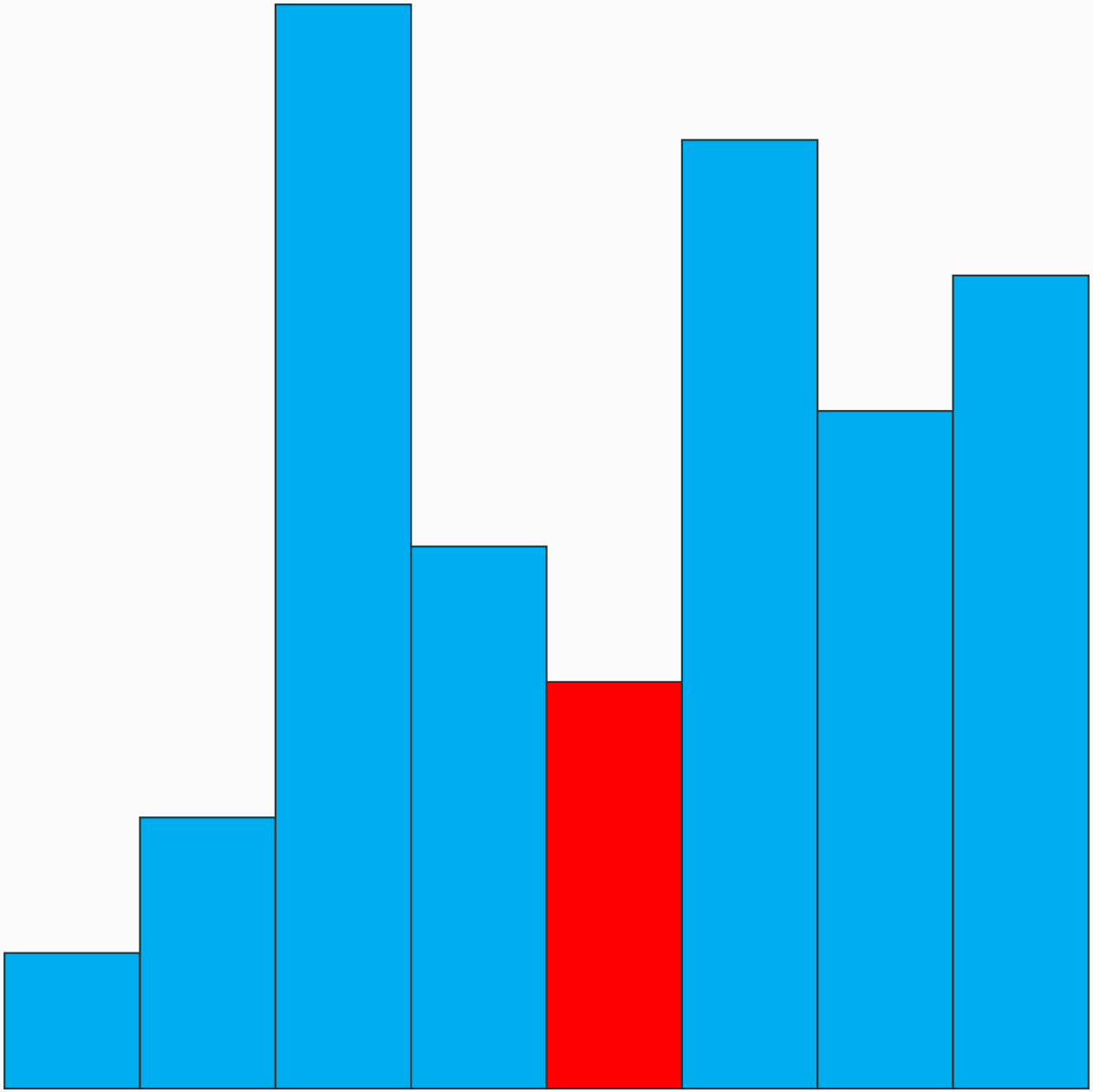
Exemple



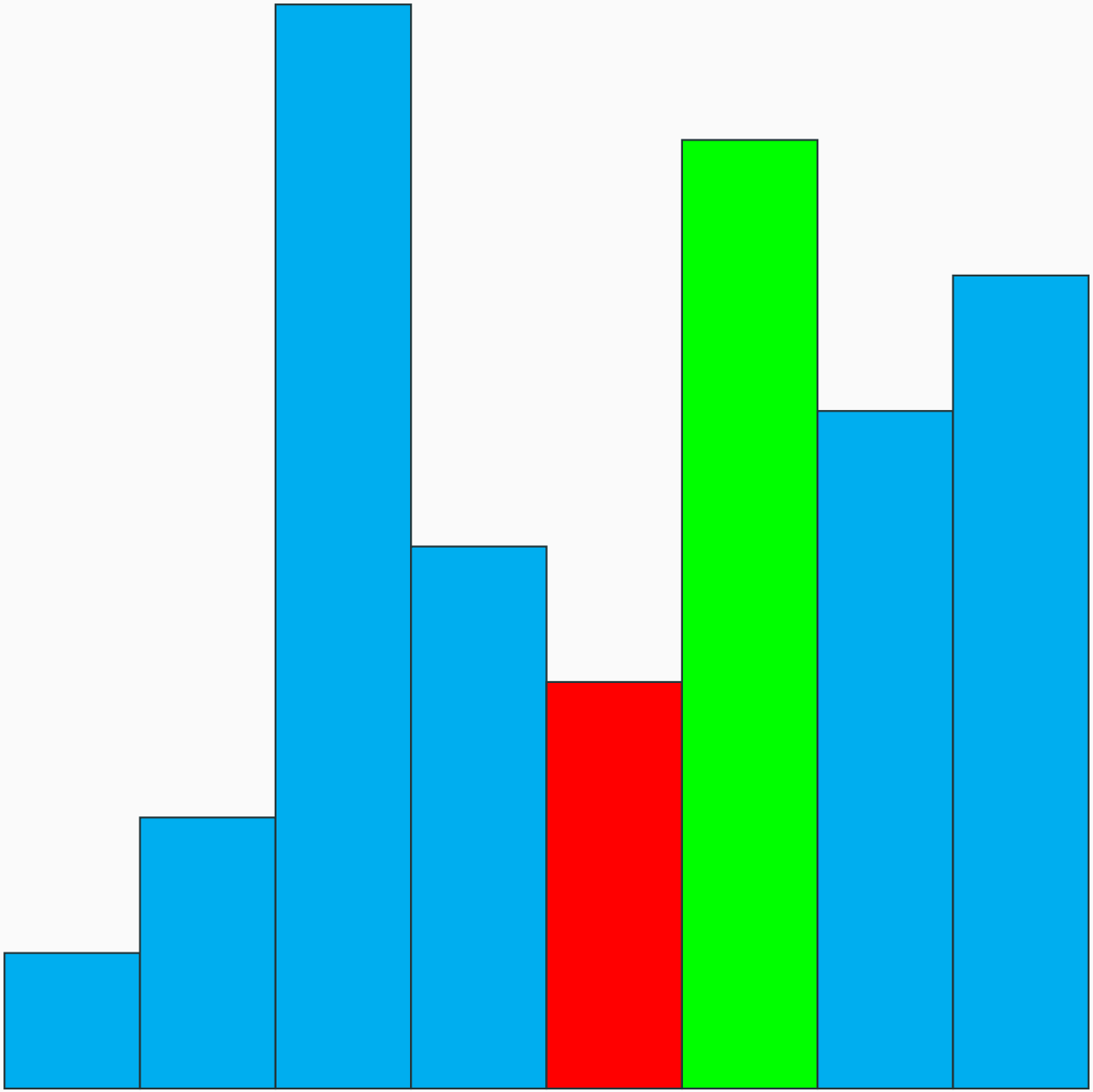
Exemple



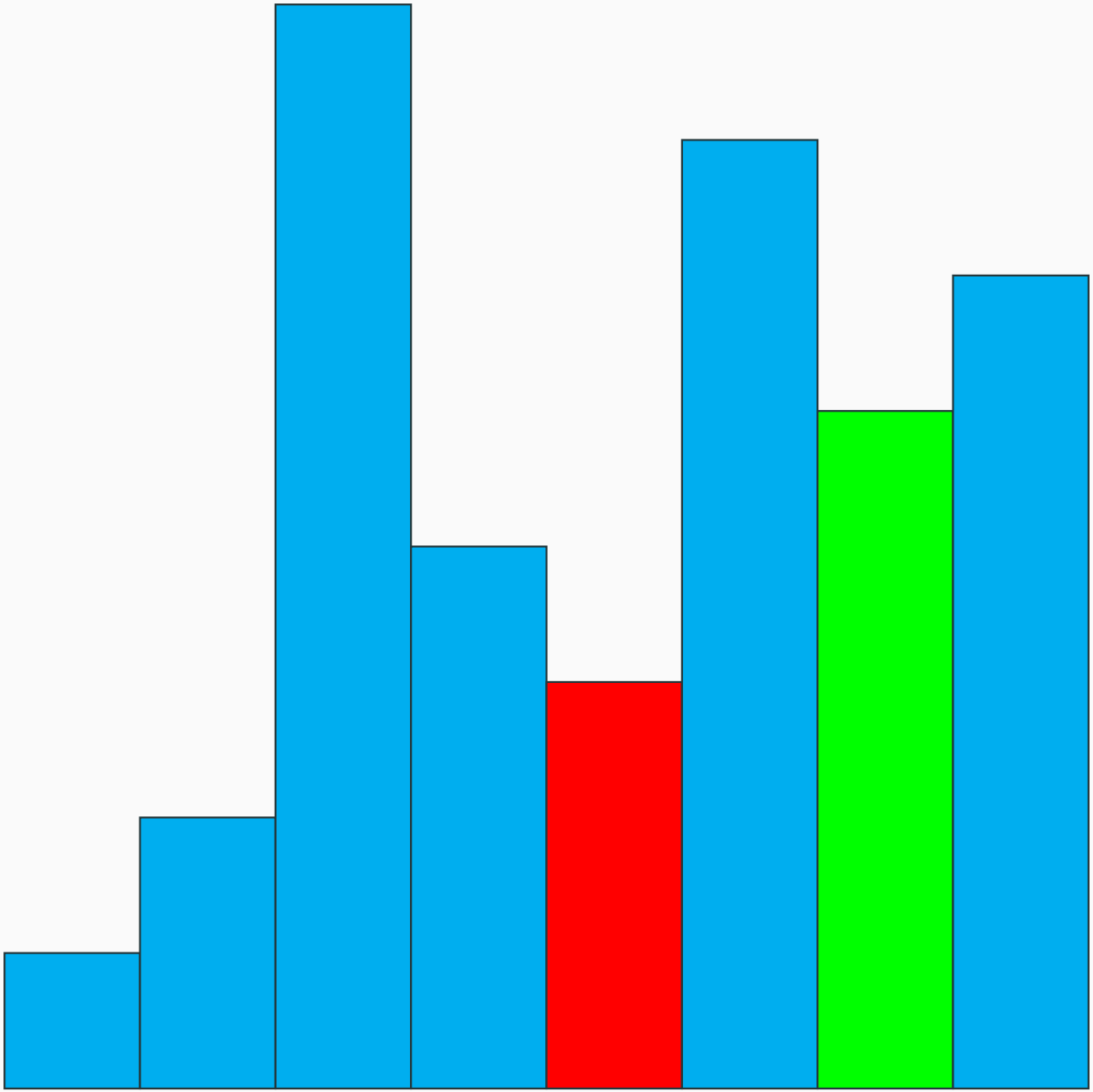
Exemple



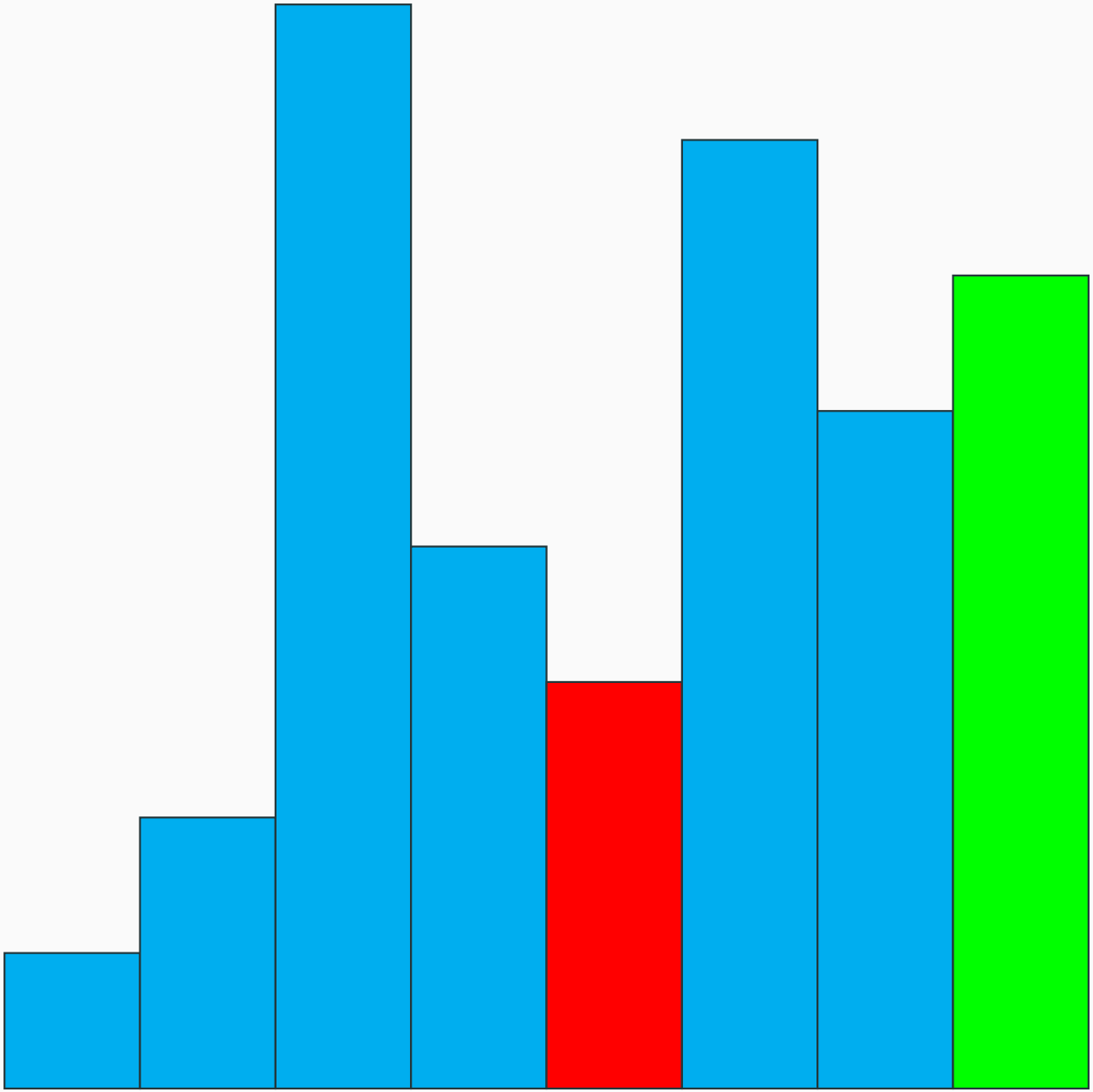
Exemple



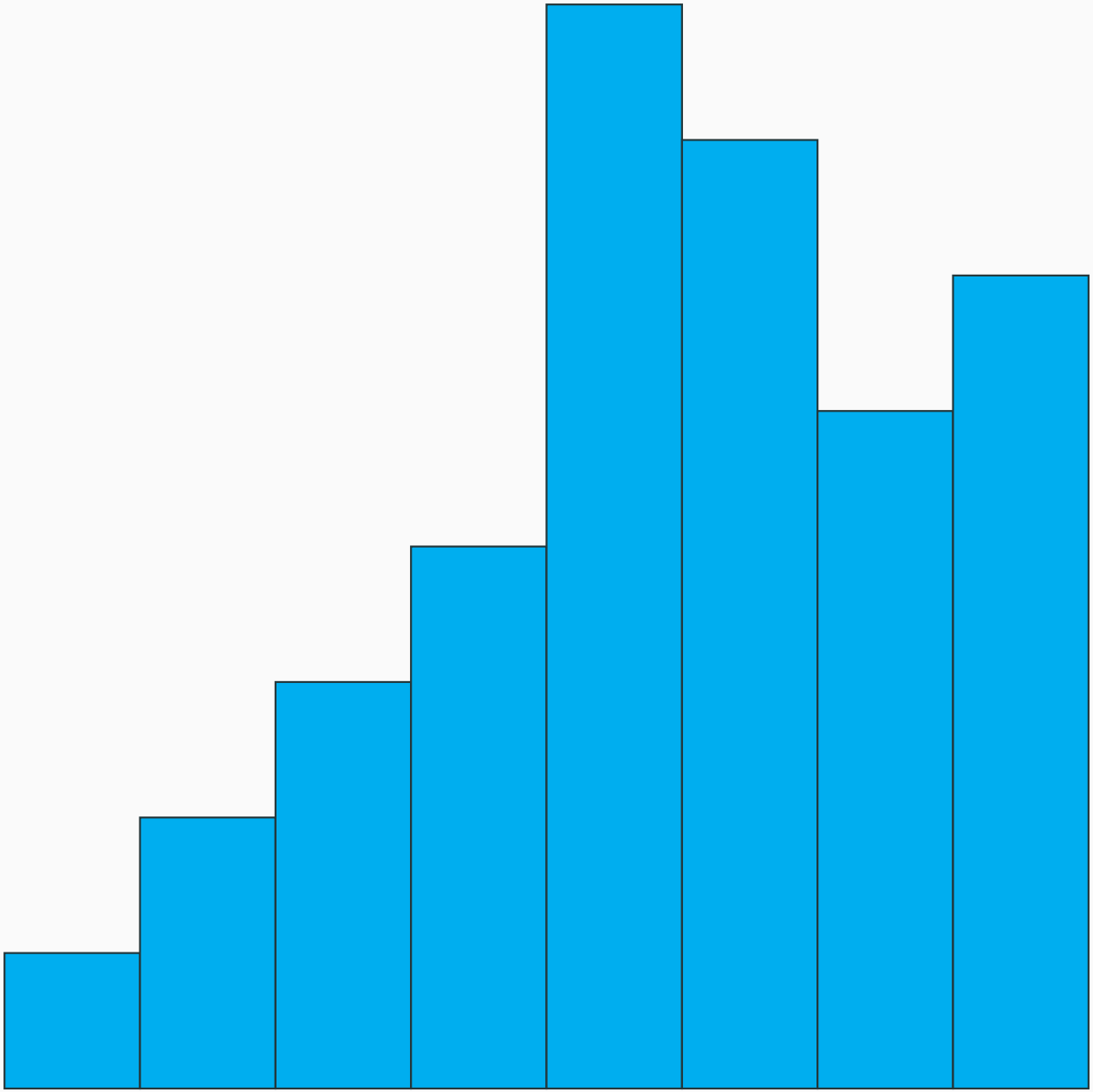
Exemple



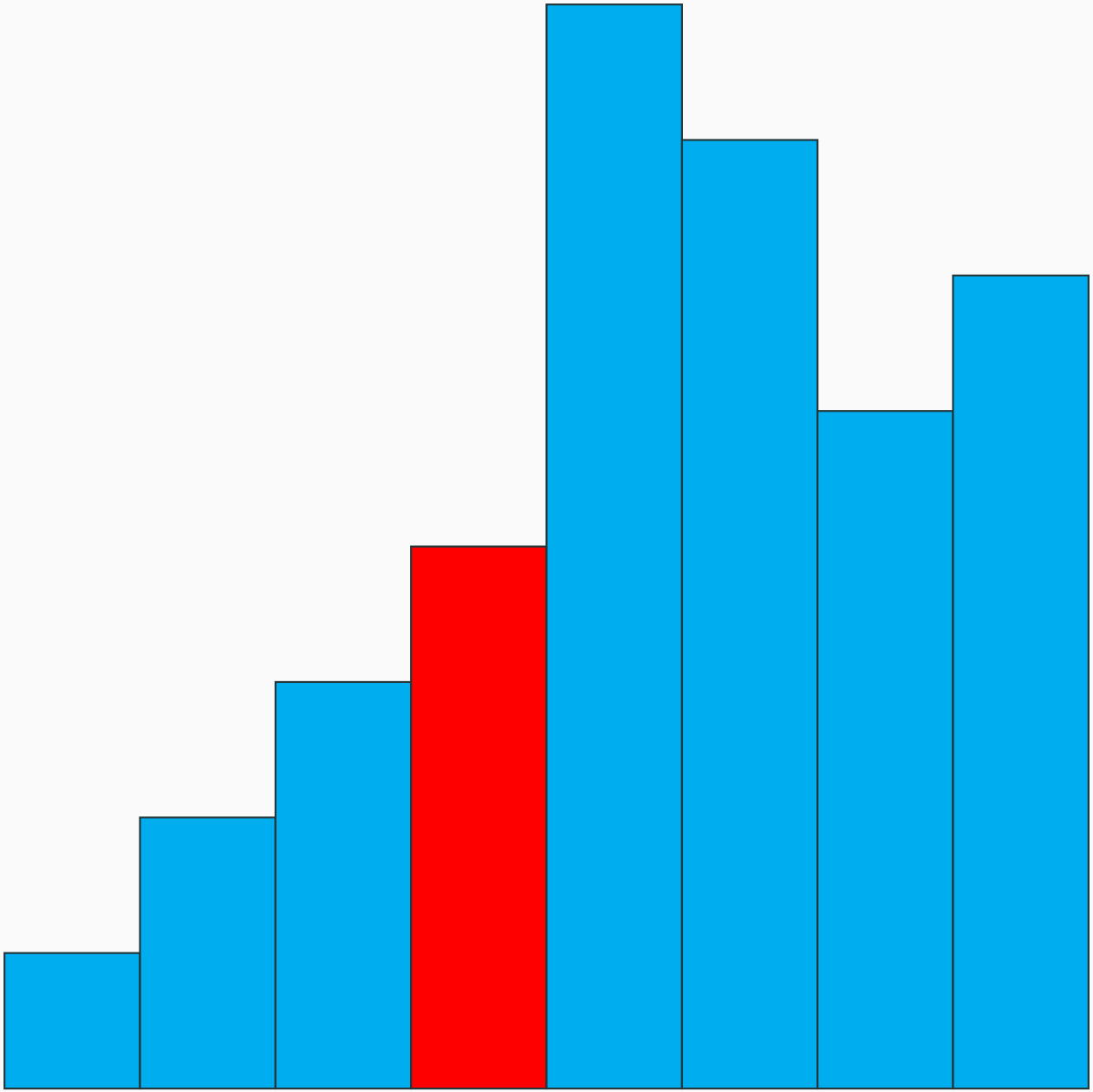
Exemple



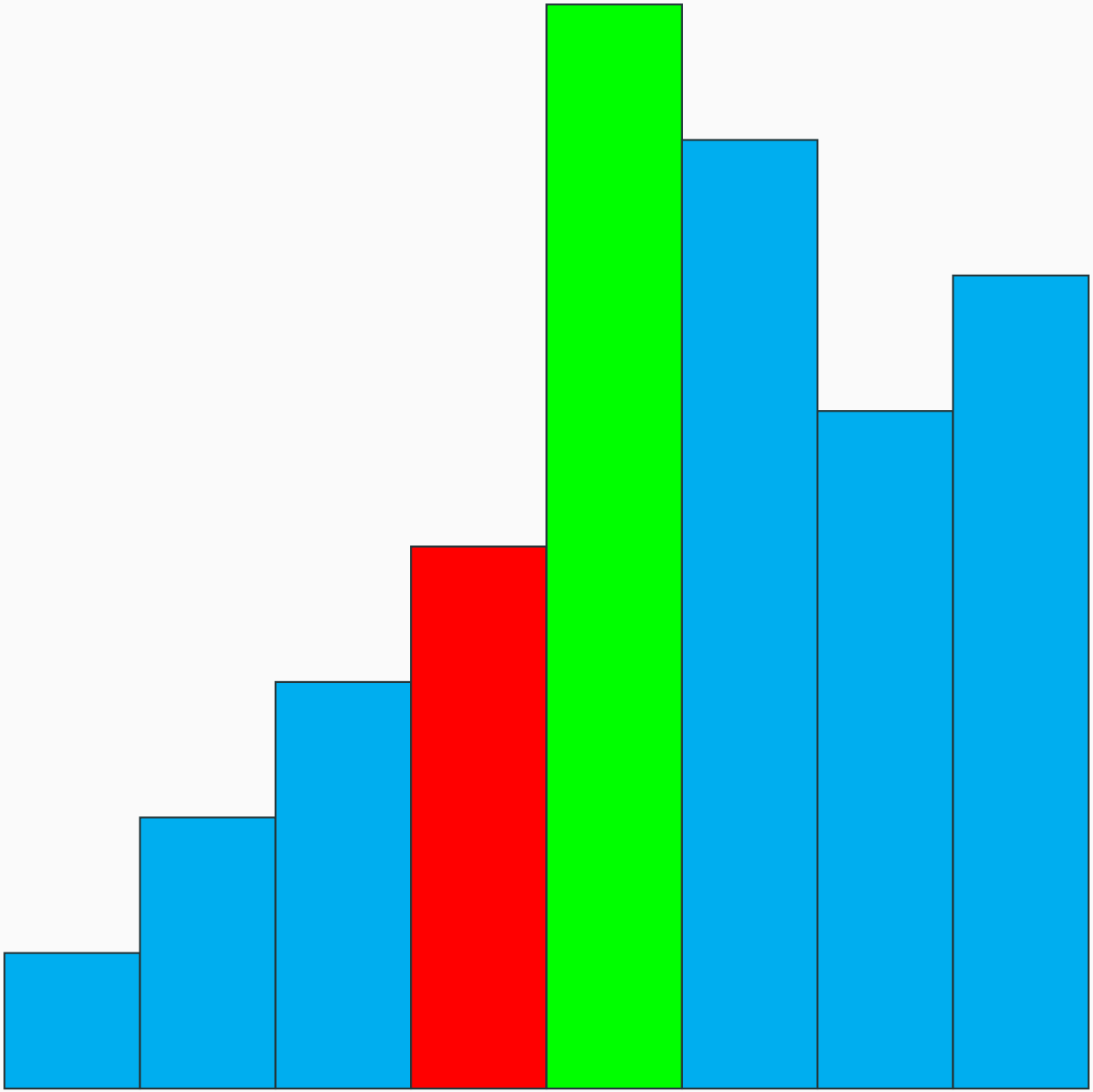
Exemple



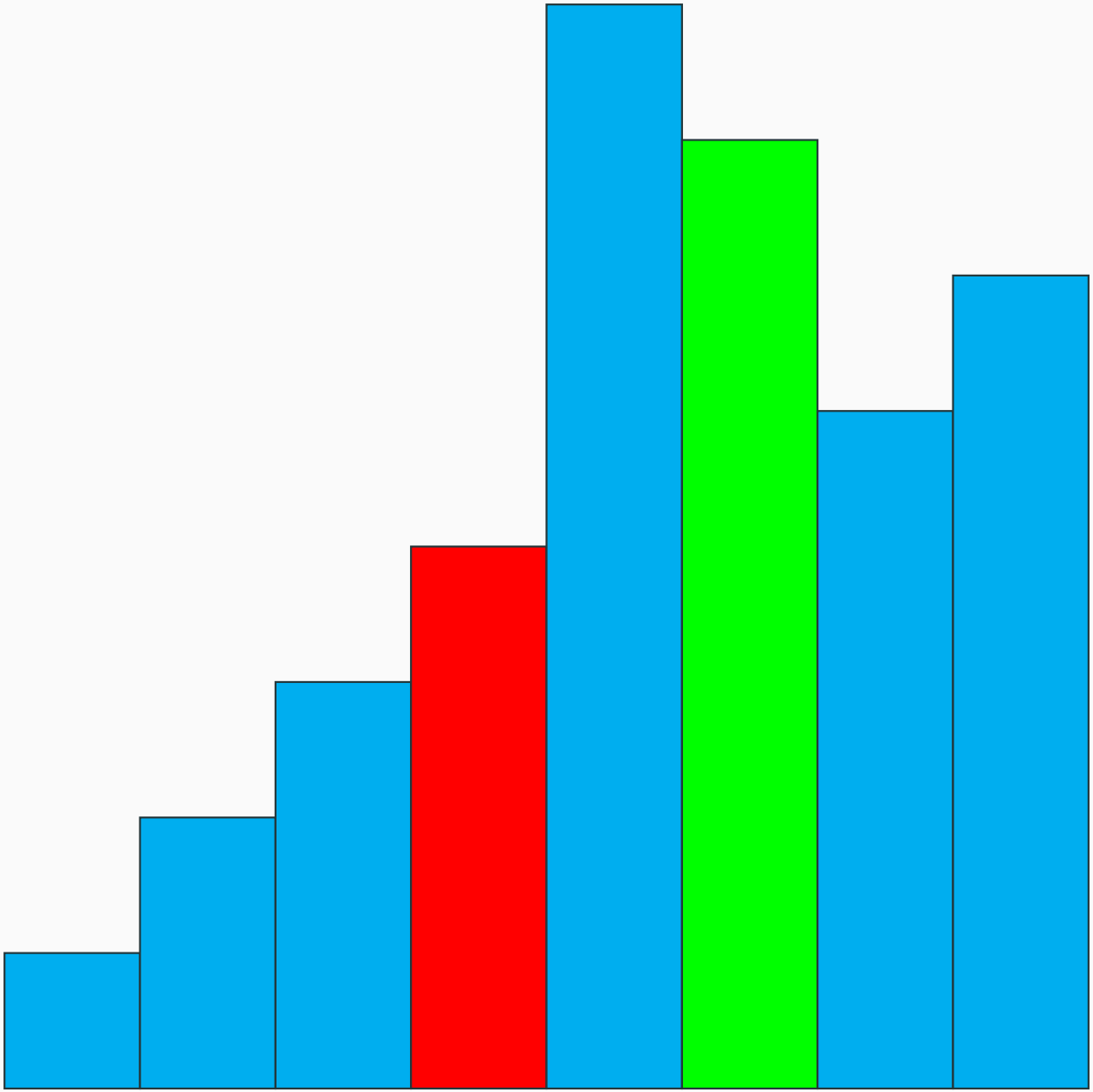
Exemple



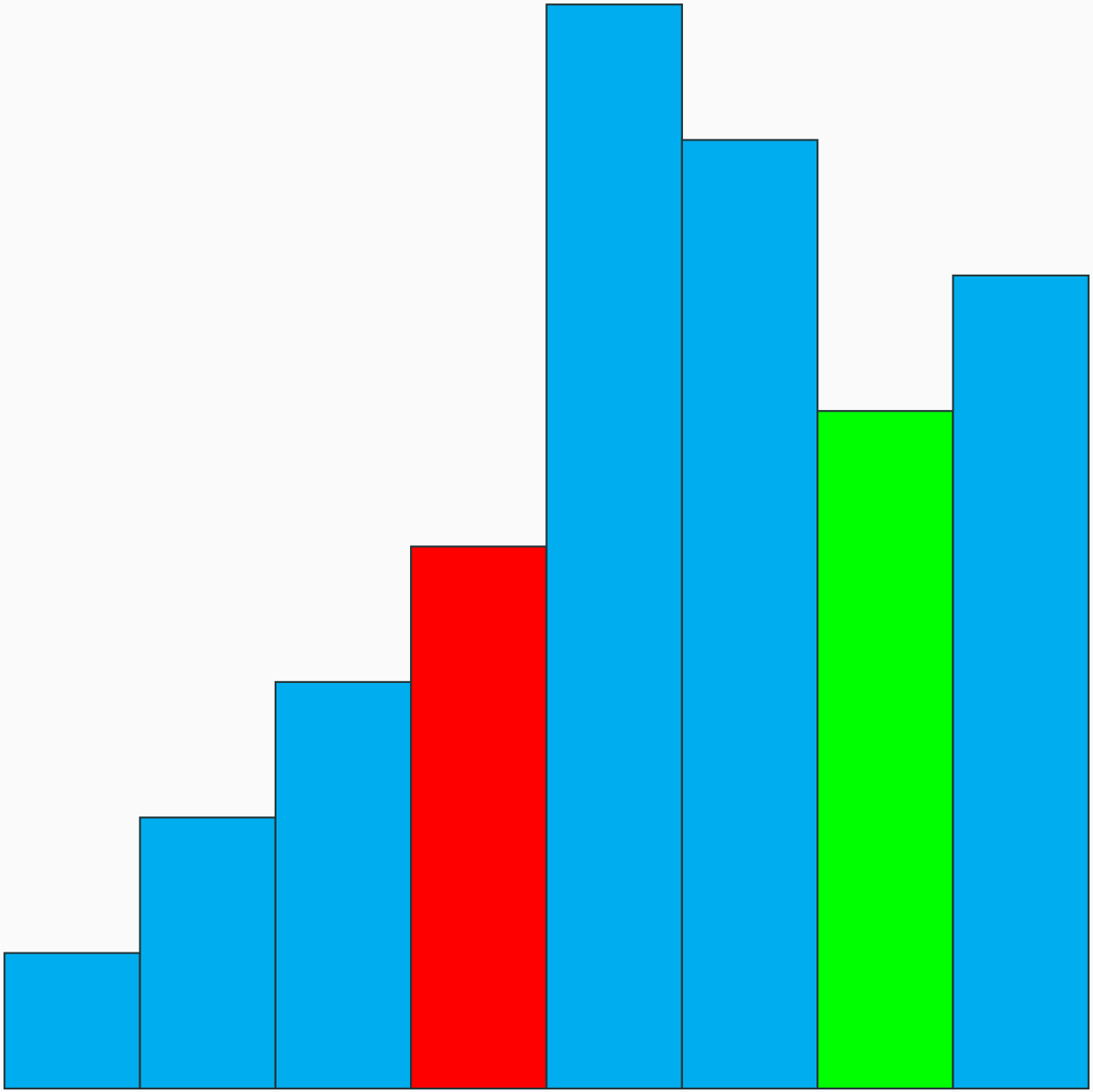
Exemple



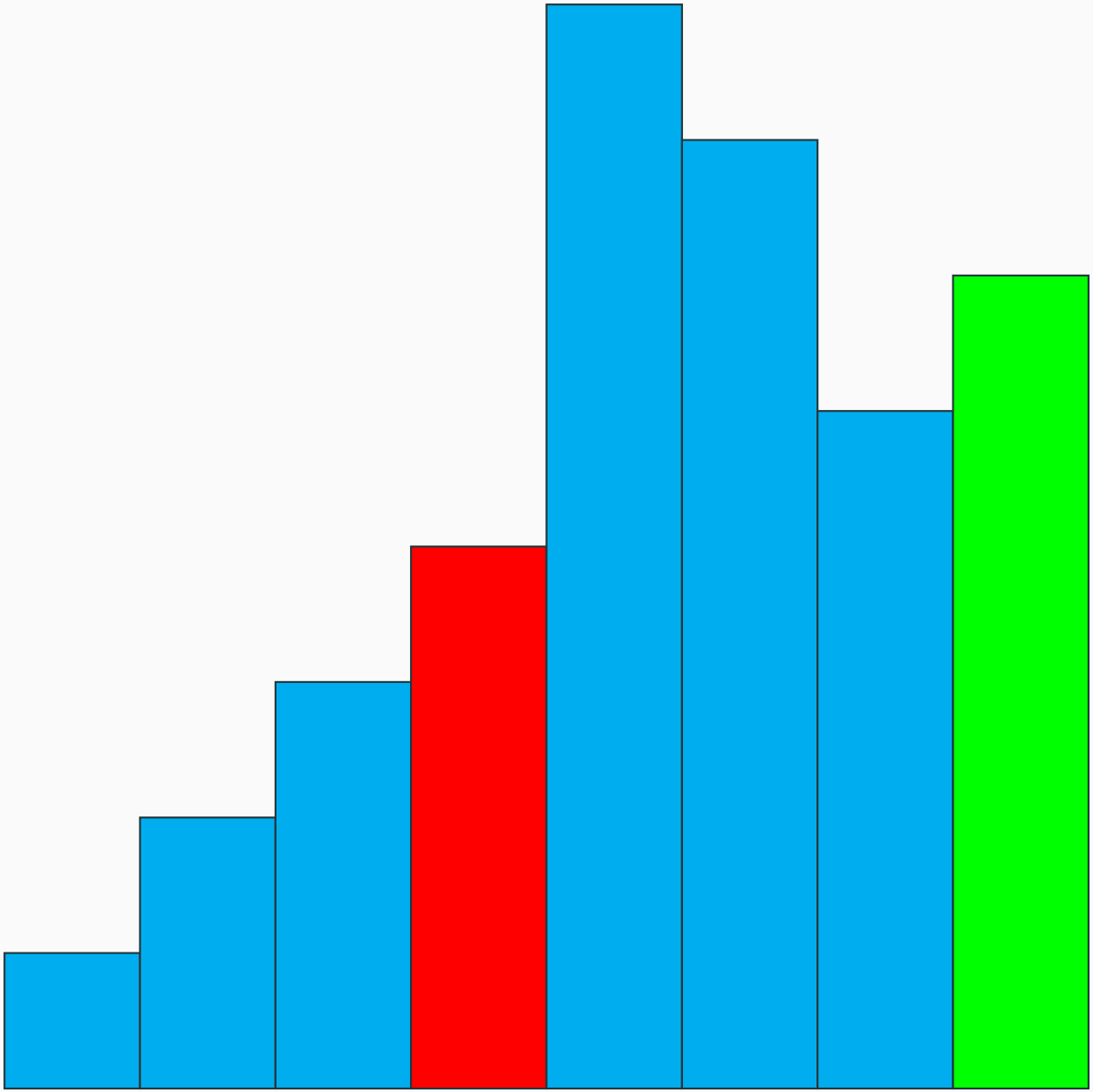
Exemple



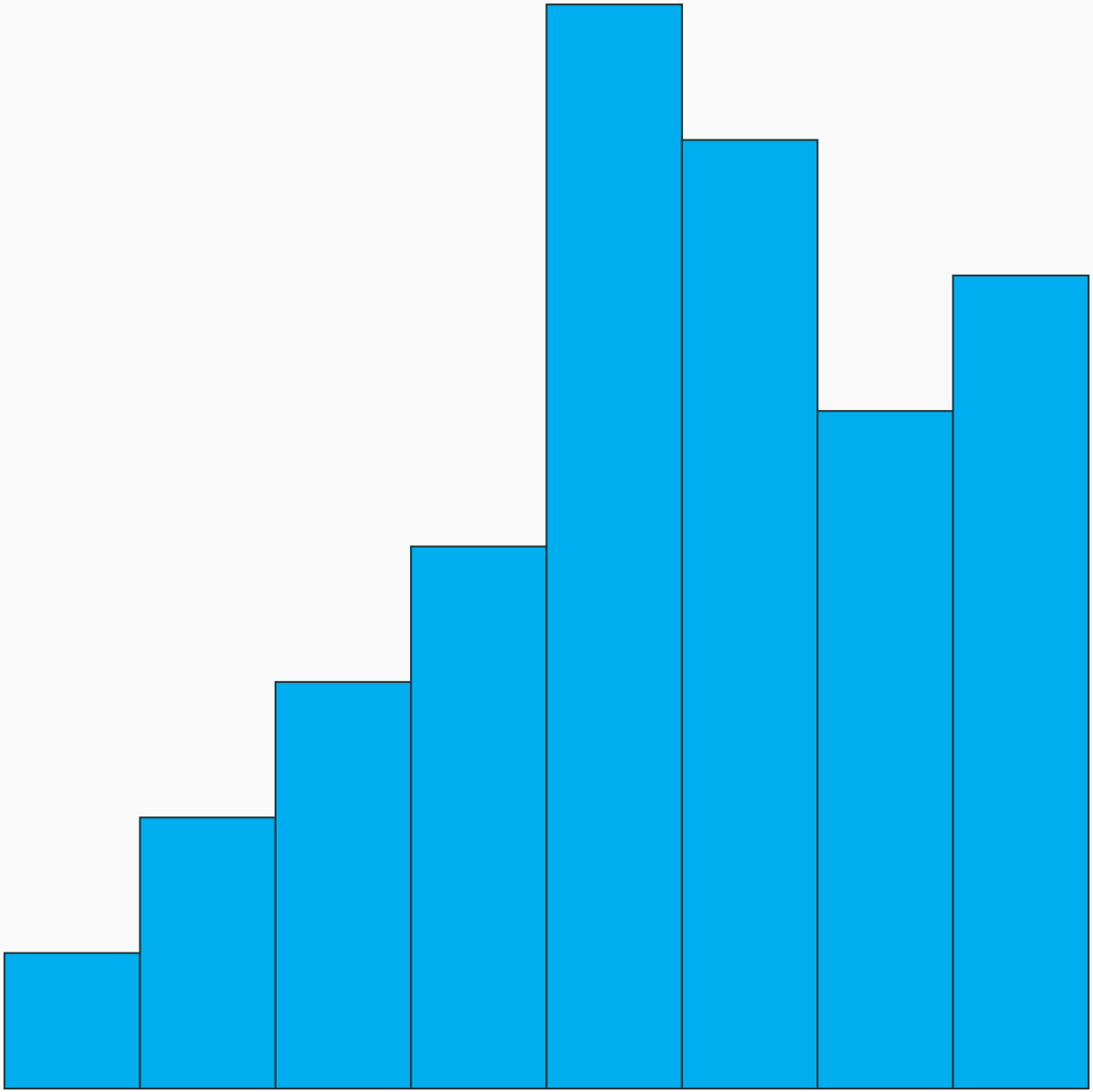
Exemple



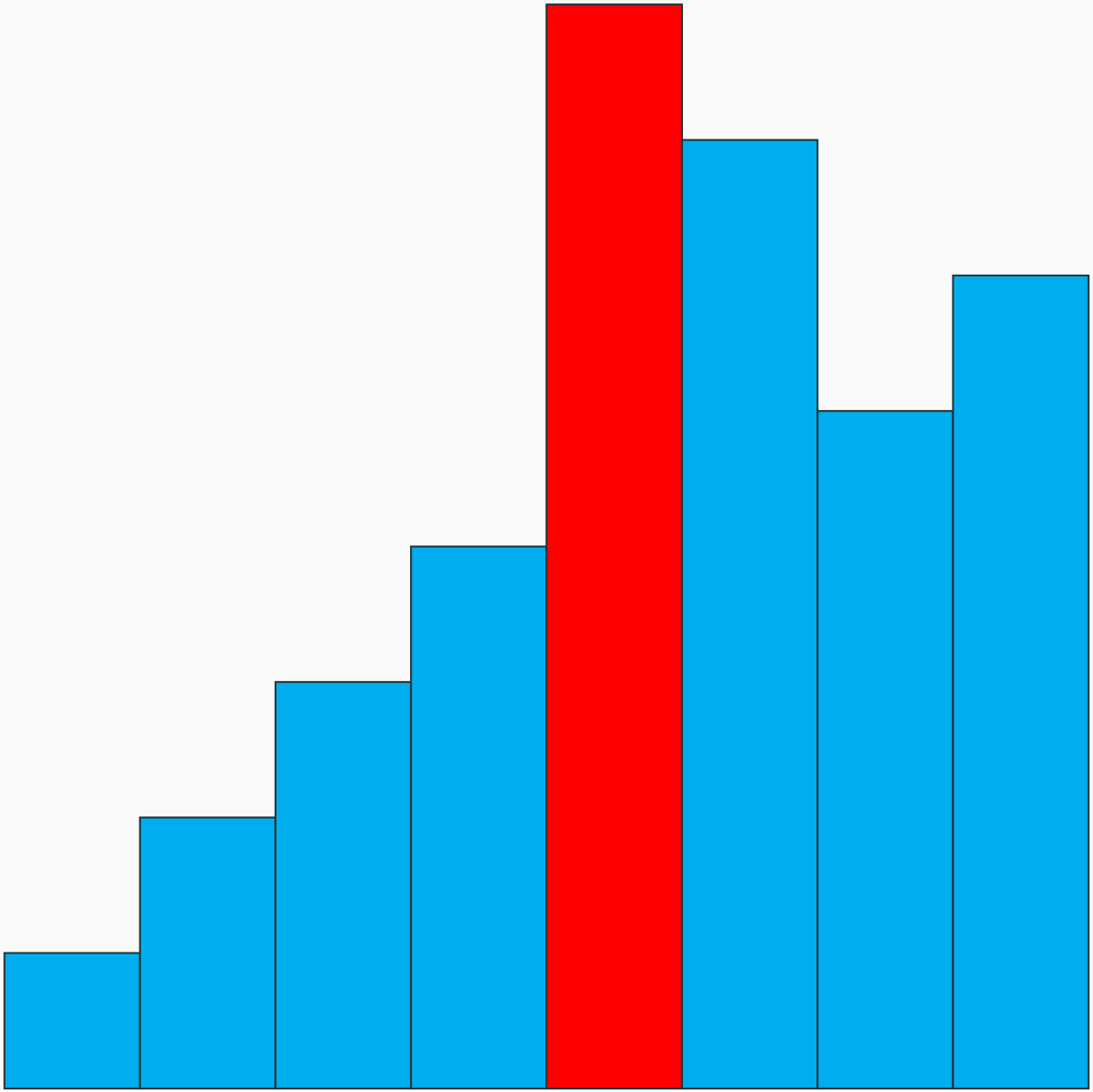
Exemple



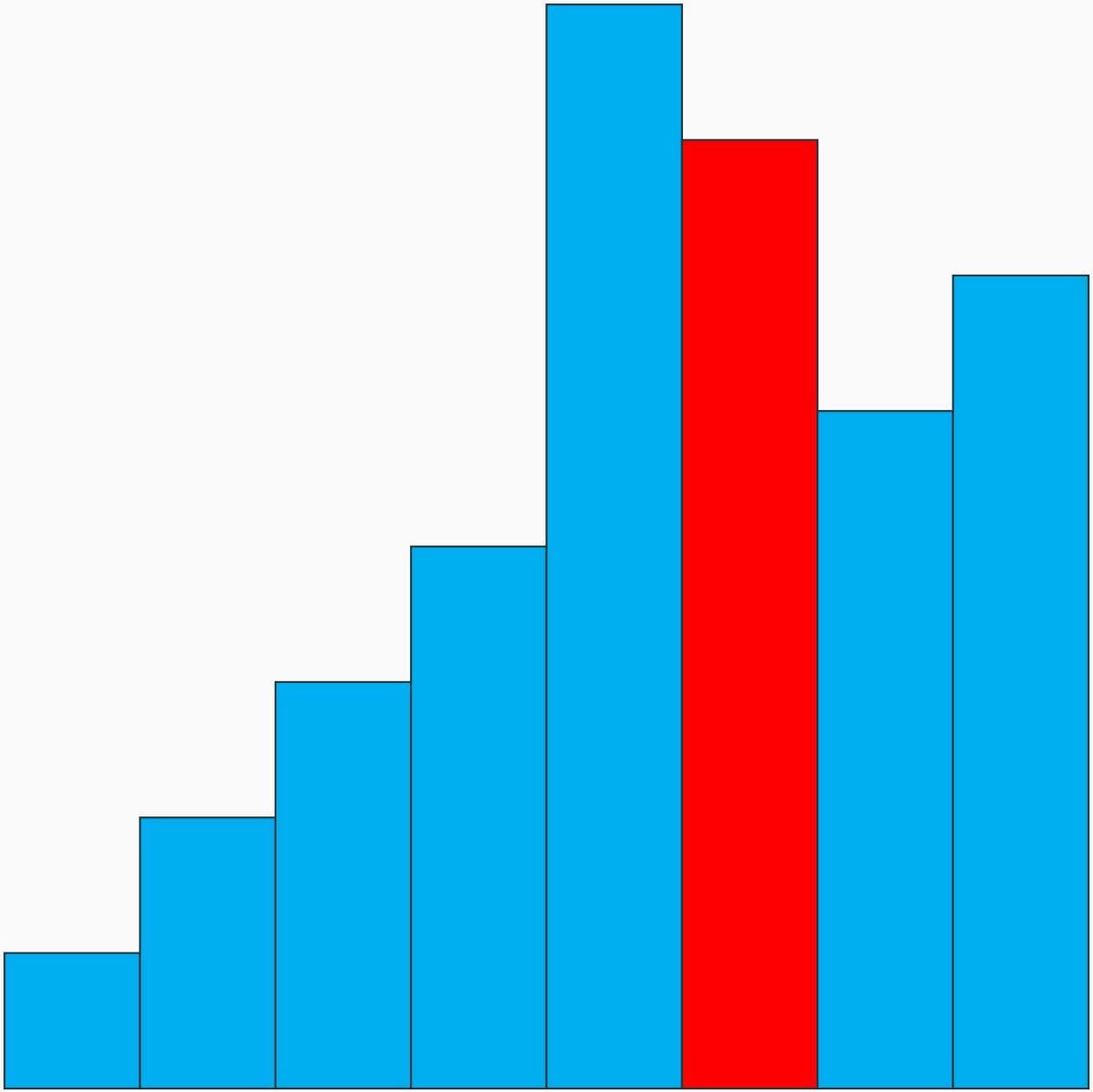
Exemple



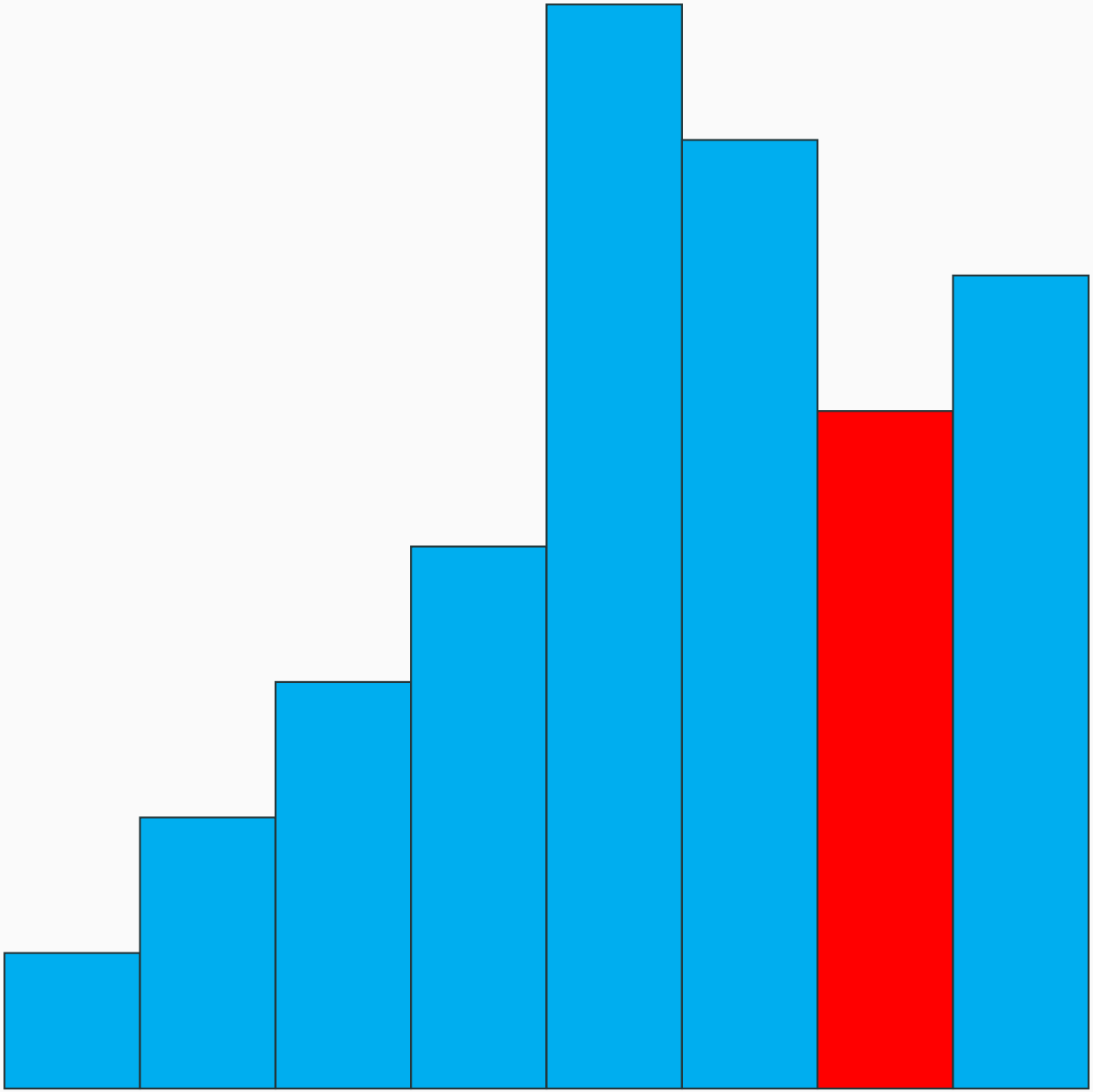
Exemple



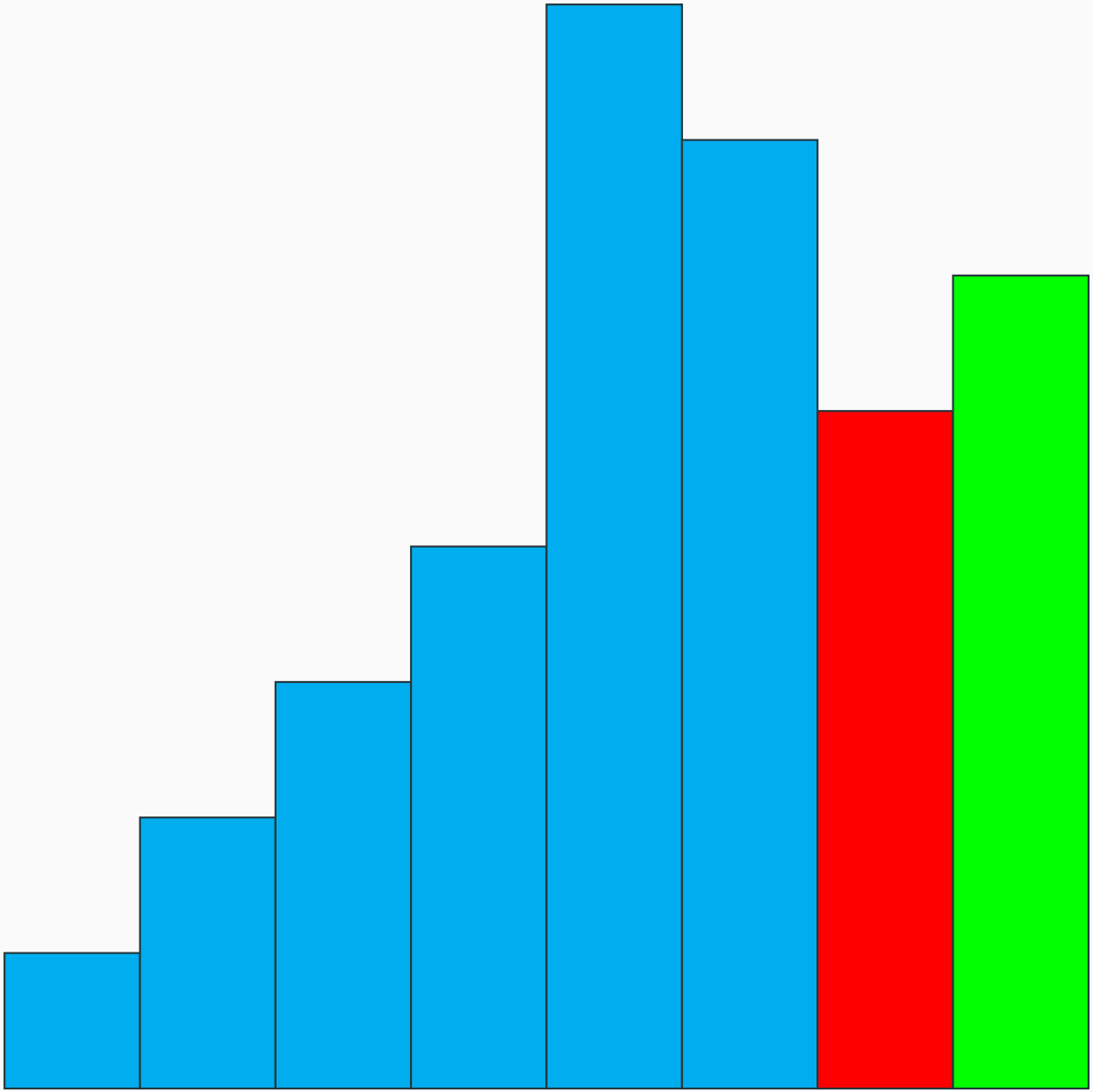
Exemple



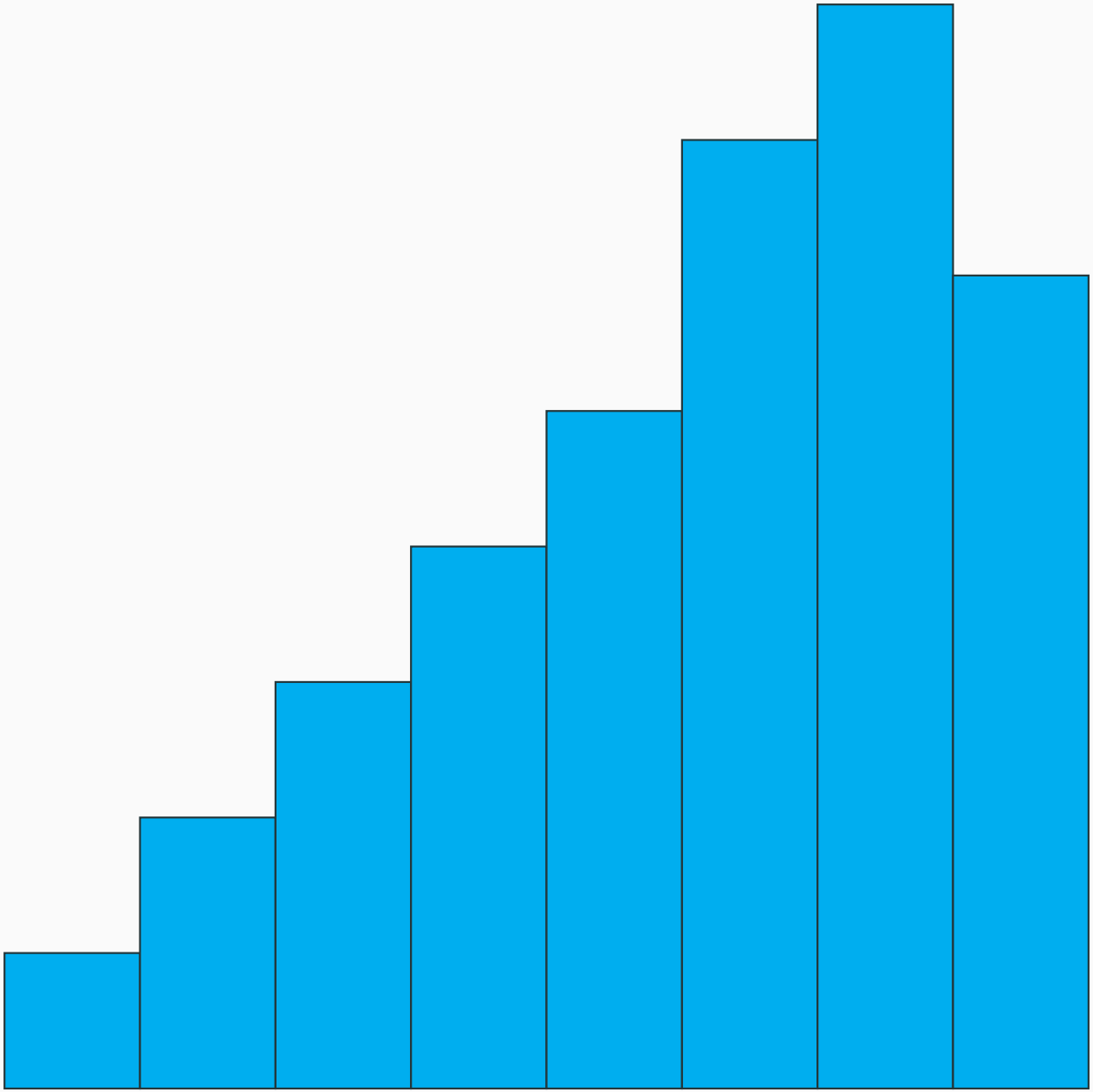
Exemple



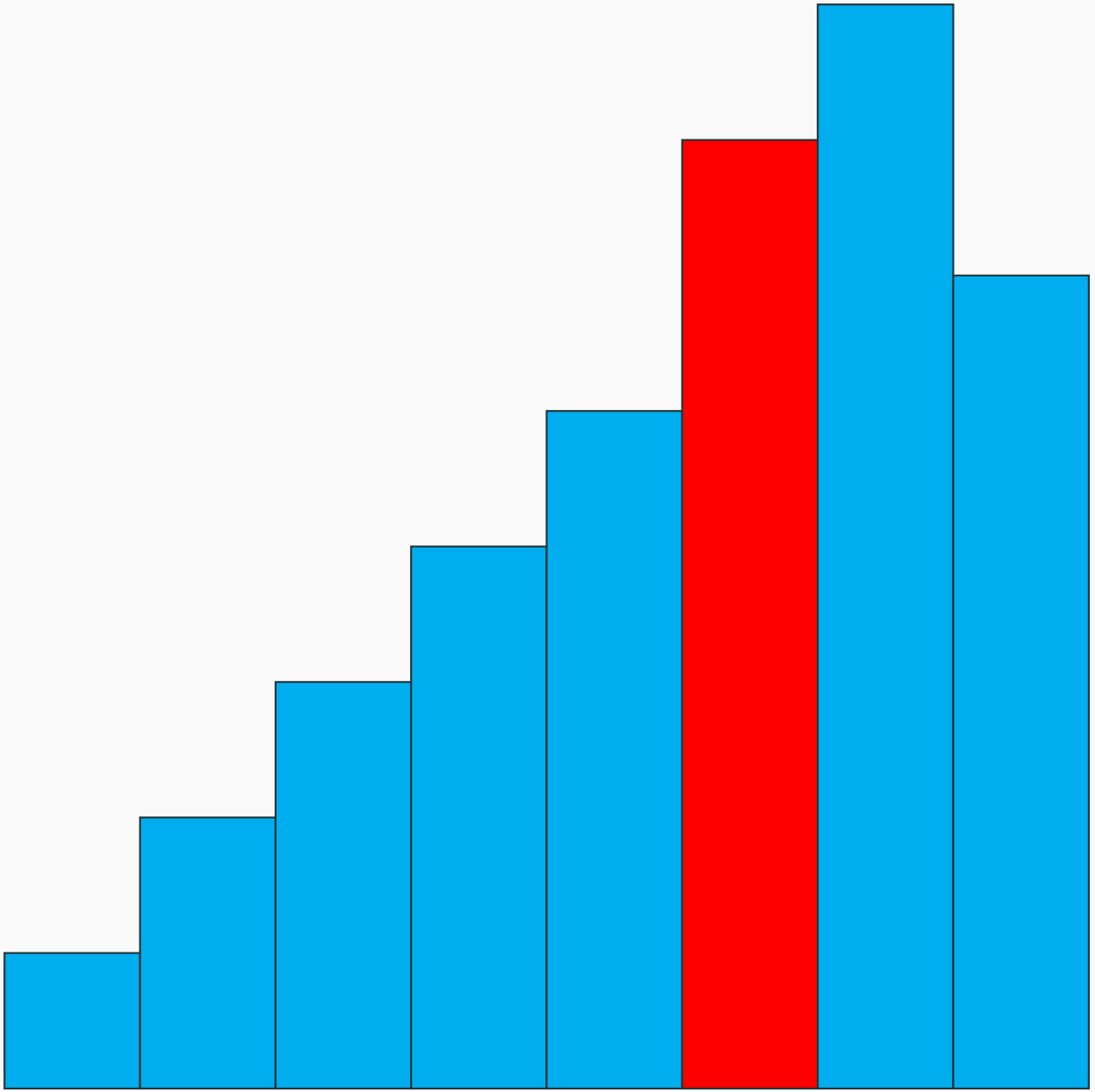
Exemple



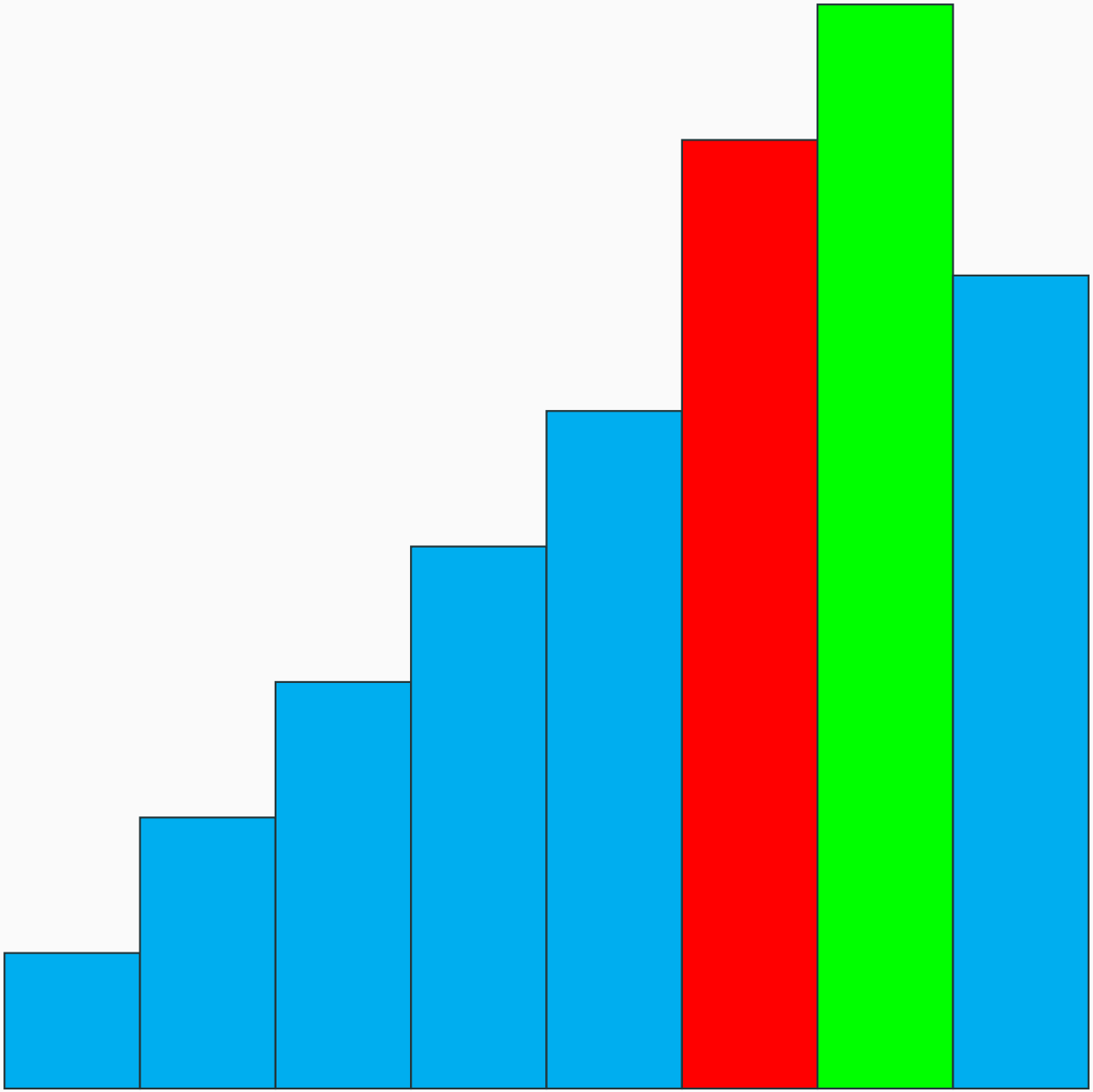
Exemple



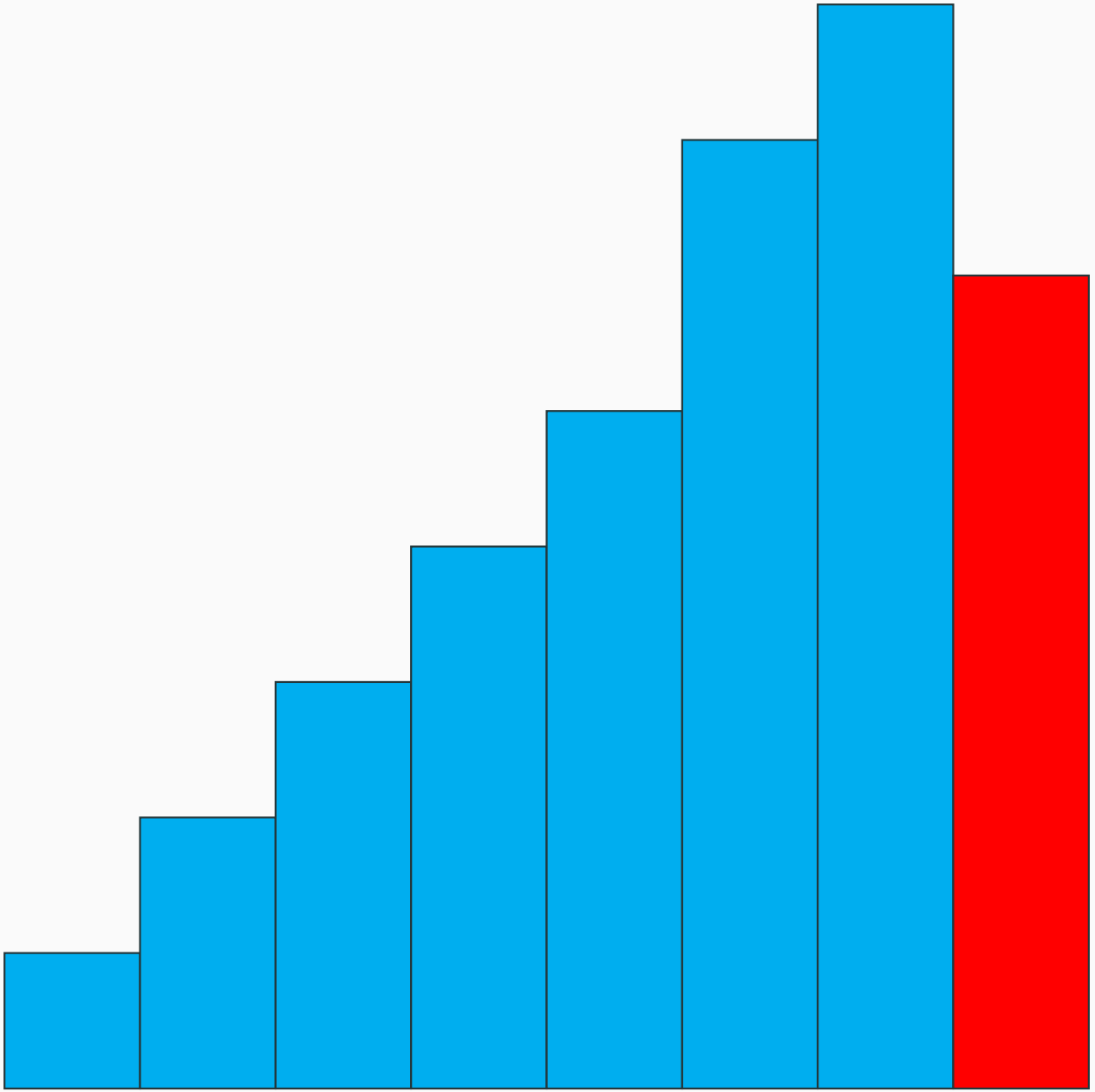
Exemple



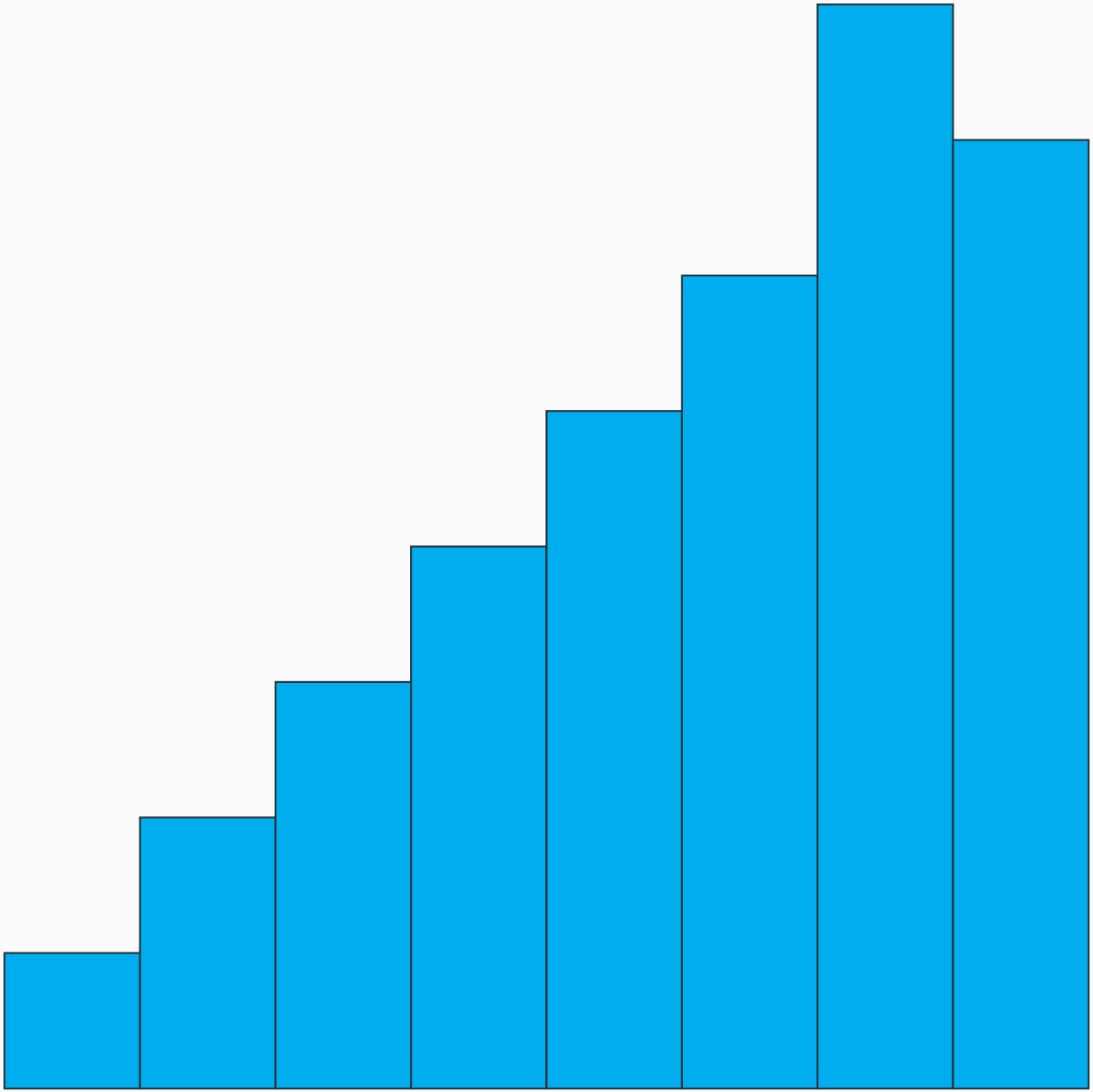
Exemple



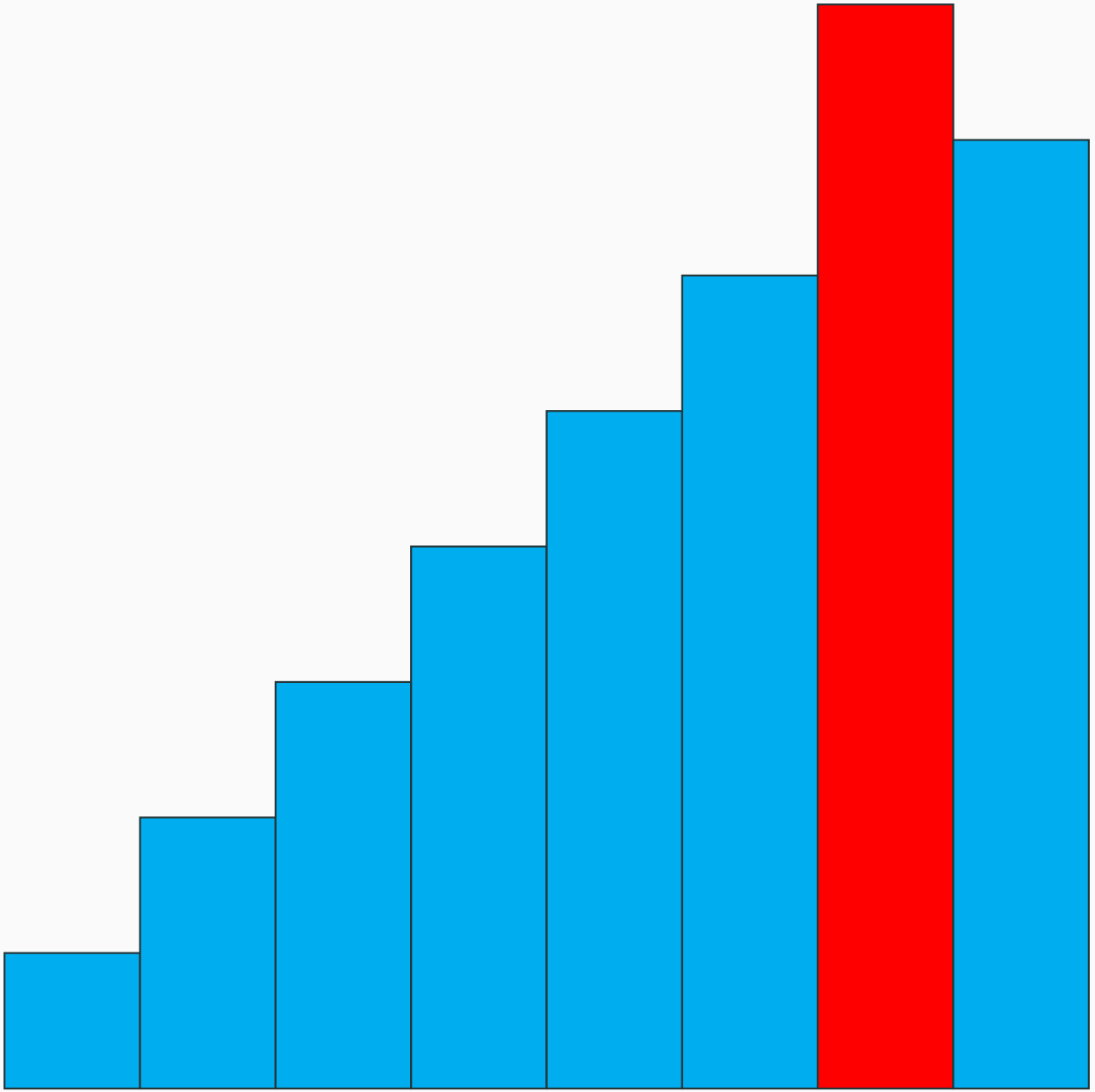
Exemple



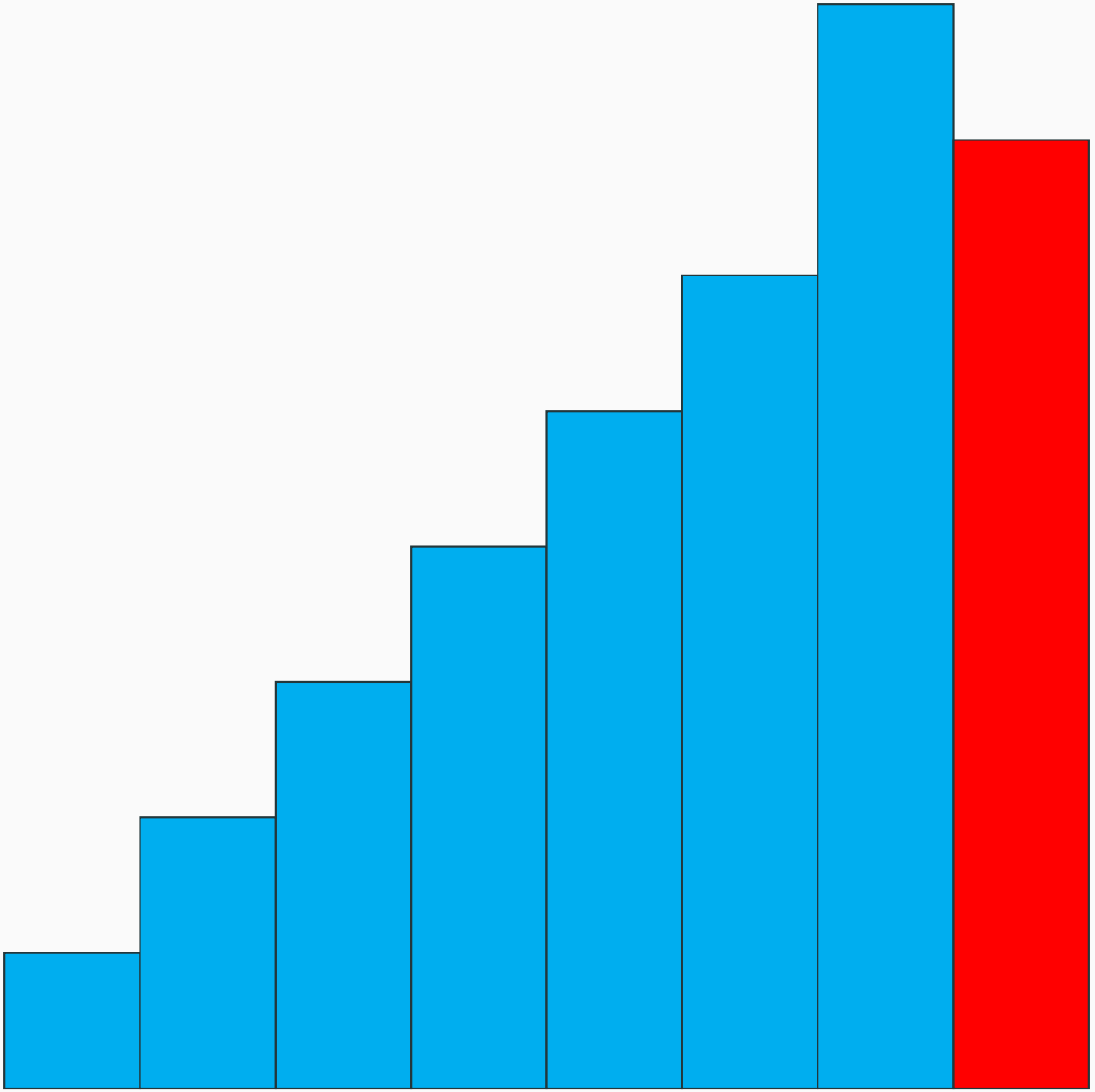
Exemple



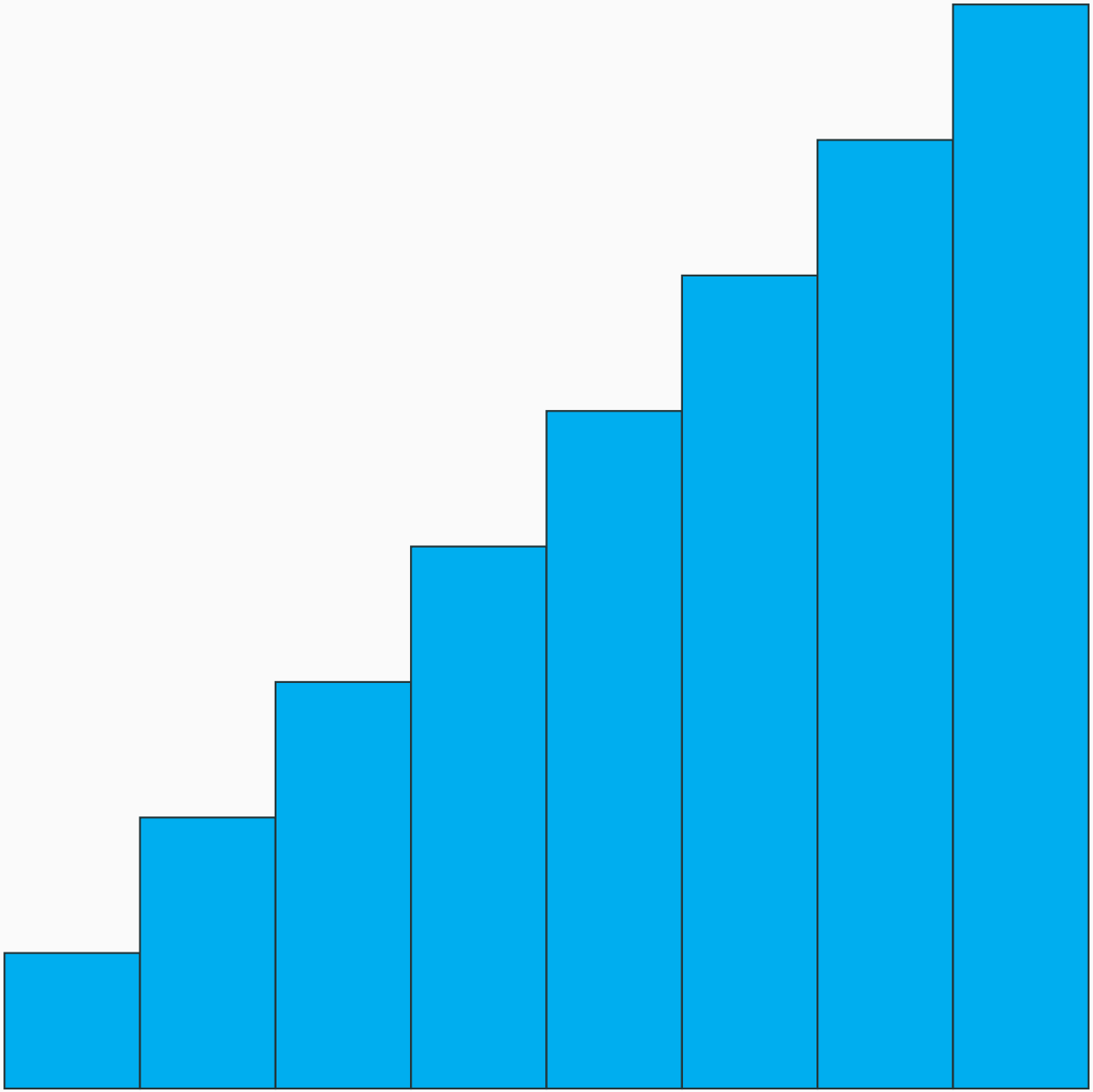
Exemple



Exemple



Exemple



Exemple : le tri par sélection

Terminaison

L'algorithme termine car il n'est constitué que de boucles `for`.

Correction

Pour prouver la correction de notre algorithme, on va d'abord analyser la fonction **`indice_min`**, puis étudier la fonction **`tri_par_selection`**.

Exemple : le tri par sélection

Tri par sélection

```
1  int indice_min(int *tab, int g, int d)
2  {
3    int i_min = g;
4    for (int j = g + 1; j <= d; j++)
5    {
6      if (tab[j] < tab[i_min])
7        i_min = j;
8    }
9    return i_min;
10 }
```

Correction de `indice_min`

Montrons que **`indice_min(tab, g, d)`** renvoie un entier **`i_min`** tel que :

$$\forall k \in [\![g, d]\!], \texttt{tab[i\_min]} \leq \texttt{tab[k]}$$

Pour cela, montrons l'invariant de boucle suivant :

$$\mathrm{Inv}(j) : \forall k \in [\![g, j-1]\!], \texttt{tab[i\_min]} \leq \texttt{tab[k]}$$

Exemple : le tri par sélection

Invariant

$$\mathrm{Inv}(j) : \forall k \in [\![g, j-1]\!], \texttt{tab[i\_min]} \leq \texttt{tab[k]}$$

Tri par sélection

```
1  int indice_min(int *tab, int g, int d)
2  {
3    int i_min = g;
4    for (int j = g + 1; j <= d; j++)
5    {
6      if (tab[j] < tab[i_min])
7        i_min = j;
8    }
9    return i_min;
10 }
```

Preuve

- $\mathrm{Inv}(g+1)$ est vrai avant la boucle, car **i_min** $= g$, et k est à valeur dans $[\![g, g]\!]$.
- Supposons que $\mathrm{Inv}(j)$ soit vrai à la ligne 5, et montrons que $\mathrm{Inv}(j+1)$ est vrai à la ligne 8 :

Exemple : le tri par sélection

Invariant

$$\mathrm{Inv}(j) : \forall k \in [\![g, j-1]\!], \texttt{tab[i\_min]} \leq \texttt{tab[k]}$$

Tri par sélection

```
1  int indice_min(int *tab, int g, int d)
2  {
3    int i_min = g;
4    for (int j = g + 1; j <= d; j++)
5    {
6      if (tab[j] < tab[i_min])
7        i_min = j;
8    }
9    return i_min;
10 }
```

Preuve

- si **tab[j]** $\geq$ **tab[i_min]**, alors la valeur de **i_min** ne change pas, et la propriété est vraie pour $k = j$ (et elle était déjà vraie par hypothèse pour $k \in [\![g, j-1]\!]$).
  Donc $\mathrm{Inv}(j+1)$ est vrai.

Exemple : le tri par sélection

Invariant

$$\mathrm{Inv}(j) : \forall k \in [\![g, j-1]\!], \texttt{tab[i\_min]} \leq \texttt{tab[k]}$$

Tri par sélection

```
1  int indice_min(int *tab, int g, int d)
2  {
3    int i_min = g;
4    for (int j = g + 1; j <= d; j++)
5    {
6      if (tab[j] < tab[i_min])
7        i_min = j;
8    }
9    return i_min;
10 }
```

Preuve

- si **tab[j]** $<$ **tab[i_min]**, alors **i_min** devient **i_min'** $= j$, on a donc bien $(k = j)$ **tab[i_min']** $\leq$ **tab[j]**, et $\forall k \in [\![g, j-1]\!]$, **tab[i_min']** $\leq$ **tab[i_min]** $\leq$ **tab[k]**.

  Donc $\mathrm{Inv}(j+1)$ est vrai.

Exemple : le tri par sélection

Invariant

$$\mathrm{Inv}(j) : \forall k \in [\![g, j-1]\!], \texttt{tab[i\_min]} \leq \texttt{tab[k]}$$

Tri par sélection

```
1  int indice_min(int *tab, int g, int d)
2  {
3    int i_min = g;
4    for (int j = g + 1; j <= d; j++)
5    {
6      if (tab[j] < tab[i_min])
7        i_min = j;
8    }
9    return i_min;
10 }
```

Preuve

Ainsi, la propriété est bien un invariant de boucle.

Donc, à la ligne 9, $\mathrm{Inv}(d+1)$ est vrai, d'où la correction de la fonction **`indice_min`**.

Exemple : le tri par sélection

Invariant

$\mathrm{Inv}'(i)$: le sous-tableau **tab[0..i-1]** est trié, et ne contient que des valeurs plus petites que celles du sous-tableau **tab[i..n-1]**.

Tri par sélection

```
1  void tri_par_selection(int *tab, int n)
2  {
3    for (int i = 0; i < n; i++)
4    {
5      int i_min = indice_min(tab, i, n-1);
6      echange(tab, i, i_min);
7    }
8  }
```

Preuve

- $\mathrm{Inv}'(0)$ est vrai avant la boucle, car le sous-tableau en question est vide.
- Supposons que $\mathrm{Inv}'(i)$ soit vrai à la ligne 4, et montrons que $\mathrm{Inv}'(i+1)$ est vrai à la ligne 7 :

Exemple : le tri par sélection

Invariant

$\mathrm{Inv}'(i)$: le sous-tableau **tab[0..i-1]** est trié, et ne contient que des valeurs plus petites que celles du sous-tableau **tab[i..n-1]**.

Tri par sélection

```
1  void tri_par_selection(int *tab, int n)
2  {
3    for (int i = 0; i < n; i++)
4    {
5      int i_min = indice_min(tab, i, n-1);
6      echange(tab, i, i_min);
7    }
8  }
```

Preuve

- Par hypothèse, on a : $\forall k \in [\![0, i-1]\!]$, **tab[k]** $\leq$ **tab[i_min]**.
- D'après la correction de **indice_min**, on a : $\forall k' \in [\![i, n-1]\!]$, **tab[i_min]** $\leq$ **tab[k']**.

Exemple : le tri par sélection

Invariant

$\mathrm{Inv}'(i)$: le sous-tableau **`tab[0..i-1]`** est trié, et ne contient que des valeurs plus petites que celles du sous-tableau **`tab[i..n-1]`**.

Preuve

Ainsi, après l'échange de la ligne 6, on a : $\forall k \in [\![0, i-1]\!], \forall k' \in [\![i, n-1]\!]$, **`tab[k]`** $\leq$ **`tab[i]`** $\leq$ **`tab[k']`**.

De plus, **`tab[0..i-1]`** étant trié par hypothèse, on a bien que **`tab[0..i]`** est trié.

Correction de `tri_par_selection`

Ainsi, la propriété est bien un invariant de boucle.

Donc, à la ligne 8, $\mathrm{Inv}'(n)$ est vrai : **`tab`** est donc trié.

Validation, test

Validation, test

Validation

Une bonne pratique de programmation est de tester nos différentes fonctions.

Dans ce cas, il faut réflechir à quels tests effectuer pour valider nos programmes, en testant tous les cas de figure possibles :

- si la condition d'un `if` ou d'un `while` comporte des conjonctions ou disjonctions, il convient d'écrire suffisament de tests pour vérifier toutes les possibilités de la satisfaire ;
- il faut trouver des tests qui forceront notre programme à passer dans chaque `if`, `else if`, et `else` ;
- il faut identifier les cas limites et les tester.

Validation, test

assert

La librairie **`assert.h`** contient une fonction **`assert`** qui prend en argument un test à effectuer.

- Si le test est vérifié, le programme continue normalement.
- Si le test n'est pas satisfait, le programme est interrompu et un message d'erreur indique quel test n'a pas fonctionné.

Validation, test

dicho.c (1/2)

```c
#include <stdbool.h>
#include <assert.h>

bool recherche_dicho(int *tab, int n, int x)
{
  int g = 0, d = n;
  while (g < d)
  {
    int m = (g + d) / 2;
    if (tab[m] == x) { return true; }
    else if (tab[m] < x) { g = m + 1; }
    else { d = m; }
  }
  return false;
}
```

dicho.c (2/2)

```c
int main()
{
  int vide[] = {};
  assert(recherche_dicho(vide, 0, 1)==false);

  int tab[] = {1,2,3,4,5};
  assert(recherche_dicho(tab, 5, 1)==true);
  assert(recherche_dicho(tab, 5, 5)==true);
  assert(recherche_dicho(tab, 5, 0)==false);
  assert(recherche_dicho(tab, 5, 8)==false);

  // test non satisfait
  assert(recherche_dicho(tab, 5, 4)==false);
}
```

Console

```
janson@mp2i:~$ gcc dicho.c -o exemple
janson@mp2i:~$ ./exemple
exemple: dicho.c:26: main: Assertion `recherche_dicho(tab, 5, 4)==false' failed.
Aborted
```

Complexité

Complexité

On sait maintenant prouver que nos algorithmes terminent et renvoient le bon résultat.

La dernière question est la suivante : quel temps mettent-ils à s'exécuter ?

Exemple

Voici le temps en secondes du calcul de 5^n pour différents n avec les algorithmes `expo` et `expo_rapide` vus précédemment.

algorithme \ `n`	10^1	10^2	10^3	10^4	10^5	10^6	10^7
`expo`	1.6×10^{-5}	3.4×10^{-5}	5.0×10^{-4}	0.014	0.97	98	11×10^3
`expo_rapide`	1.3×10^{-5}	1.5×10^{-5}	3.7×10^{-5}	6.0×10^{-4}	0.019	0.71	35

Complexité des algorithmes d'exponentiation

Complexité

Pour expliquer ces différences d'efficacité, nous allons compter combien d'opérations sont nécessaires à chacun de ces algorithmes en fonction de n.

Complexité des algorithmes d'exponentiation

Complexité

Pour expliquer ces différences d'efficacité, nous allons compter combien d'opérations sont nécessaires à chacun de ces algorithmes en fonction de n.

Exponentiation naïve

```
1  int expo(int x,int n)
2  {
3    int res = 1;
4    for (int i = 0; i<n; i++)
5    {
6      res *= x;
7    }
8    return res;
9  }
```

Exponentiation naïve

L'algorithme expo réalise une multiplication par tour de boucle **for**, donc n au total.

Complexité des algorithmes d'exponentiation

Exponentiation rapide

Pour l'algorithme d'exponentiation rapide, c'est un peu plus compliqué, et on va simplement donner une majoration.

Complexité des algorithmes d'exponentiation

Exponentiation rapide

```
1  int expo_rapide(int x, int n)
2  {
3    int y = x, z = 1, m = n;
4    while (m>0)
5    {
6      int q = m / 2, r = m%2;
7      if (r == 1)
8      {
9        z *= y;
10     }
11     y *= y;
12     m = q;
13   }
14   return z;
15 }
```

Complexité

Au pire, l'algorithme effectue deux multiplications à chaque passage dans la boucle **while**.

Le nombre de tours de boucle effectués correspond au nombre de chiffres de n en base 2, qui est de l'ordre de $\frac{\ln(n)}{\ln(2)} = \log_2(n)$.

Ainsi, on effectue dans le pire des cas de l'ordre de $2 \times \log_2(n)$ multiplications.

Remarques

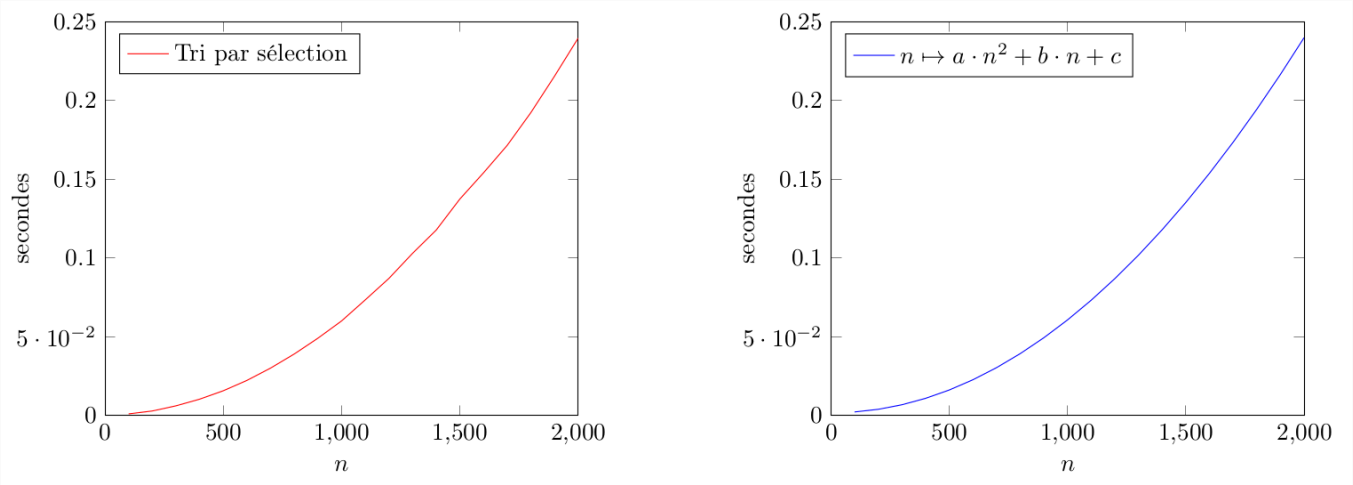
Remarques

- L'important ici ne réside pas dans les constantes du résultat : ce qui est essentiel c'est que le calcul de la puissance par l'algorithme naïf fait de l'ordre de n multiplications alors que l'algorithme d'exponentiation rapide en fait seulement de l'ordre de $\ln(n)$.
- Les temps de calcul des algorithmes ne suivent pas vraiment une progression linéaire en n (pour le premier) ou logarithmique (pour le second).
  Cela s'explique par le fait que les entiers manipulés étaient de taille variable : 5^{10^7} est un entier de plus de 20 millions de bits (6 millions de chiffres en base 10).
  Il est évidemment plus difficile de multiplier des entiers de cette taille que des entiers comme 2 et 3.

Temps d'exécution

Le graphique ci-dessous montre le temps d'exécution de l'algorithme de tri par sélection sur des tableaux de tailles allant de 100 à 2000, constituées d'entiers tirés aléatoirement entre 0 et 10000.

On remarque que le temps de calcul coïncide avec la fonction polynomiale de degré 2 donnée par $x \mapsto ax^2 + bx + c$ avec $a = 6.01 \times 10^{-8}$, $b = -1.25 \times 10^{-6}$, et $c = 1.72 \times 10^{-3}$.



Complexité

Ordre de grandeur

Les coefficients précédents dépendent de la machine sur laquelle on a effectué les tests, du langage utilisé, et de l'implémentation précise de l'algorithme.

Mais ce qui est inhérent à l'algorithme, c'est que le temps de calcul est un polynôme de degré 2.

Complexité : définitions et méthodes

Qu'est-ce que la complexité ?

La **complexité** d'une fonction sur un ensemble de paramètres est le prix à payer en termes de ressources pour mener à bien le calcul.

Différents types de ressources peuvent être évalués :

- Le temps de calcul CPU : nombre d'opérations élémentaires réalisées par le processeur, ce qui est directement lié au temps de calcul.
- La mémoire nécessaire : mémoire RAM ou sur disque dur.
- ...

Complexité : définitions et méthodes

Définition

La **complexité** est la mesure de l'efficacité d'un programme pour un type de ressources :

- complexité **temporelle** : temps de calcul.
- complexité **spatiale** : espace mémoire.

En pratique

De nos jours, la complexité temporelle est en général plus importante que la complexité spatiale.

L'étude de la complexité d'une fonction consiste à estimer son coût en ressource en fonction des entrées.

En pratique

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

Pour nous, les entrées seront essentiellement constituées d'entiers, de flottants, ou de tableaux.

- 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 lui-même ;
  - ou de son nombre de chiffres (sa taille) : $\log_2(n)$.

Complexité : définitions et méthodes

Exemple

Le choix dépendra en général du contexte :

- 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 $\log_2(n)$.
- Pour calculer $n! \mod q$ où q est un nombre fixé, la donnée pertinente est n lui-même.

Complexité temporelle

Coût temporel

Concentrons-nous d'abord sur la complexité en temps.

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, affecter un entier ou un flottant, etc.

Estimer le coût en temps d'une fonction sur une entrée de taille donnée signifie estimer le nombre de ces opérations élémentaires effectuées par la fonction sur l'entrée.

Complexité spatiale

Coût spatial

La complexité spatiale consiste à estimer la mémoire nécessaire à une fonction pour son exécution, en plus de celles des entrées.

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.

Méthode générale pour calculer la complexité

Méthode générale

Voici la méthode générale pour obtenir une majoration (un O) de la complexité d'un algorithme :

- Les opérations **élémentaires** (affectations, entrée/sortie de fonction, opérations arithmétiques, ...) sont comptées avec un coût constant : $O(1)$.
- La complexité d'un **boucle** est égale à la somme des complexités de chaque passage dans la boucle.
  - En général, il est suffisant de majorer cette complexité par le nombre de tours de boucles multiplié par la complexité maximale d'un tour de boucle.
  - Attention toutefois, cela conduit parfois à la surestimer.

Méthode générale pour calculer la complexité

Méthode générale

Voici la méthode générale pour obtenir une majoration (un O) de la complexité d'un algorithme :

- La complexité d'un bloc **conditionnel** `if`, `else if`, ..., `else` est majoré par le maximum des complexités de chaque cas.
- L'appel à une fonction compte comme le coût de cette fonction sur les paramètres où elle est appelée.
  ⚠ Attention à ne pas oublier ces coûts.

Applications aux algorithmes précédents

Exemple : algorithmes linéaires

Les algorithmes de recherche dans un tableau ou du calcul de la somme des éléments d'un tableau ont une complexité en $O(n)$, où n est la taille du tableau.

En effet, ces algorithmes sont basés sur une boucle `for` qui s'exécute n fois pour parcourir le tableau.

De même, l'algorithme d'exponentiation naïve a une complexité en $O(n)$.

Applications aux algorithmes précédents

Moyenne des éléments d'un tableau

```
1  int moyenne(int *tab, int n)
2  {
3    return somme(tab, n) / n;
4  }
```

Exemple : appel de fonction

L'algorithme ci-dessus a également une complexité en $O(n)$, car on effectue un nombre fini d'opérations élémentaires, plus l'appel à la fonction **`somme`**.

Applications aux algorithmes précédents

Exemple : exponentiation rapide

L'algorithme d'exponentiation rapide calcule x^n en effectuant autant de tours de boucle que n a de chiffres en base 2, et chaque tour de boucle se fait avec une complexité constante.

On en déduit une complexité en $O(\log_2(n))$.

Remarque

Dans ce cas, la base du logarithme n'a pas d'importance, car $\log_a(n) = \Theta(\log_b(n))$ pour n'importe quelles bases a et b.

On peut donc simplement écrire $O(\log(n))$ sans préciser la base.

Si l'on veut préciser que cette borne est optimale, on peut écrire que sa complexité est en $\Theta(\log(n))$.

Applications aux algorithmes précédents

Recherche dichotomique

```
1   bool recherche_dicho(int *tab, int n, int x)
2   {
3     int g = 0, d = n;
4     while (g < d)
5     {
6       int m = (g + d) / 2;
7       if (tab[m] == x) { return true; }
8       else if (tab[m] < x) { g = m + 1; }
9       else { d = m; }
10    }
11    return false;
12  }
```

Complexité

Là aussi, chaque tour de boucle se fait en temps constant.

Il reste à estimer le nombre de tours de boucles effectués par l'algorithme.

Complexité de la recherche dichotomique

Notons n la taille du tableau, et d_i et g_i les valeurs des variables d et g après i tours de boucle. On note aussi $t_i = d_i - g_i$.

Initialement, $d_0 = n$ et $g_0 = 0$, donc $t_0 = n$.

Ensuite,

- si d prend la valeur m après un tour de boucle supplémentaire, on a $d_{i+1} = \lfloor \frac{d_i+g_i}{2} \rfloor \leq \frac{d_i+g_i}{2}$, et $g_{i+1} = g_i$, donc $t_{i+1} = d_{i+1} - g_{i+1} \leq \frac{d_i-g_i}{2} \leq \frac{t_i}{2}$;
- sinon, g prend la valeur $m+1$ après un tour de boucle supplémentaire, donc $g_{i+1} = 1 + \lfloor \frac{d_i+g_i}{2} \rfloor \geq \frac{d_i+g_i+1}{2}$, et $d_{i+1} = d_i$, donc $t_{i+1} \leq \frac{d_i-g_i-1}{2} \leq \frac{t_i}{2}$.

Complexité de la recherche dichotomique

Complexité

Ainsi, dans tous les cas on a $t_{i+1} \leq \frac{t_i}{2}$.

On vérifie donc aisément par récurrence que $0 \leq t_k \leq \frac{n}{2^k}$, et donc que $t_k = 0$ pour $k > \log_2(n)$.

Comme la boucle s'arrête lorsque $d - g = 0$, on conclut que le nombre d'opérations effectuées est en $O(\log(n))$.

Applications aux algorithmes précédents

Tri par sélection

```c
void echange(int *tab, int i, int j)
{
  int temp = tab[i];
  tab[i] = tab[j];
  tab[j] = temp;
}

int indice_min(int *tab, int g, int d)
{
  int i_min = g;
  for (int j = g + 1; j <= d; j++)
  {
    if (tab[j] < tab[i_min])
      i_min = j;
  }
  return i_min;
}

void tri_par_selection(int *tab, int n)
{
  for (int i = 0; i < n; i++)
  {
    int i_min = indice_min(tab, i, n-1);
    echange(tab, i, i_min);
  }
}
```

Exemple

- La complexité de **echange** est en $O(1)$.
- **indice_min** s'effectue en $O(d - g)$, car on effectue $d - g$ passages dans la boucle **for**, et chaque passage s'effectue en $O(1)$.

Applications aux algorithmes précédents

Tri par sélection

```c
void echange(int *tab, int i, int j)
{
  int temp = tab[i];
  tab[i] = tab[j];
  tab[j] = temp;
}

int indice_min(int *tab, int g, int d)
{
  int i_min = g;
  for (int j = g + 1; j <= d; j++)
  {
    if (tab[j] < tab[i_min])
      i_min = j;
  }
  return i_min;
}

void tri_par_selection(int *tab, int n)
{
  for (int i = 0; i < n; i++)
  {
    int i_min = indice_min(tab, i, n-1);
    echange(tab, i, i_min);
  }
}
```

Exemple

- La complexité de **tri_par_selection** est en $O(n^2)$, car on passe n fois dans la boucle **for**, et chaque passage s'effectue en $O(n)$ à cause de l'appel à **indice_min**.

Quelques ordres de grandeur

$f(n)$	$\ln(n)$	$\sqrt{n}$	n	n^2	n^3	2^n	$n!$	n^n
$n_{\max}$	très gros !	3.6×10^{21}	6×10^{10}	244948	3914	35	13	10

Ordres de grandeur

Le tableau ci-dessus présente le n maximal tel qu'un algorithme nécessitant $f(n)$ opérations élémentaires pour s'exécuter sur une entrée de taille n termine en moins d'une minute.

On suppose que l'algorithme exécute 10^9 opérations élémentaires par seconde.

Remarquez le fossé entre les algorithmes linéaires, quadratiques, ou cubiques (ou plus généralement polynomiaux) et les algorithmes au moins exponentielles (les trois dernières).

Ces derniers ne sont en pratique utilisables que pour de très petites valeurs de n.

Différents types de complexité

Taille des entrées

Considérons un algorithme pour lequel on peut associer à chaque entrée une notion de taille (par exemple, le nombre d'éléments d'un tableau). Notre algorithme n'aura peut être pas la même complexité pour toutes les entrées d'une certaine taille.

Pour $n \in \mathbb{N}$, on note I_n l'ensemble des entrées de taille n de cet algorithme. Pour une entrée e, on note :

- $t(e)$ le nombre d'opérations élémentaires effectuées par l'algorithme sur l'entrée e ;
- $s(e)$ l'espace mémoire utilisé par l'algorithme sur l'entrée e (sans compter la taille des entrées).

Différents types de complexité

Complexité dans le pire des cas

La plupart du temps, on cherche une borne supérieure de la complexité. On parle alors du **pire cas** de l'algorithme.

On appelle :

- **complexité temporelle dans le pire des cas** la suite : $C_{\text{pire}}(n) = \max_{e \in I_n} t(e)$.
- **complexité spatiale dans le pire des cas** la suite : $C^s_{\text{pire}}(n) = \max_{e \in I_n} s(e)$.

Différents types de complexité

Autres types de complexité

Si deux algorithmes résolvant le même problème ont la même complexité dans le pire des cas, on cherchera parfois à les départager en étudiant d'autres types de complexité.

Différents types de complexité

Complexité dans le meilleur des cas

On peut alors regarder une borne inférieure sur la complexité.

On appelle :

- **complexité temporelle dans le meilleur des cas** la suite : $C_{\text{meilleur}}(n) = \min_{e \in I_n} t(e)$.
- **complexité spatiale dans le meilleur des cas** la suite : $C^s_{\text{meilleur}}(n) = \min_{e \in I_n} s(e)$.

Différents types de complexité

Complexité en moyenne

Une notion plus intéressante en pratique (mais plus difficile à calculer) est de considérer que toutes les entrées de taille n on la même probabilité d'être passées en argument de notre fonction, et d'analyser ce qu'il va se passer **en moyenne**.

On appelle :

- **complexité temporelle en moyenne** la suite : $C_{\text{moyenne}}(n) = \frac{1}{|I_n|} \sum_{e \in I_n} t(e)$.
- **complexité spatiale en moyenne** la suite : $C^s_{\text{moyenne}}(n) = \frac{1}{|I_n|} \sum_{e \in I_n} s(e)$.

Différents types de complexité

Complexité amortie

Dans le cadre de l'étude des structures de données (prochain chapitre), il est fréquent d'avoir la situation suivante :

- un algorithme a une très bonne complexité dans le meilleur des cas, mais une mauvaise complexité dans le pire des cas ;
- on sait que si le pire des cas se produit une fois, il ne se produira plus les $n-1$ prochaines fois (on aura donc le meilleur cas pour les $n-1$ prochains appels à la fonction).

Pour analyser ce genre de situation, on aura alors recours à la **complexité amortie** :

$$C_{\text{amortie}}(n) = \frac{C_{\text{pire}}(n) + (n-1)C_{\text{meilleur}}(n)}{n}$$

Conclusion

Dernière remarque

Utilisation de fonctions importées

Il nous arrivera d'utiliser des fonctions disponibles dans des librairies. Dans ce cas, pour analyser la complexité de nos programmes, il ne faudra pas oublier de compter ces fonctions, qui n'auront pas forcément une complexité constante.