ITMO UNIVERSITY

How to Win Coding Competitions: Secrets of Champions

Week 3: Sorting and Search Algorithms
Lecture 6: Mergesort

Maxim Buzdalov
Saint Petersburg 2016

Another Divide-and-Conquer algorithm

Another Divide-and-Conquer algorithm

- Works recursively

Another Divide-and-Conquer algorithm

- Works recursively
- If subarray size is 1, do nothing

Another Divide-and-Conquer algorithm

- Works recursively
- If subarray size is 1, do nothing
- Otherwise:
  - Split the array into two (nearly) equal halves
  - Sort these halves recursively
  - Merge the results into a sorted subarray

Another Divide-and-Conquer algorithm

- Works recursively
- If subarray size is 1, do nothing
- Otherwise:
  - Split the array into two (nearly) equal halves
  - Sort these halves recursively
  - Merge the results into a sorted subarray
- Differences to Quicksort

Another Divide-and-Conquer algorithm

- Works recursively
- If subarray size is 1, do nothing
- Otherwise:
  - Split the array into two (nearly) equal halves
  - Sort these halves recursively
  - Merge the results into a sorted subarray
- Differences to Quicksort
  - Quicksort: split into two parts such that $L \preceq R$, sort recursively
  - Mergesort: split into two parts, sort recursively, merge answers

Another Divide-and-Conquer algorithm

- Works recursively
- If subarray size is 1, do nothing
- Otherwise:
  - Split the array into two (nearly) equal halves
  - Sort these halves recursively
  - Merge the results into a sorted subarray
- Differences to Quicksort
  - Quicksort: split into two parts such that $L \preceq R$, sort recursively
  - Mergesort: split into two parts, sort recursively, merge answers
- This algorithm needs extra scratch memory

```
procedure MergeSort(A, W, ⪯, s, t)
    if s + 1 = t then return end if
    m ← (s + t)/2
    MergeSort(A, W, ⪯, s, m)
    MergeSort(A, W, ⪯, m, t)

    i ← s, j ← m, k ← s
    while i < m or j < t do
        if j = t or (i < m and A[i] ⪯ A[j]) then
            W[k] ← A[i], i ← i + 1, k ← k + 1
        else
            W[k] ← A[j], j ← j + 1, k ← k + 1
        end if
    end while

    for i ∈ [s; t) do A[i] ← W[i] end for
end procedure
```

procedure $\textsc{Mergesort}(A, W, \preceq, s, t)$
  if $s + 1 = t$ **then return end if**
  $m \leftarrow (s + t)/2$
  $\textsc{Mergesort}(A, W, \preceq, s, m)$
  $\textsc{Mergesort}(A, W, \preceq, m, t)$

  $i \leftarrow s, j \leftarrow m, k \leftarrow s$
  while $i < m$ **or** $j < t$ **do**
    if $j = t$ **or** $(i < m$ **and** $A[i] \preceq A[j])$ **then**
      $W[k] \leftarrow A[i], i \leftarrow i + 1, k \leftarrow k + 1$
    else
      $W[k] \leftarrow A[j], j \leftarrow j + 1, k \leftarrow k + 1$
    end if
  end while

  for $i \in [s; t)$ **do** $A[i] \leftarrow W[i]$ **end for**
end procedure

- Attention: the ending index is exclusive unlike quicksort

procedure $\textsc{Mergesort}(A, W, \preceq, s, t)$
    if $s + 1 = t$ **then return end if**
    $m \leftarrow (s + t)/2$
    $\textsc{Mergesort}(A, W, \preceq, s, m)$
    $\textsc{Mergesort}(A, W, \preceq, m, t)$

    $i \leftarrow s, j \leftarrow m, k \leftarrow s$
    while $i < m$ **or** $j < t$ **do**
        if $j = t$ **or** $(i < m$ **and** $A[i] \preceq A[j])$ **then**
            $W[k] \leftarrow A[i], i \leftarrow i + 1, k \leftarrow k + 1$
        else
            $W[k] \leftarrow A[j], j \leftarrow j + 1, k \leftarrow k + 1$
        end if
    end while

    for $i \in [s; t)$ **do** $A[i] \leftarrow W[i]$ **end for**
end procedure

- Attention: the ending index is exclusive unlike quicksort
- Split into two equal halves (up to ± 1)

```
procedure MergeSort(A, W, ⪯, s, t)
    if s + 1 = t then return end if
    m ← (s + t)/2
    MergeSort(A, W, ⪯, s, m)
    MergeSort(A, W, ⪯, m, t)

    i ← s, j ← m, k ← s
    while i < m or j < t do
        if j = t or (i < m and A[i] ⪯ A[j]) then
            W[k] ← A[i], i ← i + 1, k ← k + 1
        else
            W[k] ← A[j], j ← j + 1, k ← k + 1
        end if
    end while

    for i ∈ [s; t) do A[i] ← W[i] end for
end procedure
```

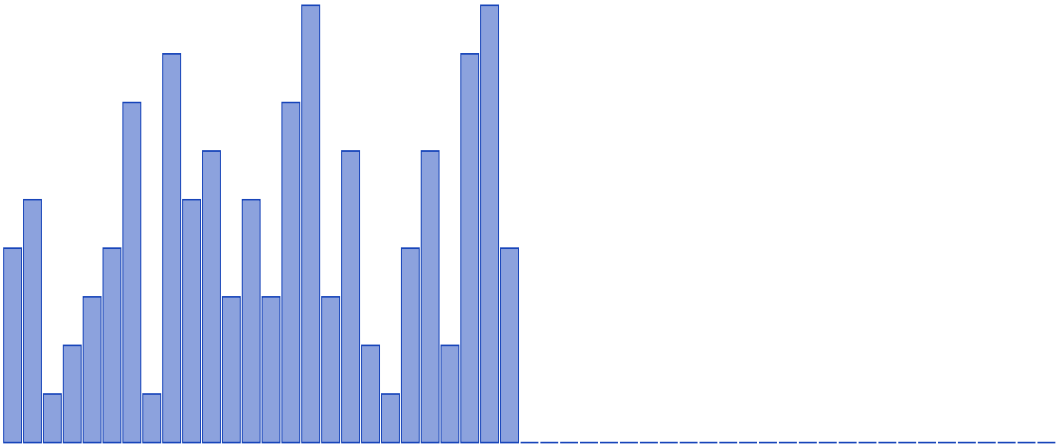
- Attention: the ending index is exclusive unlike quicksort
- Split into two equal halves (up to ± 1)
- Green code is responsible for merging sorted halves

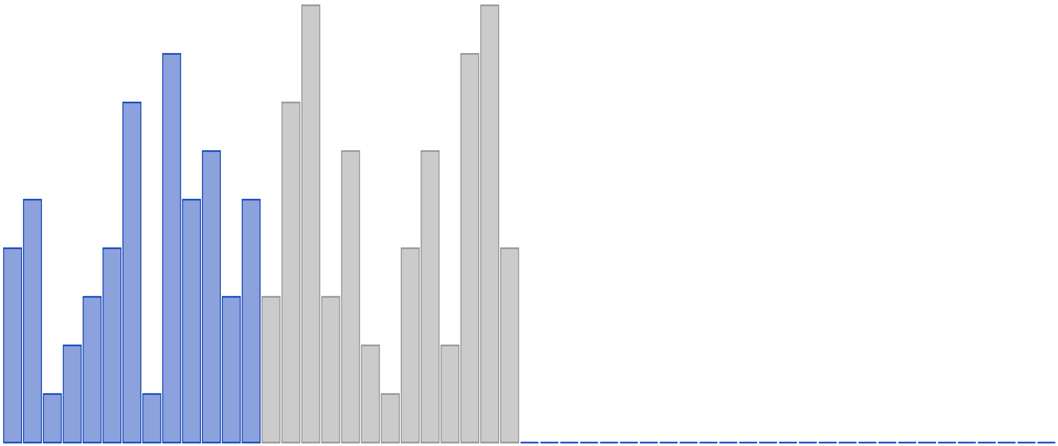
```
procedure MERGESORT(A, W, ≼, s, t)
    if s + 1 = t then return end if
    m ← (s + t)/2
    MERGESORT(A, W, ≼, s, m)
    MERGESORT(A, W, ≼, m, t)

    i ← s, j ← m, k ← s
    while i < m or j < t do
        if j = t or (i < m and A[i] ≼ A[j]) then
            W[k] ← A[i], i ← i + 1, k ← k + 1
        else
            W[k] ← A[j], j ← j + 1, k ← k + 1
        end if
    end while

    for i ∈ [s; t) do A[i] ← W[i] end for
end procedure
```

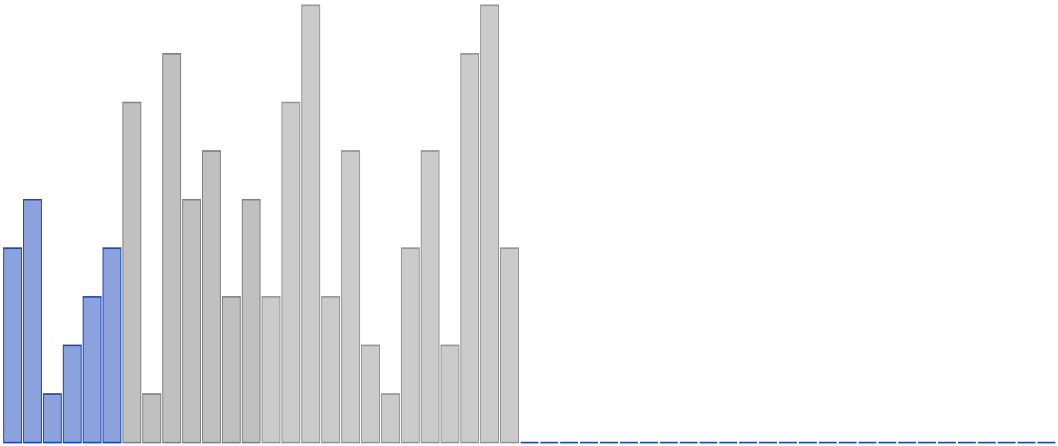
- Attention: the ending index is exclusive unlike quicksort
- Split into two equal halves (up to ± 1)
- Green code is responsible for merging sorted halves
- The scratch array W is used to perform merging easily
  - In-place merge is possible, but is either slower or very complicated

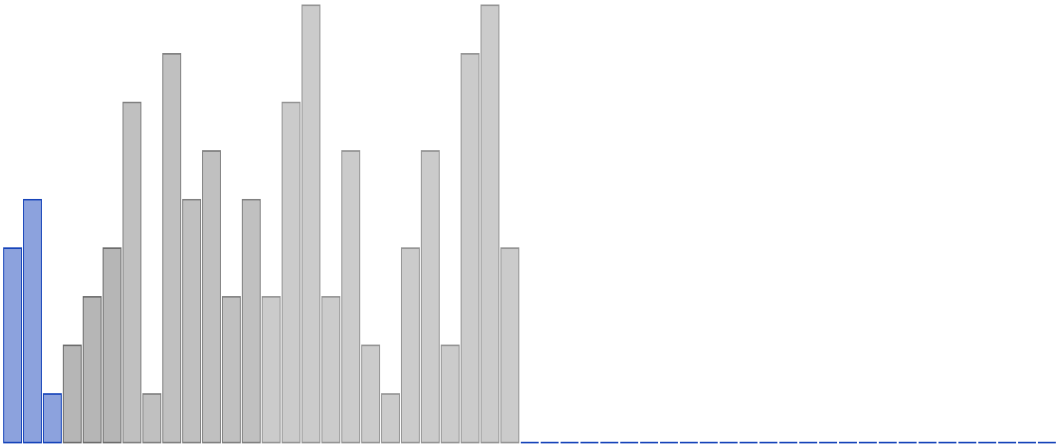
```
procedure MergeSort(A, W, ≼, s, t)
    if s + 1 = t then return end if
    m ← (s + t)/2
    MergeSort(A, W, ≼, s, m)
    MergeSort(A, W, ≼, m, t)

    i ← s, j ← m, k ← s
    while i < m or j < t do
        if j = t or (i < m and A[i] ≼ A[j]) then
            W[k] ← A[i], i ← i + 1, k ← k + 1
        else
            W[k] ← A[j], j ← j + 1, k ← k + 1
        end if
    end while

    for i ∈ [s; t) do A[i] ← W[i] end for
end procedure
```

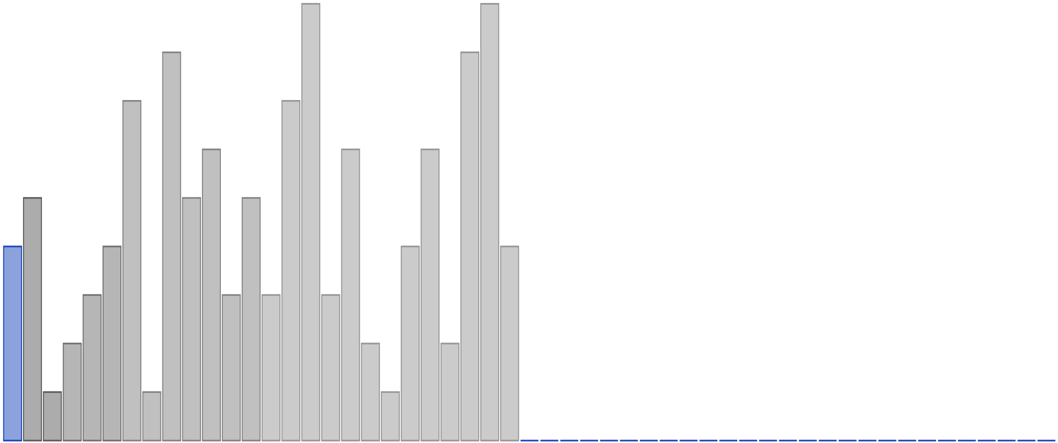
- Attention: the ending index is exclusive unlike quicksort
- Split into two equal halves (up to ± 1)
- Green code is responsible for merging sorted halves
- The scratch array W is used to perform merging easily
  - In-place merge is possible, but is either slower or very complicated
- Merge runs in $\Theta(t - s)$
  - Every scratch element is written exactly once

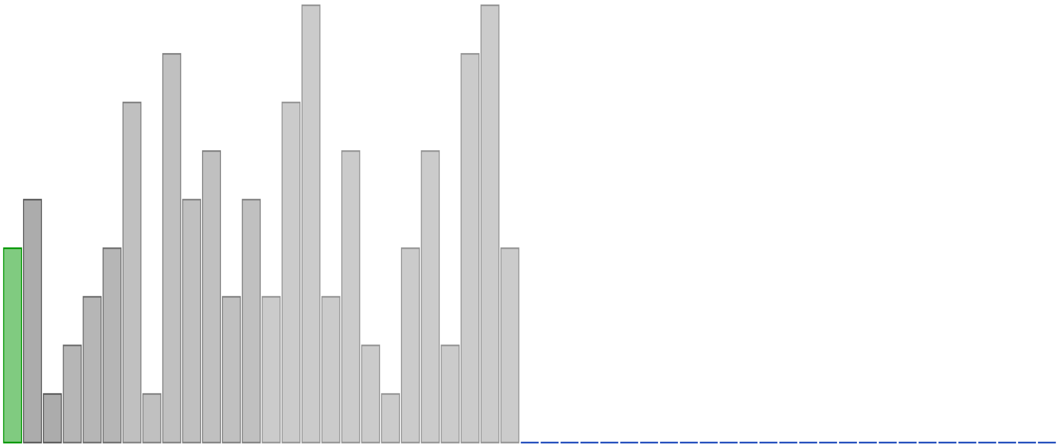


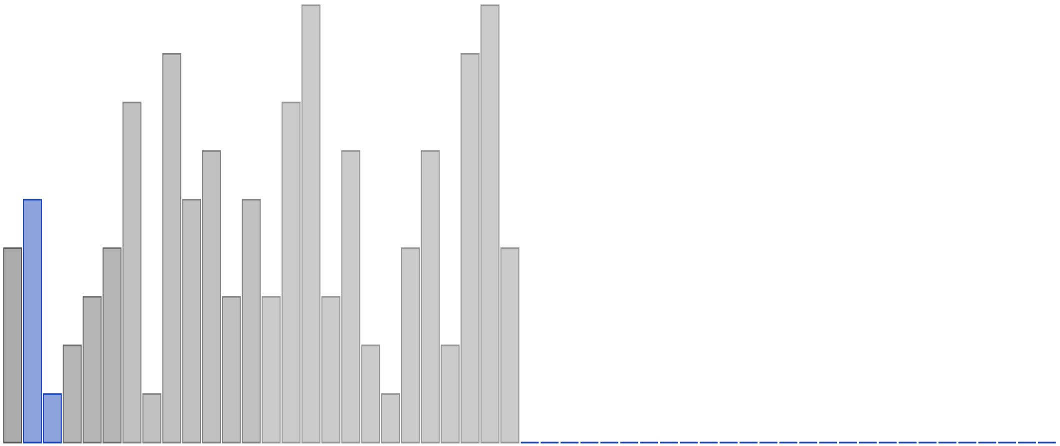


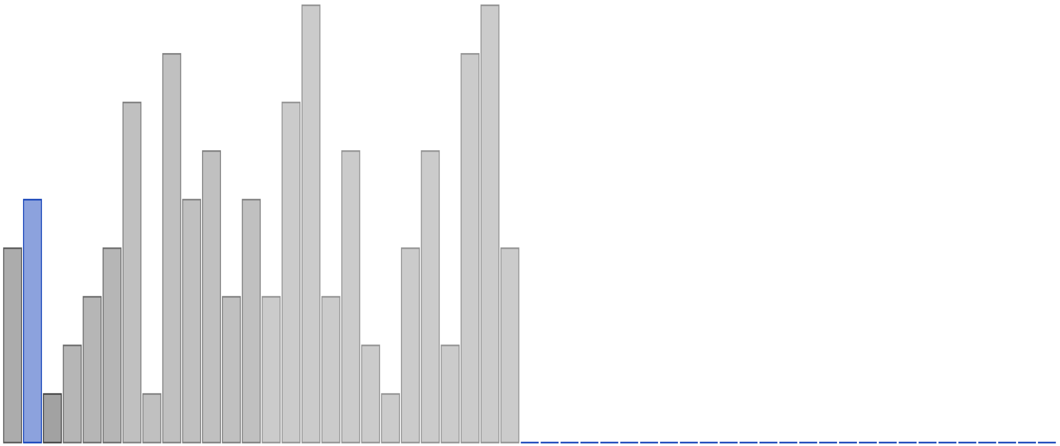


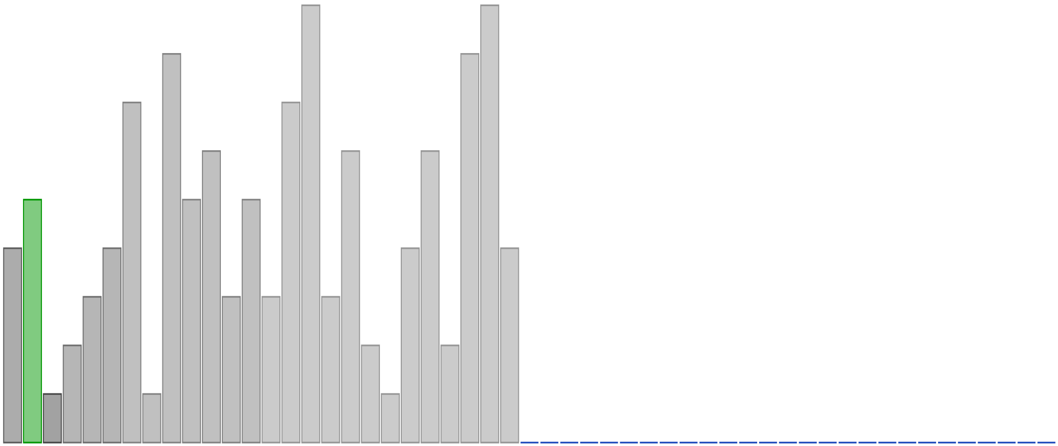


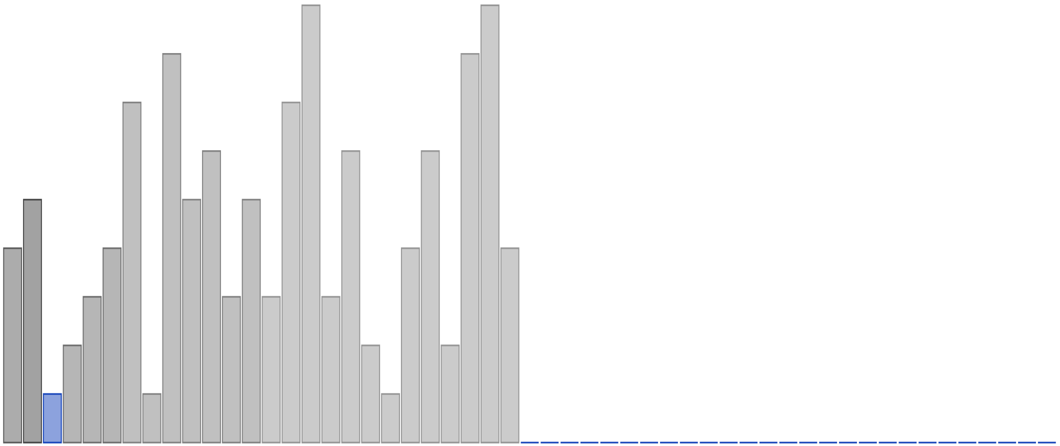


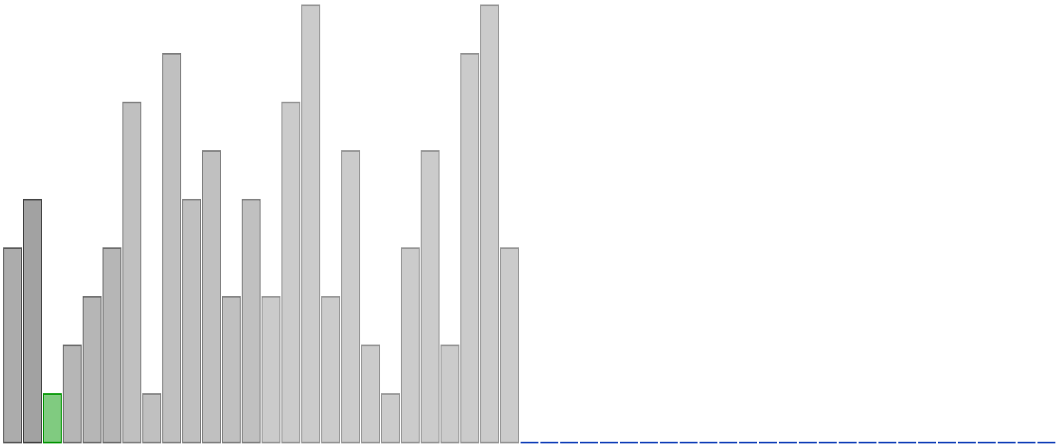


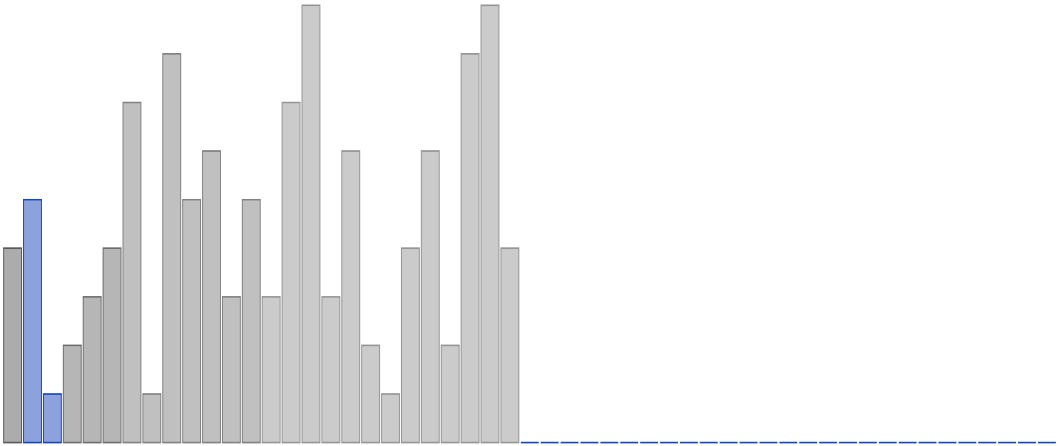


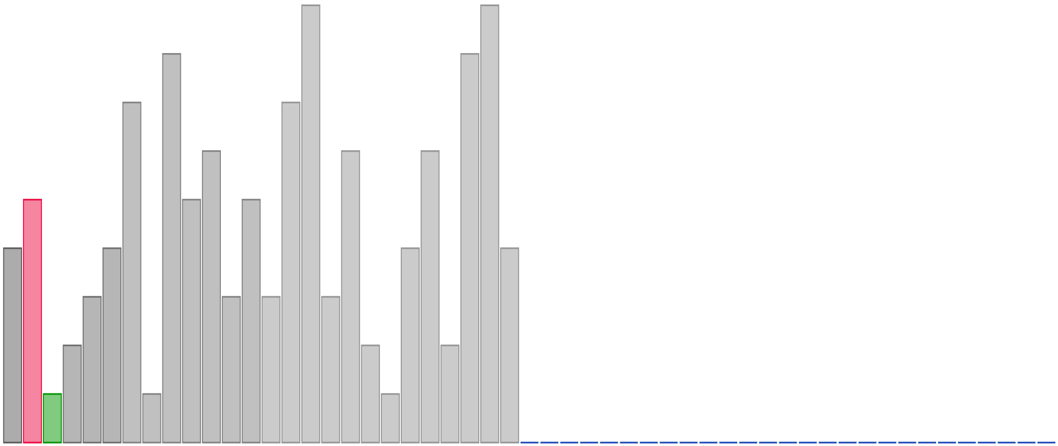


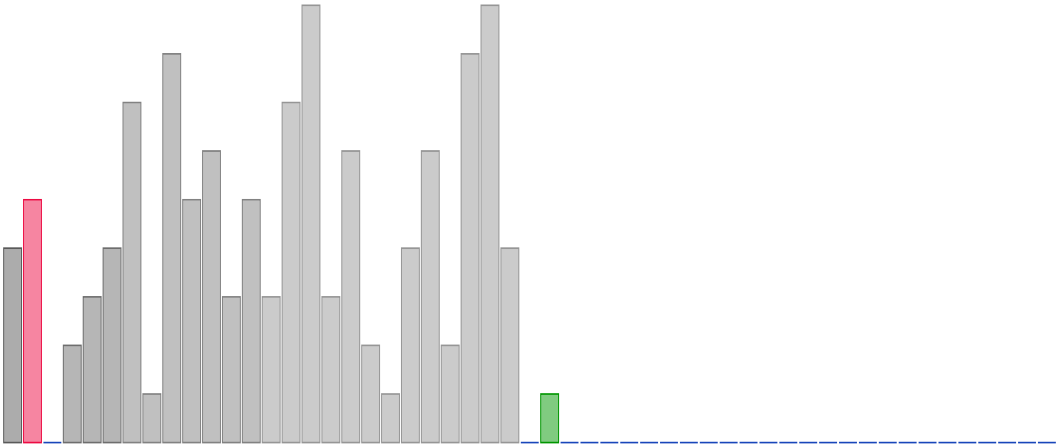


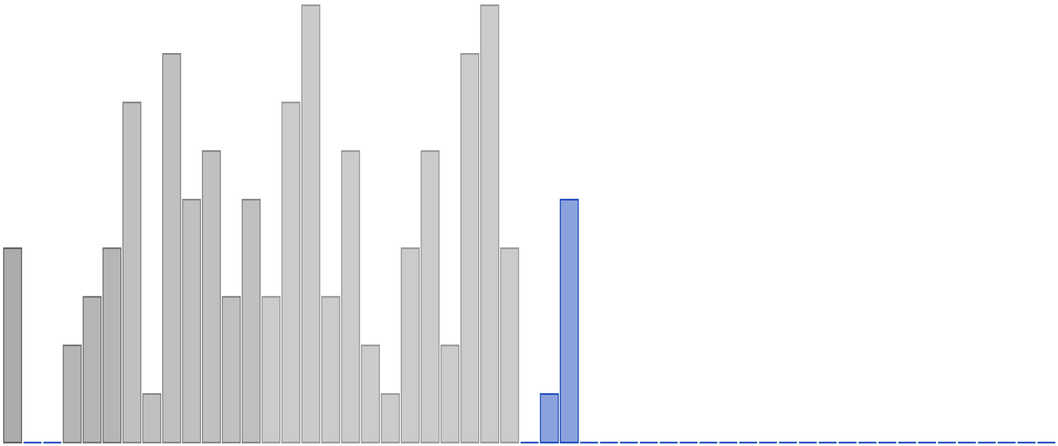


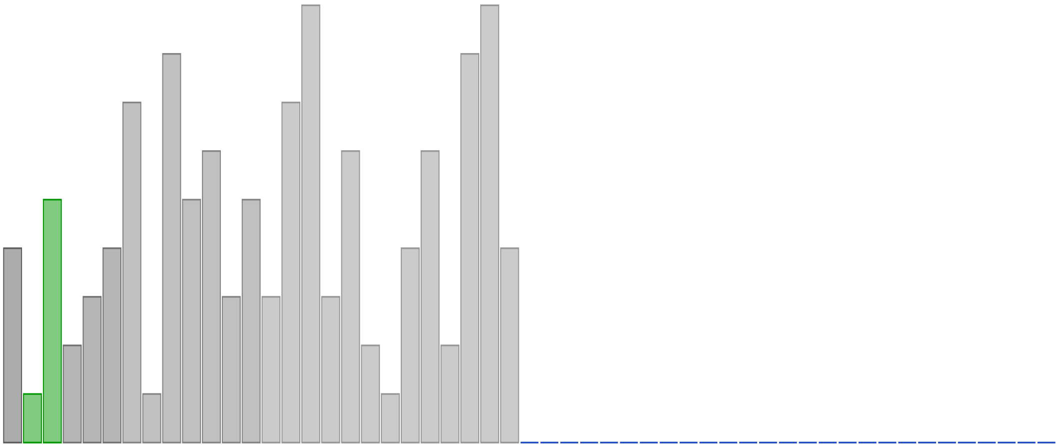


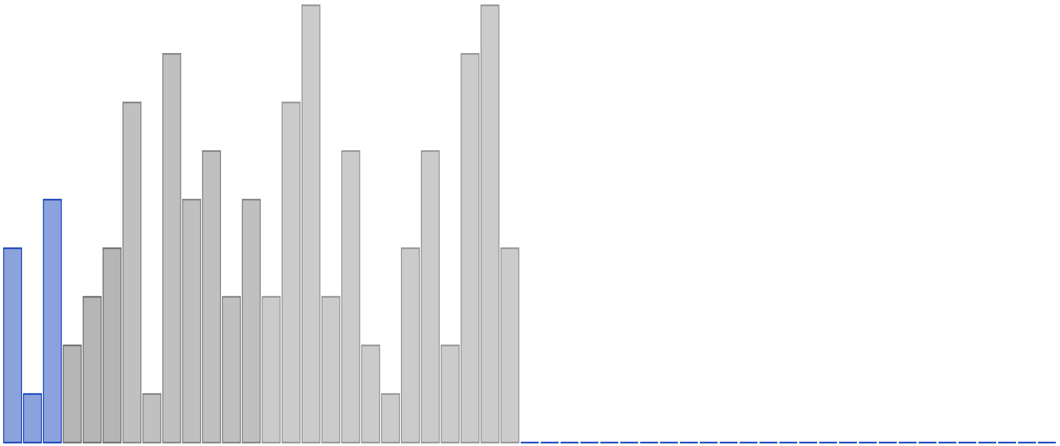


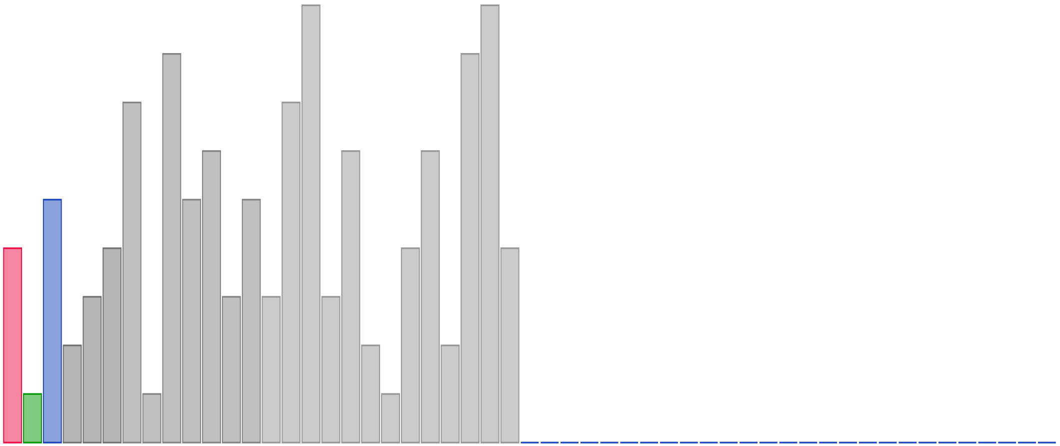


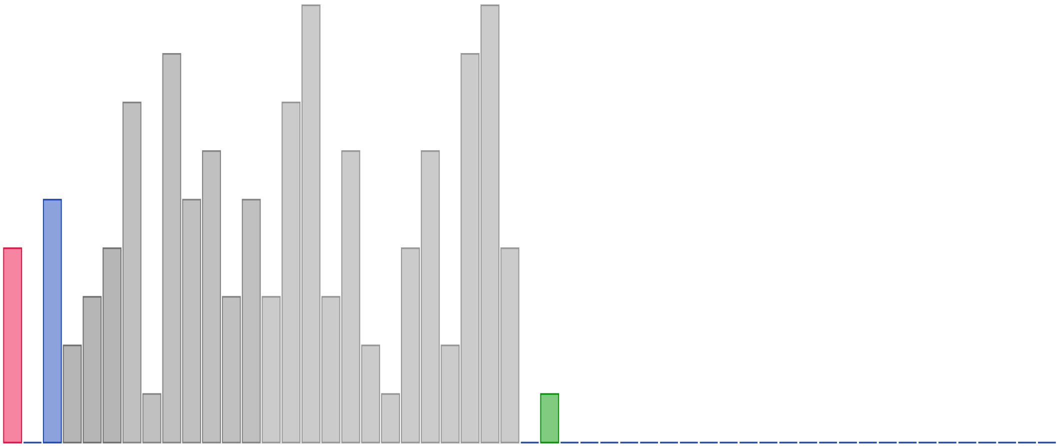


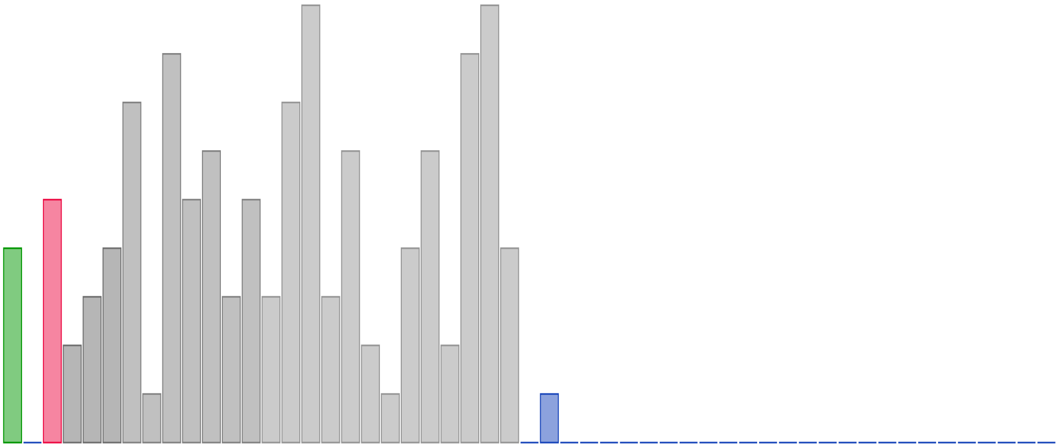


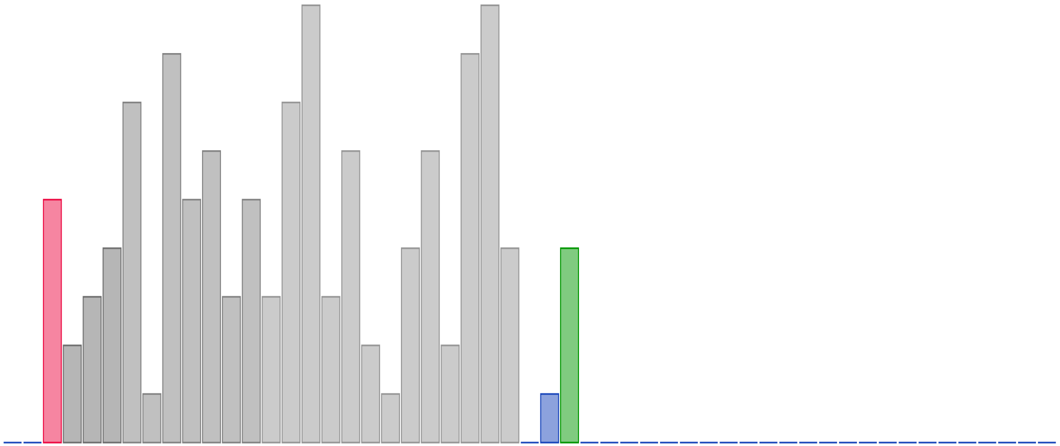


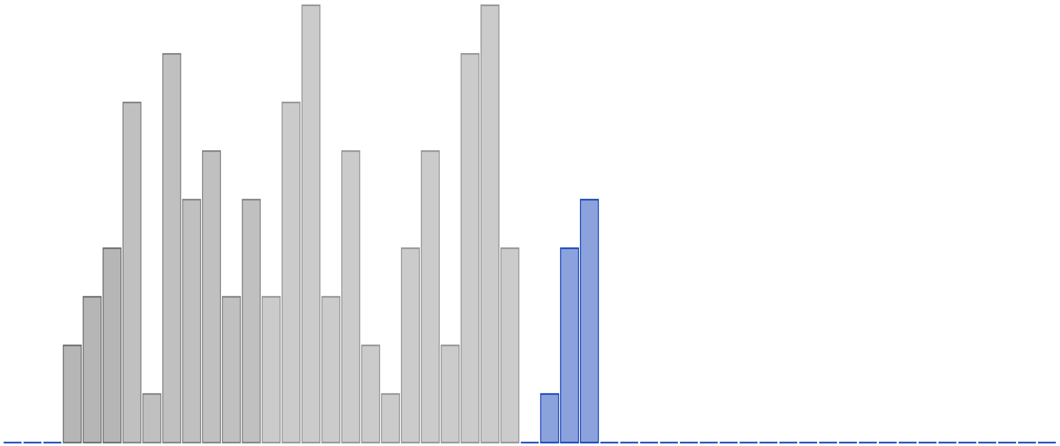


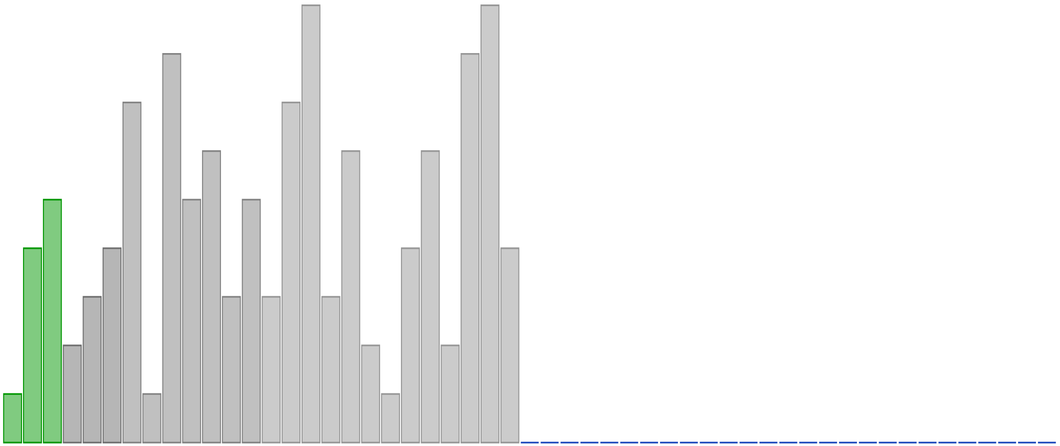


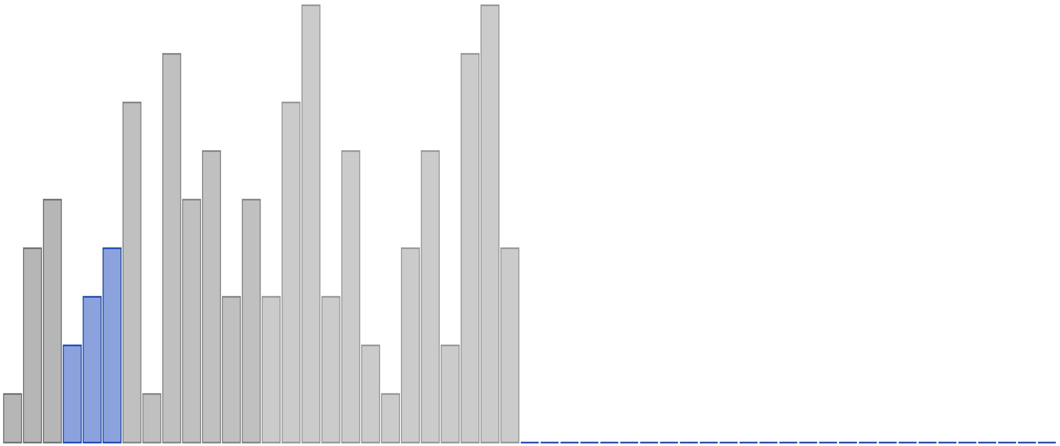


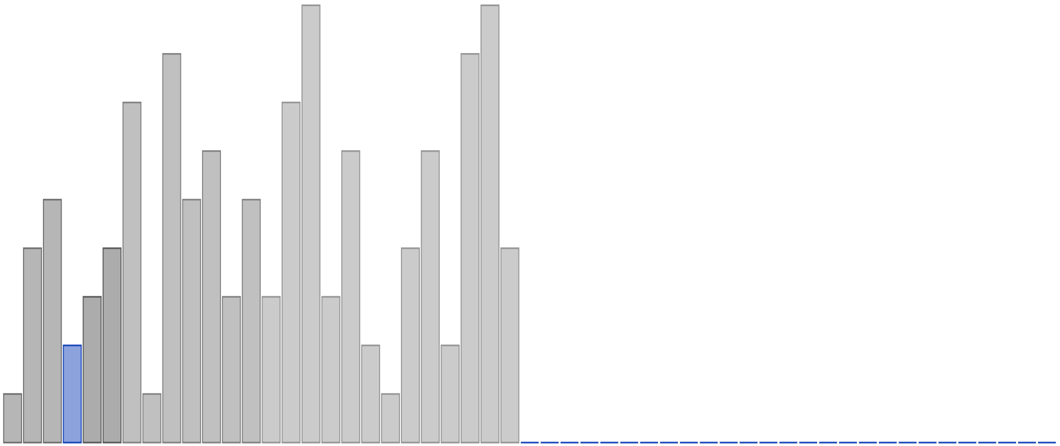


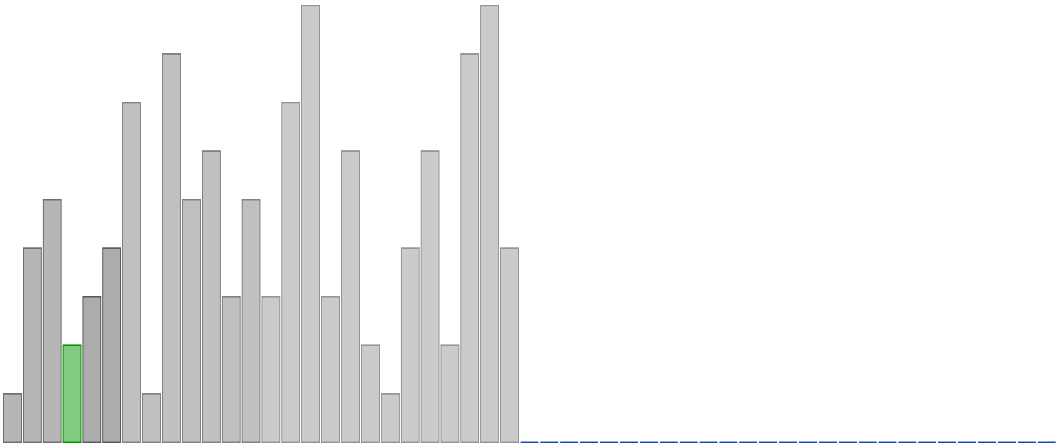


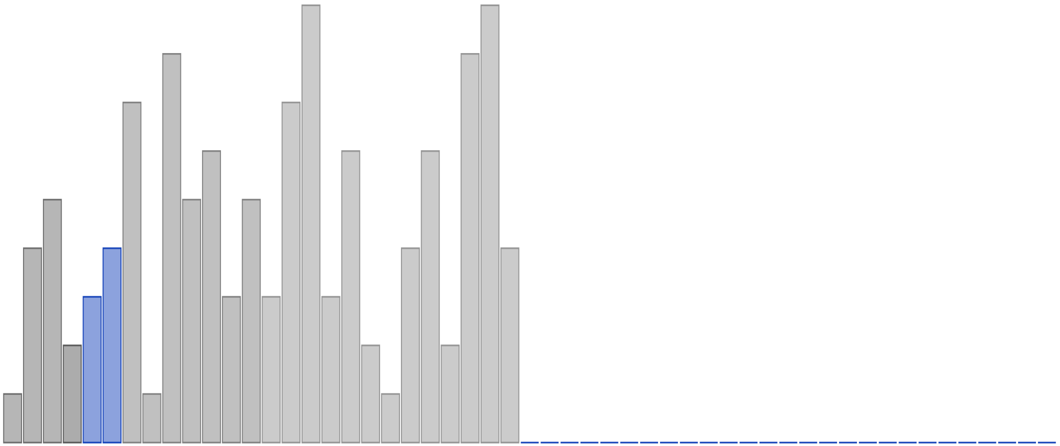


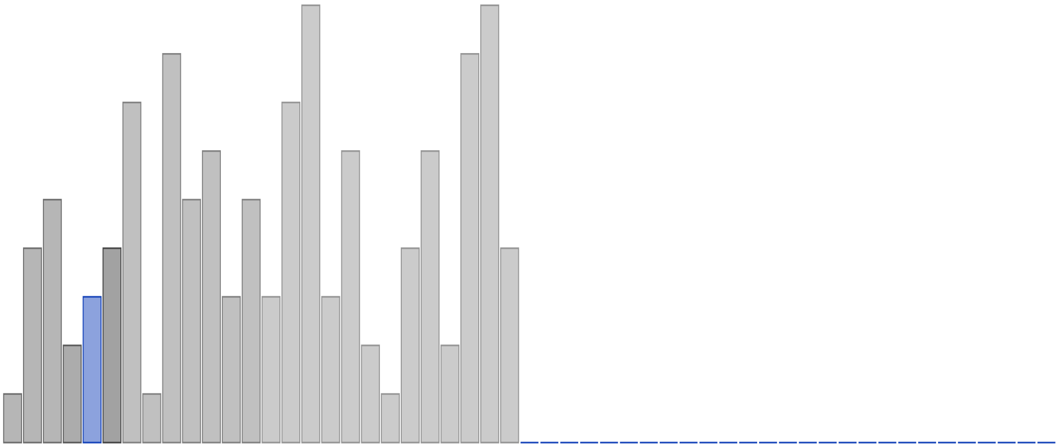


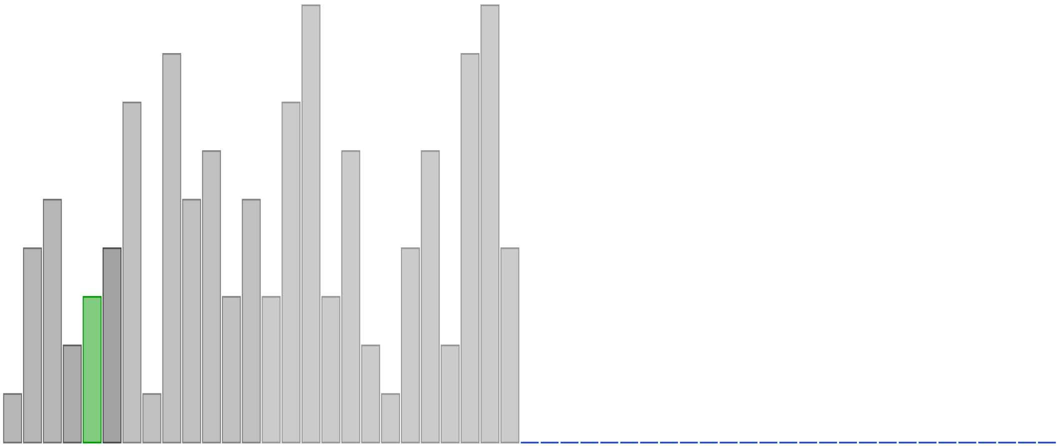


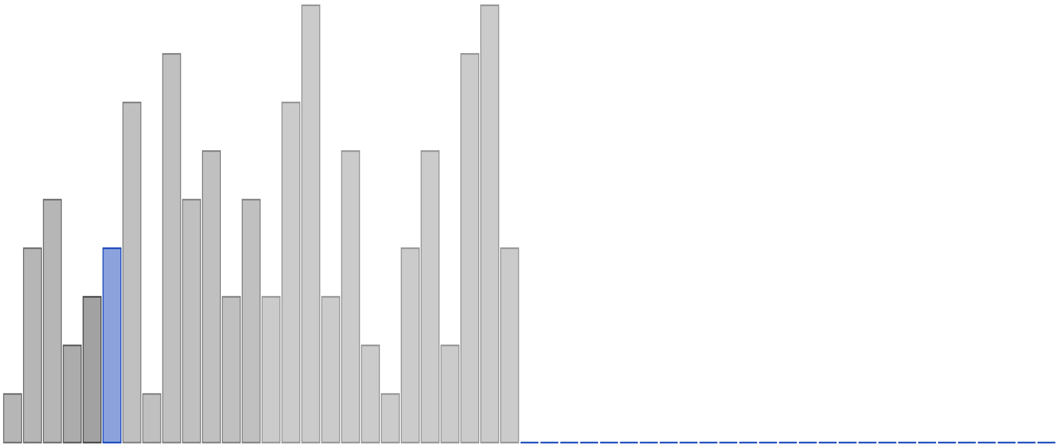


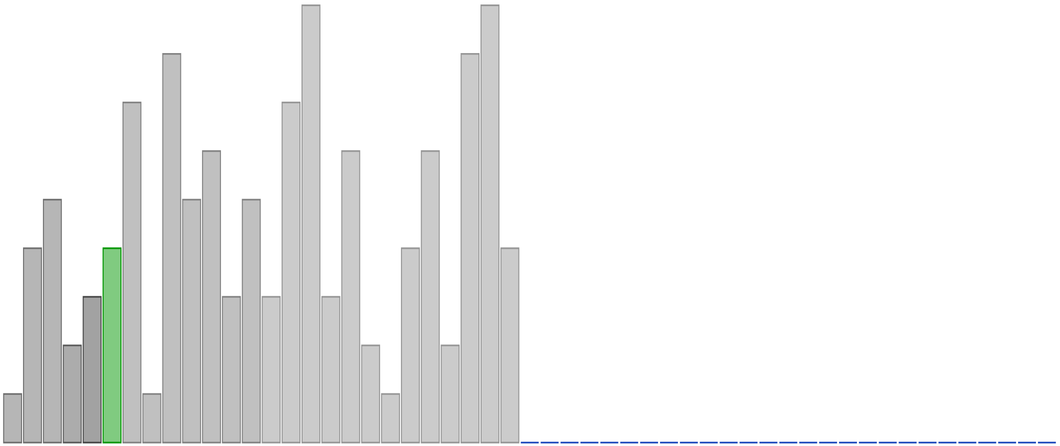


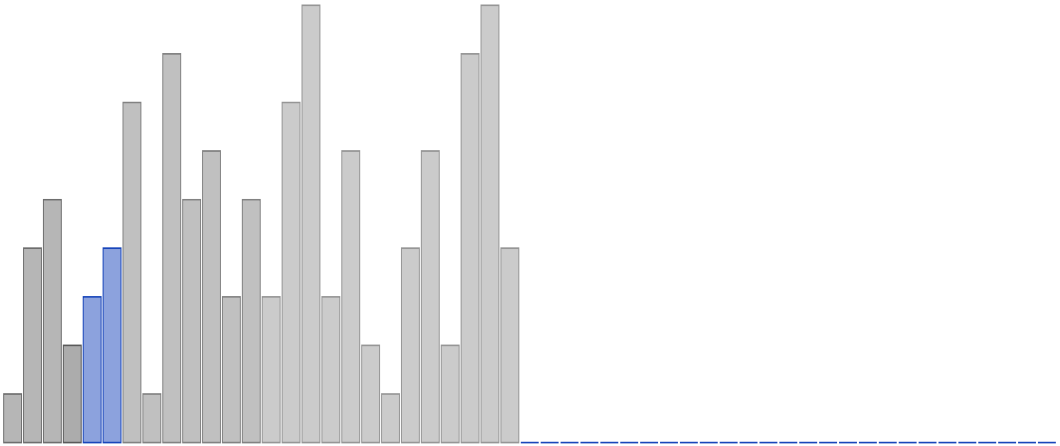


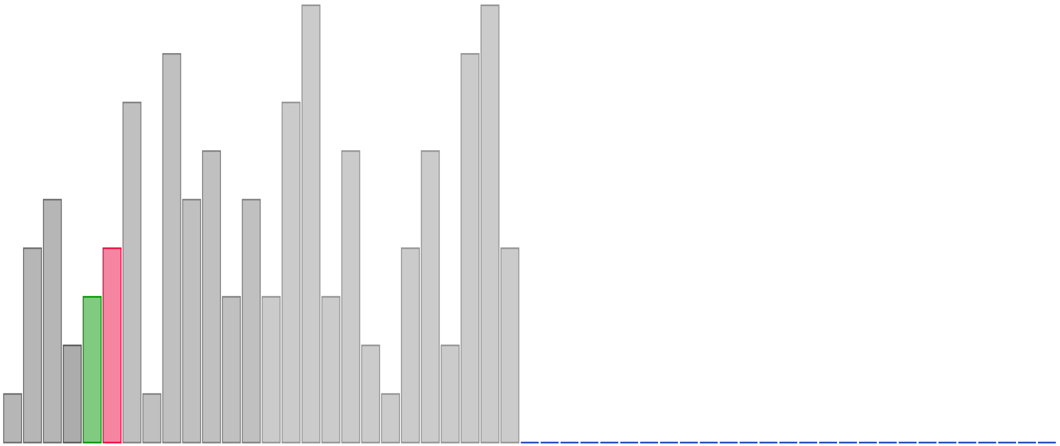


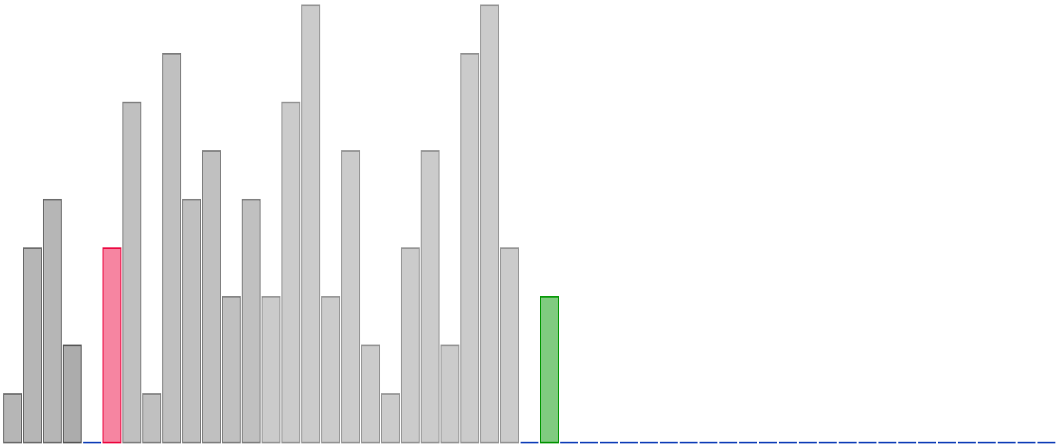


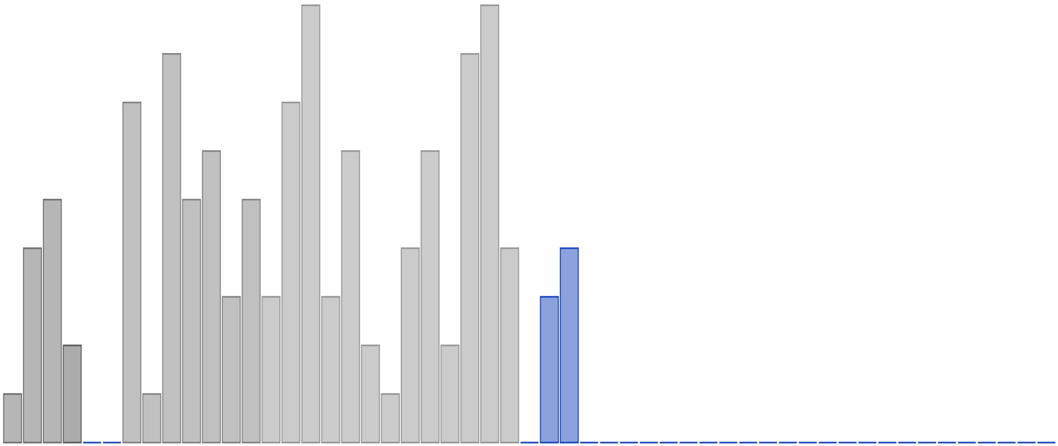


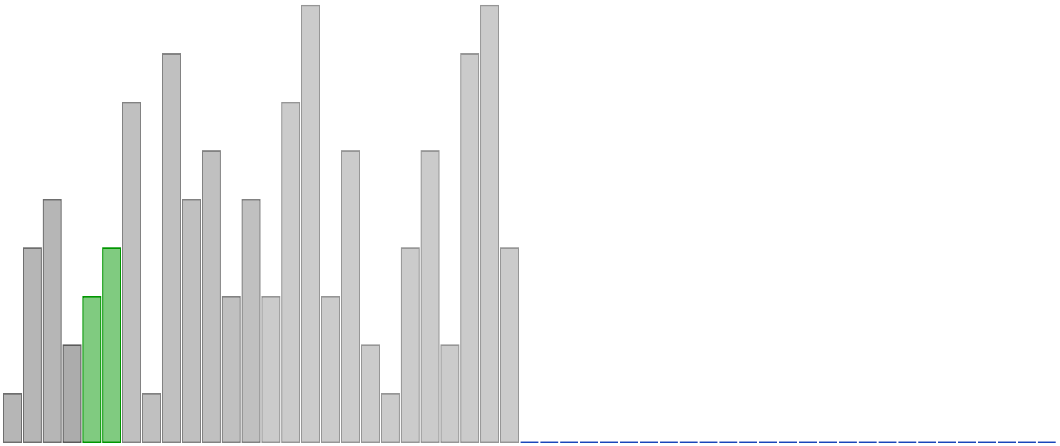


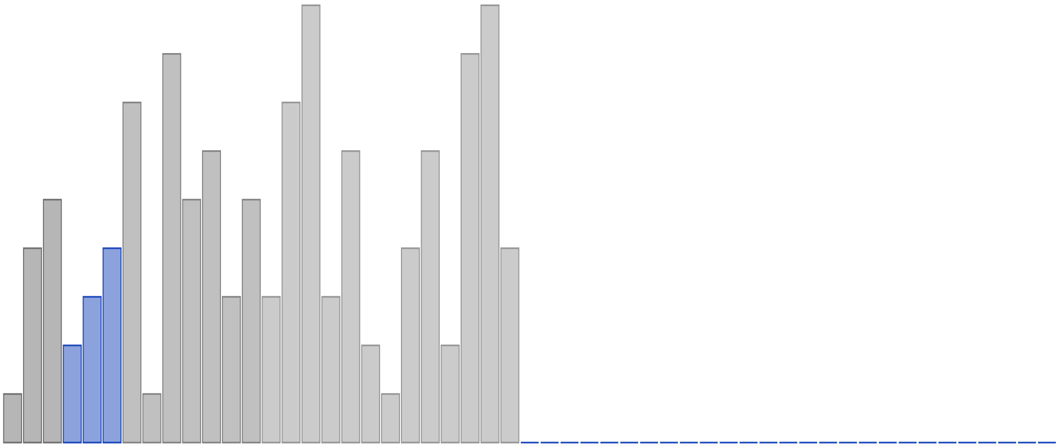


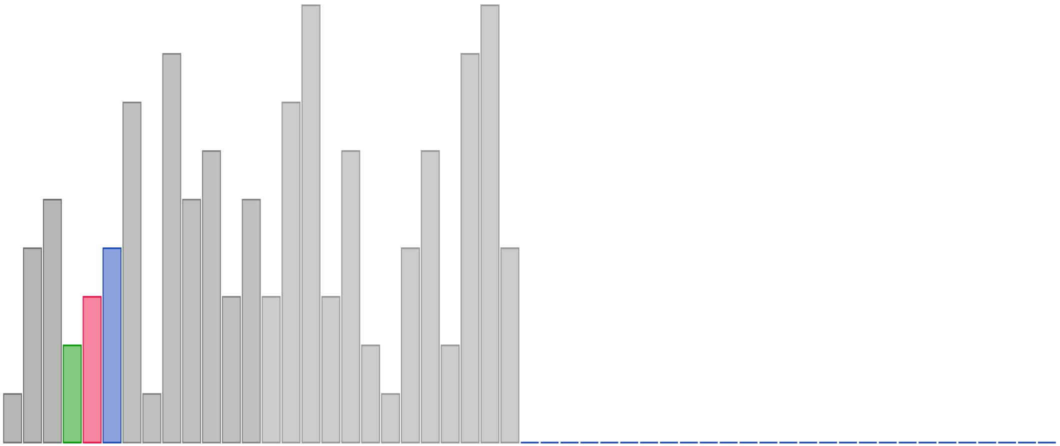


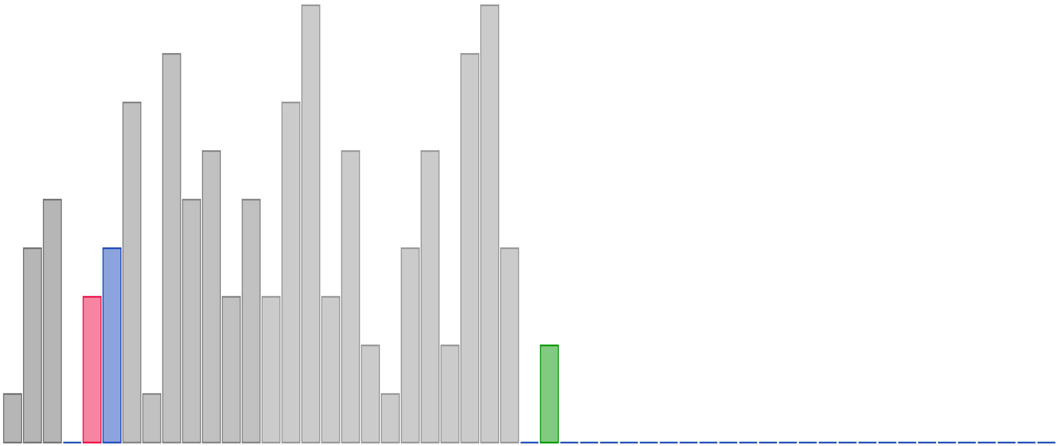


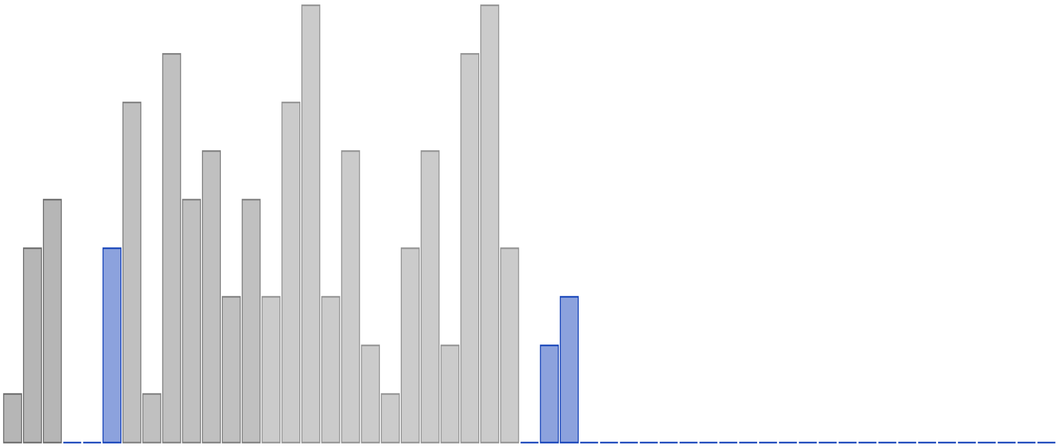


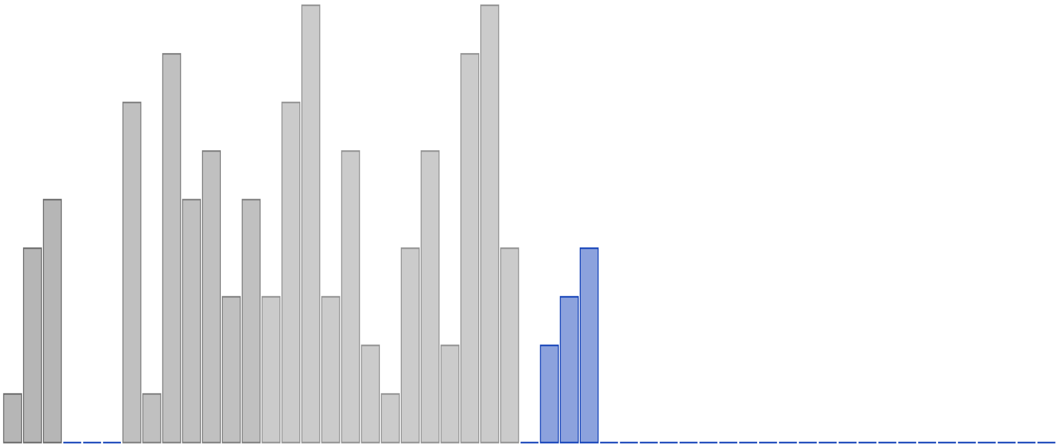


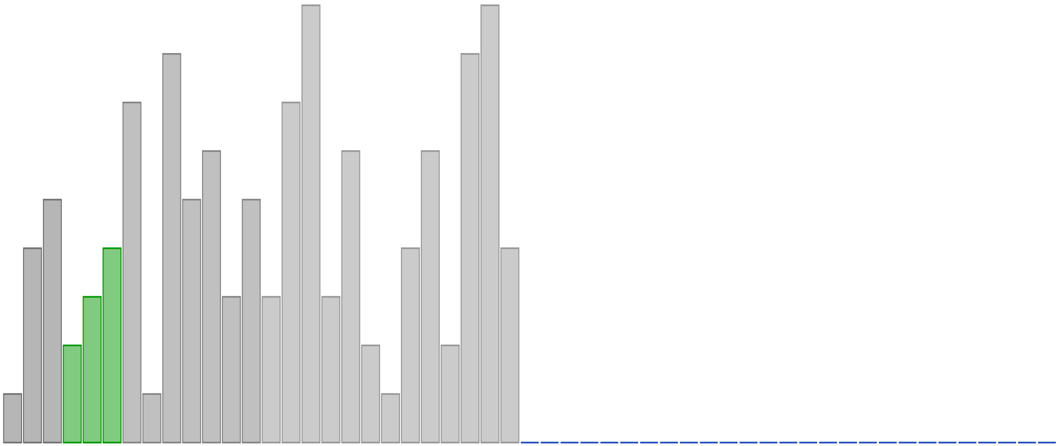


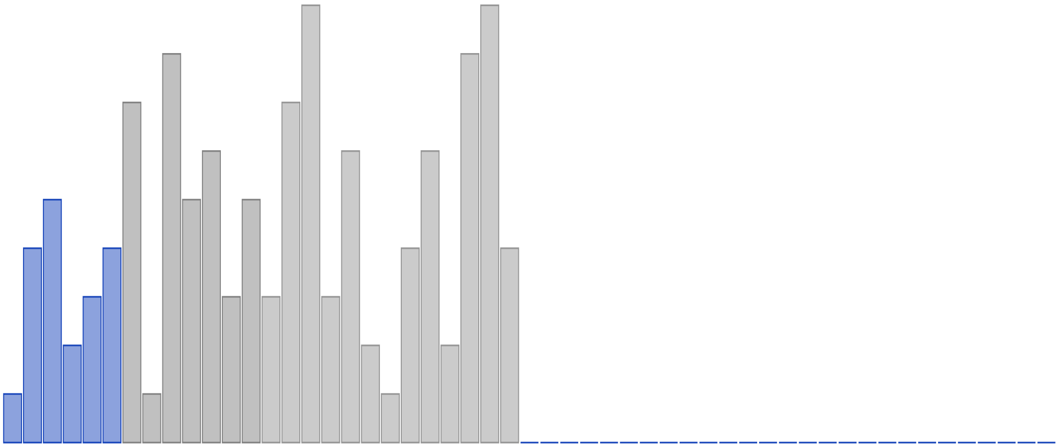


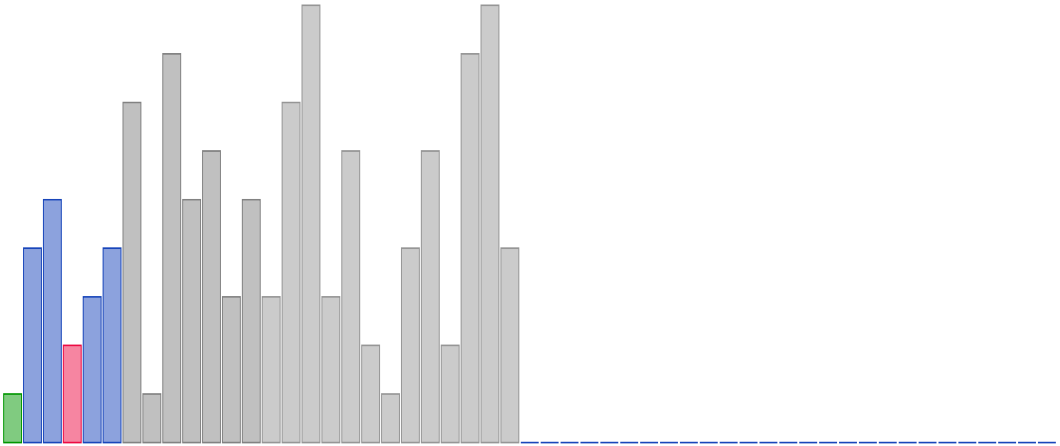


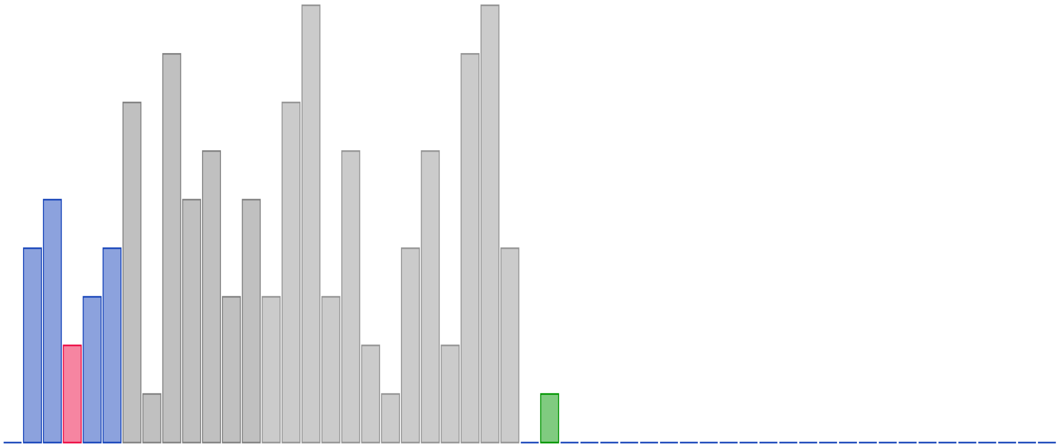


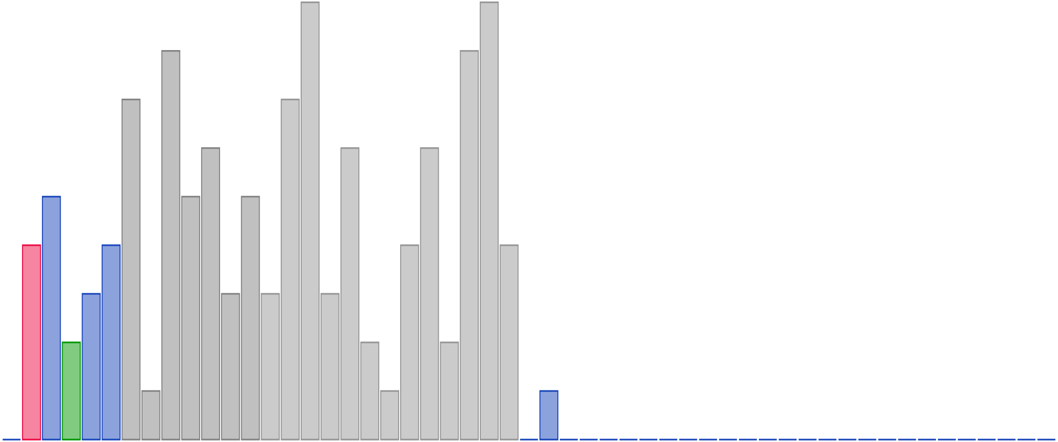


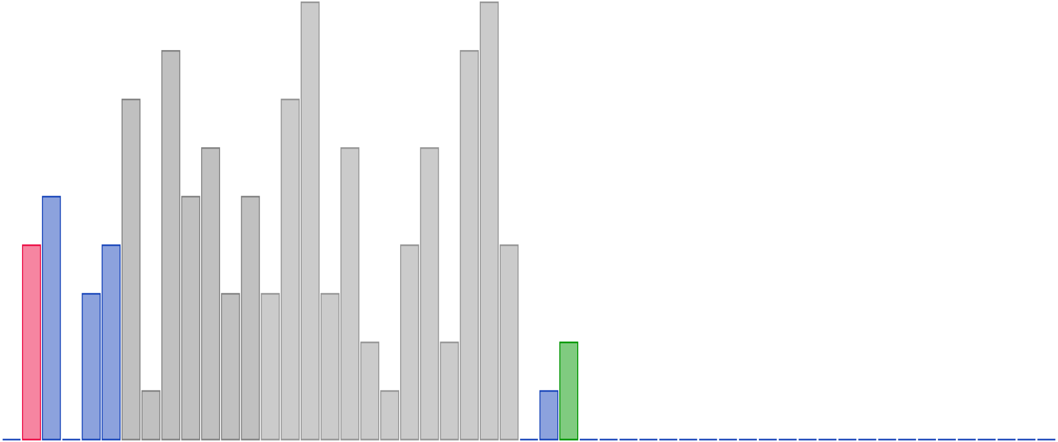


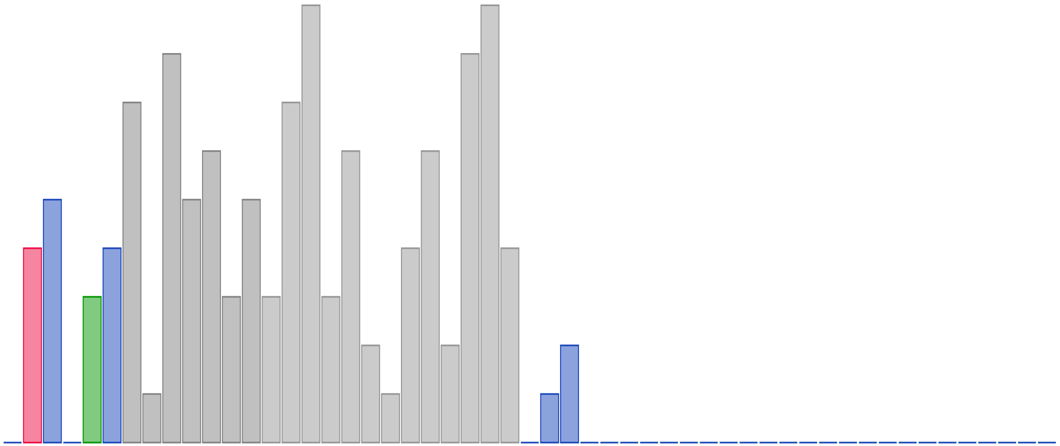


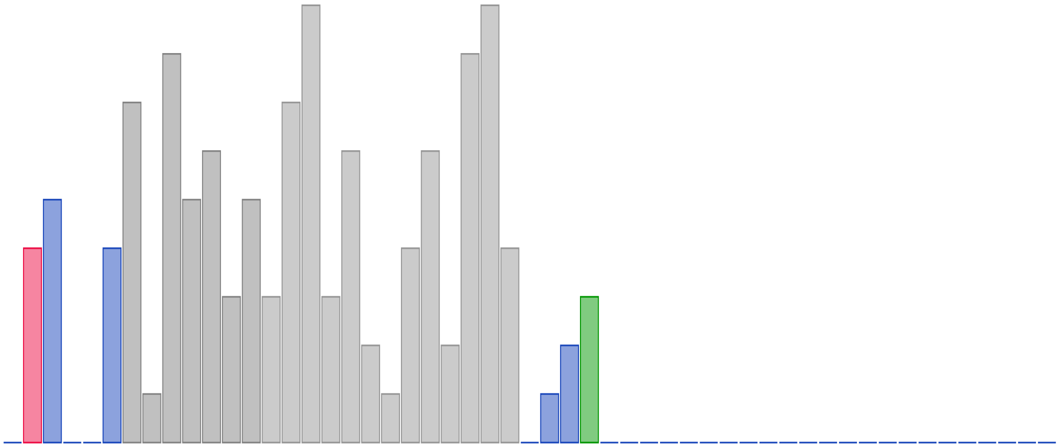


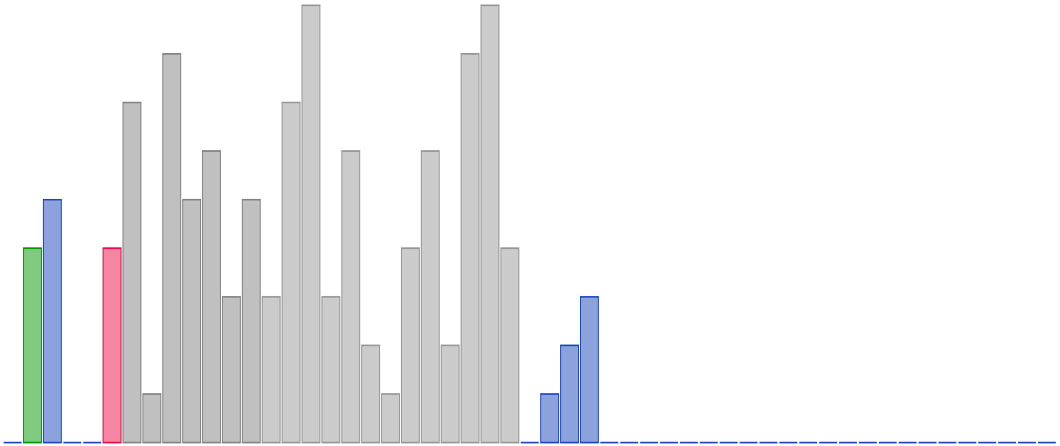


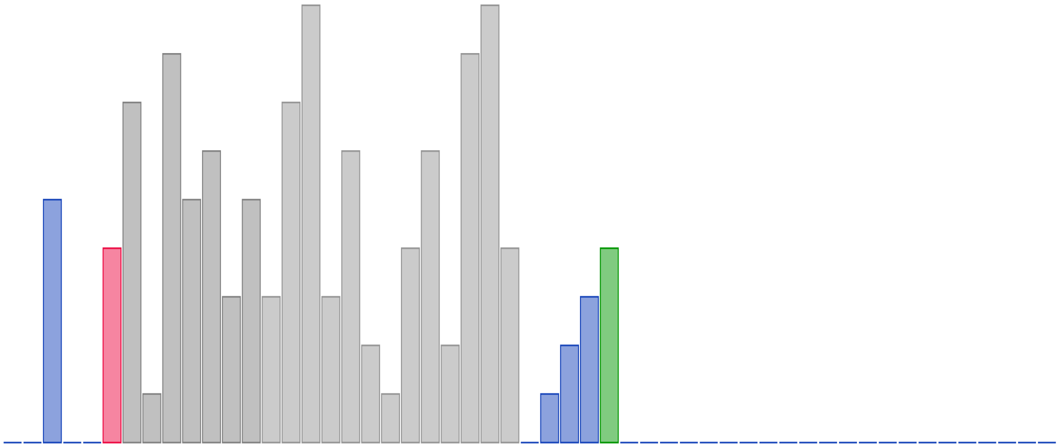


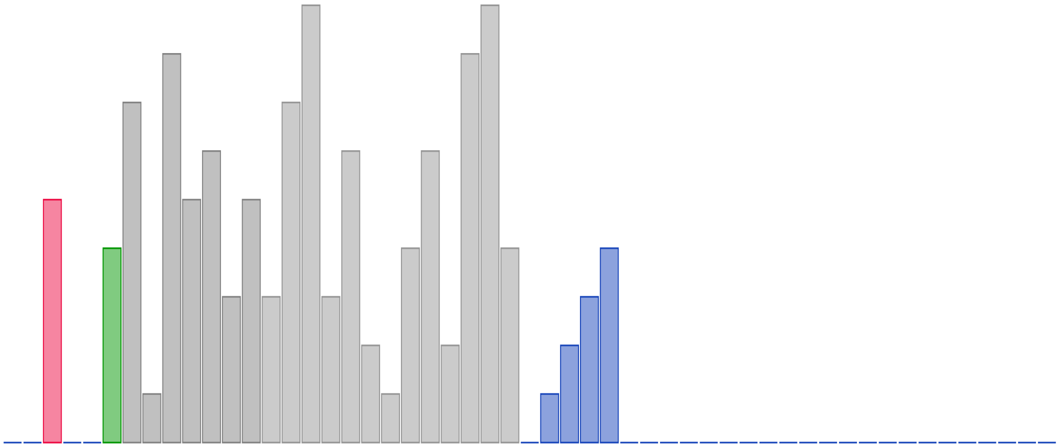


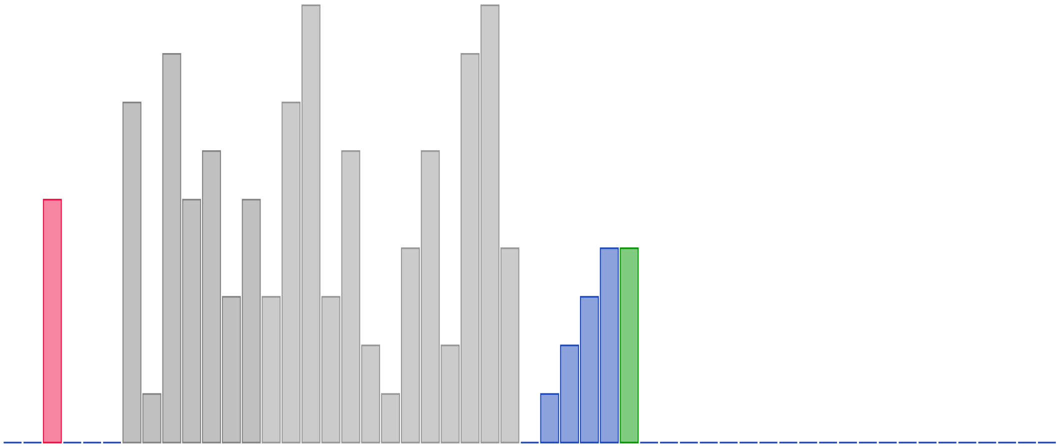


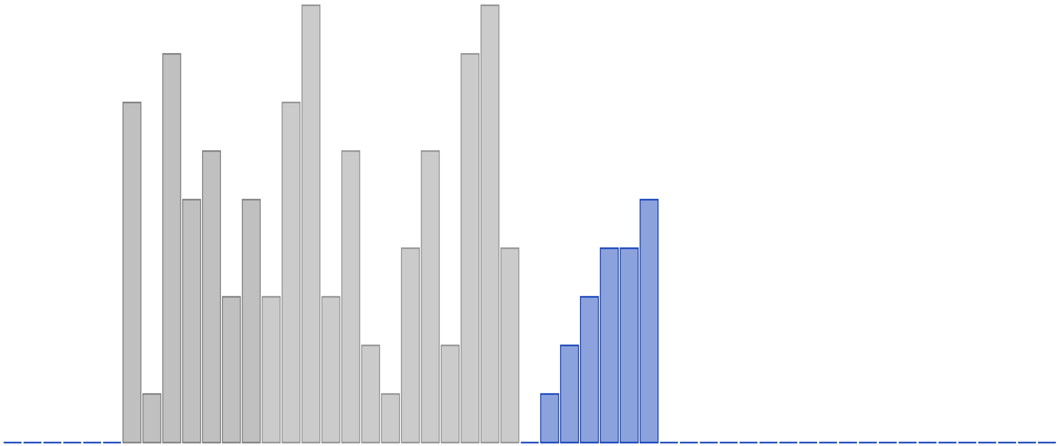


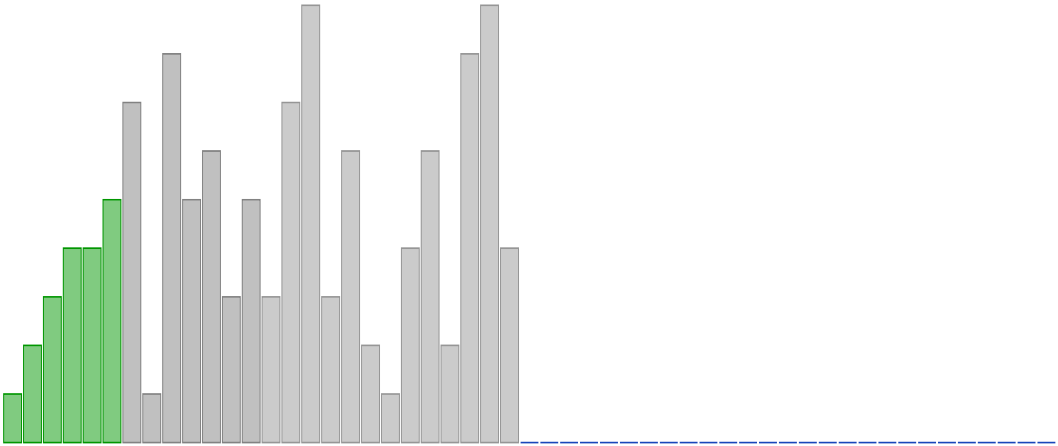


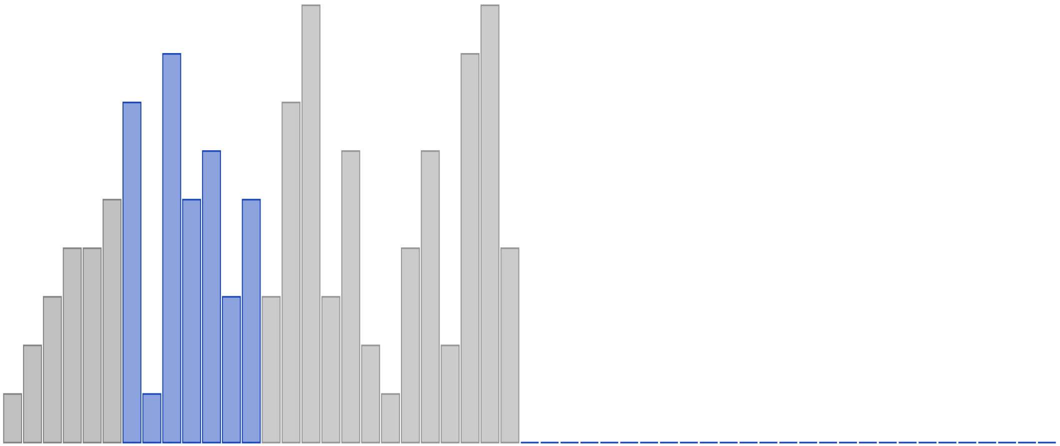


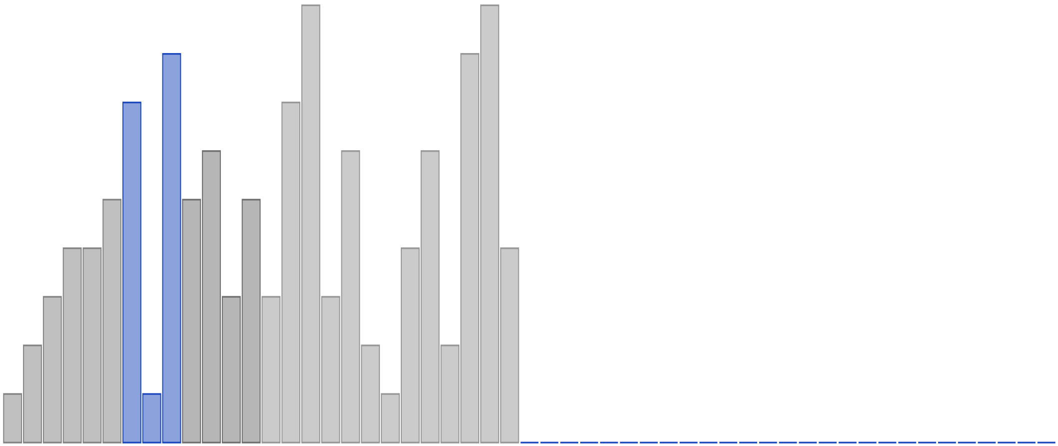


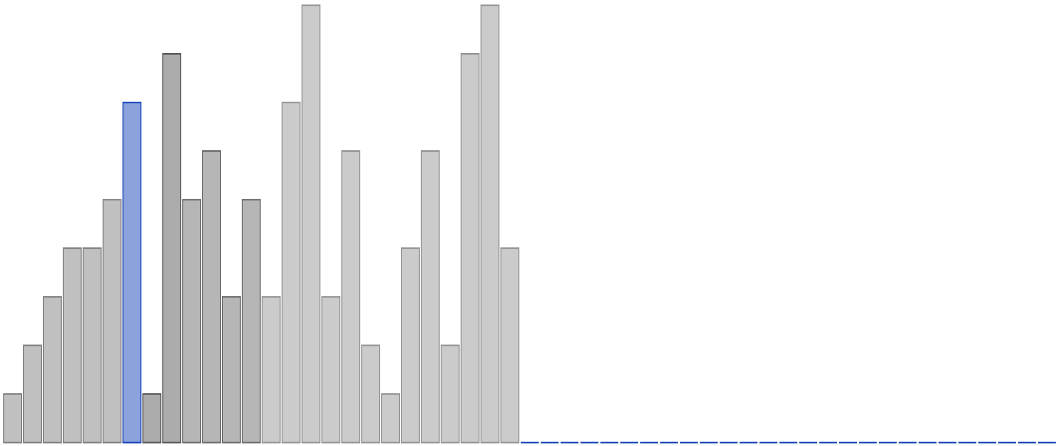


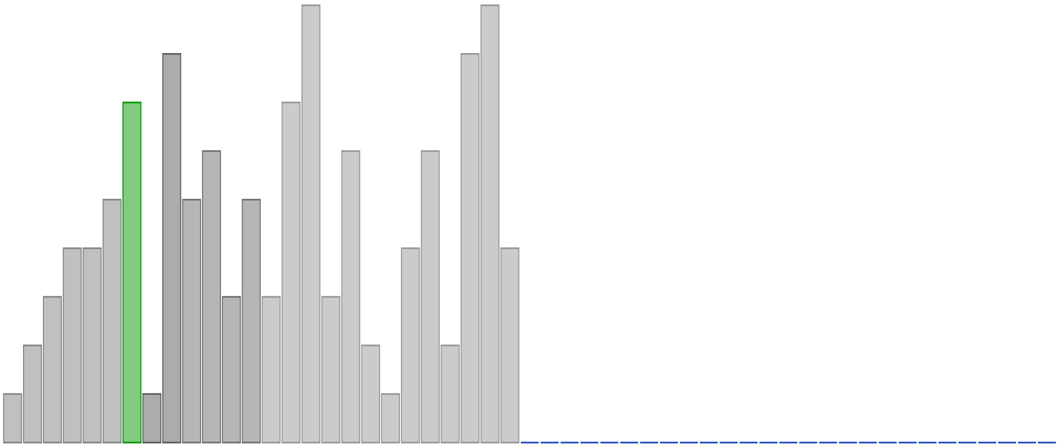


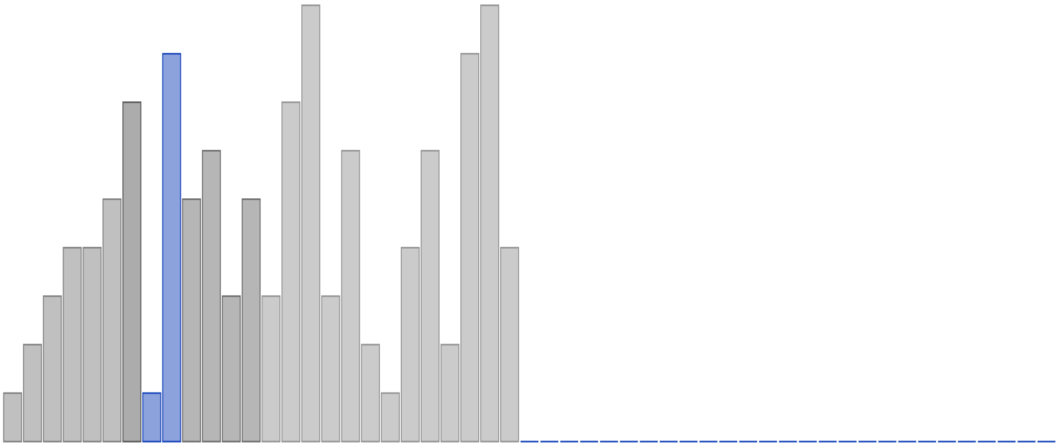


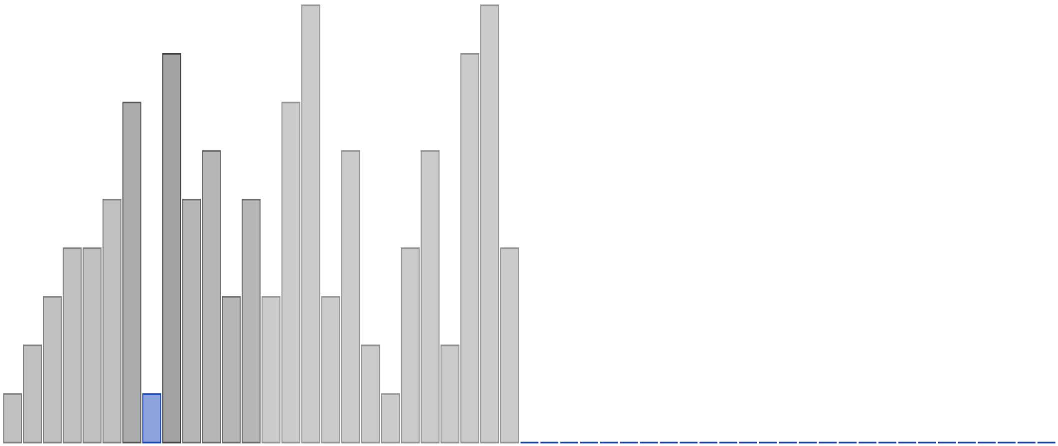


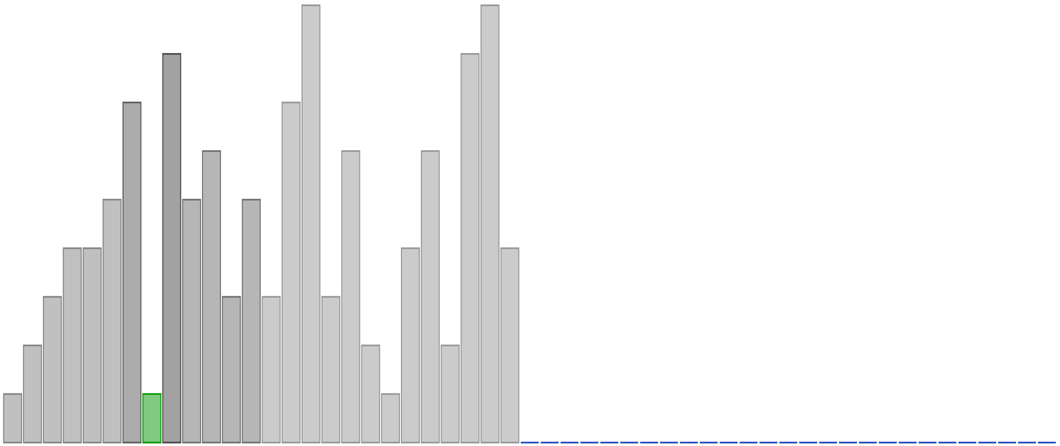


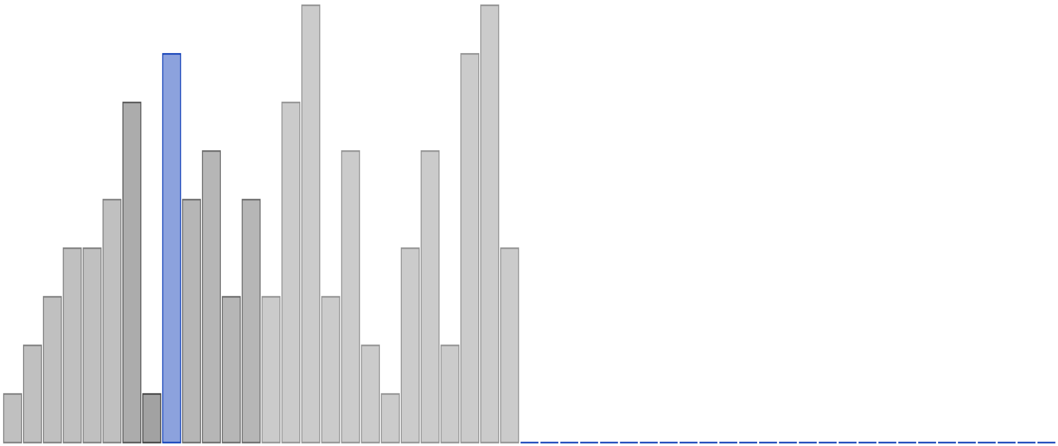


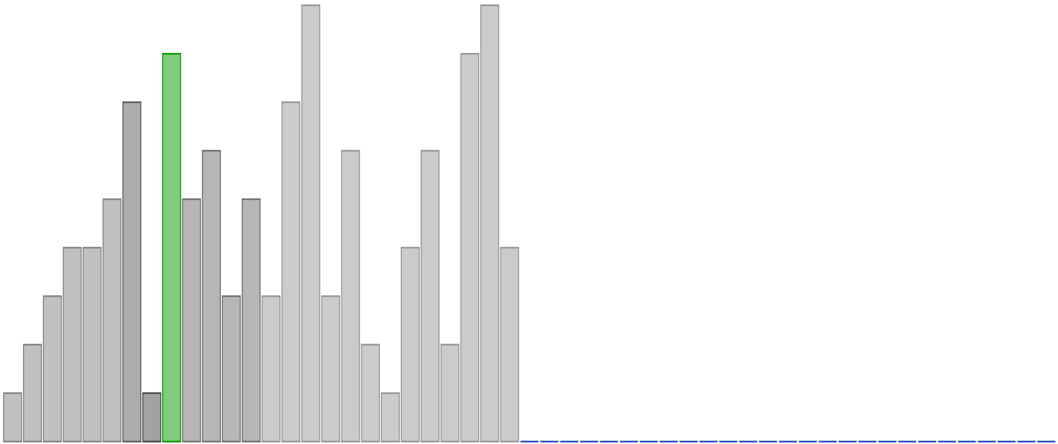


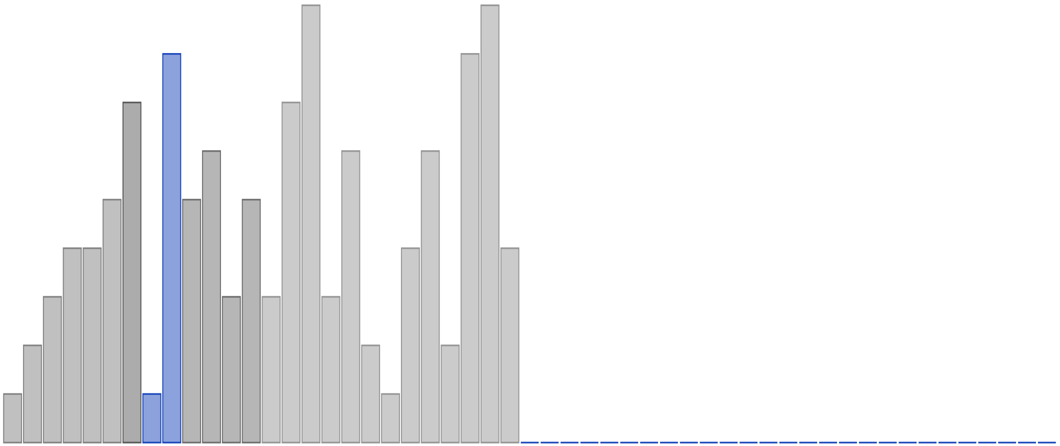


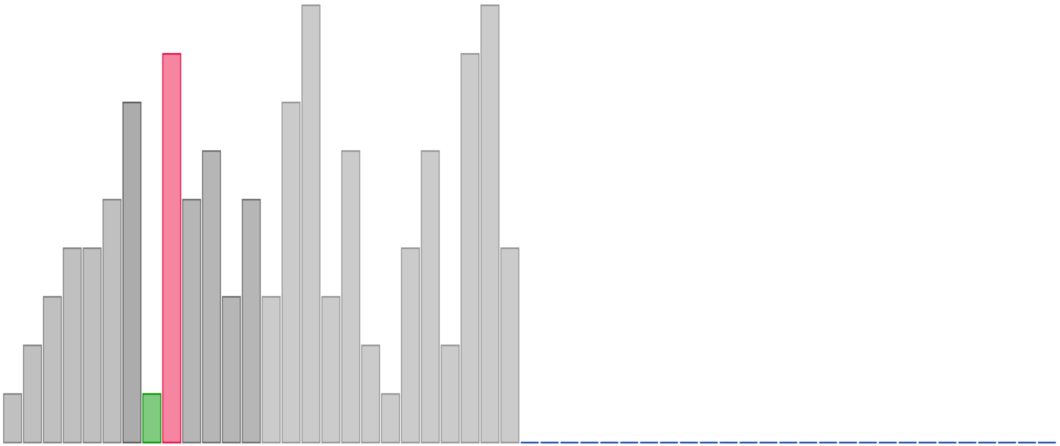


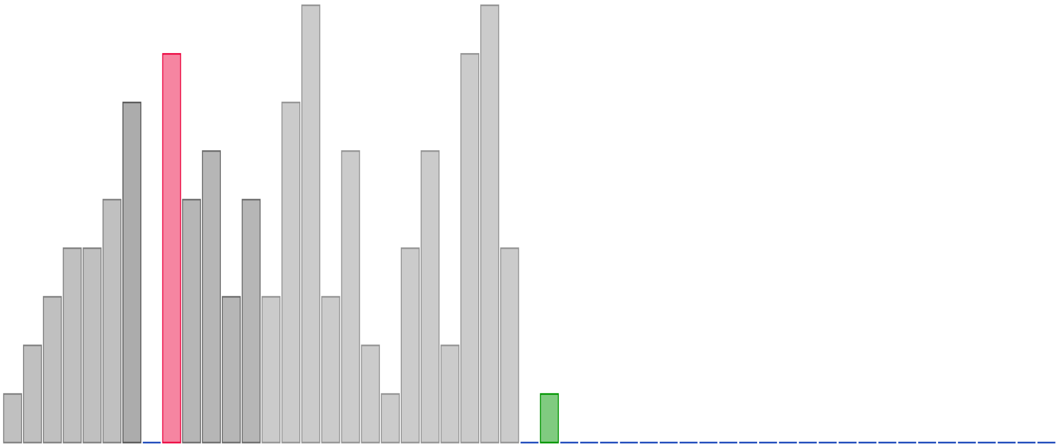


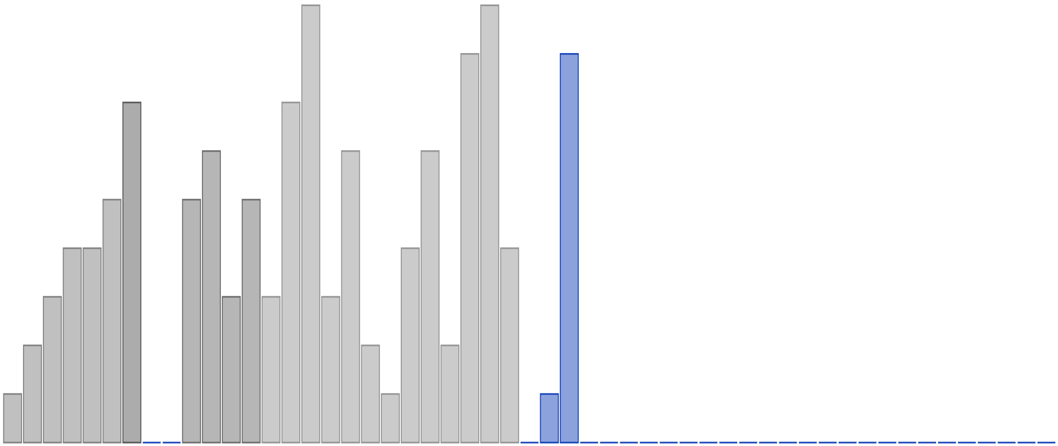


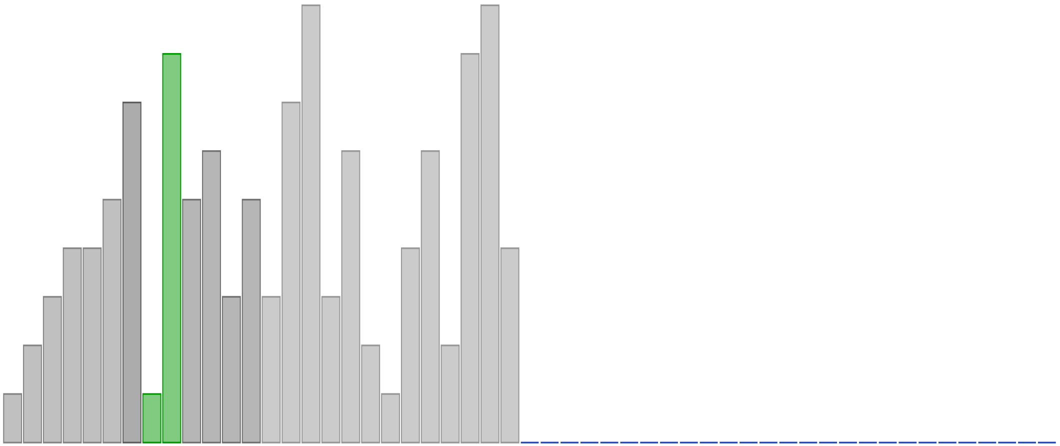


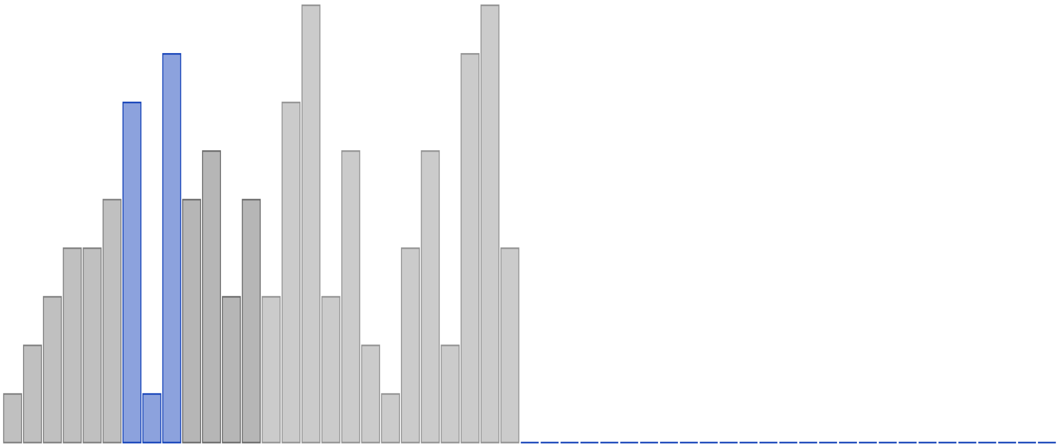


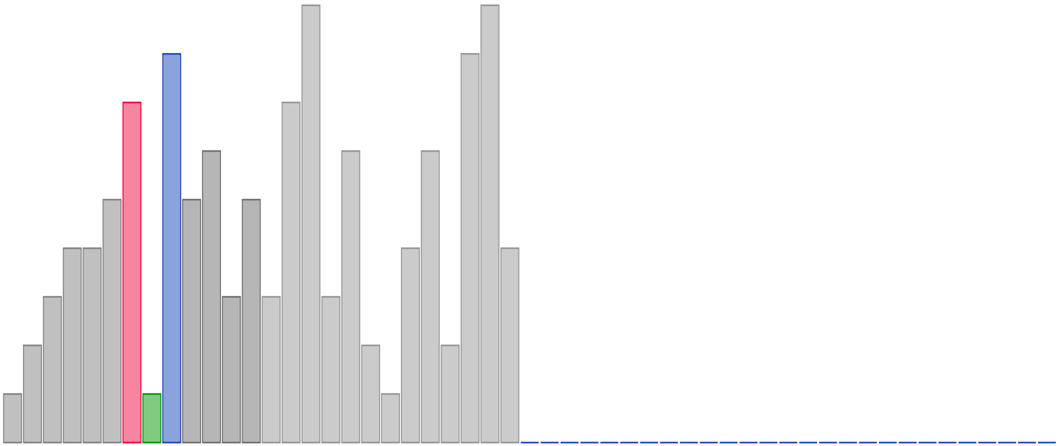


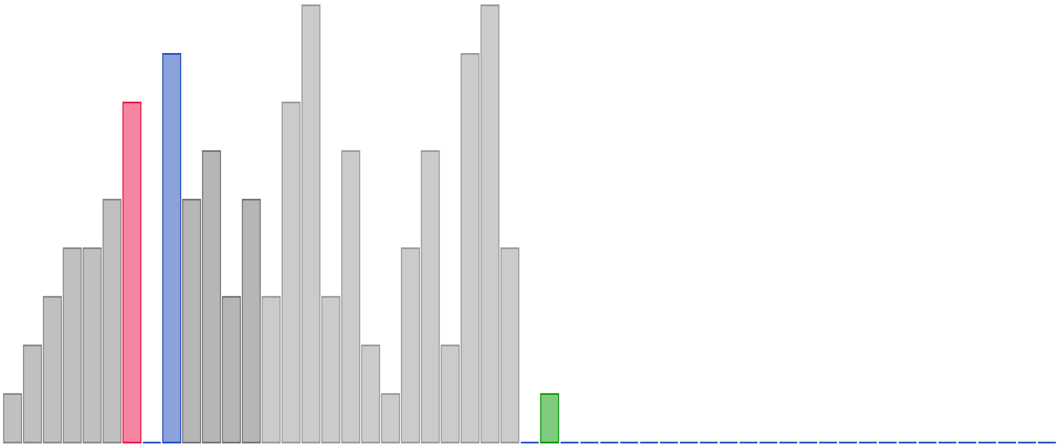


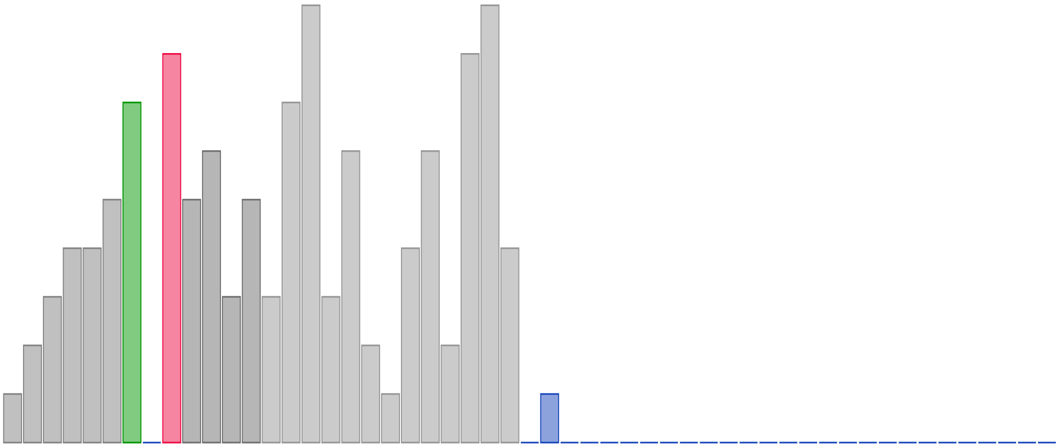


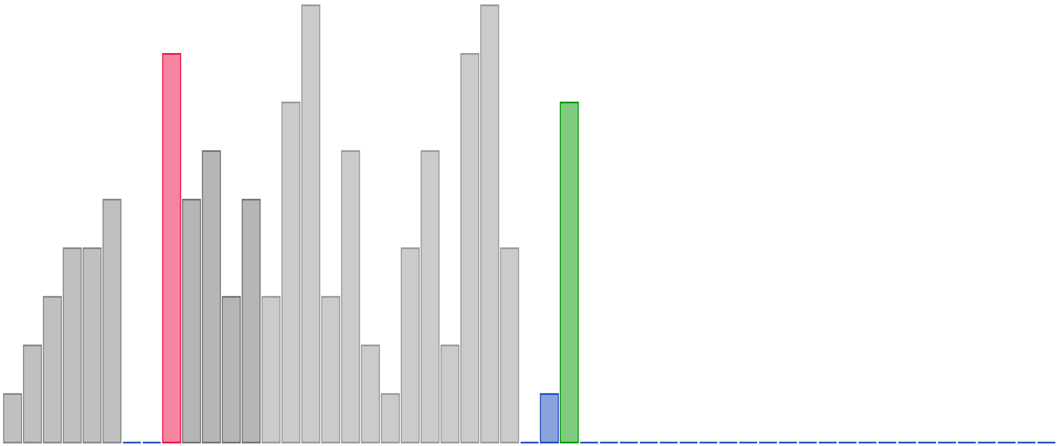


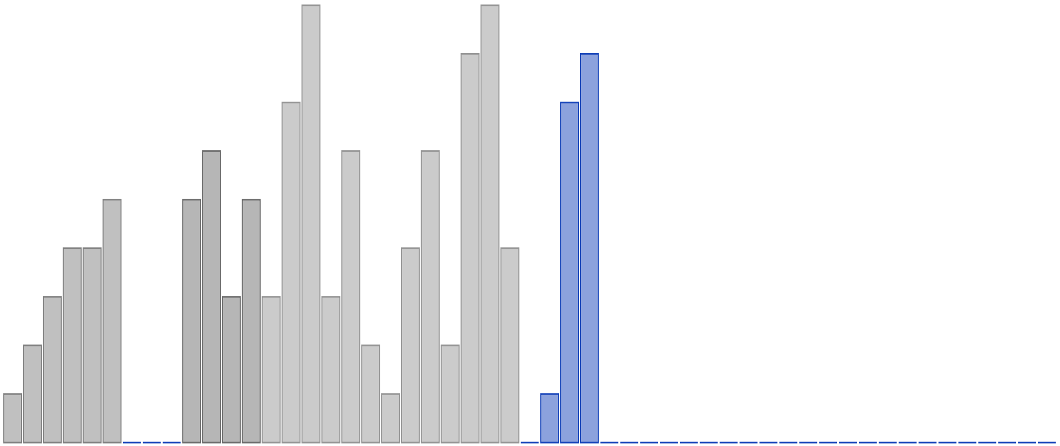


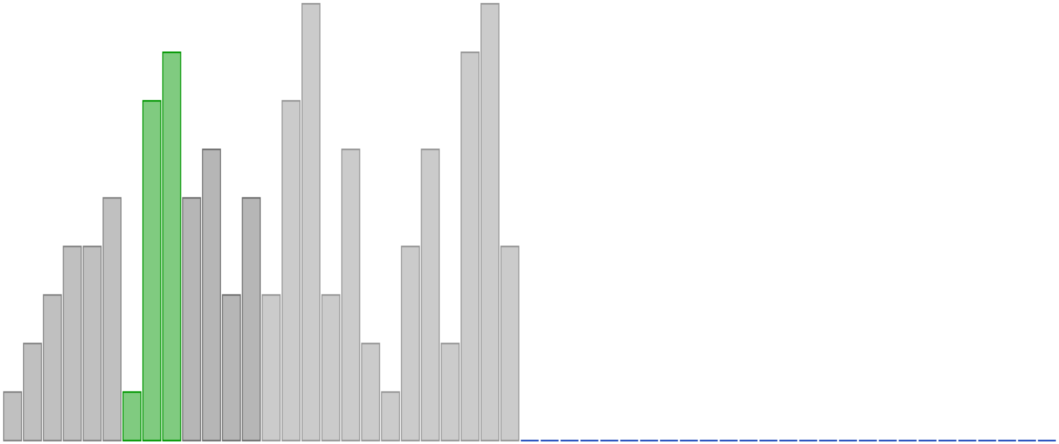


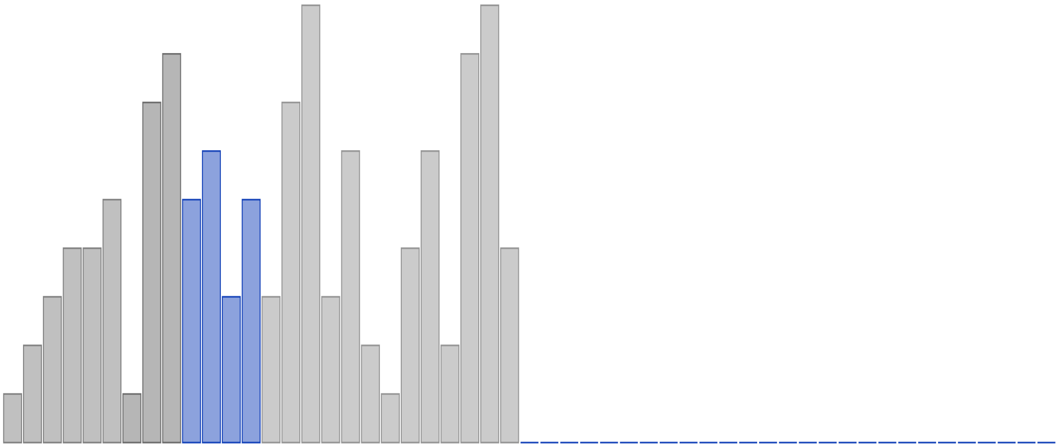


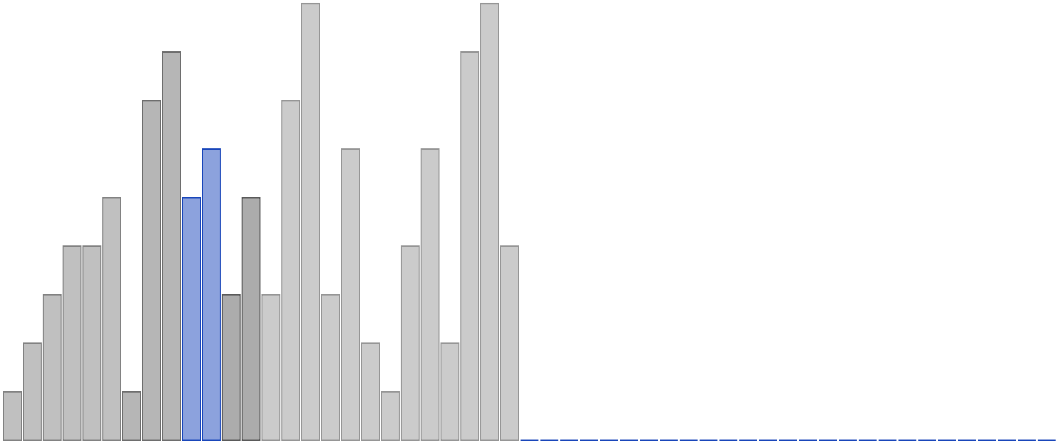


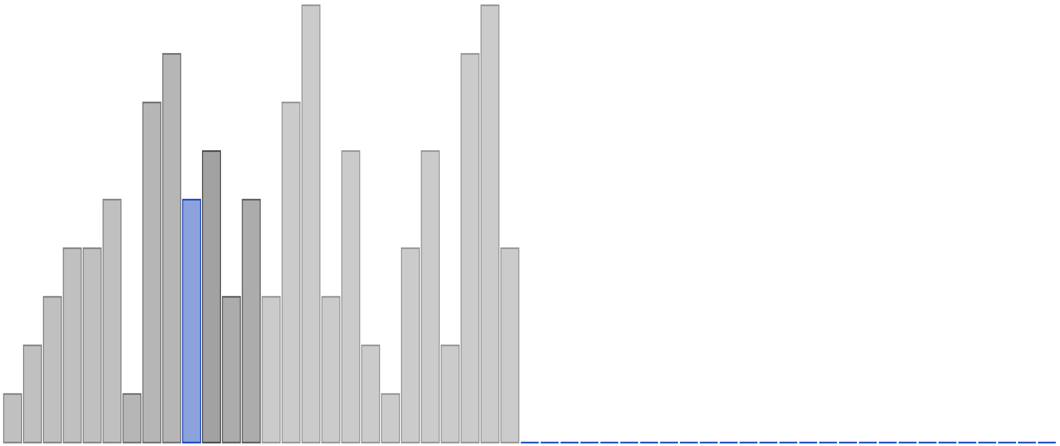


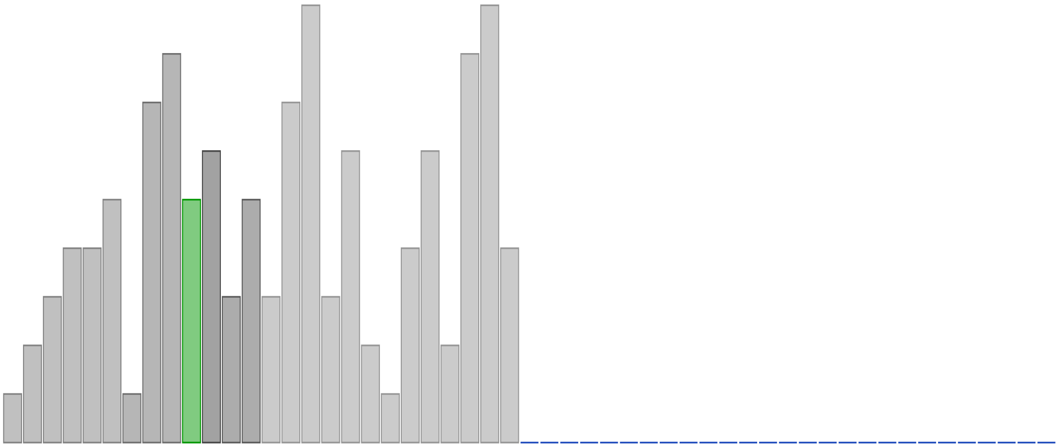


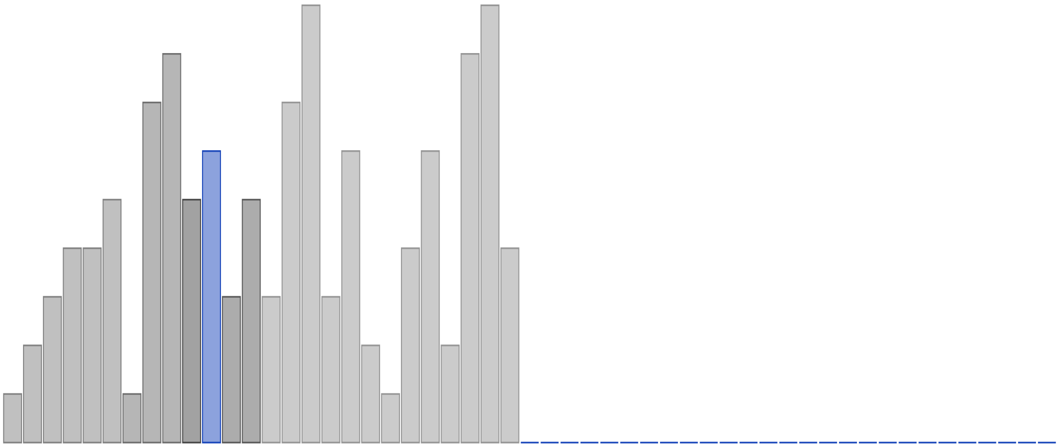


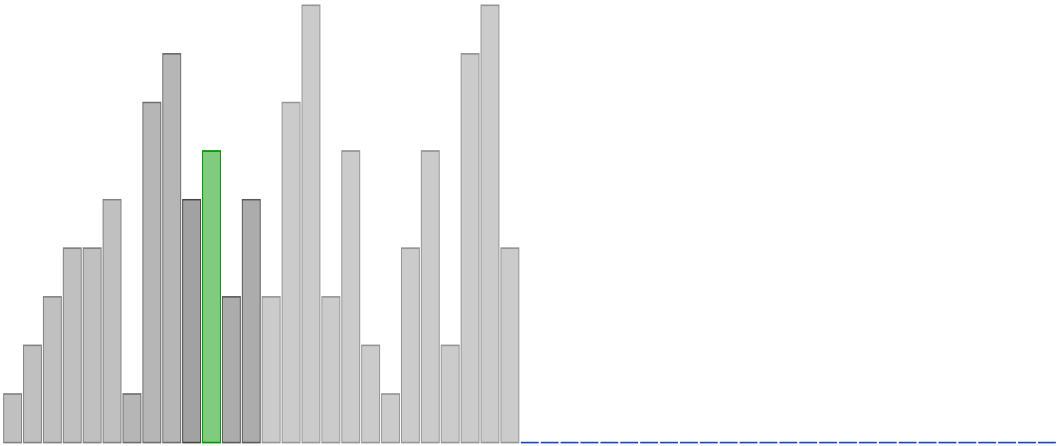


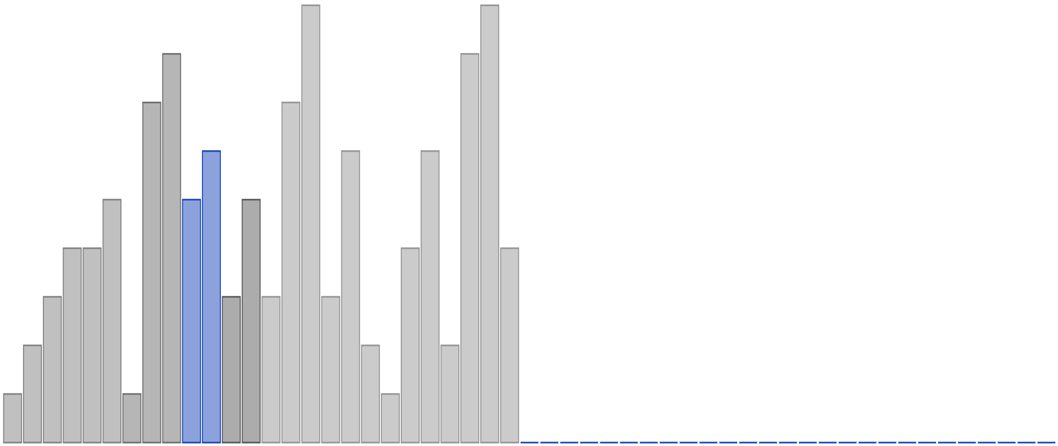


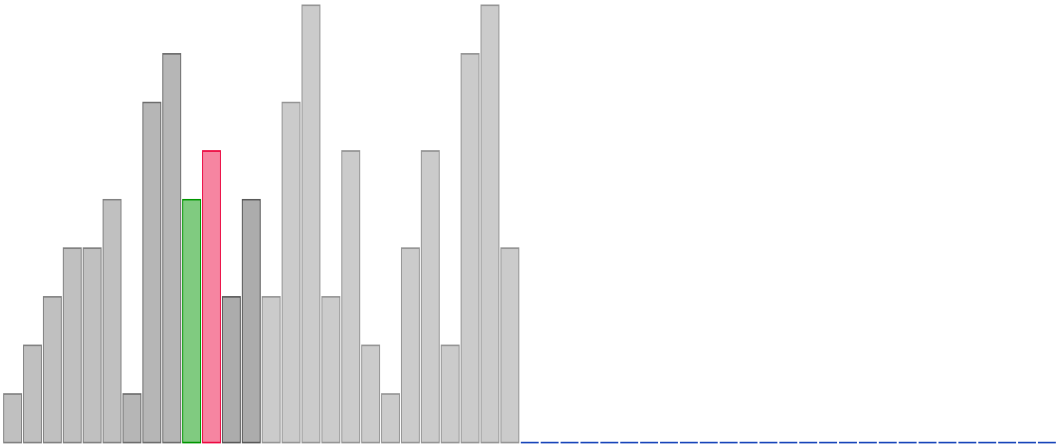


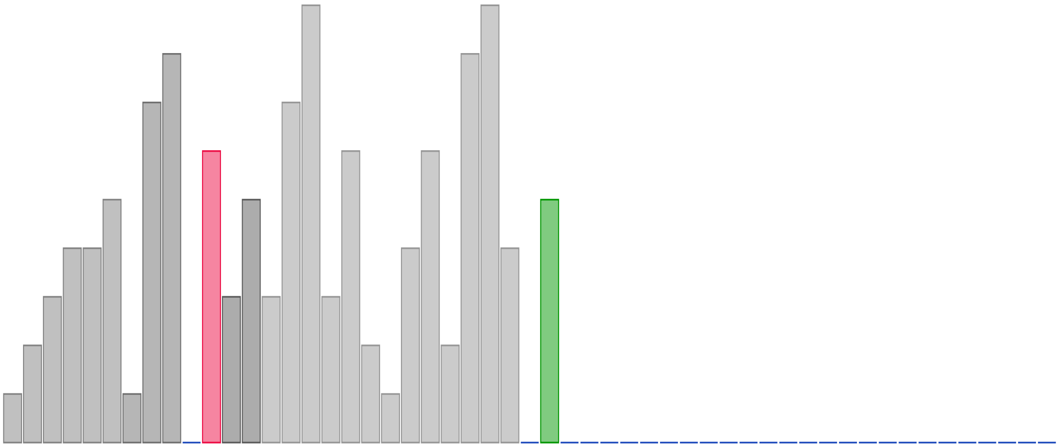


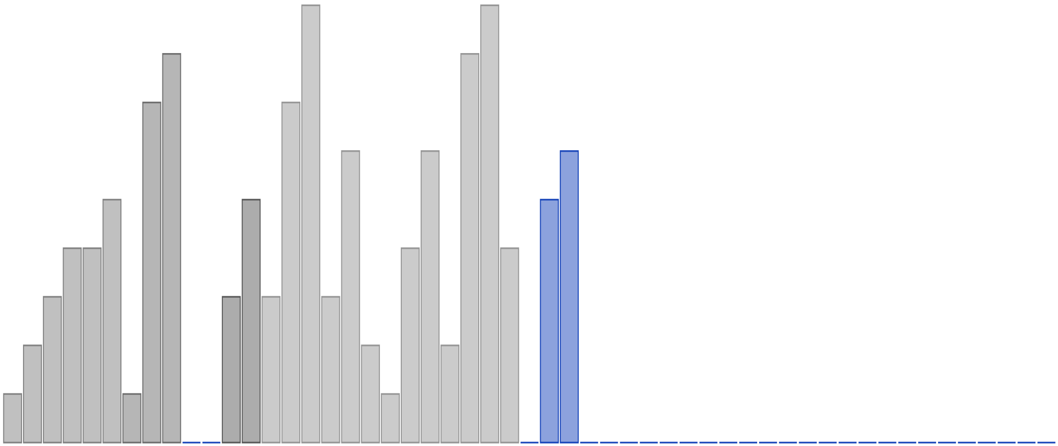


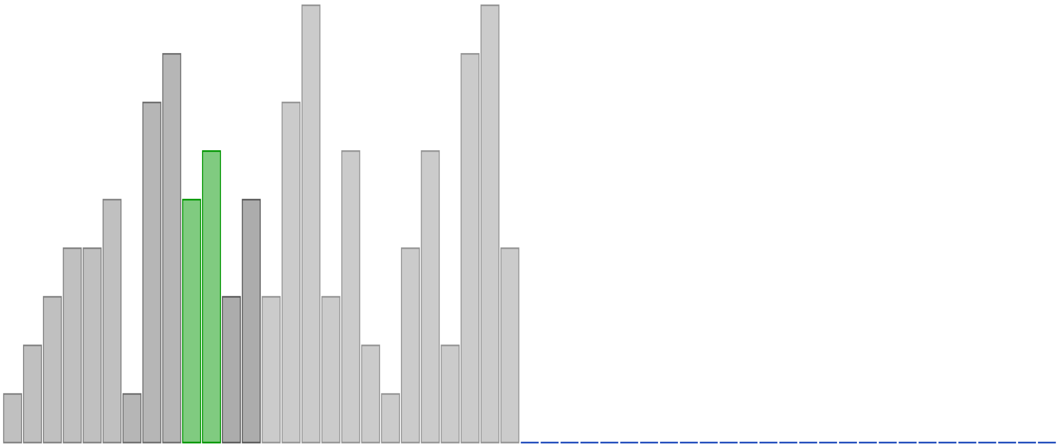


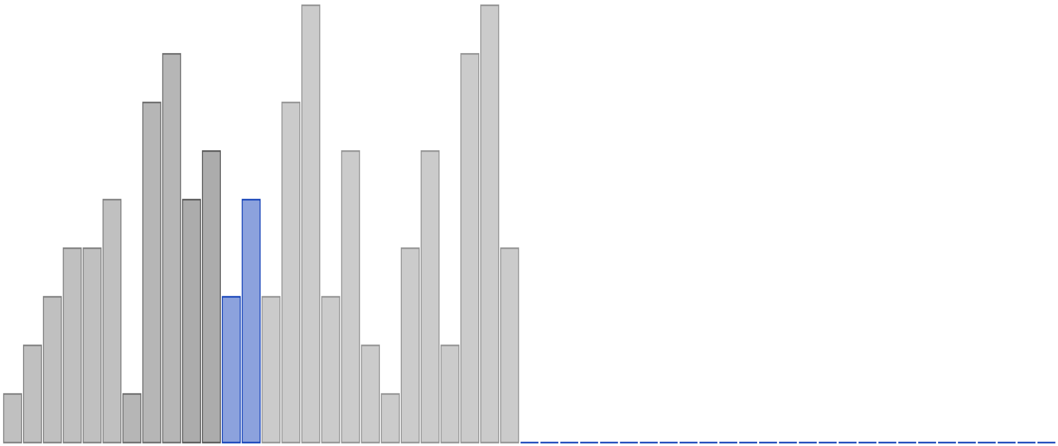


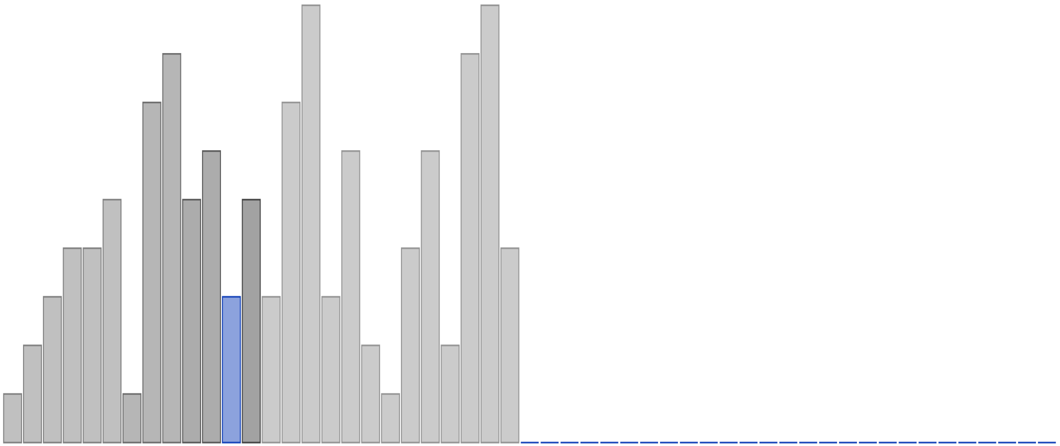


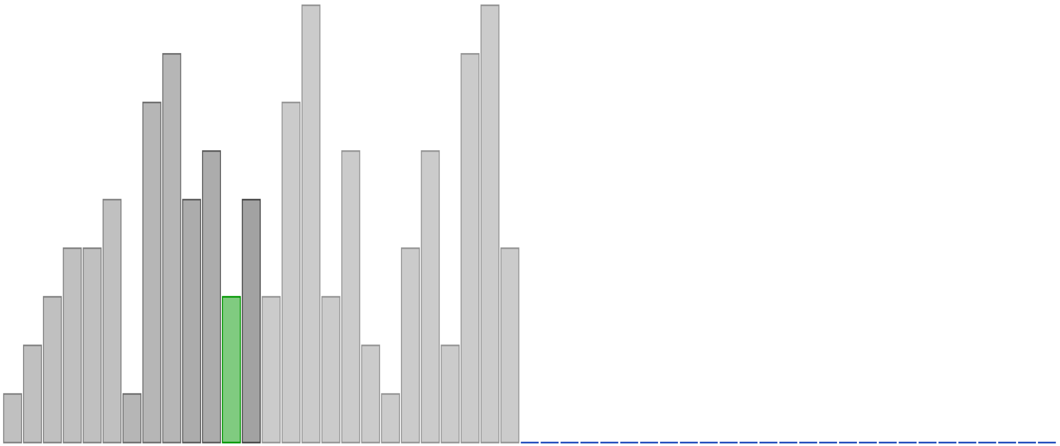


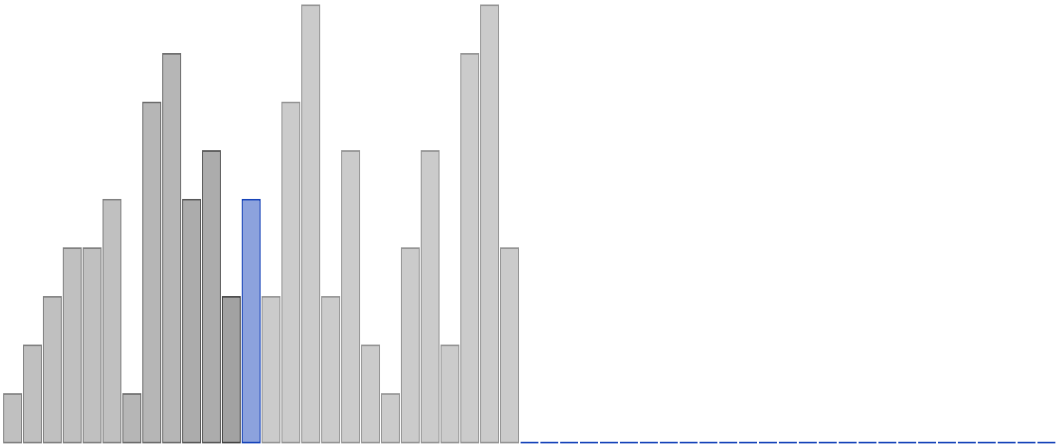


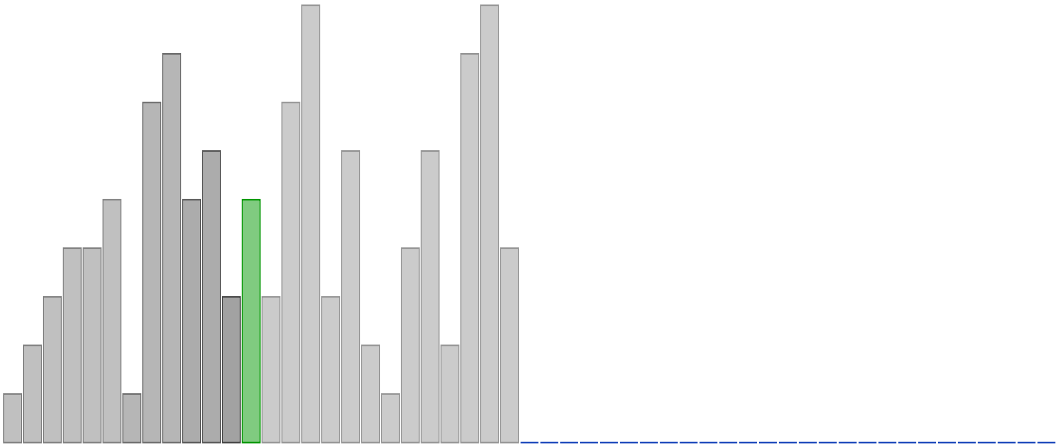


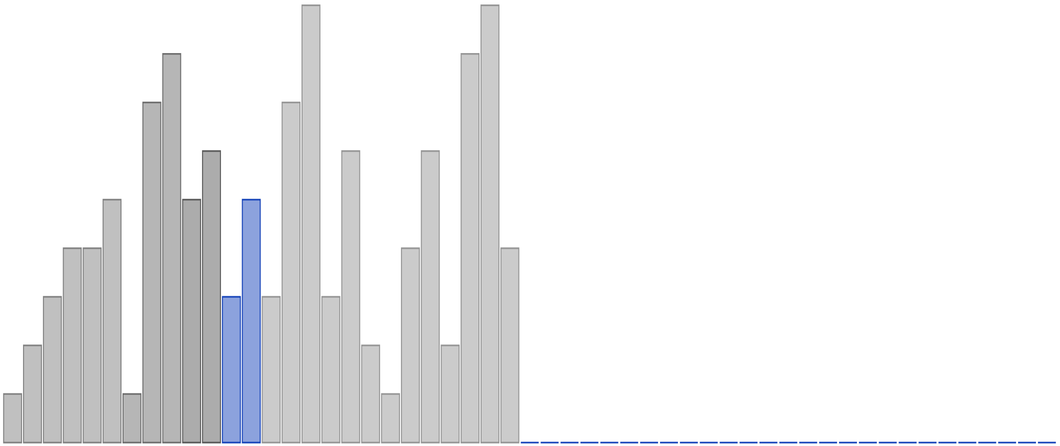


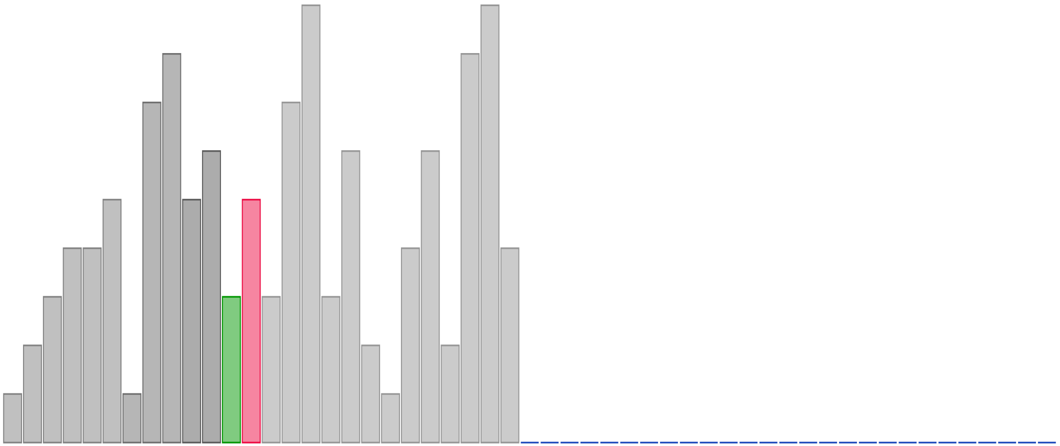


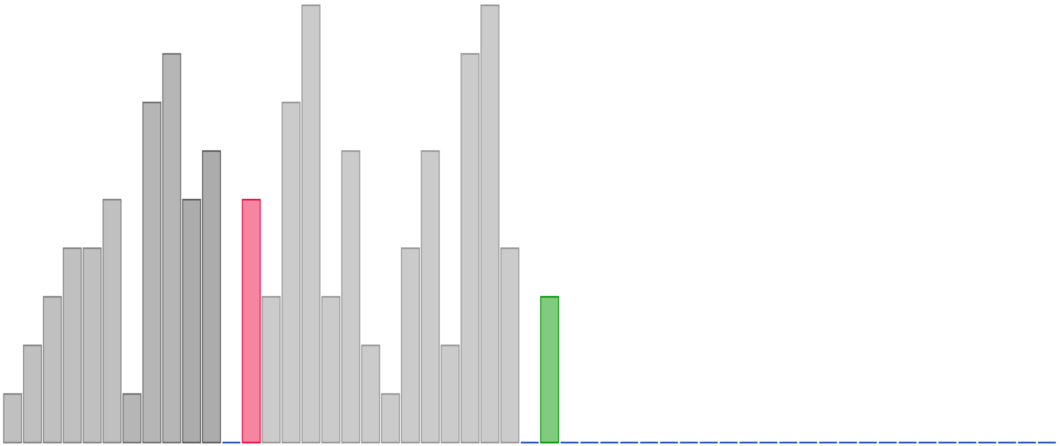


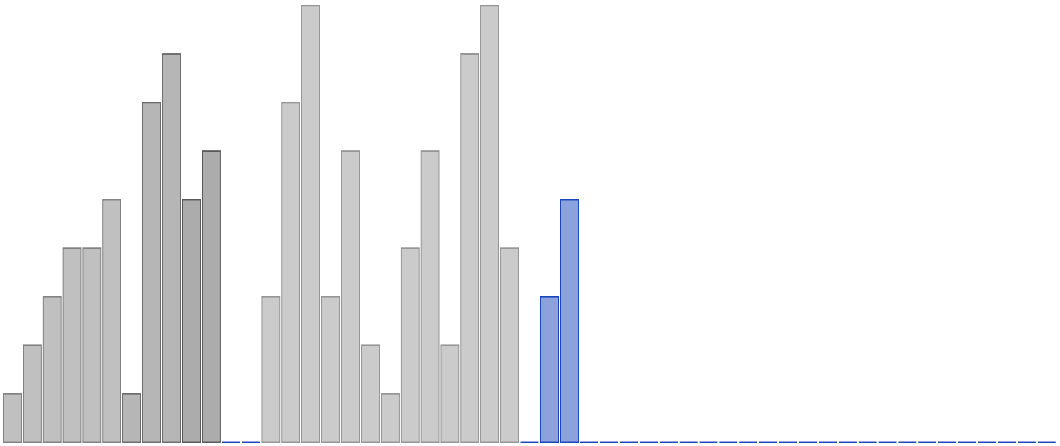


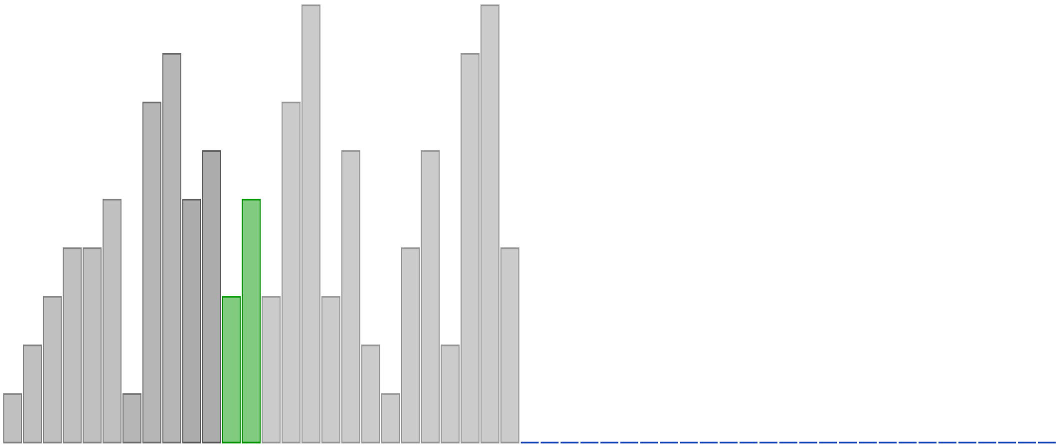


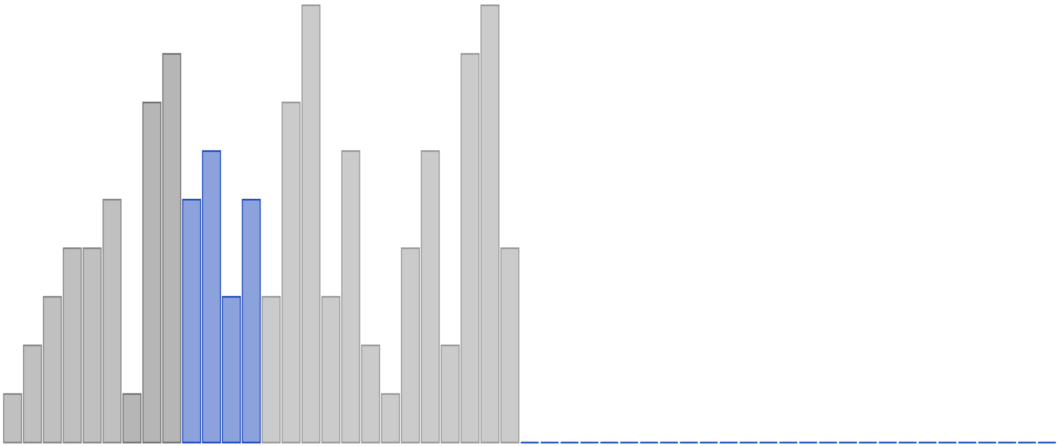


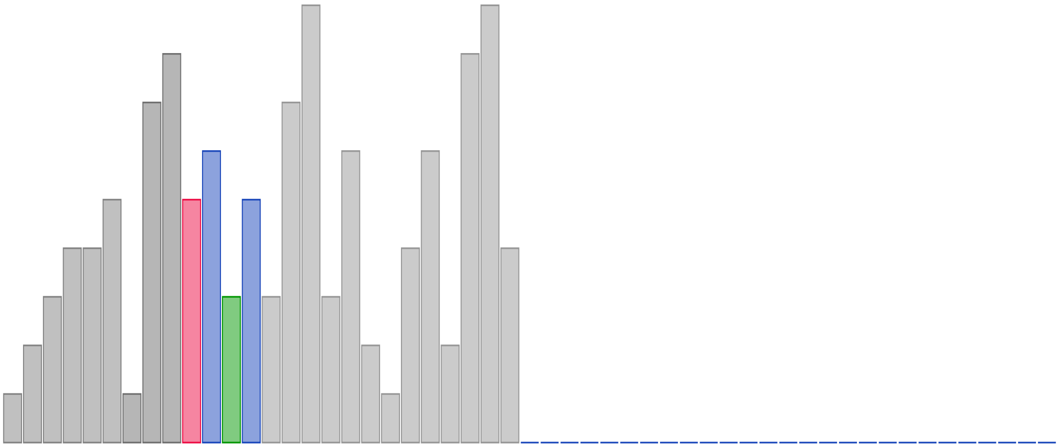


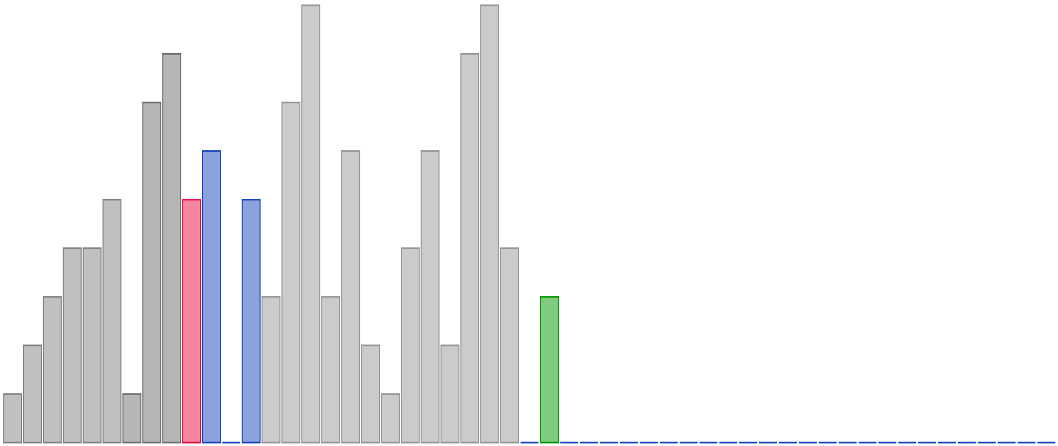


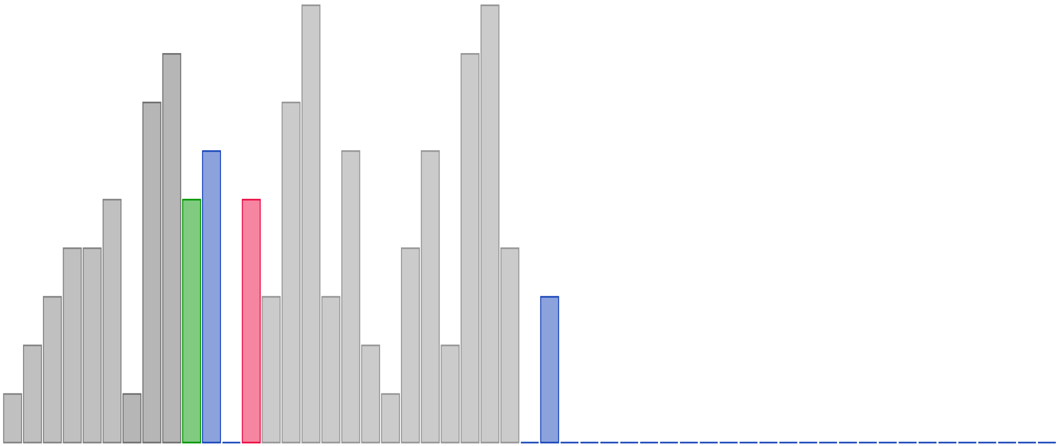


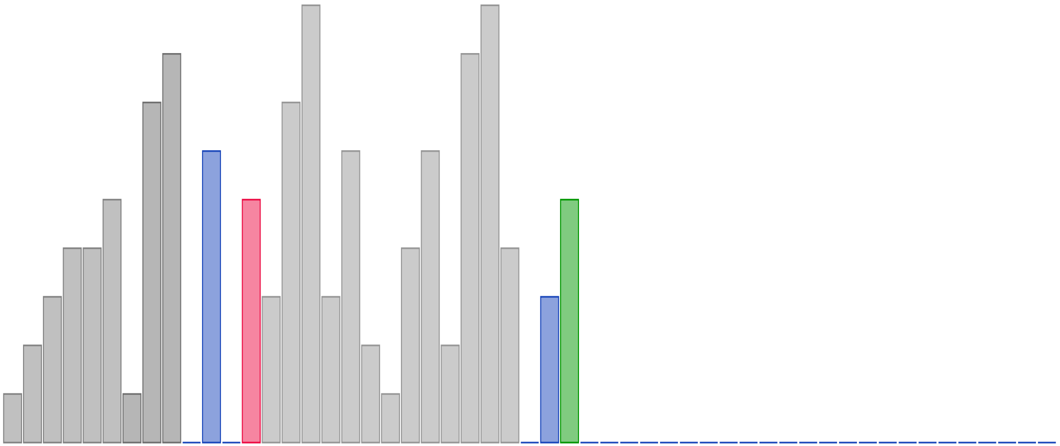


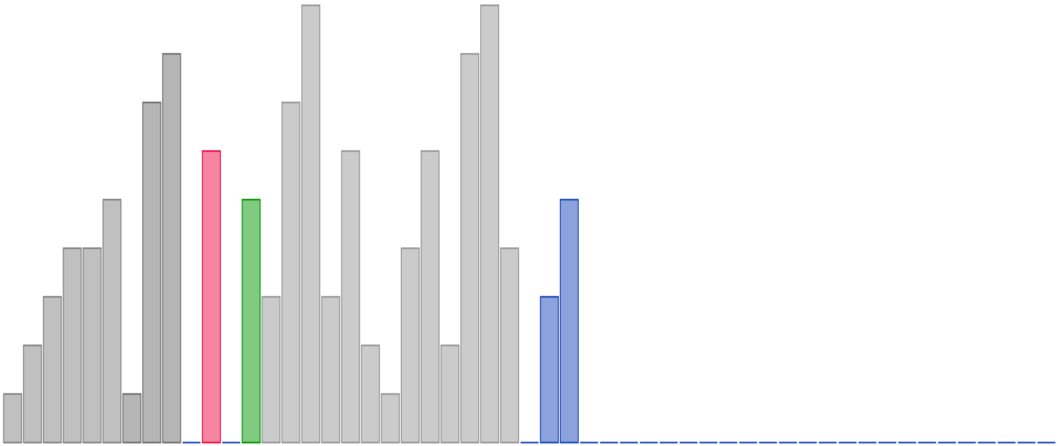


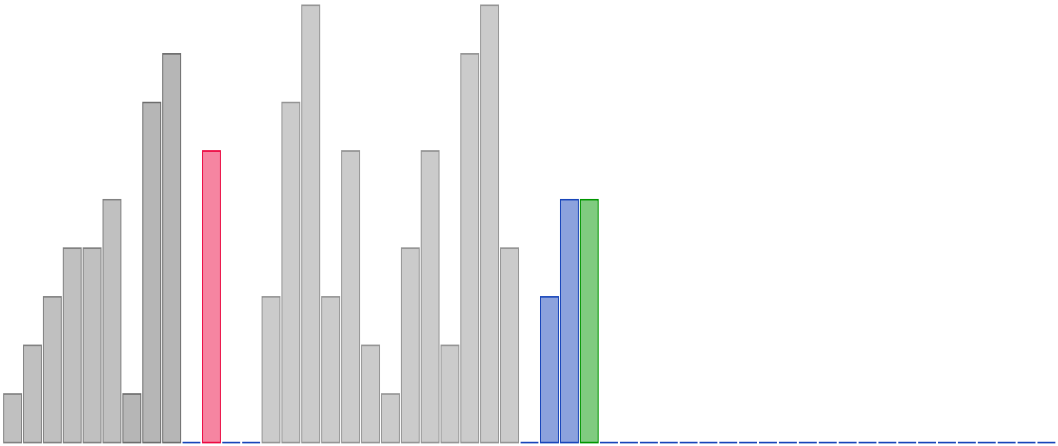


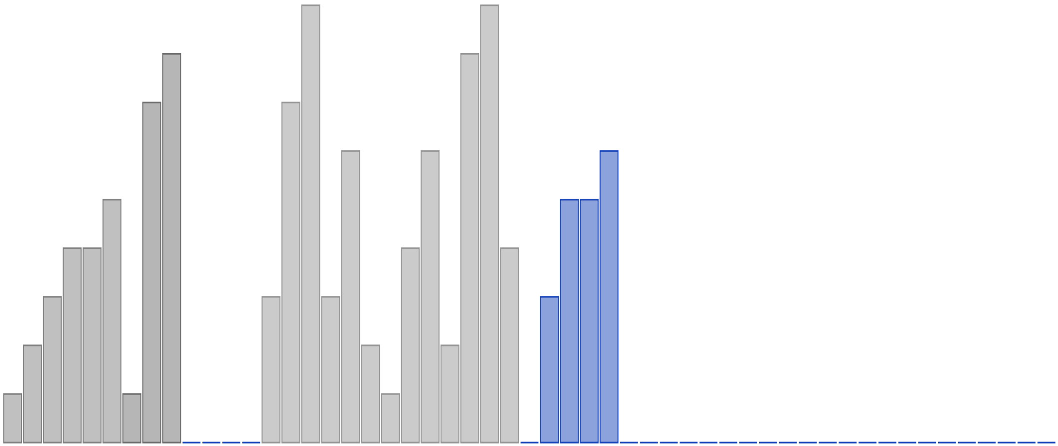


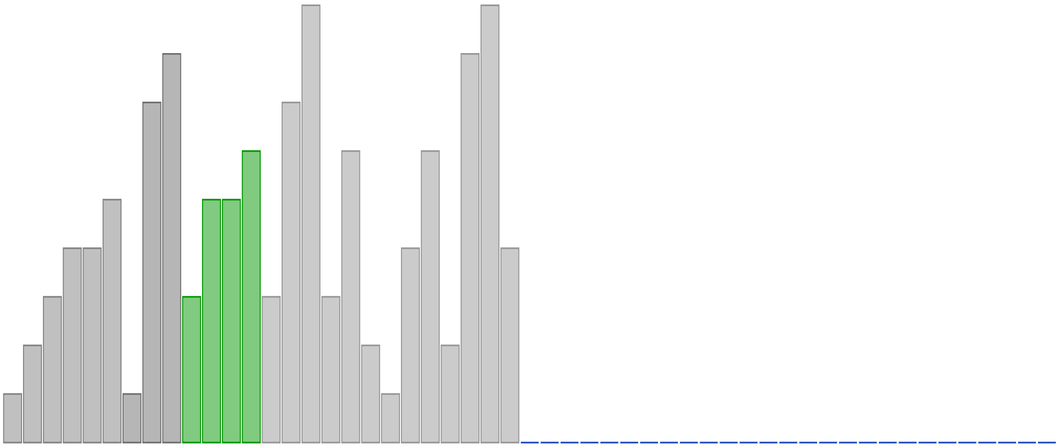


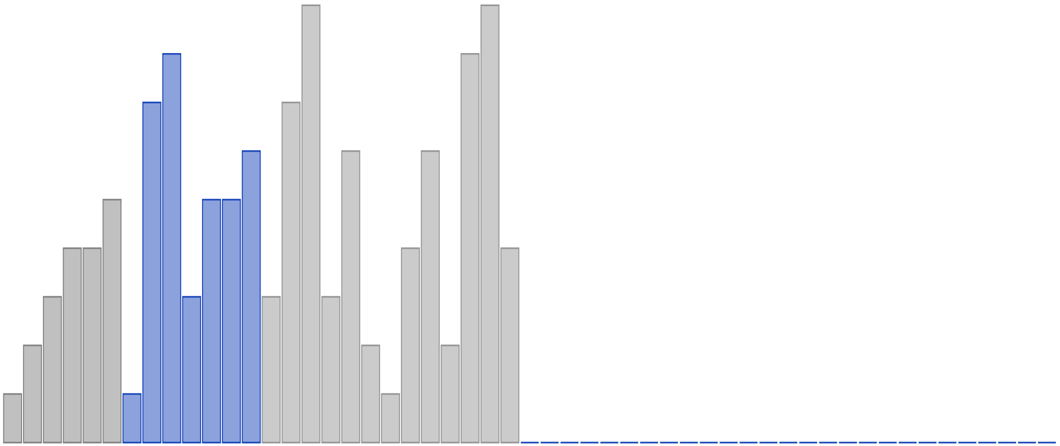


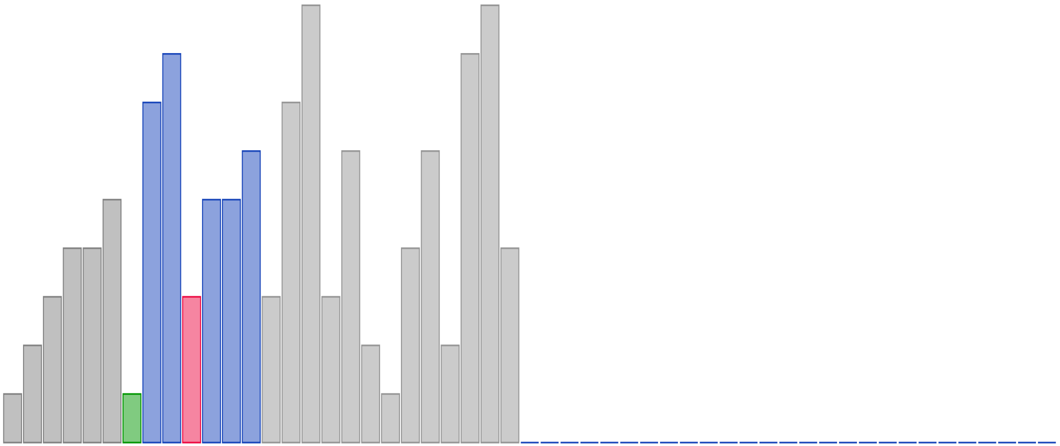


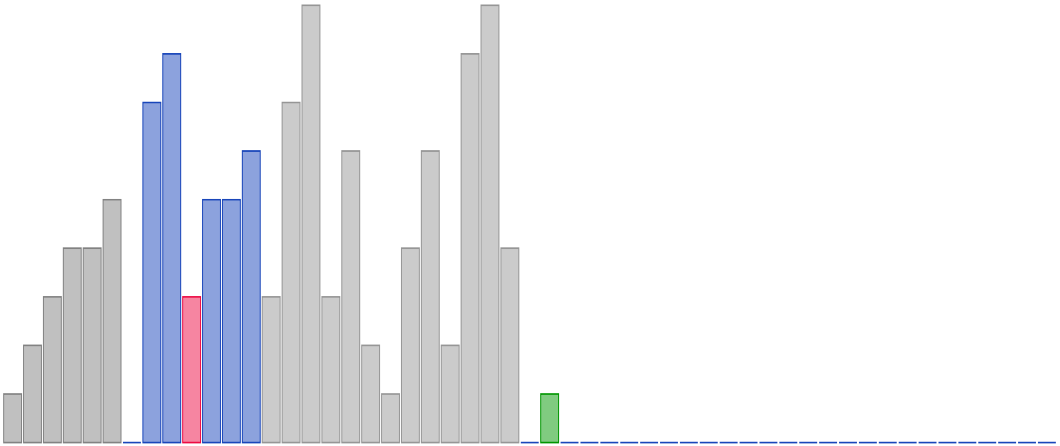


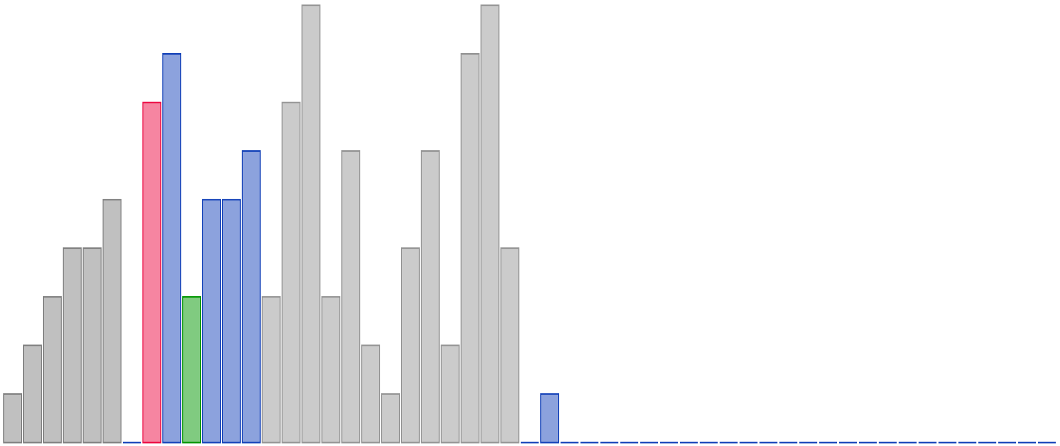


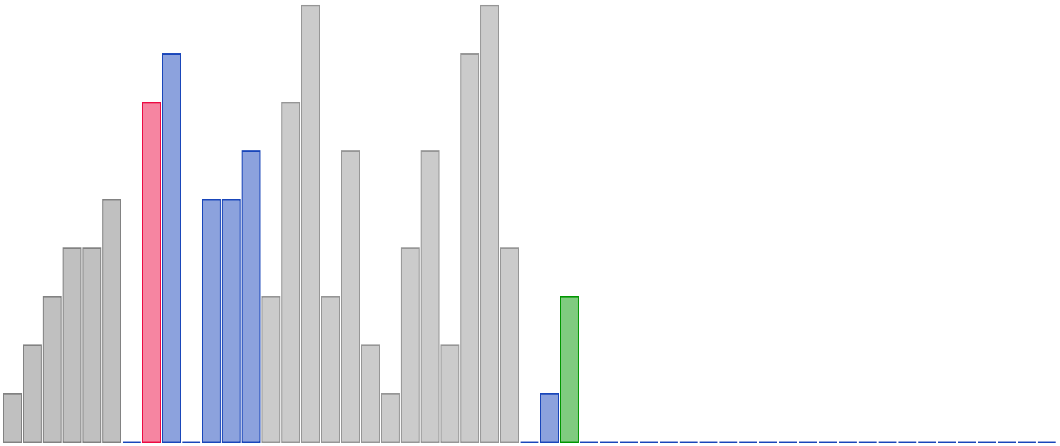


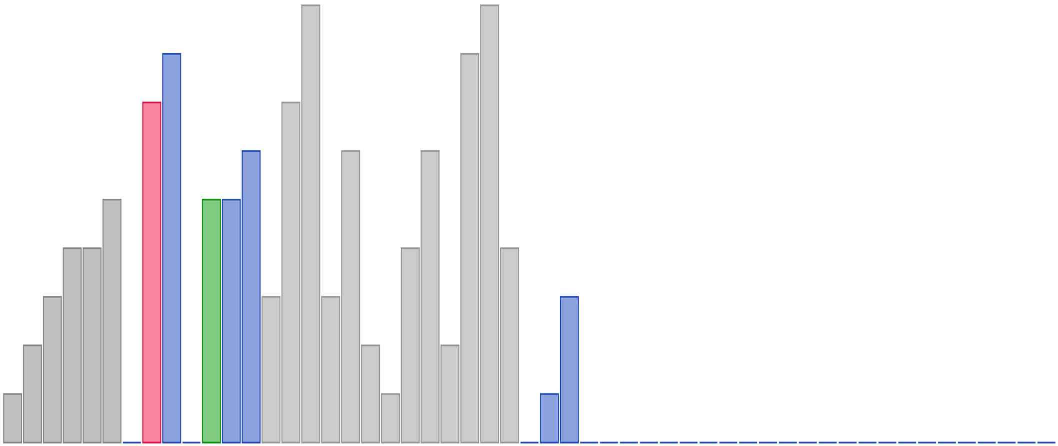




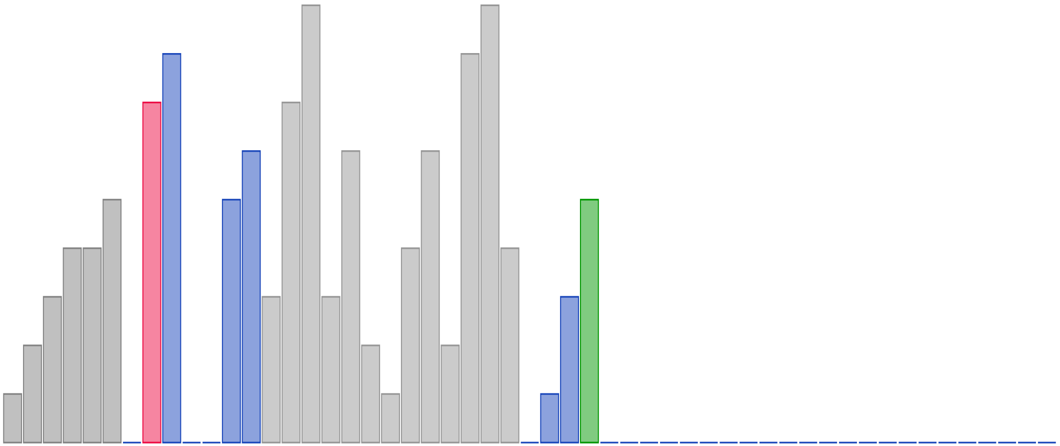


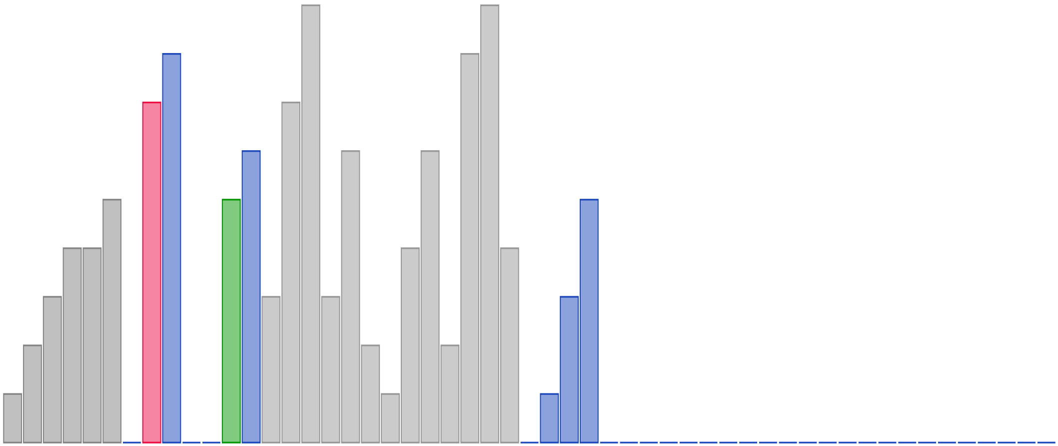


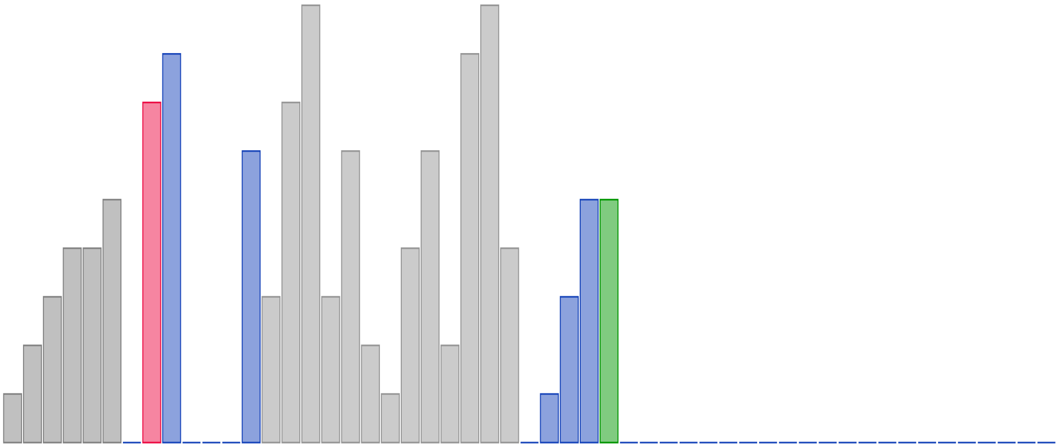


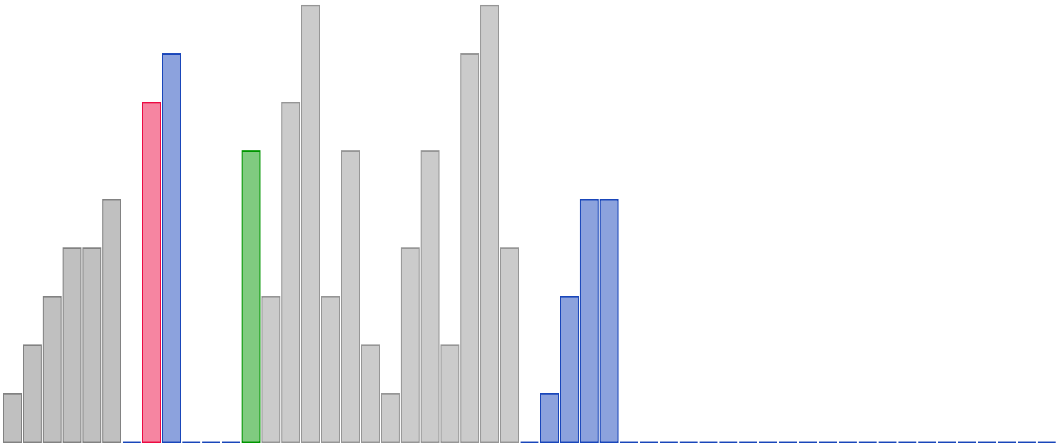


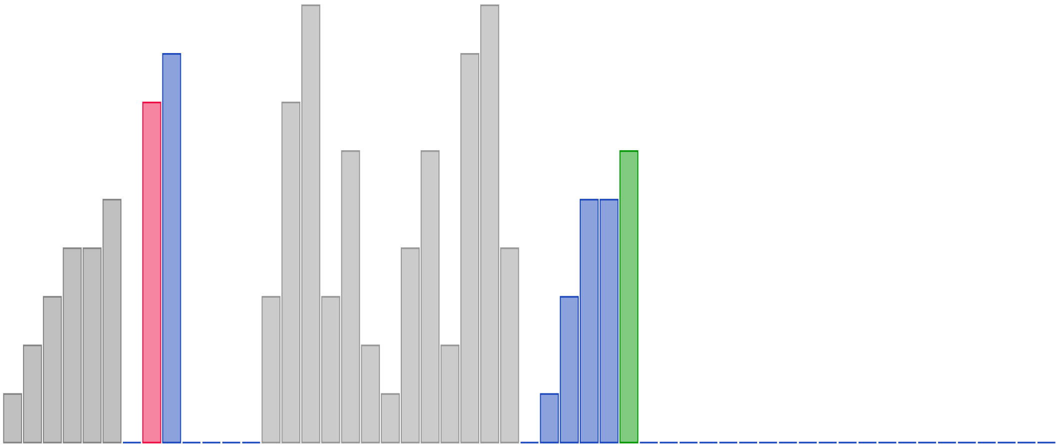
Mergesort: Example

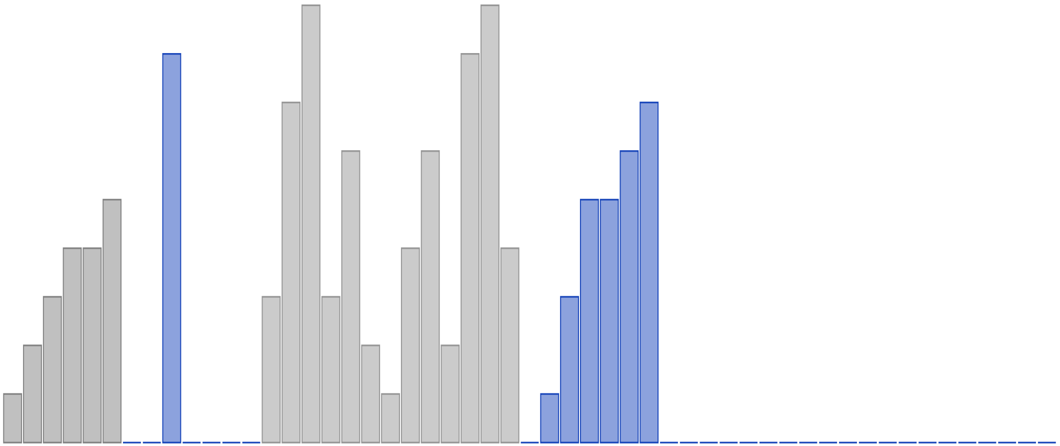




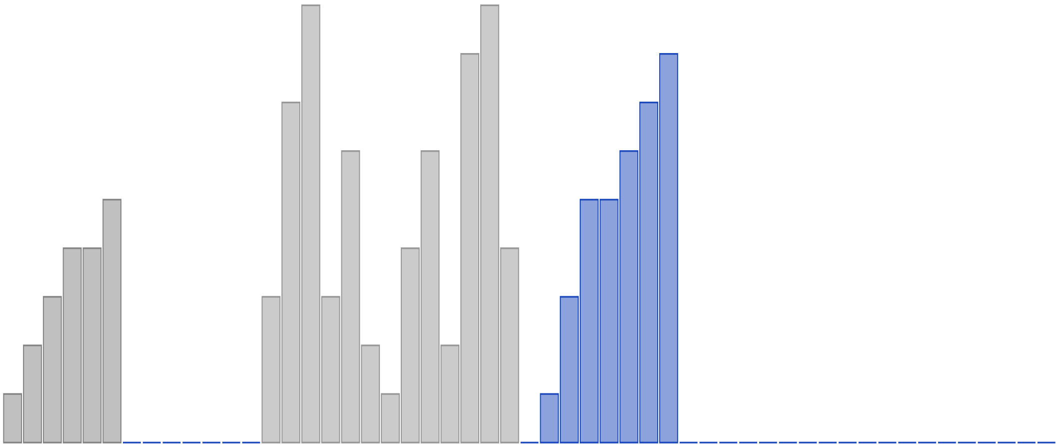


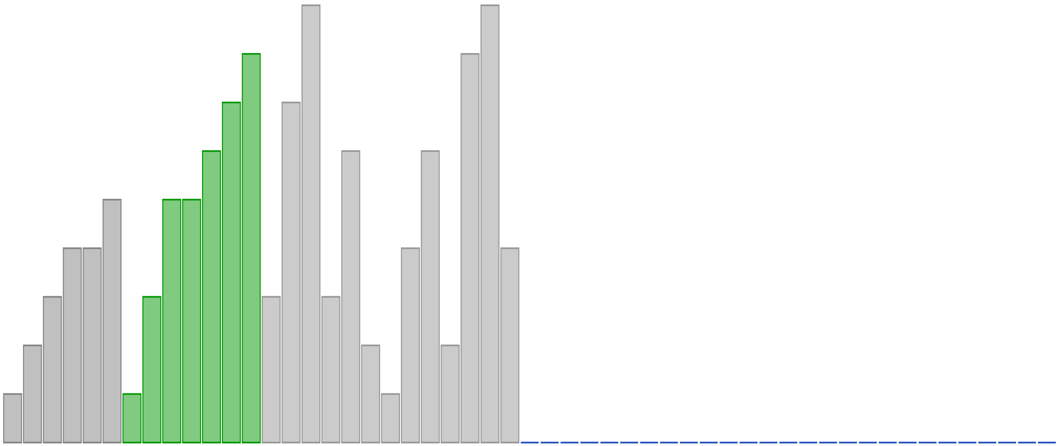


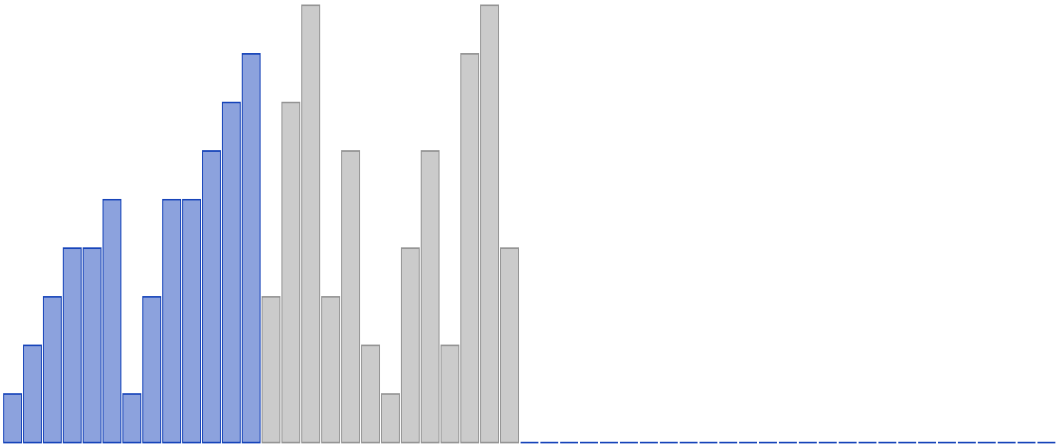


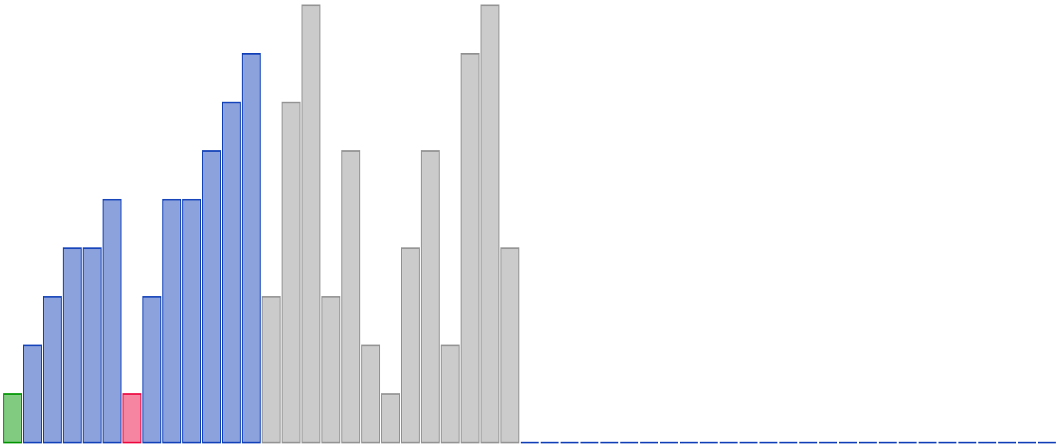


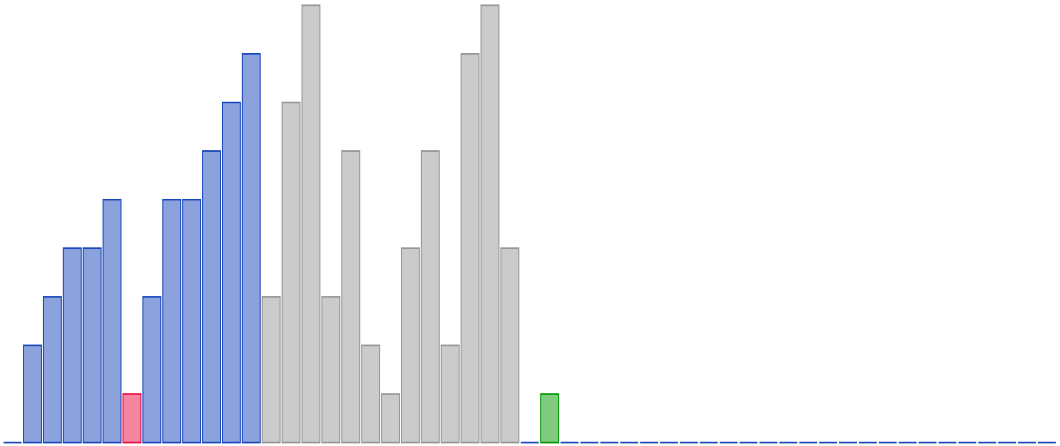
Mergesort: Example

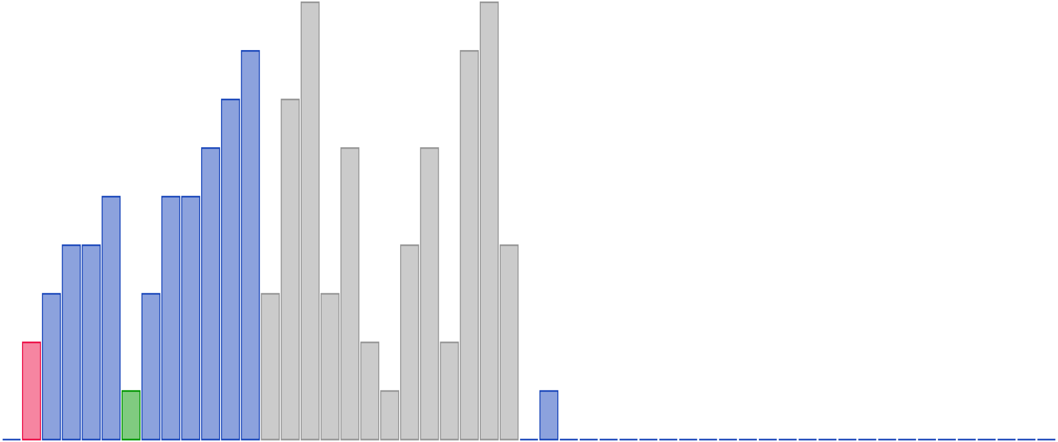


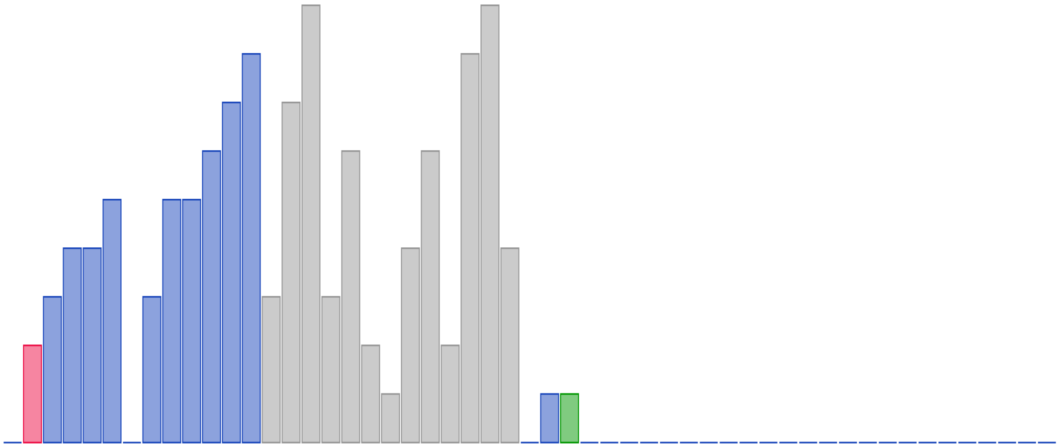


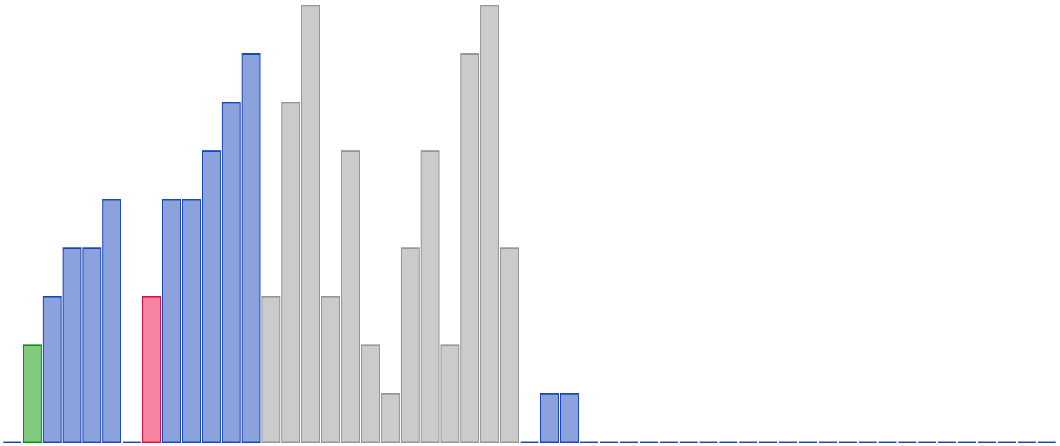


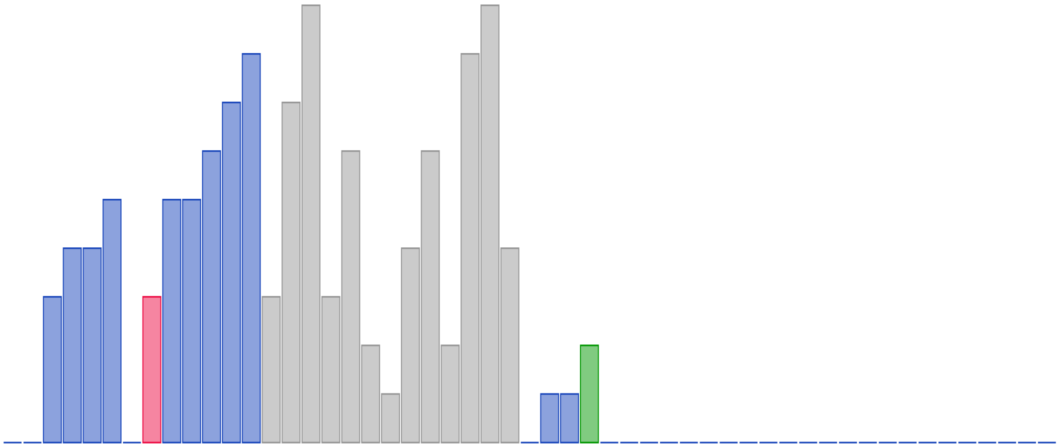


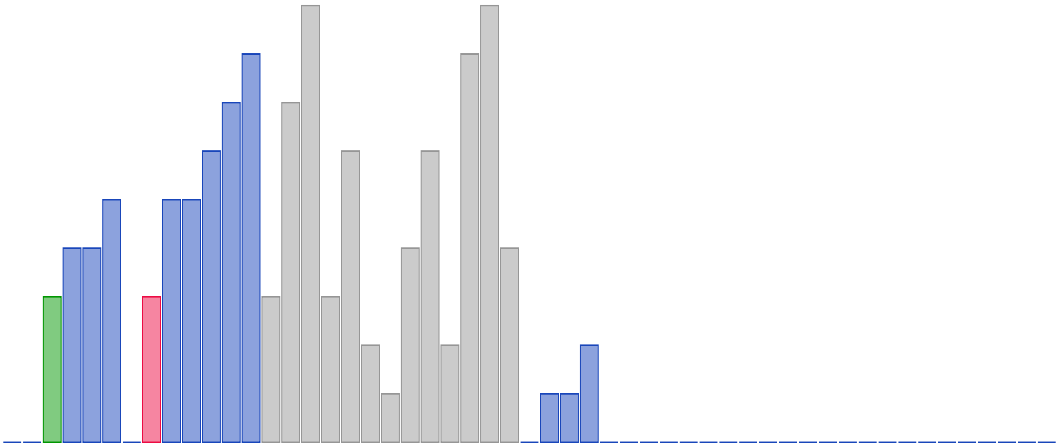


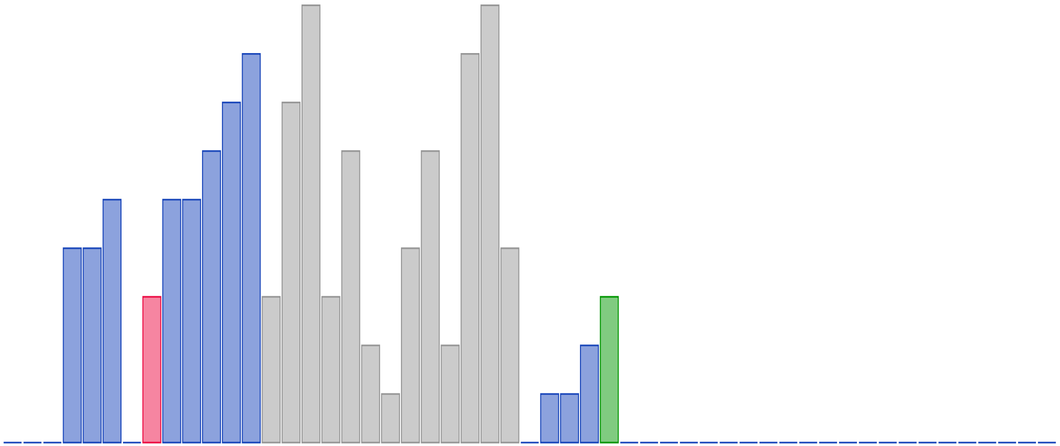


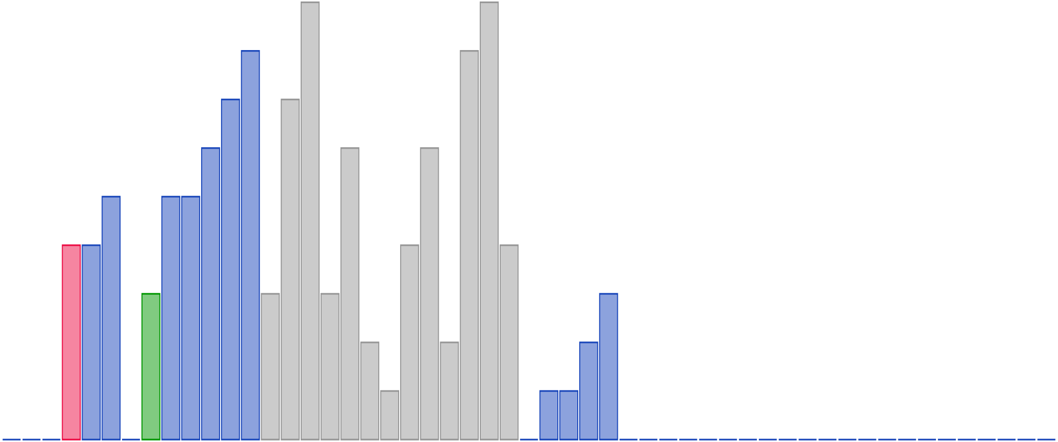


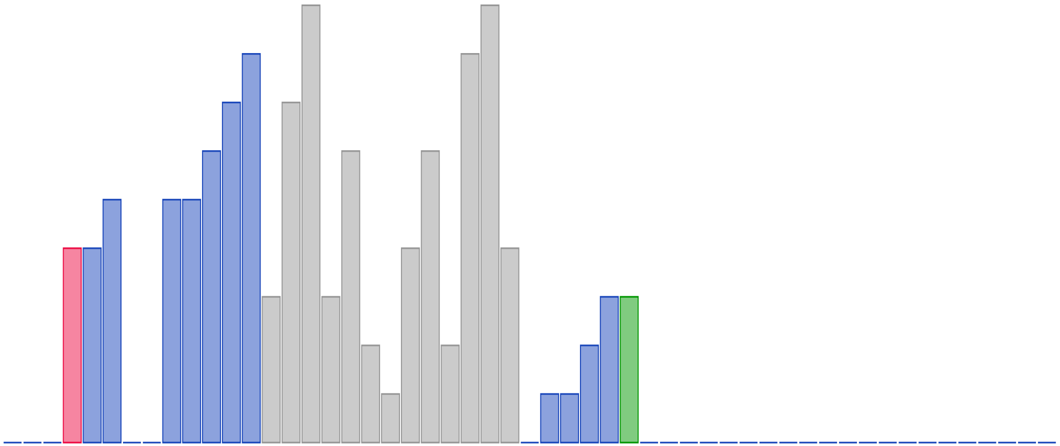


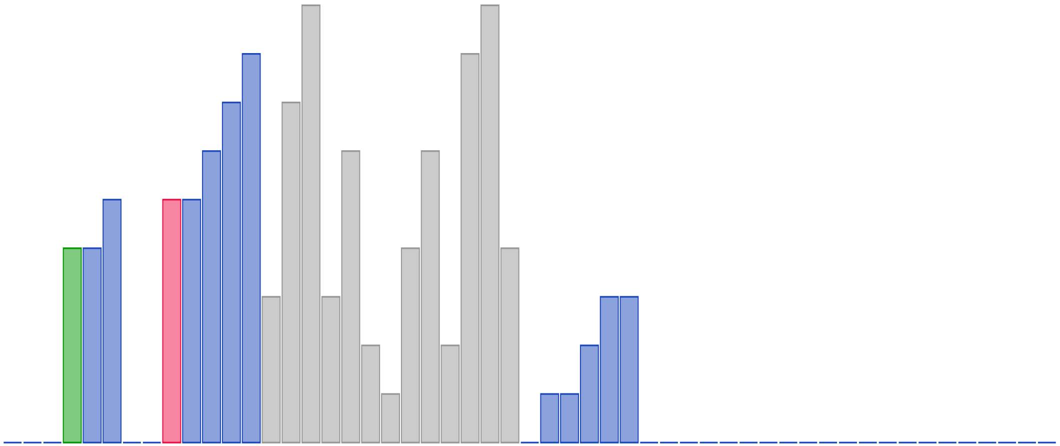


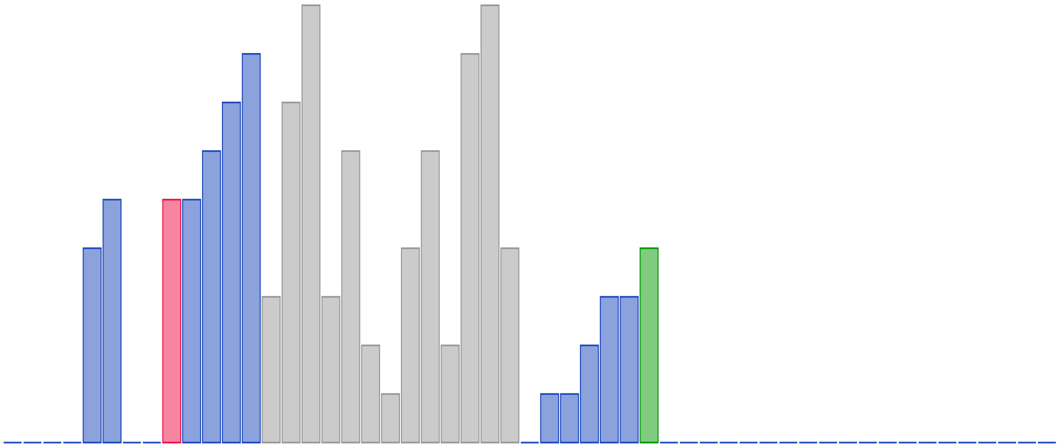


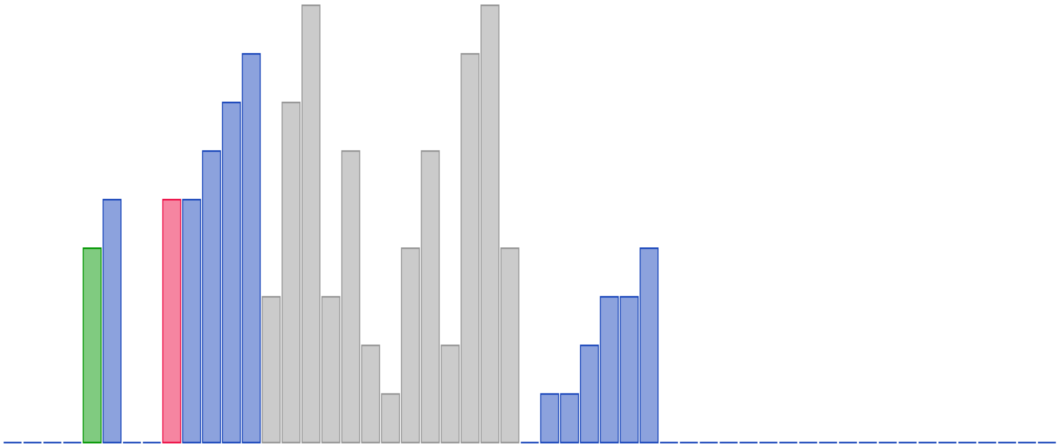


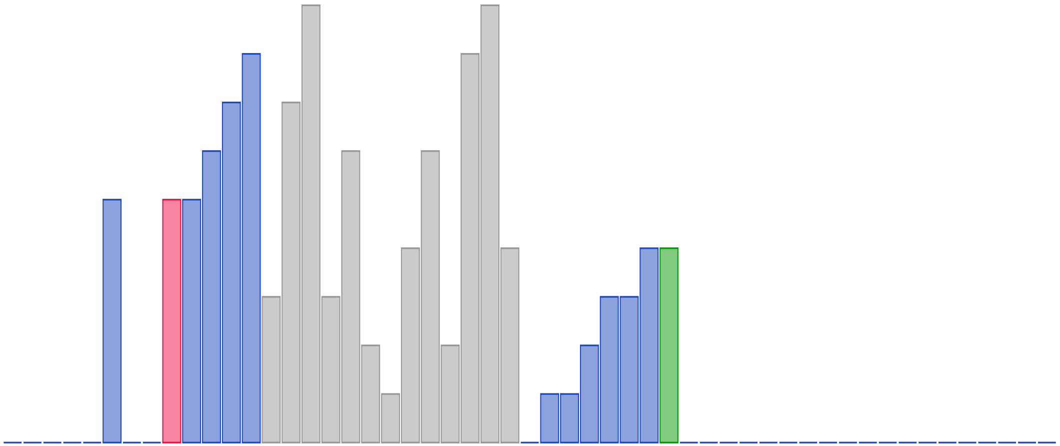


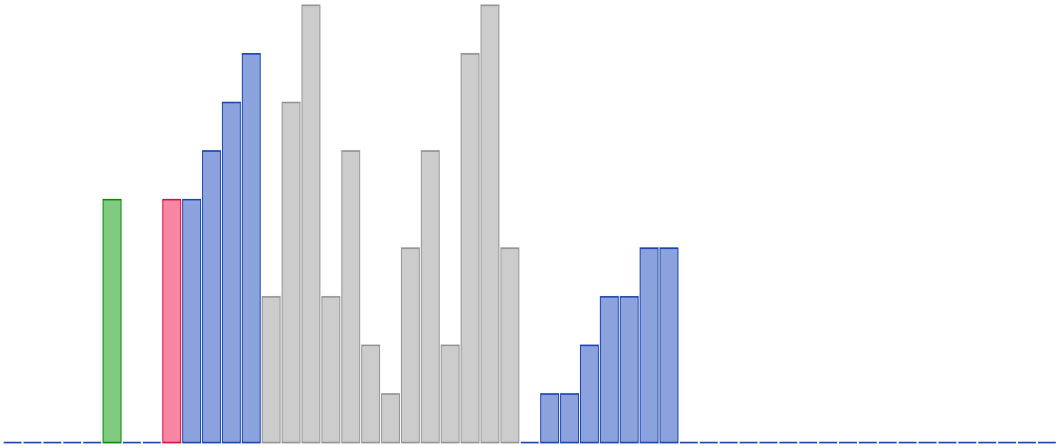


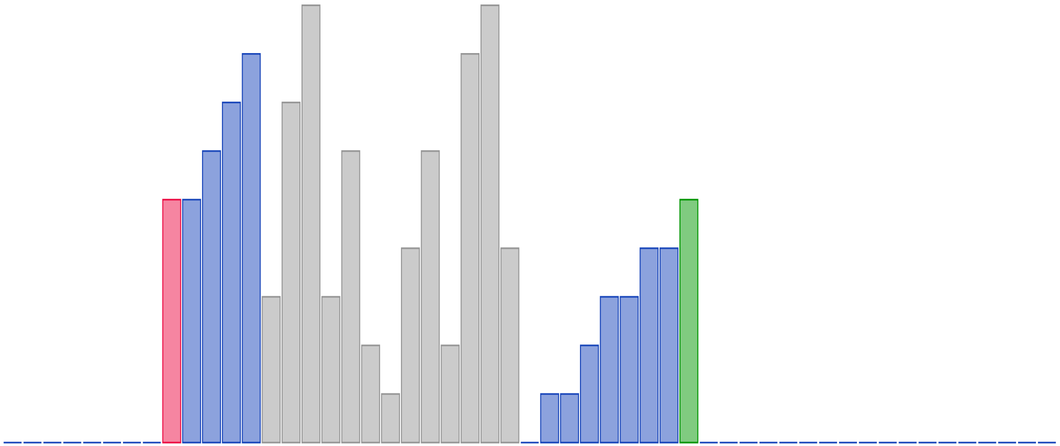


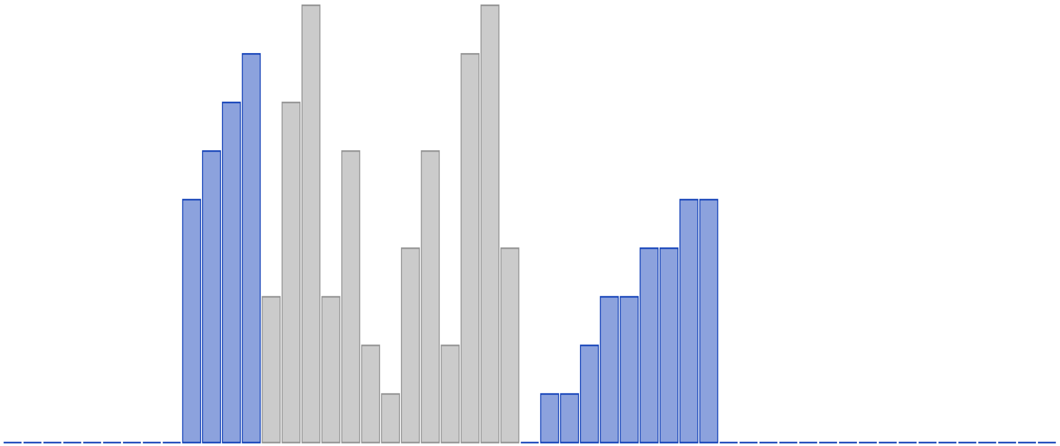


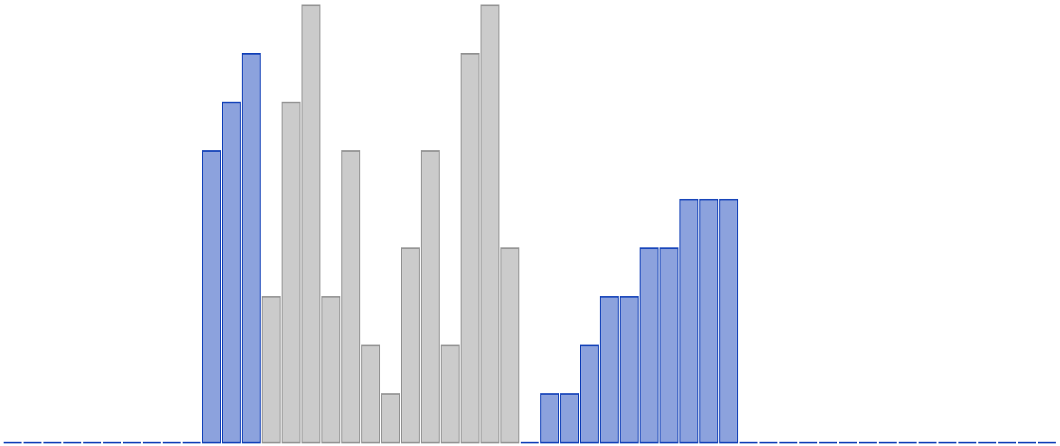


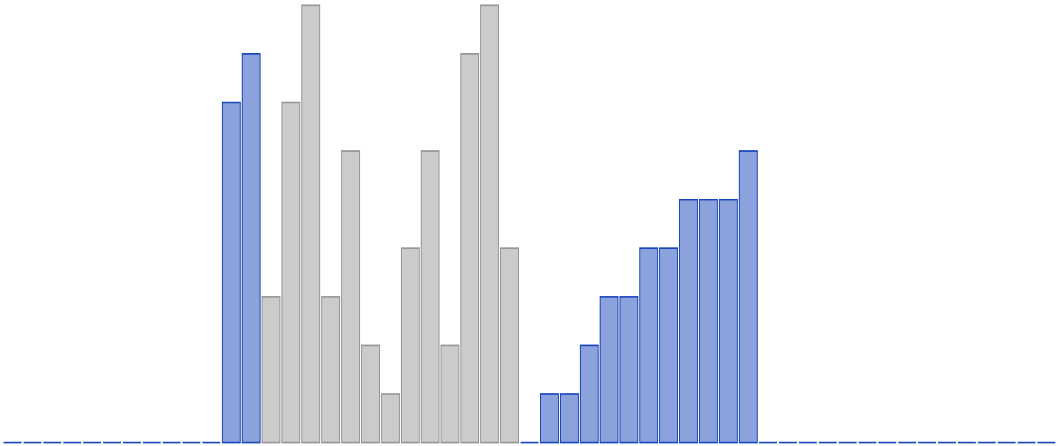


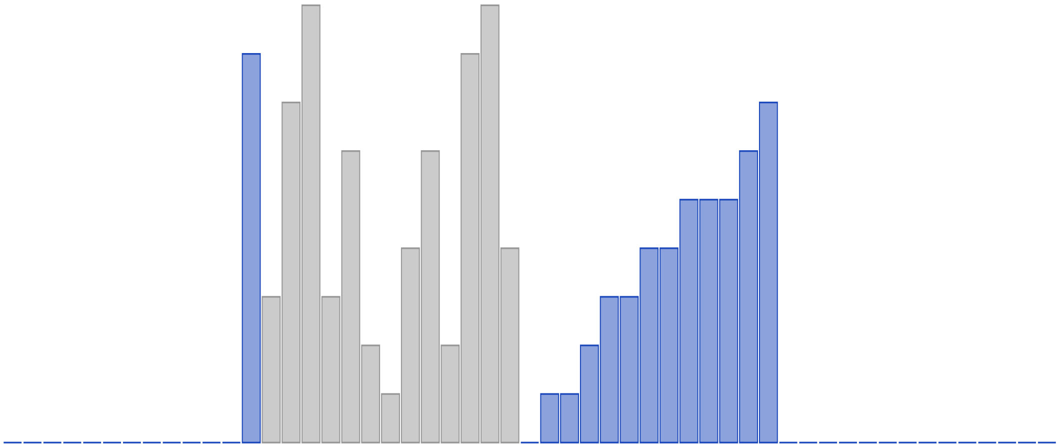


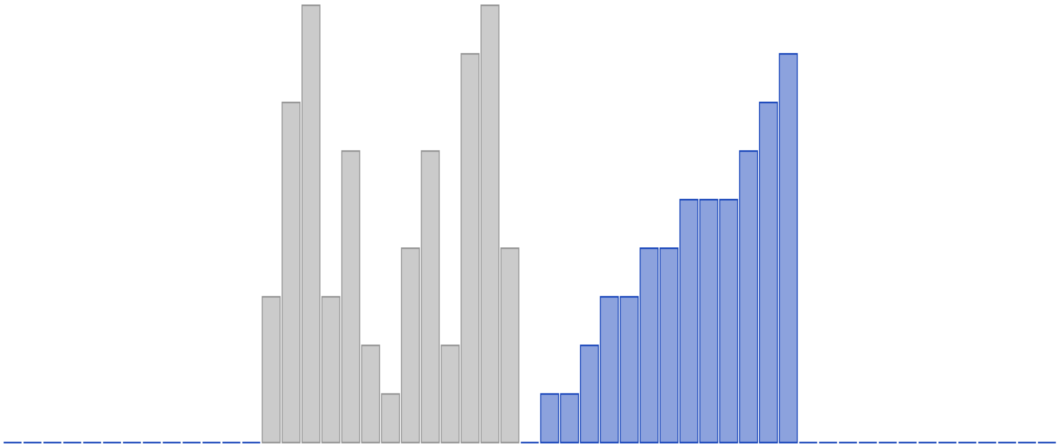


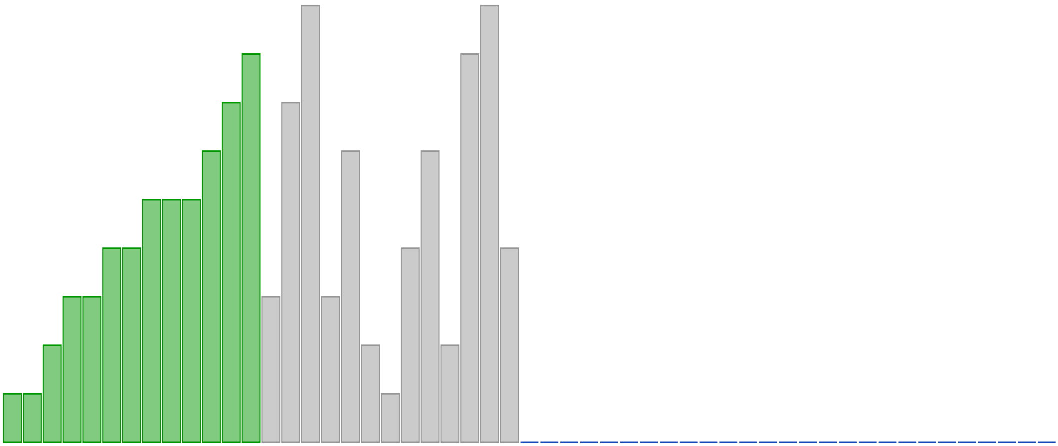


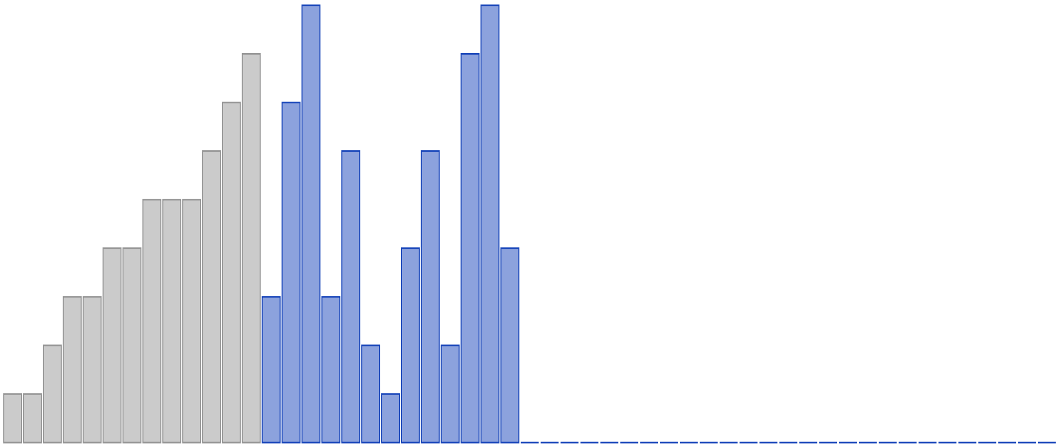


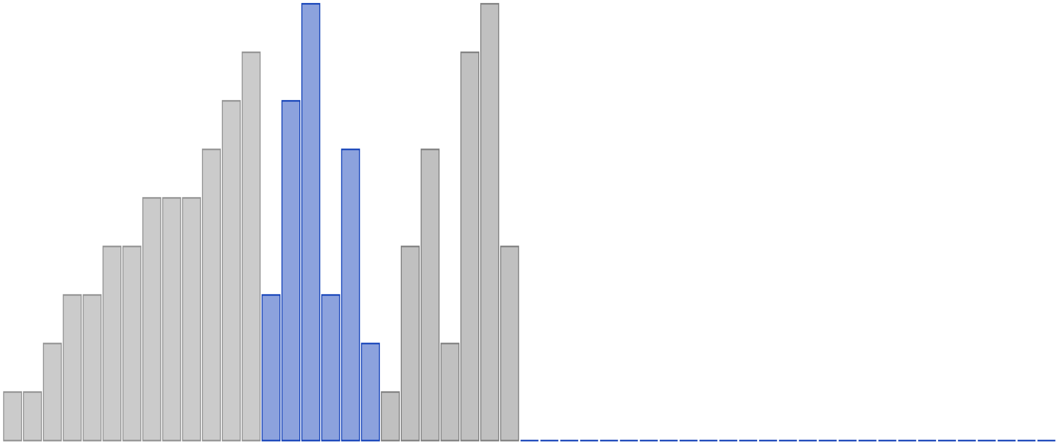


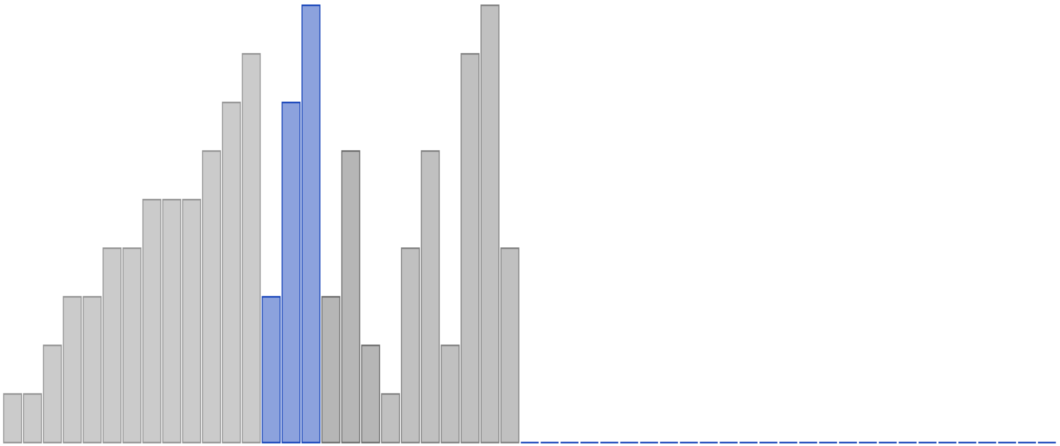


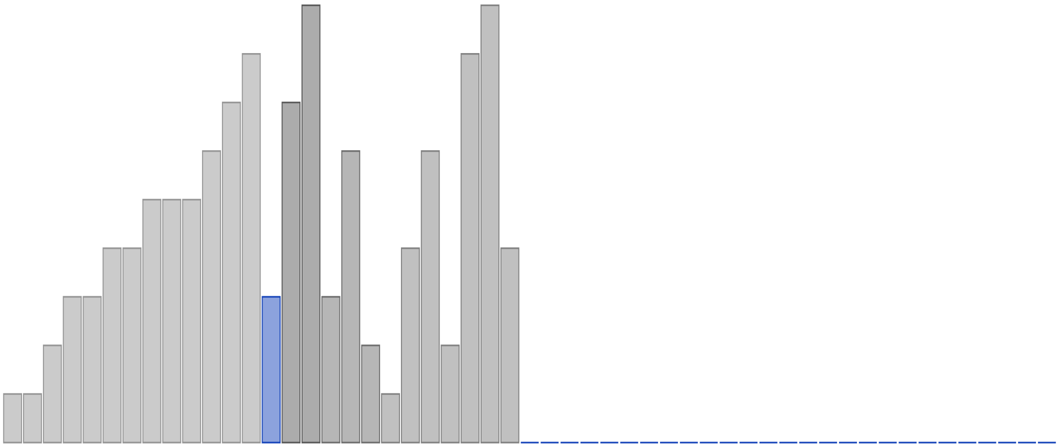


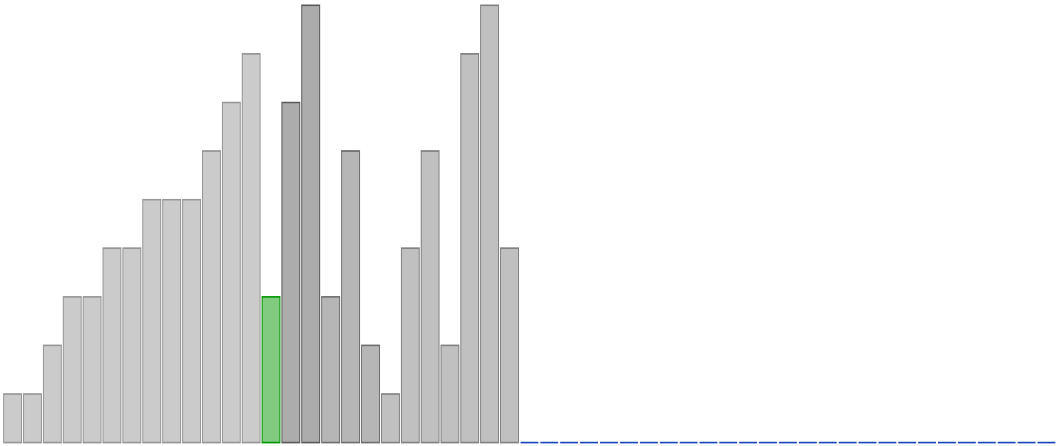


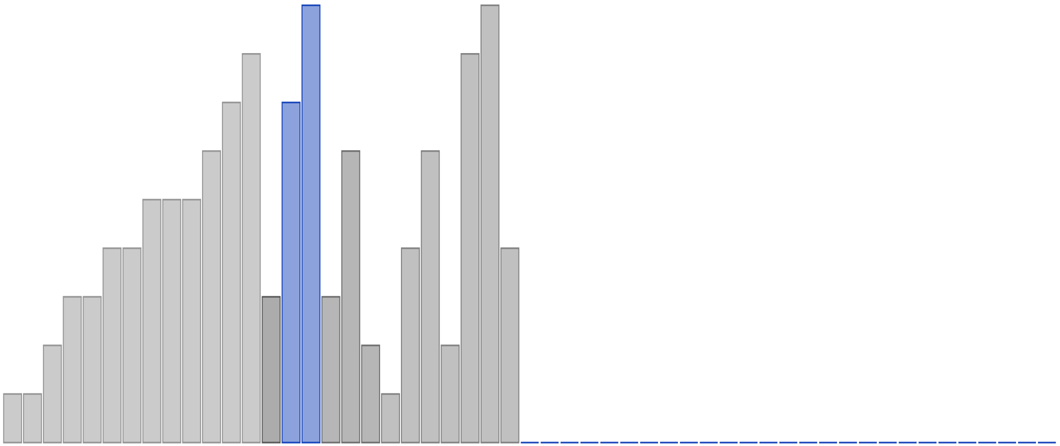


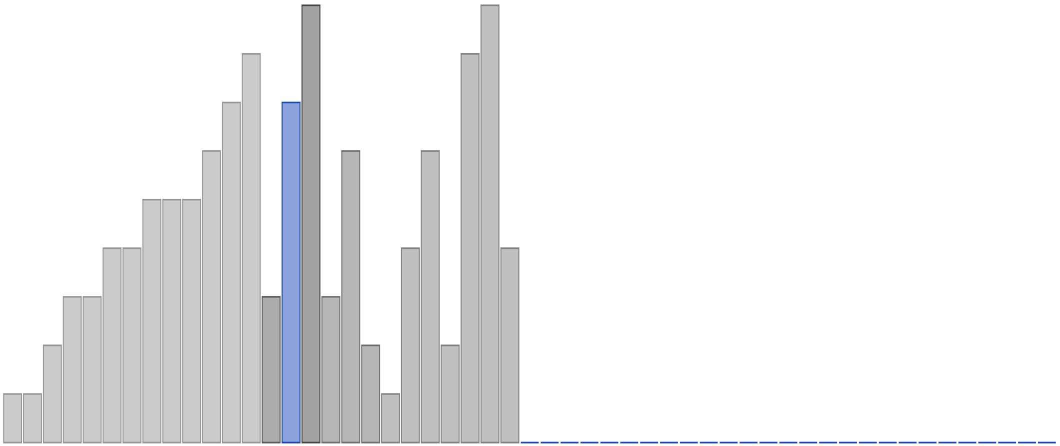


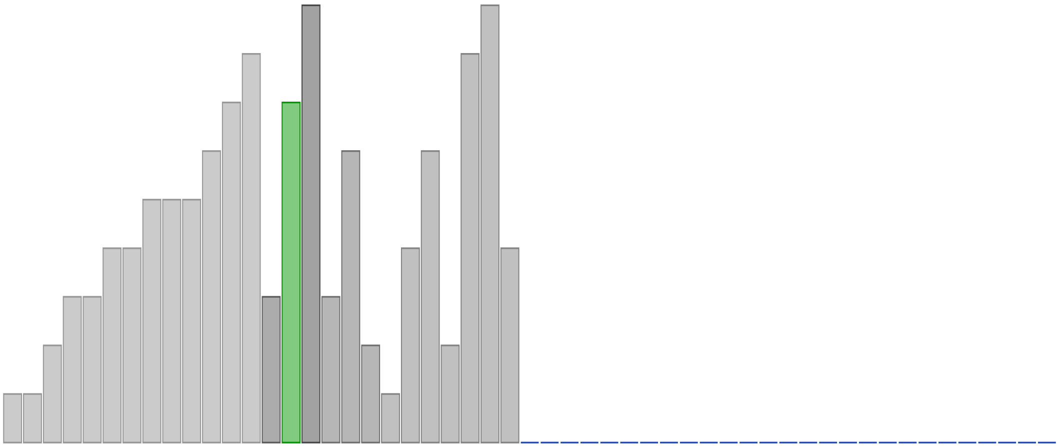


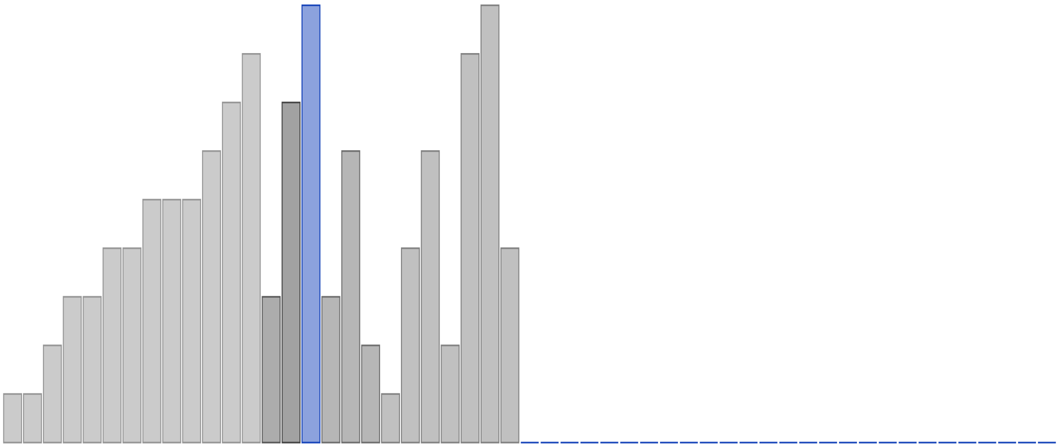


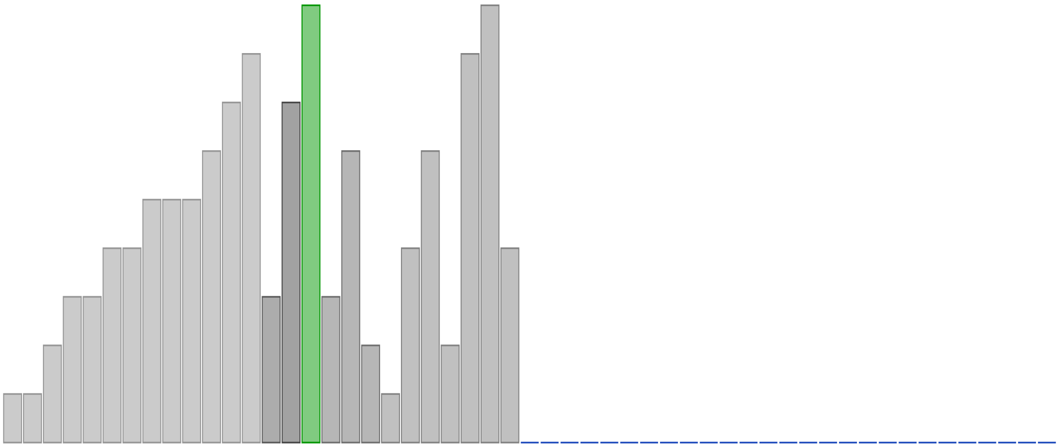


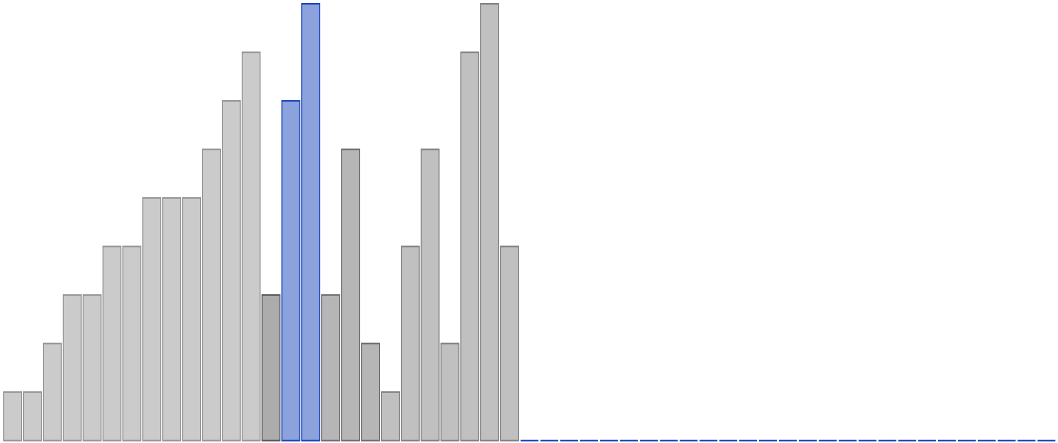


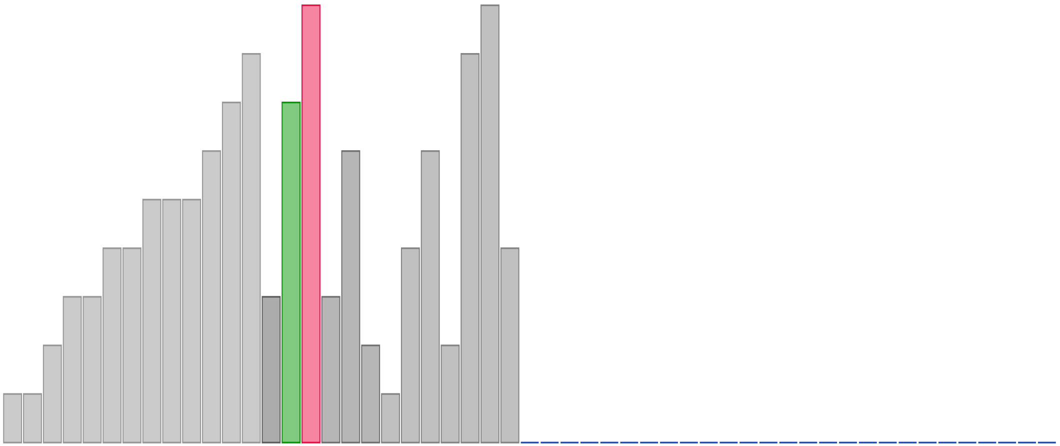


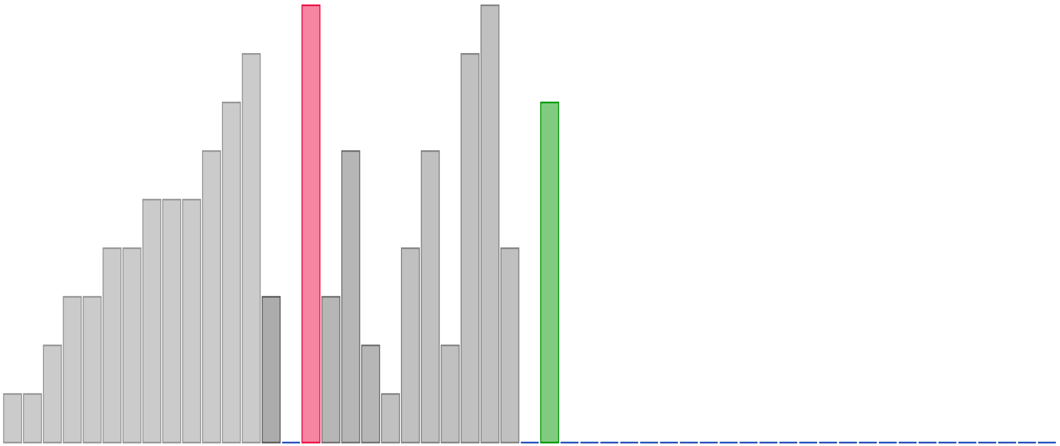


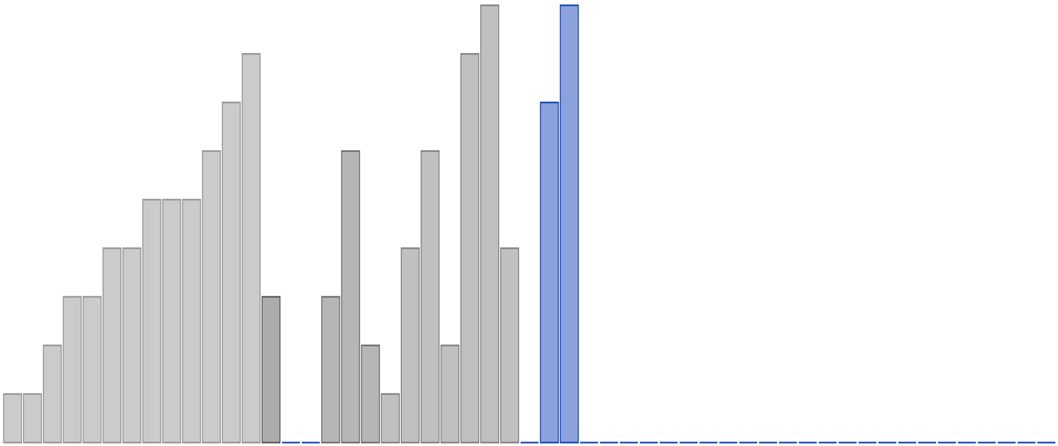


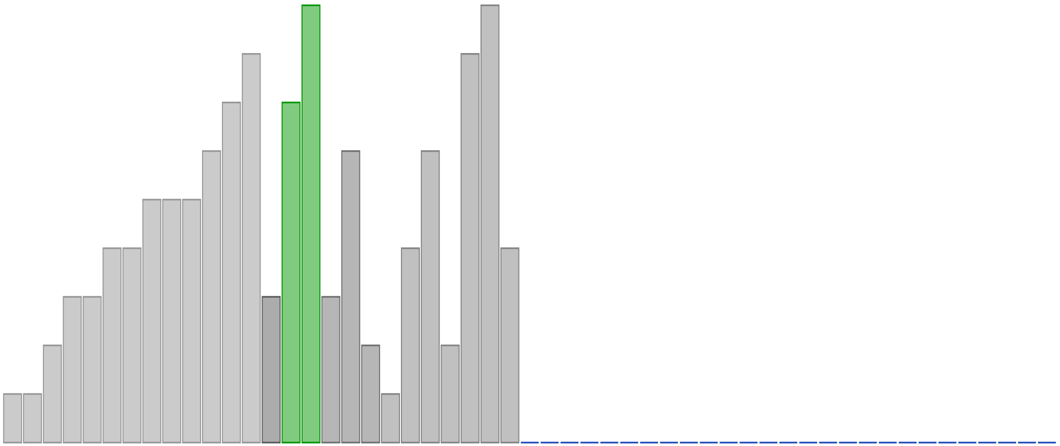


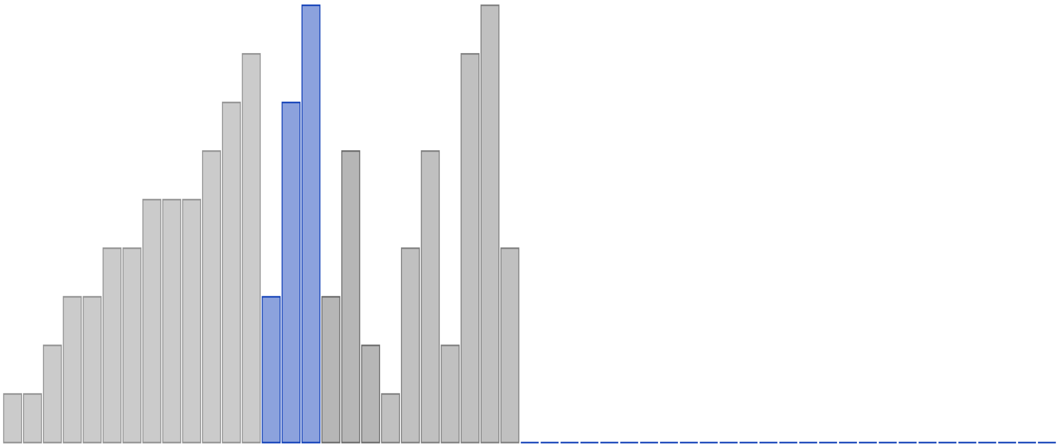


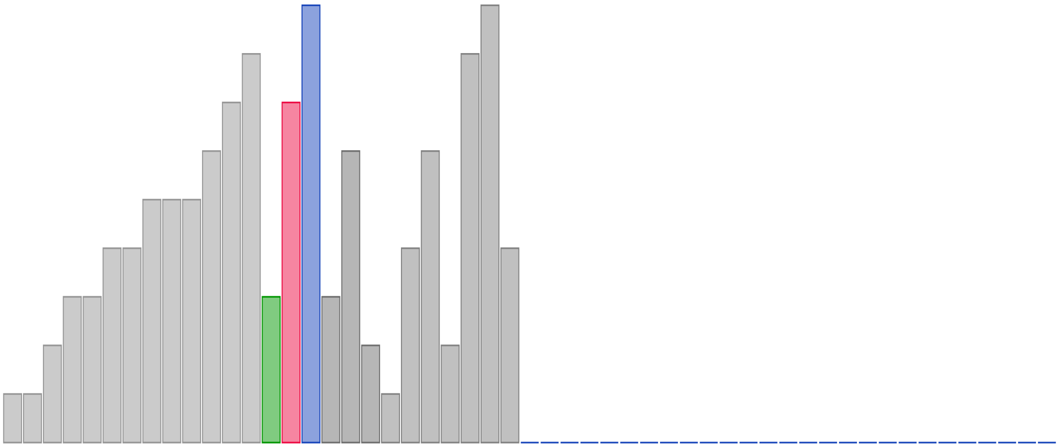


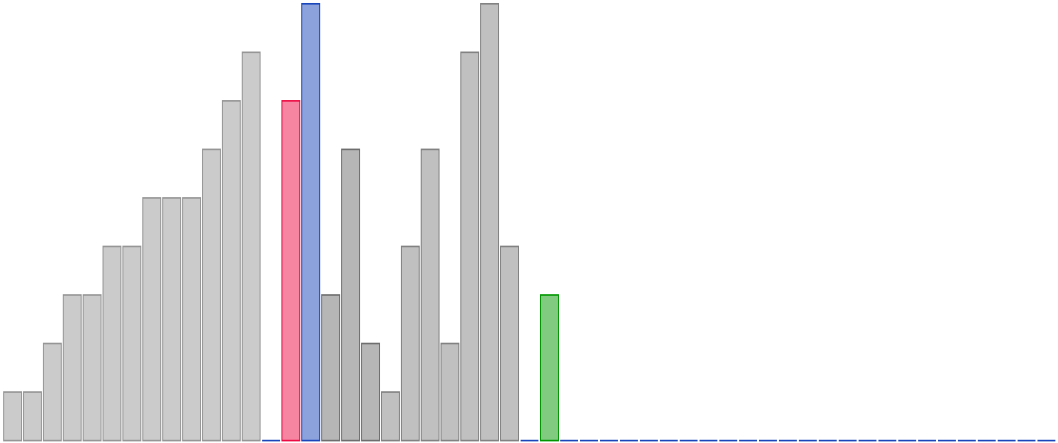


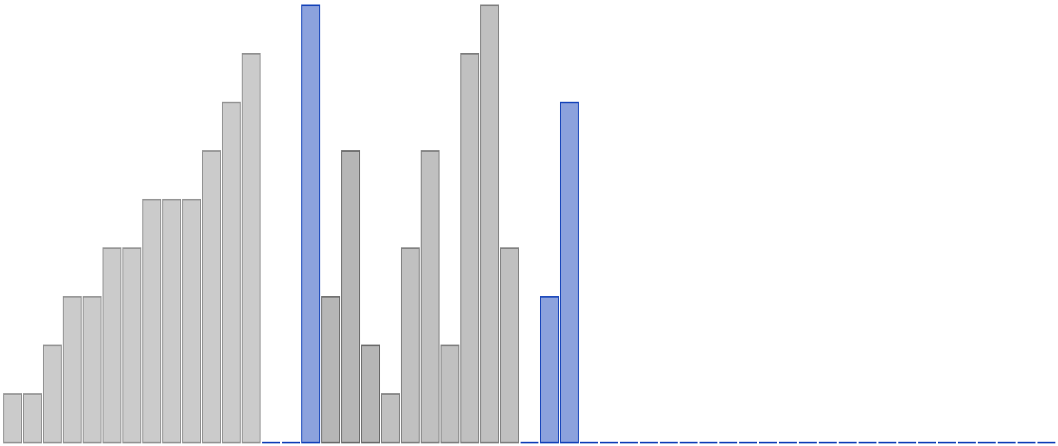


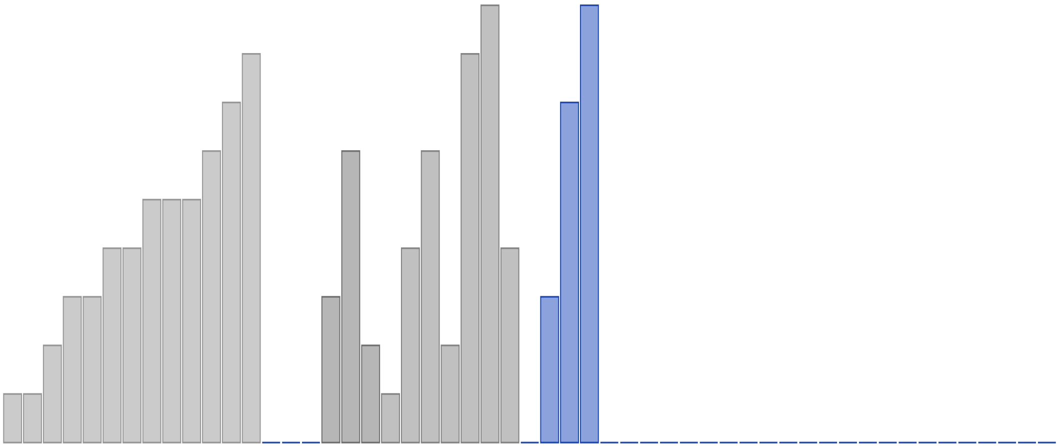


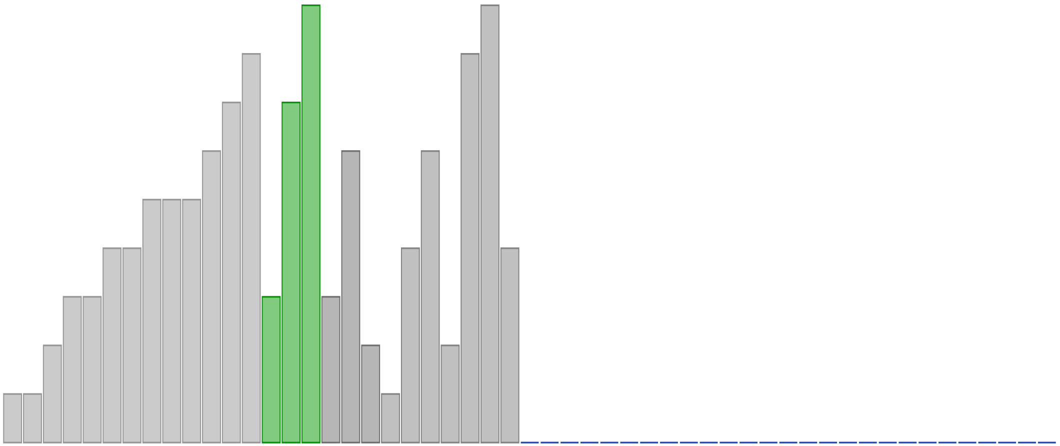


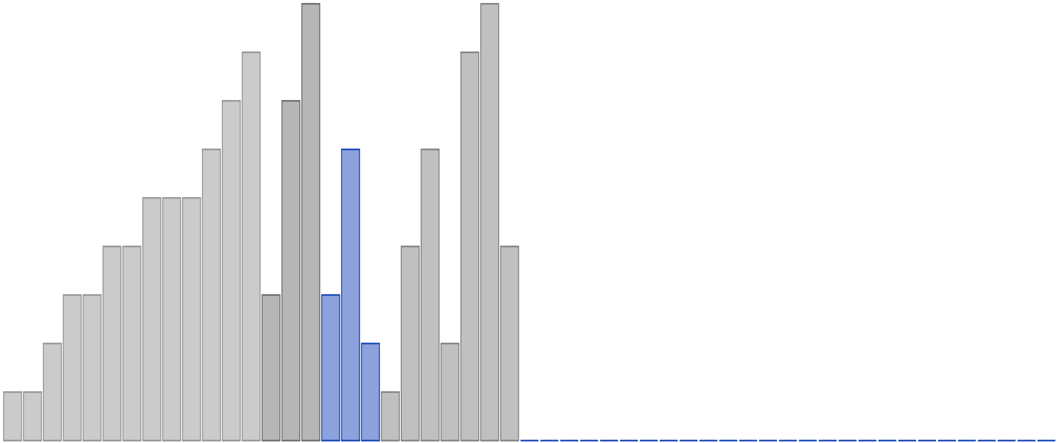


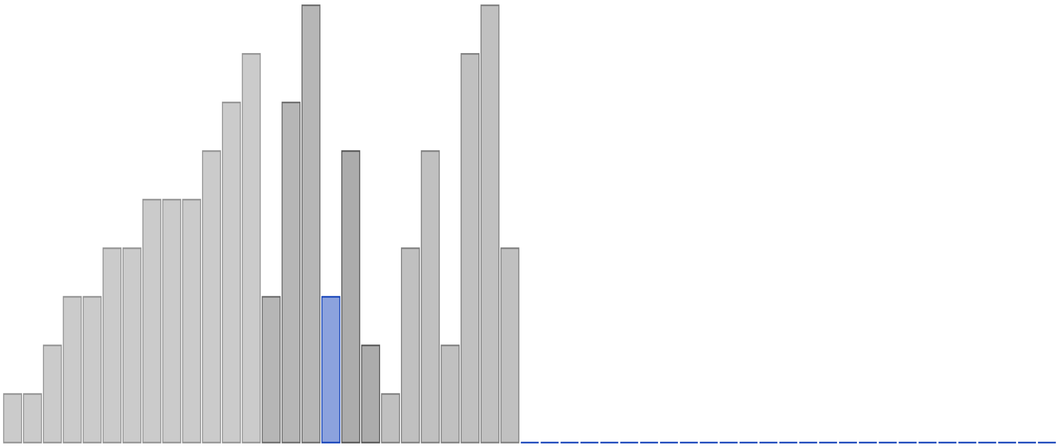


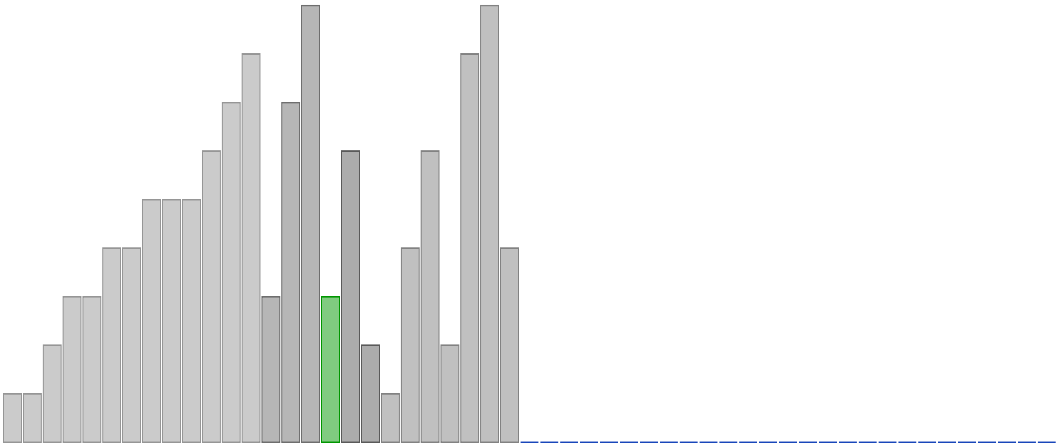


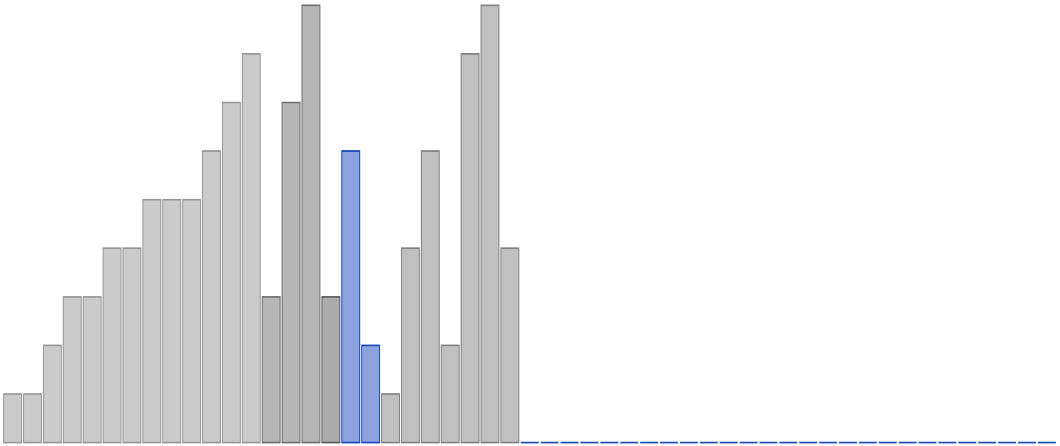


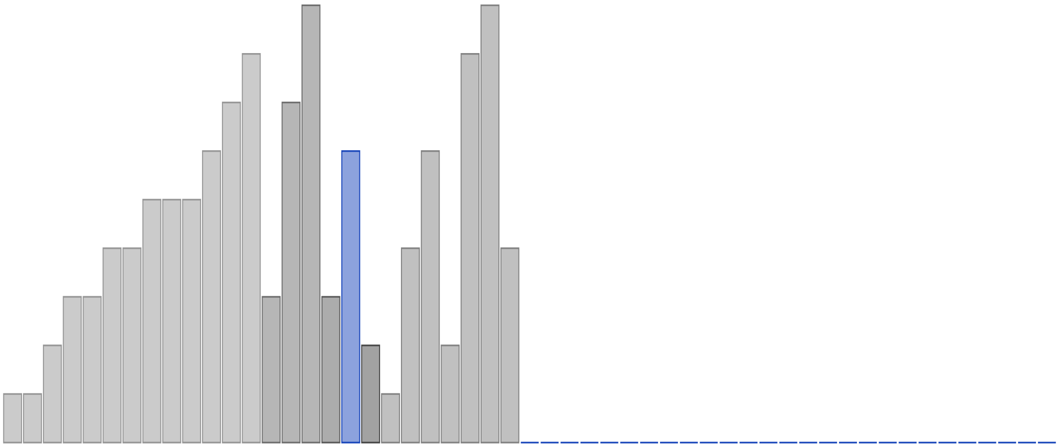


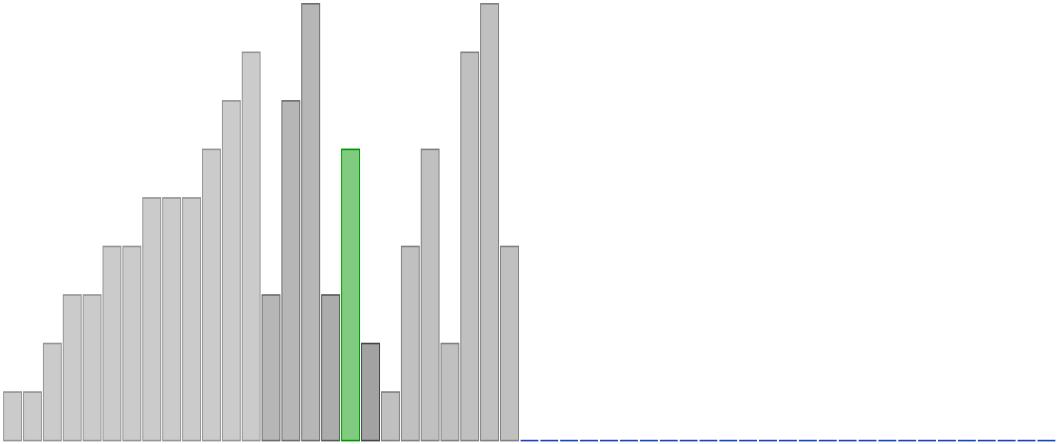


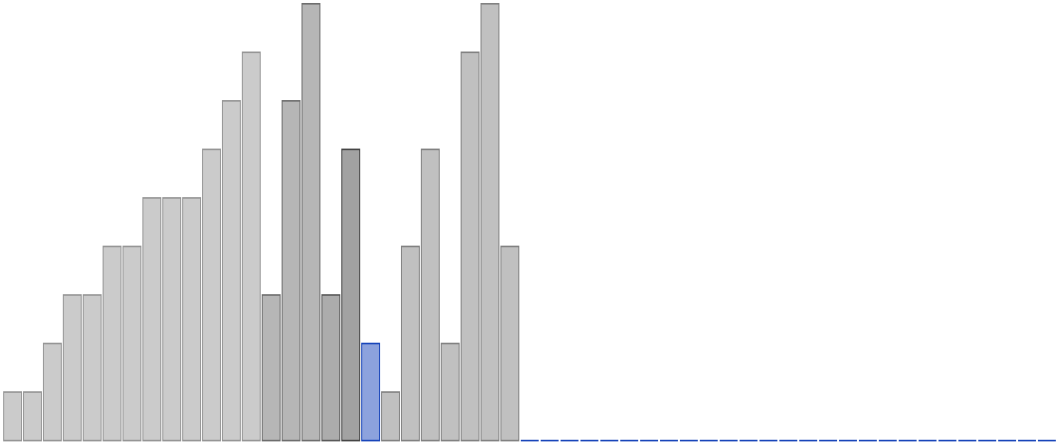


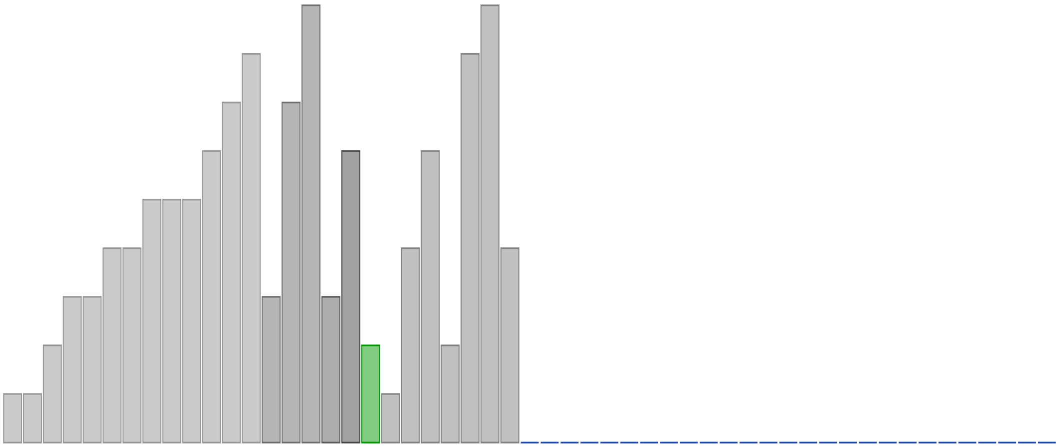


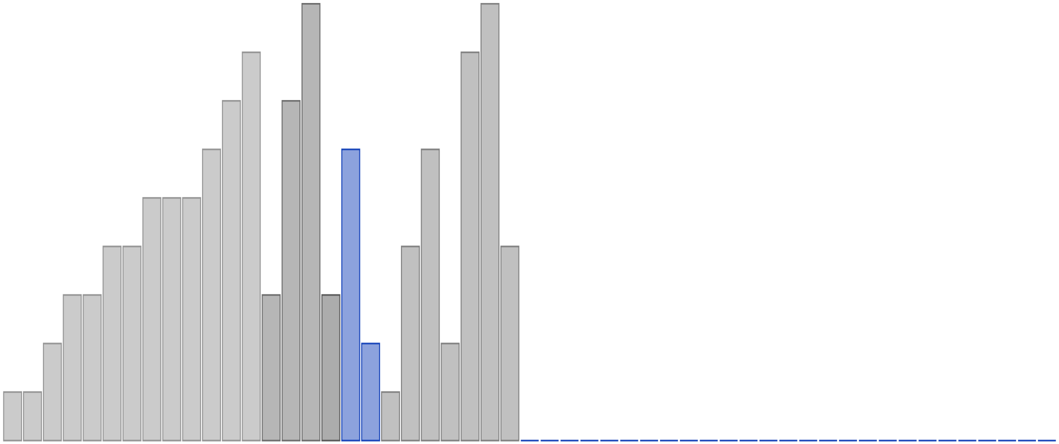


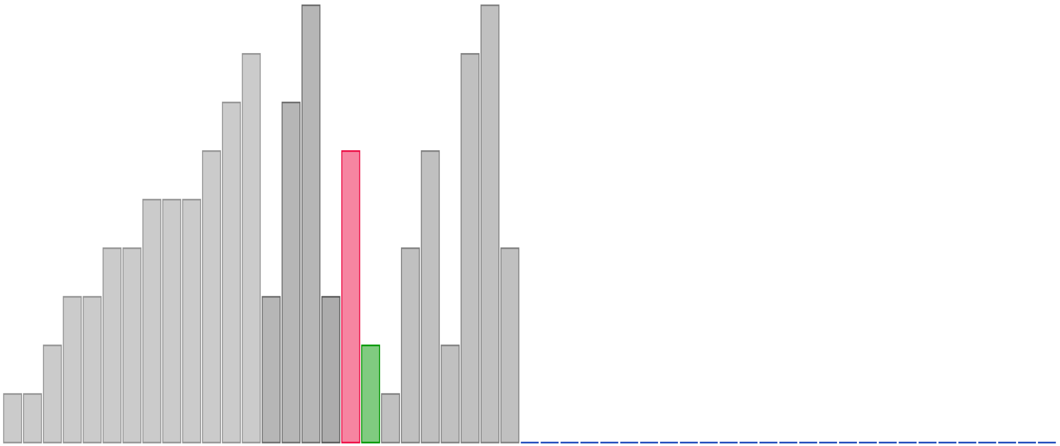


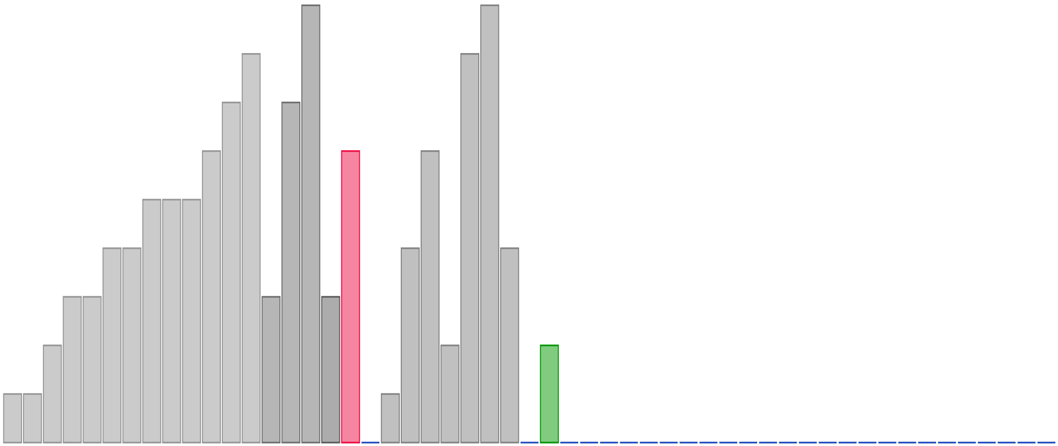


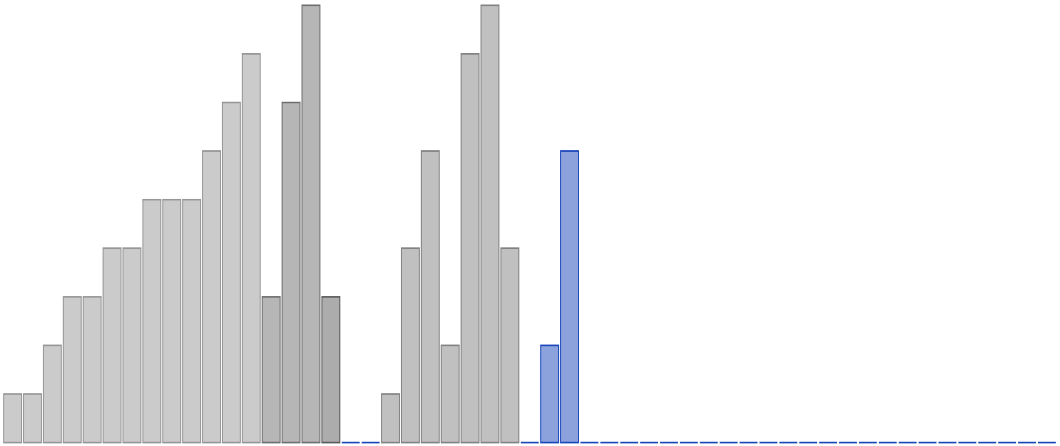


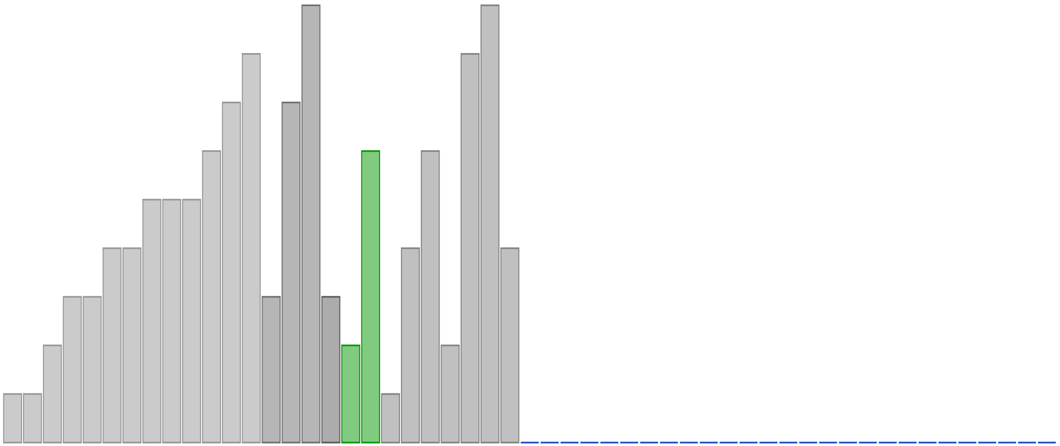


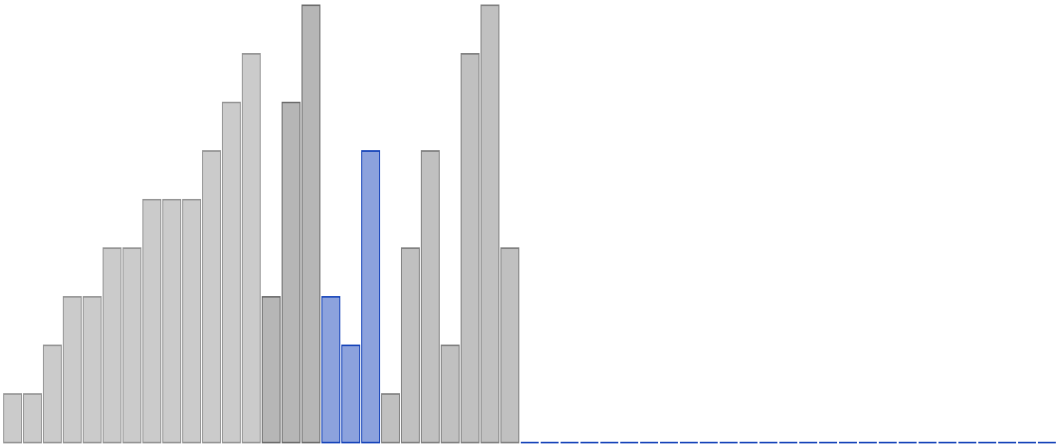


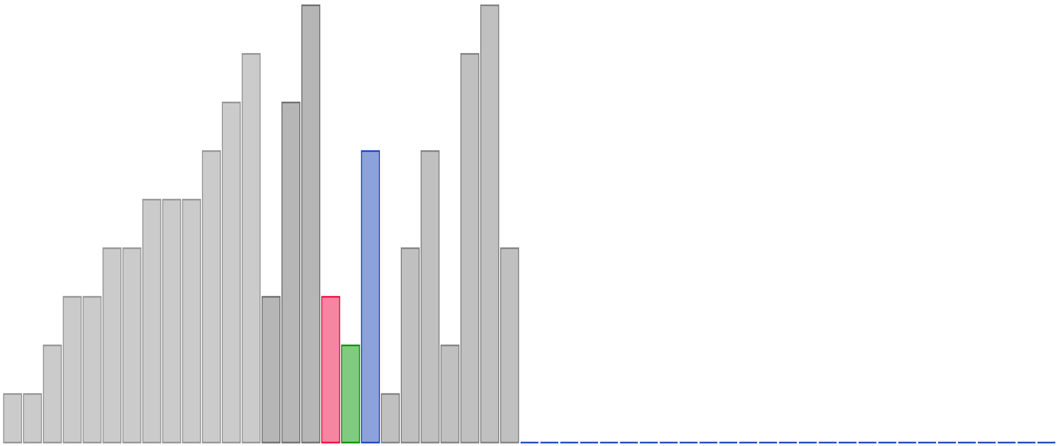


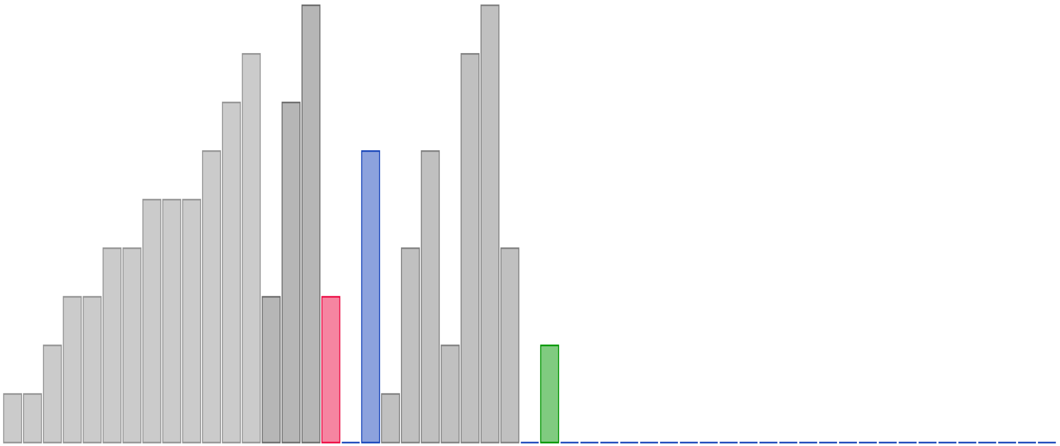


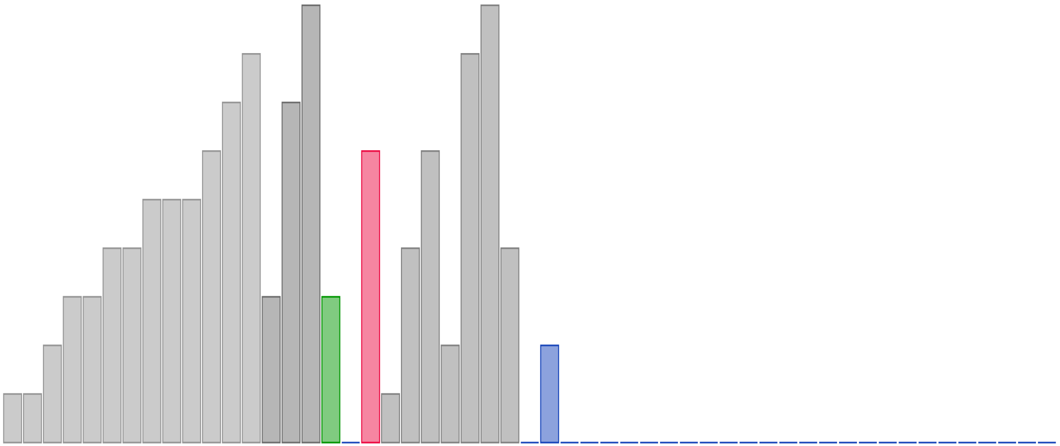


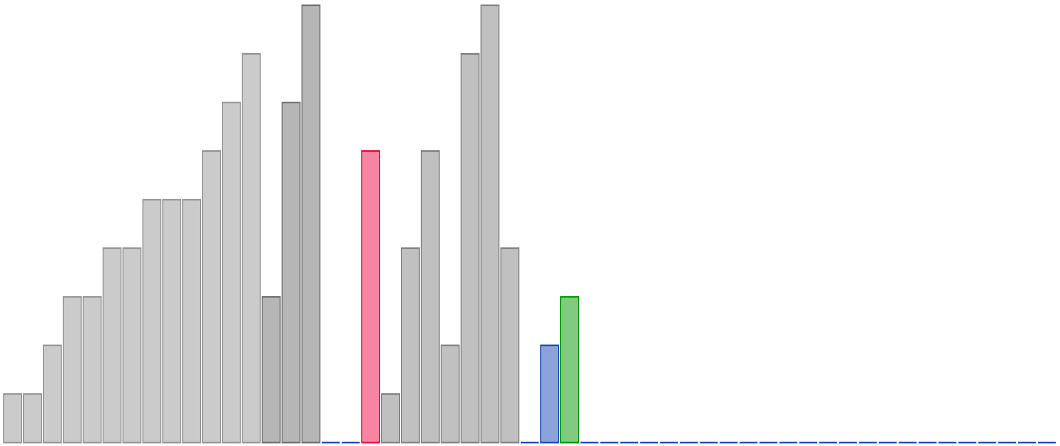


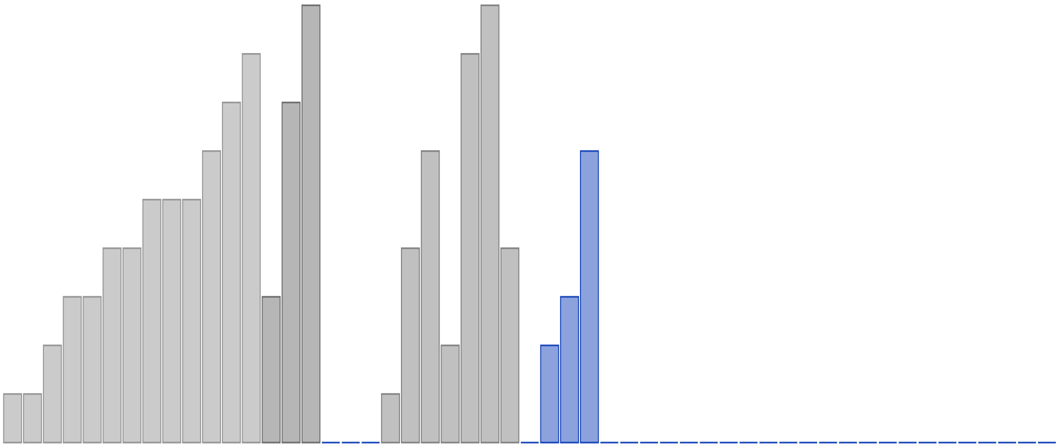


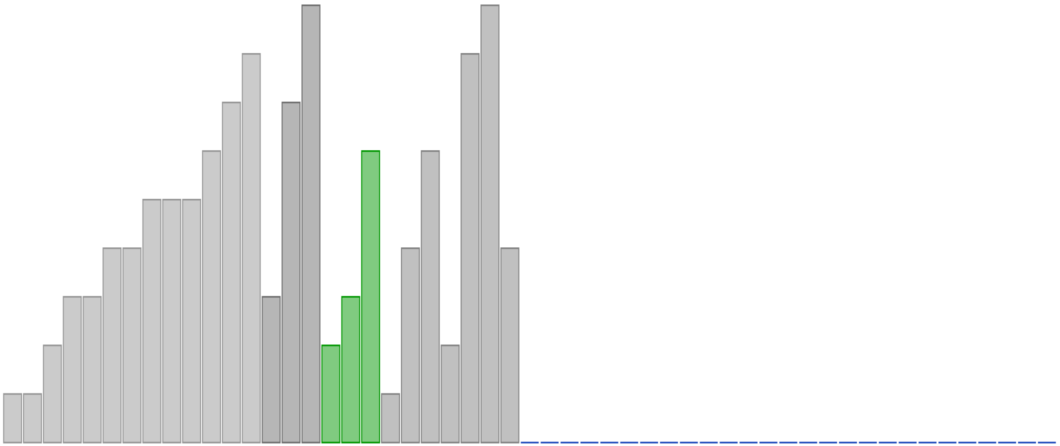


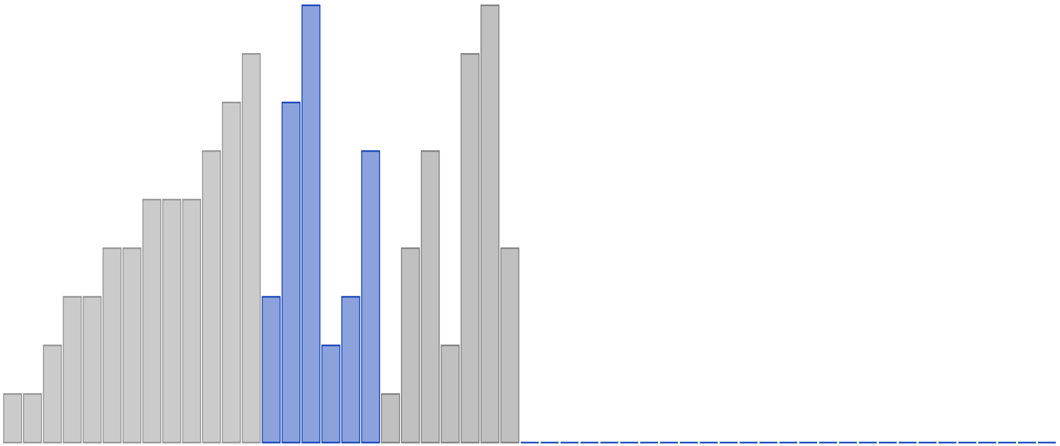


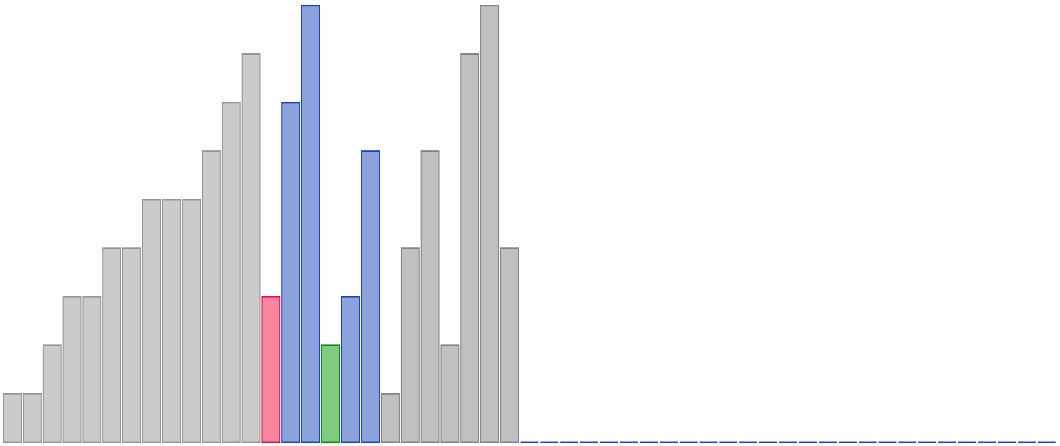


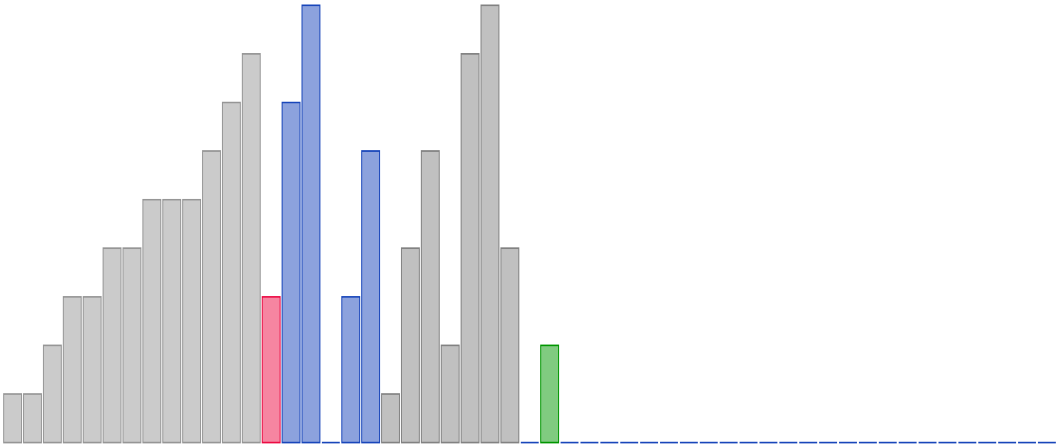


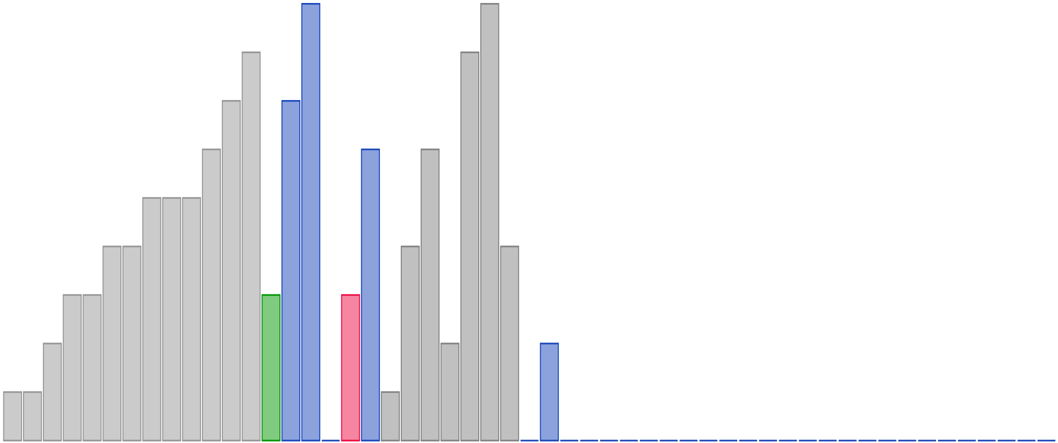


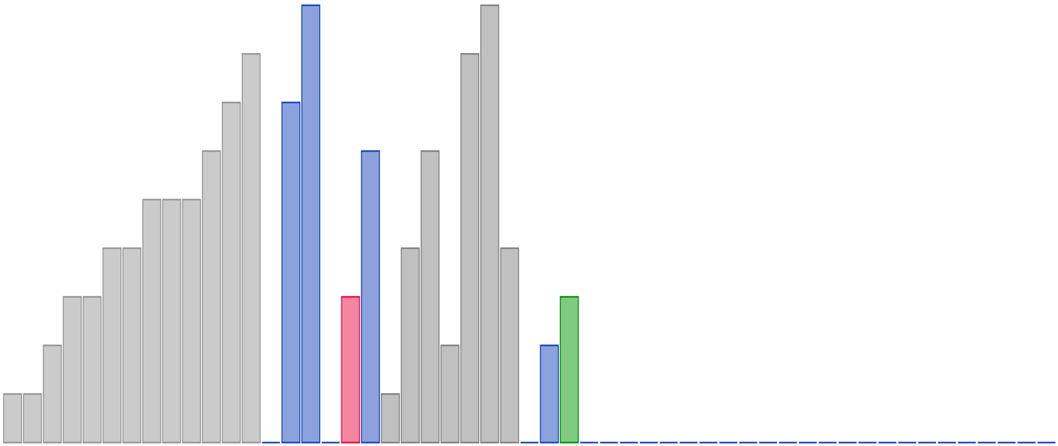




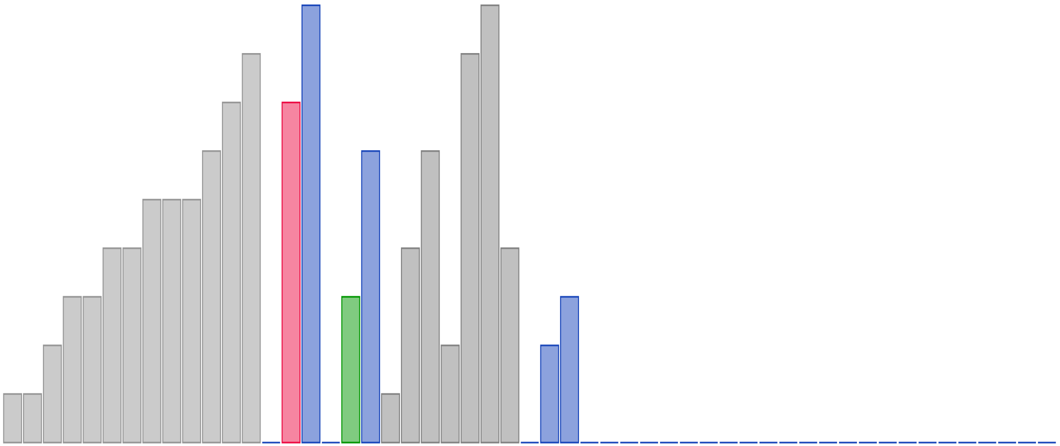


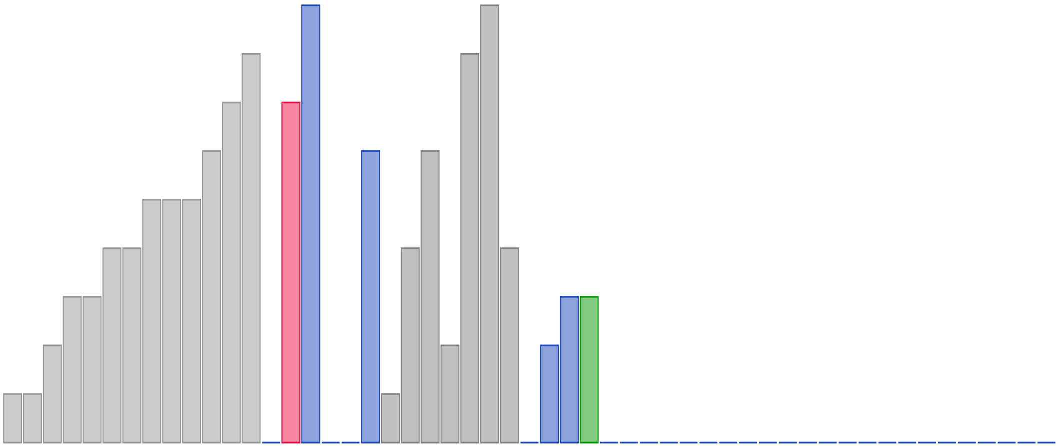


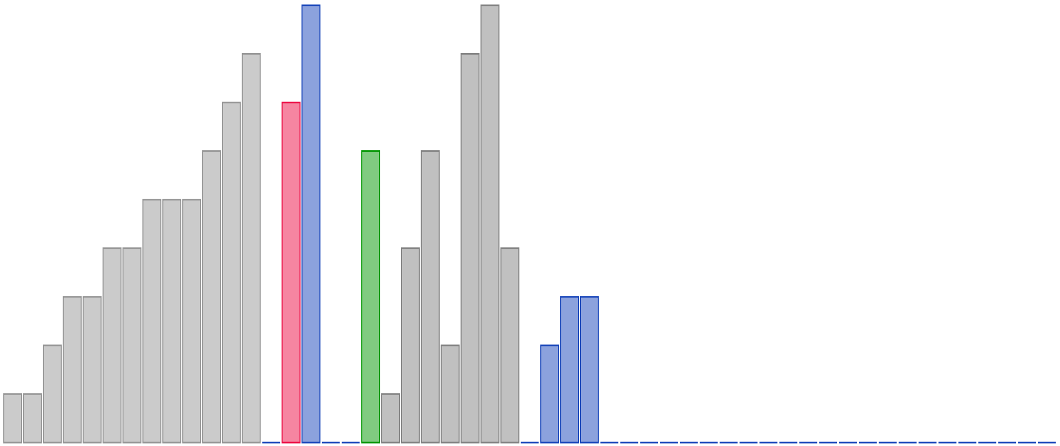


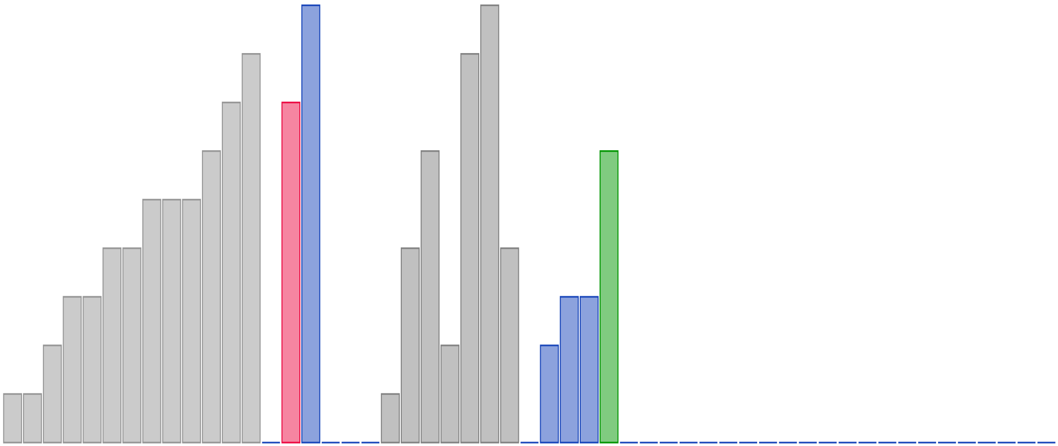


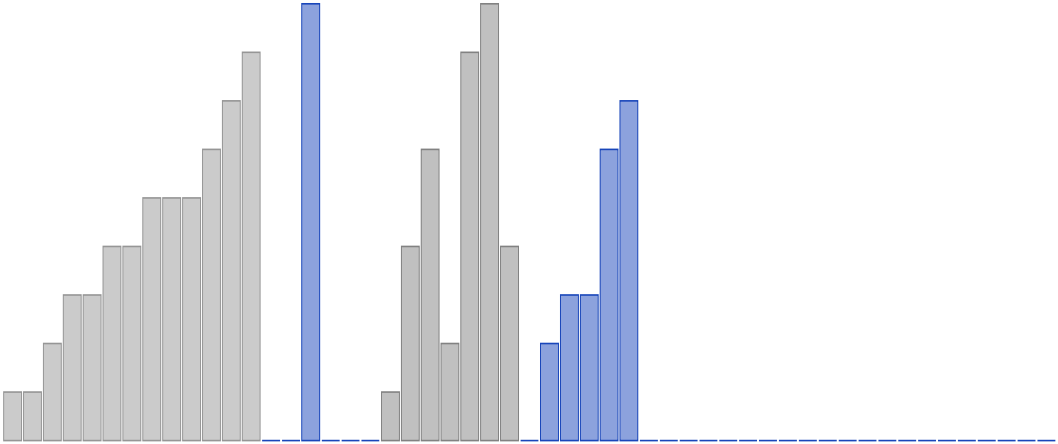
Mergesort: Example

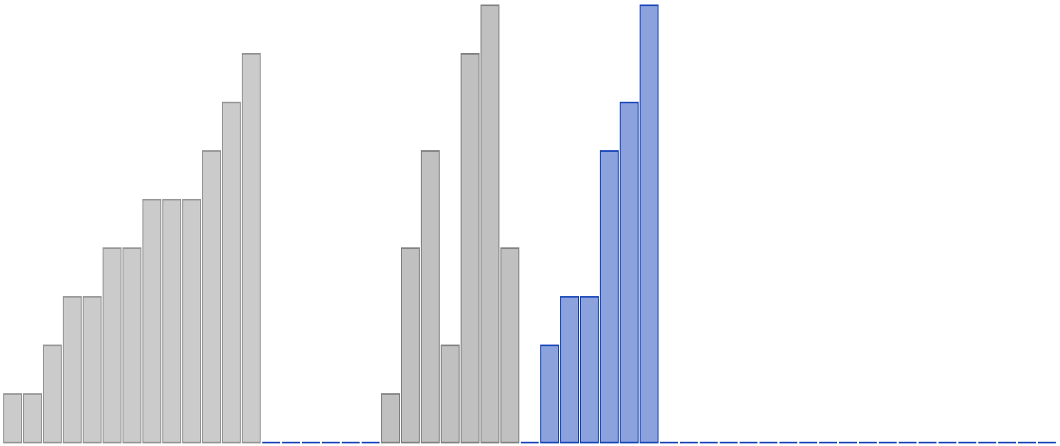


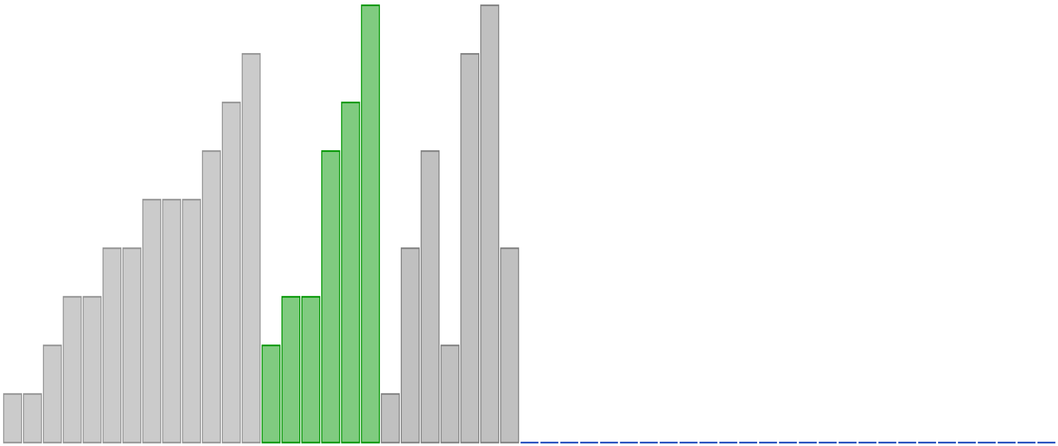


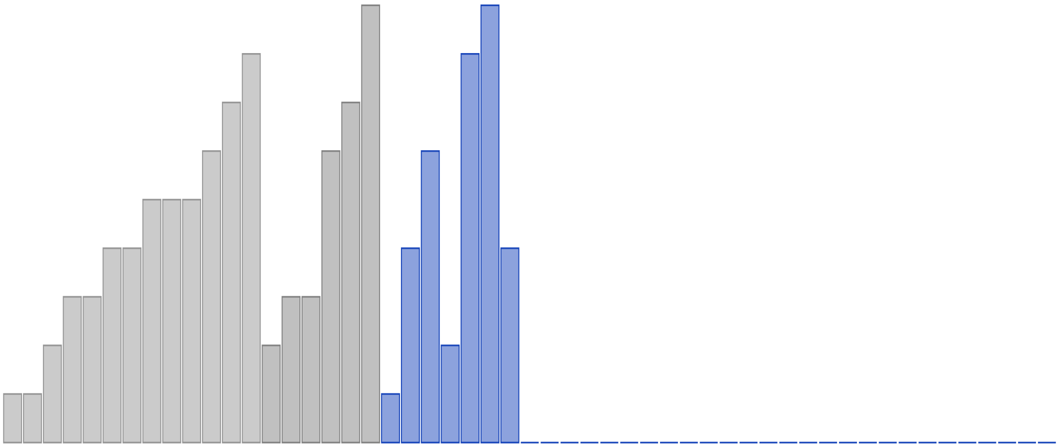


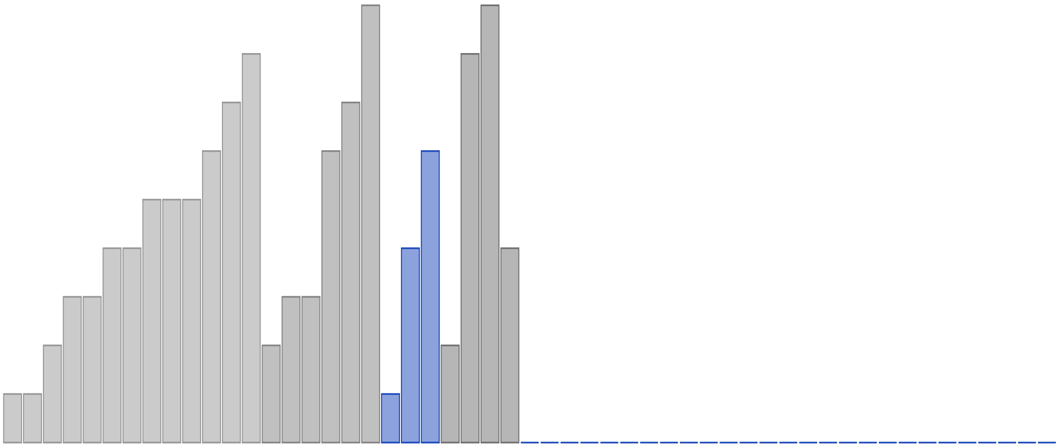


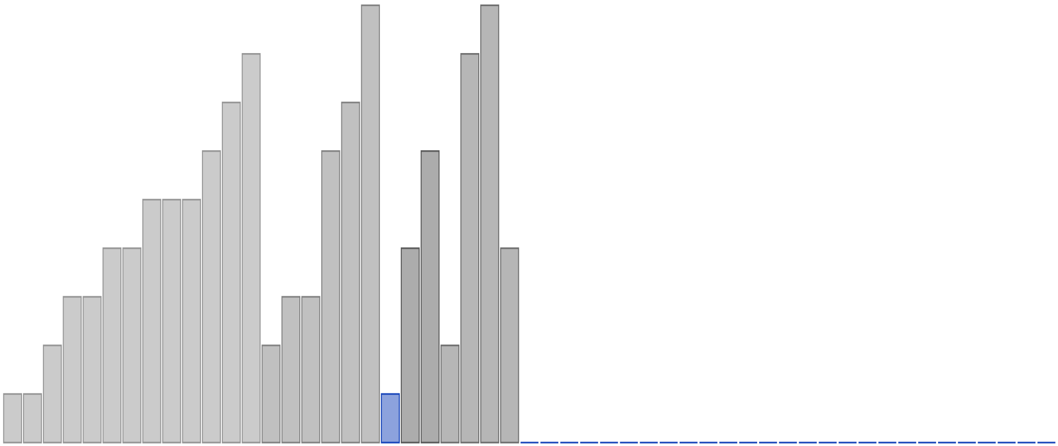


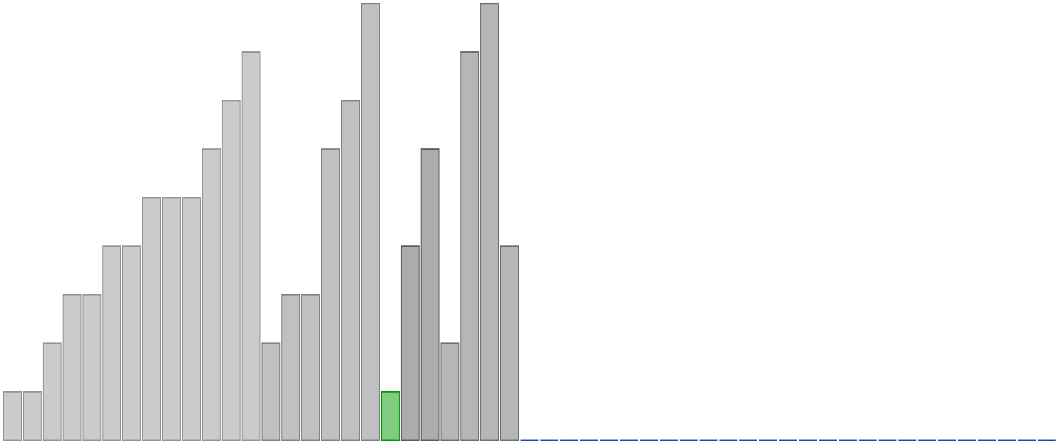


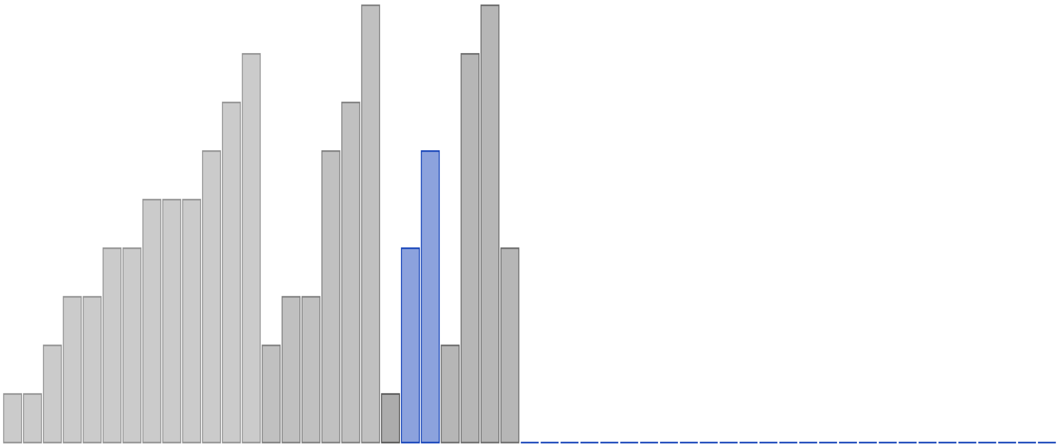


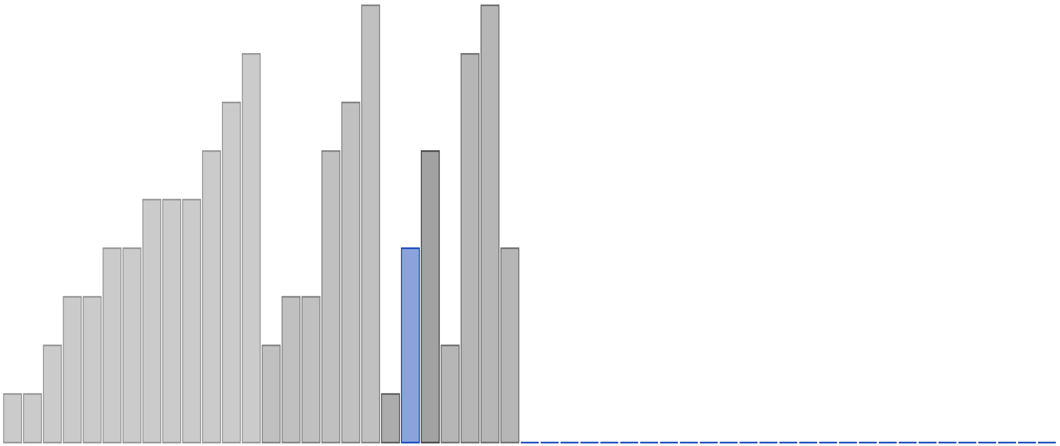


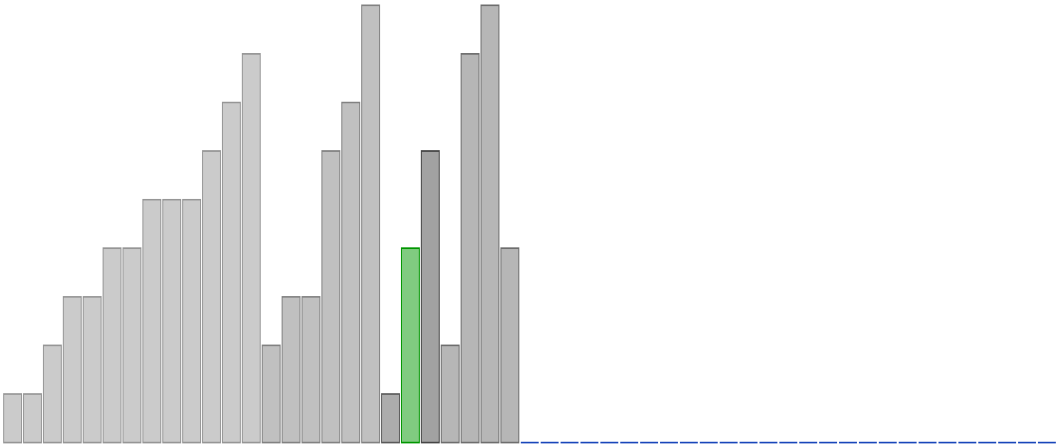


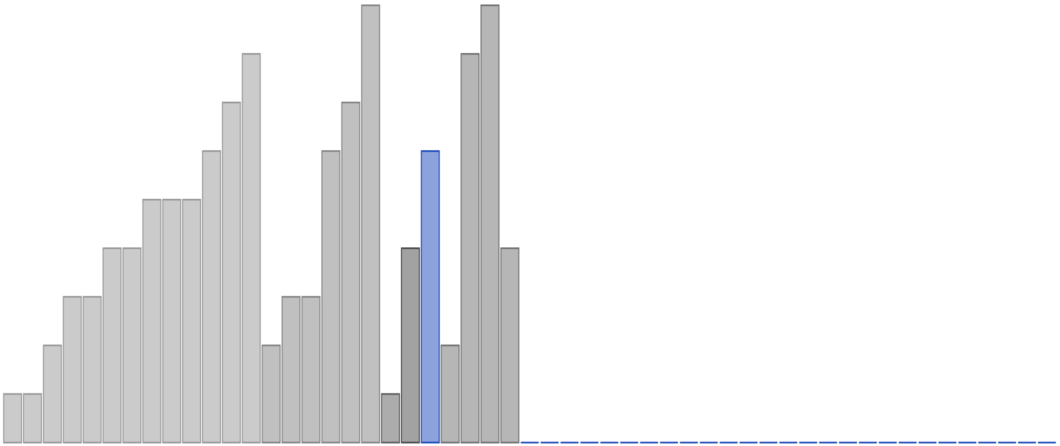


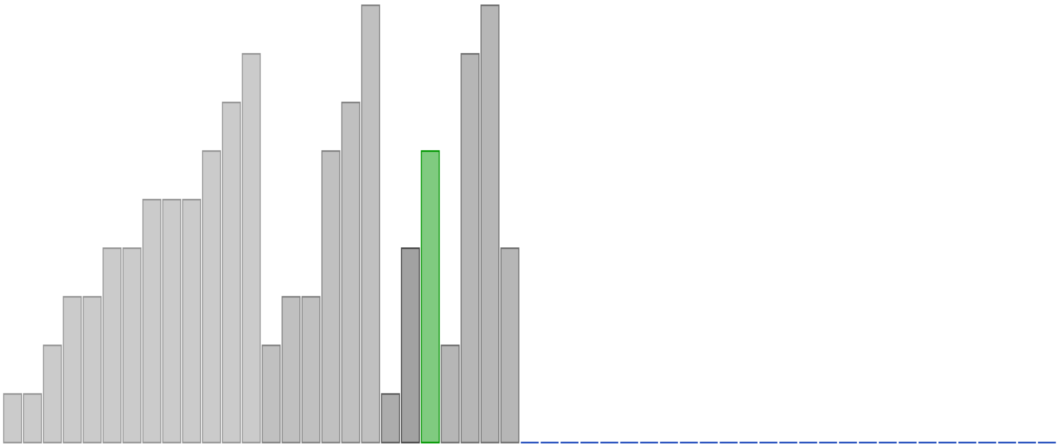


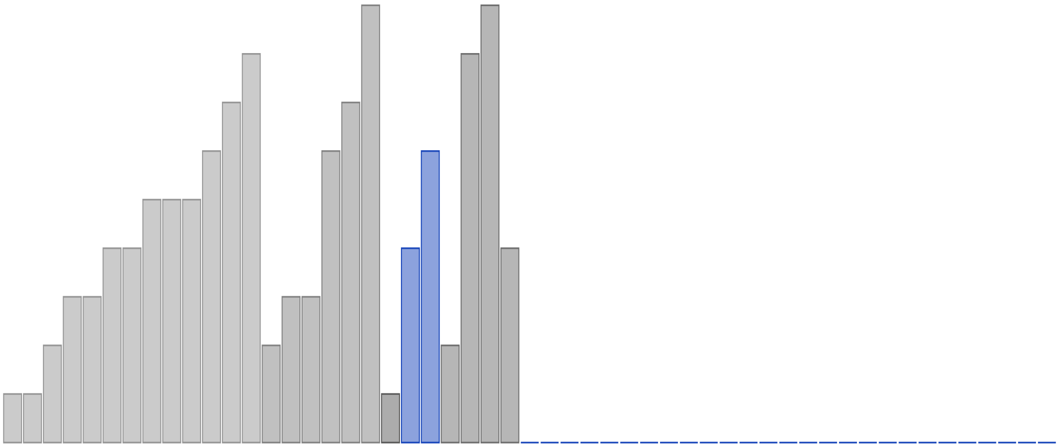


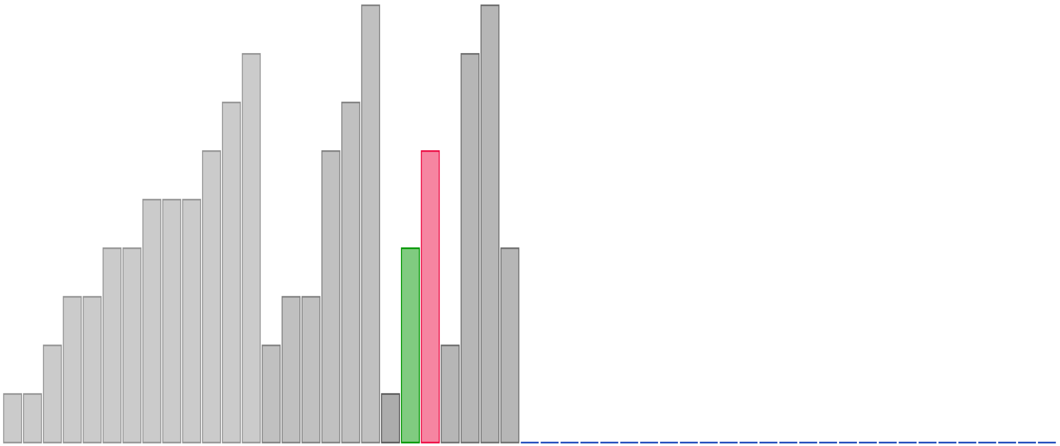


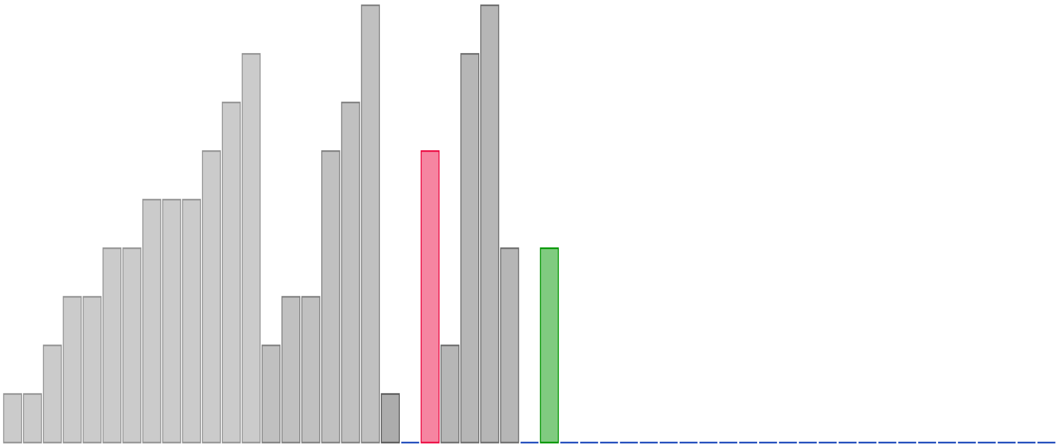


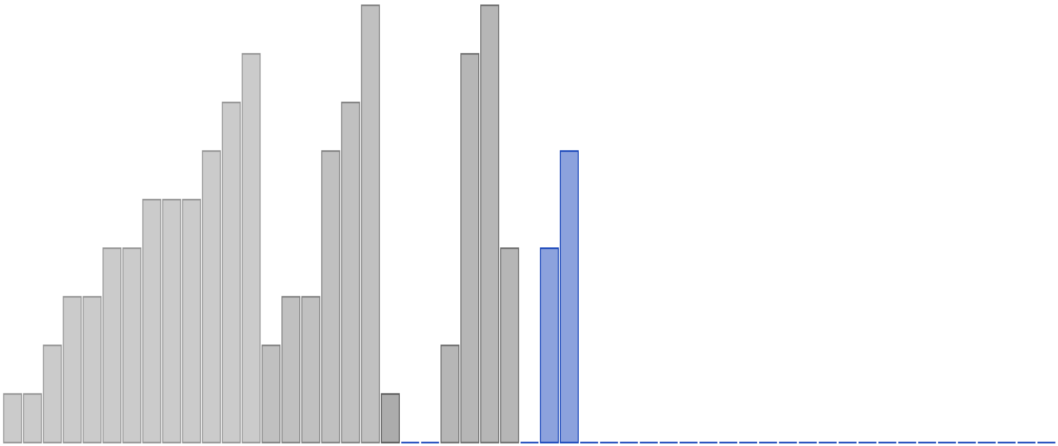


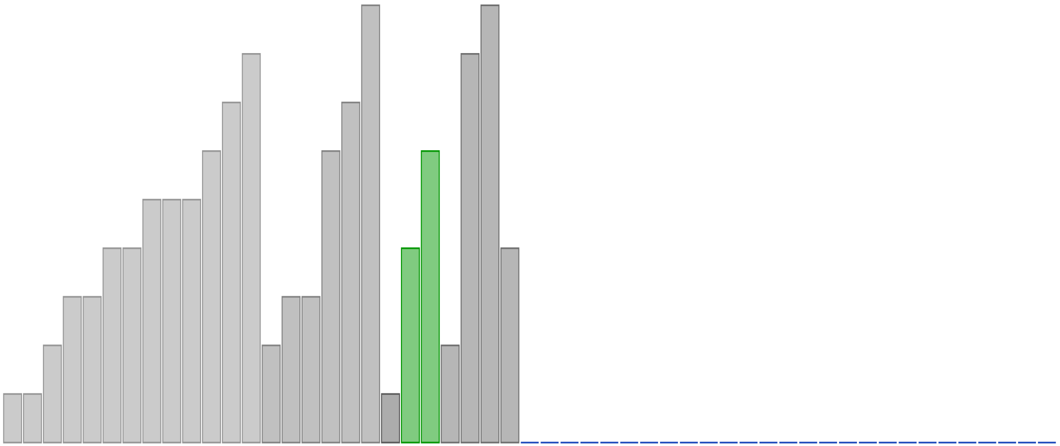


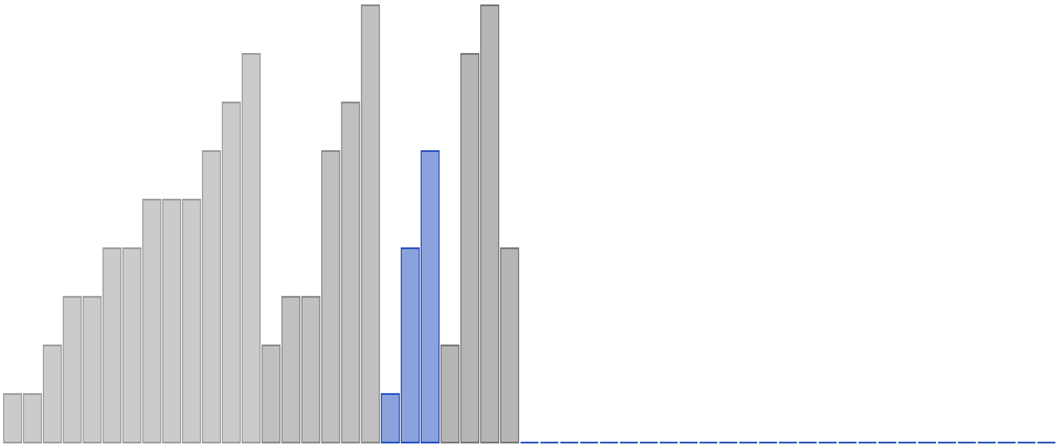


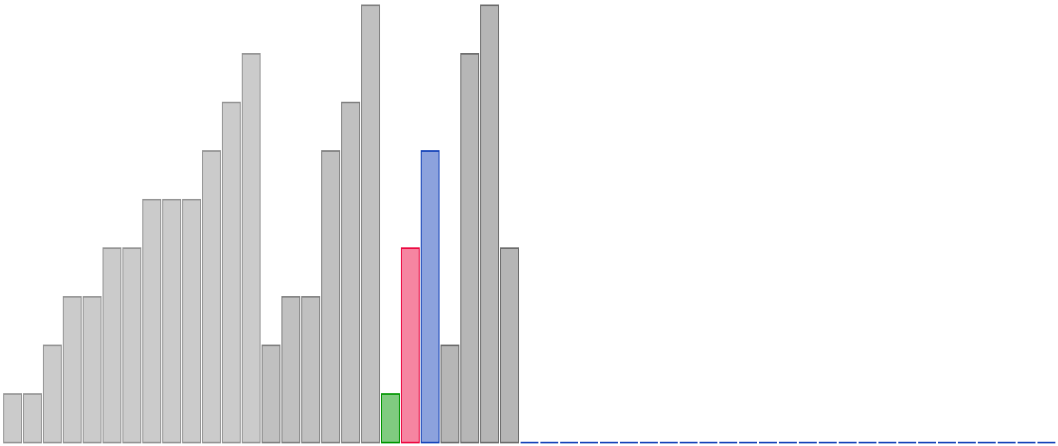


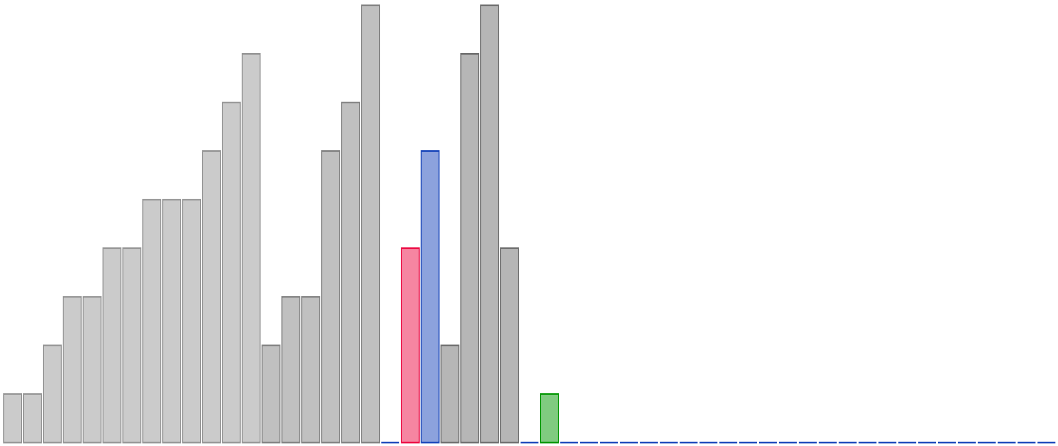


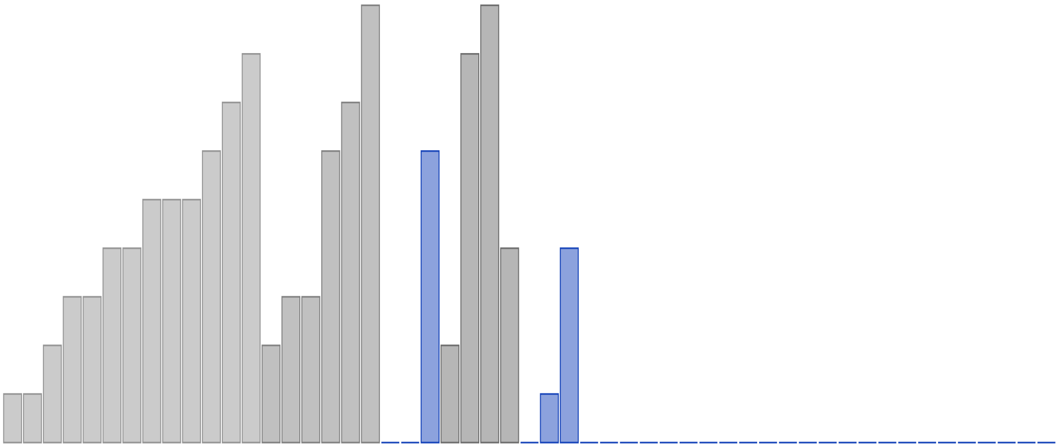


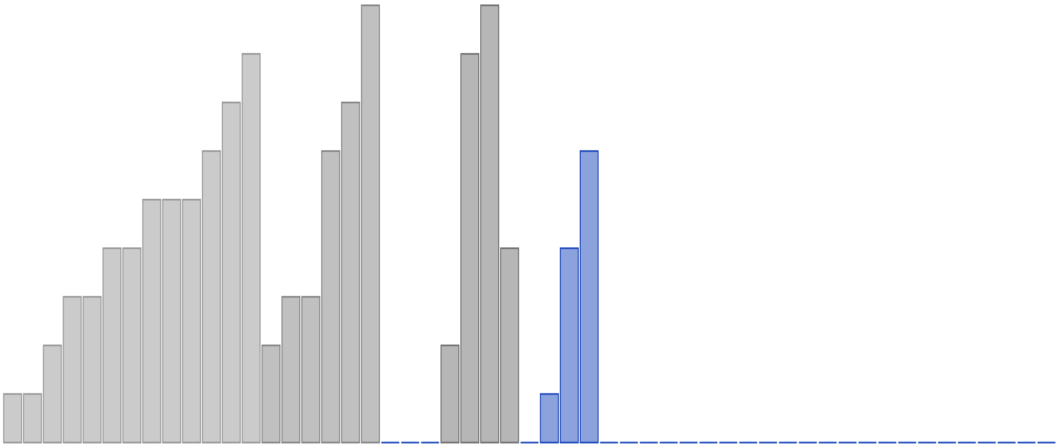


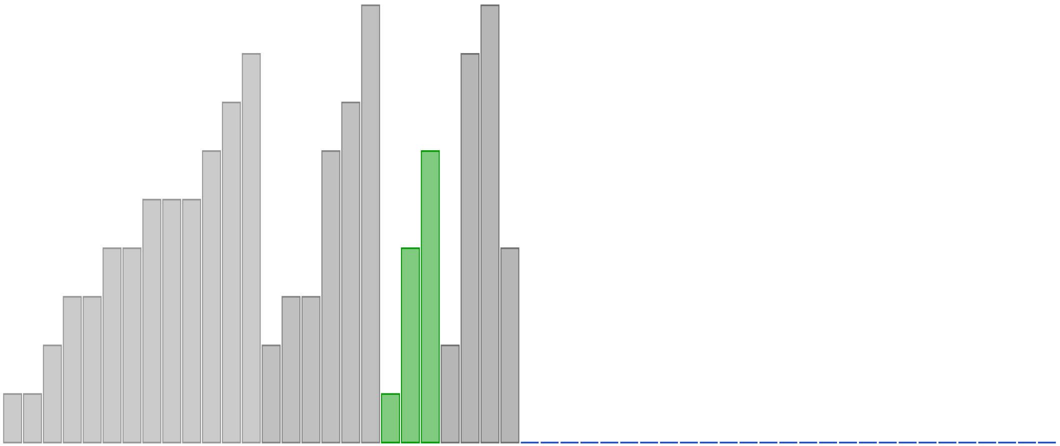


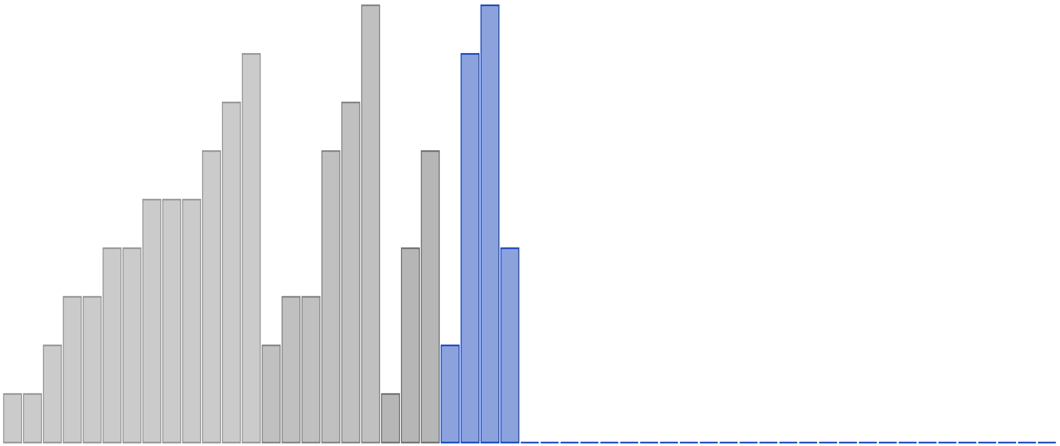


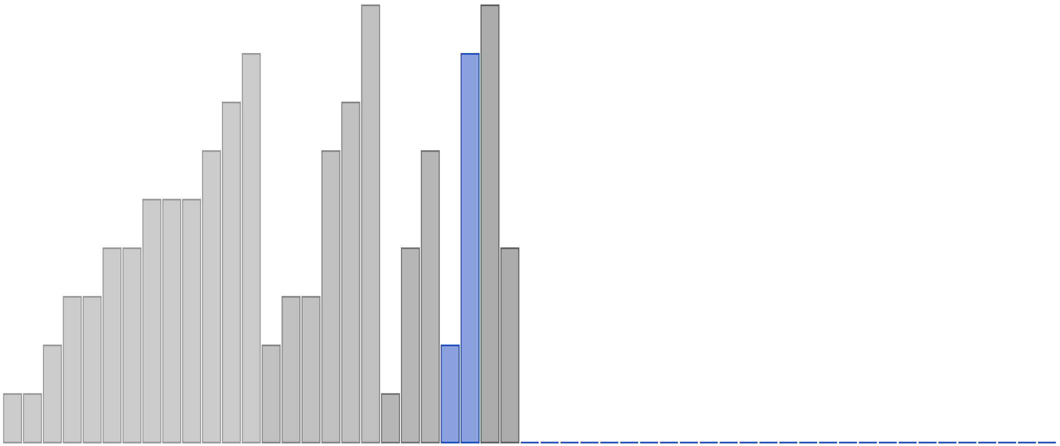


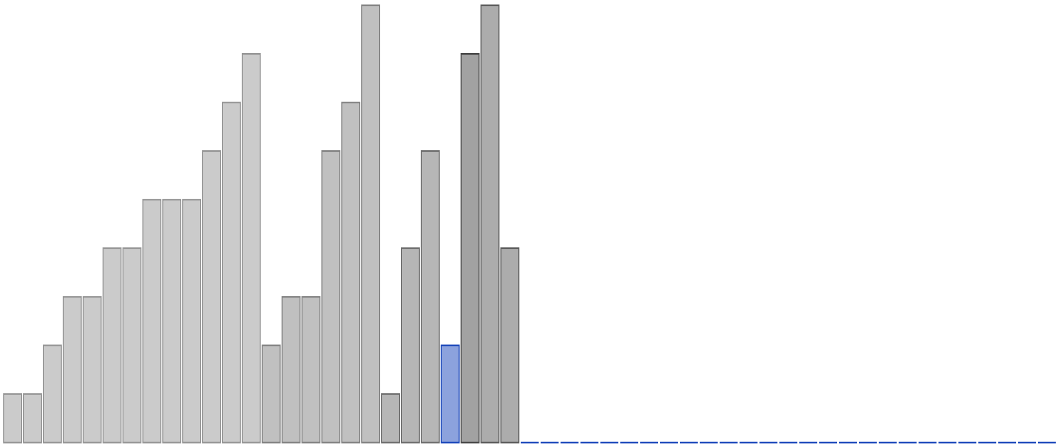


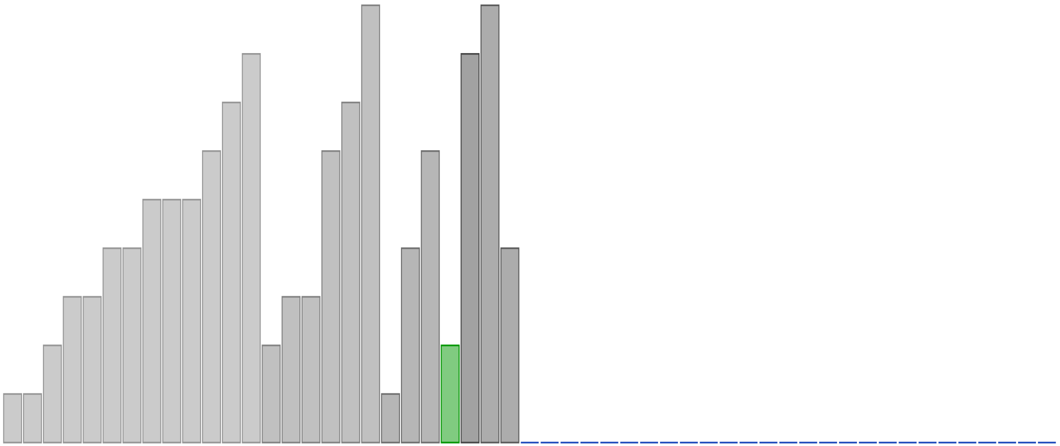


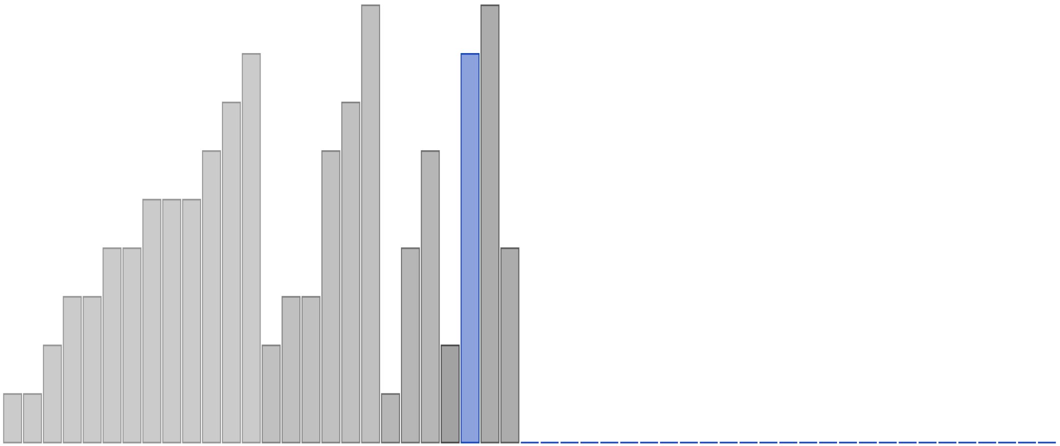


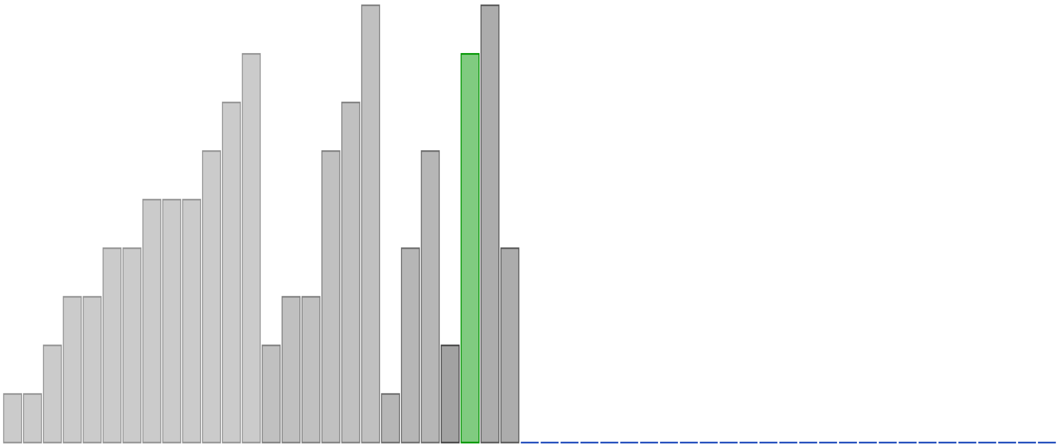


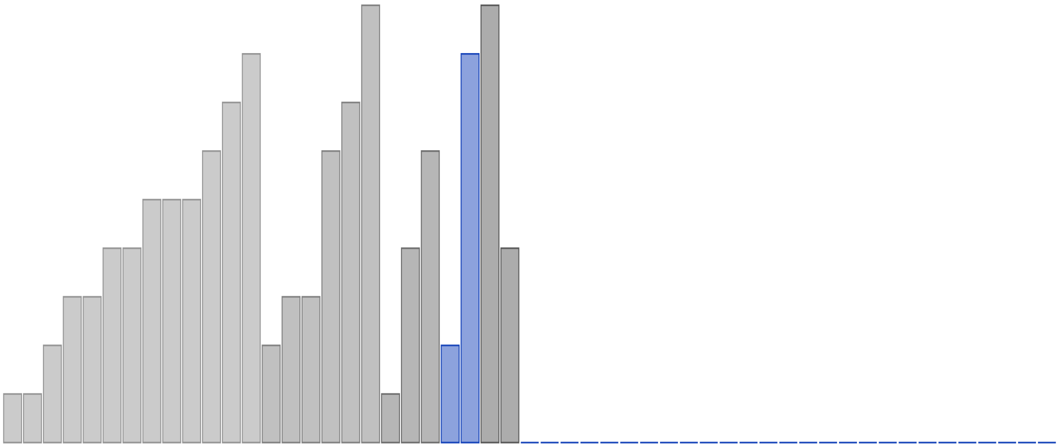


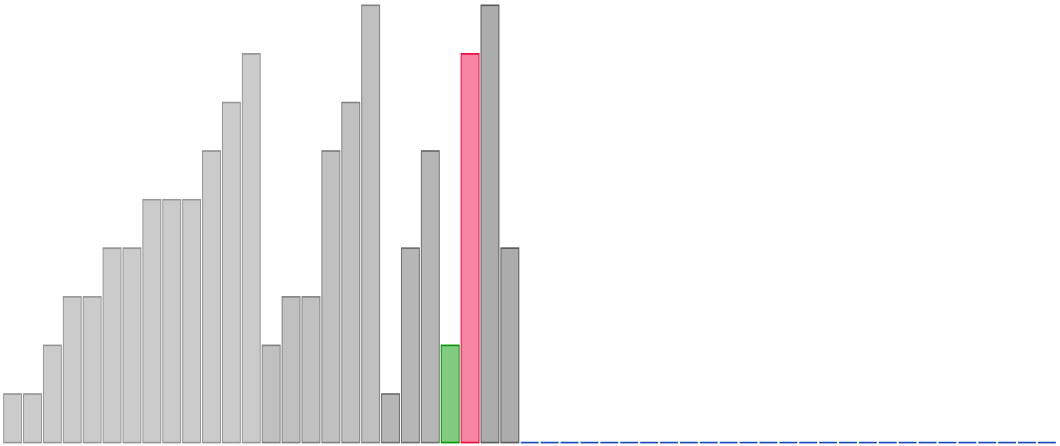


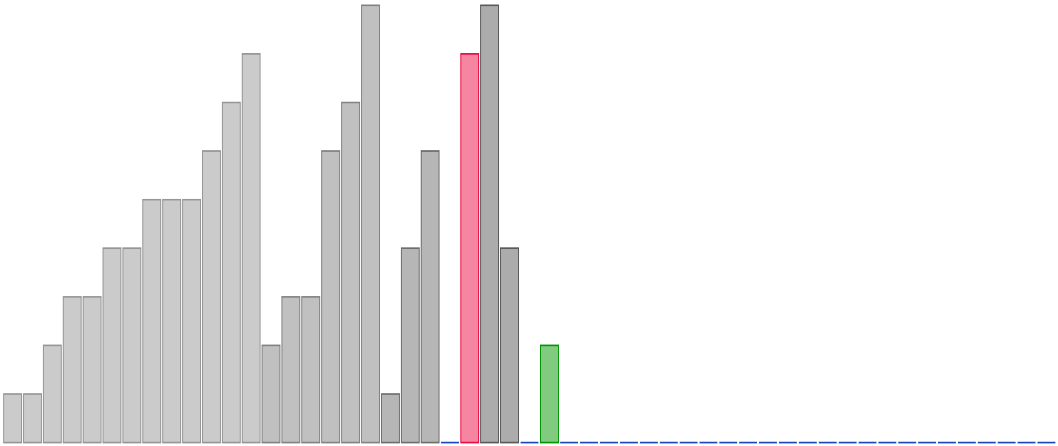


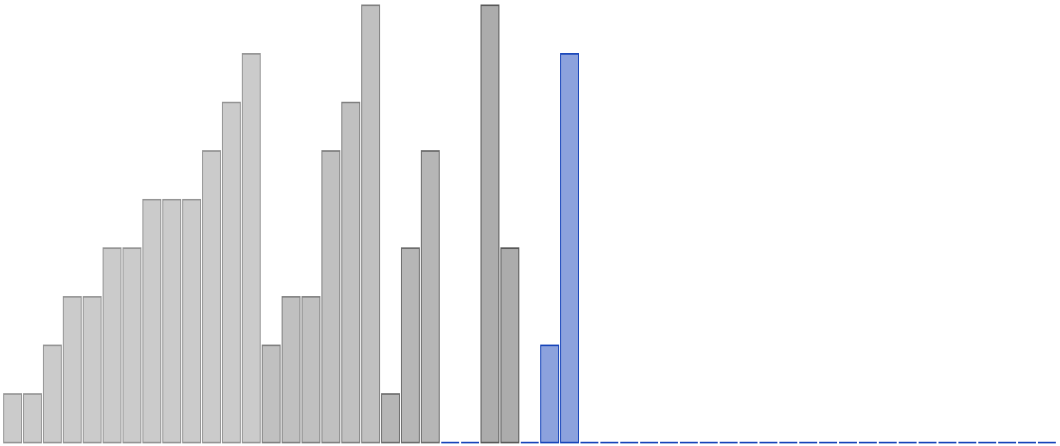


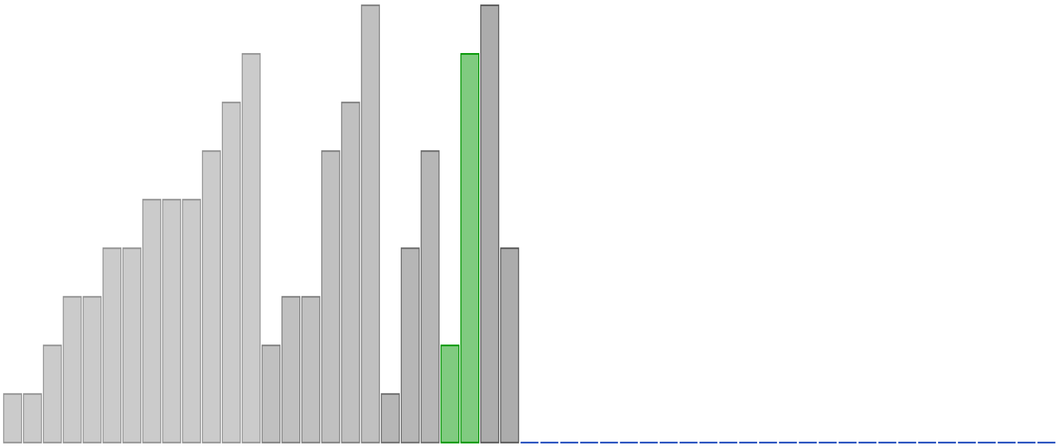


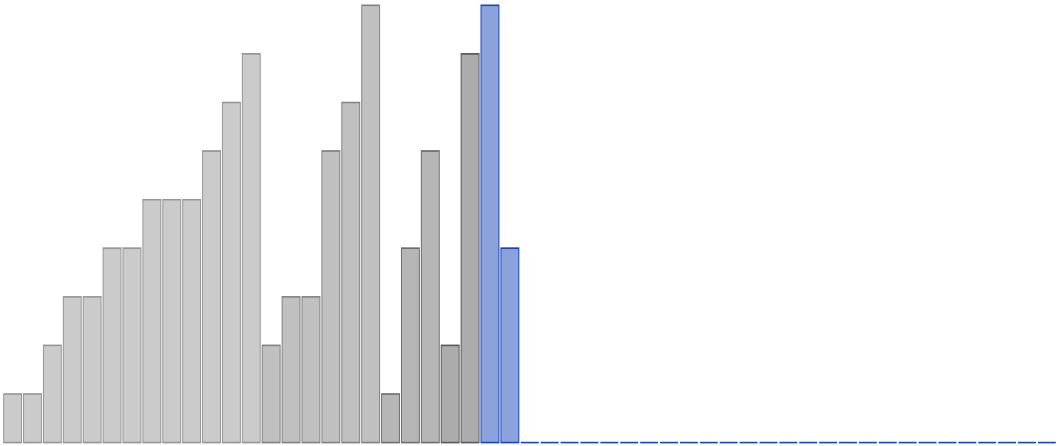


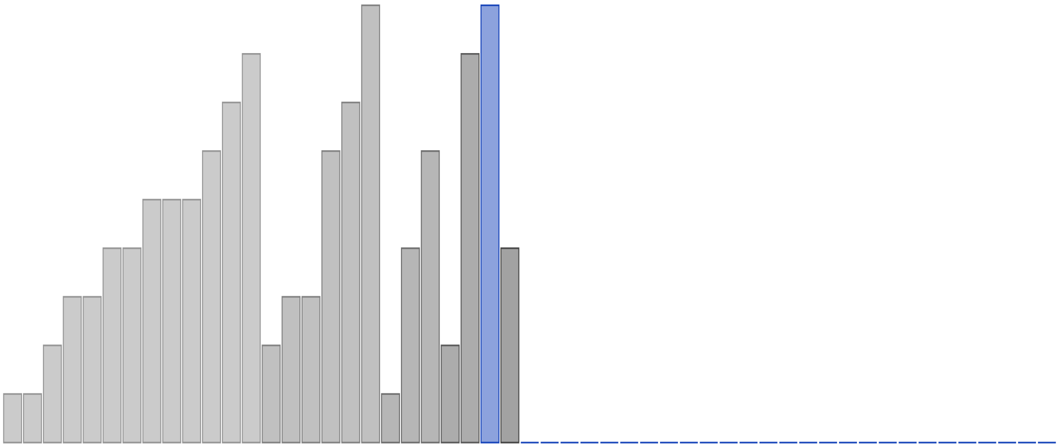


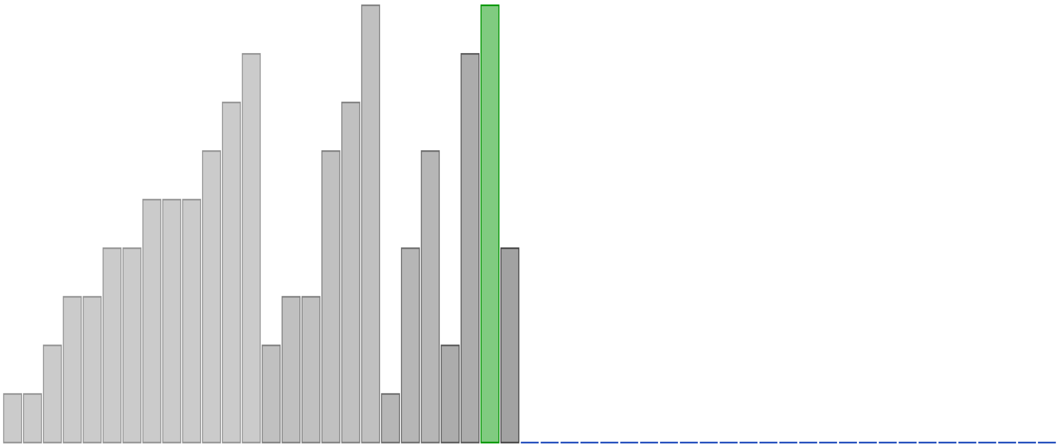


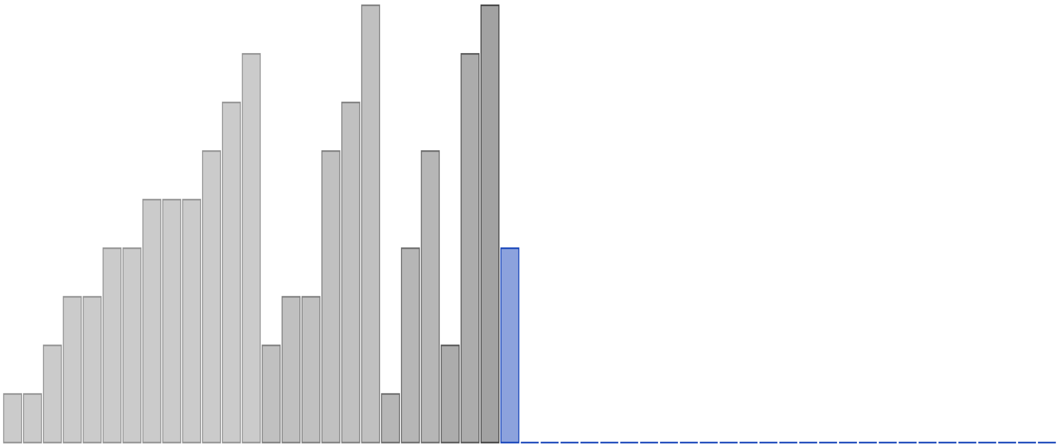


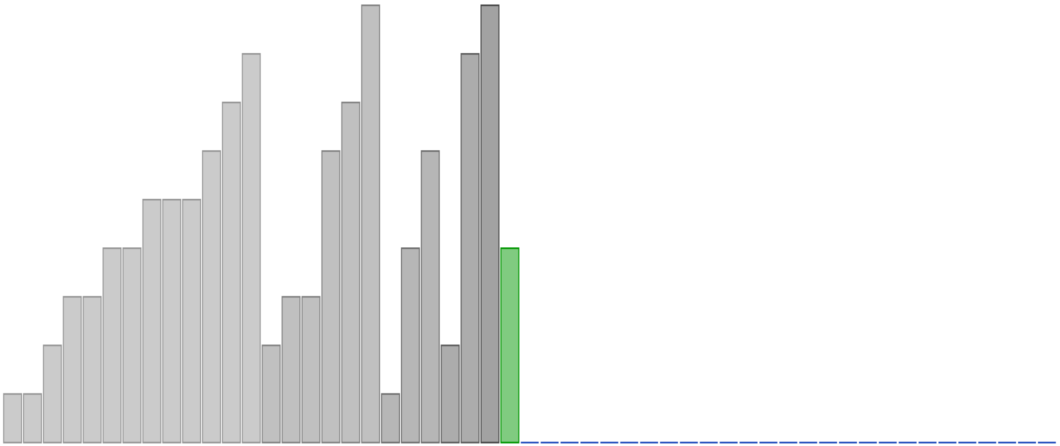


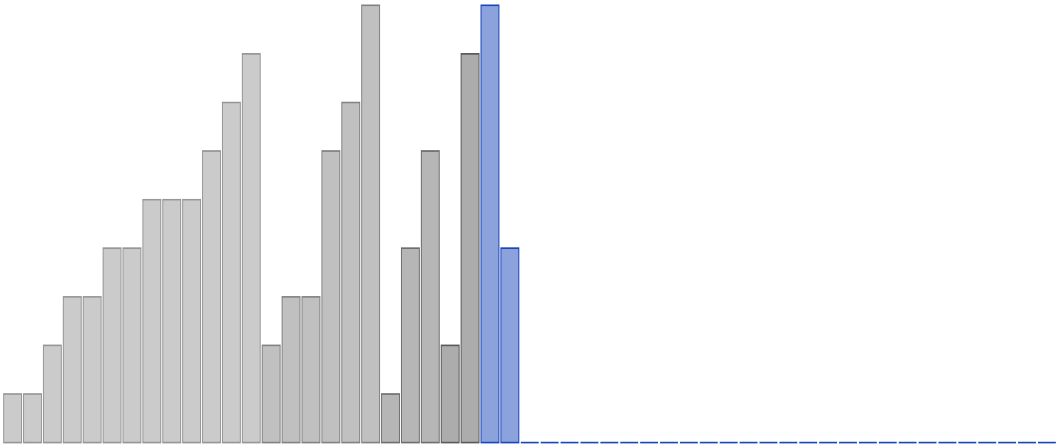


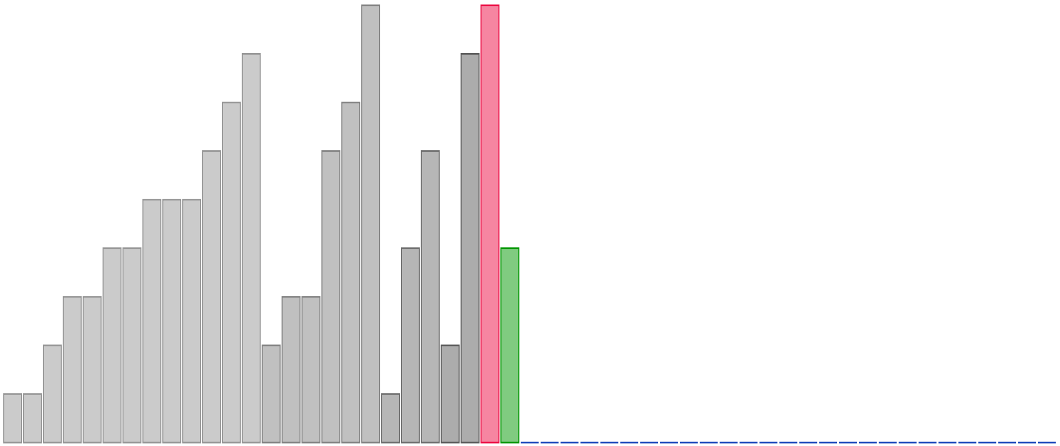


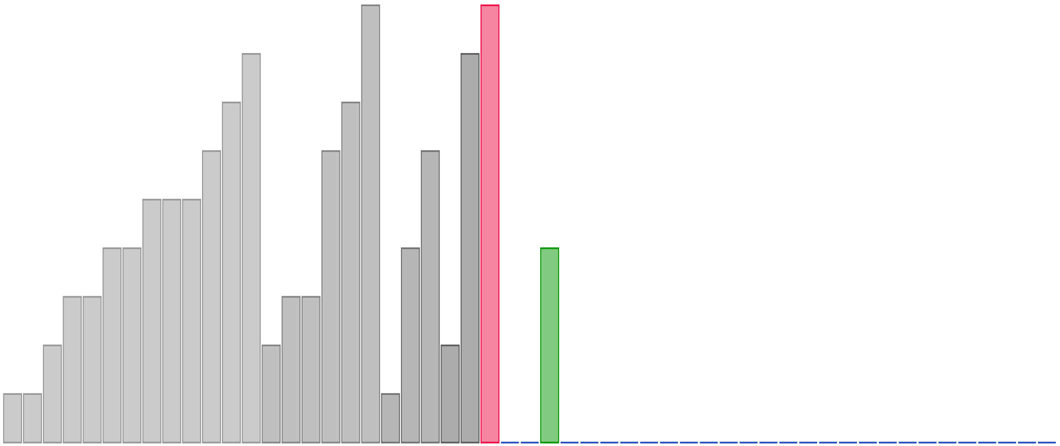


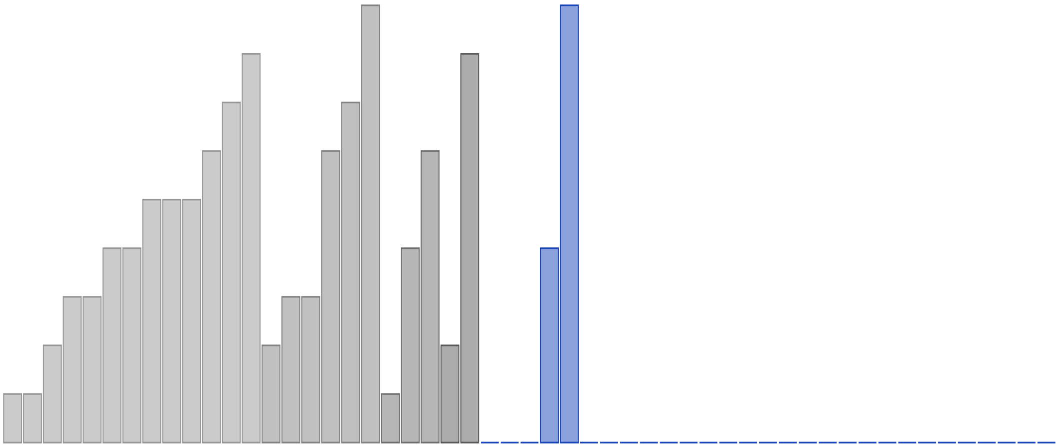


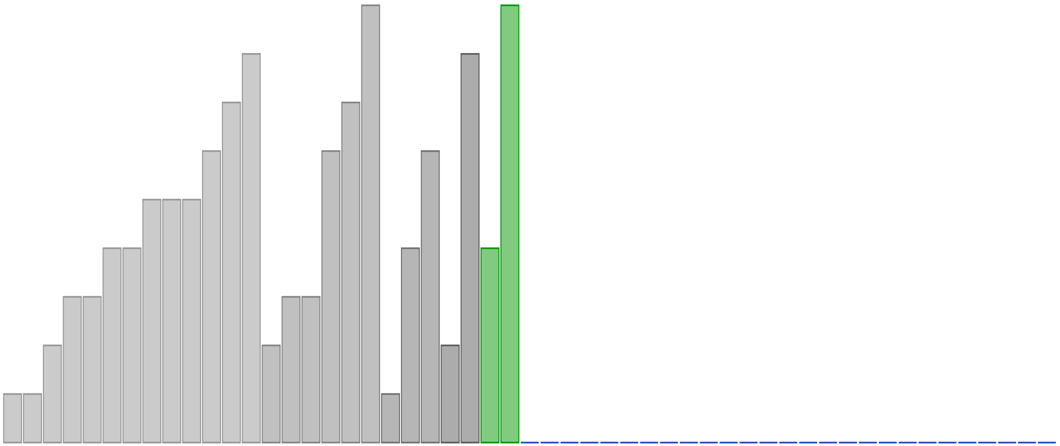


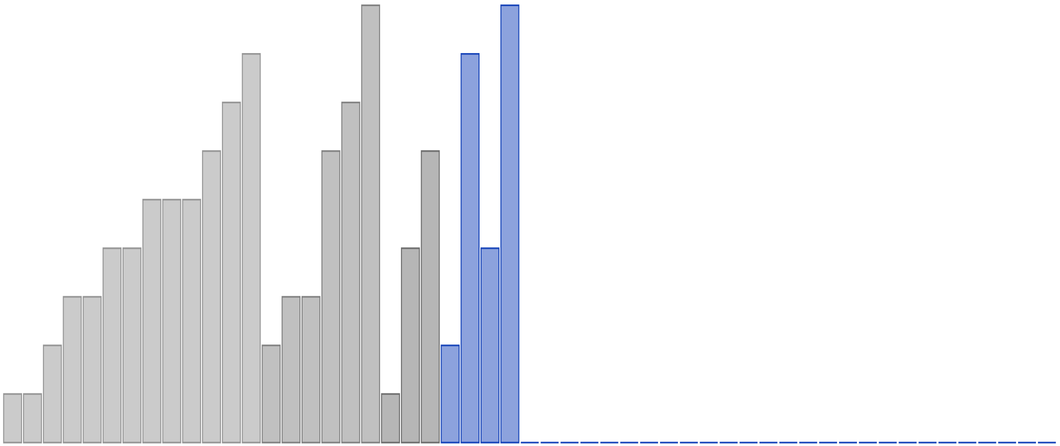


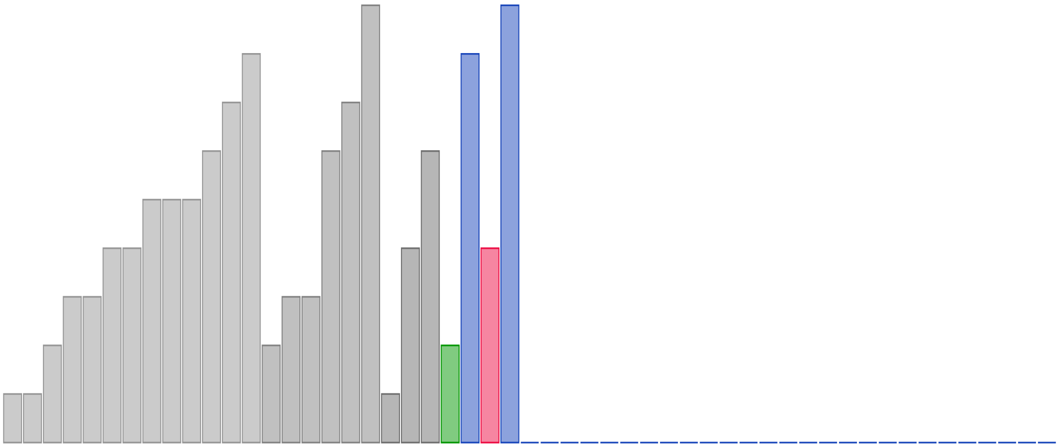


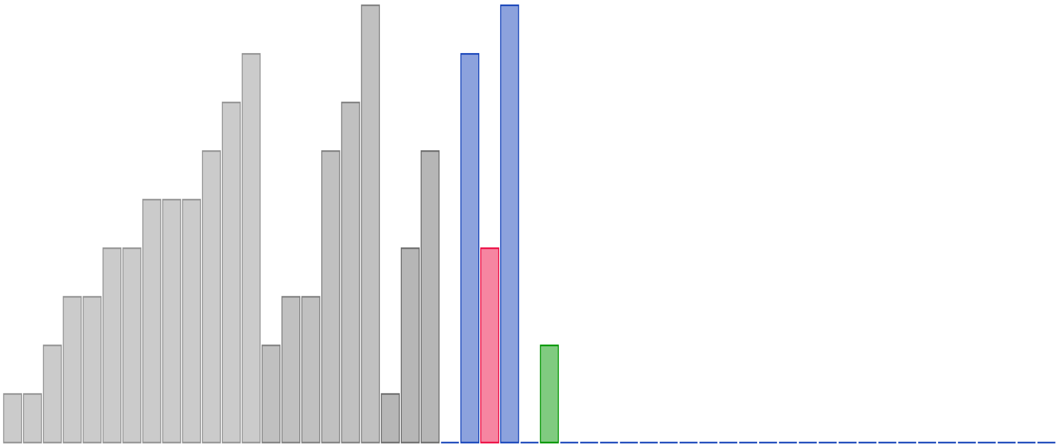


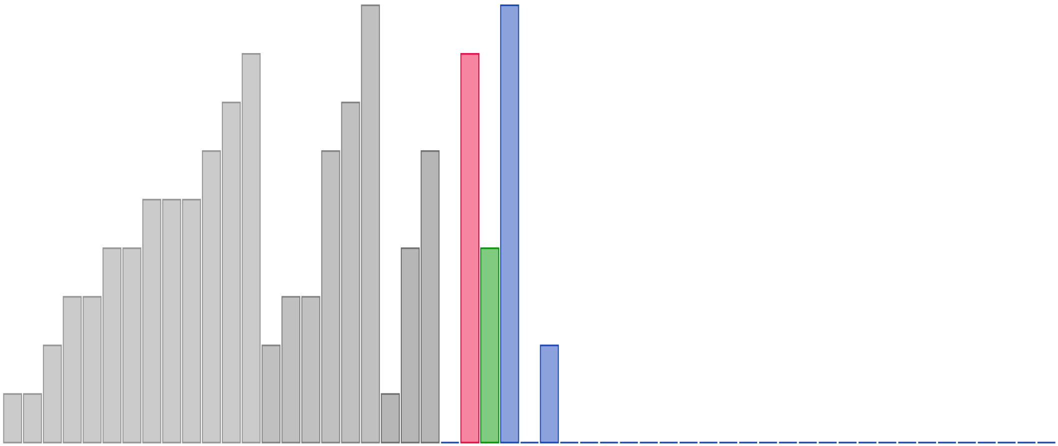


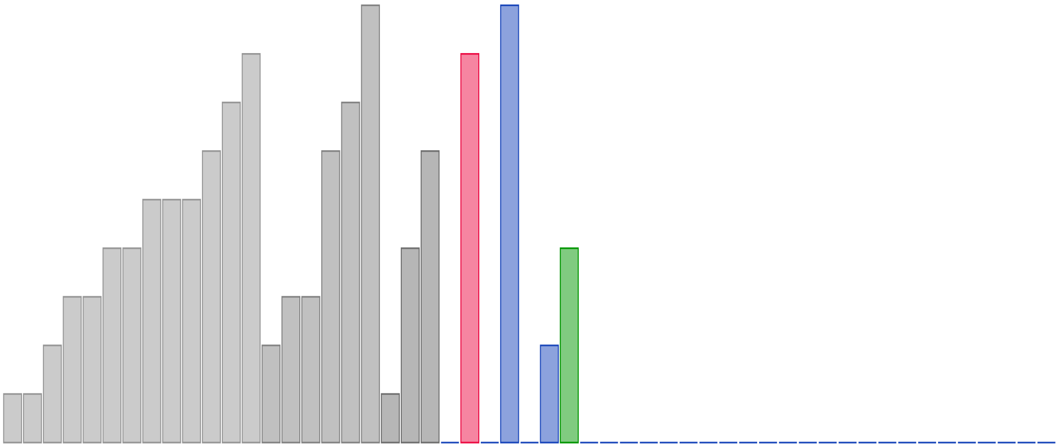


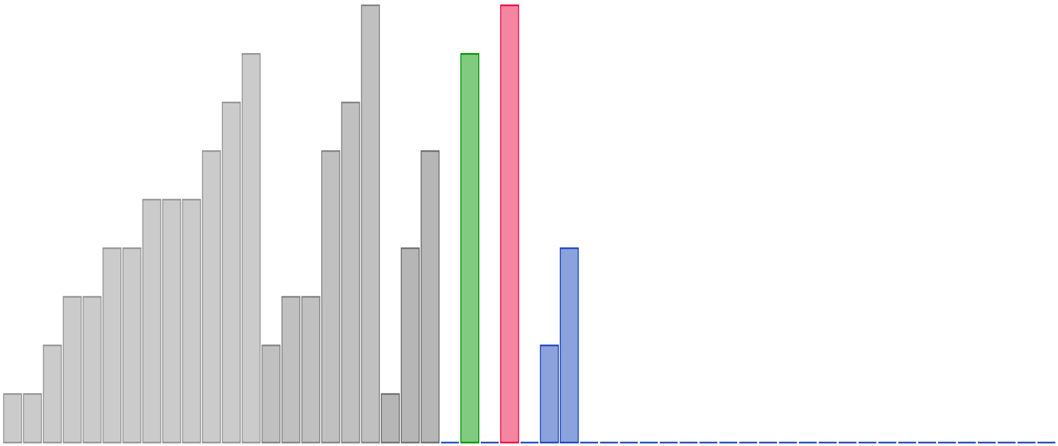


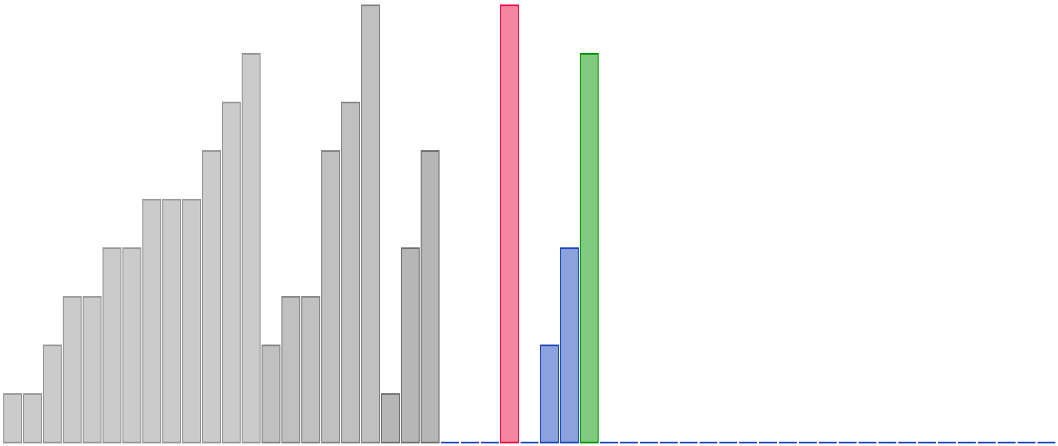


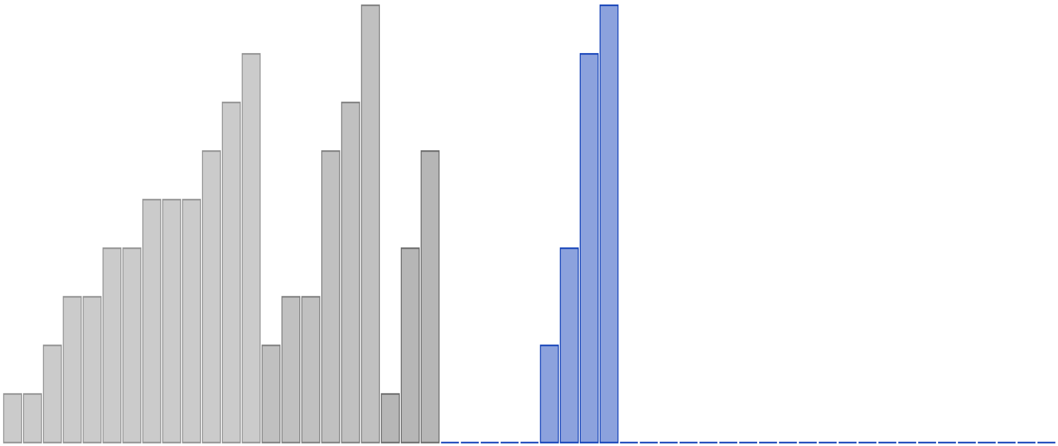


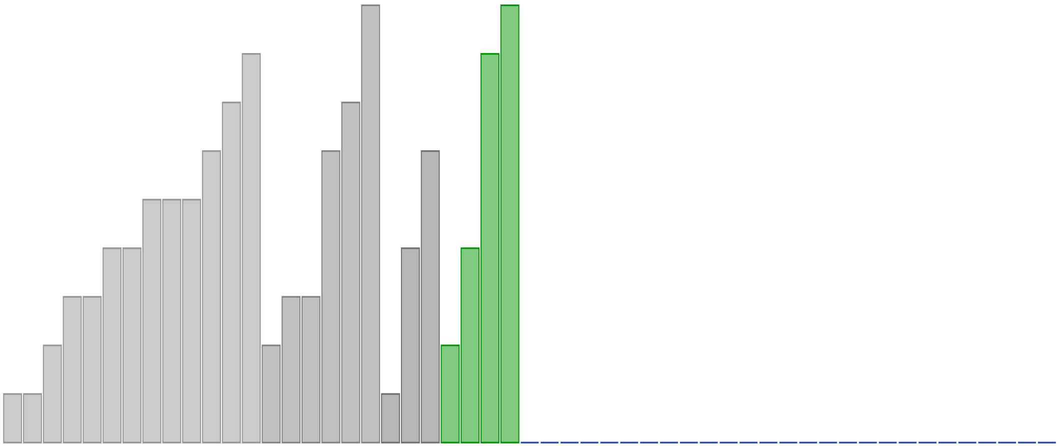


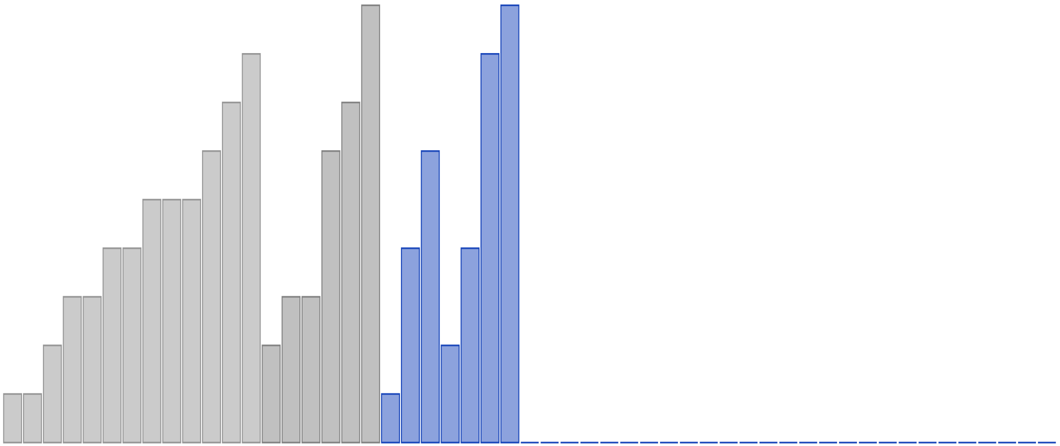


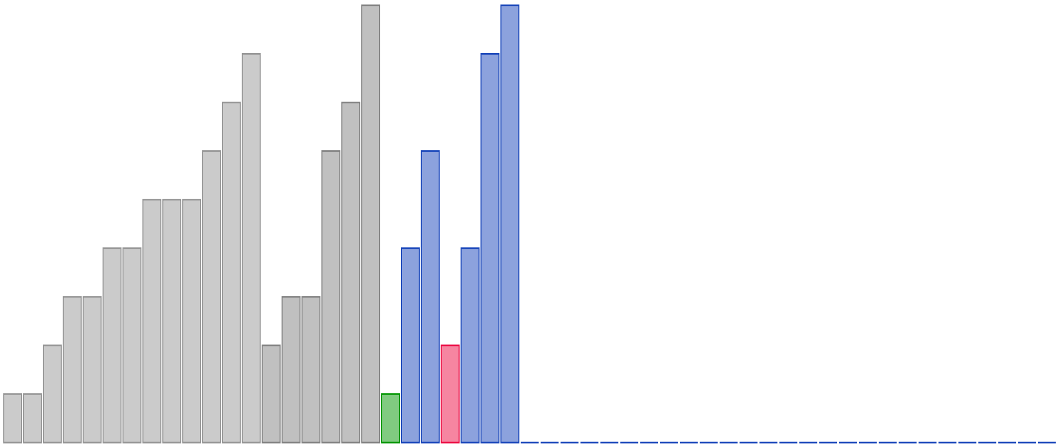


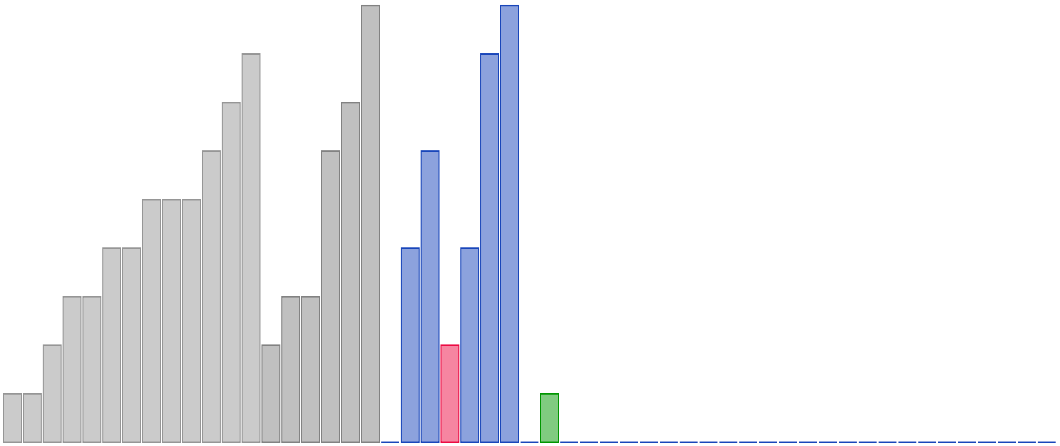


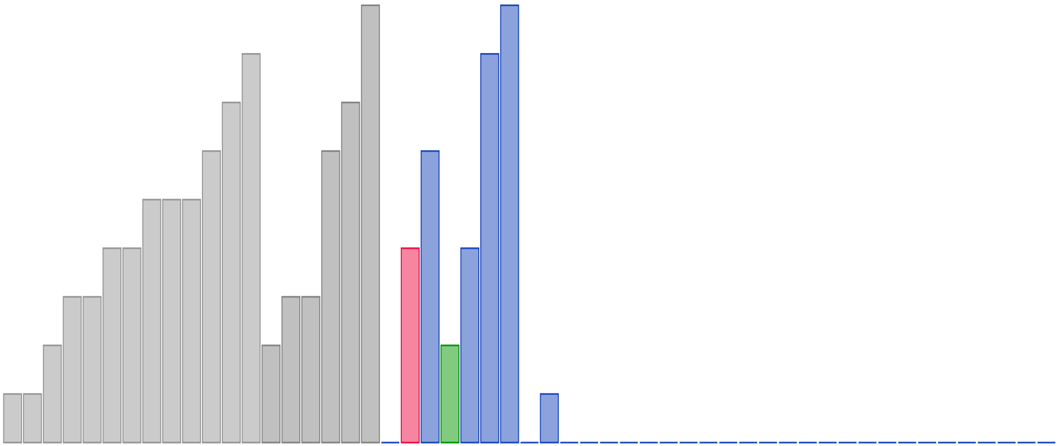


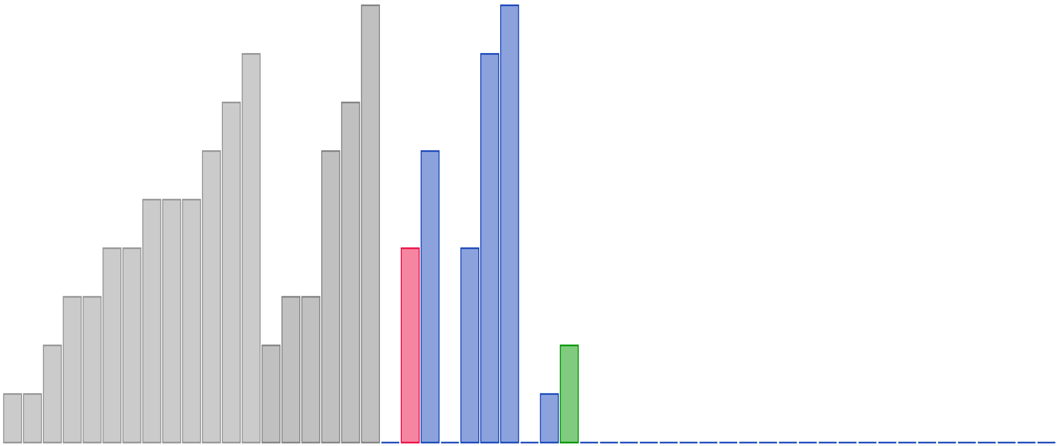


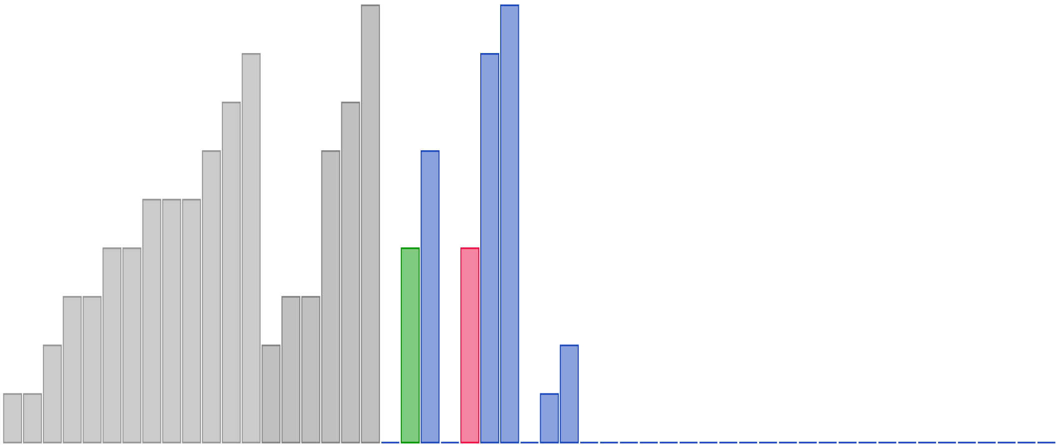


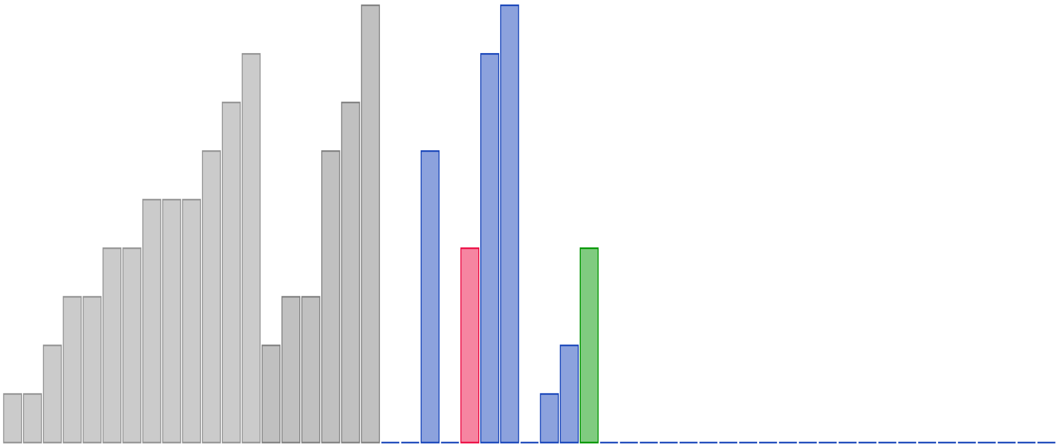


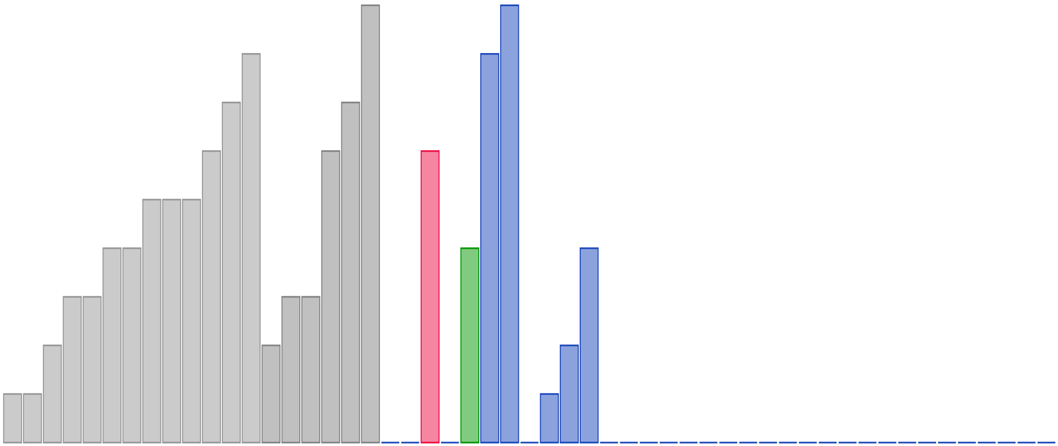


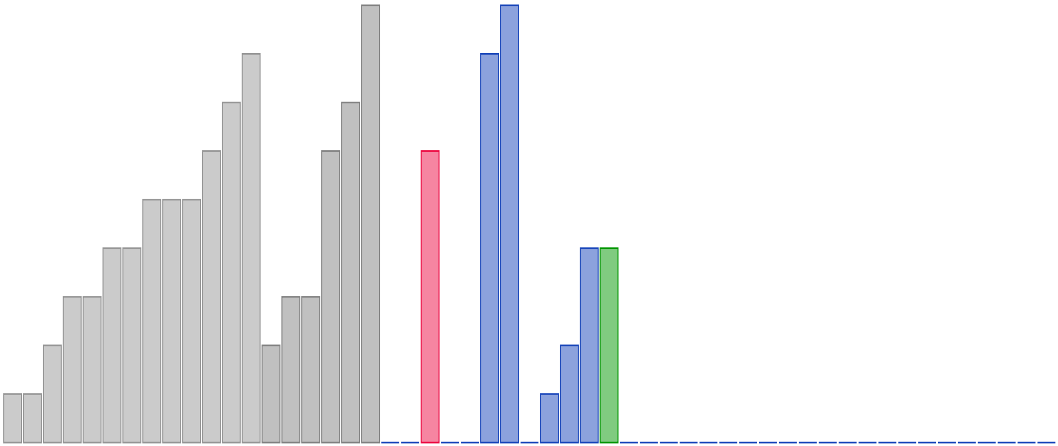


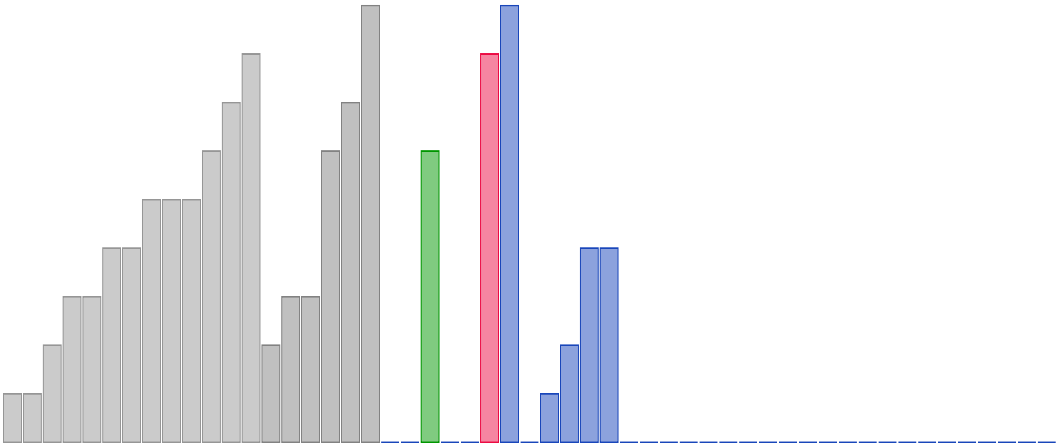


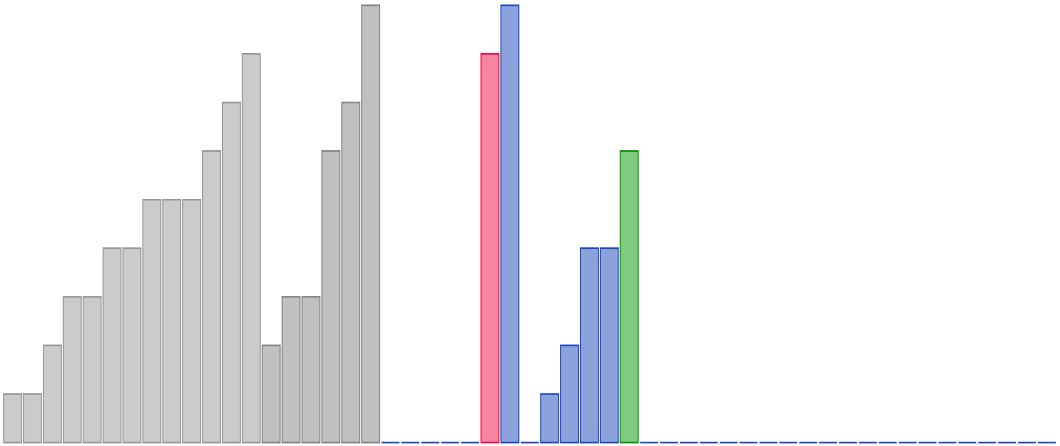


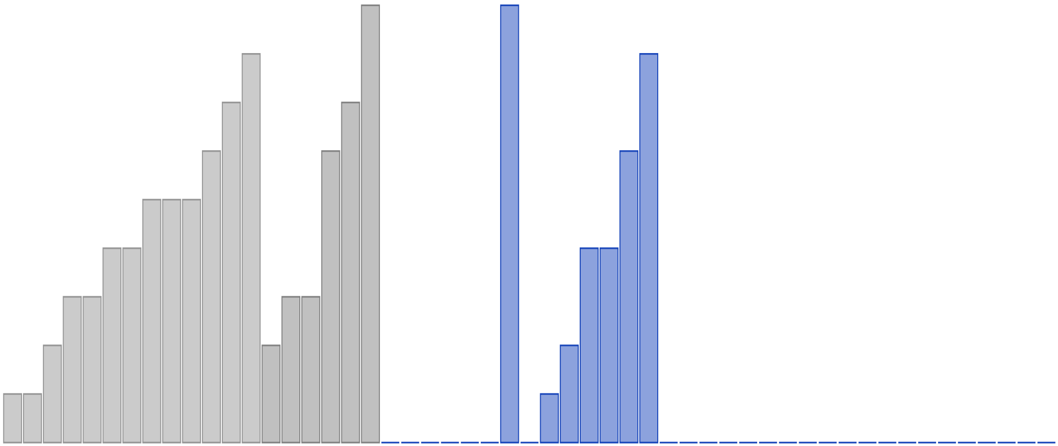


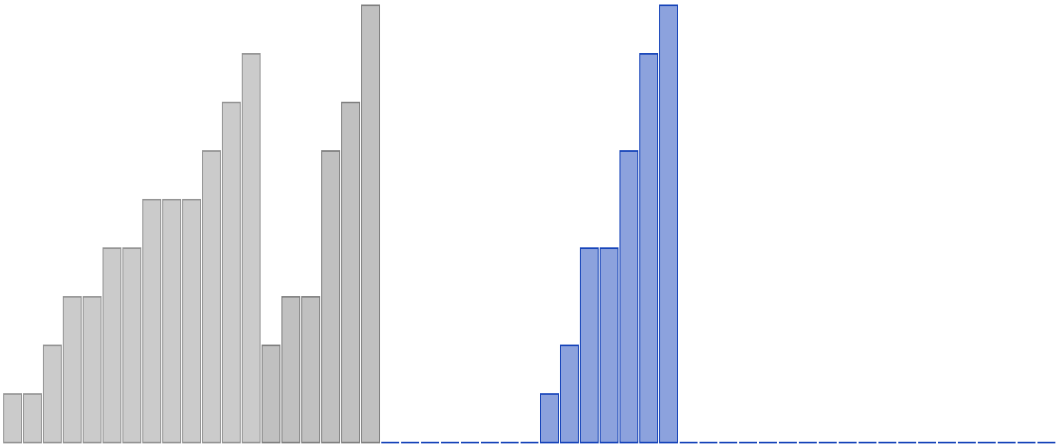


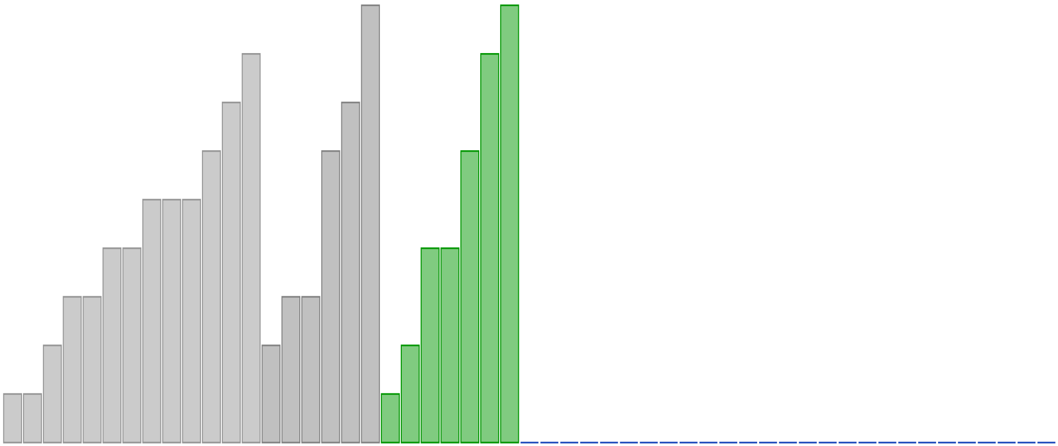


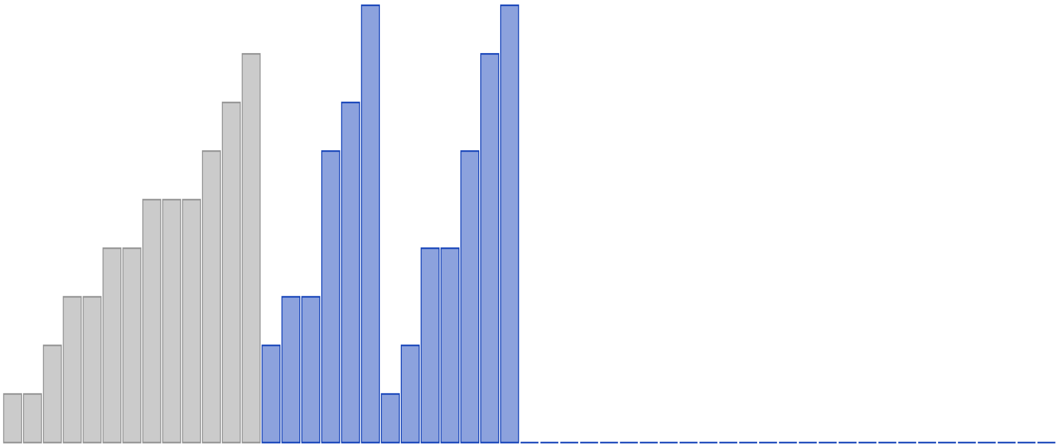


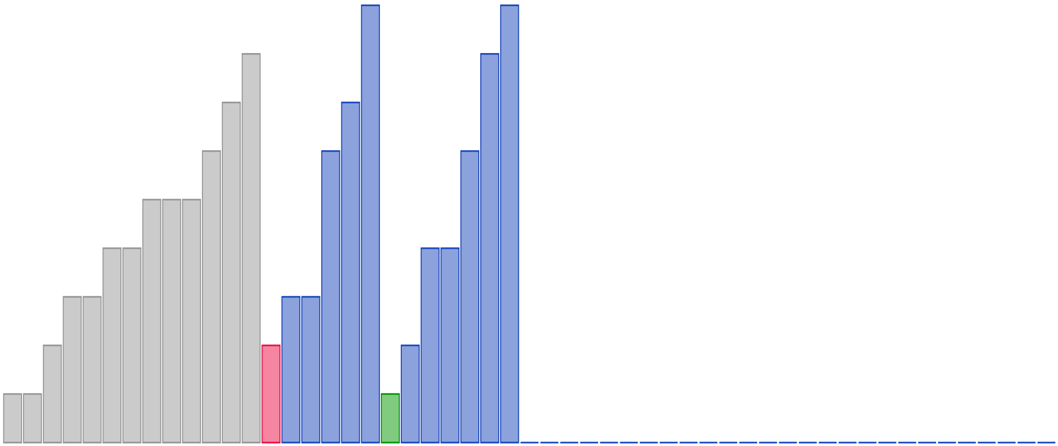


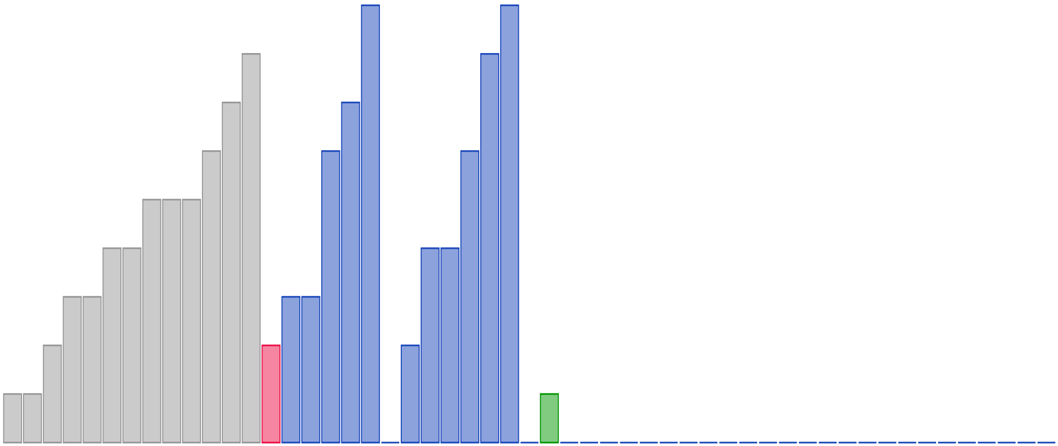


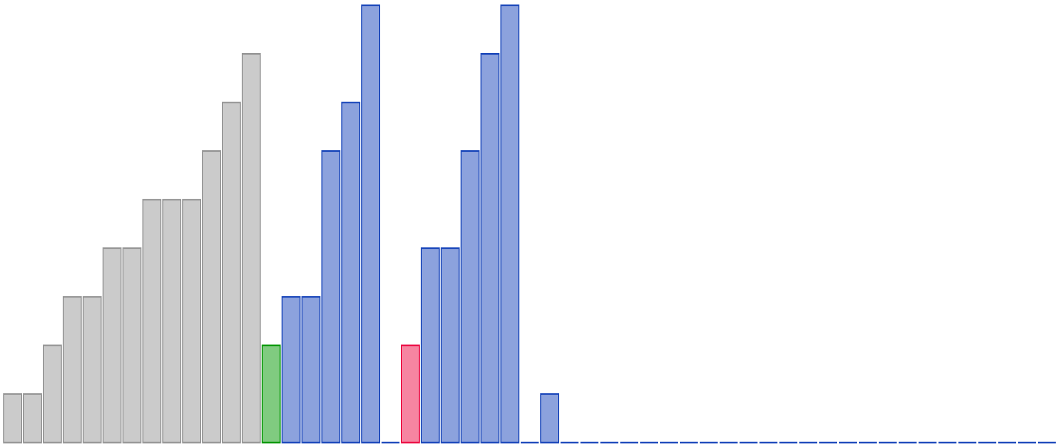


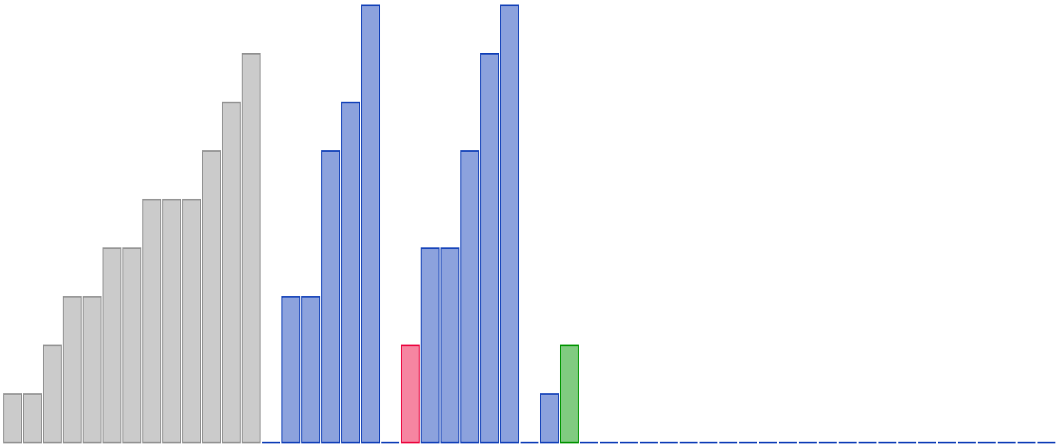


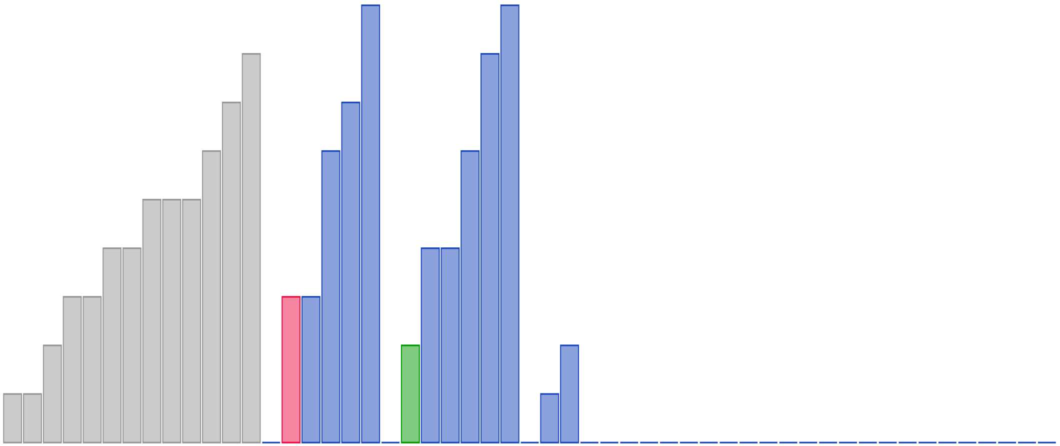


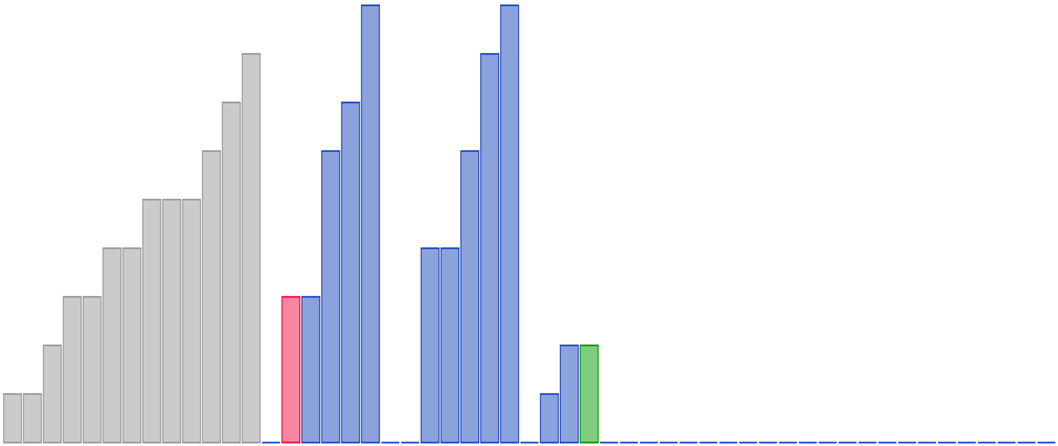


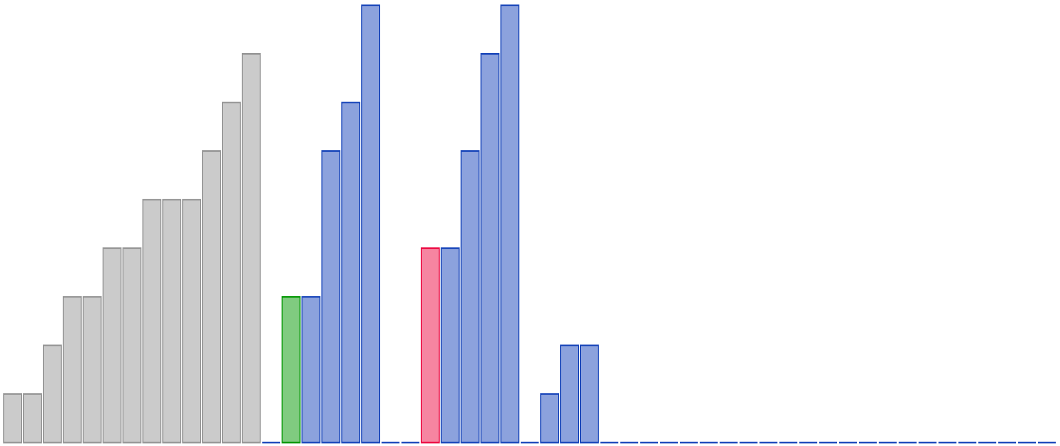


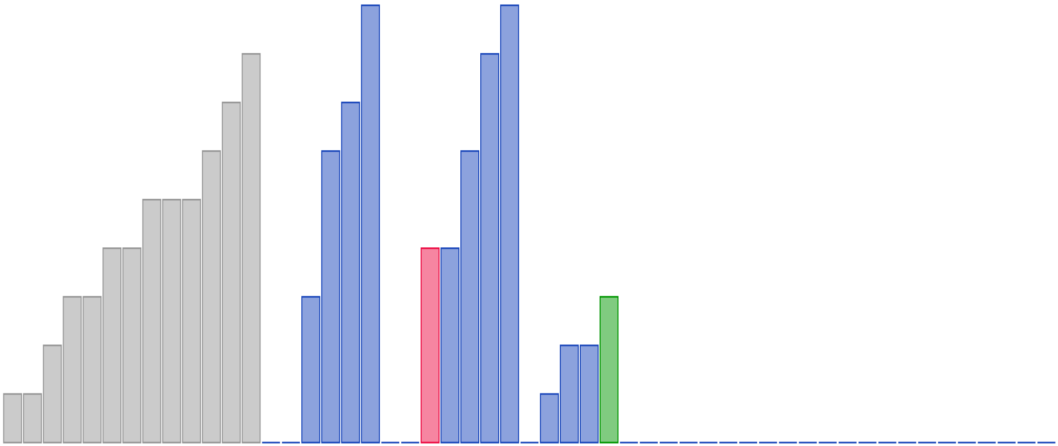


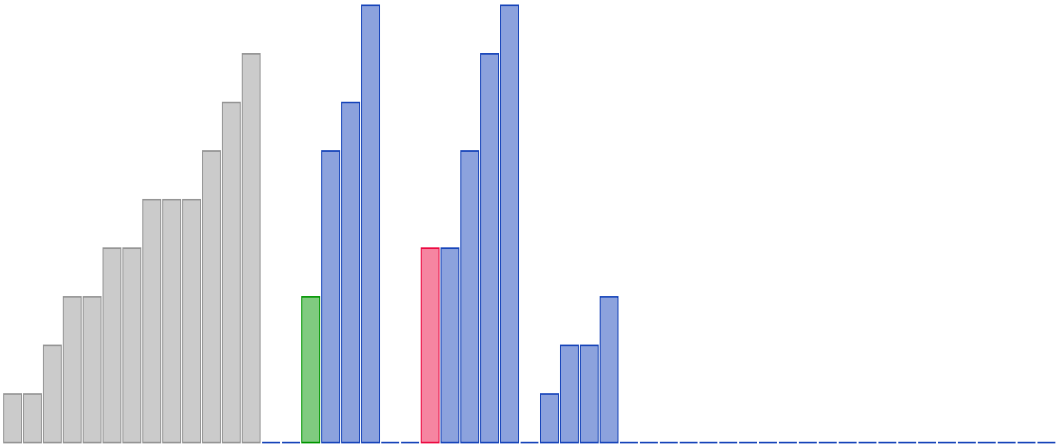


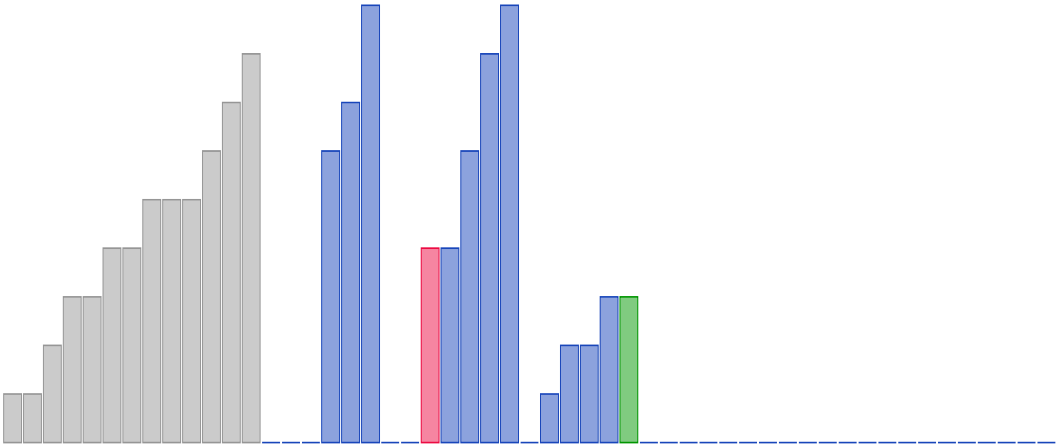


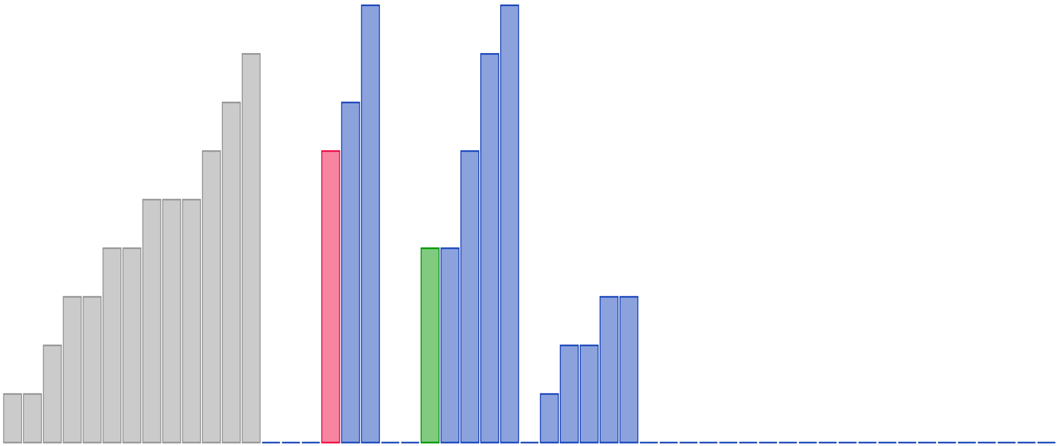


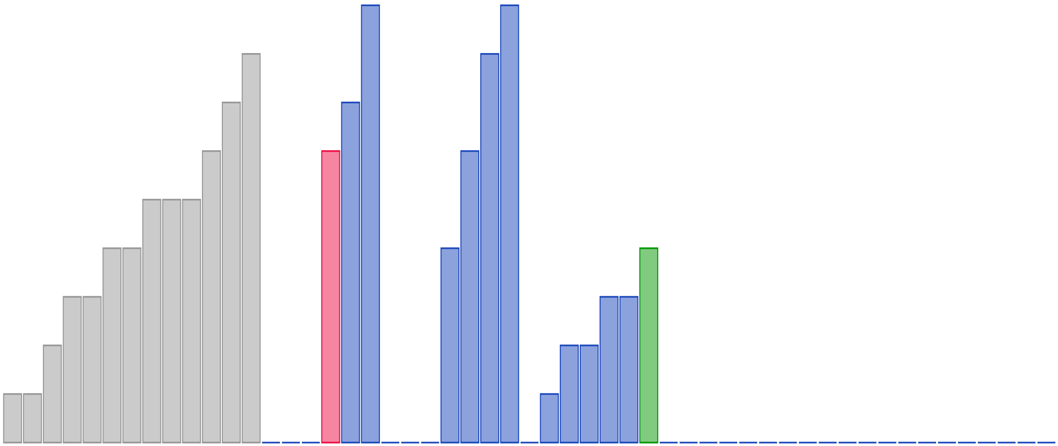


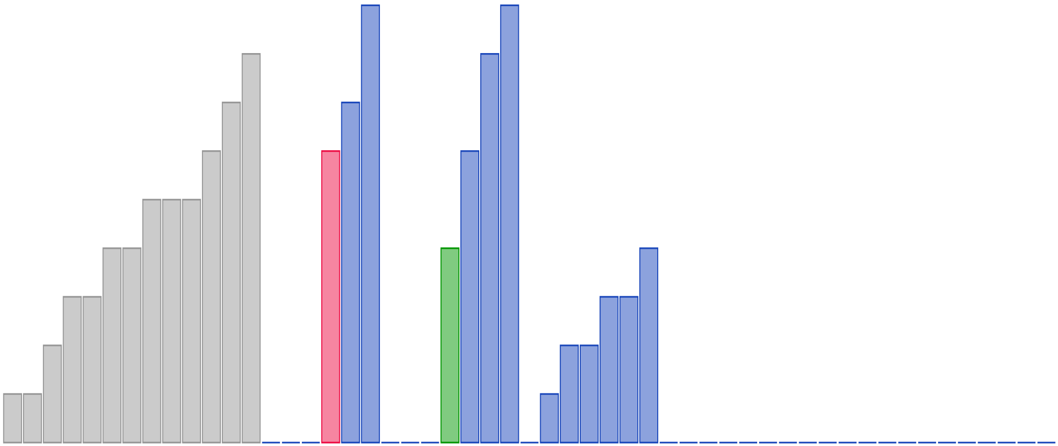


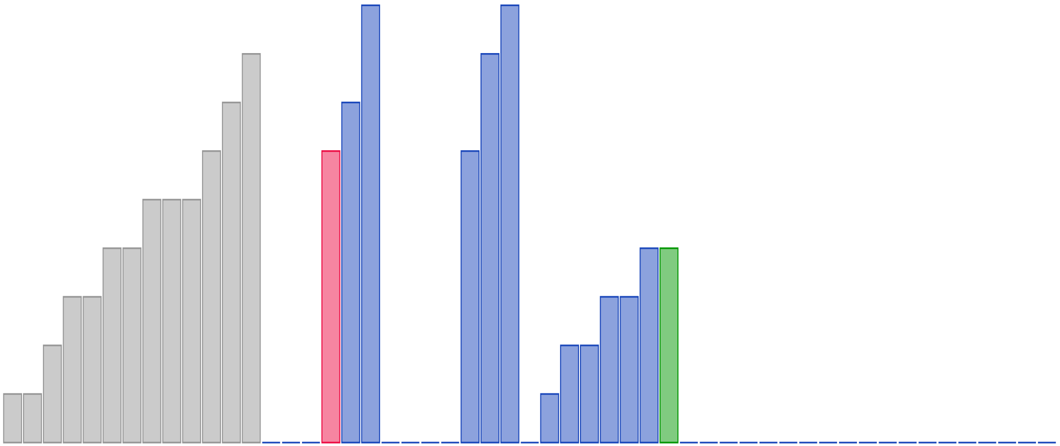


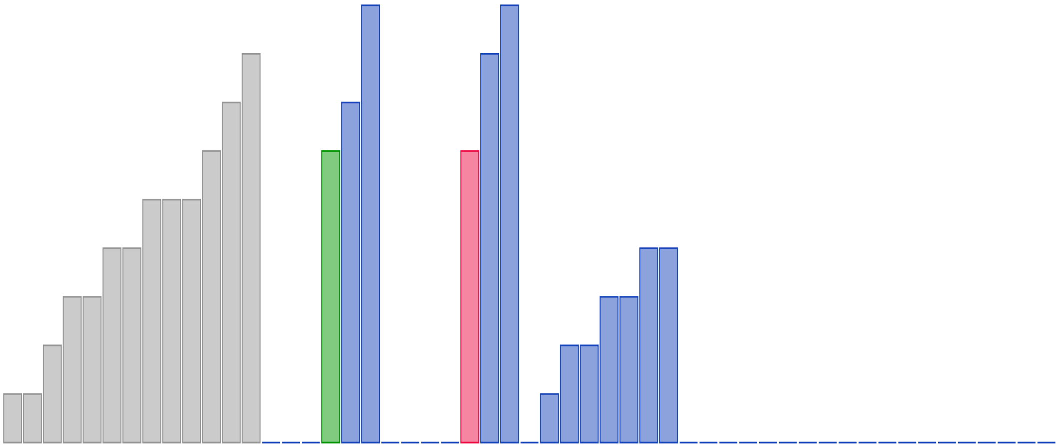


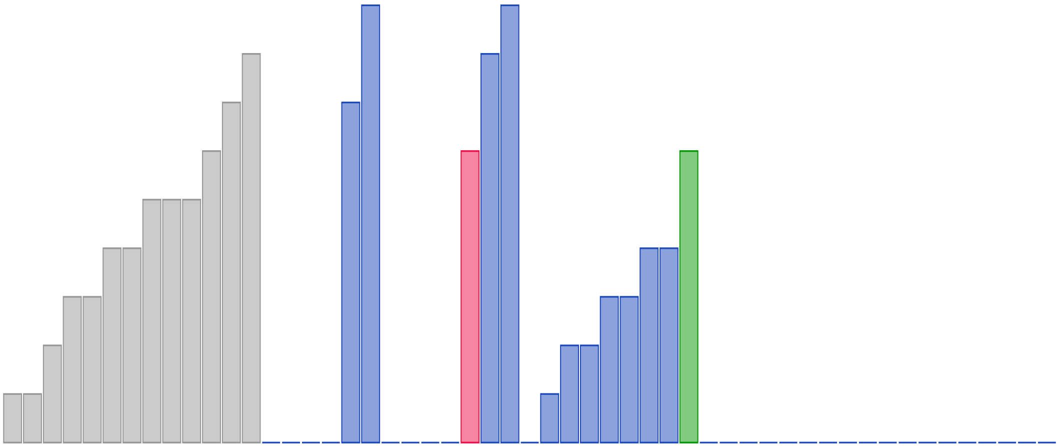


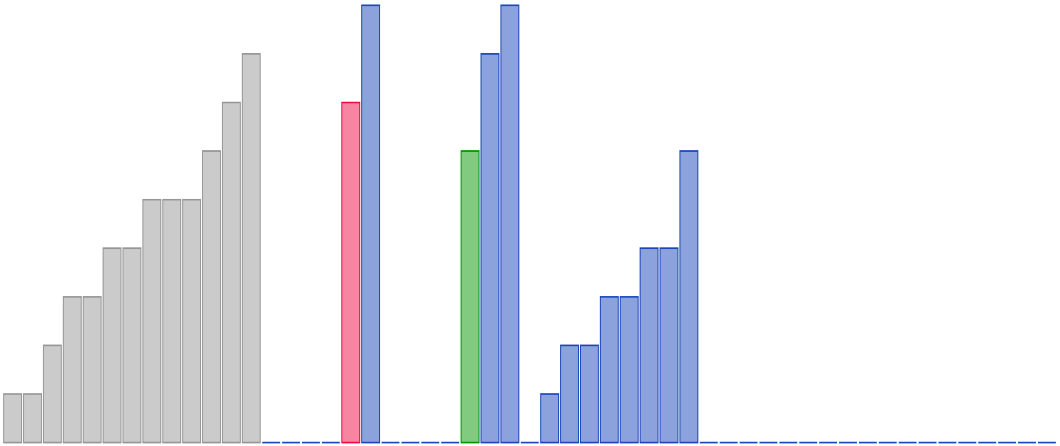


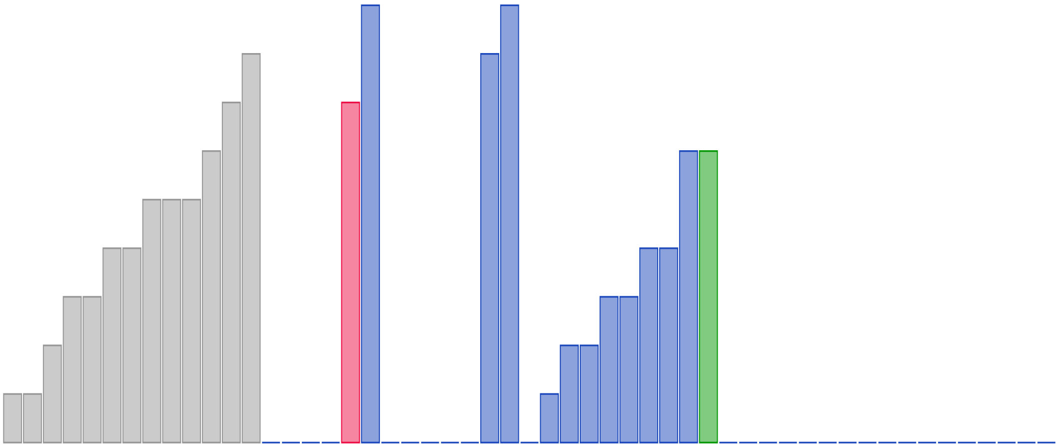


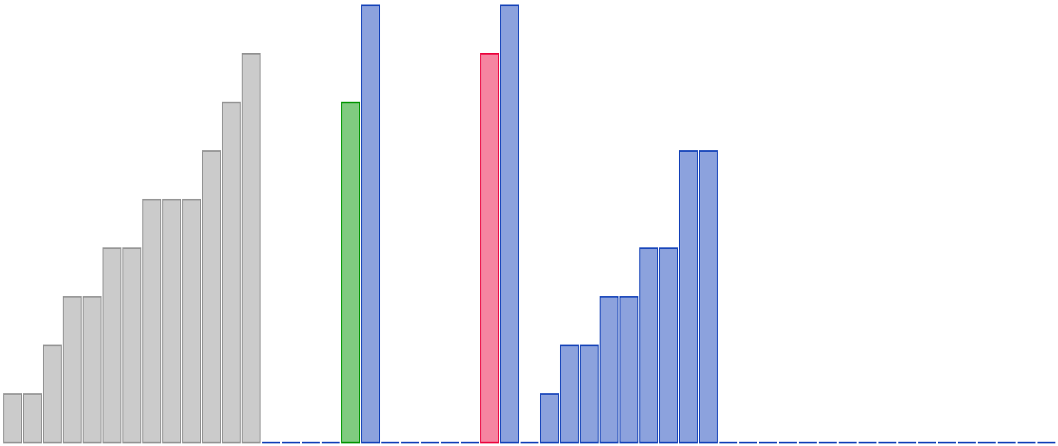


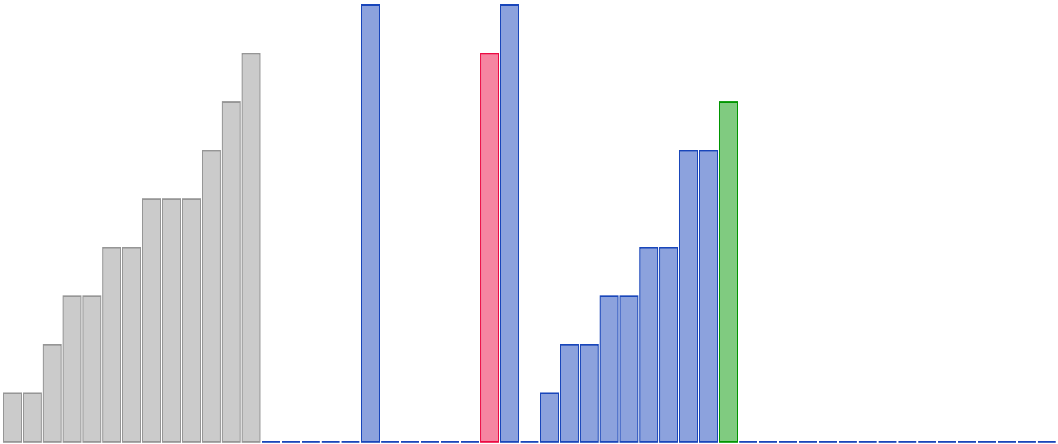


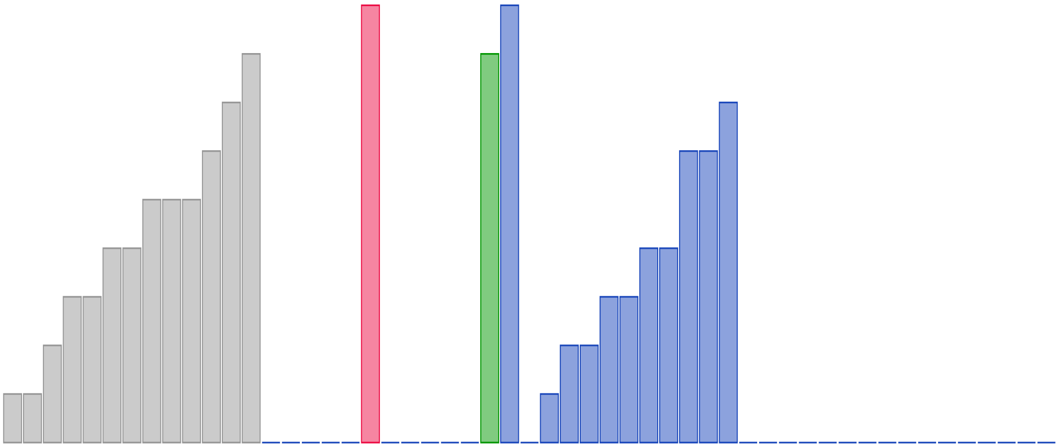


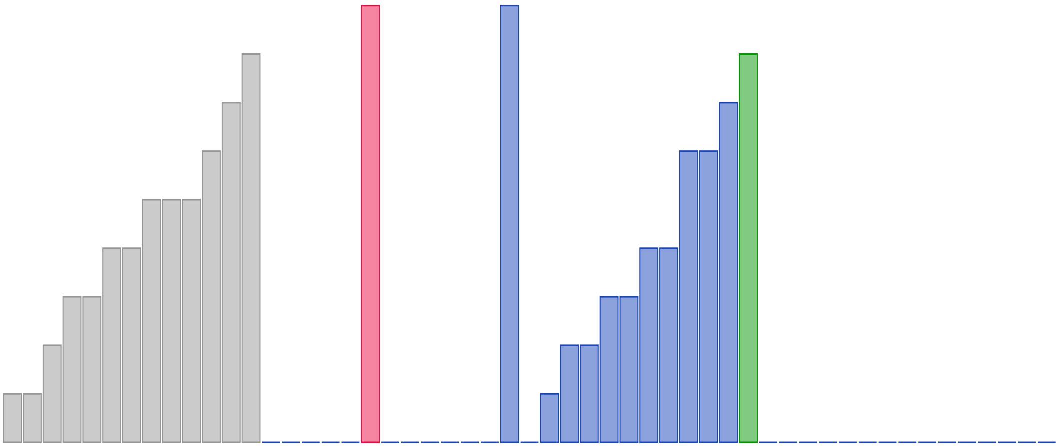


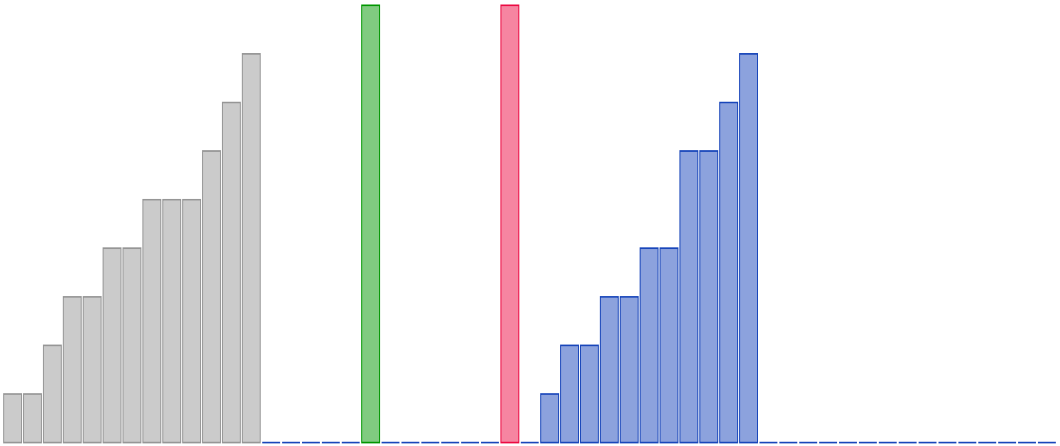


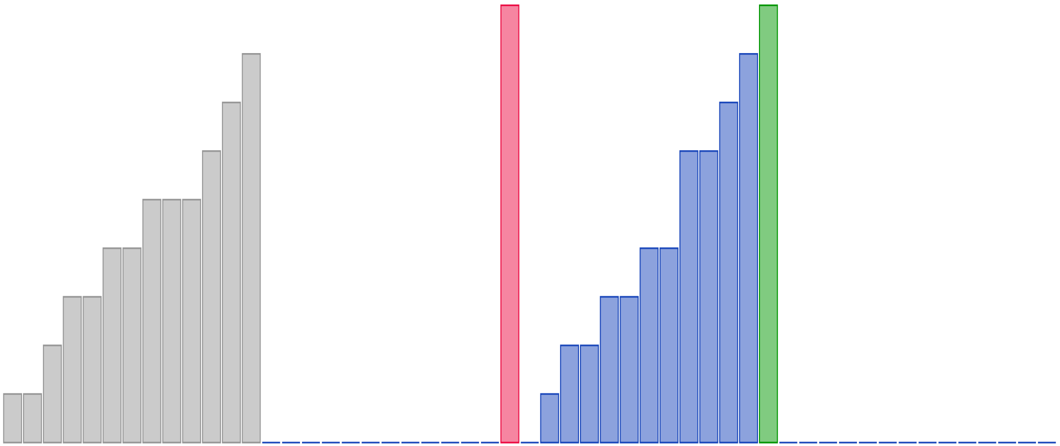


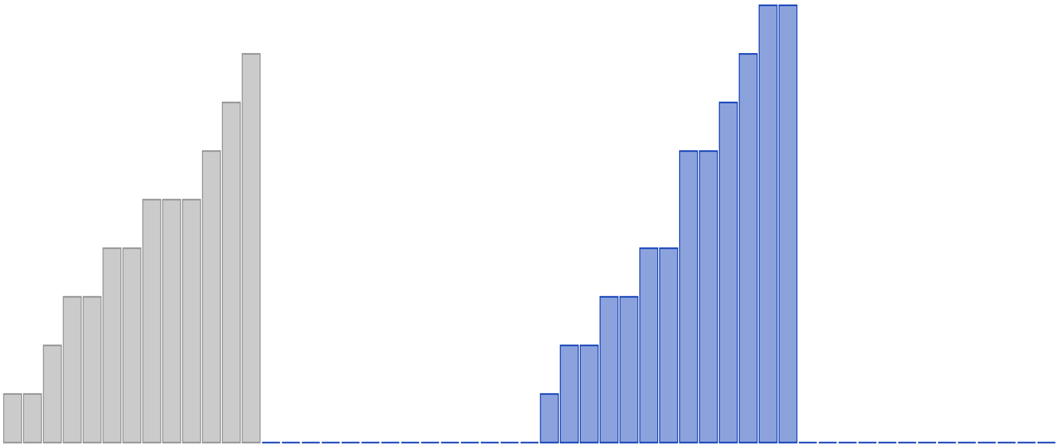


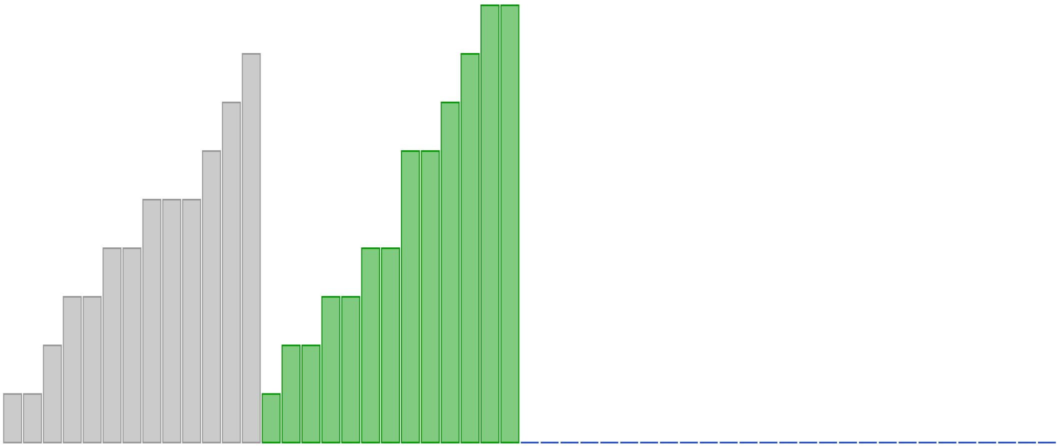


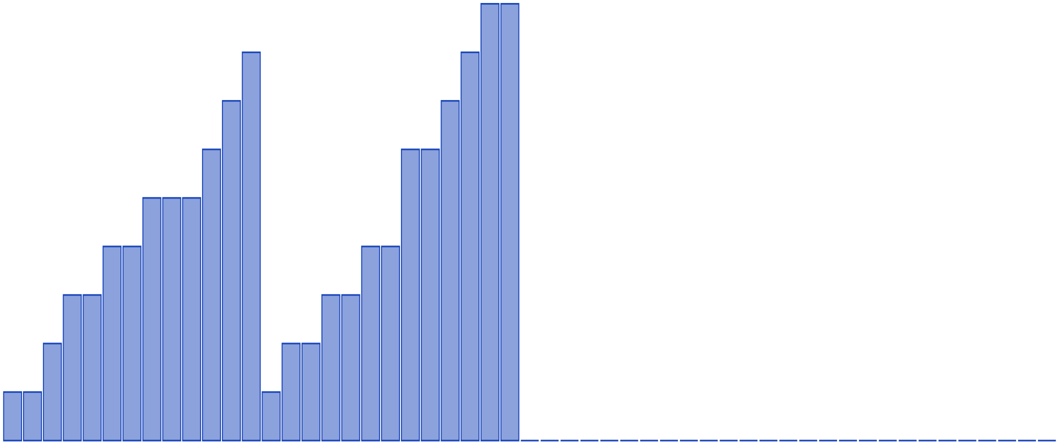


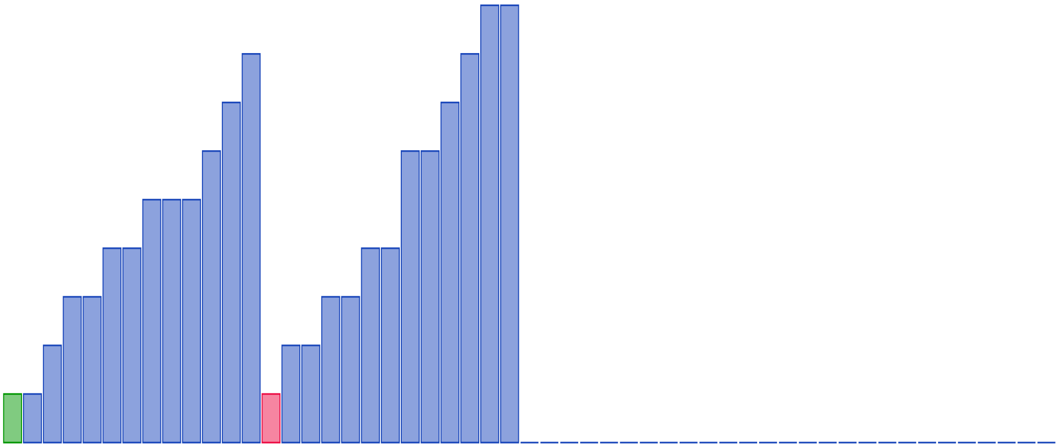


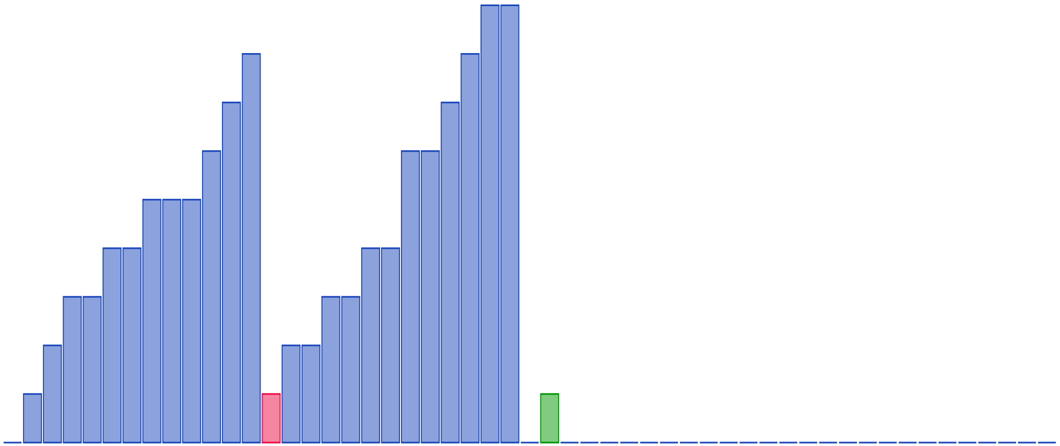


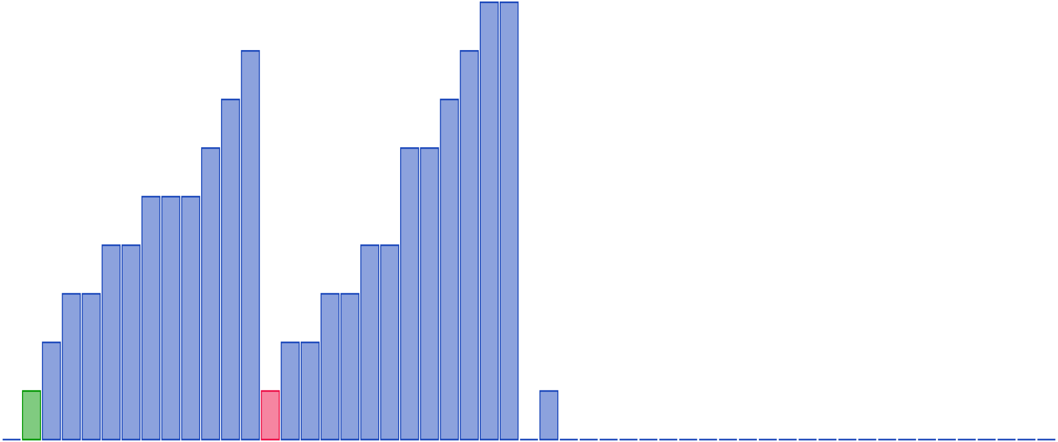


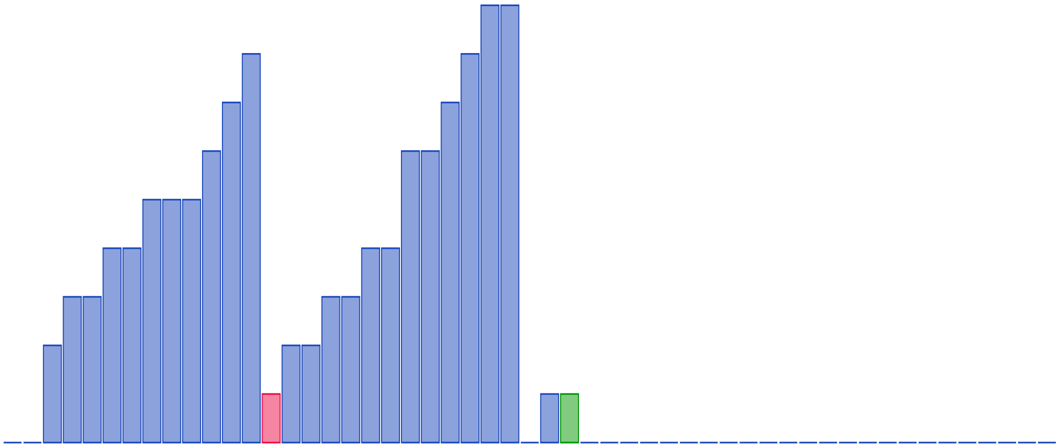


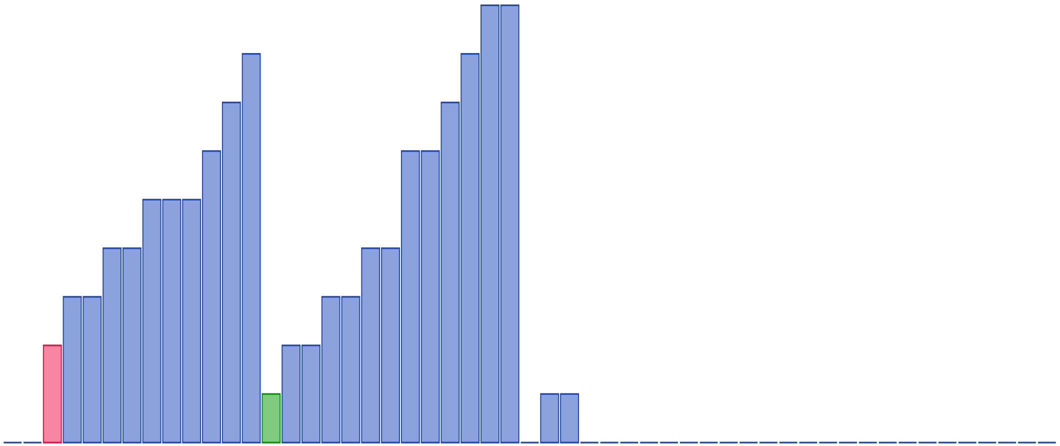


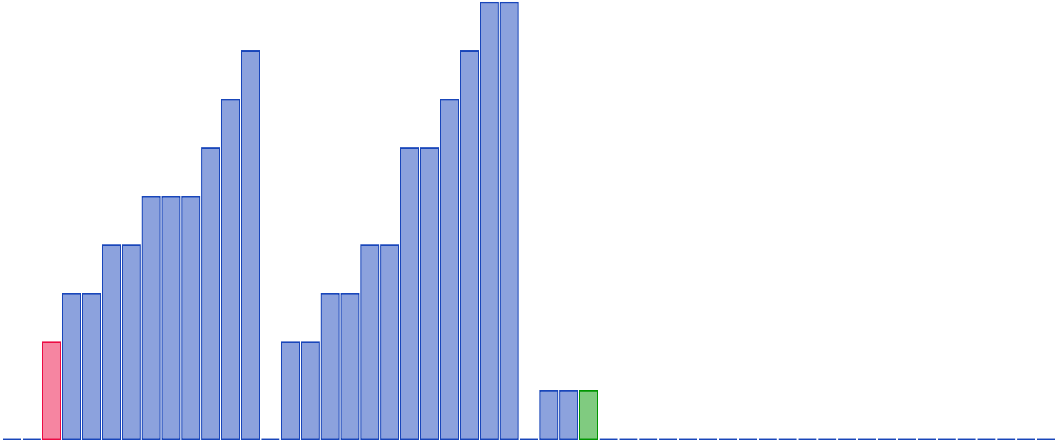


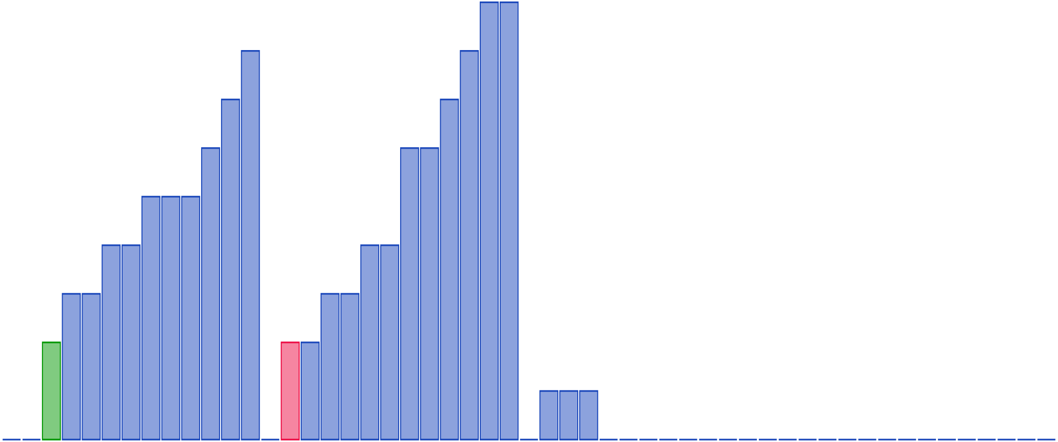


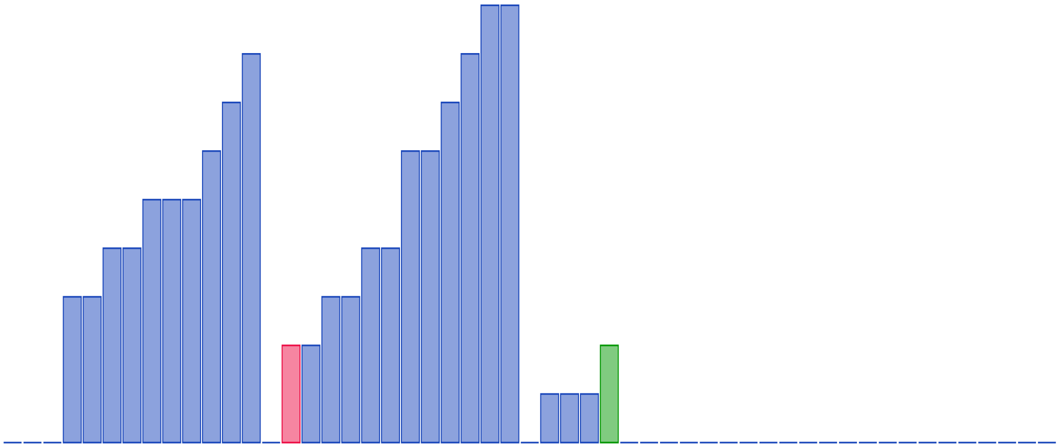


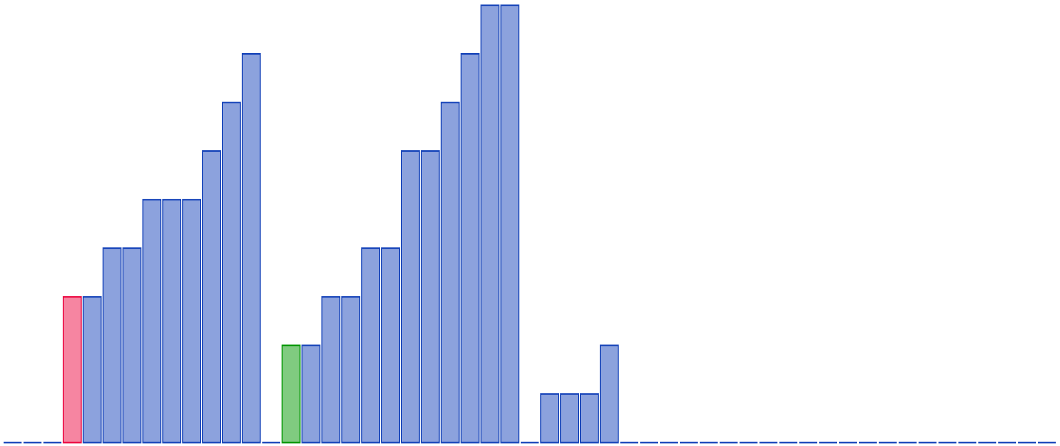


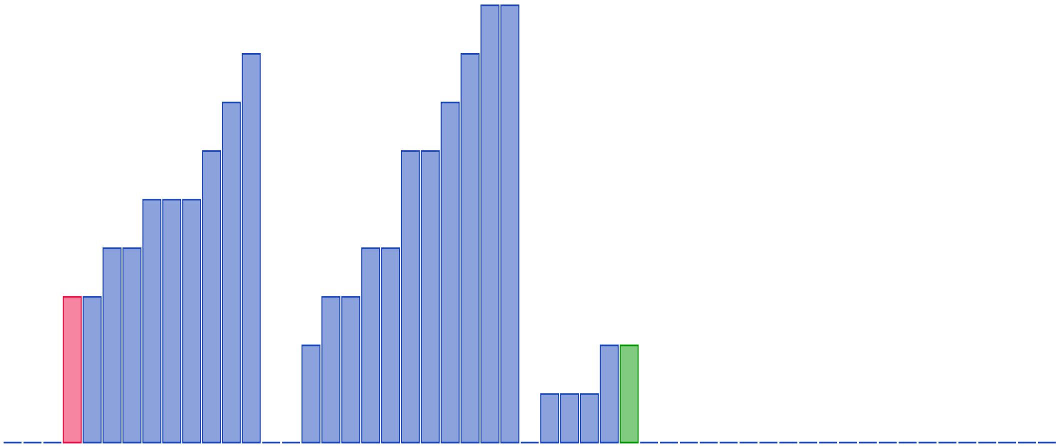


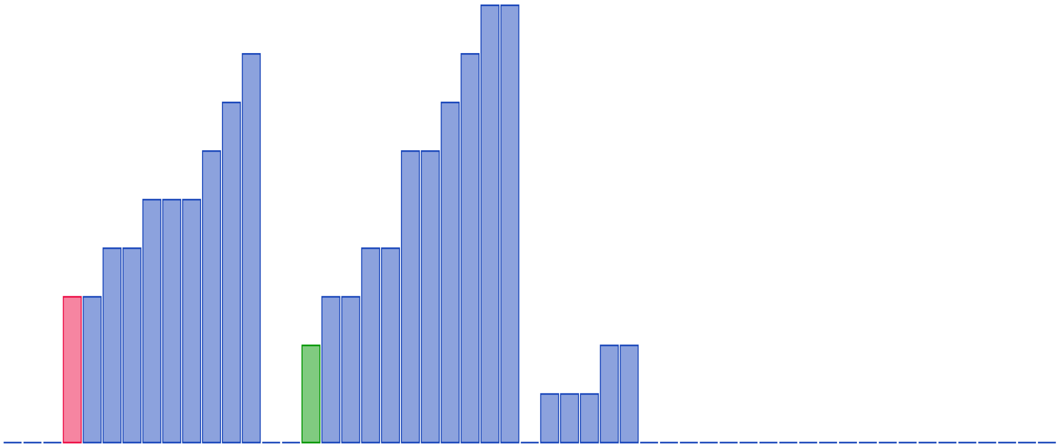


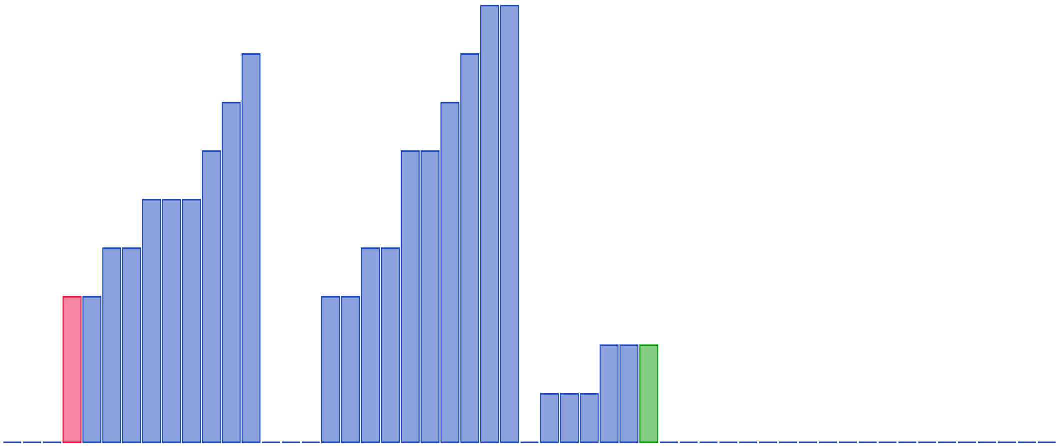


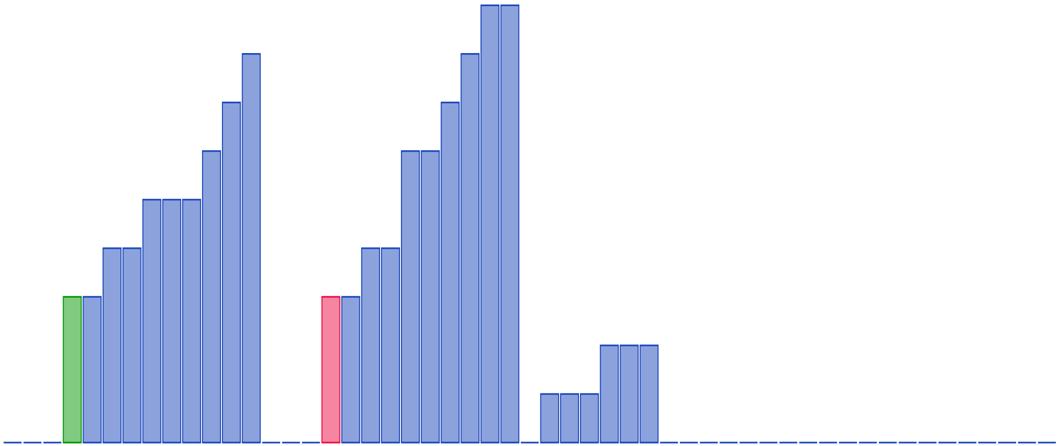


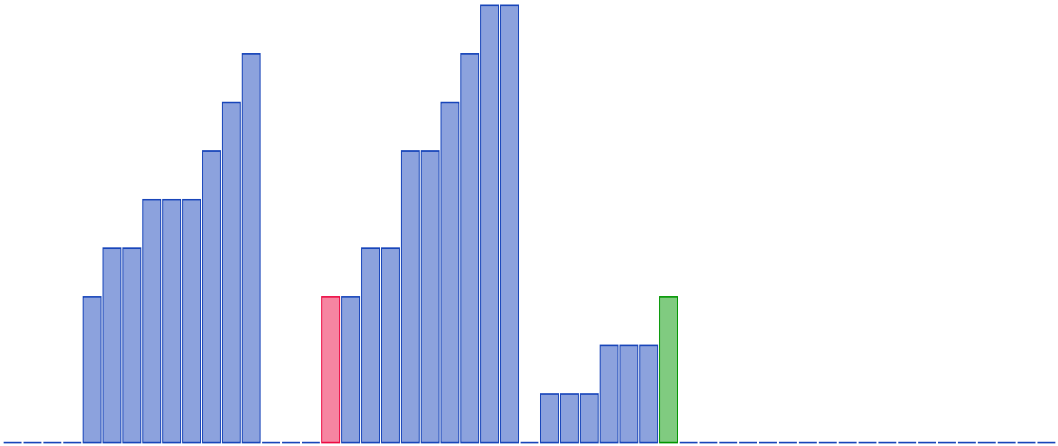


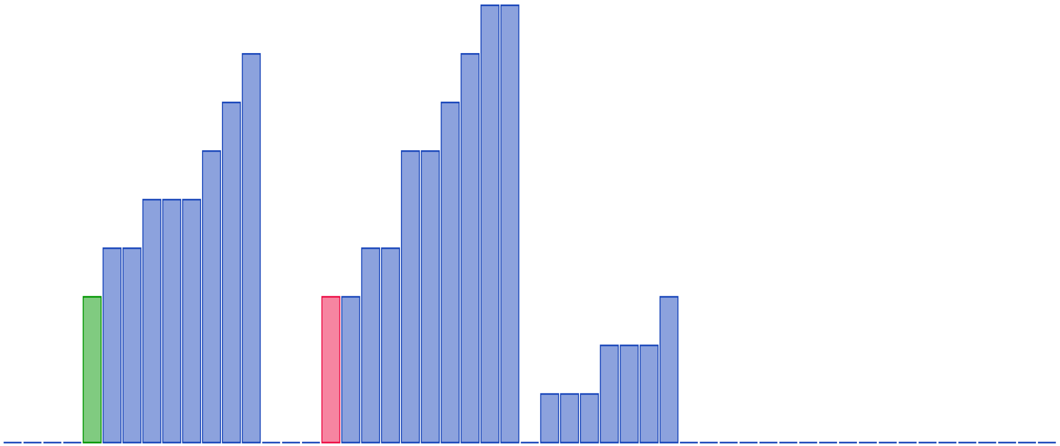


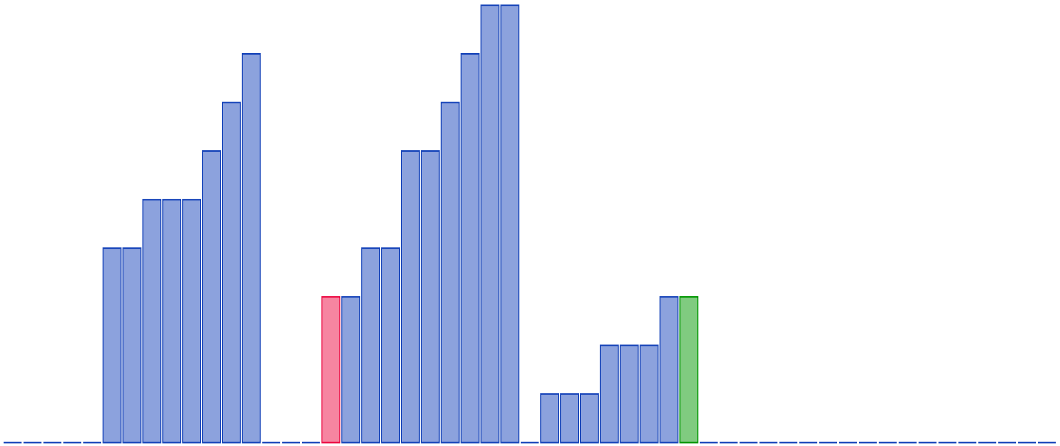


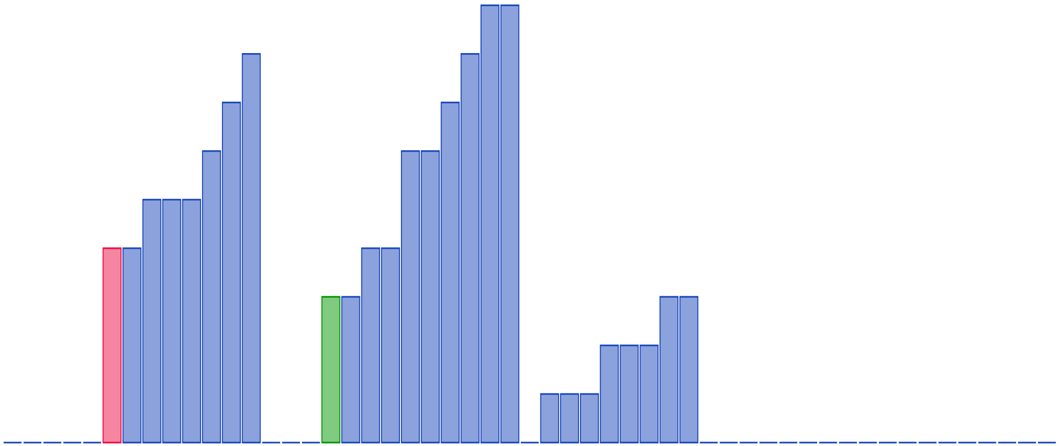


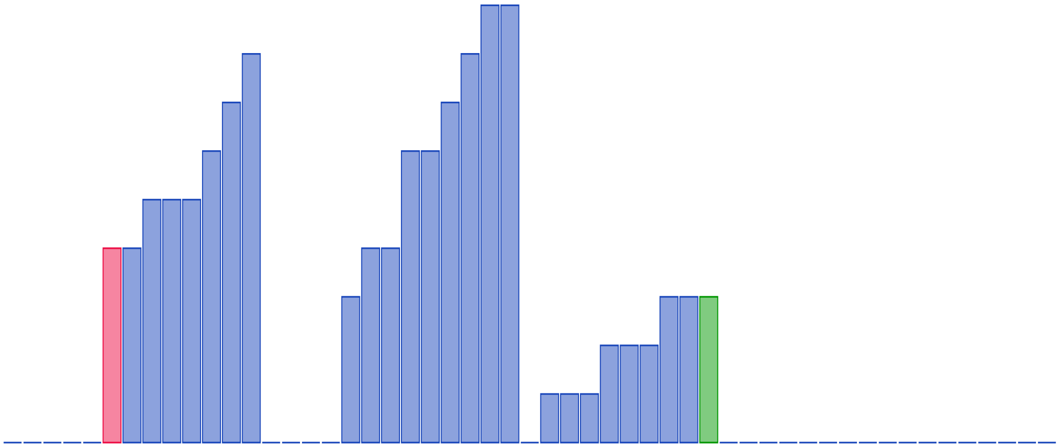


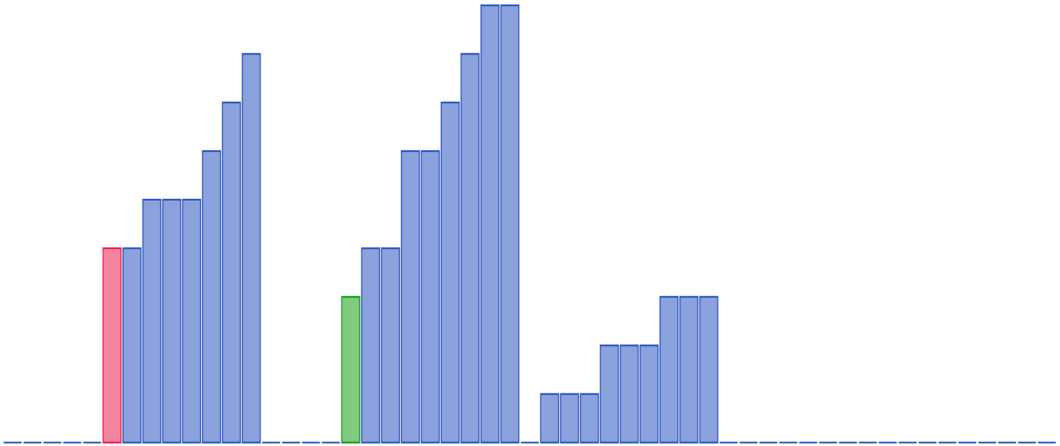


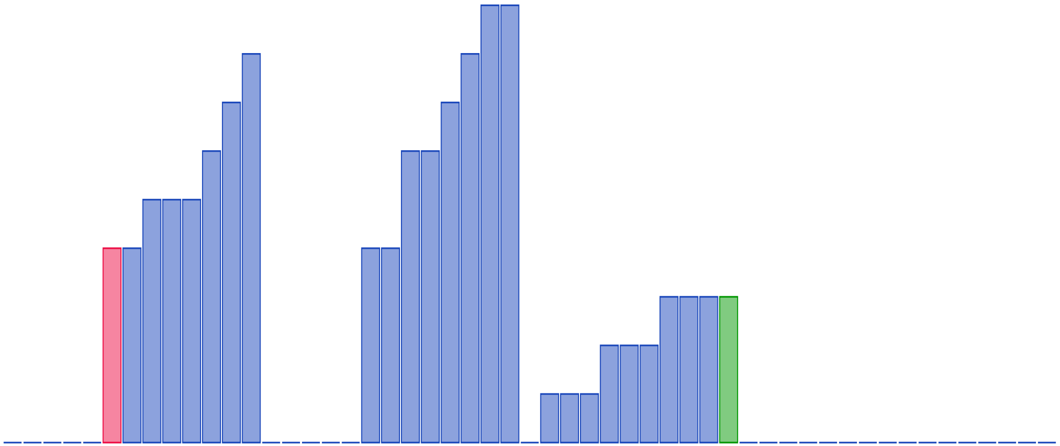


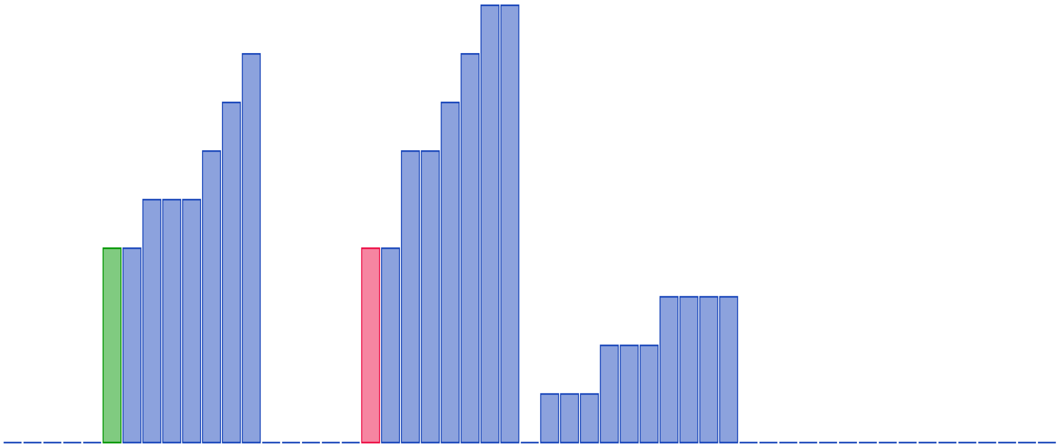


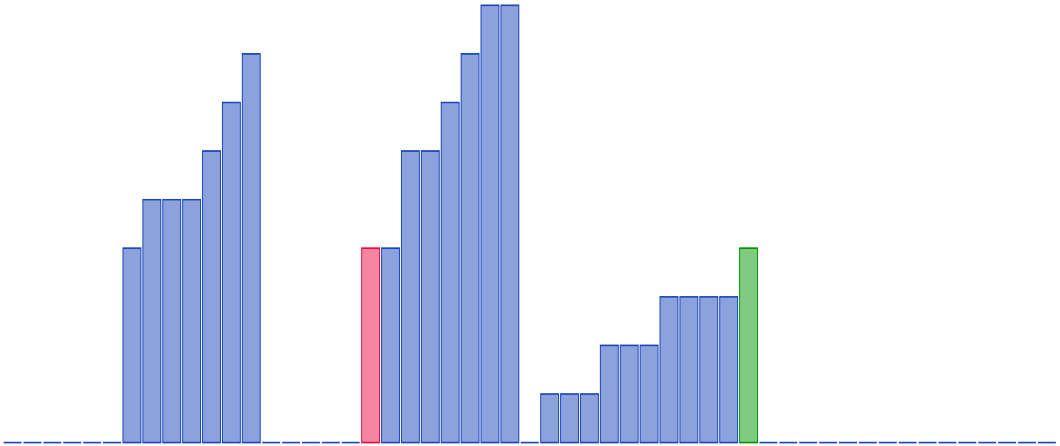


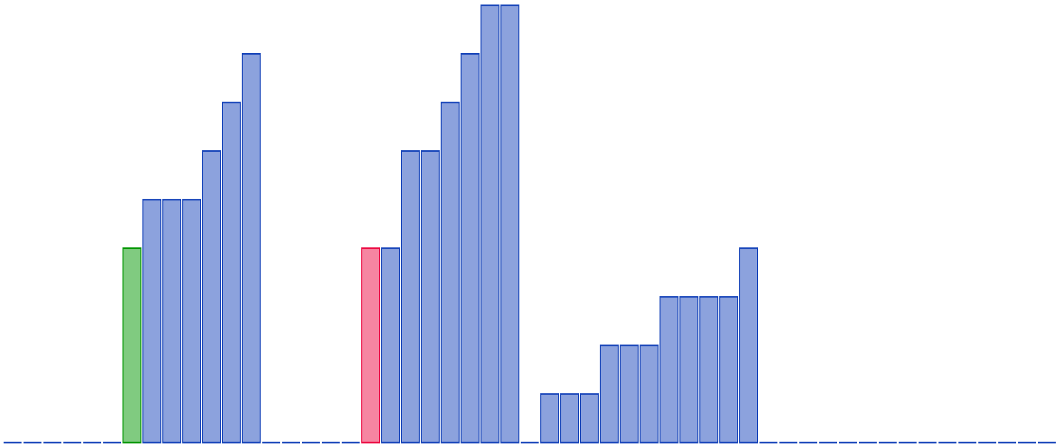


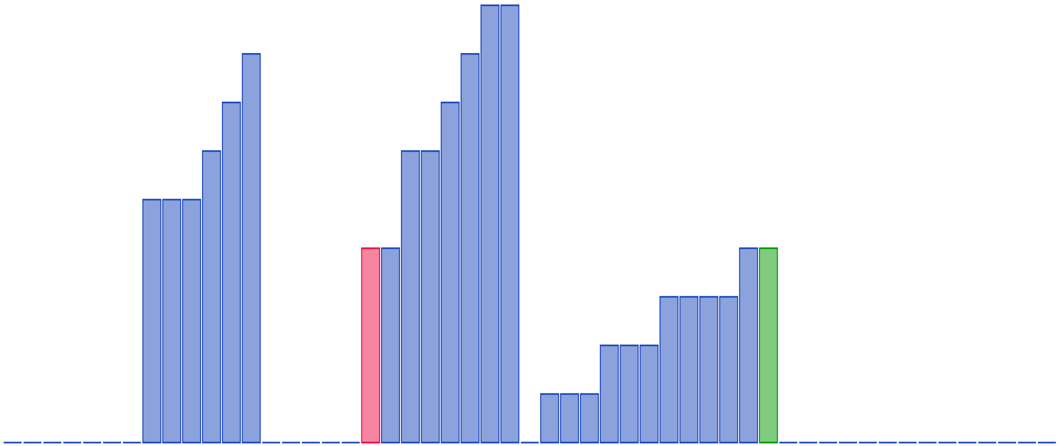


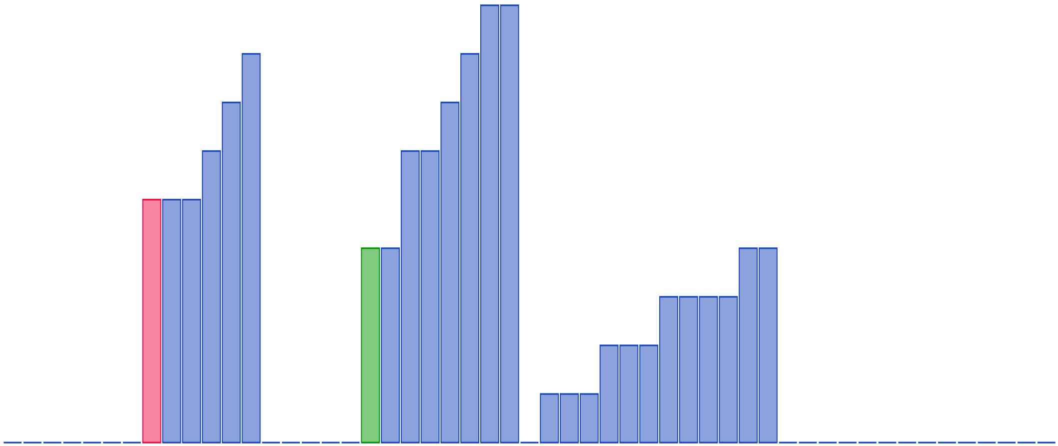


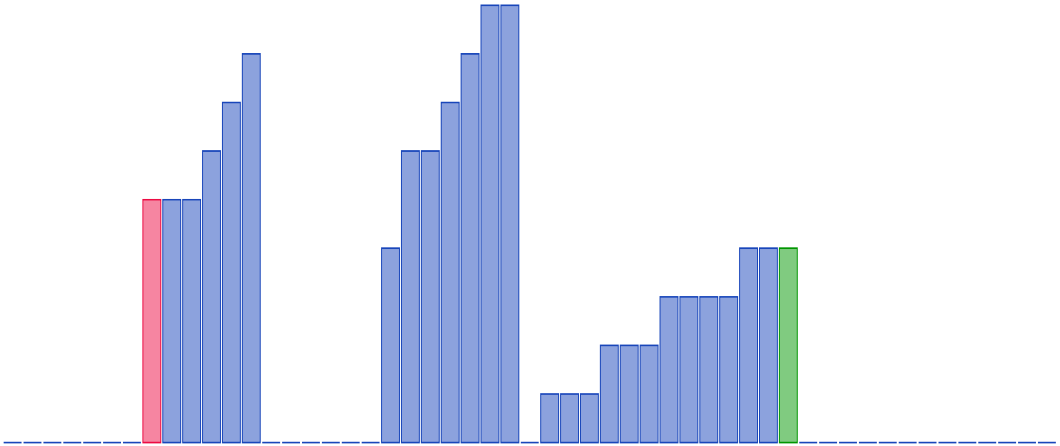




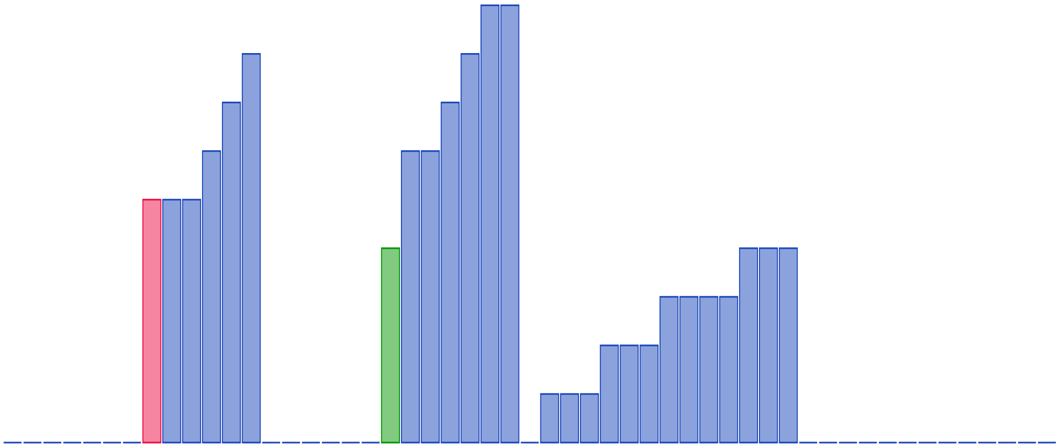


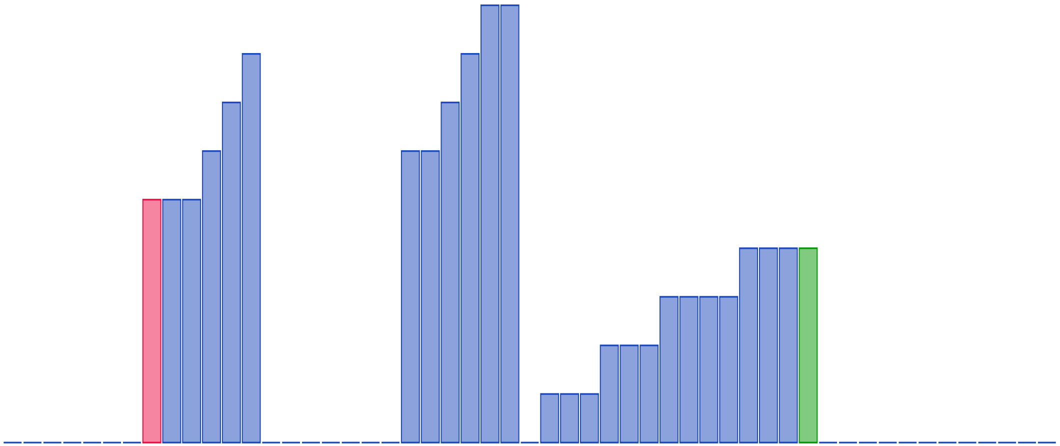


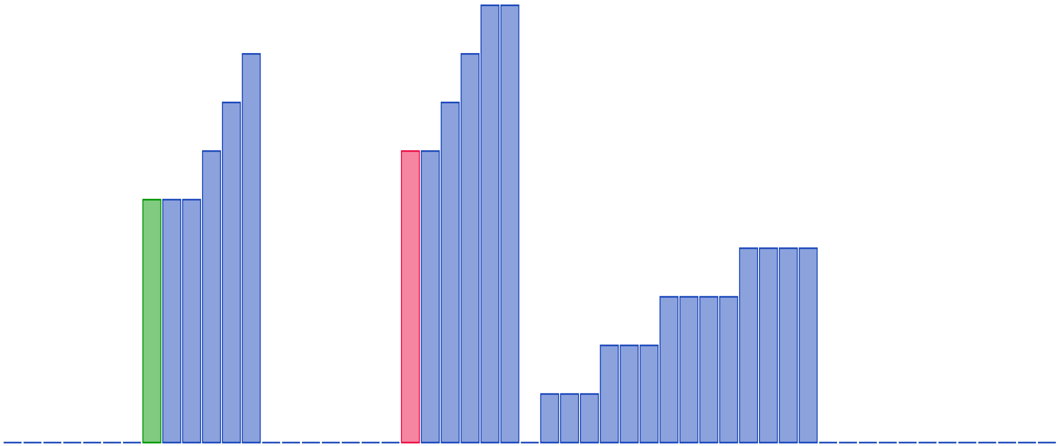


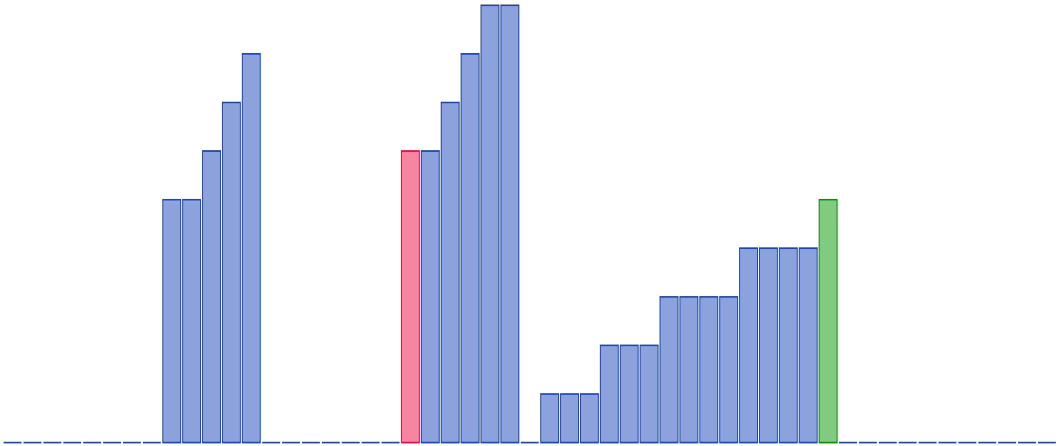


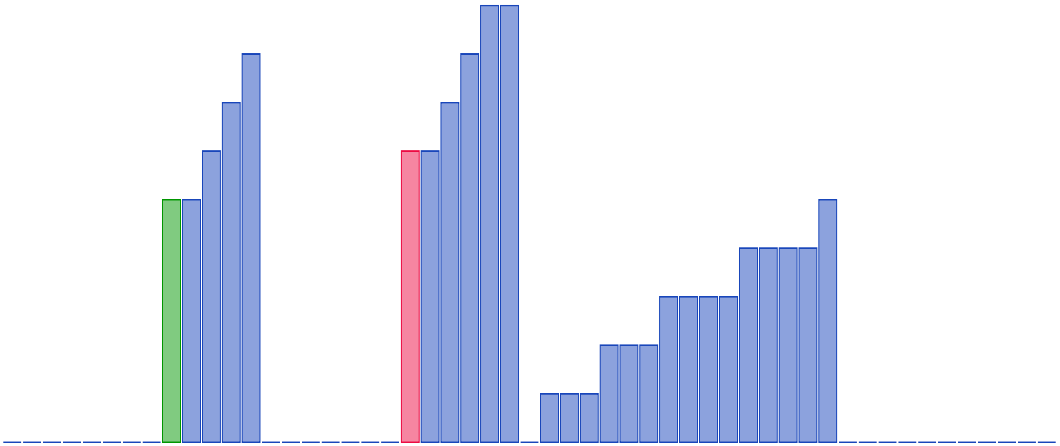
Mergesort: Example

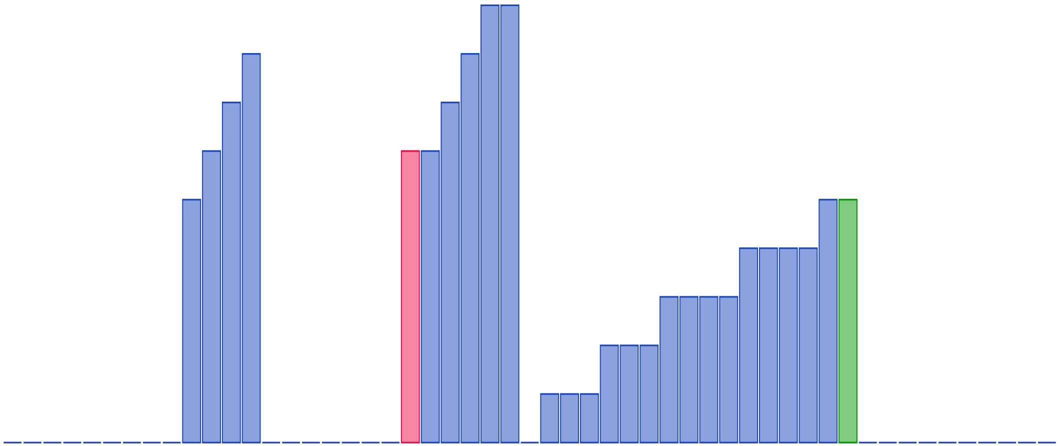


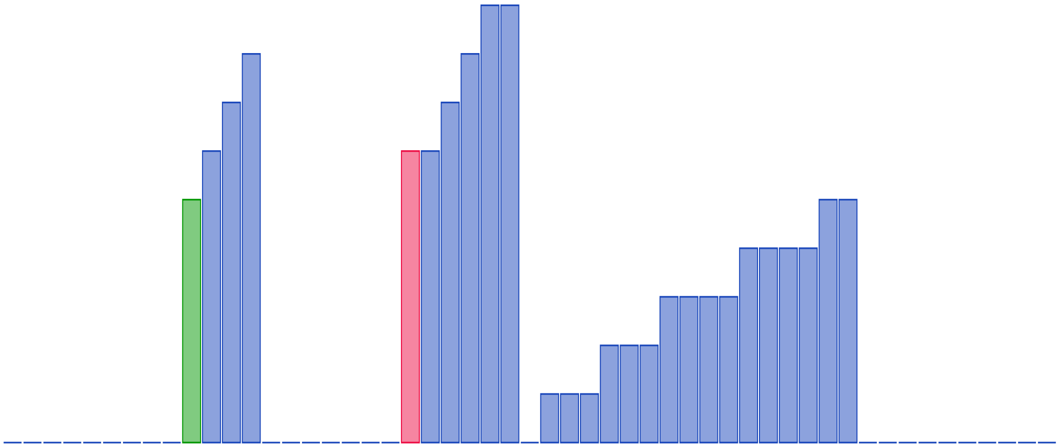


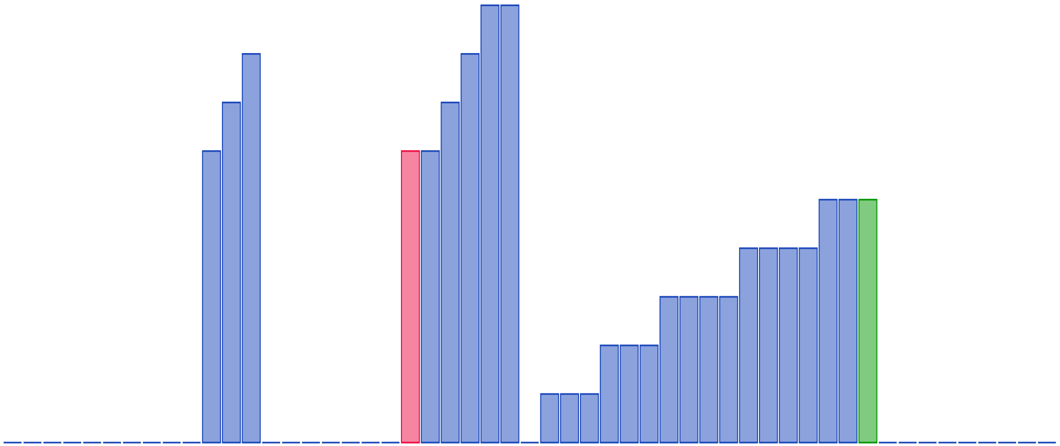


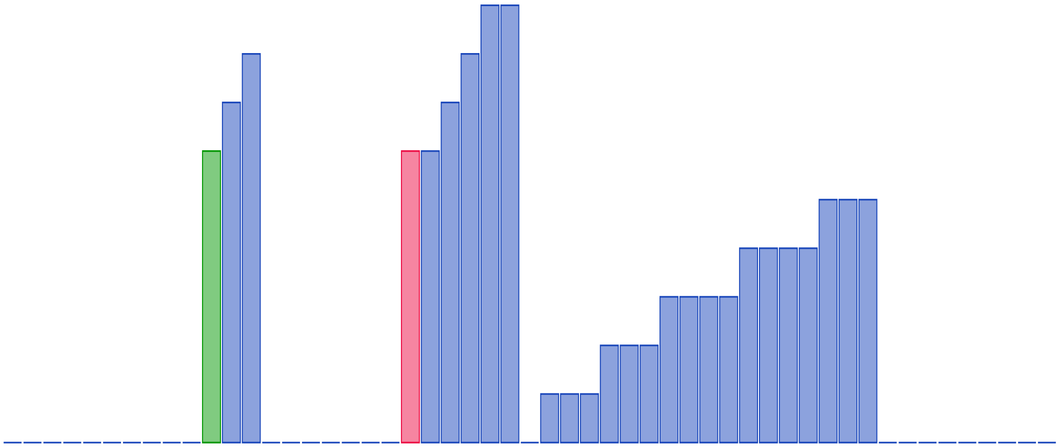


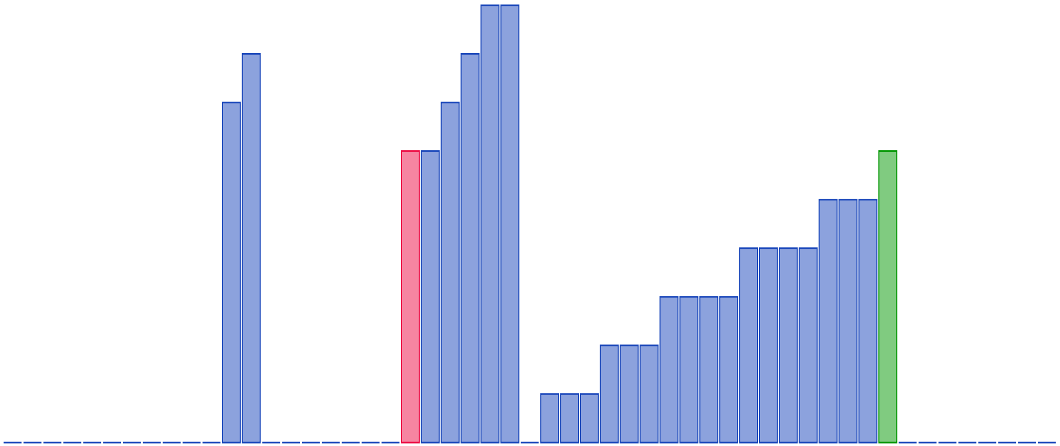


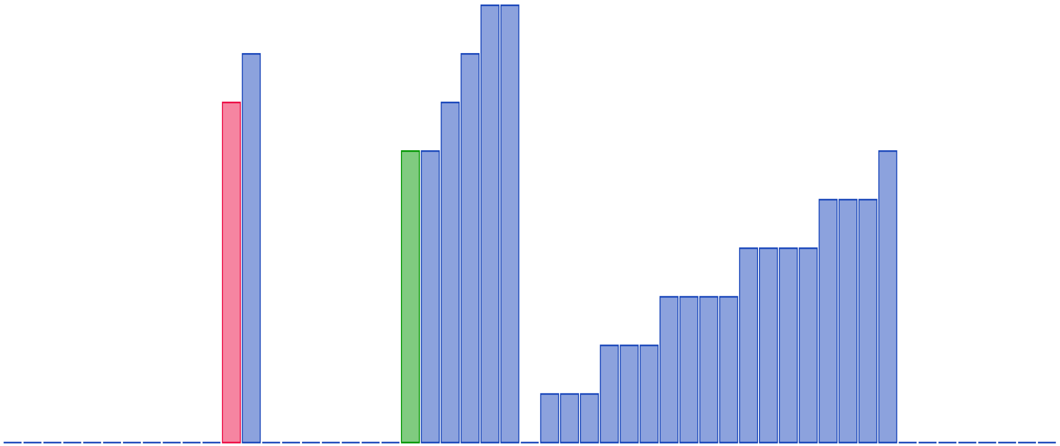


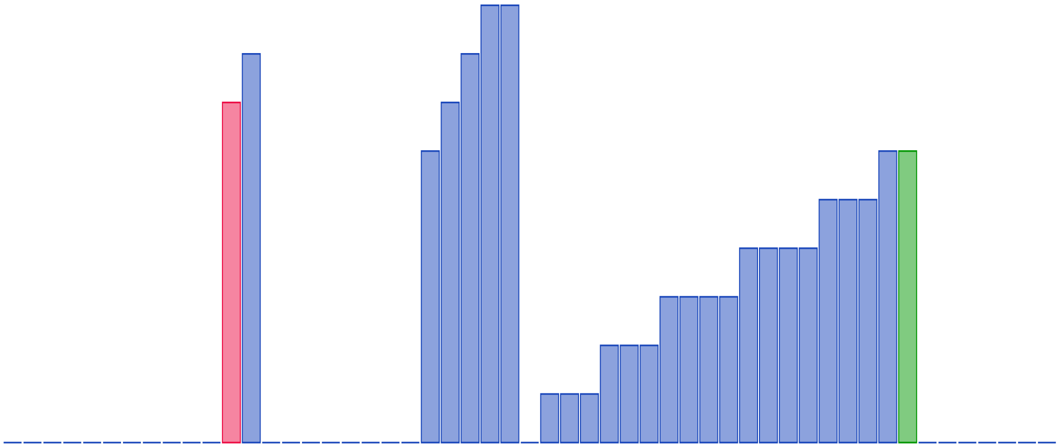


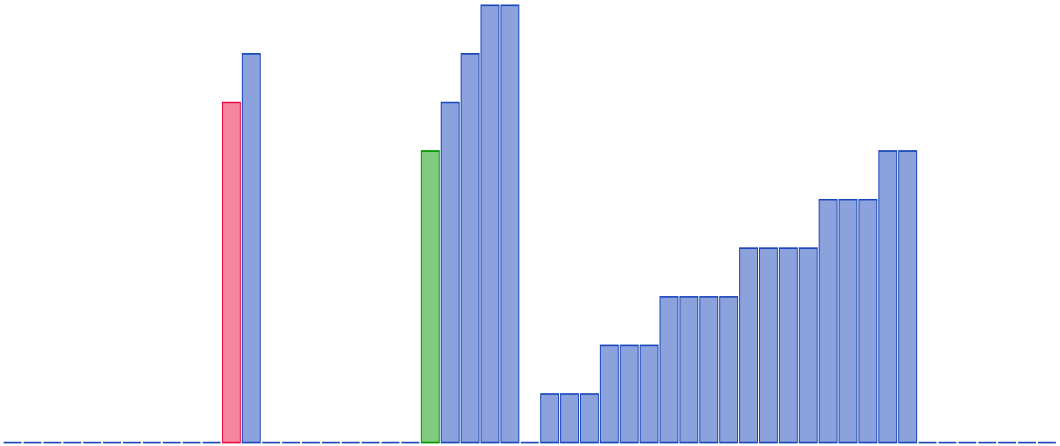


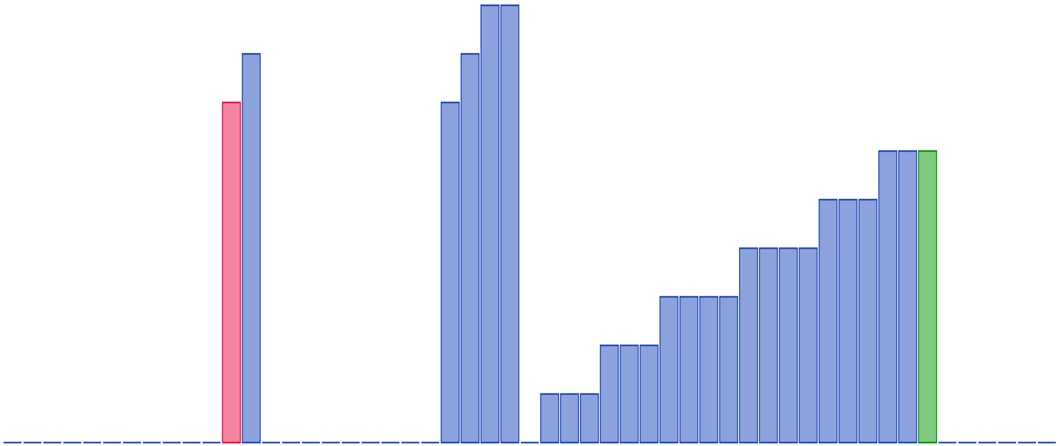


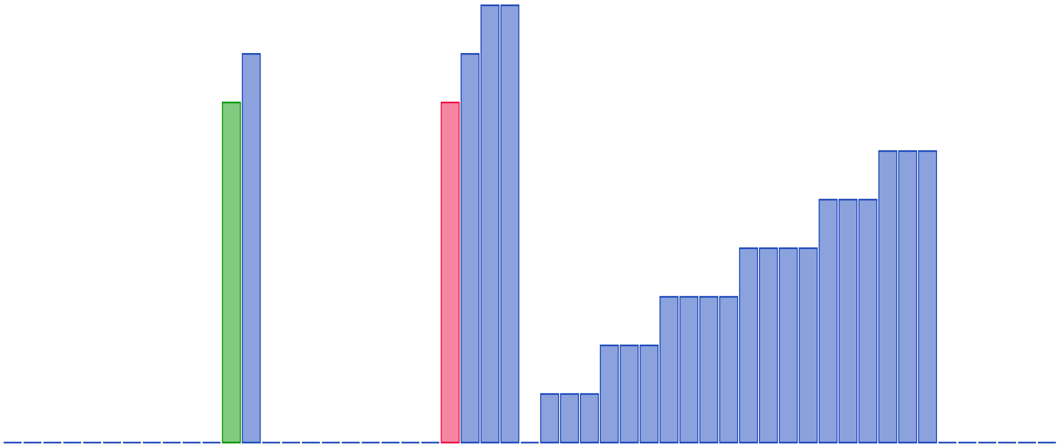


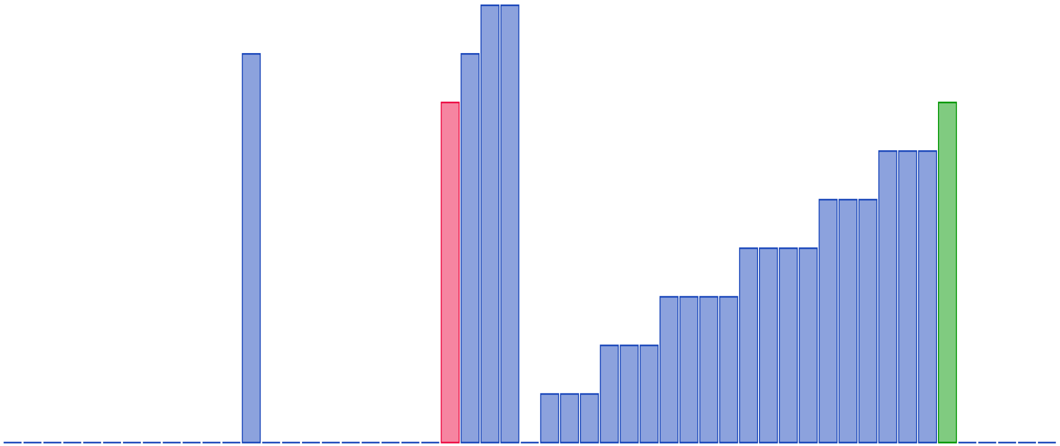


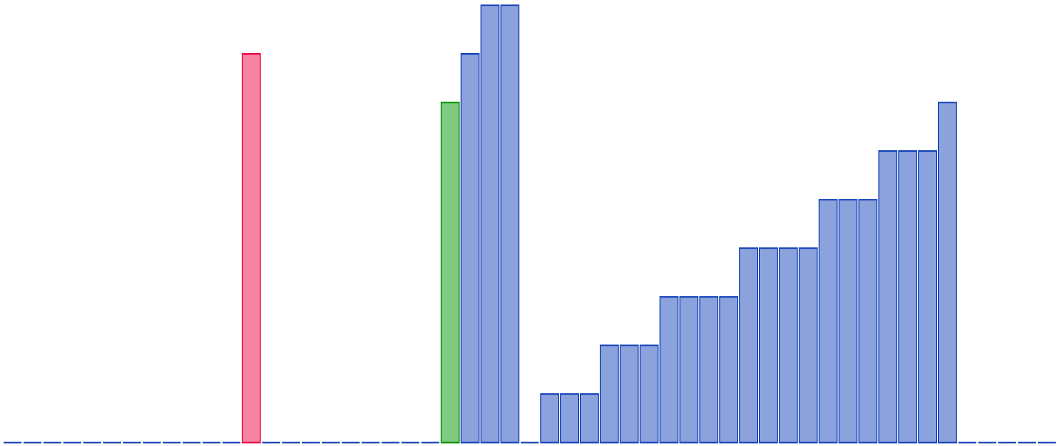


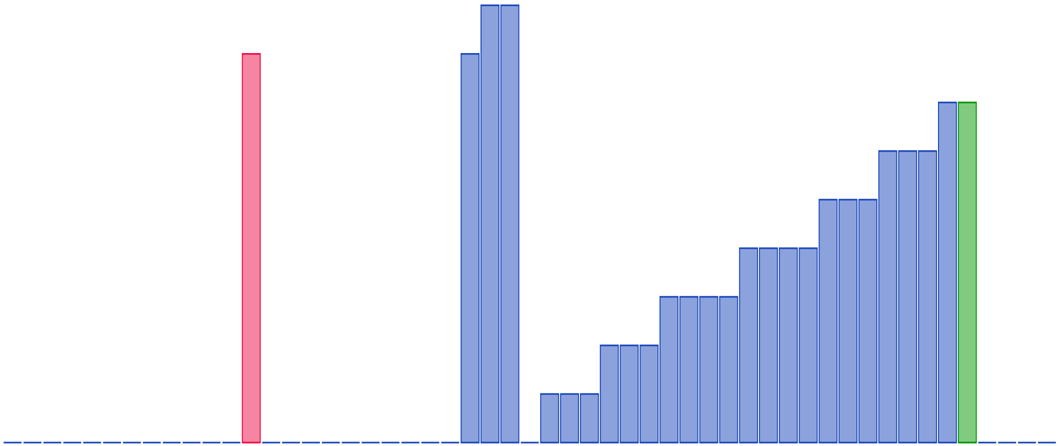


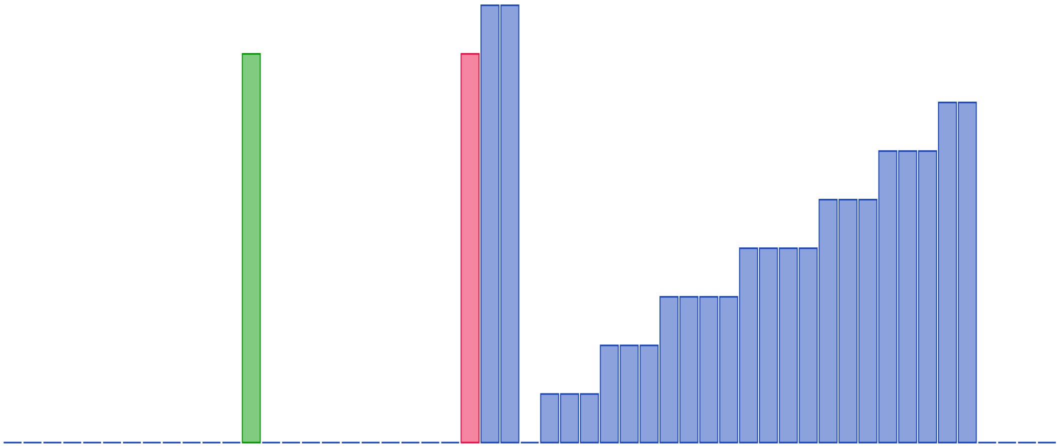


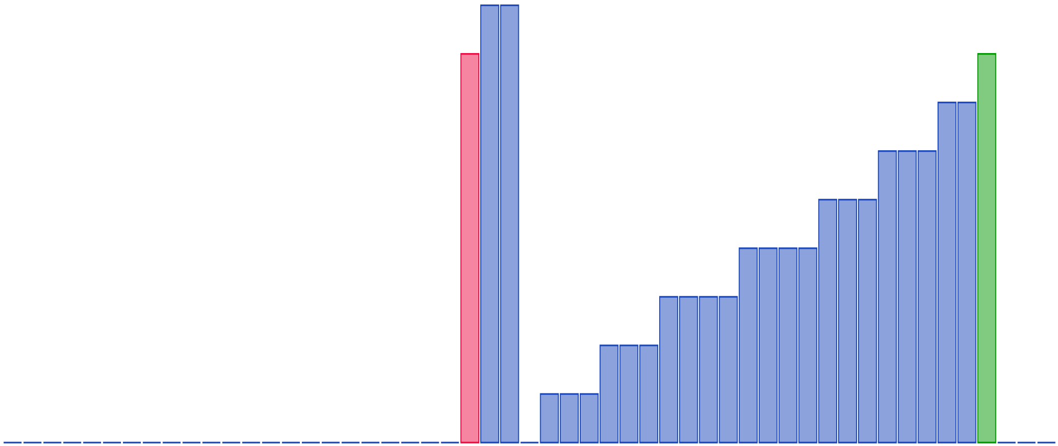


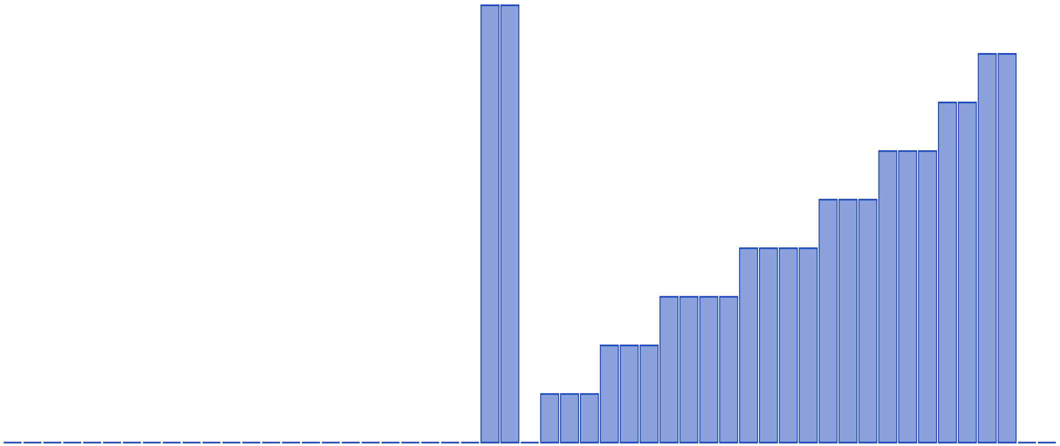


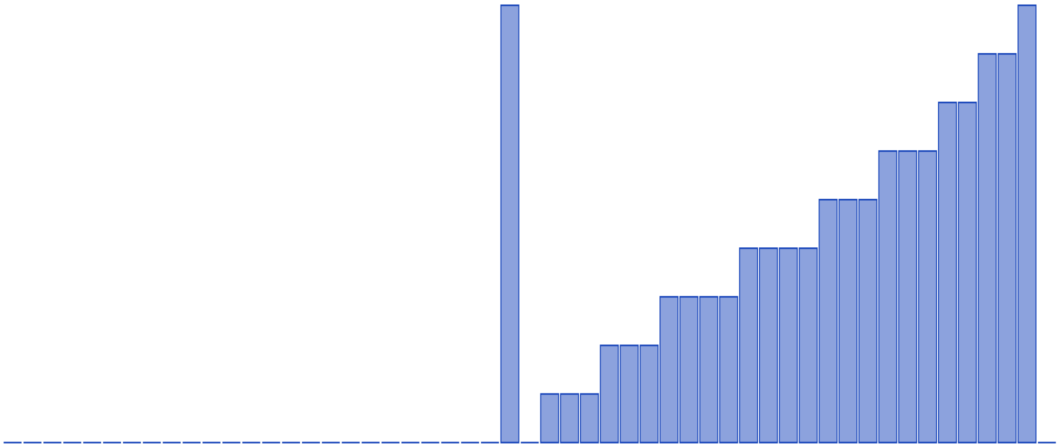


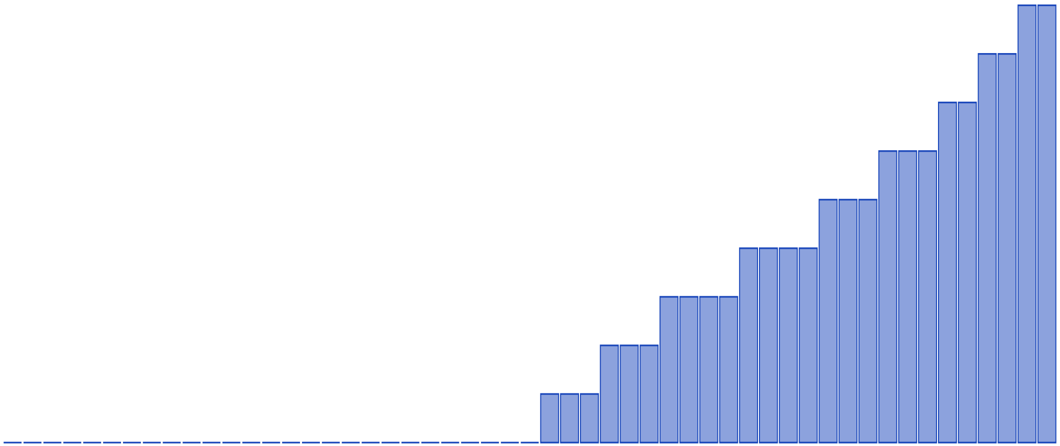


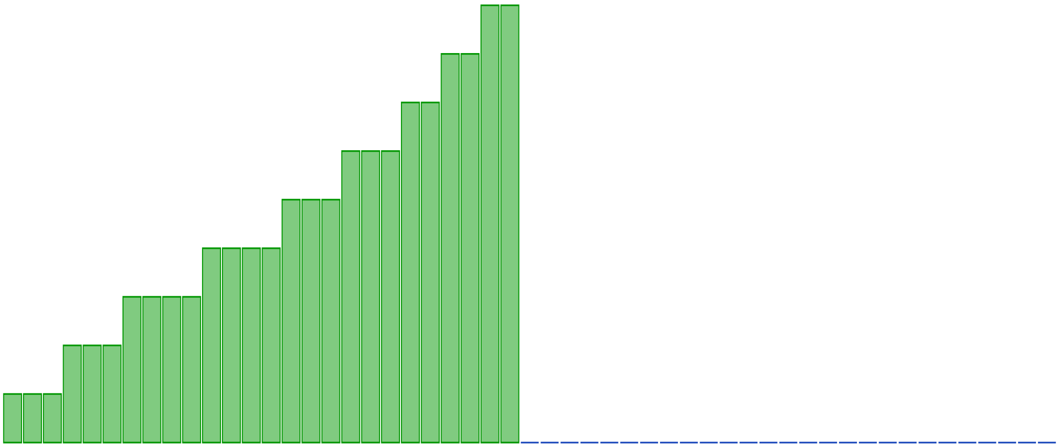












- What should the merge procedure do?
  - Given two sorted arrays, construct a sorted array from them

$i \leftarrow s$, $j \leftarrow m$, $k \leftarrow s$
while $i < m$ **or** $j < t$ **do**
  if $j = t$ **or** $(i < m$ **and** $A[i] \preceq A[j])$ **then**
    $W[k] \leftarrow A[i]$, $i \leftarrow i + 1$, $k \leftarrow k + 1$
  else
    $W[k] \leftarrow A[j]$, $j \leftarrow j + 1$, $k \leftarrow k + 1$
  end if
end while

- What should the merge procedure do?
  - Given two sorted arrays, construct a sorted array from them
- Why does it do this?
  - When $A[i]$ with $s \le i < m$ is moved?
    - After all not greater elements $s \le t < i$
    - After all elements from $[m; e)$ which are smaller than $A[i]$

$i \leftarrow s$, $j \leftarrow m$, $k \leftarrow s$
while $i < m$ **or** $j < t$ **do**
    if $j = t$ **or** $(i < m$ **and** $A[i] \preceq A[j])$ **then**
        $W[k] \leftarrow A[i]$, $i \leftarrow i + 1$, $k \leftarrow k + 1$
    else
        $W[k] \leftarrow A[j]$, $j \leftarrow j + 1$, $k \leftarrow k + 1$
    end if
end while

- What should the merge procedure do?
  - Given two sorted arrays, construct a sorted array from them
- Why does it do this?
  - When $A[i]$ with $s \le i < m$ is moved?
    - After all not greater elements $s \le t < i$
    - After all elements from $[m; e)$ which are smaller than $A[i]$
  - When $A[j]$ with $m \le j < e$ is moved?
    - After all not greater elements $m \le t < j$
    - After all elements from $[s; m)$ which are not greater than $A[j]$

$i \leftarrow s$, $j \leftarrow m$, $k \leftarrow s$
while $i < m$ **or** $j < t$ **do**
  if $j = t$ **or** $(i < m$ **and** $A[i] \preceq A[j])$ **then**
    $W[k] \leftarrow A[i]$, $i \leftarrow i + 1$, $k \leftarrow k + 1$
  else
    $W[k] \leftarrow A[j]$, $j \leftarrow j + 1$, $k \leftarrow k + 1$
  end if
end while

- What should the merge procedure do?
  - Given two sorted arrays, construct a sorted array from them
- Why does it do this?
  - When $A[i]$ with $s \le i < m$ is moved?
    - After all not greater elements $s \le t < i$
    - After all elements from $[m; e)$ which are smaller than $A[i]$
  - When $A[j]$ with $m \le j < e$ is moved?
    - After all not greater elements $m \le t < j$
    - After all elements from $[s; m)$ which are not greater than $A[j]$
  - So, all elements are moved precisely in the sorted order

$i \leftarrow s$, $j \leftarrow m$, $k \leftarrow s$
while $i < m$ **or** $j < t$ **do**
  if $j = t$ **or** ($i < m$ **and** $A[i] \preceq A[j]$) **then**
    $W[k] \leftarrow A[i]$, $i \leftarrow i + 1$, $k \leftarrow k + 1$
  else
    $W[k] \leftarrow A[j]$, $j \leftarrow j + 1$, $k \leftarrow k + 1$
  end if
end while

- What should the merge procedure do?
  - Given two sorted arrays, construct a sorted array from them
- Why does it do this?
  - When $A[i]$ with $s \le i < m$ is moved?
    - After all not greater elements $s \le t < i$
    - After all elements from $[m; e)$ which are smaller than $A[i]$
  - When $A[j]$ with $m \le j < e$ is moved?
    - After all not greater elements $m \le t < j$
    - After all elements from $[s; m)$ which are not greater than $A[j]$
  - So, all elements are moved precisely in the sorted order
- The overall correctness of mergesort simply follows

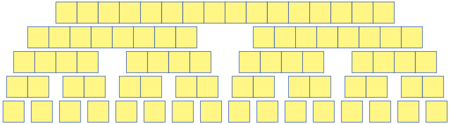
$i \leftarrow s$, $j \leftarrow m$, $k \leftarrow s$
while $i < m$ **or** $j < t$ **do**
  if $j = t$ **or** $(i < m$ **and** $A[i] \preceq A[j])$ **then**
    $W[k] \leftarrow A[i]$, $i \leftarrow i + 1$, $k \leftarrow k + 1$
  else
    $W[k] \leftarrow A[j]$, $j \leftarrow j + 1$, $k \leftarrow k + 1$
  end if
end while

Running time is always $\Theta(N \log N)$.

Running time is always $\Theta(N \log N)$.
Proof:

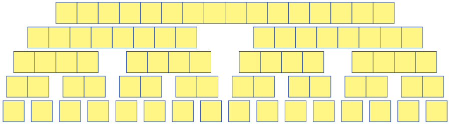
Running time is always $\Theta(N \log N)$.
Proof:

- Look at the call tree to the right



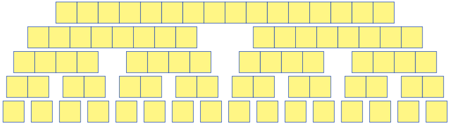
Running time is always $\Theta(N \log N)$.
Proof:

- Look at the call tree to the right
- Maximum depth: $\Theta(\log N)$, as every subarray size is exactly half of its parent's size



Running time is always $\Theta(N \log N)$.
Proof:

- Look at the call tree to the right
- Maximum depth: $\Theta(\log N)$, as every subarray size is exactly half of its parent's size
- All merges at given depth run in $\Theta(N)$



Running time is always $\Theta(N \log N)$.
Proof:

- Look at the call tree to the right
- Maximum depth: $\Theta(\log N)$, as every subarray size is exactly half of its parent's size
- All merges at given depth run in $\Theta(N)$
- Overall running time: $\Theta(N \log N)$