APS105
Winter 2012

Jonathan Deber
jdeber -at- cs -dot- toronto -dot- edu

Lecture 34
April 11, 2012

Today

- Even More Sorting

Sorting an Existing List

- Simpler but inefficient algorithms
  - e.g., bubble sort, selection sort, insertion sort
- Complicated but faster algorithms
  - e.g., merge sort, quicksort

Bubble Sort

- Simplest of the "exchange sorts"

```
while (the list is unsorted)
{
    Go through the list, and compare pairs of items.
    if (they're out of order)
    {
        Swap them.
    }
}
```

23 49 10 4 28 88 64 37

23 49 10 4 28 88 64 37

23 10 49 4 28 88 64 37

23 10 49 4 28 88 64 37

23 10 4 49 28 88 64 37

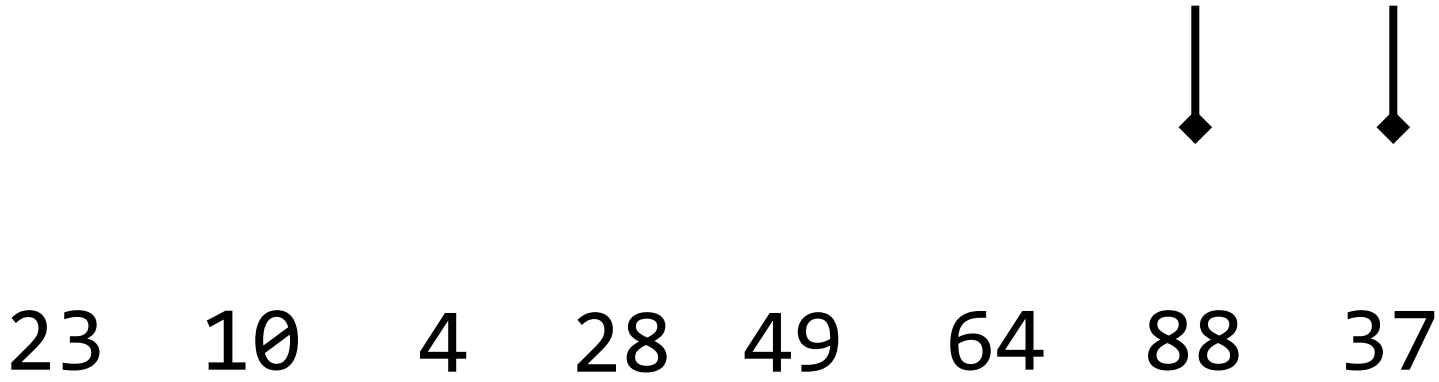
23 10 4 49 28 88 64 37

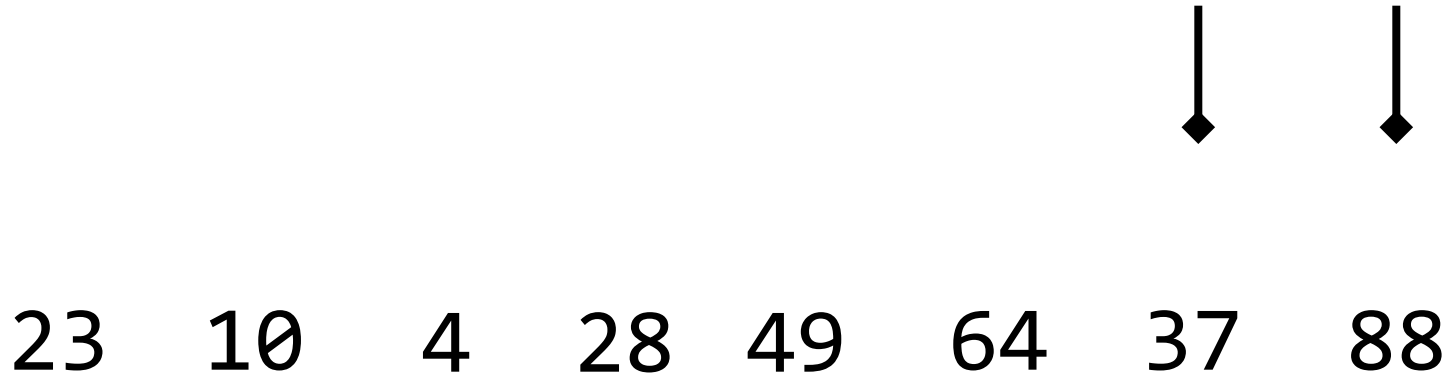
23 10 4 28 49 88 64 37

23 10 4 28 49 88 64 37

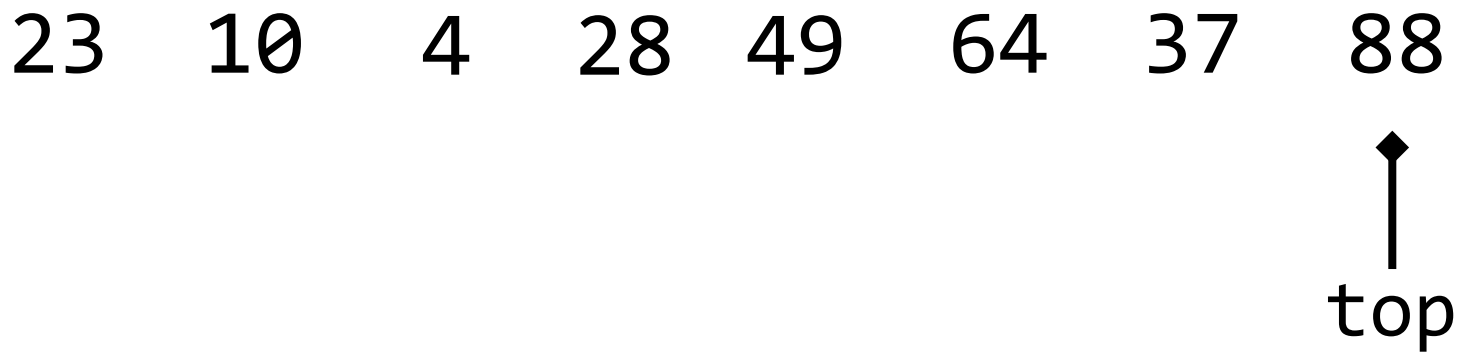
23 10 4 28 49 88 64 37

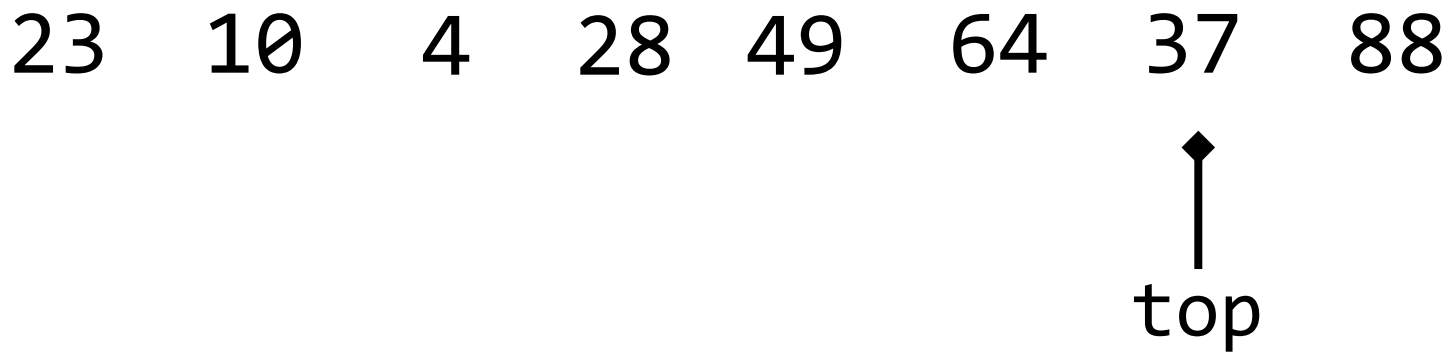

23 10 4 28 49 64 88 37

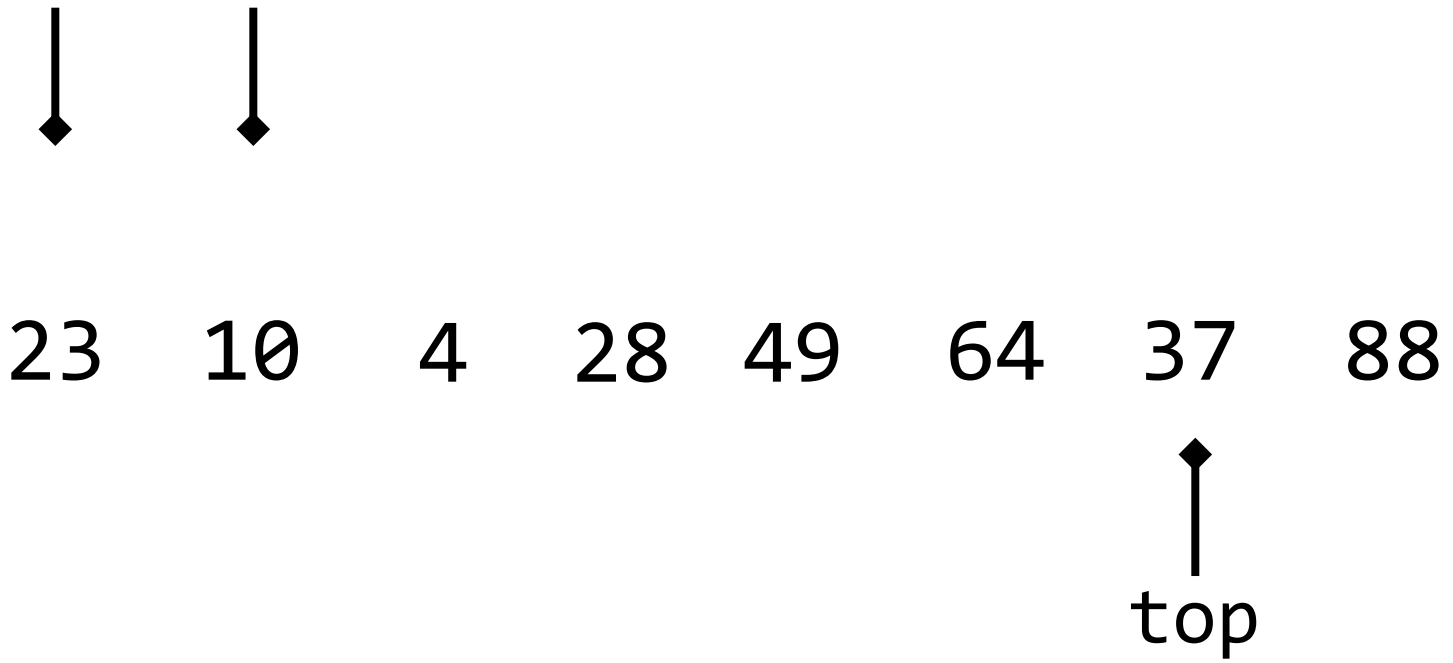

23 10 4 28 49 64 88 37

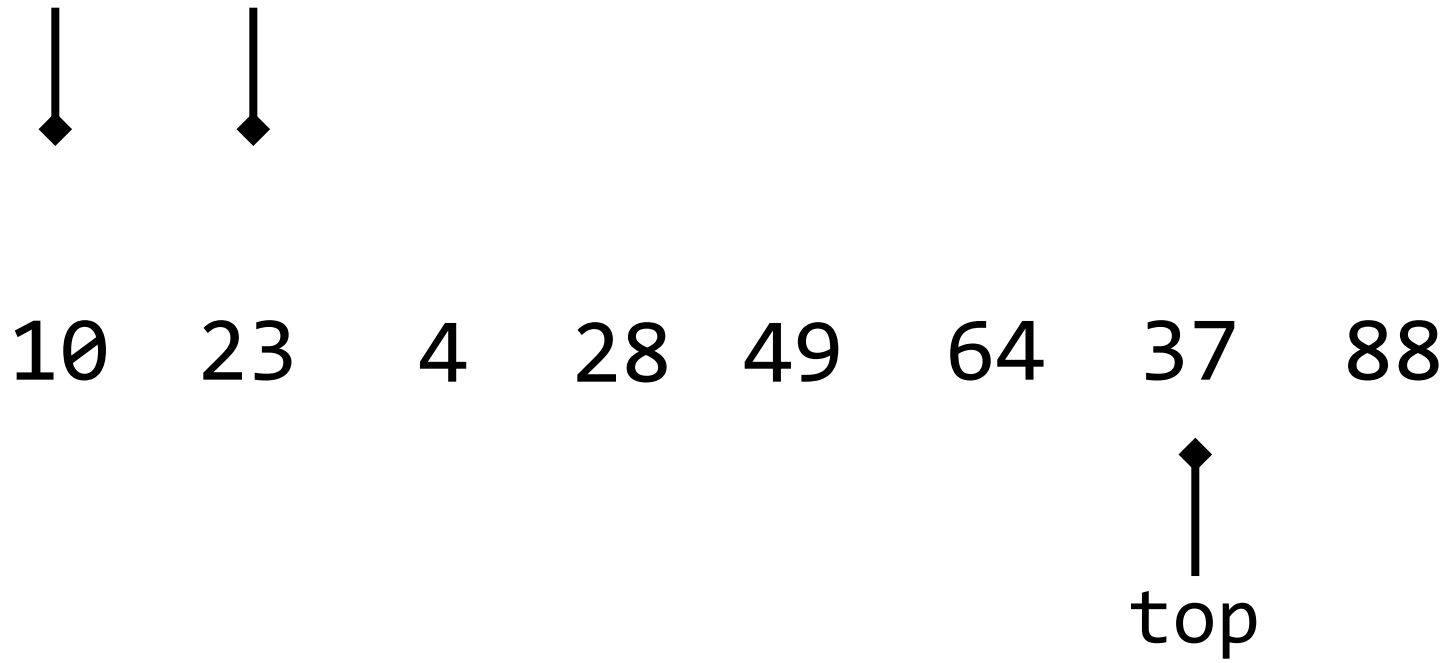

23 10 4 28 49 64 37 88

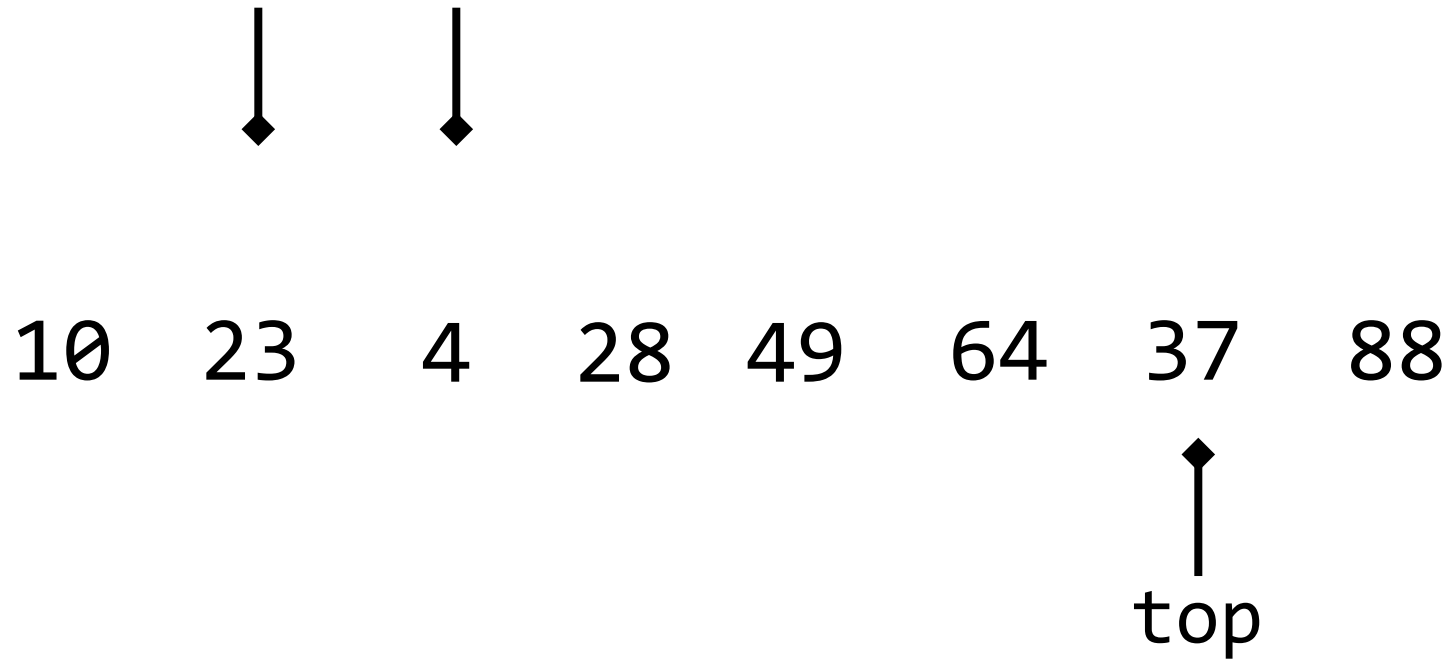
23 10 4 28 49 64 37 88

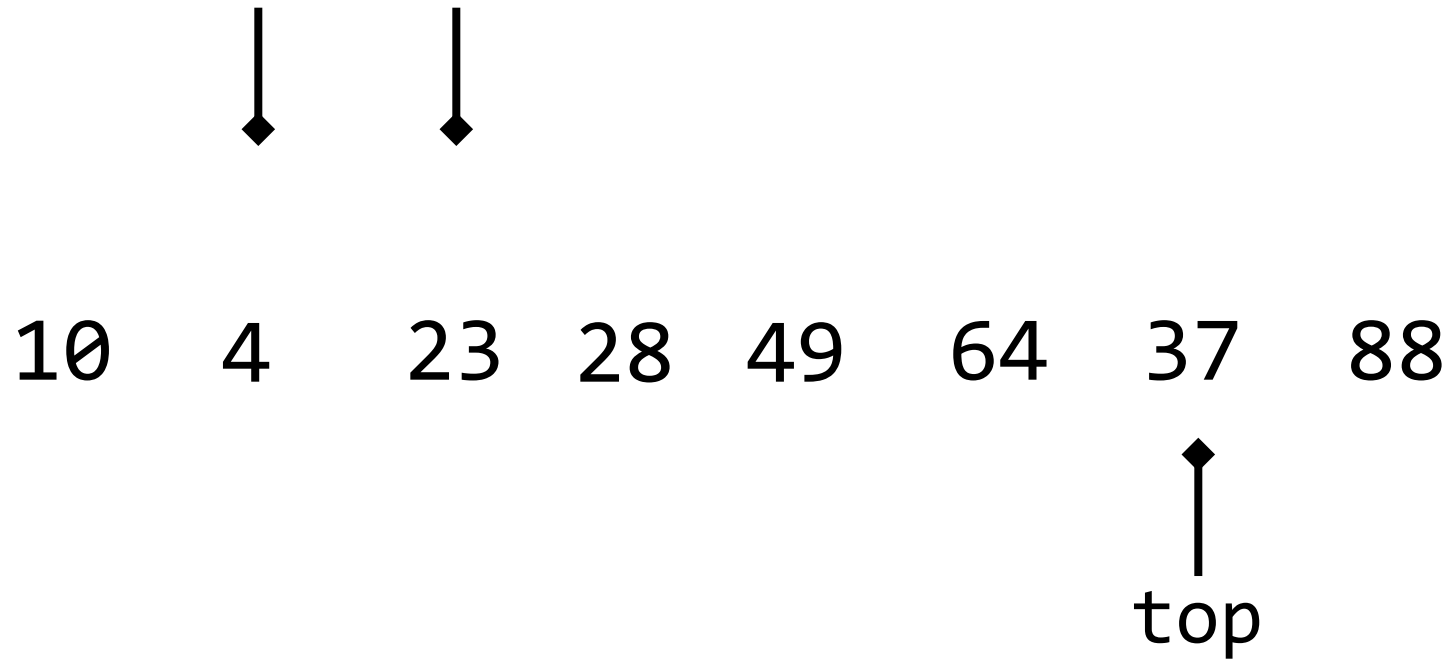

23 10 4 28 49 64 37 88
top

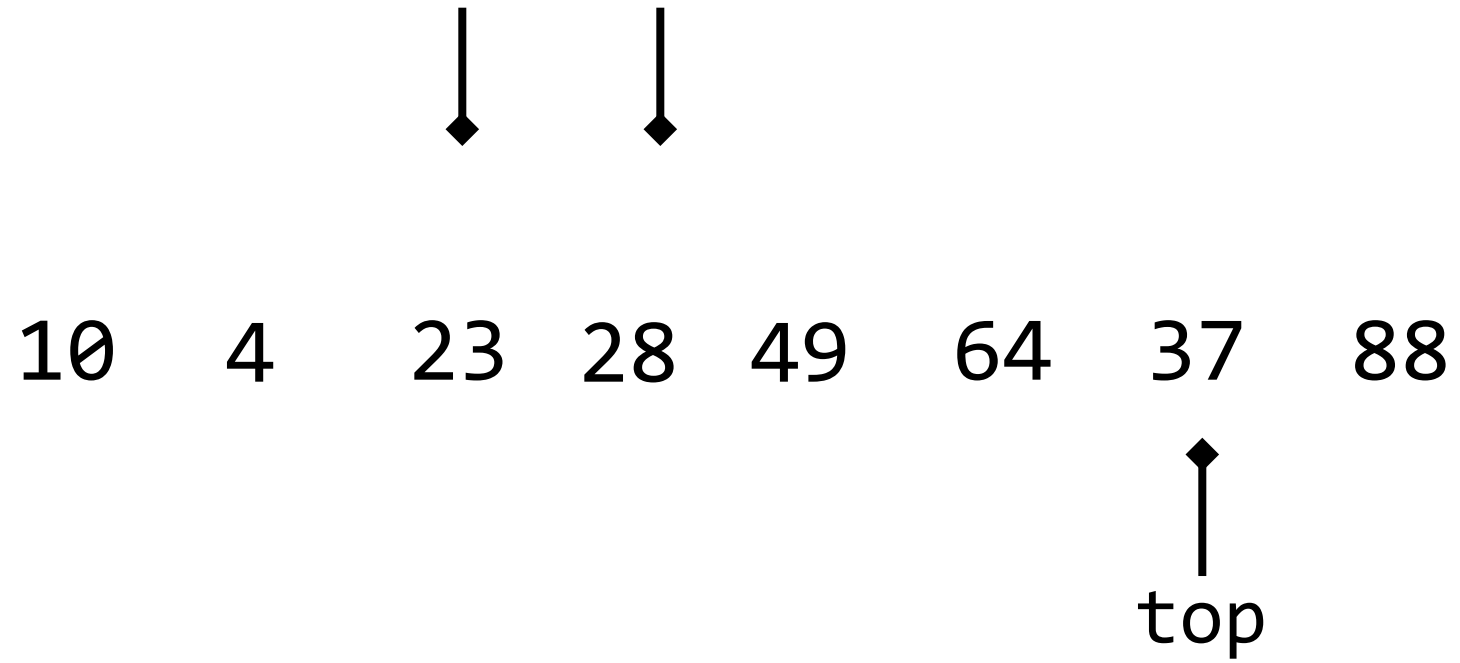

23 10 4 28 49 64 37 88
top

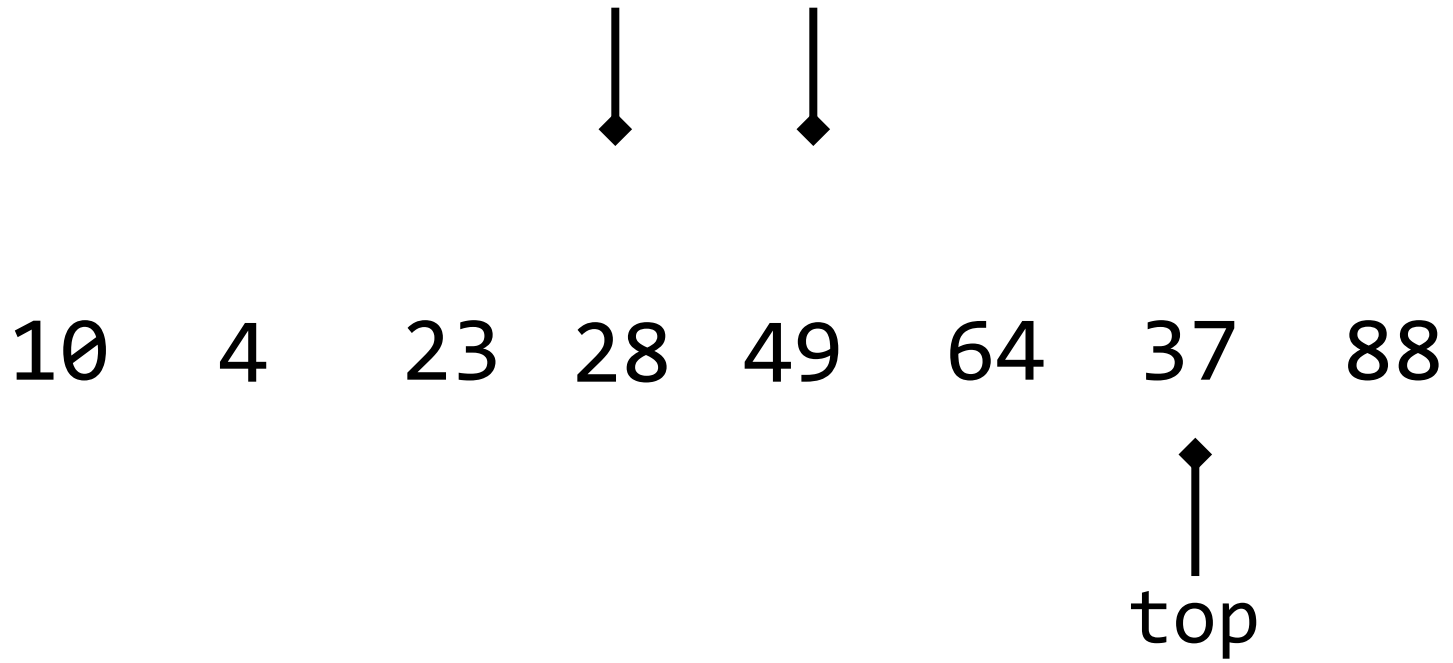

23 10 4 28 49 64 37 88
top

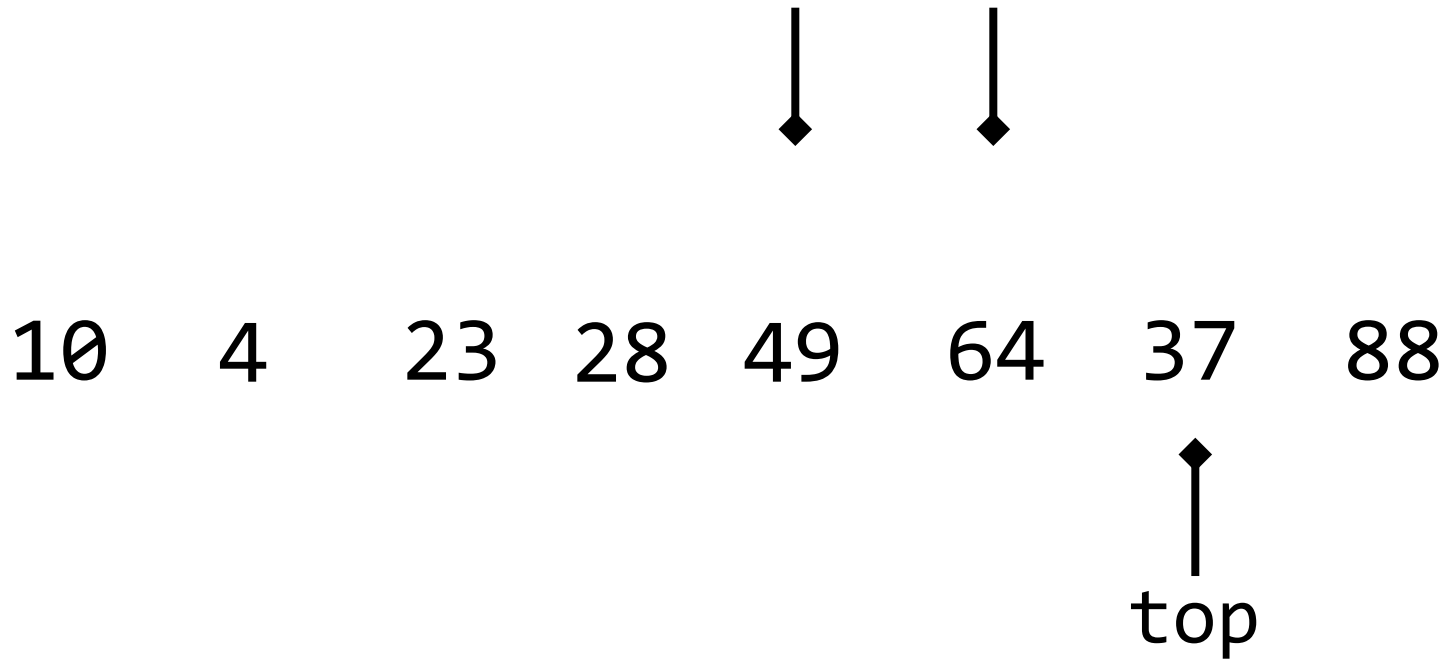

10 23 4 28 49 64 37 88
top

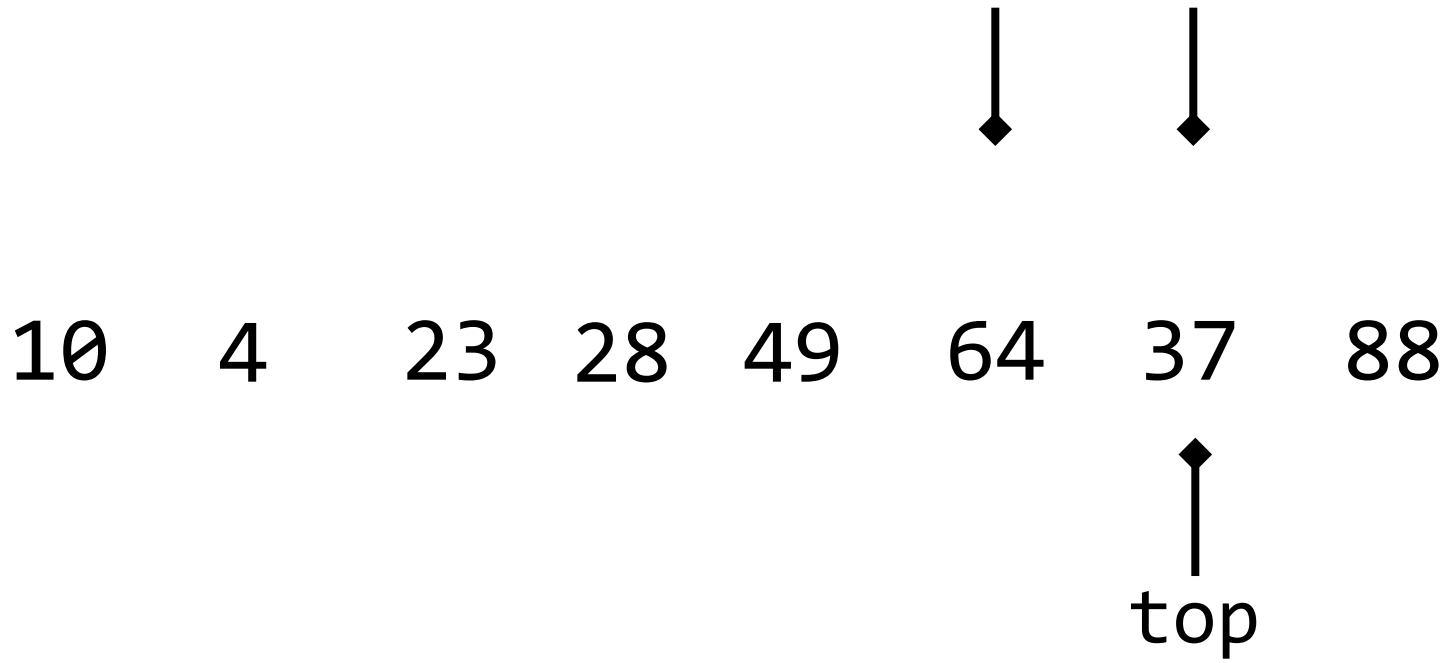

10 23 4 28 49 64 37 88
top

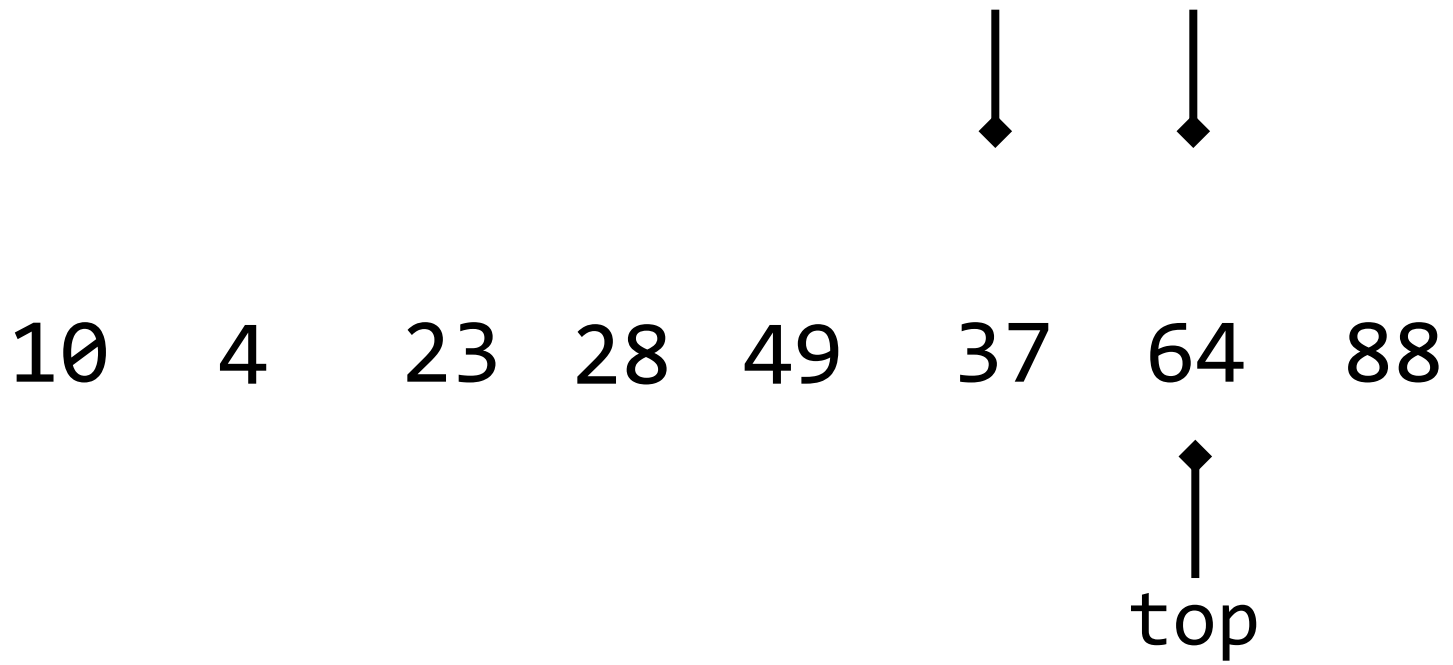

10 4 23 28 49 64 37 88
top

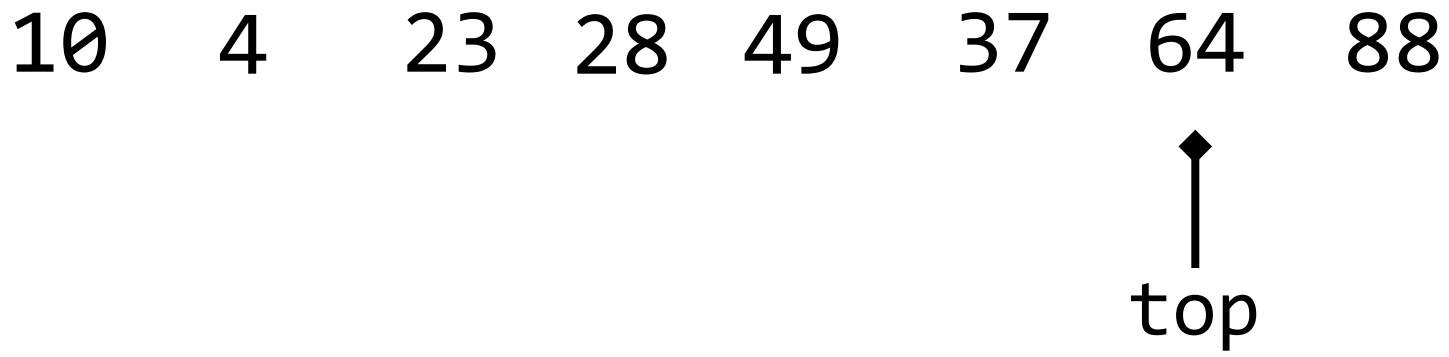

10 4 23 28 49 64 37 88
top

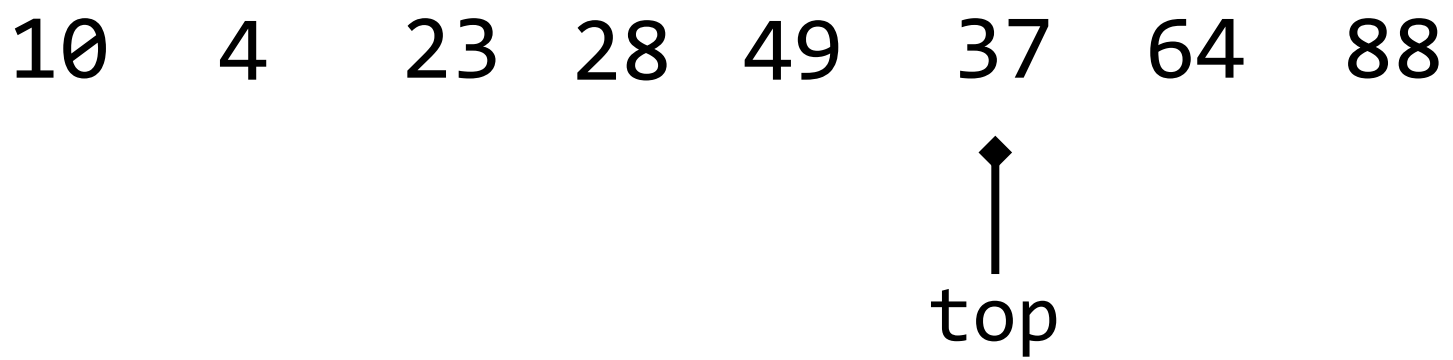

10 4 23 28 49 64 37 88
top


10 4 23 28 49 64 37 88
top


10 4 23 28 49 64 37 88
top


10 4 23 28 49 37 64 88
top


10 4 23 28 49 37 64 88
top



etc…

```
void bubbleSort(int list[], int n)
{
  bool isSorted = false;

  while (!isSorted)
  {
    isSorted = true;
    for (int i = 0; i < n - 1; i++)
    {
      if (list[i] > list[i + 1])
      {
        swap(&list[i], &list[i + 1]);
        isSorted = false;
      }
    }
  }
}
```

```
void bubbleSort(int list[], int n)
{
  bool isSorted = false;

  for ( int top = n - 1; top > 0 && !isSorted; top--)
  {
    isSorted = true;
    for (int i = 0; i < top  ; i++)
    {
      if (list[i] > list[i + 1])
      {
        swap(&list[i], &list[i + 1]);
        isSorted = false;
      }
    }
  }
}
```

Bubble Sort Performance

- Goes through the entire list, compares each item to every other item
- So it's approximately n * n = n^2 comparisons
- It does handle "nearly sorted" pretty well

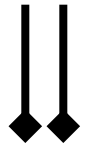
Selection Sort

- Simplest of the "selection sorts"

```
for (each position in the list)
{
    Starting at that position, search through
     the remaining list and find the smallest item.

    Swap that item to the current position.
}
```

23 49 10 4 28 88 64 37

Smallest: 23



23 49 10 4 28 88 64 37

Smallest: 23

23 49 10 4 28 88 64 37

Smallest: 10

23 49 10 4 28 88 64 37

Smallest: 4

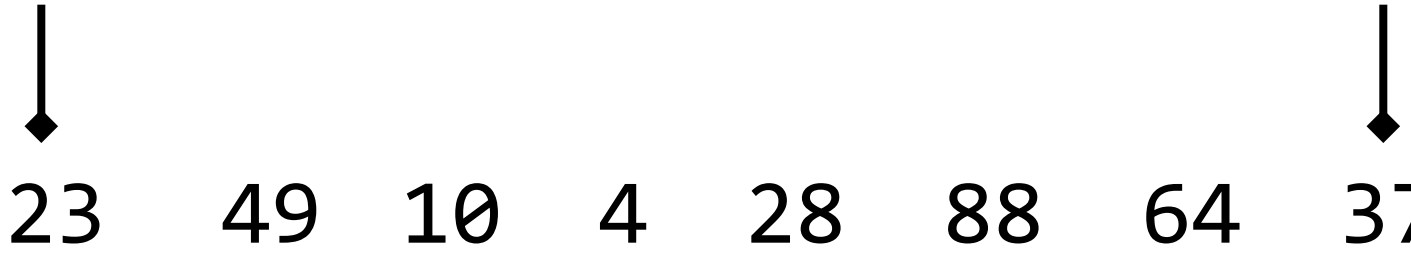
23 49 10 4 28 88 64 37

Smallest: 4

Smallest: 4

Smallest: 4

Smallest: 4



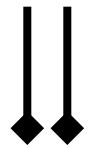
Smallest: 4

23 49 10 4 28 88 64 37

Smallest:

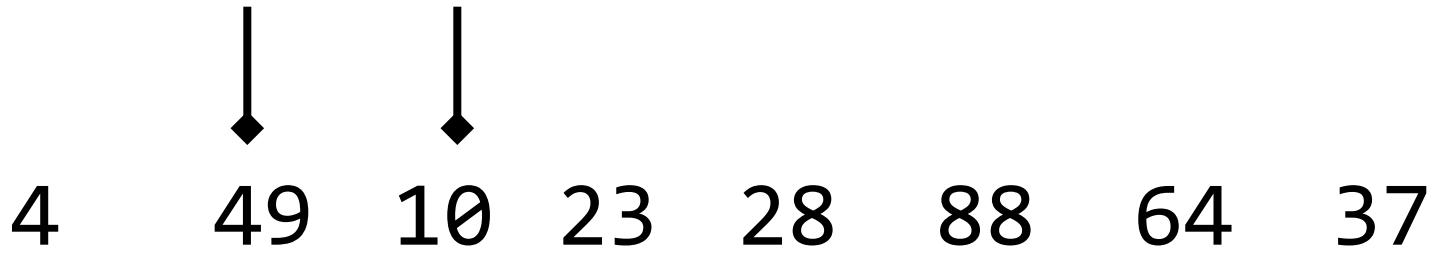
4 49 10 23 28 88 64 37

Smallest: 49

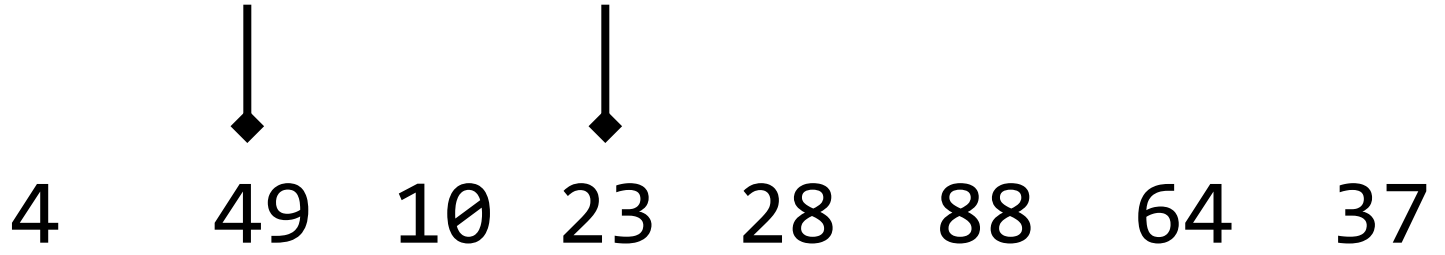


4 49 10 23 28 88 64 37

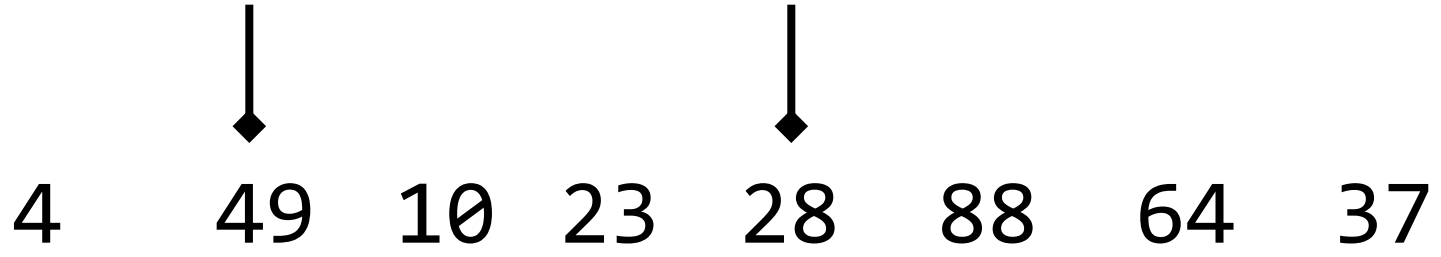
Smallest: 10



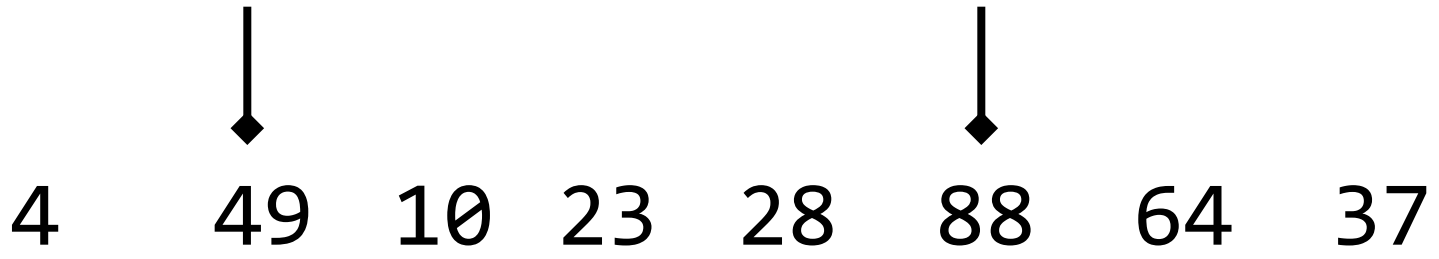
Smallest: 10



Smallest: 10

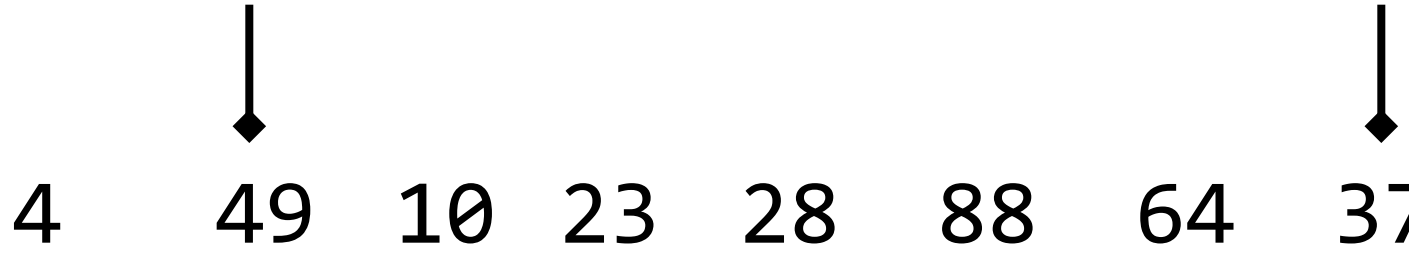


Smallest: 10



Smallest: 10

Smallest: 10



Smallest: 10

4 49 10 23 28 88 64 37

Smallest: 10

4 10 49 23 28 88 64 37

Smallest:

4 10 49 23 28 88 64 37

etc...

```
void selectionSort(int list[], int n)
{
   for (int bot = 0; bot < n; bot++)
   {
      int indexOfSmallest = bot;
      for (int i = bot + 1; i < n; i++)
      {
         if (list[i] < list[indexOfSmallest])
         {
            indexOfSmallest = i;
         }
      }

      swap(&list[bot], &list[indexOfSmallest]);
   }
}
```

Selection Sort Performance

- Doesn't adapt; always goes through the entire list each time
- Goes through the entire list, compares each item to every other item
- So it's approximately n * n = n^2 comparisons
- Minimizes the number of swaps (occasionally useful)

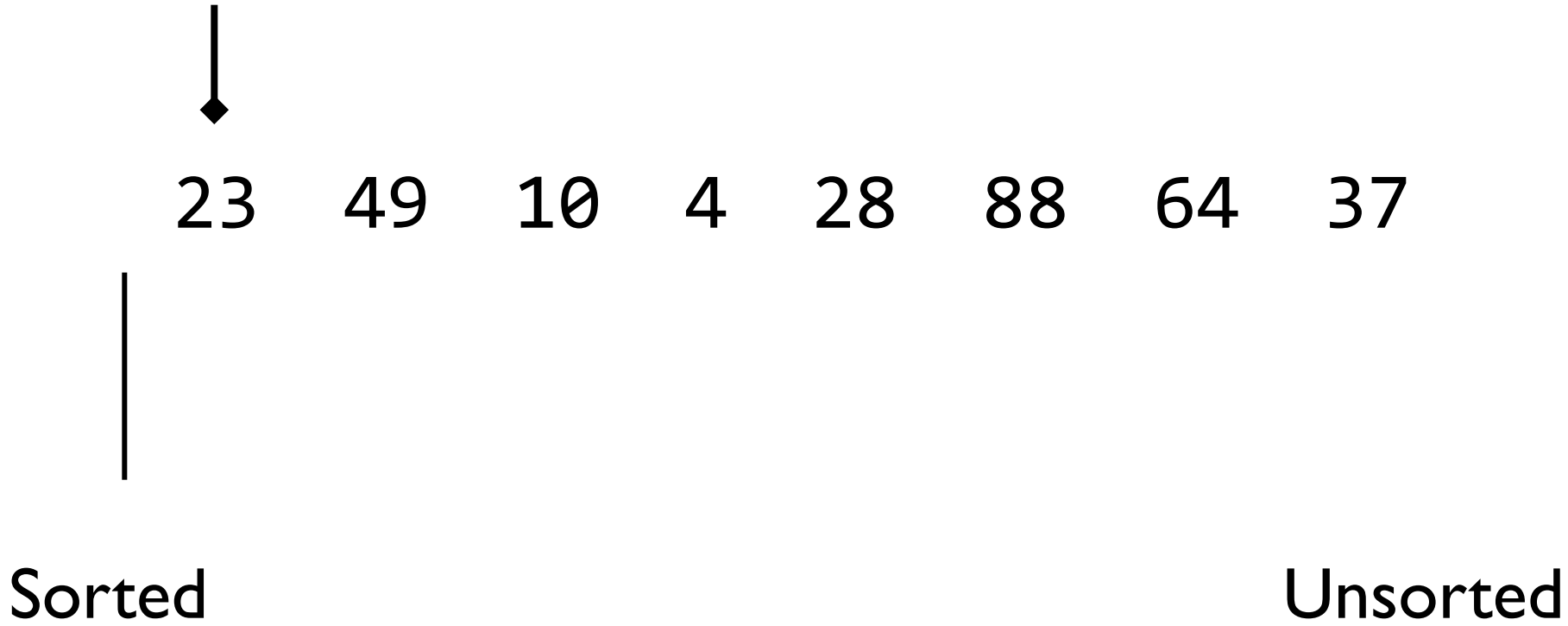
Insertion Sort

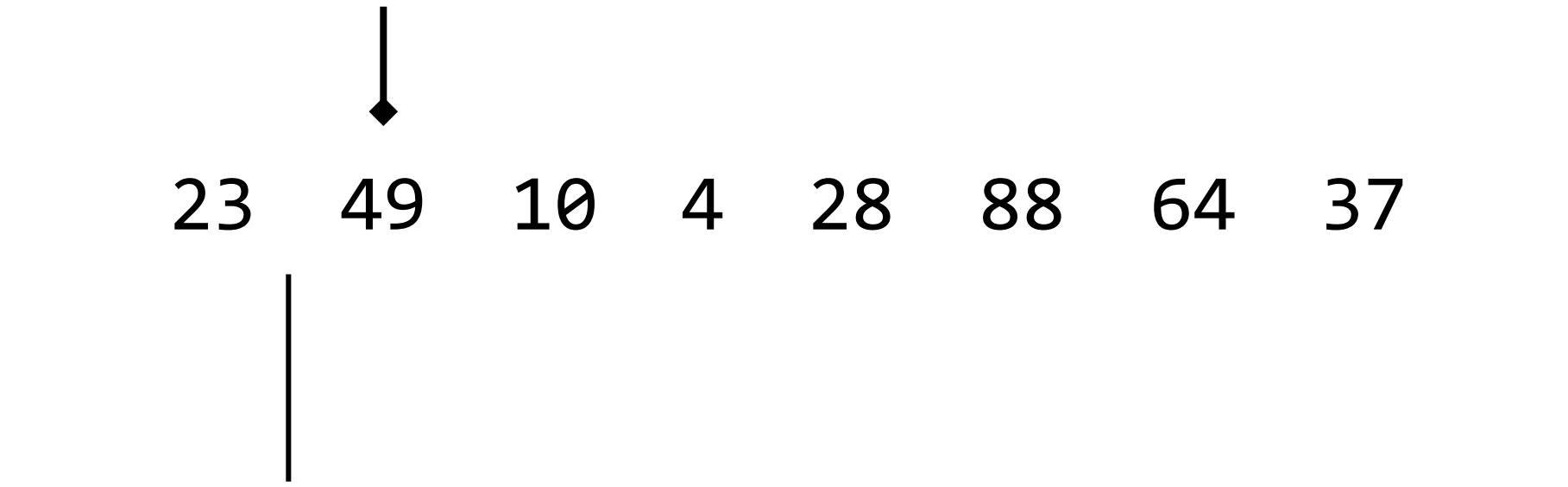
- Simplest of the "insertion sorts"
- This is probably the sort algorithm most commonly used by normal people for day-to-day tasks

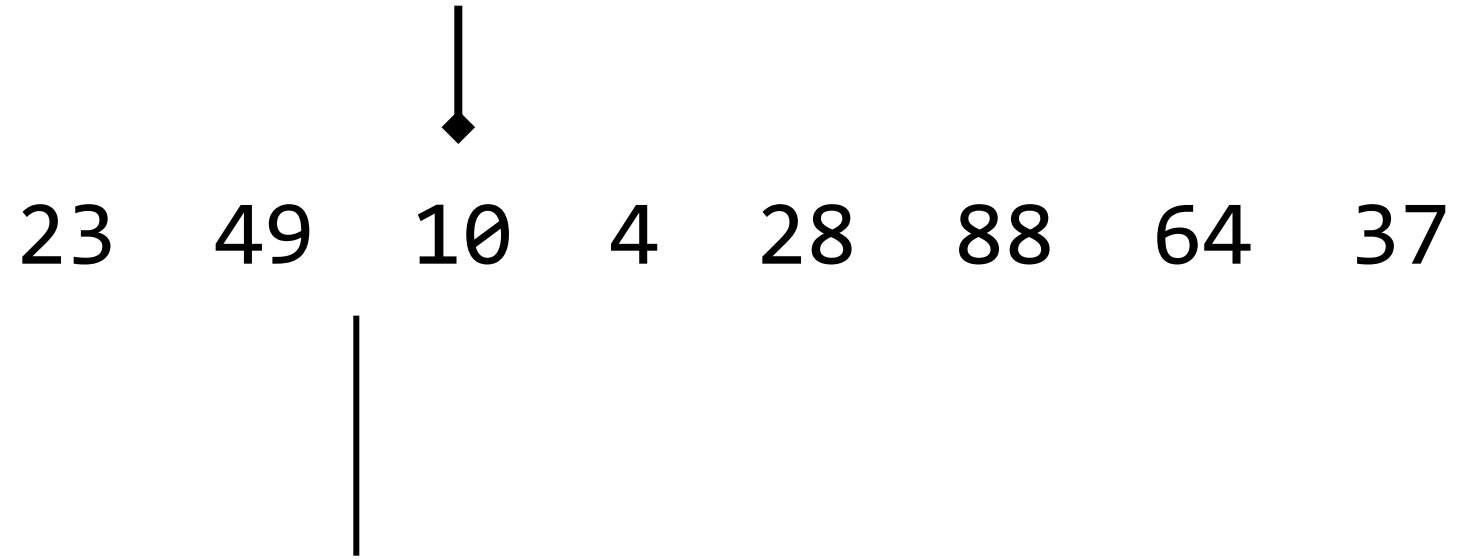
Divide the list into two parts, a "sorted" part and an "unsorted" part. (Right now, "sorted" is empty, and "unsorted" contains the entire list.)

```
for (each item in the unsorted list)
{
    Insert it in the appropriate place in the sorted list
}
```

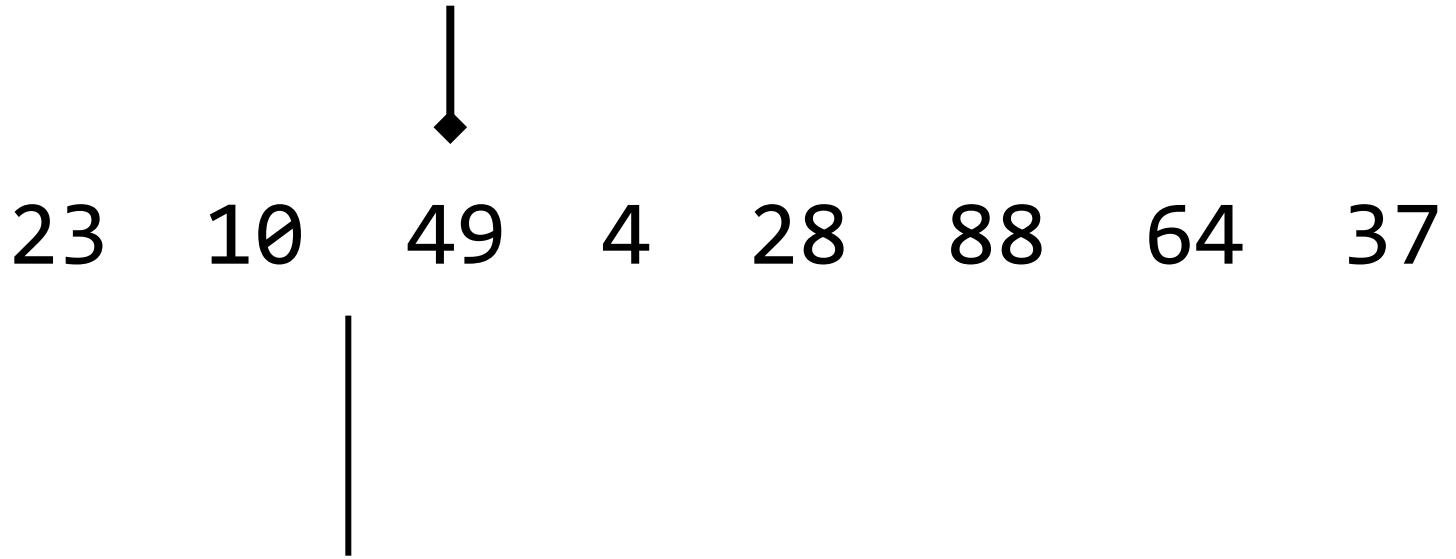

23 49 10 4 28 88 64 37
Sorted
Unsorted


23 49 10 4 28 88 64 37
Sorted
Unsorted


23 49 10 4 28 88 64 37
Sorted
Unsorted

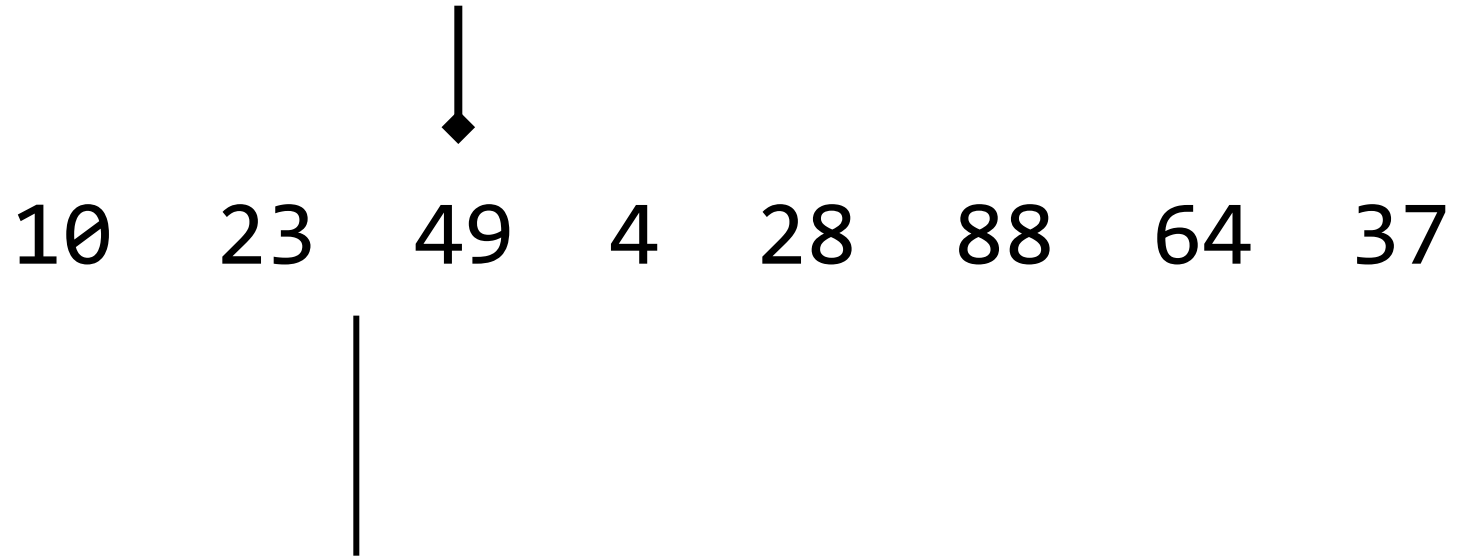


Sorted Unsorted

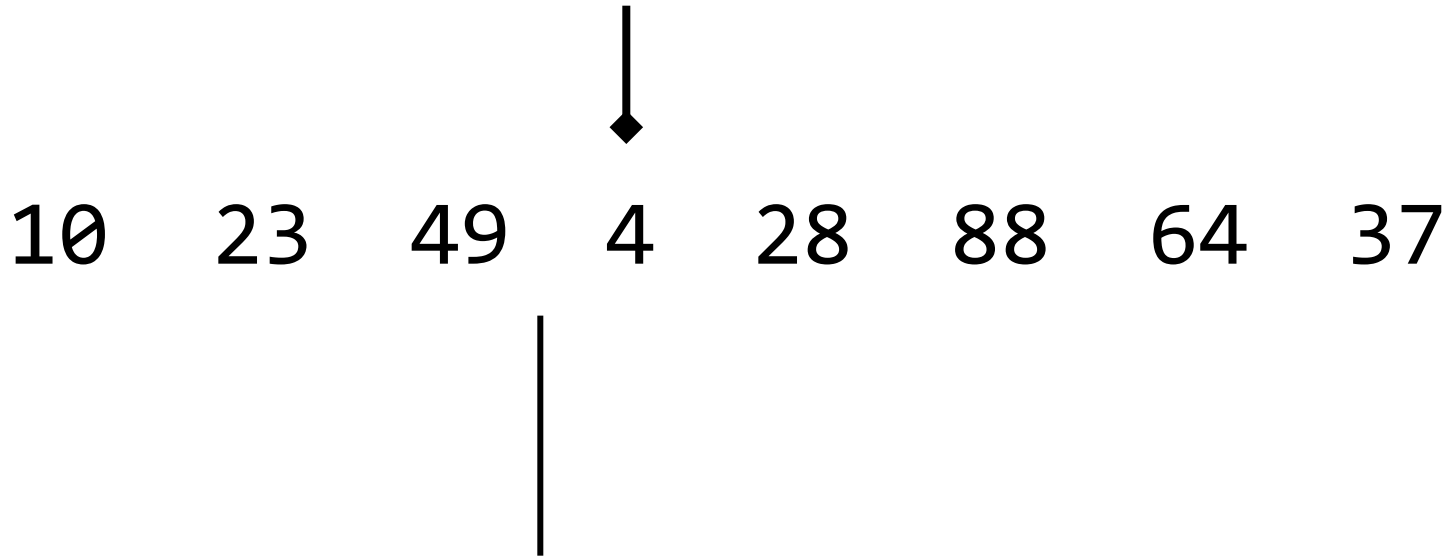


Sorted

Unsorted

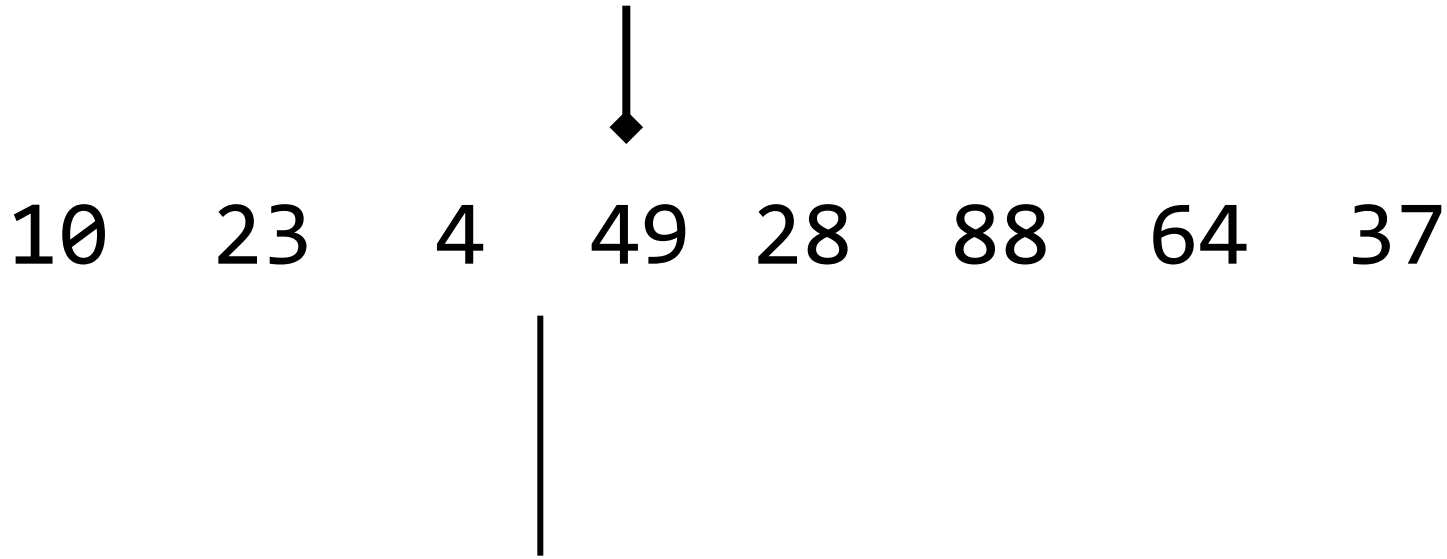


Sorted Unsorted



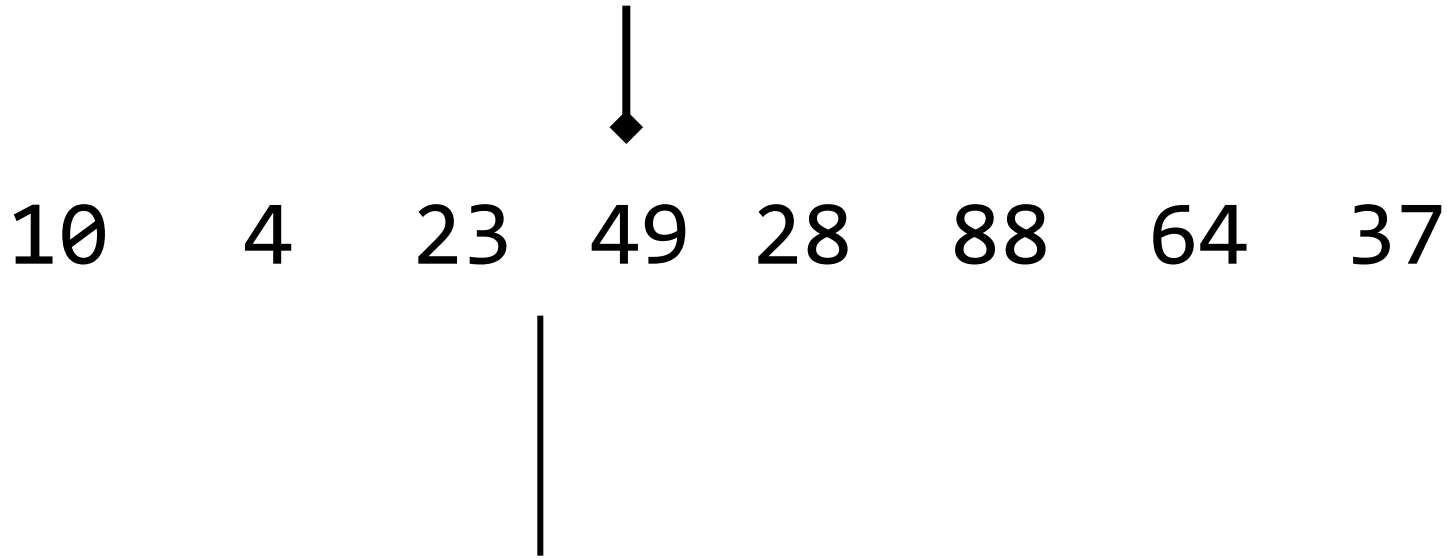
Sorted

Unsorted



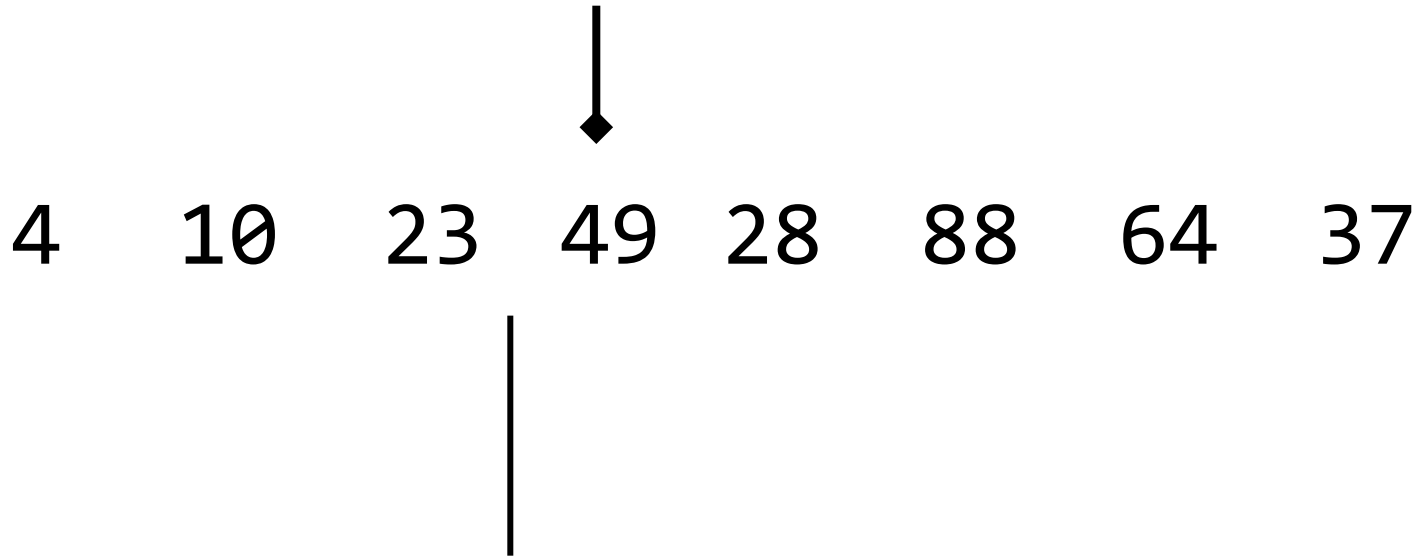
Sorted

Unsorted



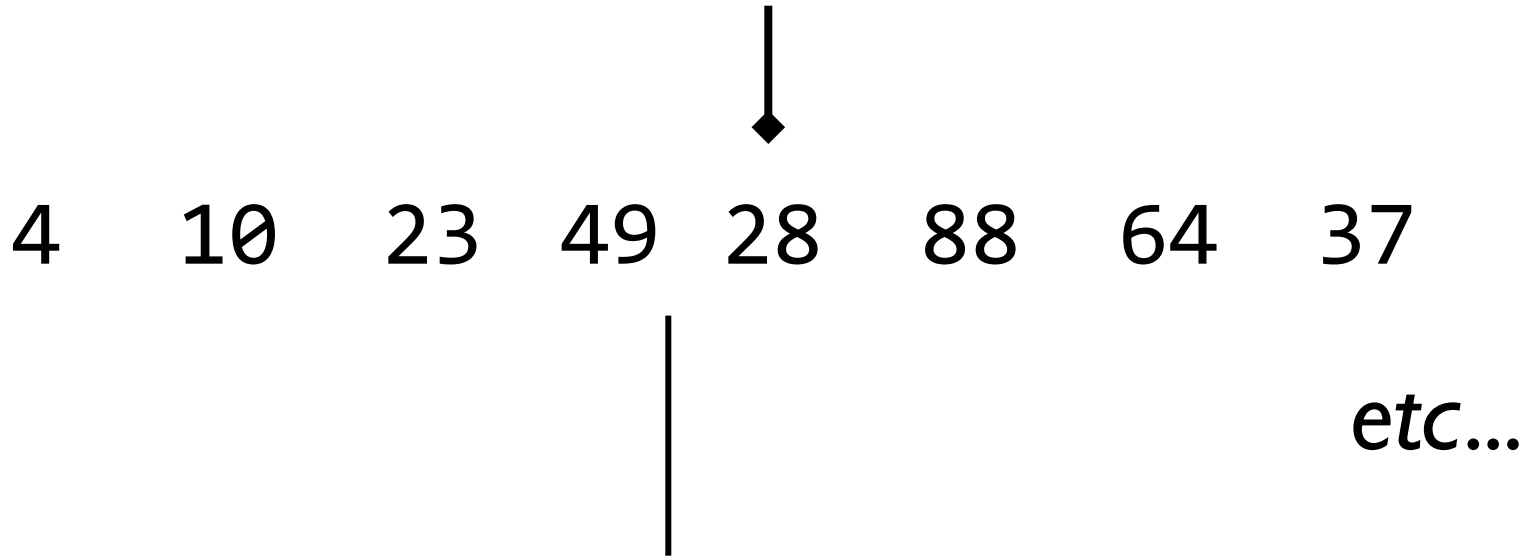
Sorted

Unsorted



Sorted

Unsorted



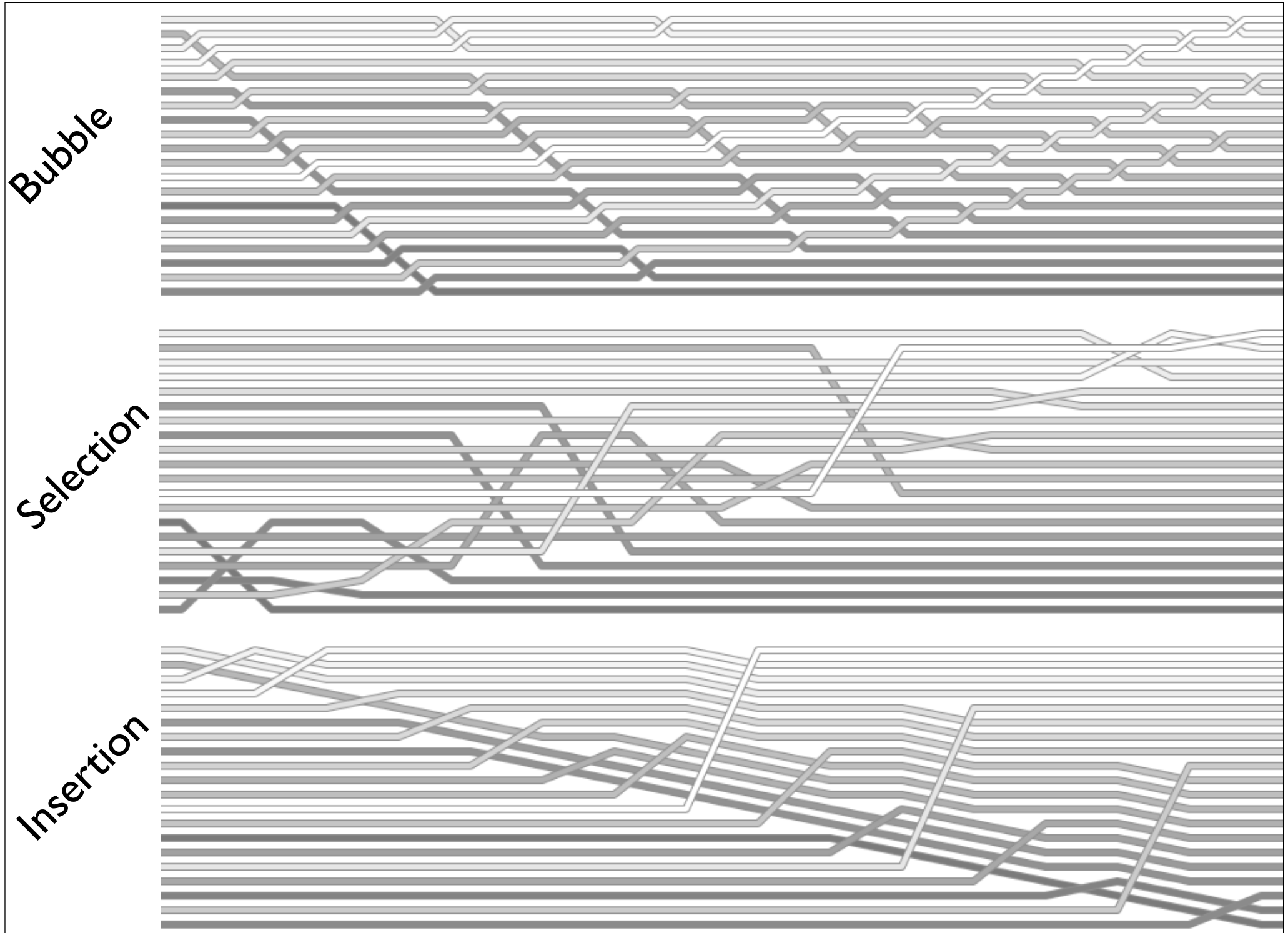
Sorted

Unsorted

```
void insertionSort(int list[], int n)
{
  for (int bot = 1; bot < n; bot++)
  {
    for (int i = bot; i > 0 && list[i] < list[i-1]; i--)
    {
      swap(&list[i], &list[i - 1]);
    }
  }
}
```

Insertion Sort Performance

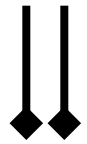
- Goes through the entire list, compares each item to every other item
- So (again) it's approximately $n * n = n^2$ comparisons worst case
- Closer to n comparisons if the list is almost sorted



http://sortvis.org/

23 49 10 4 28 88 64 37

Smallest: 23



23 49 10 4 28 88 64 37

Smallest: 23

23 49 10 4 28 88 64 37

Smallest: 10

23 49 10 4 28 88 64 37

Smallest: 4

23 49 10 4 28 88 64 37

Smallest: 4

23 49 10 4 28 88 64 37

Smallest: 4

23 49 10 4 28 88 64 37

Smallest: 4

```
23   49  10   4   28   88   64   37
```

Smallest: 4

23 49 10 4 28 88 64 37

Smallest: 4

4 49 10 23 28 88 64 37

```
4   10   49 23  28   88   64   37
```

If we compare every element against every other element:

$$n + n + n + n + n + n + n + n$$

$$n * n = n^2 \qquad\qquad n = 8$$

If we compare every element against remaining elements:

$$n + (n-1) + (n-2) + (n-3) + (n-4) + (n-5) + (n-6) + (n-7)$$

$$n + (n-1) + (n-2) + (n-3) + (n-4) + (n-5) + (n-6) + (n-7)$$

$$n + n + n + n + n + n + n + n - 1 - 2 - 3 - 4 - 5 - 6 - 7$$

$$n * n - 1 - 2 - 3 - 4 - 5 - 6 - 7$$

$$n * n - \sum_{i=1}^{7} i$$

Actual number of comparisons

$$n * n - \sum_{i=1}^{n-1} i$$


a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
0
1
3
5
4
6
9


a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]
3
1
2
6
0
4
9
6


a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]
0
1
7
8
5
4
9
6

Sorting Dances

Dances for several algorithms are available at:

http://www.youtube.com/user/AlgoRythmics

Merge Sort

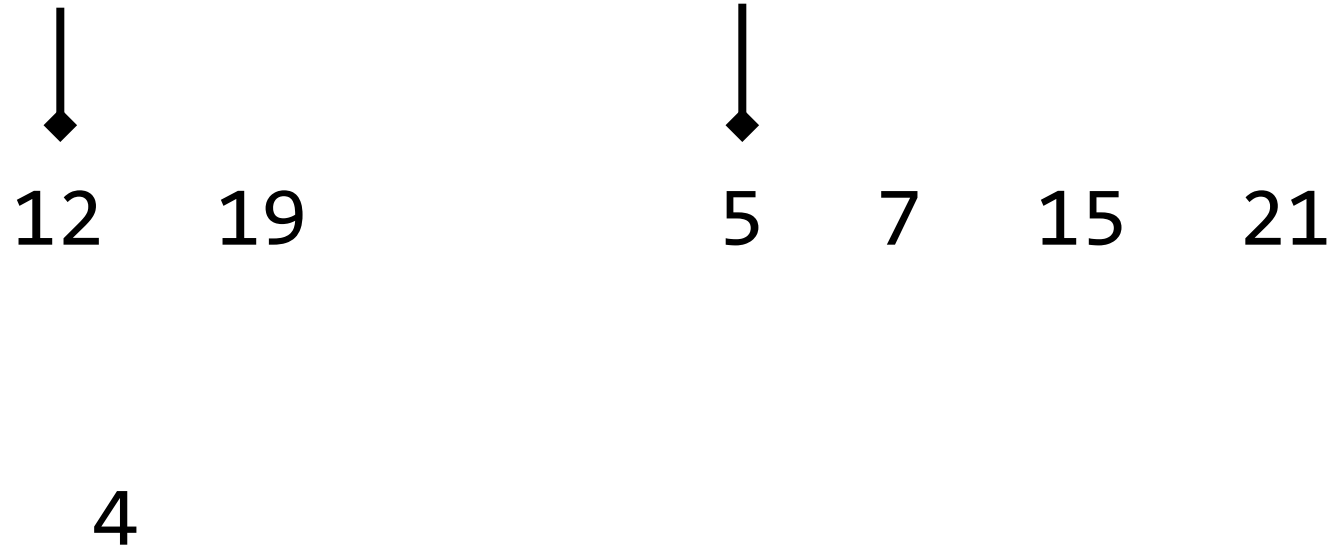
- Recursive sort built on idea of merging

Merge Sort

- Recursive sort built on idea of merging

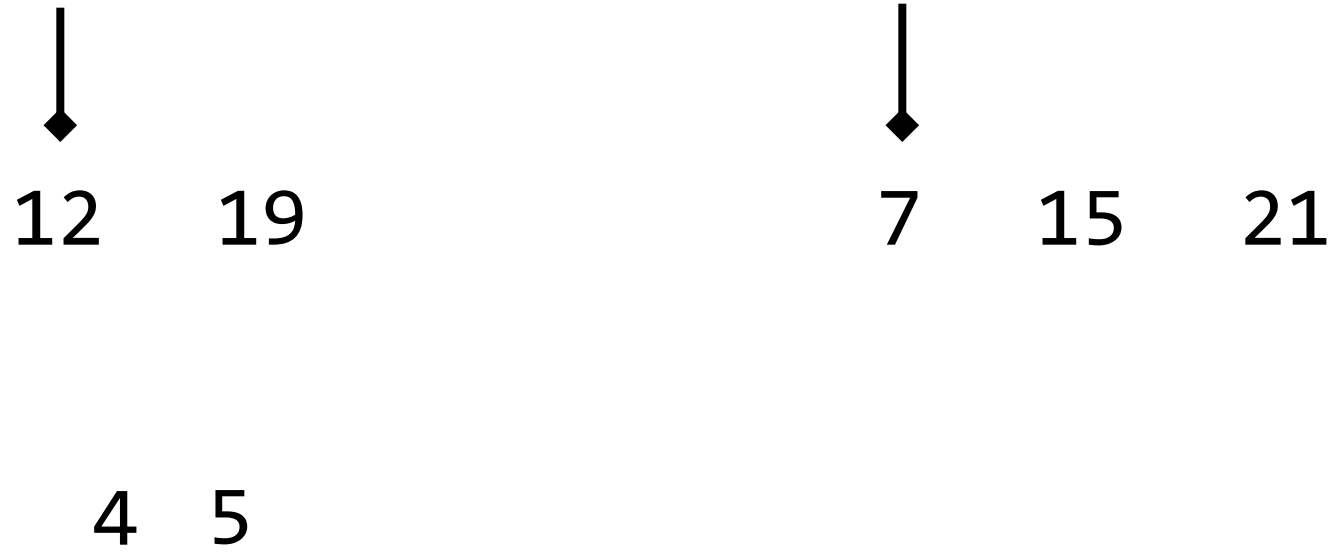
Merge Sort

- Recursive sort built on idea of merging



Merge Sort

- Recursive sort built on idea of merging



Merge Sort

- Recursive sort built on idea of merging

Merge Sort

- Recursive sort built on idea of merging

4 5 7 12

Merge Sort

- Recursive sort built on idea of merging

4 5 7 12 15

Merge Sort

- Recursive sort built on idea of merging

4 5 7 12 15 19

Merge Sort

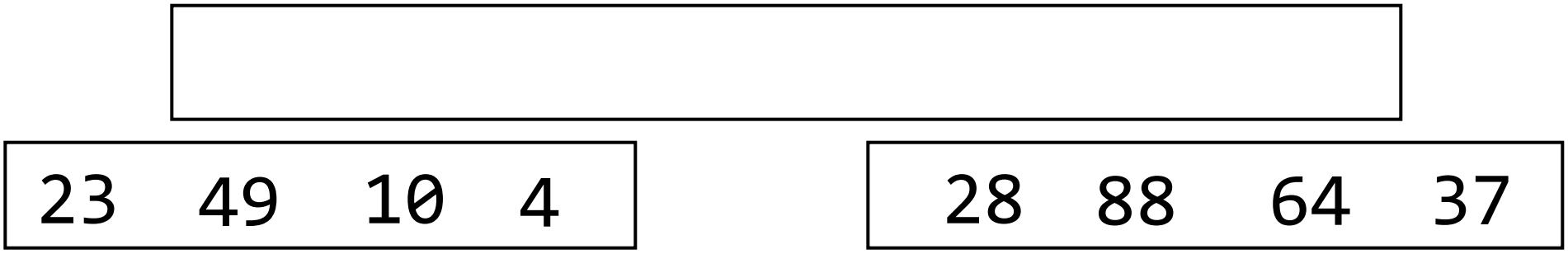
- Recursive sort built on idea of merging

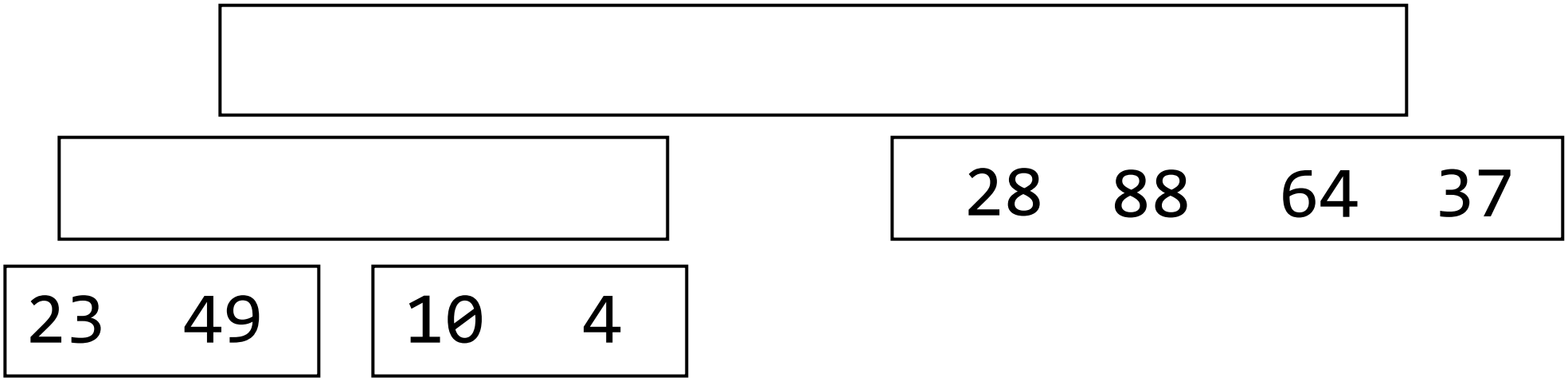
4 5 7 12 15 19 21

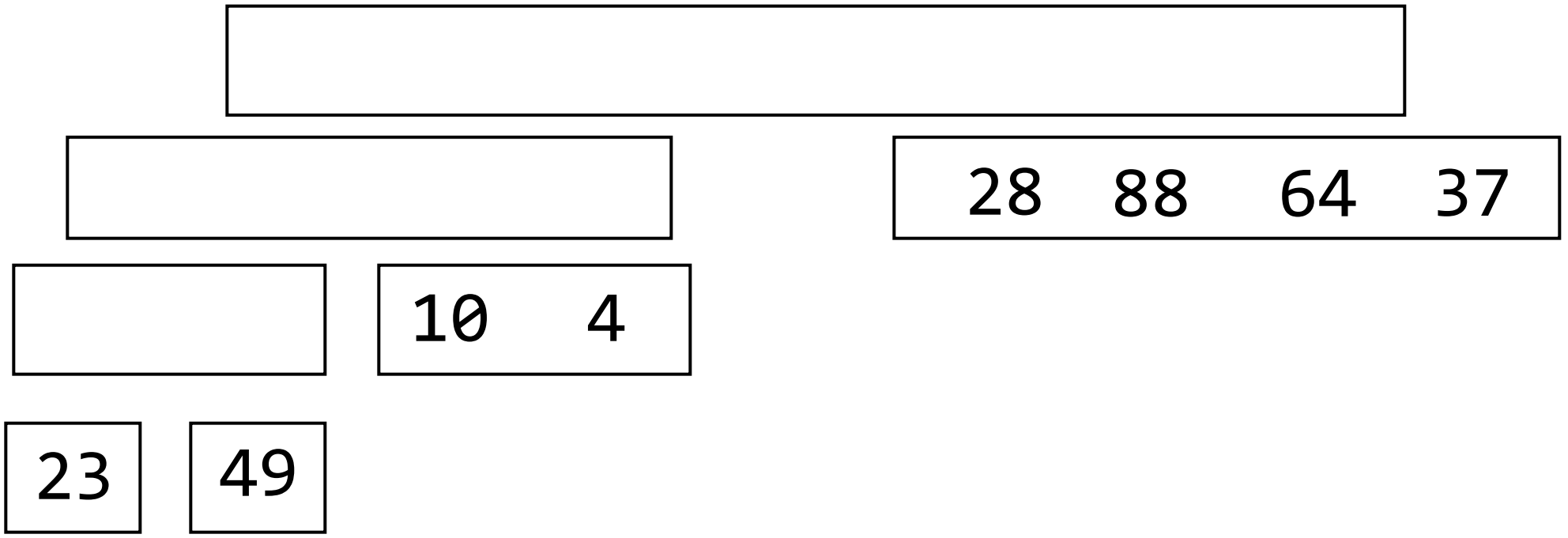
Merge Sort

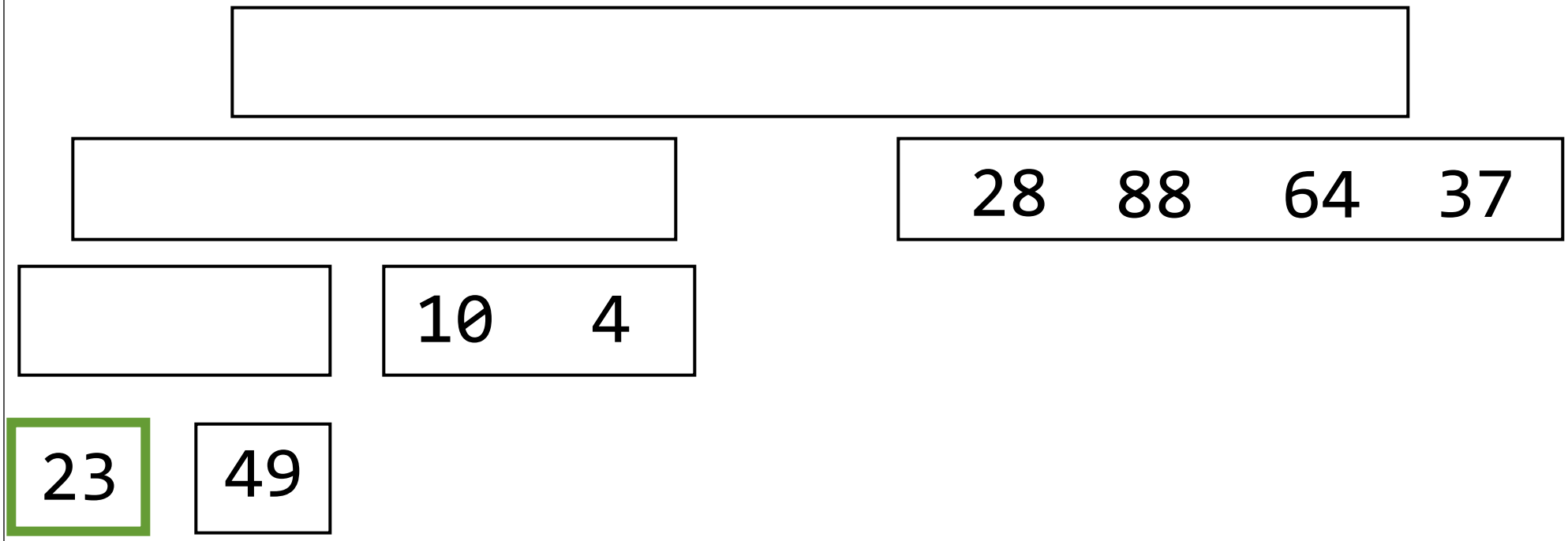
```
if (the list is of length 0 or 1)
{
   We're done.
}
else
{
   Divide the list into two halves.
   Merge sort the first half.
   Merge sort the second half.
   Merge the two sorted sublists.
}
```

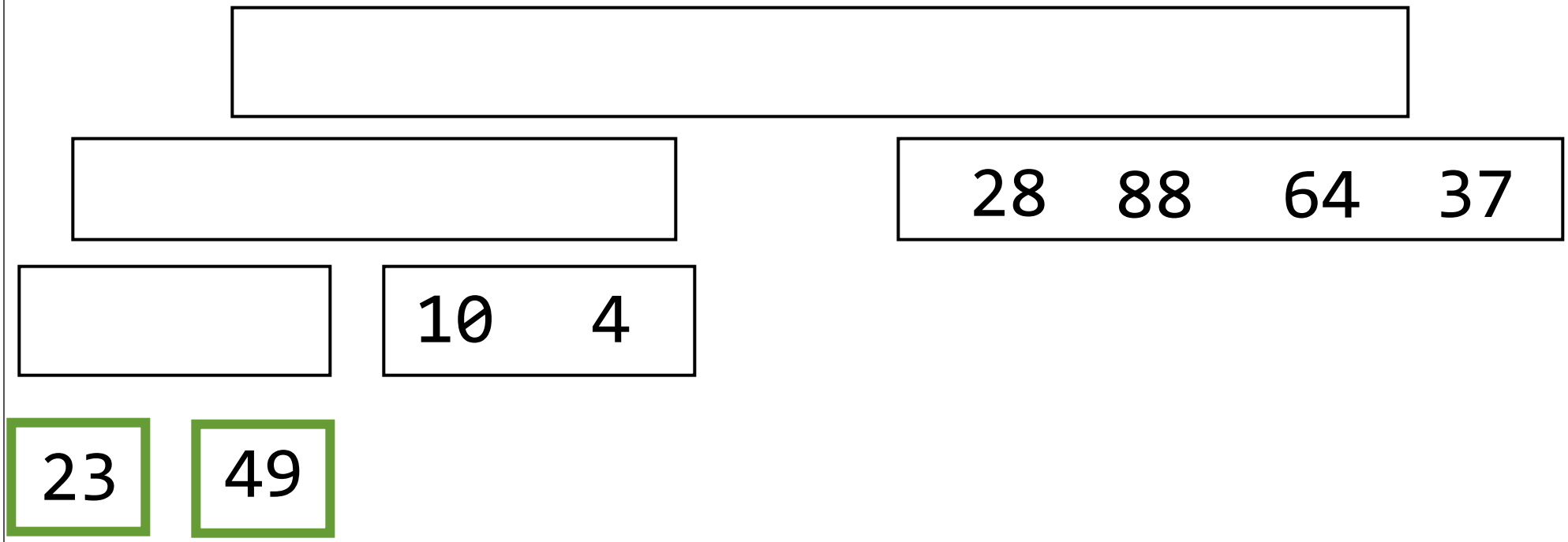
23	49	10	4	28	88	64	37

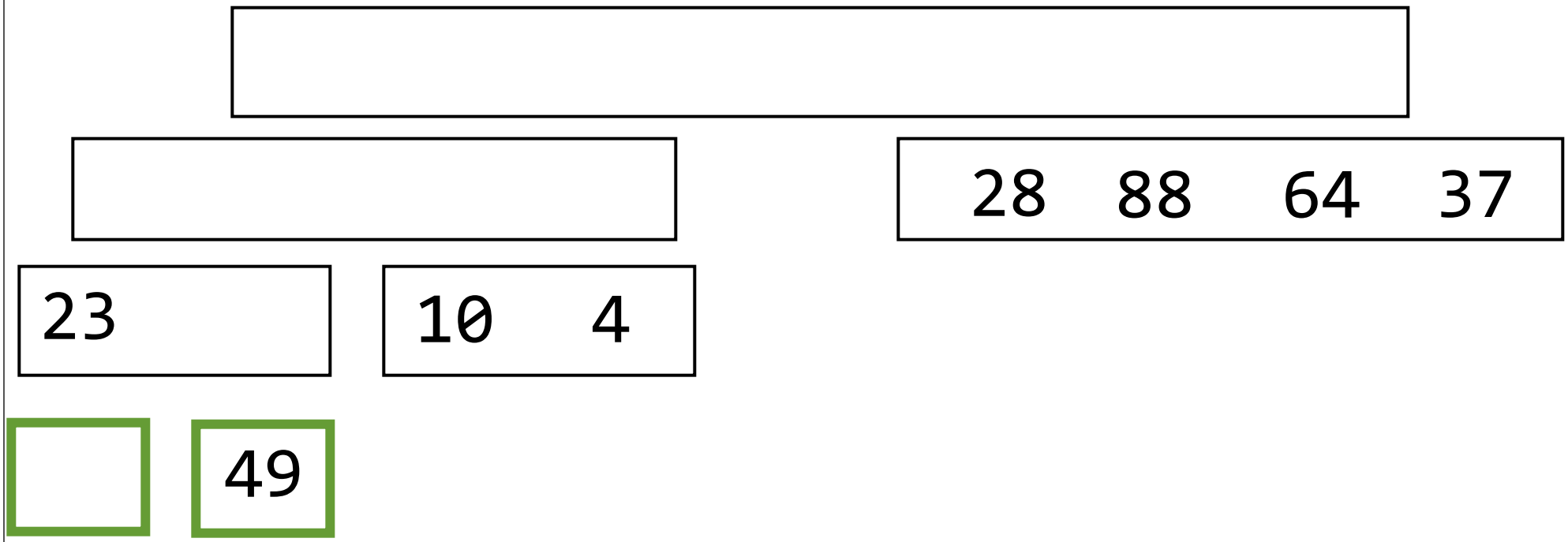

23 49 10 4
28 88 64 37

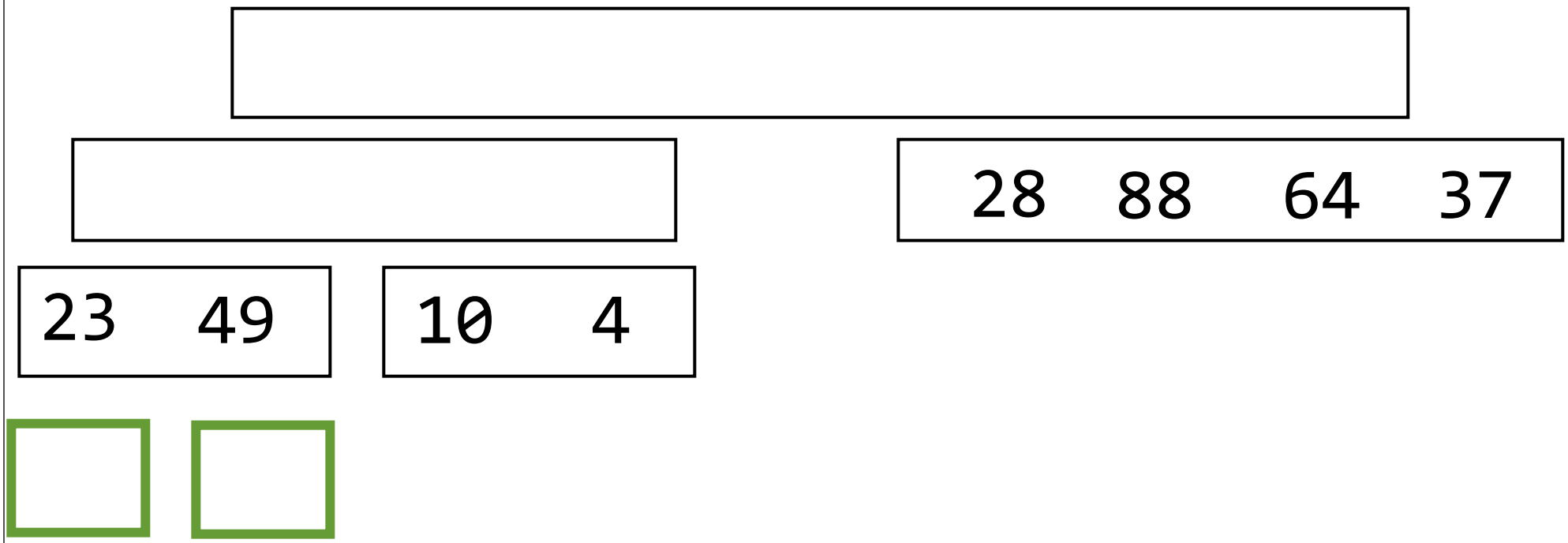

28 88 64 37
23 49
10 4

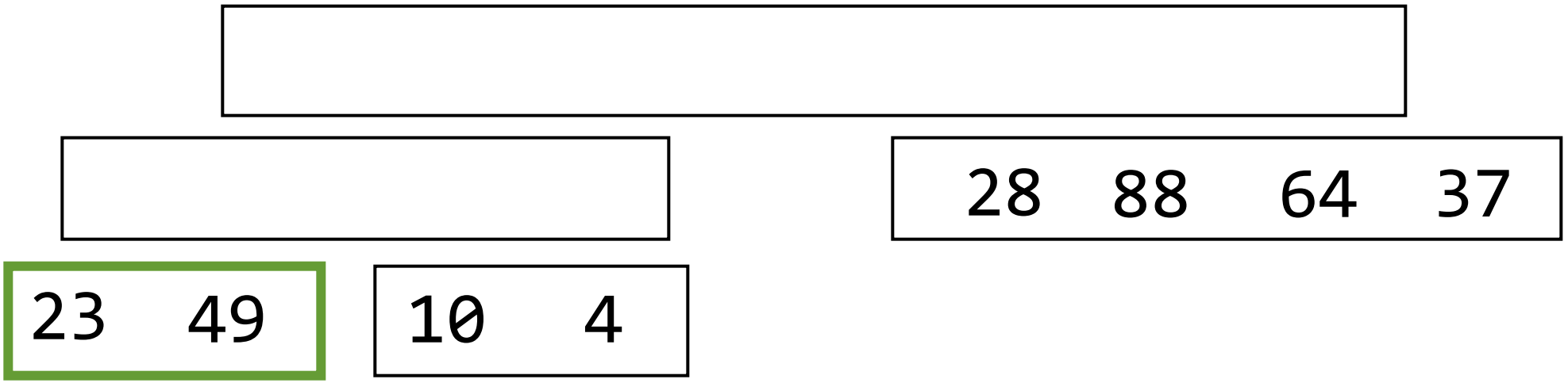

28 88 64 37
10 4
23
49

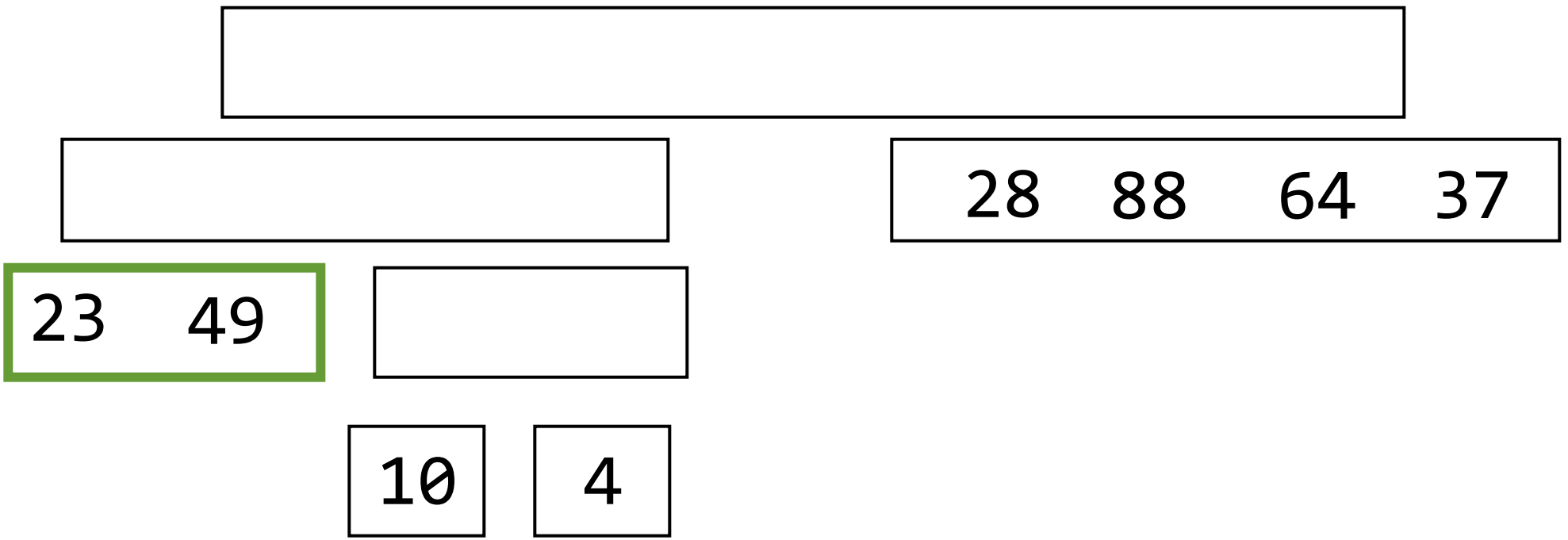

28 88 64 37
10 4
23
49

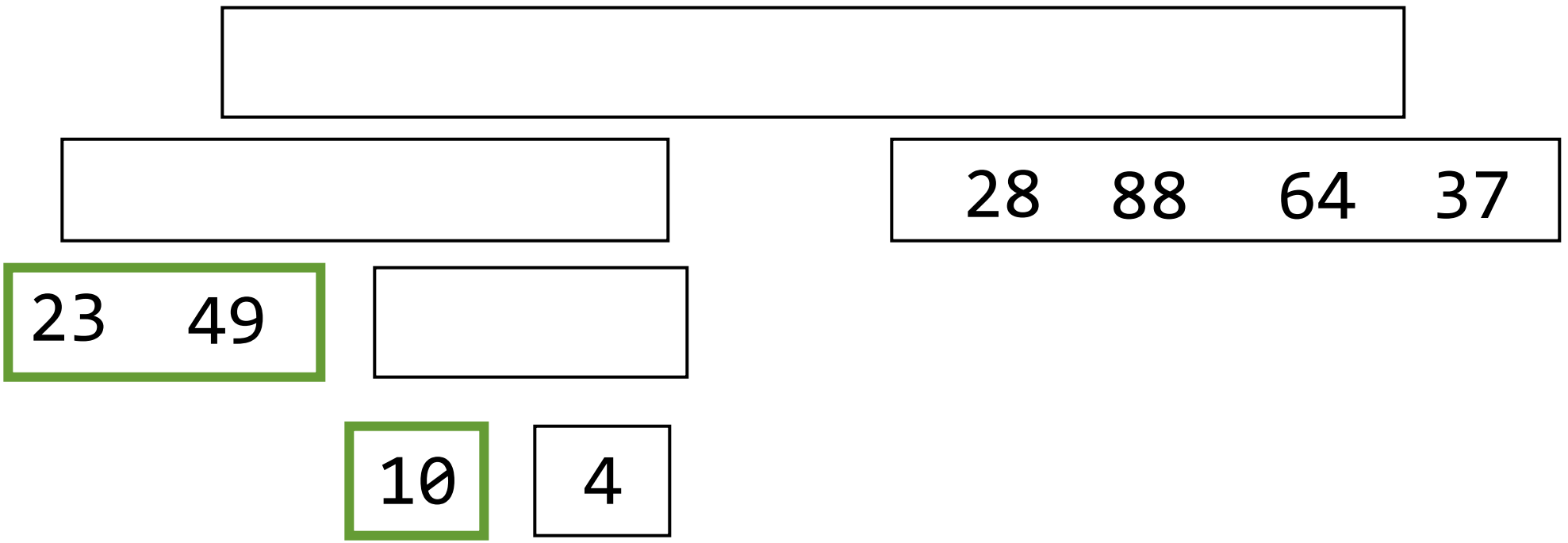

28 88 64 37
10 4
23
49

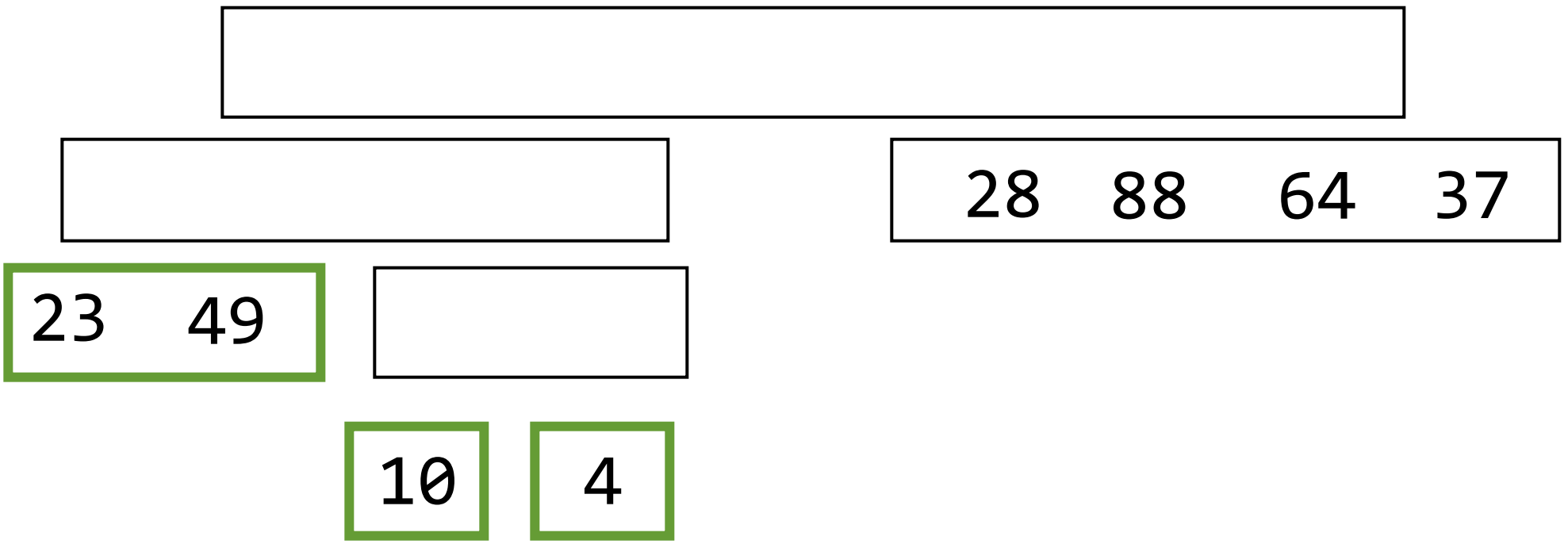

28 88 64 37
23
10 4
49

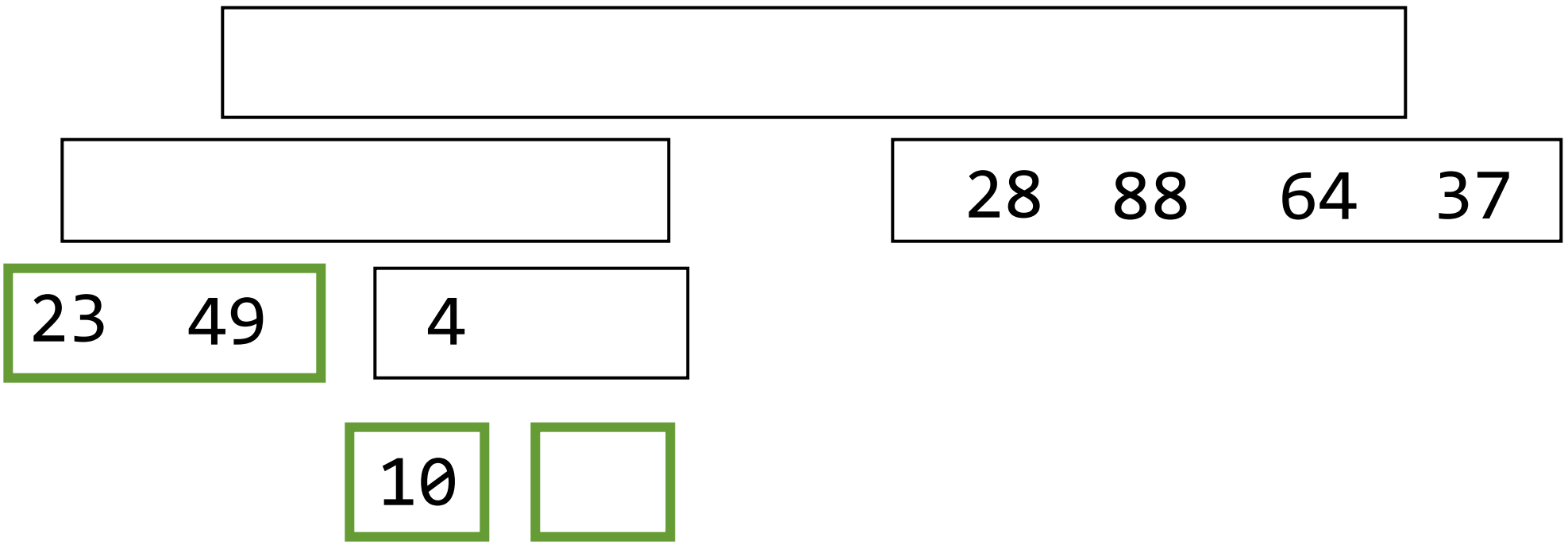

28 88 64 37
23 49
10 4

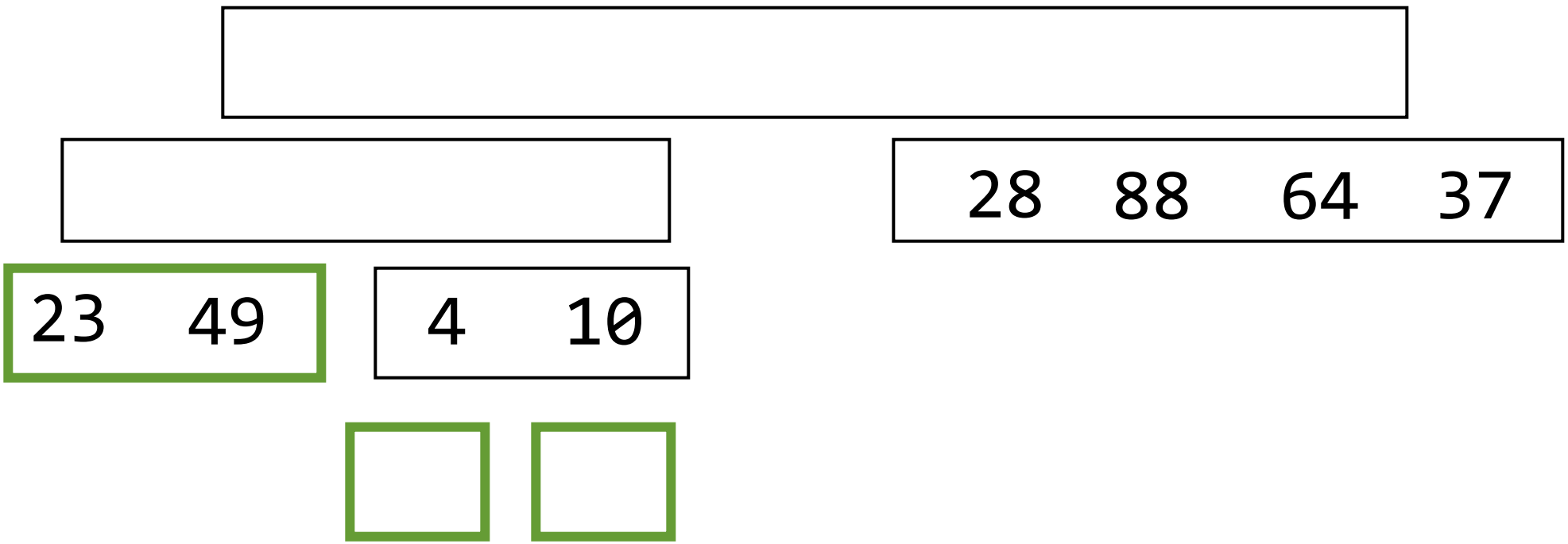

28 88 64 37
23 49
10 4

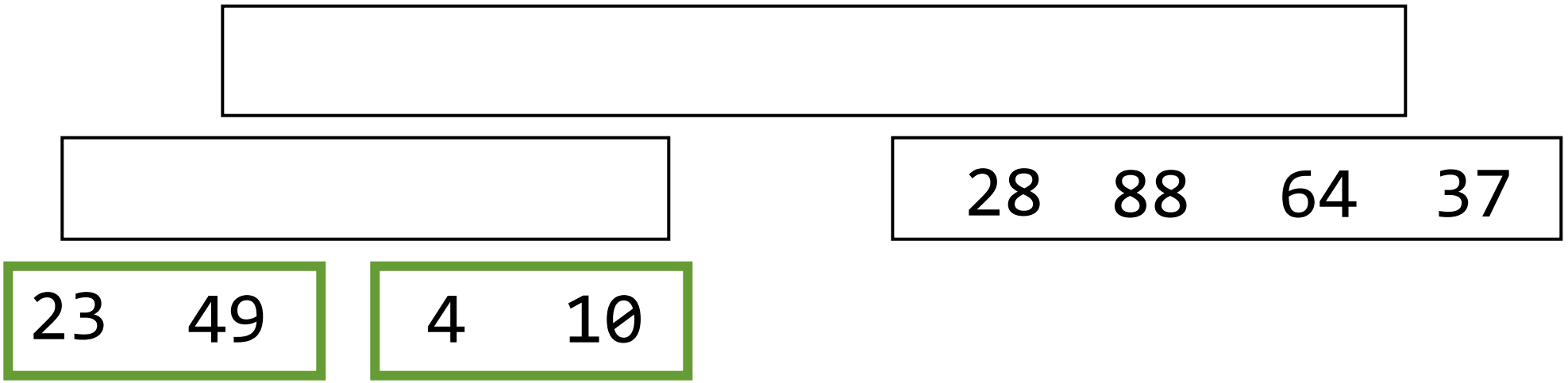

28 88 64 37
23 49
10
4

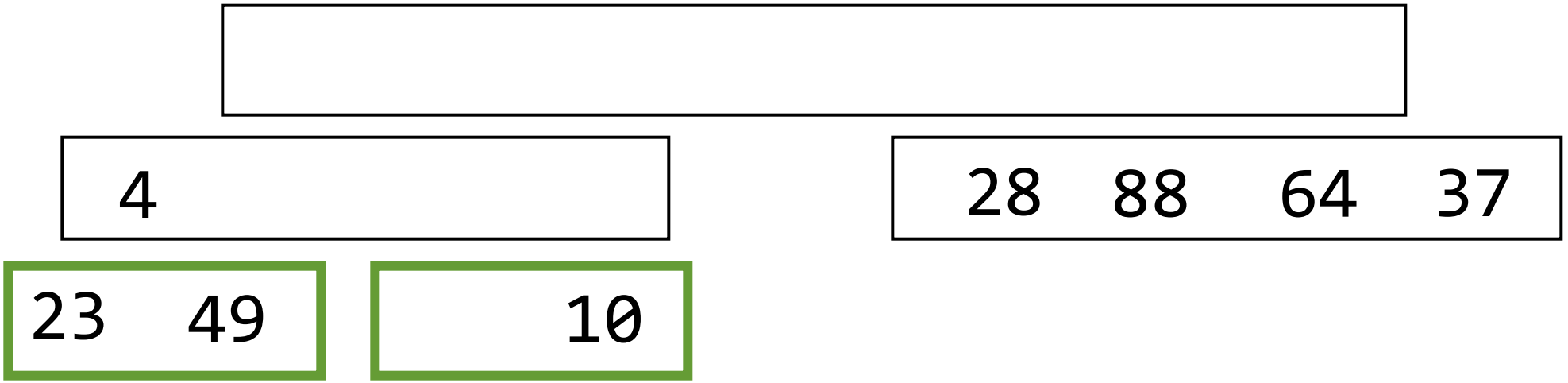

28 88 64 37
23 49
10 4

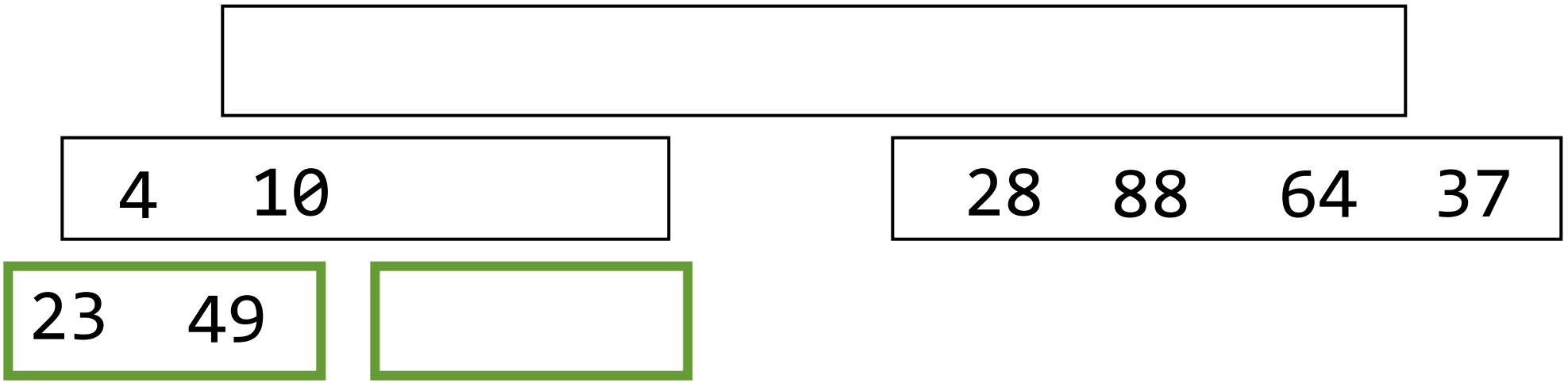

28 88 64 37
23 49
10 4

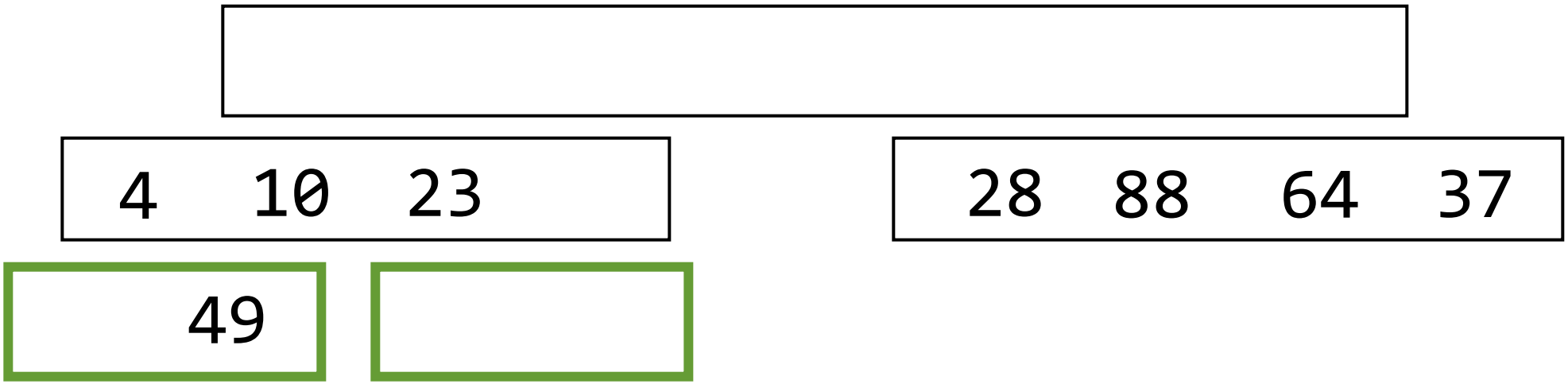

28 88 64 37
23 49
4
10

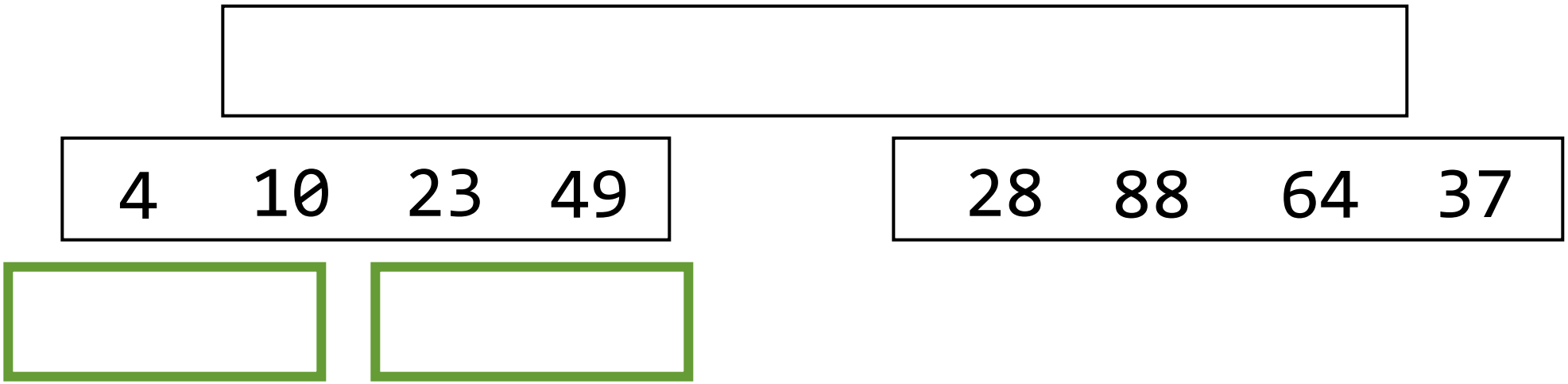

28 88 64 37
23 49
4 10

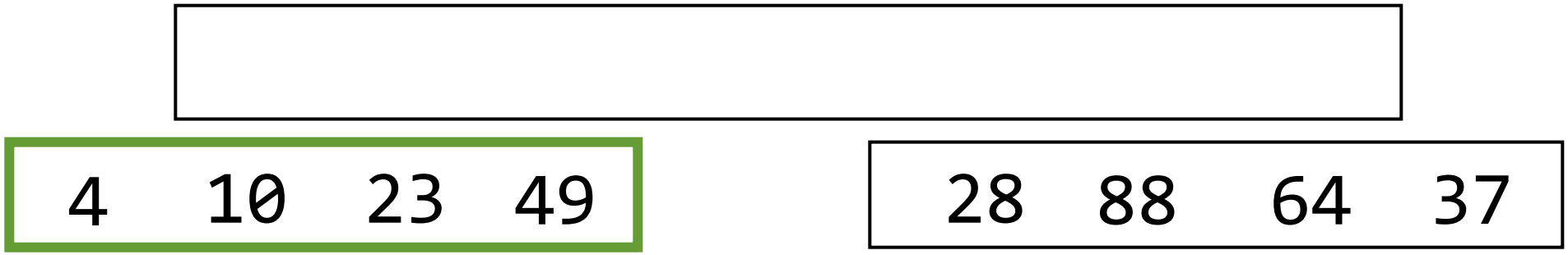

28 88 64 37
23 49
4 10

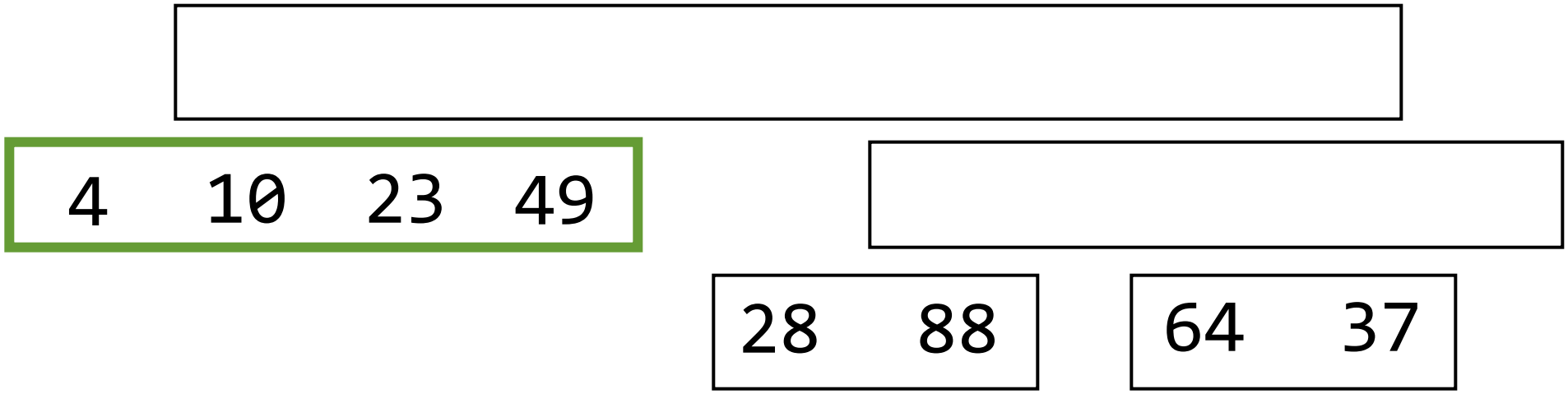

4
28 88 64 37
23 49
10

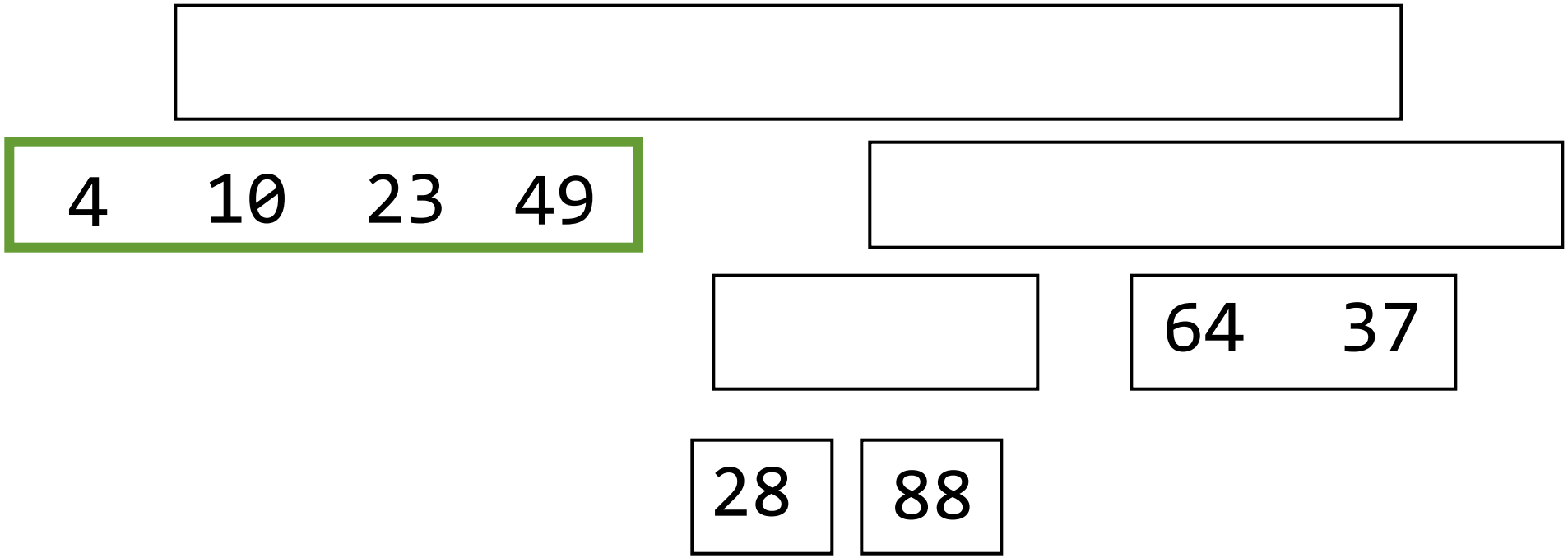

4 10
28 88 64 37
23 49

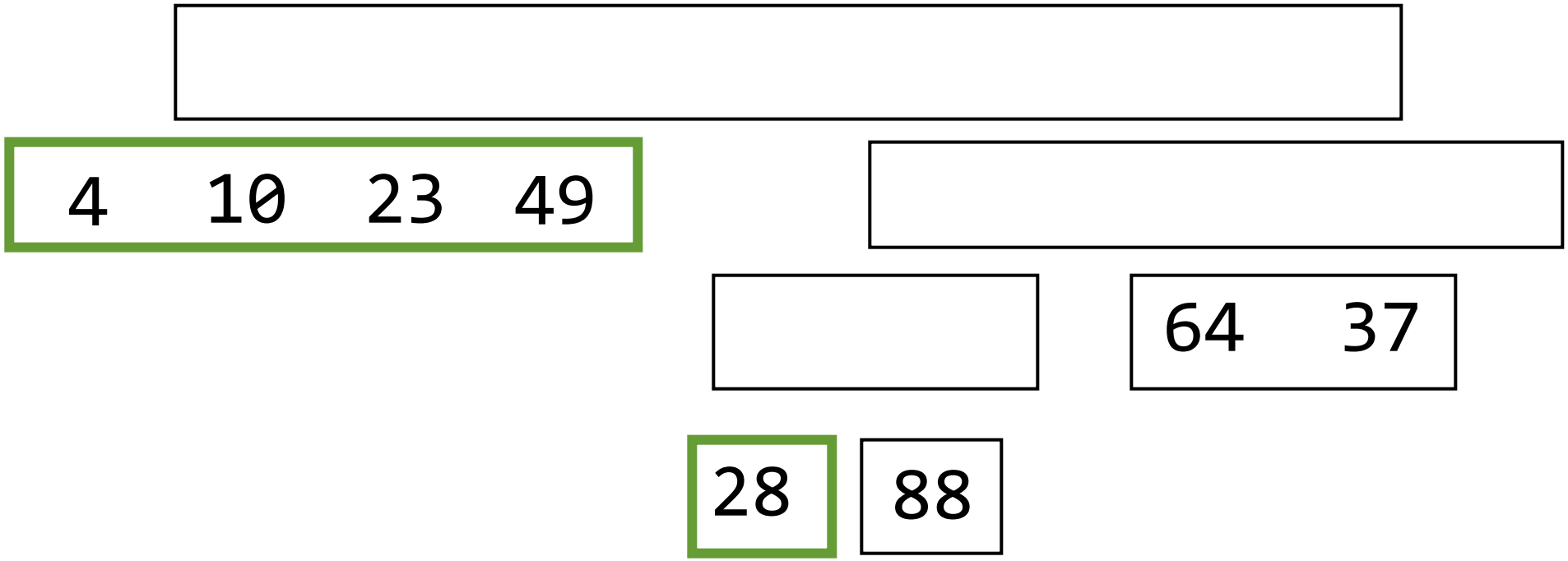

4 10 23
28 88 64 37
49

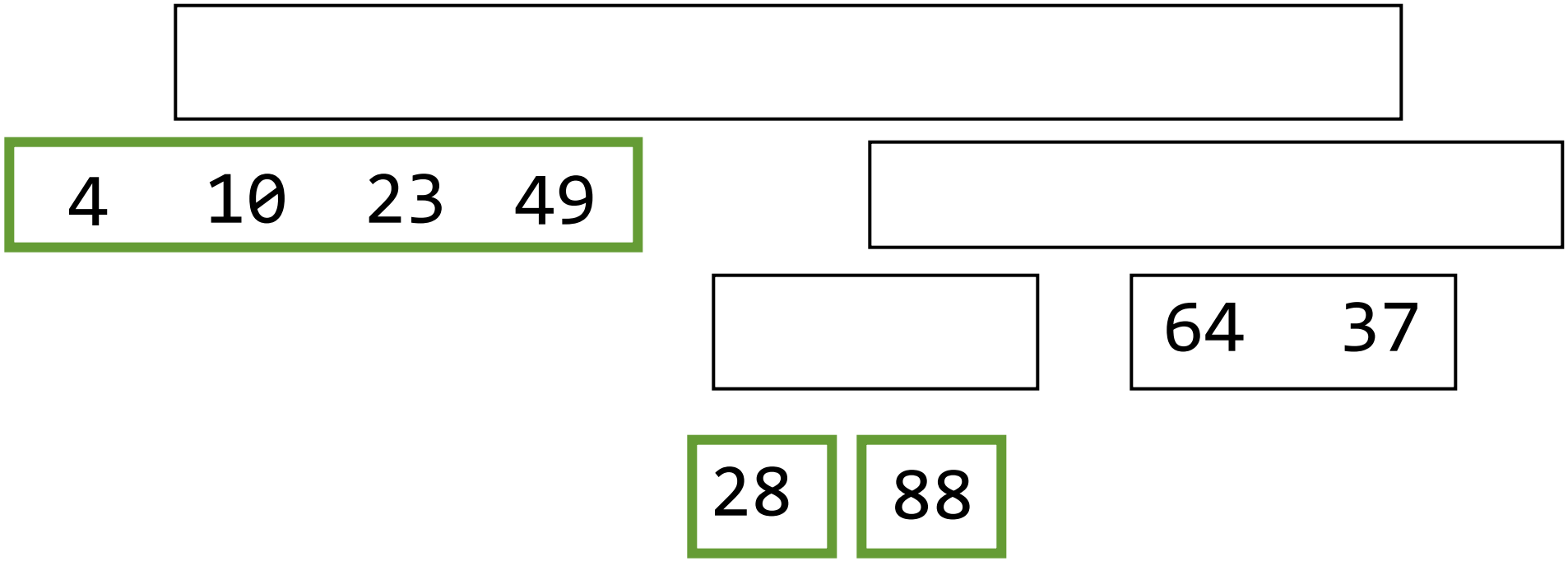

4 10 23 49
28 88 64 37

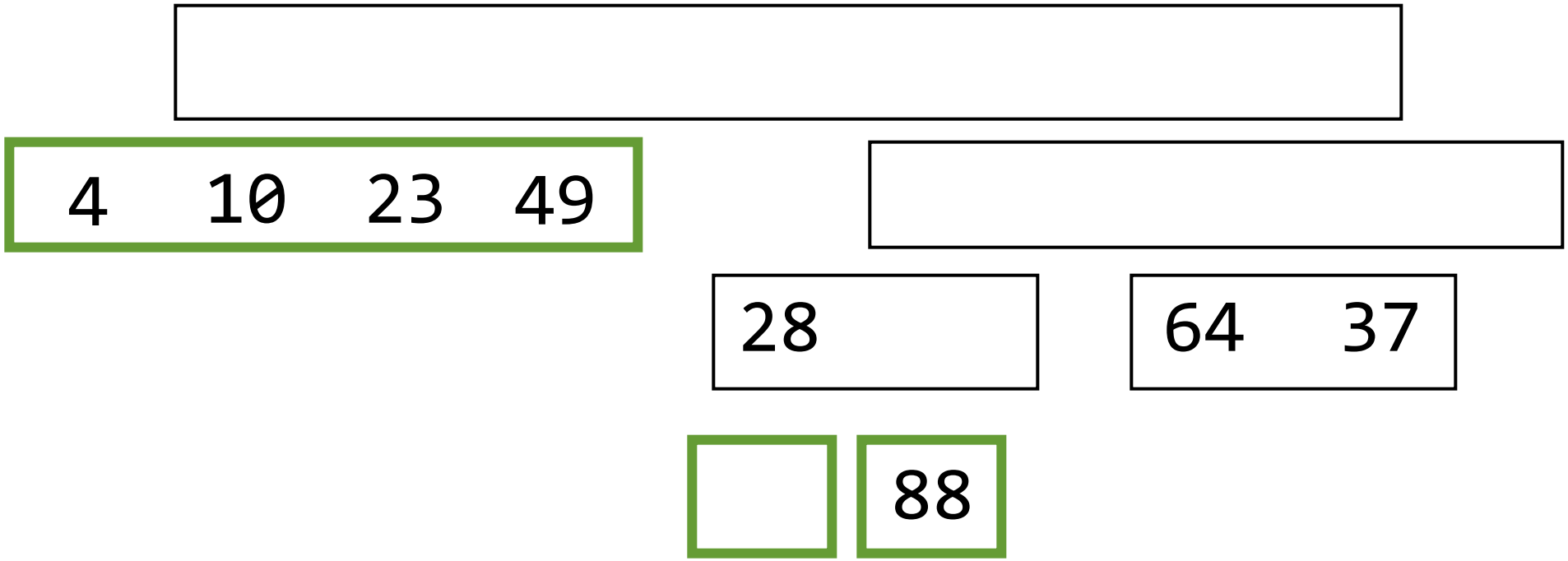

4 10 23 49
28 88 64 37

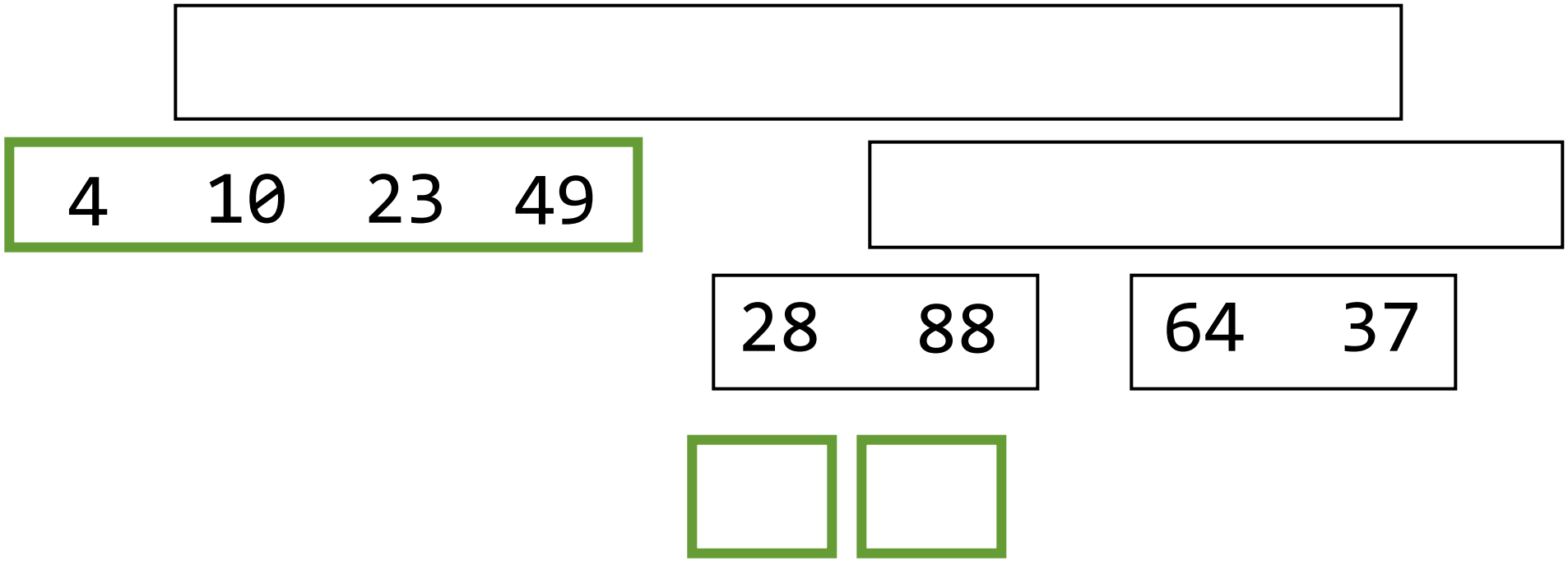

4 10 23 49
28 88
64 37

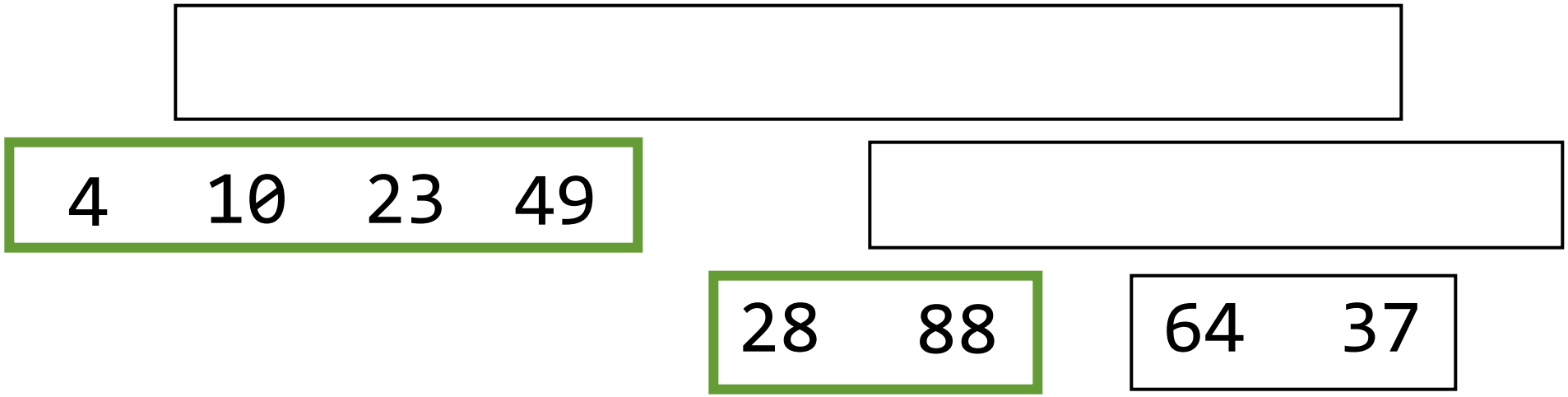

4 10 23 49
64 37
28
88

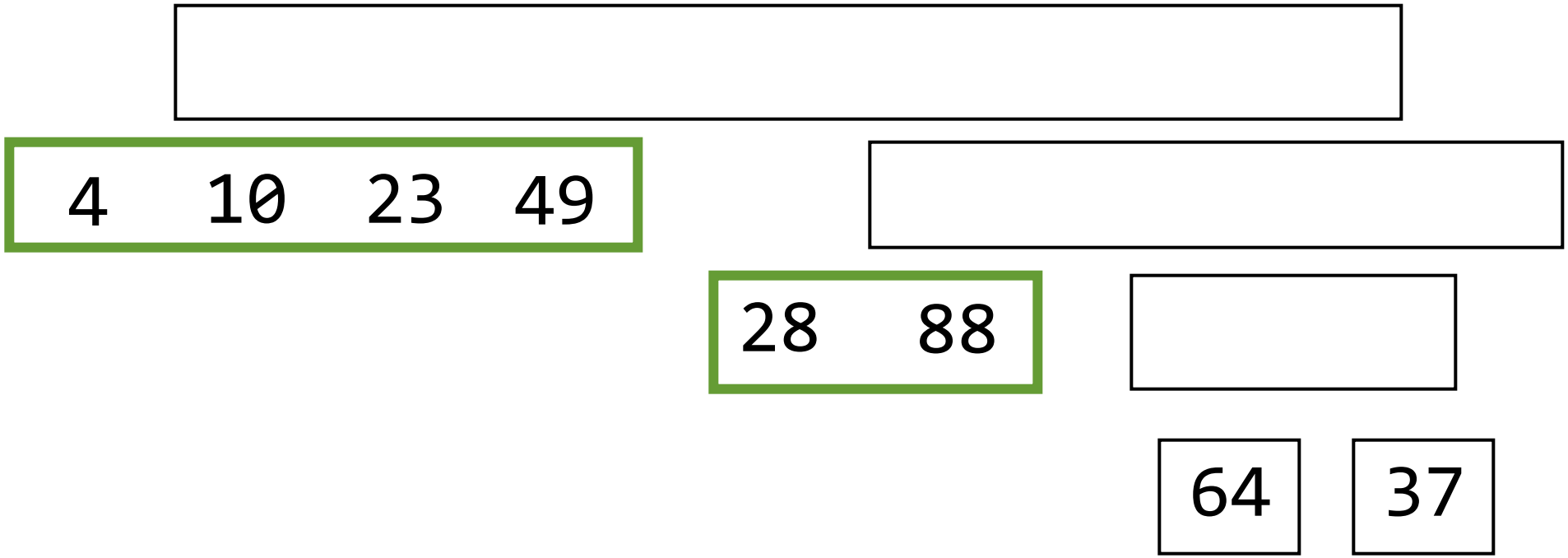

4 10 23 49
64 37
28
88

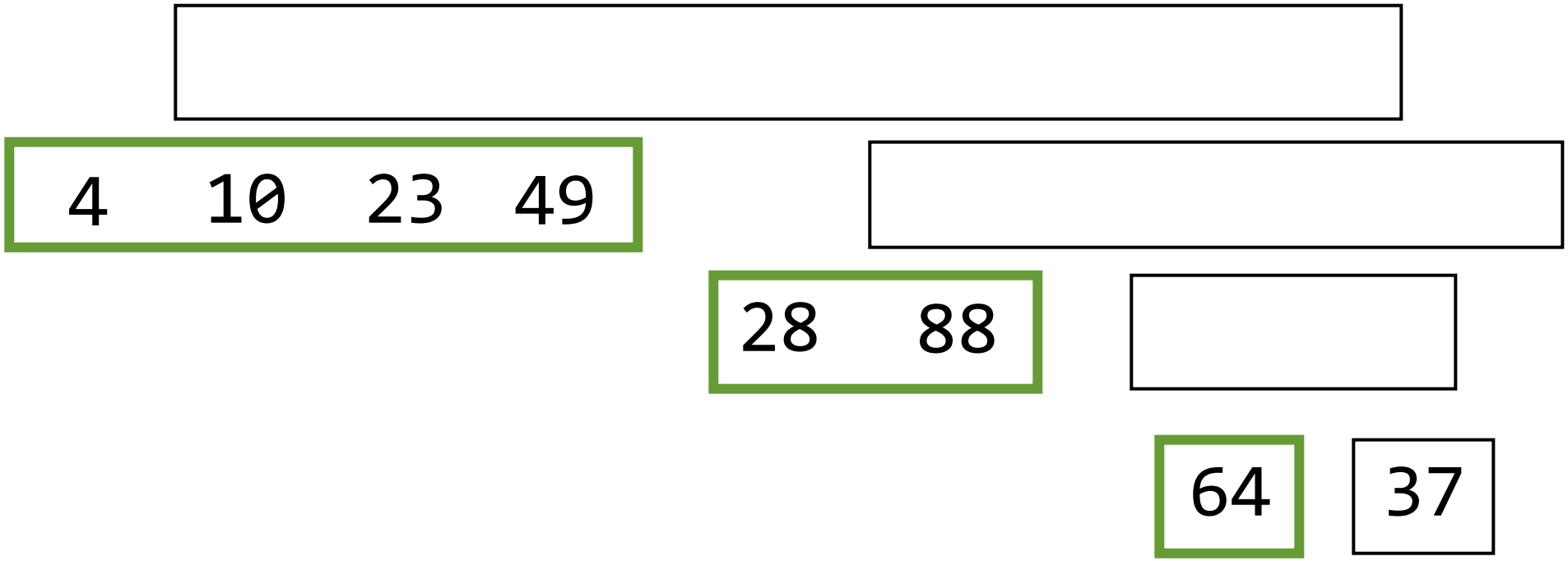

4 10 23 49
64 37
28 88

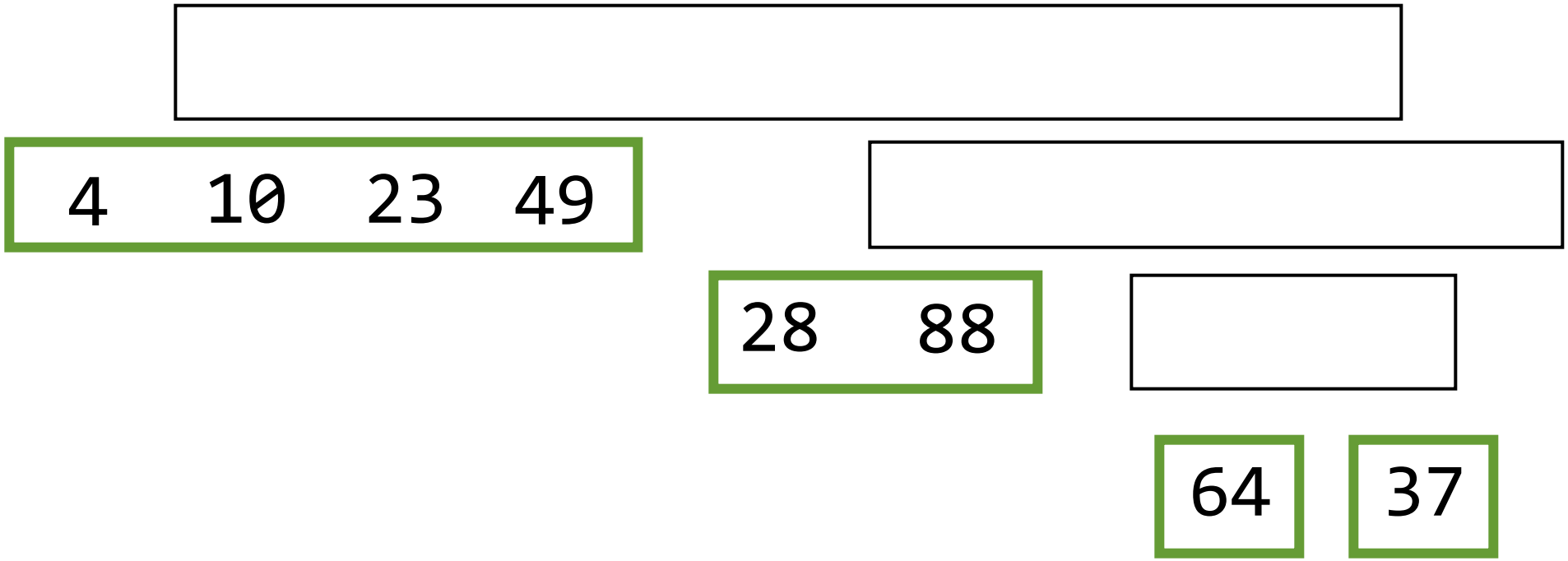

4 10 23 49
28
64 37
88

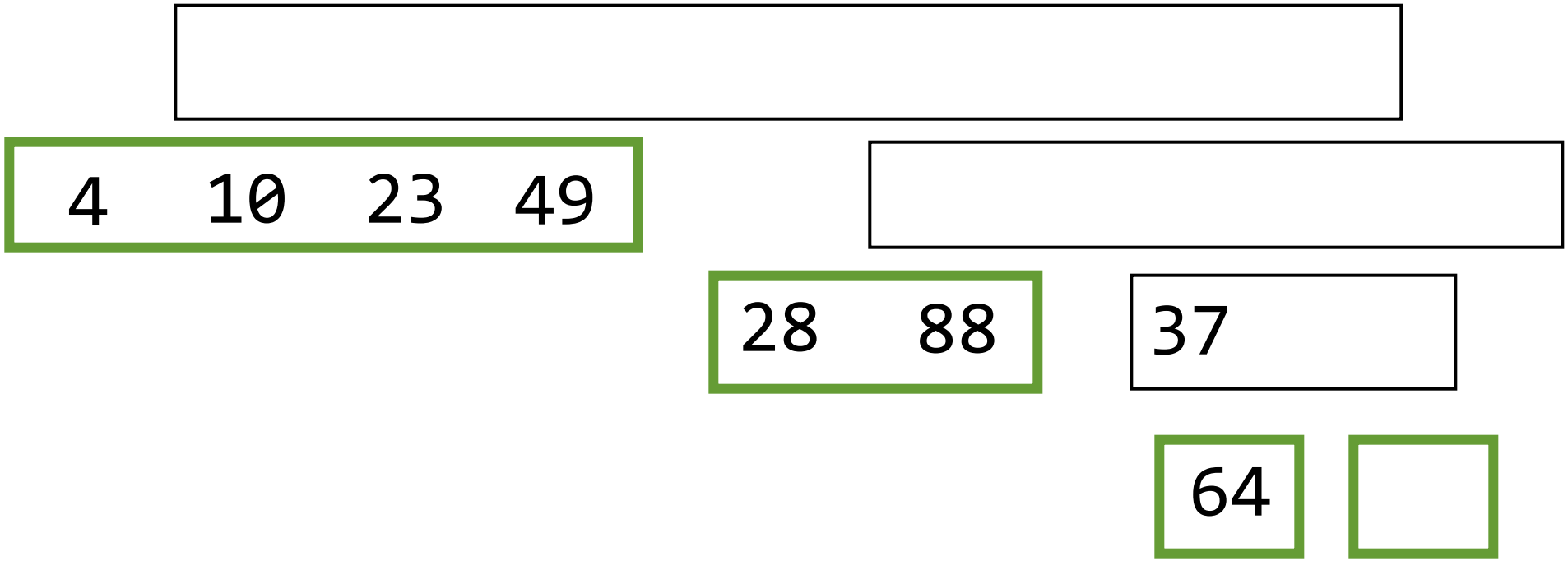

4 10 23 49
28 88
64 37

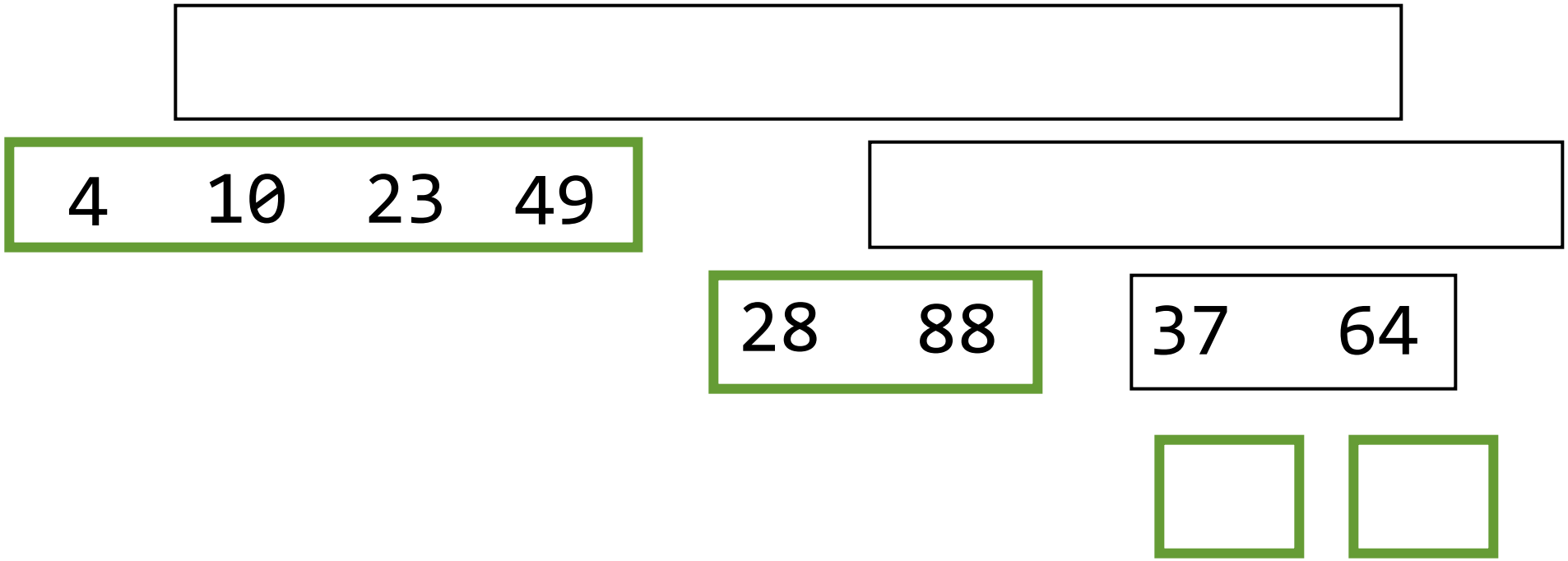

4 10 23 49
28 88
64 37

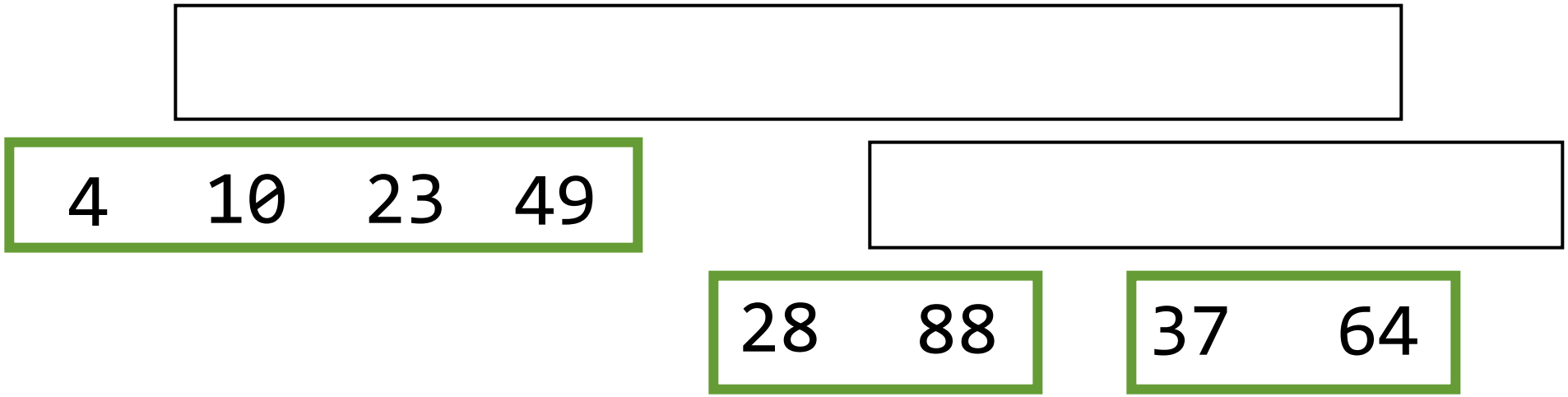

4 10 23 49
28 88
64
37

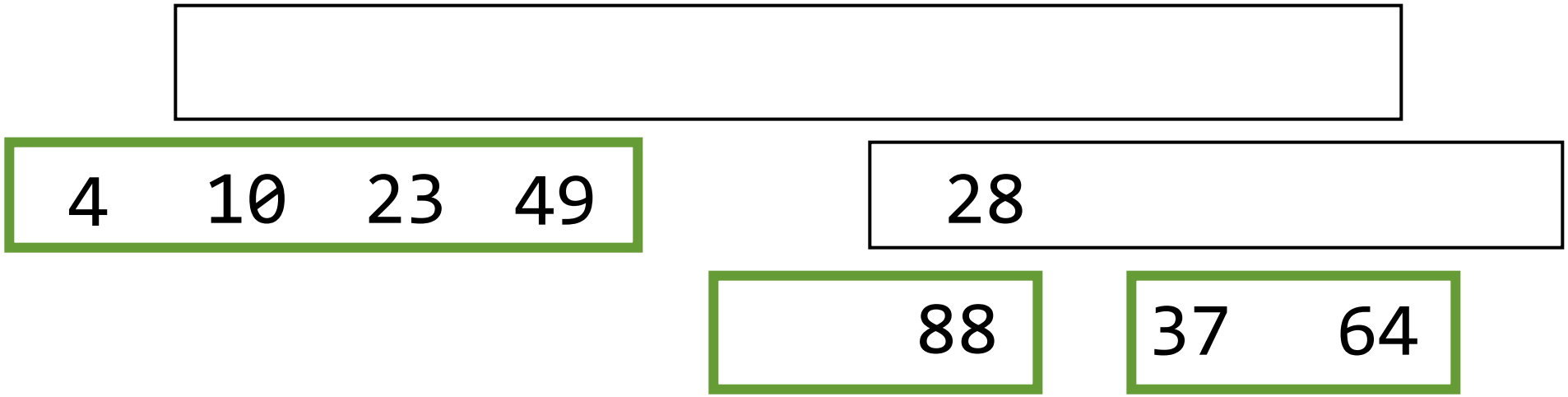

4 10 23 49
28 88
64
37

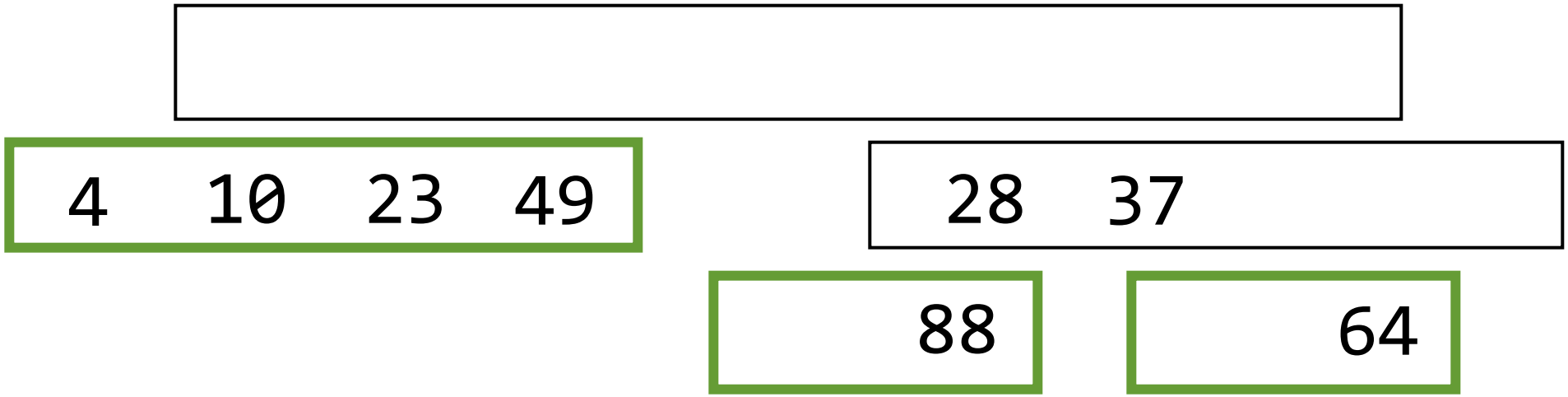

4 10 23 49
28 88
64 37

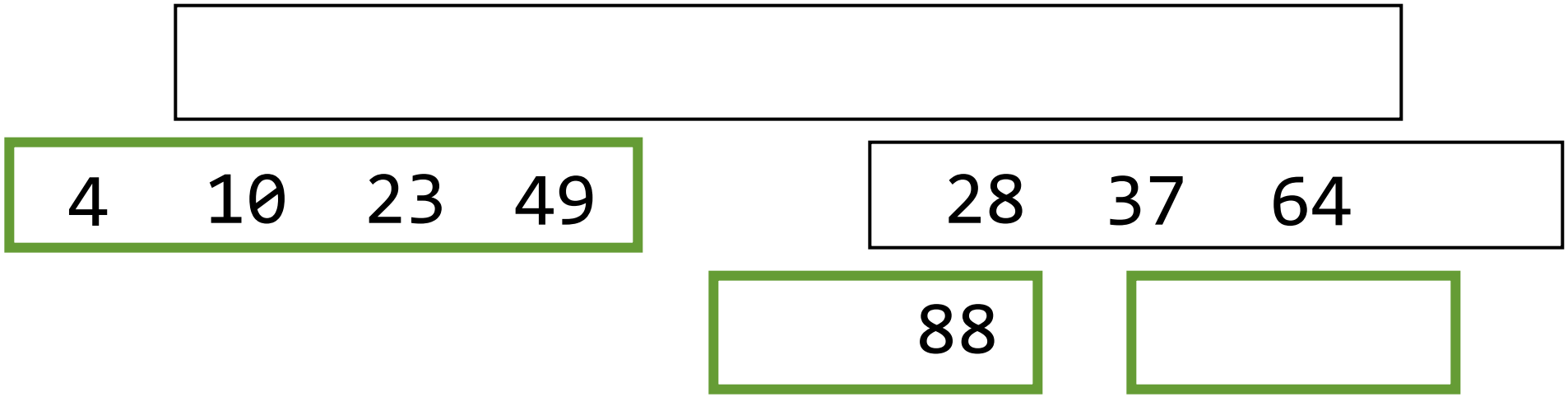

4 10 23 49
28 88
37
64

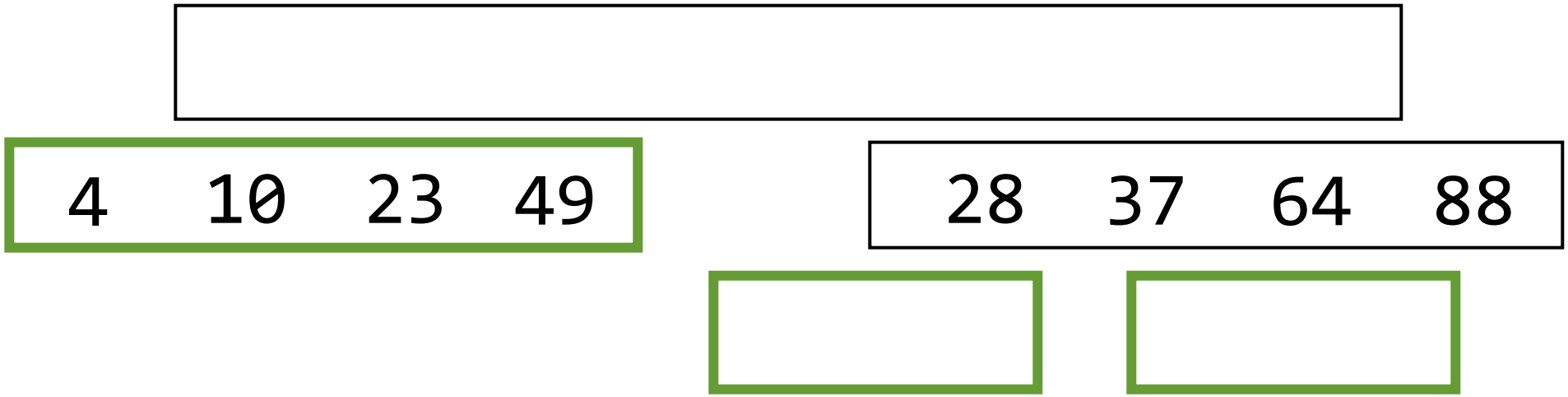

4 10 23 49
28 88
37 64


4 10 23 49
28 88
37 64


4 10 23 49
28
88
37 64


4 10 23 49
28 37
88
64


4 10 23 49
28 37 64
88

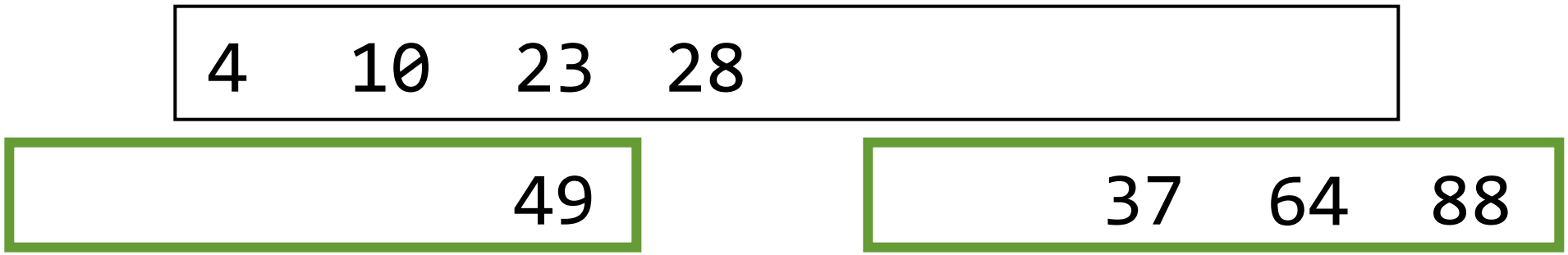

4 10 23 49
28 37 64 88

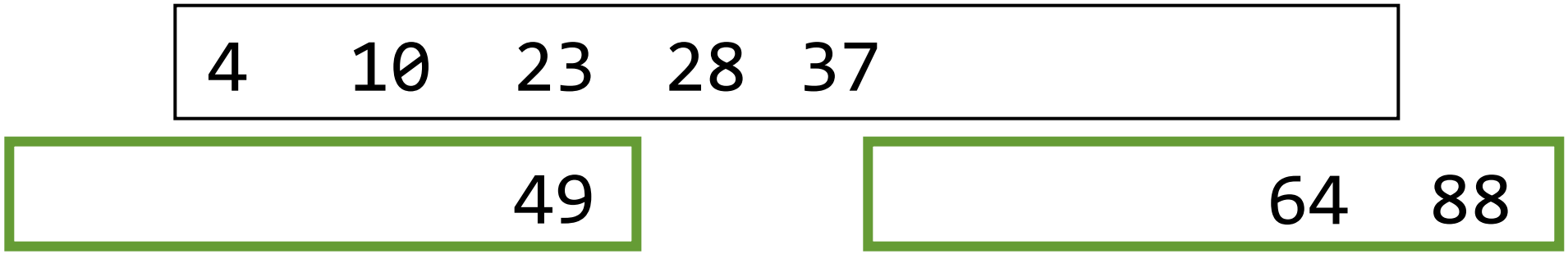

4 10 23 49
28 37 64 88

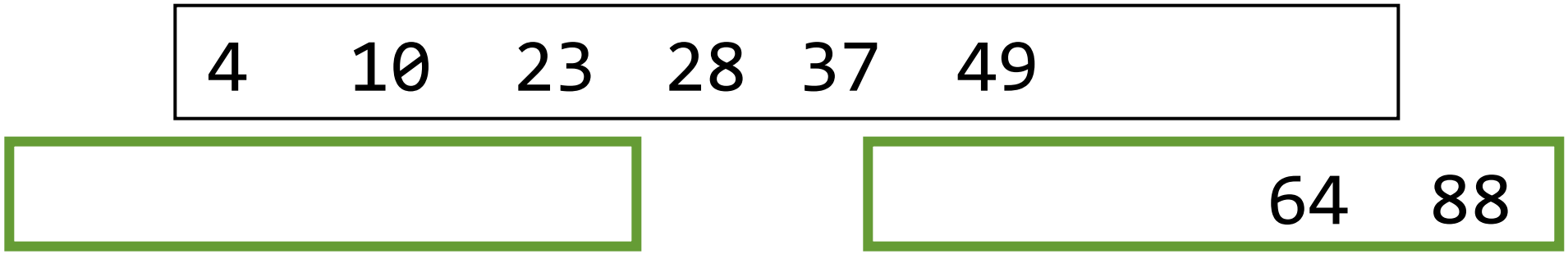

4
10 23 49
28 37 64 88

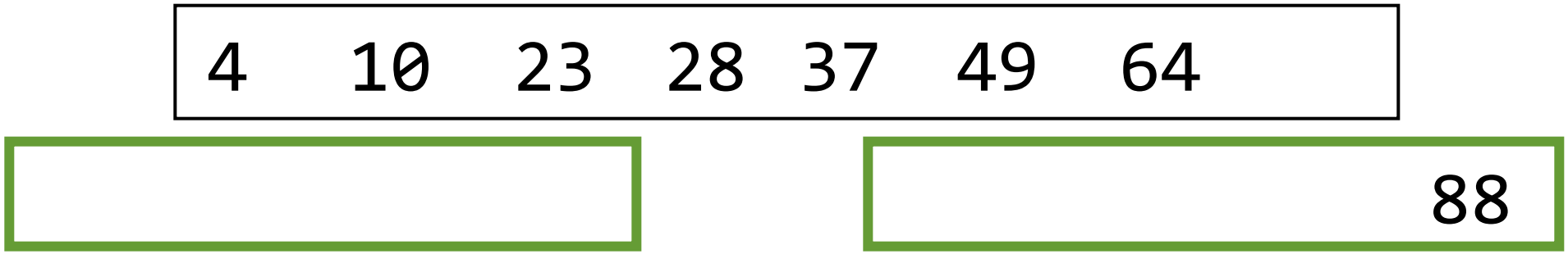

4 10
23 49
28 37 64 88

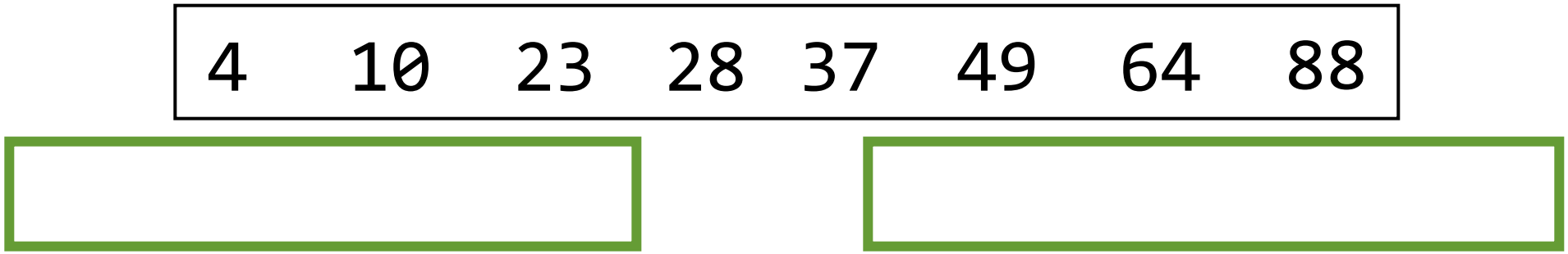

4 10 23
49
28 37 64 88


4 10 23 28
49
37 64 88

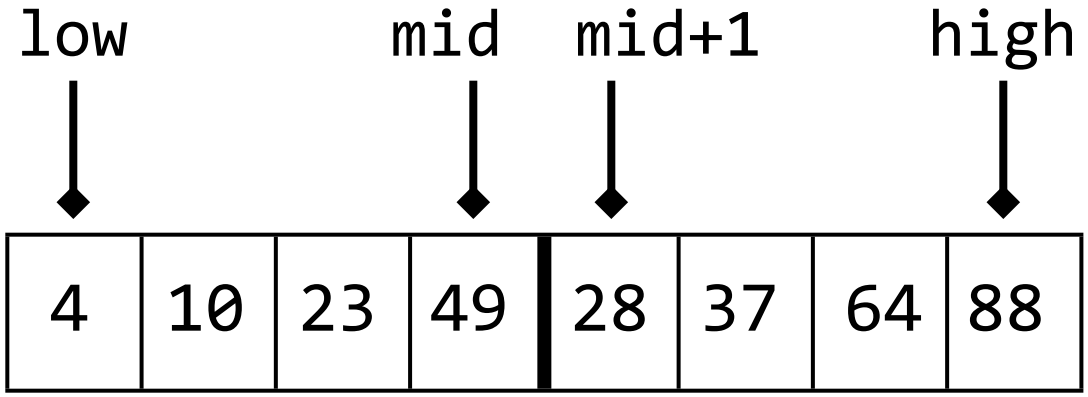

4 10 23 28 37
49
64 88


4 10 23 28 37 49
64 88


4 10 23 28 37 49 64
88


4 10 23 28 37 49 64 88


4 10 23 28 37 49 64 88

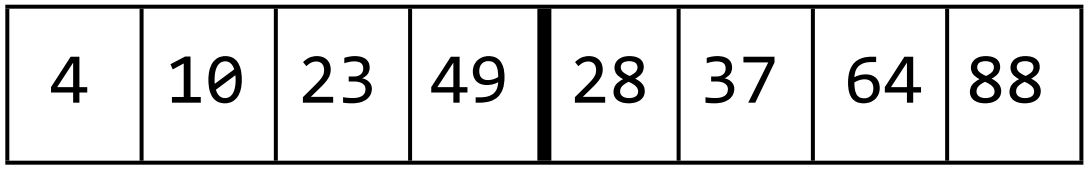


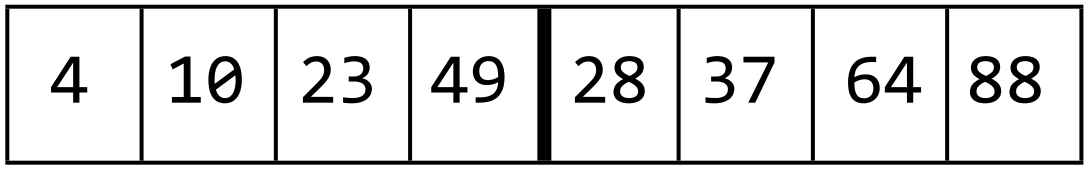

```
if (the list is of length 0 or 1)
{
   We're done.
}
else
{
   Divide the list into two halves.
   Merge sort the first half.
   Merge sort the second half.
   Merge the two sorted sublists.
}
```

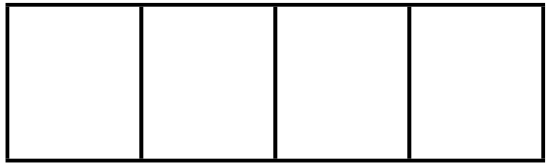
```
void mergeSort(int list[], int n, int low, int high)
{
  if (high - low >= 1)
  {
    int mid = (high + low) / 2;

    mergeSort(list, n, low, mid);
    mergeSort(list, n, mid + 1, high);
    merge( ... );
```

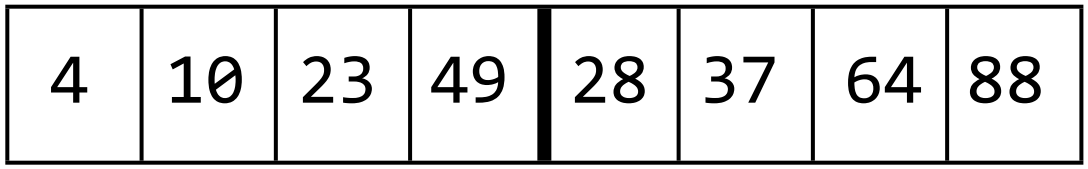
Merging

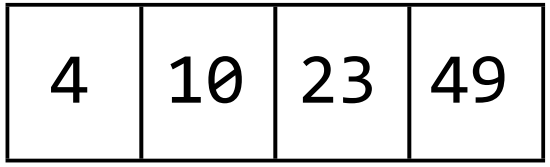
- You can merge *in-place*, but it's more difficult
- Generally we take additional space to do a merge
- As such, merge sort usually takes additional space

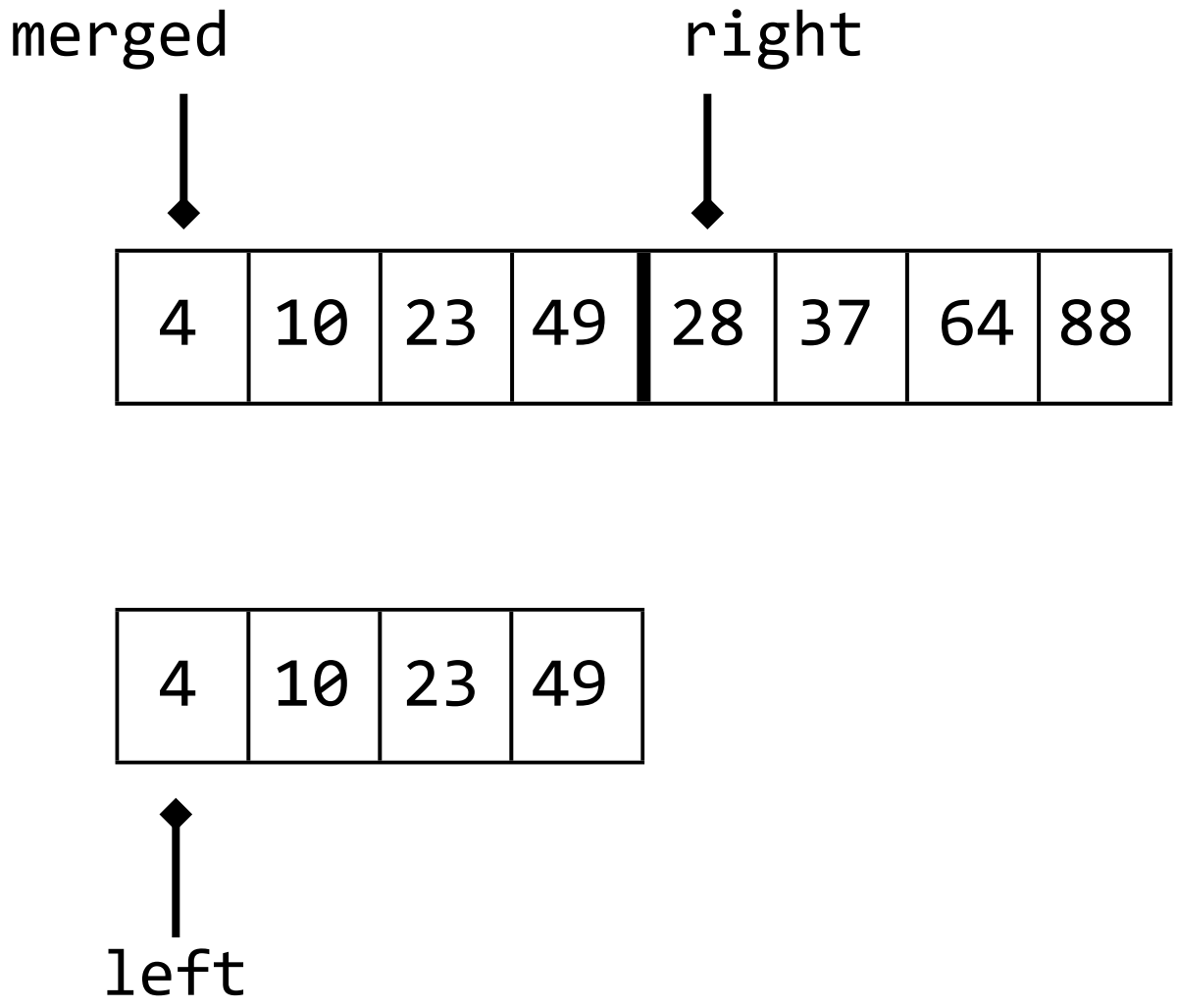

4
10
23
49
28
37
64
88

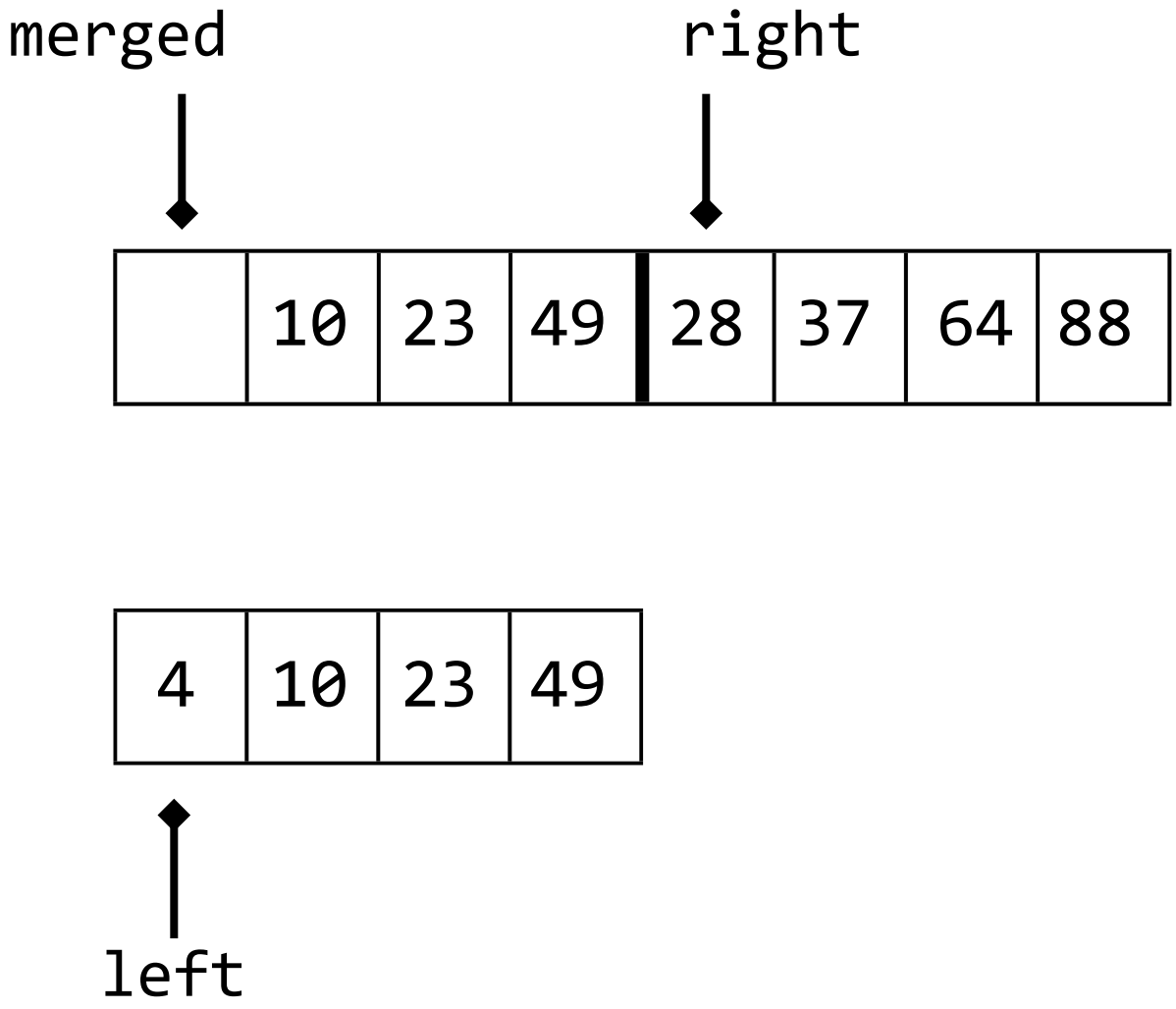

4
10
23
49
28
37
64
88

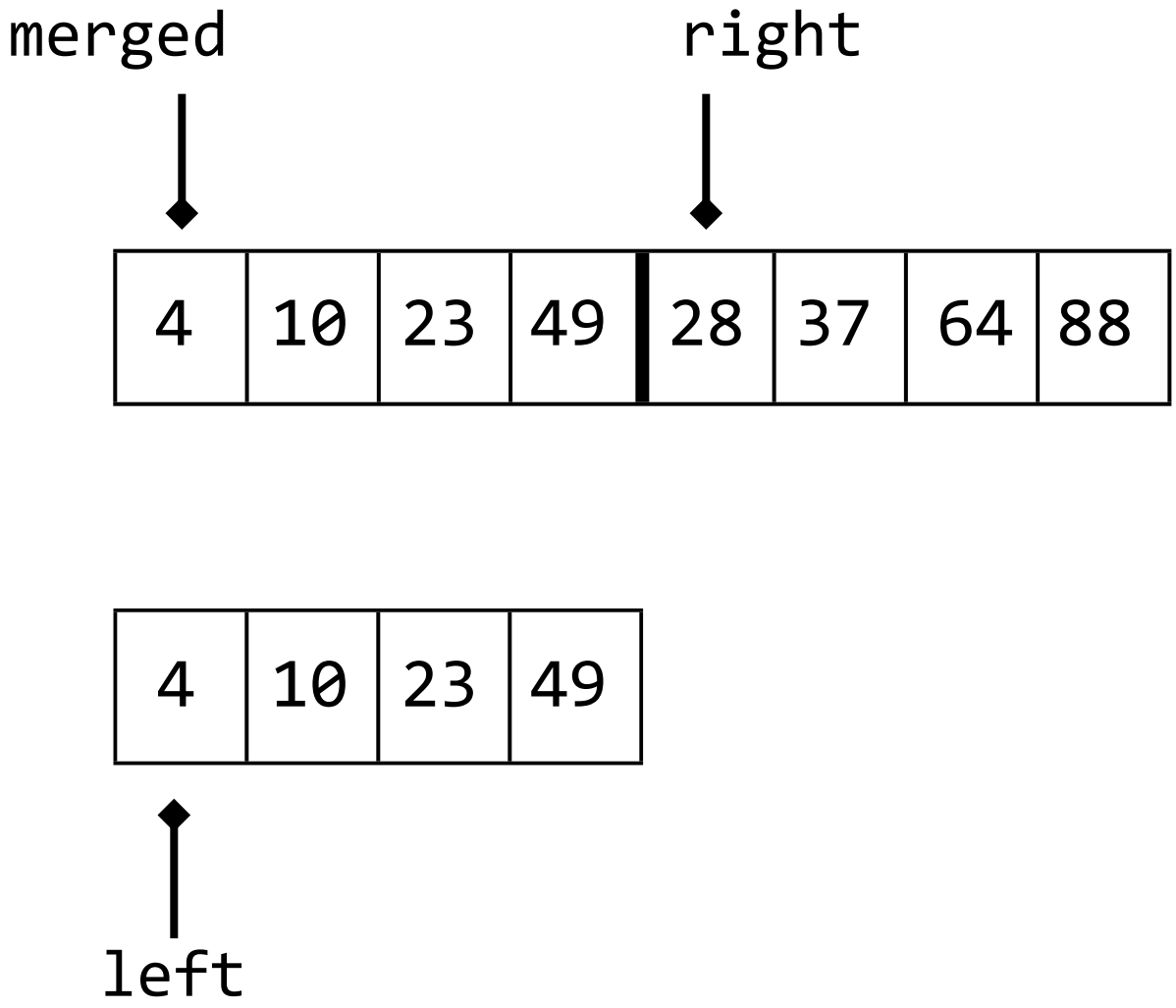


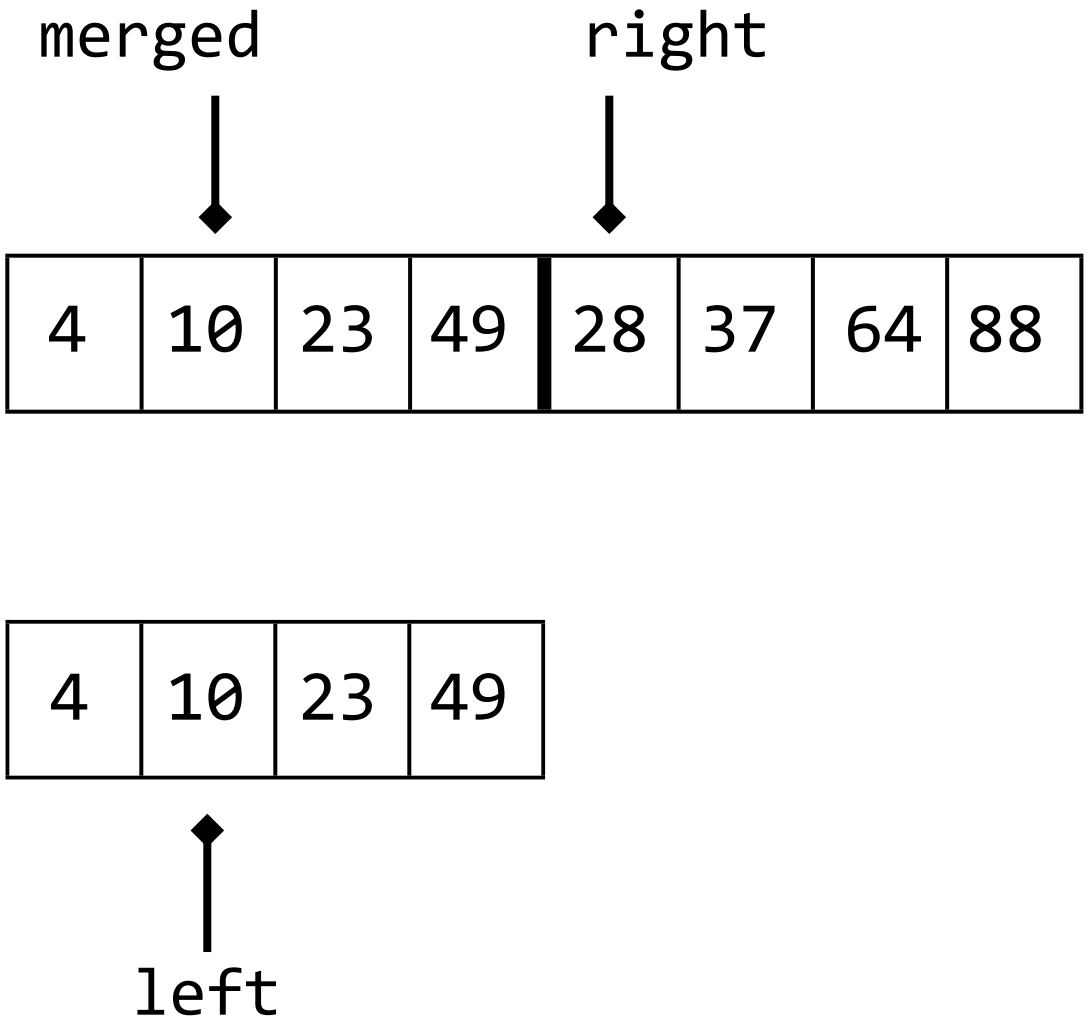

4
10
23
49
28
37
64
88

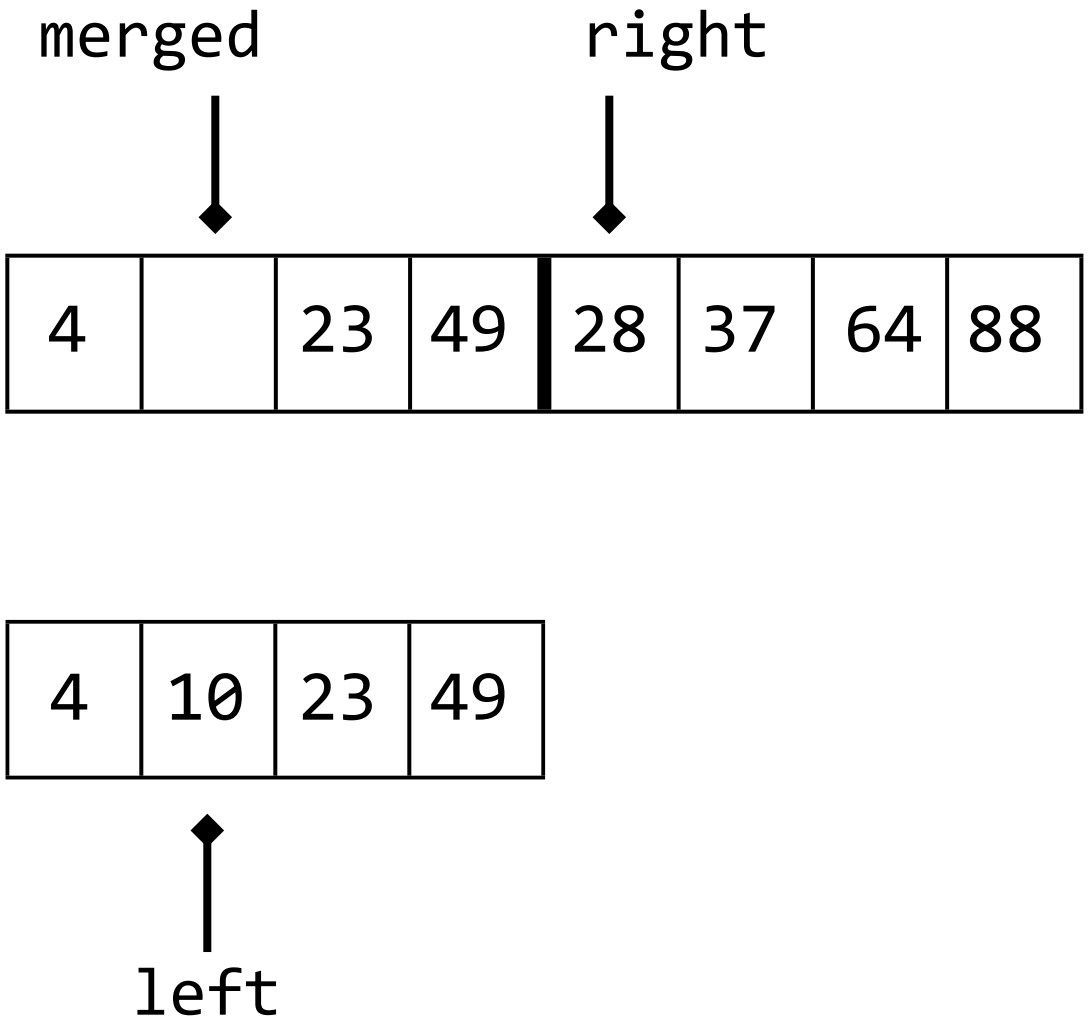

4
10
23
49

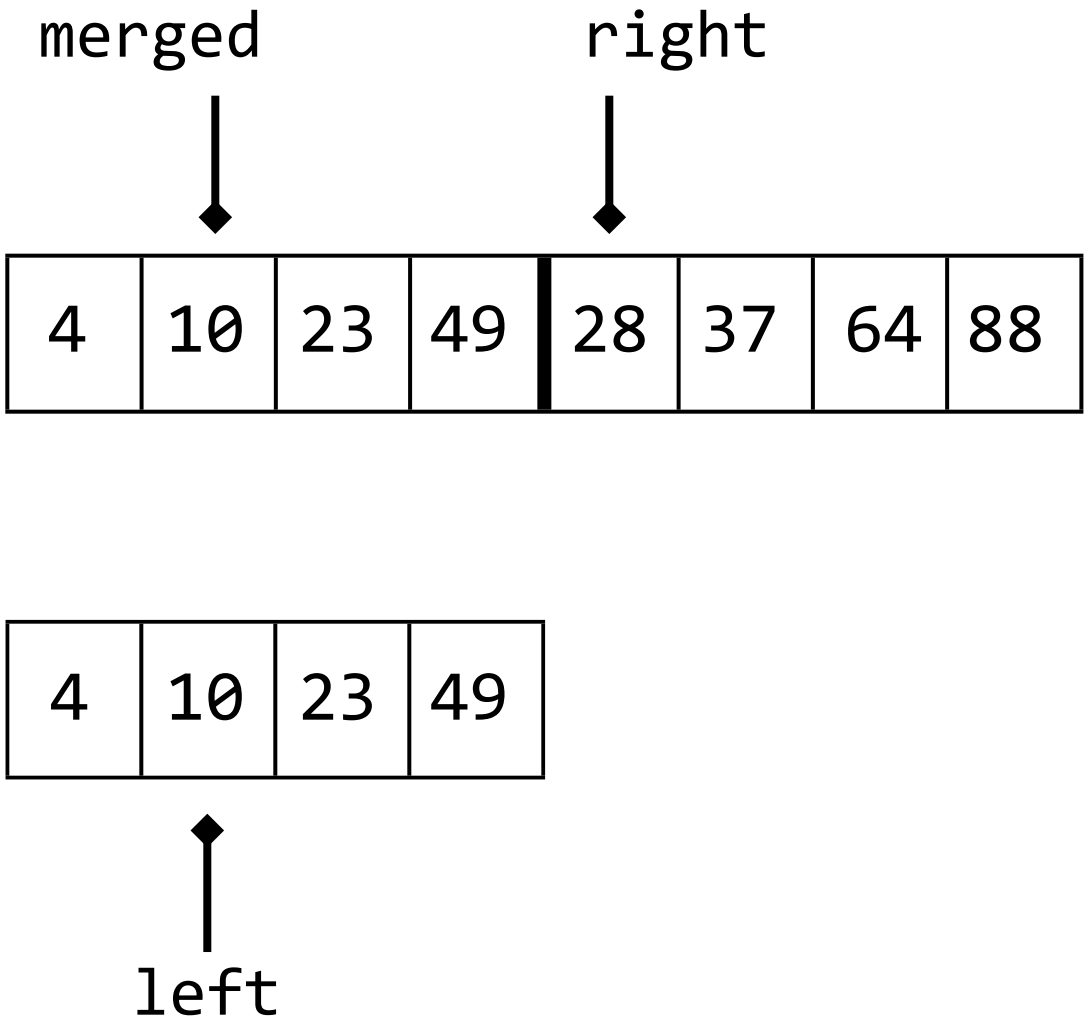

merged
right
4
10
23
49
28
37
64
88
4
10
23
49
left

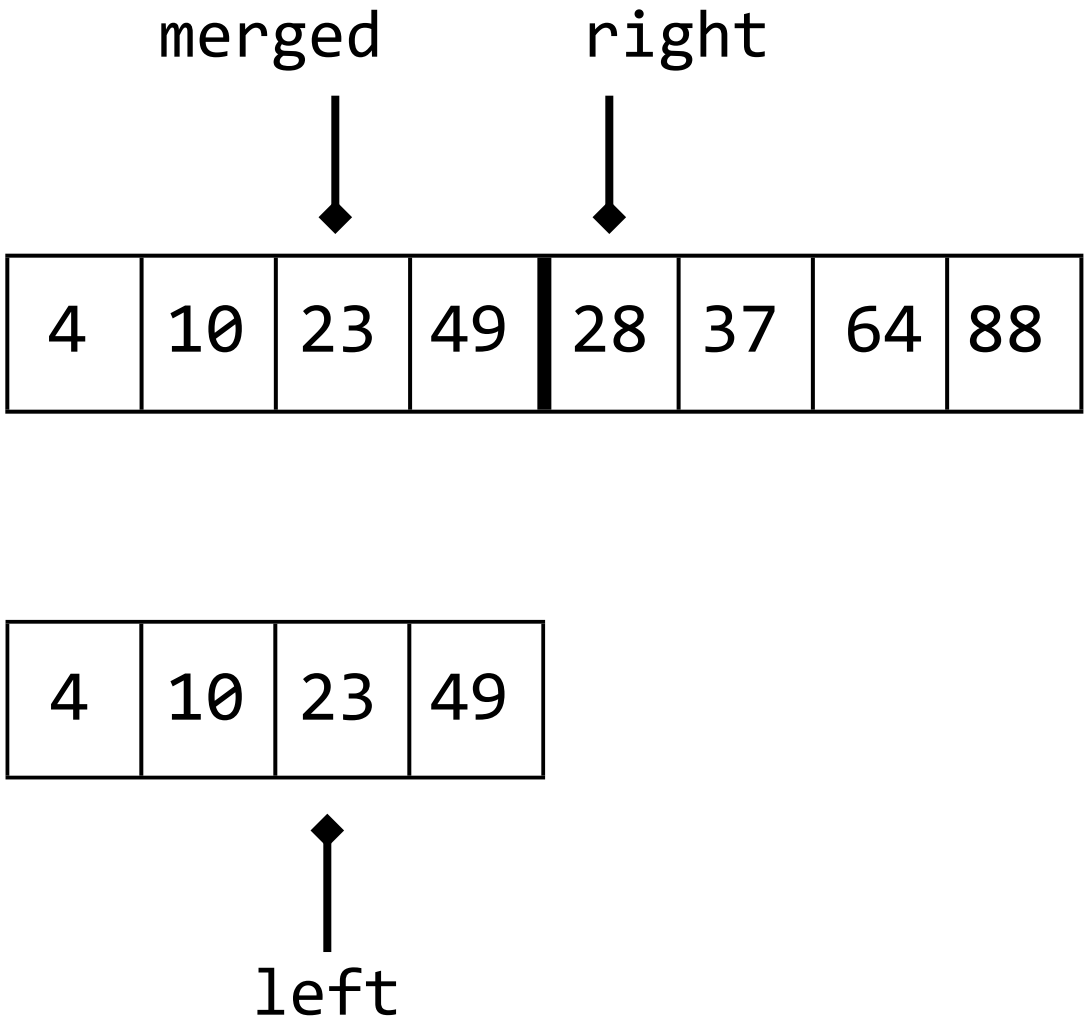

merged
right
10
23
49
28
37
64
88
4
10
23
49
left

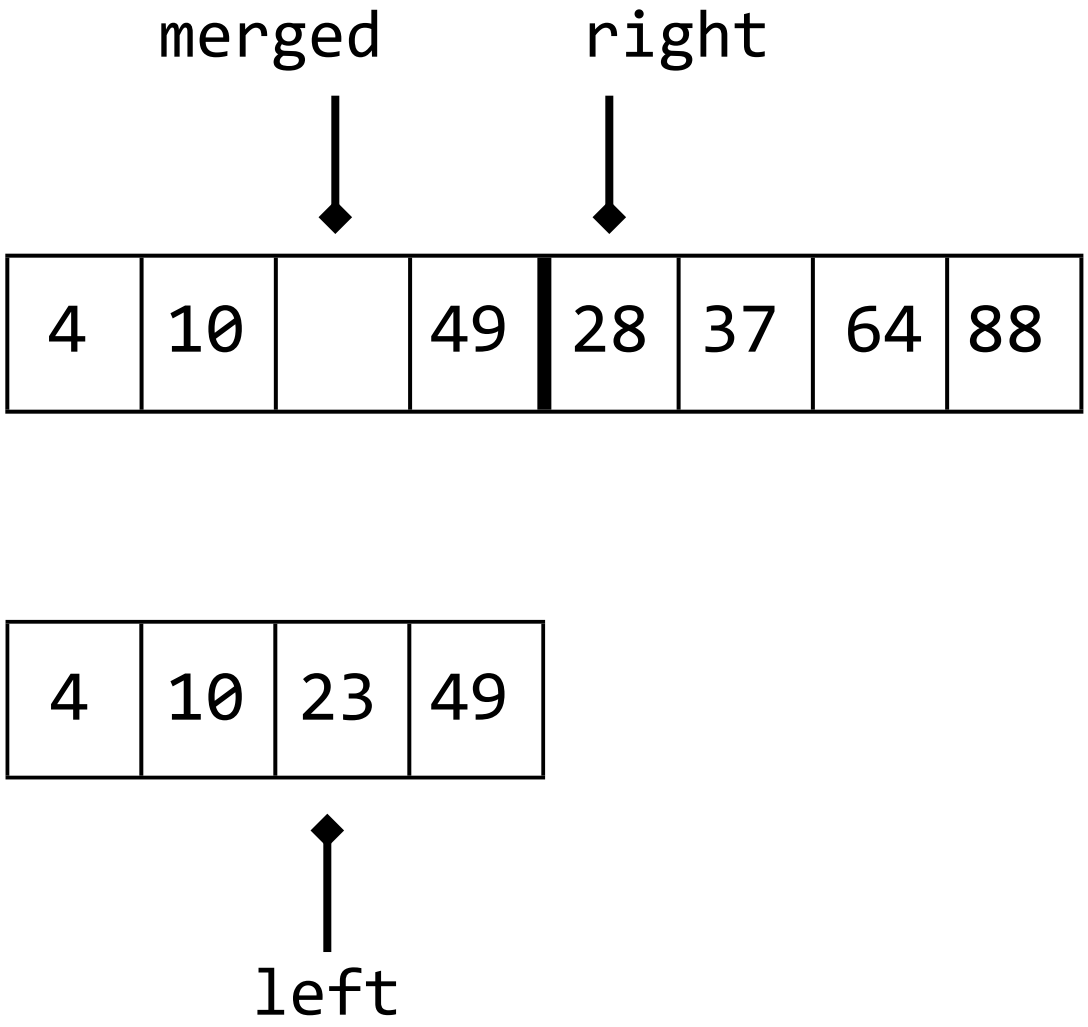

merged
right
4
10
23
49
28
37
64
88
4
10
23
49
left

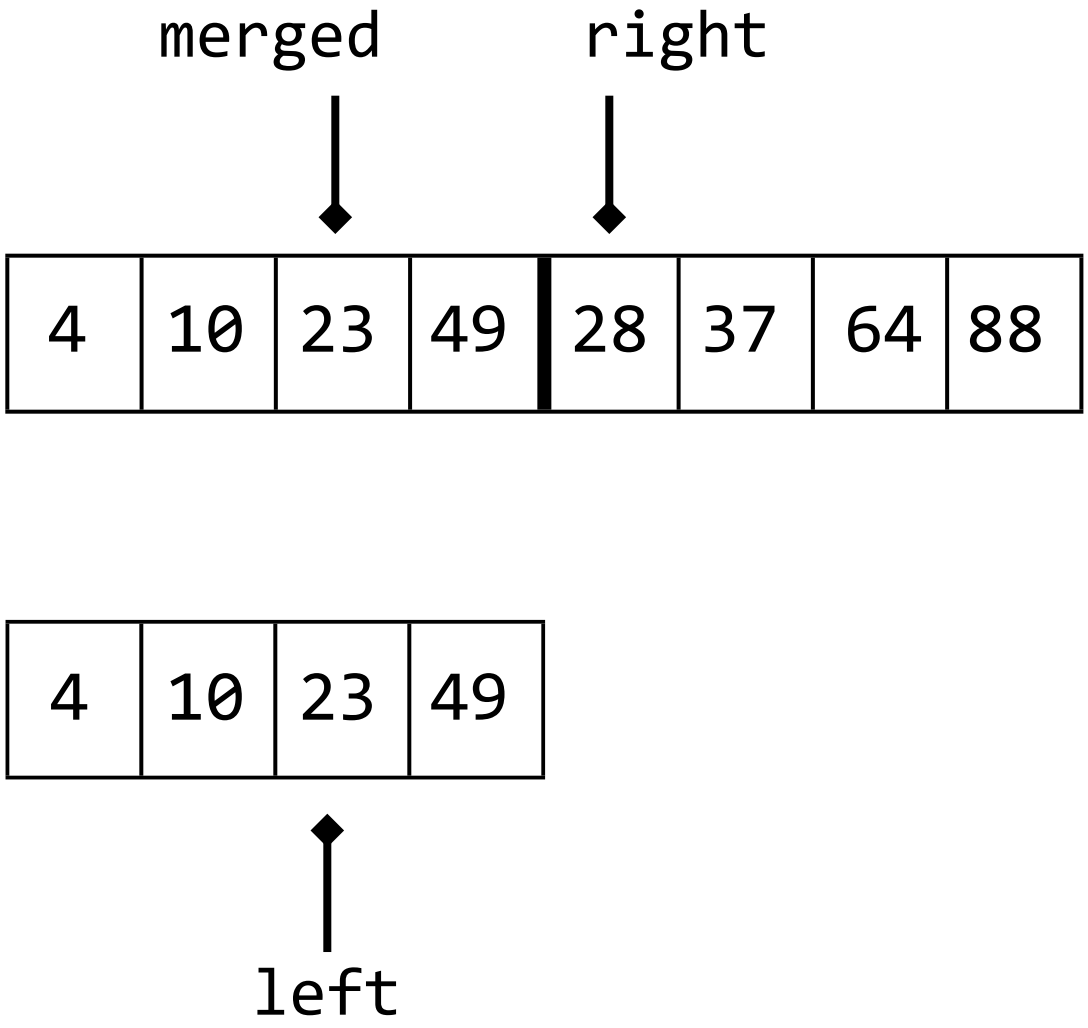

merged
right
4
10
23
49
28
37
64
88
4
10
23
49
left

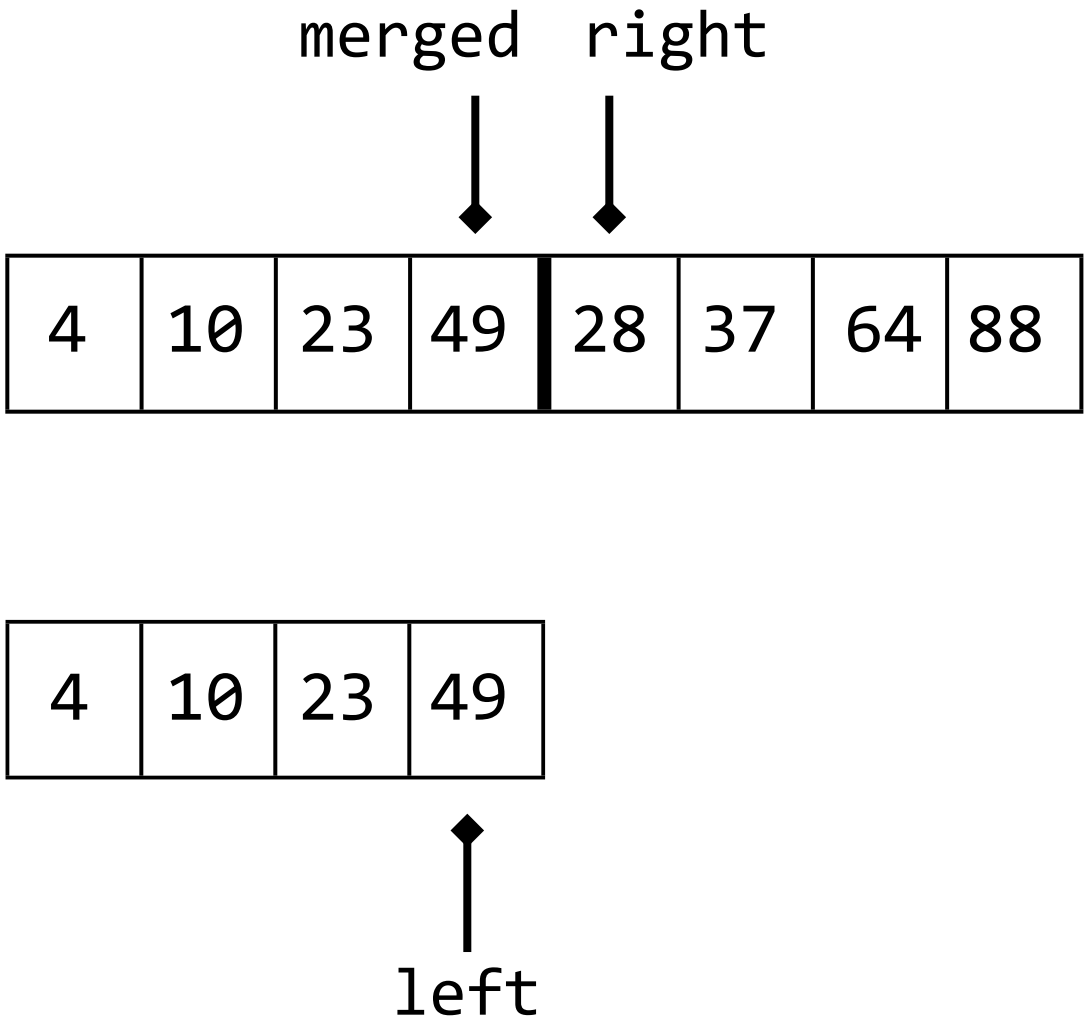

merged
right
4
23
49
28
37
64
88
4
10
23
49
left

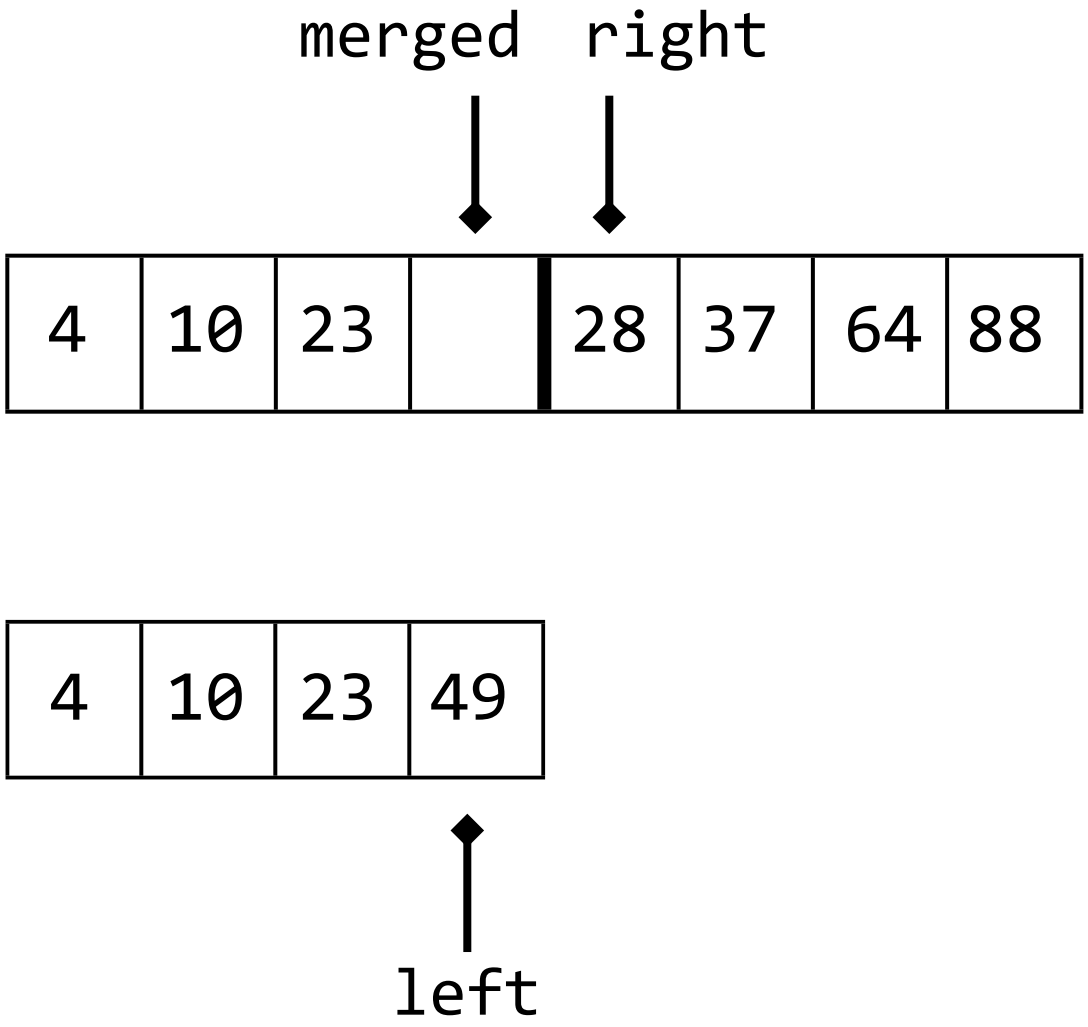

merged
right
4
10
23
49
28
37
64
88
4
10
23
49
left

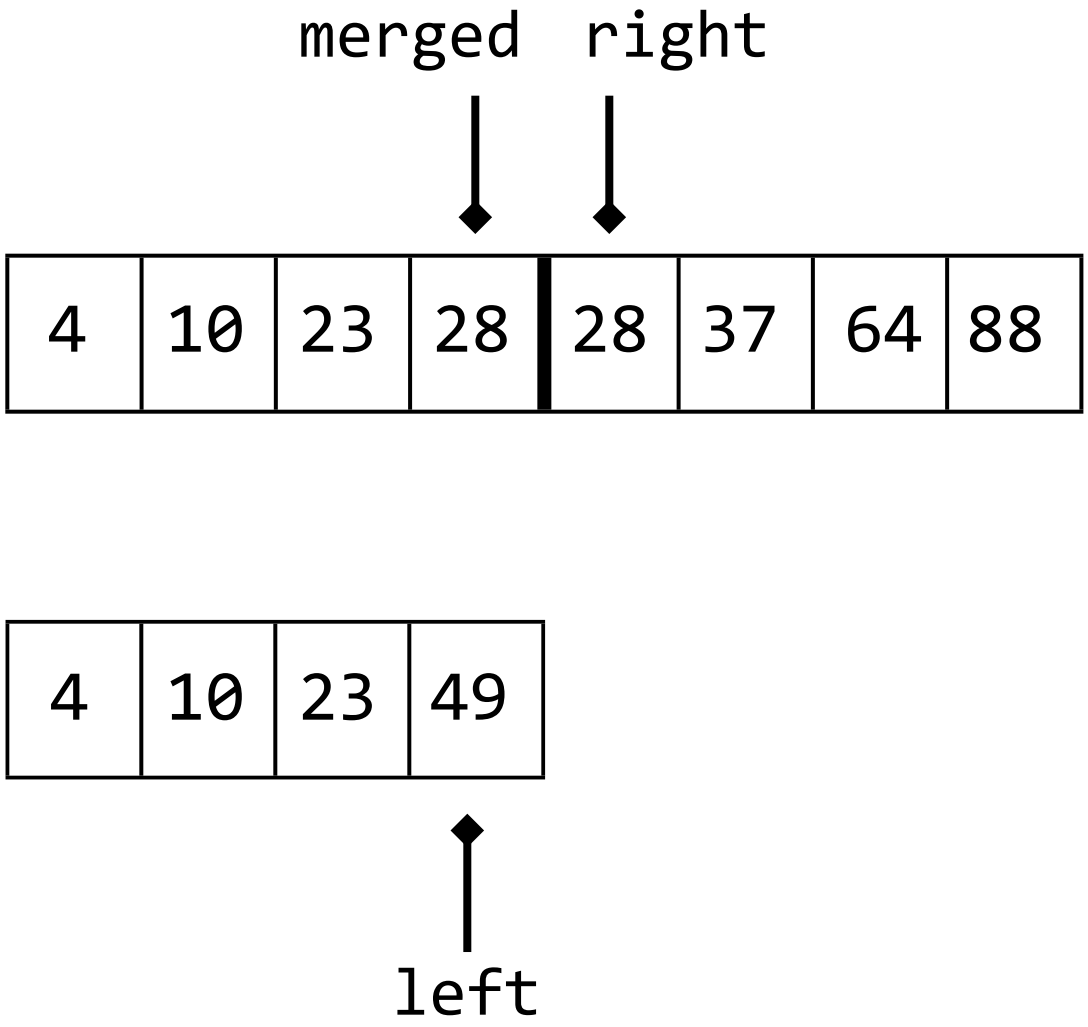

merged
right
4
10
23
49
28
37
64
88
4
10
23
49
left

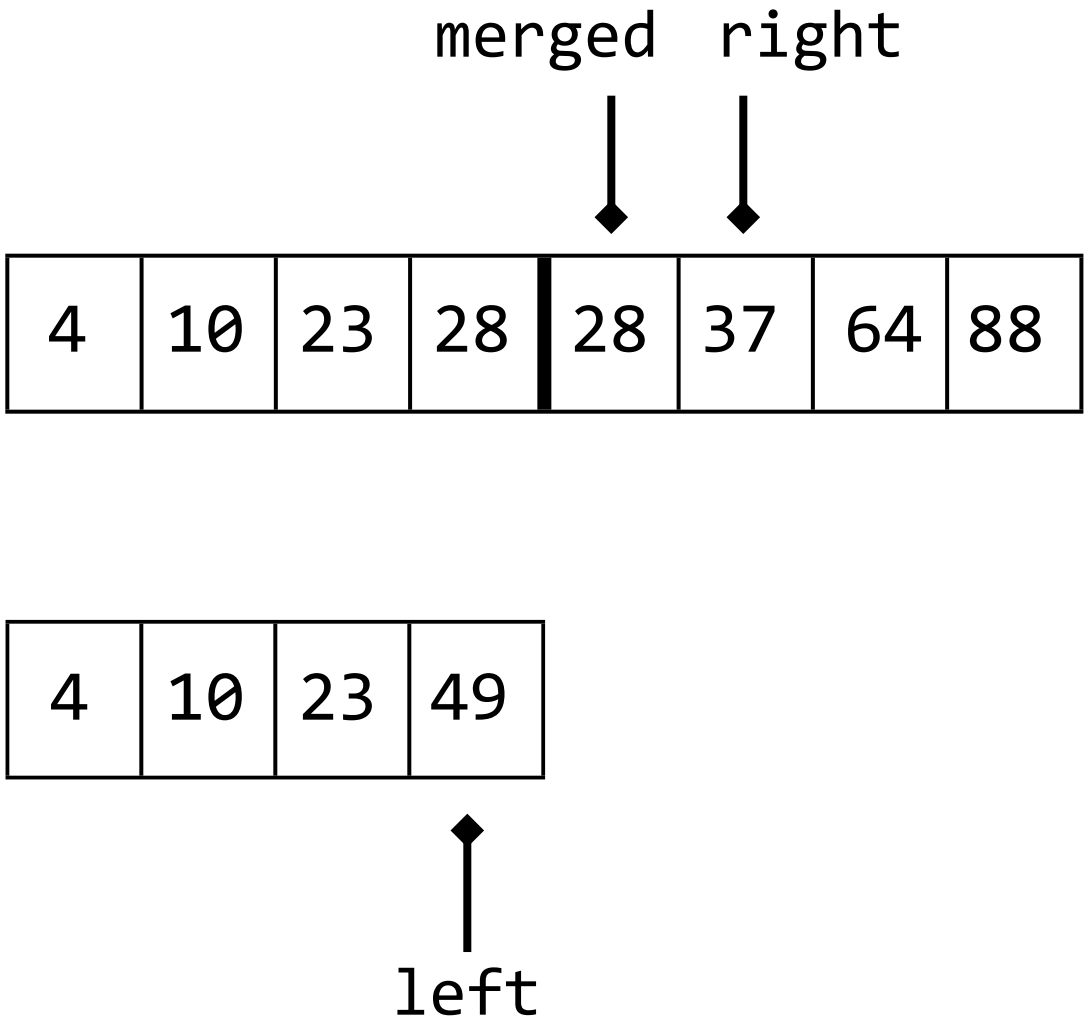

merged
right
4
10
49
28
37
64
88
4
10
23
49
left

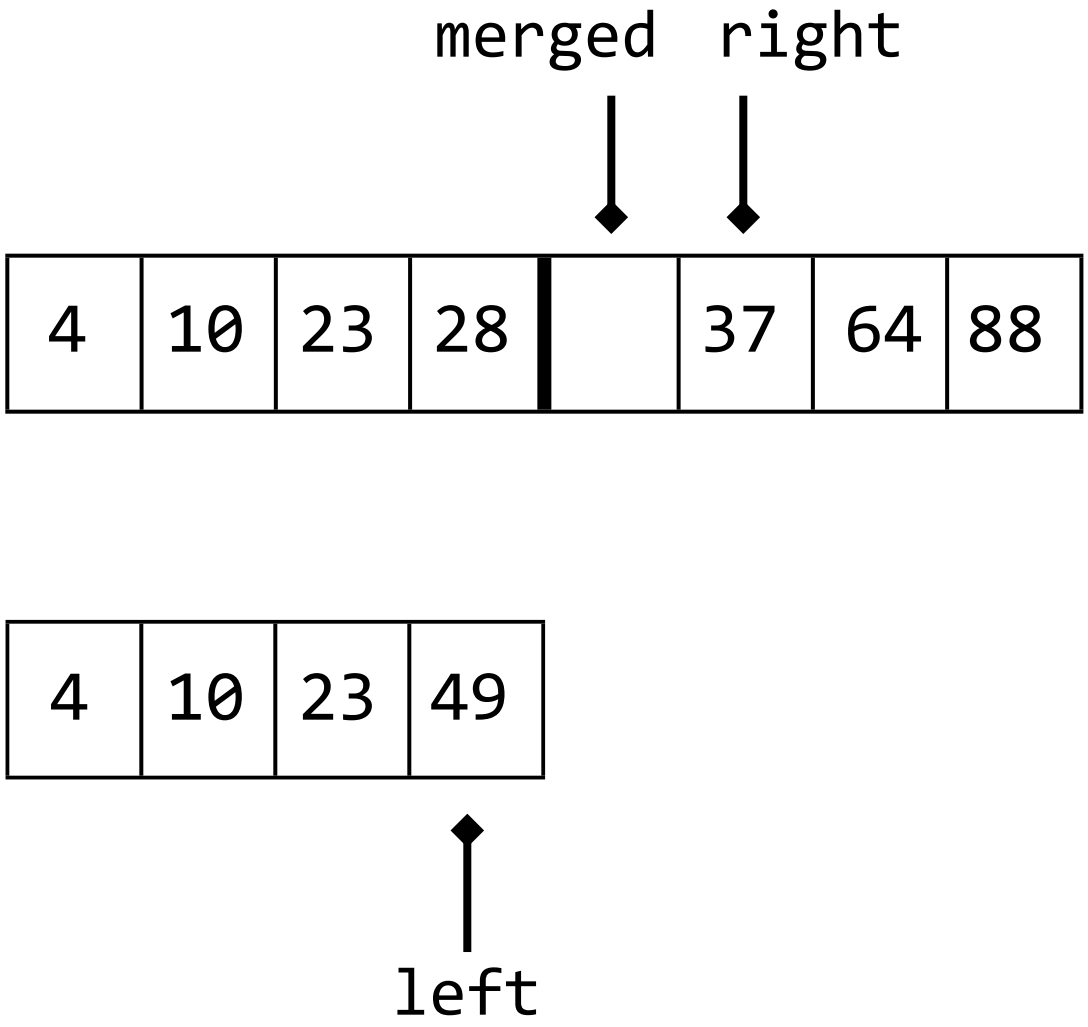

merged
right
4
10
23
49
28
37
64
88
4
10
23
49
left

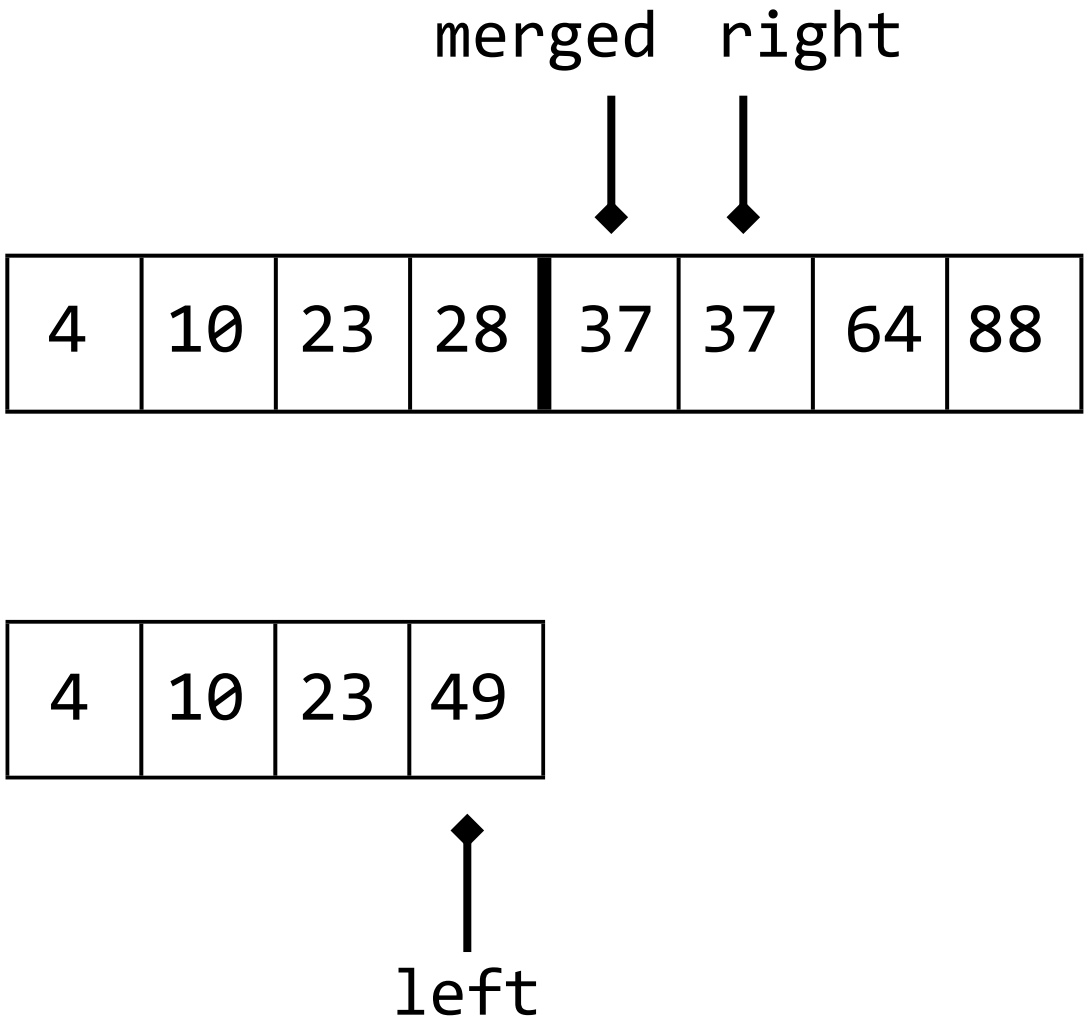

merged
right
4
10
23
49
28
37
64
88
4
10
23
49
left

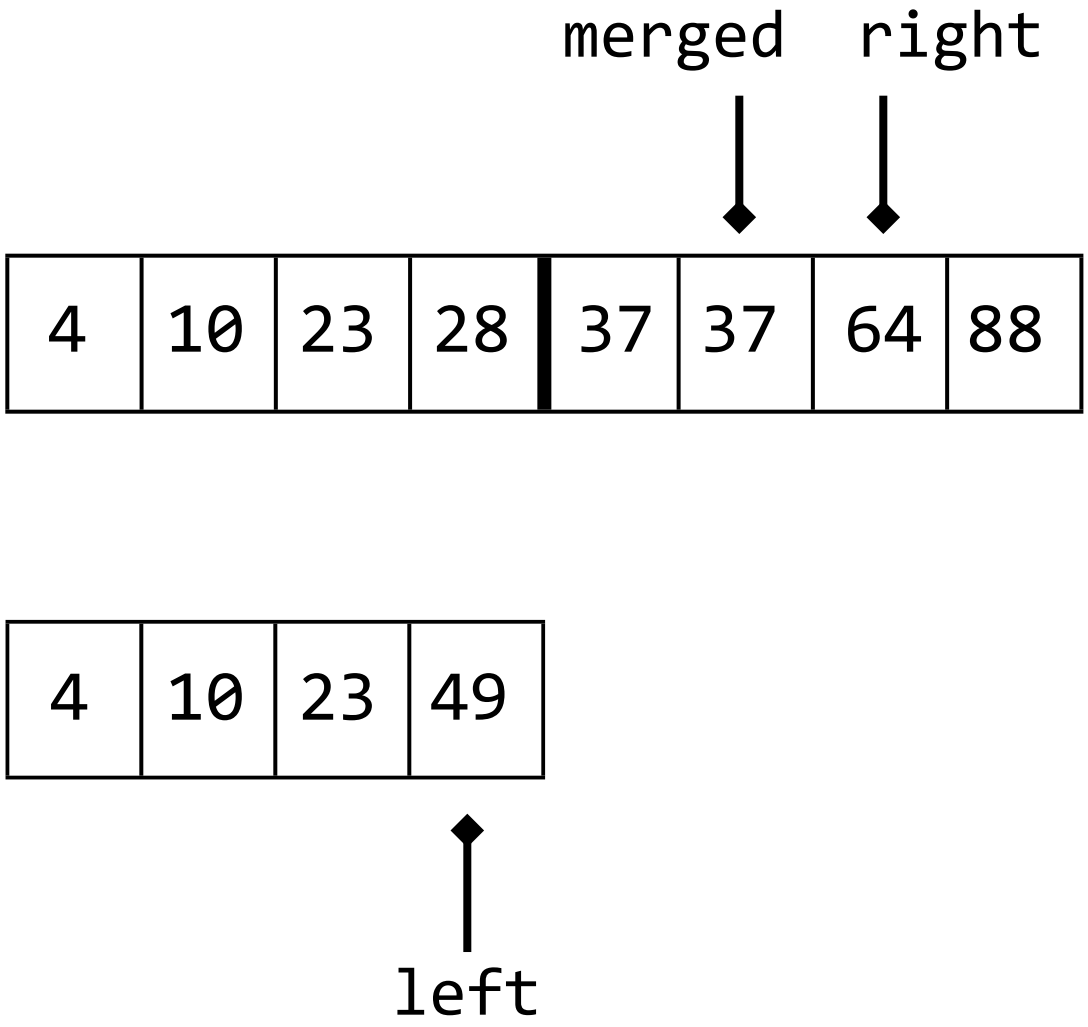

merged
right
4
10
23
28
37
64
88
4
10
23
49
left

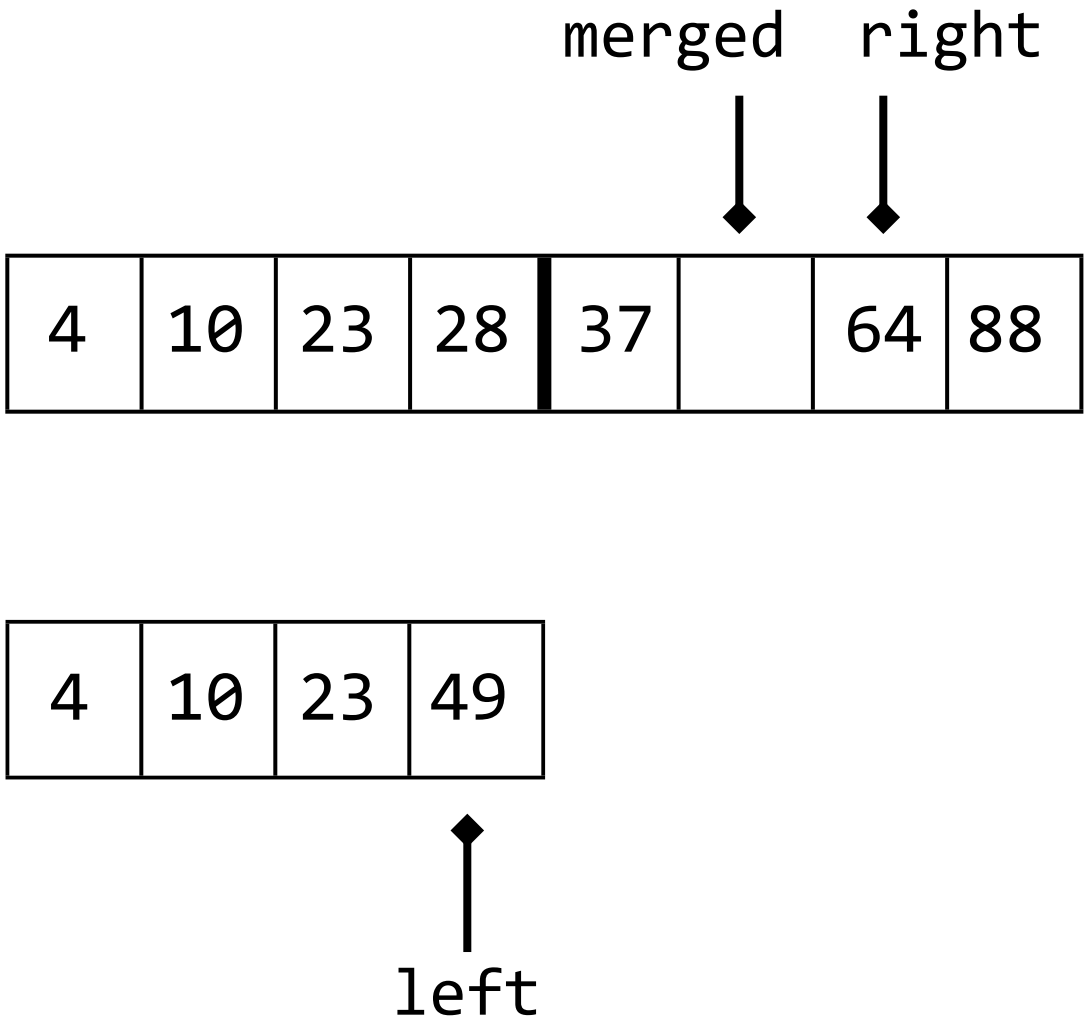

merged
right
4
10
23
28
28
37
64
88
4
10
23
49
left

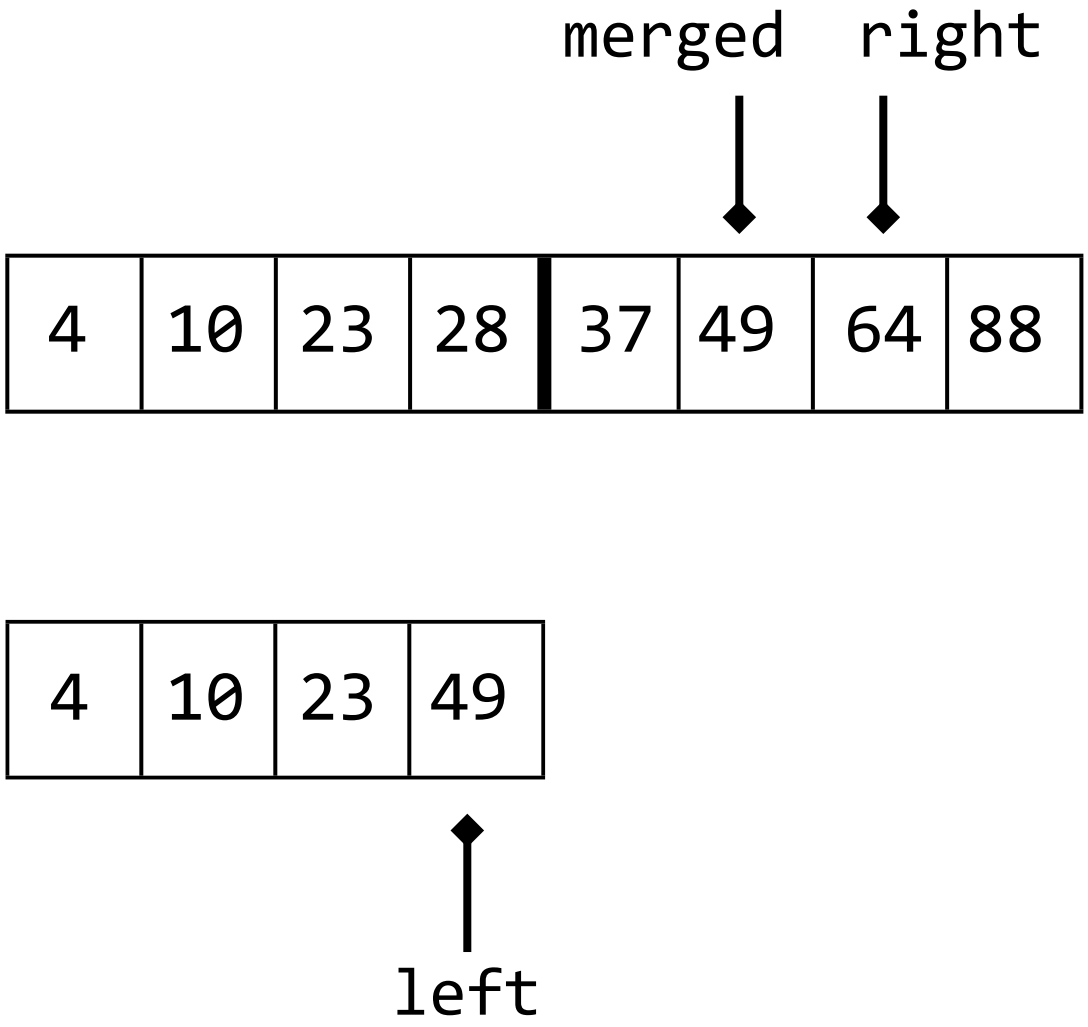

merged
right
4
10
23
28
28
37
64
88
4
10
23
49
left

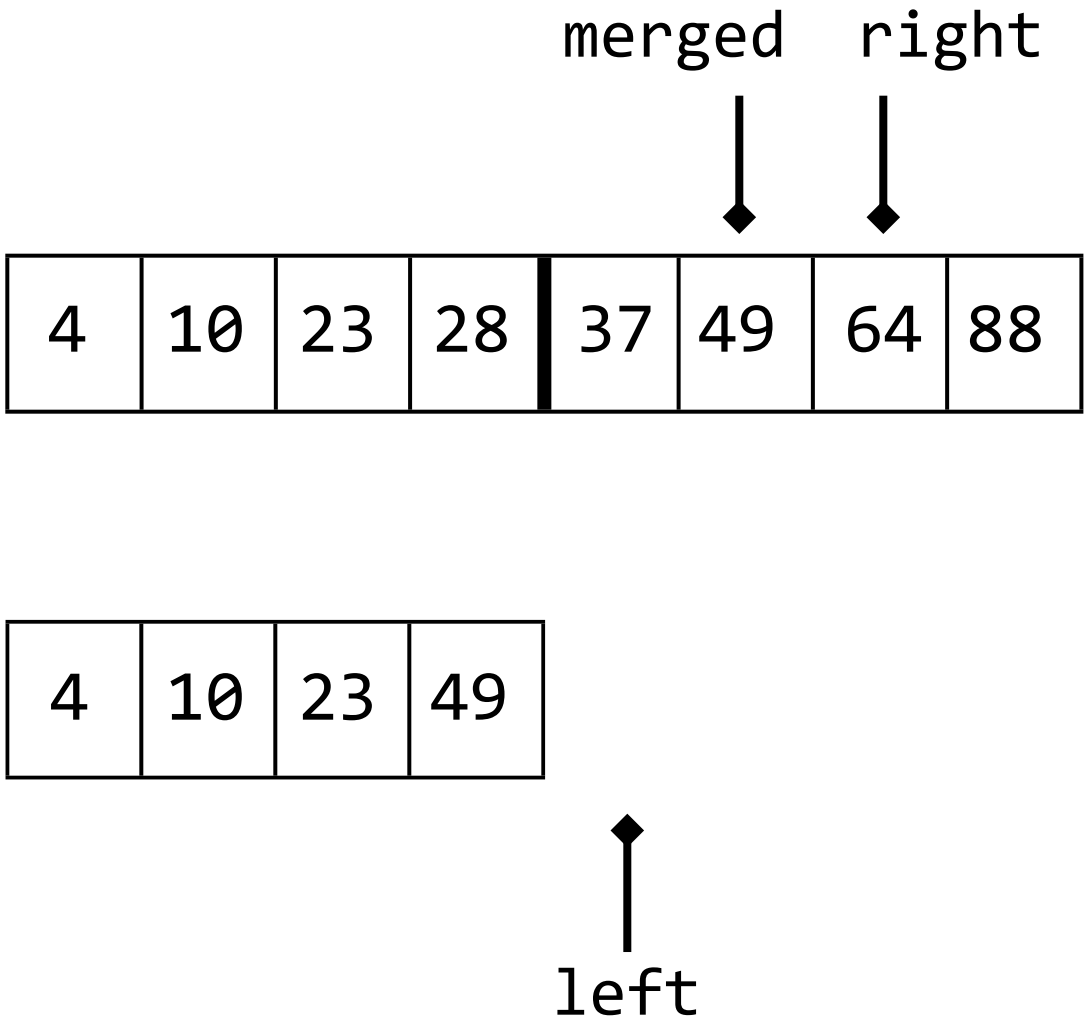

merged
right
4
10
23
28
37
64
88
4
10
23
49
left

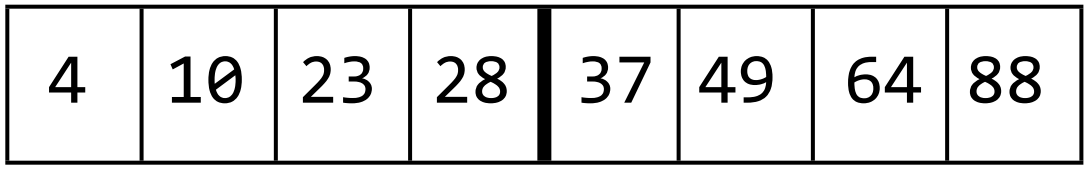

merged
right
4
10
23
28
37
37
64
88
4
10
23
49
left


merged
right
4
10
23
28
37
37
64
88
4
10
23
49
left

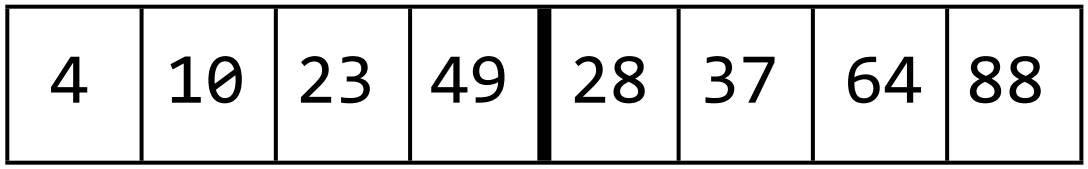

merged
right
4
10
23
28
37
64
88
4
10
23
49
left

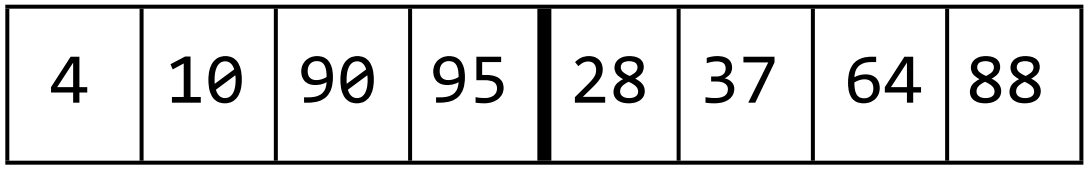

merged
right
4
10
23
28
37
49
64
88
4
10
23
49
left

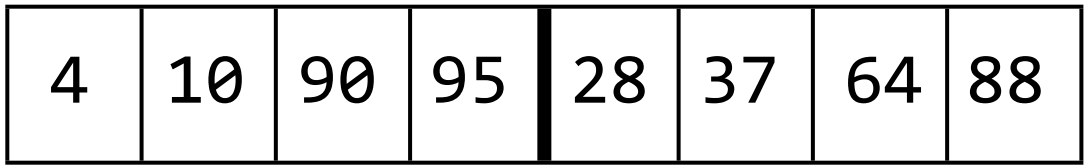

merged
right
4
10
23
28
37
49
64
88
4
10
23
49
left

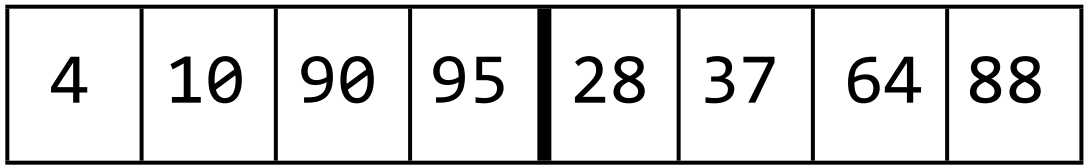

4
10
23
28
37
49
64
88

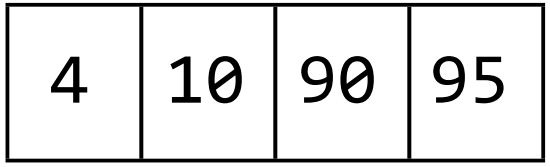
```
int *temp = malloc( n * sizeof(int) );
for (int i = low; i <= mid; i++)
{
  temp[i] = list[i];
}

int left = low;
int right = mid + 1;
int merged = low;
while (left <= mid && right <= high)
{
  if (list[right] < temp[left])
  {
    list[merged] = list[right];
    right++;
  }
  else
  {
    list[merged] = temp[left];
    left++;
  }
  merged++;
}

free(temp);
```

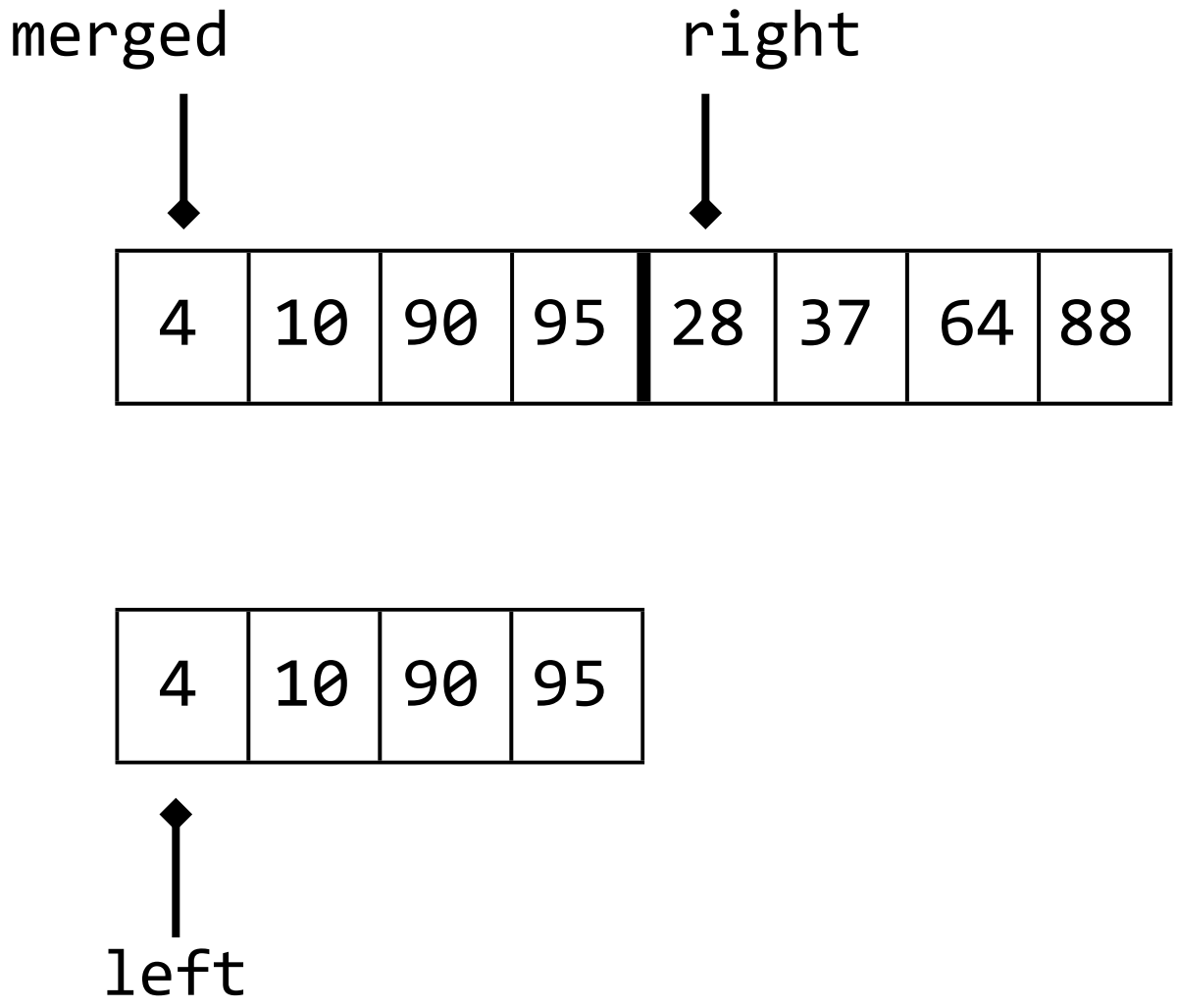

4
10
23
49
28
37
64
88

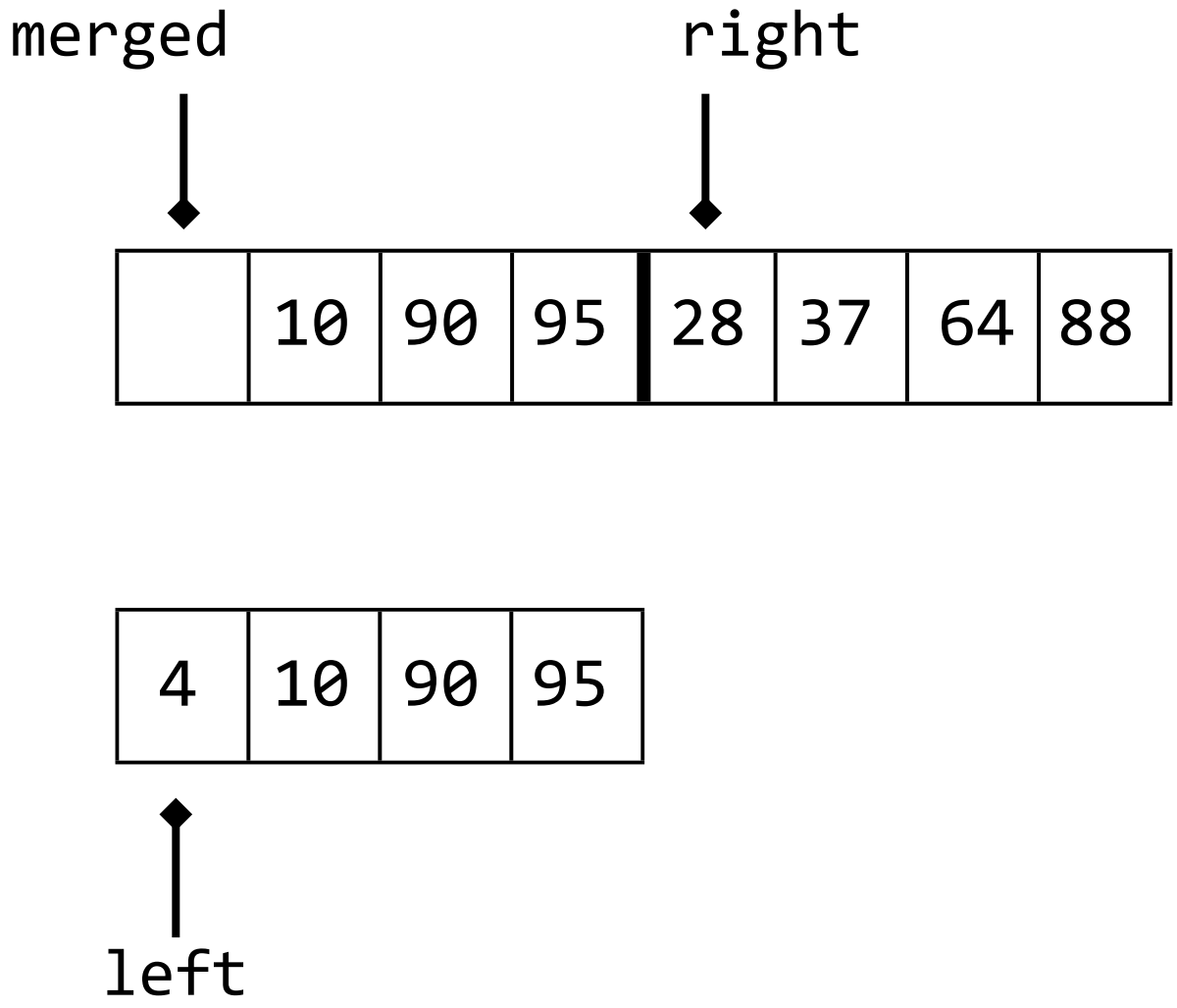

4
10
90
95
28
37
64
88

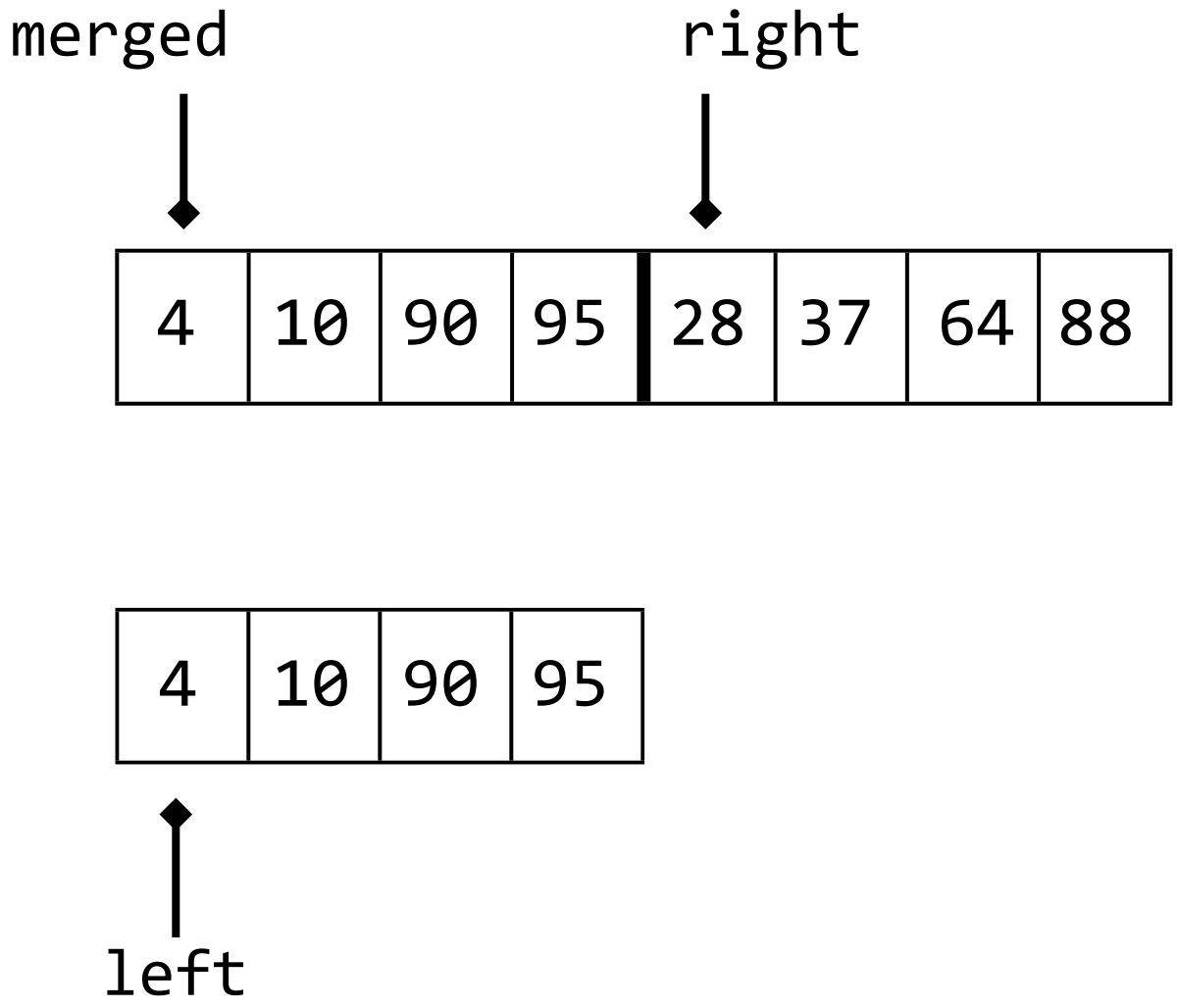

4
10
90
95
28
37
64
88

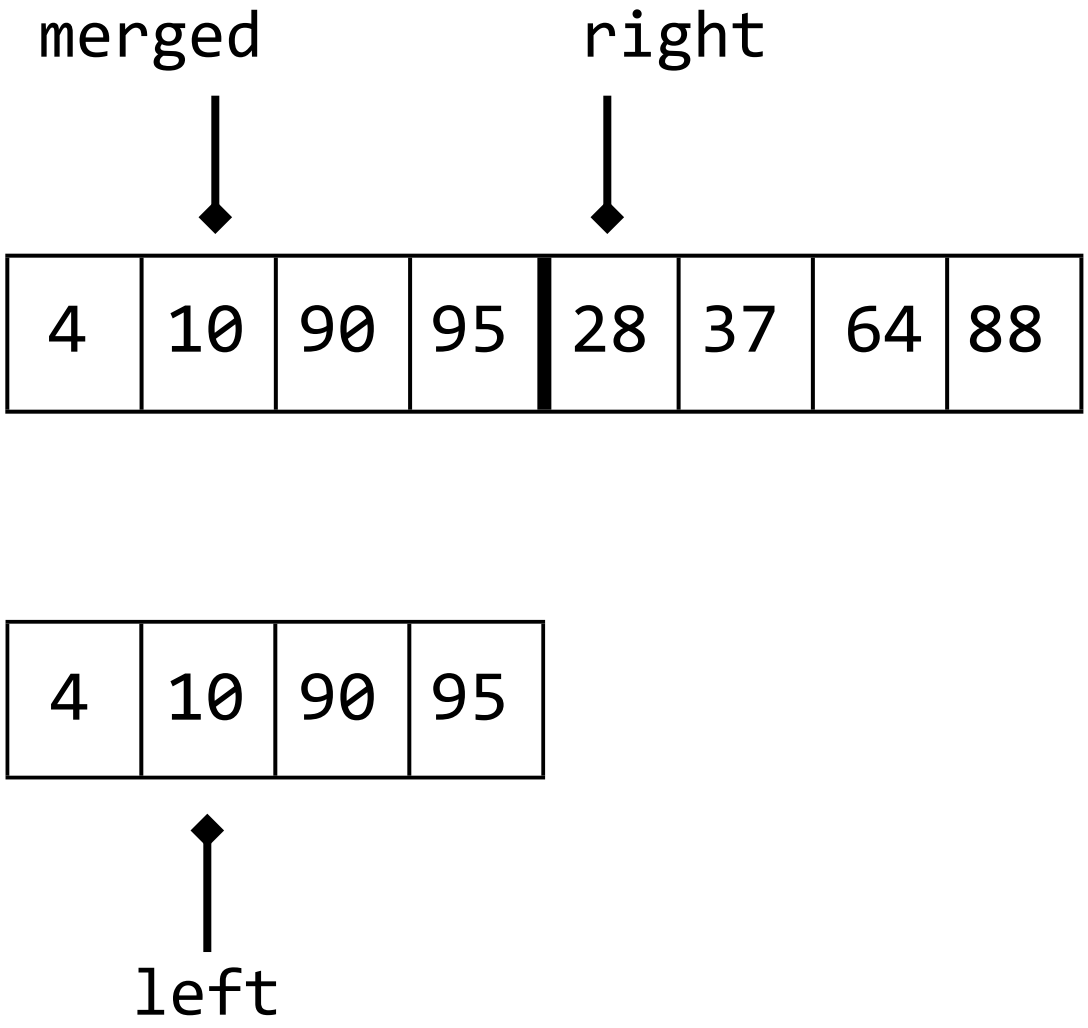

4
10
90
95
28
37
64
88

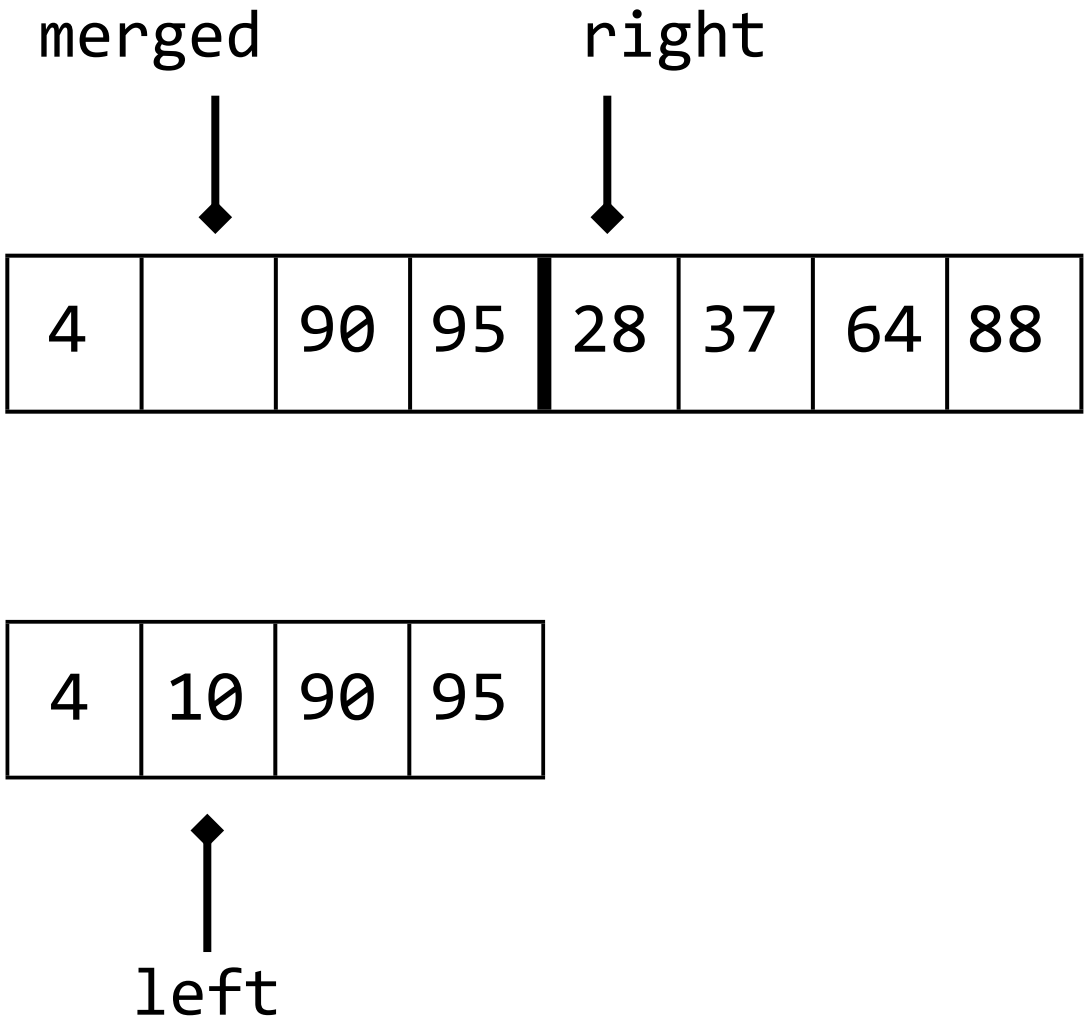

4
10
90
95

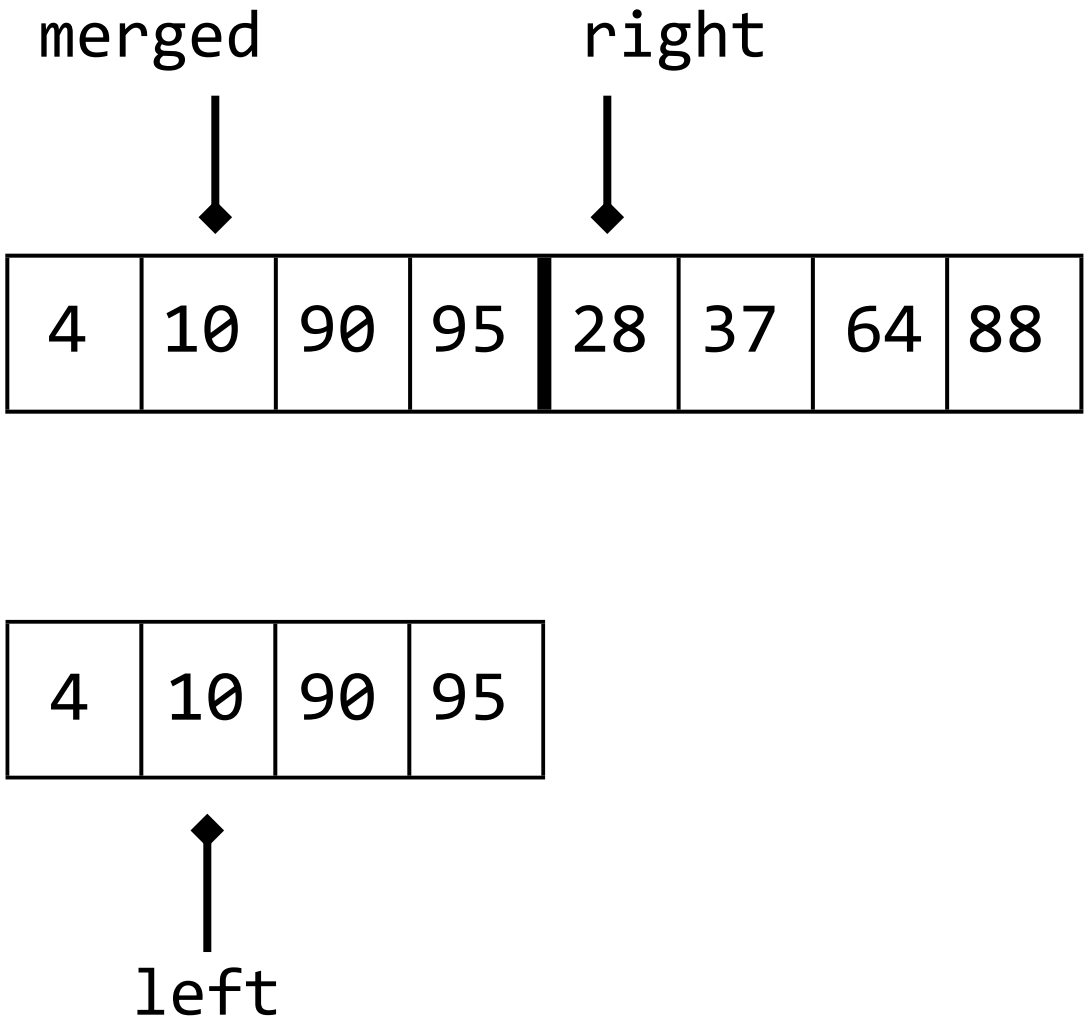

merged
right
4
10
90
95
28
37
64
88
4
10
90
95
left

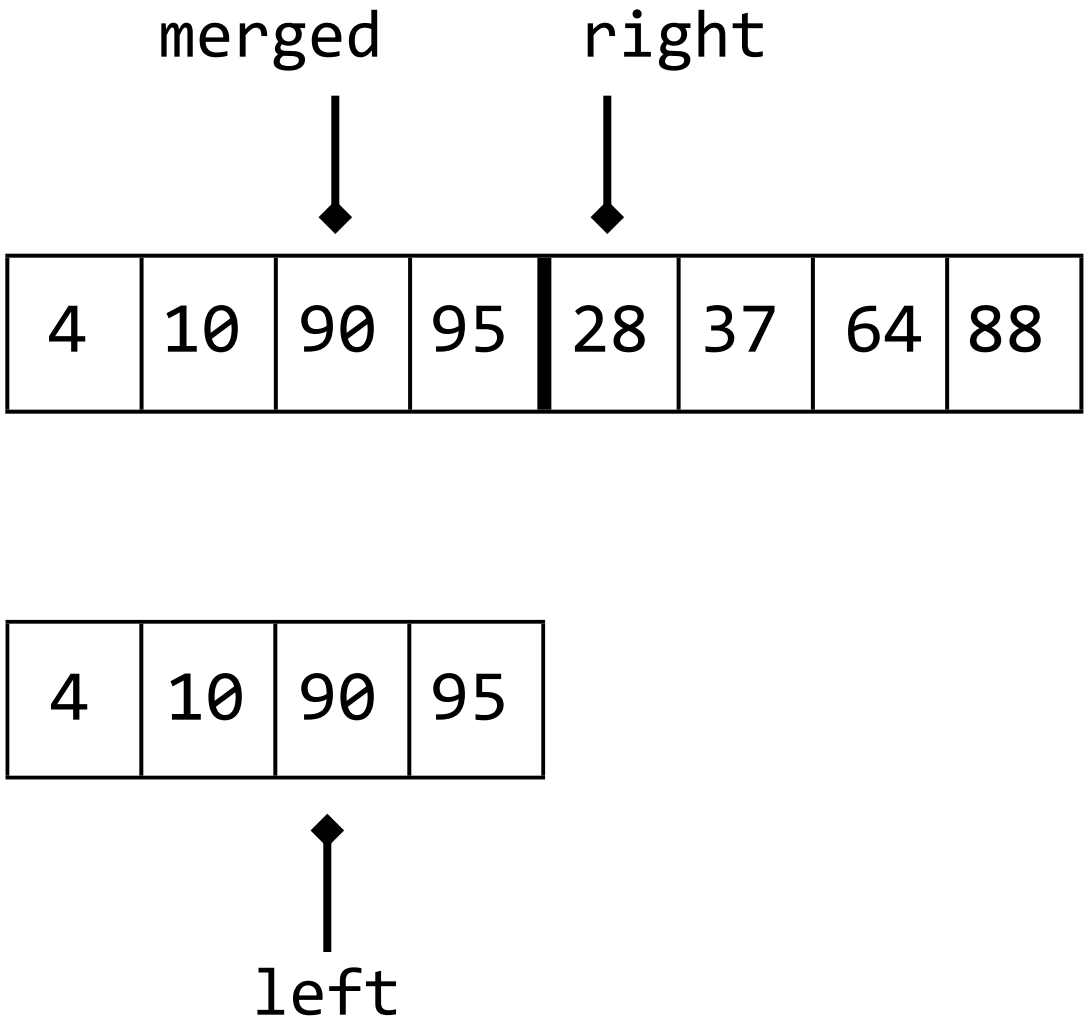

merged
right
10
90
95
28
37
64
88
4
10
90
95
left

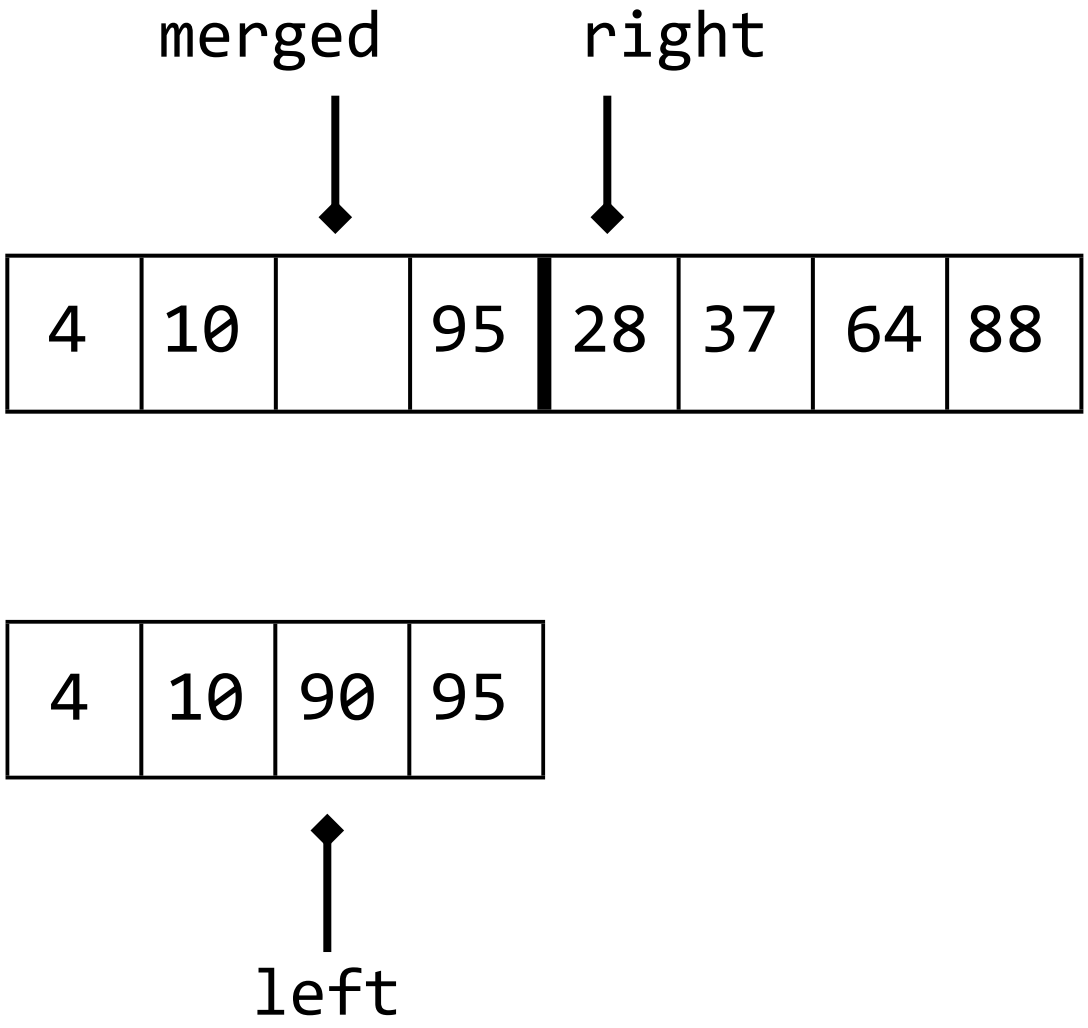

merged
right
4
10
90
95
28
37
64
88
4
10
90
95
left

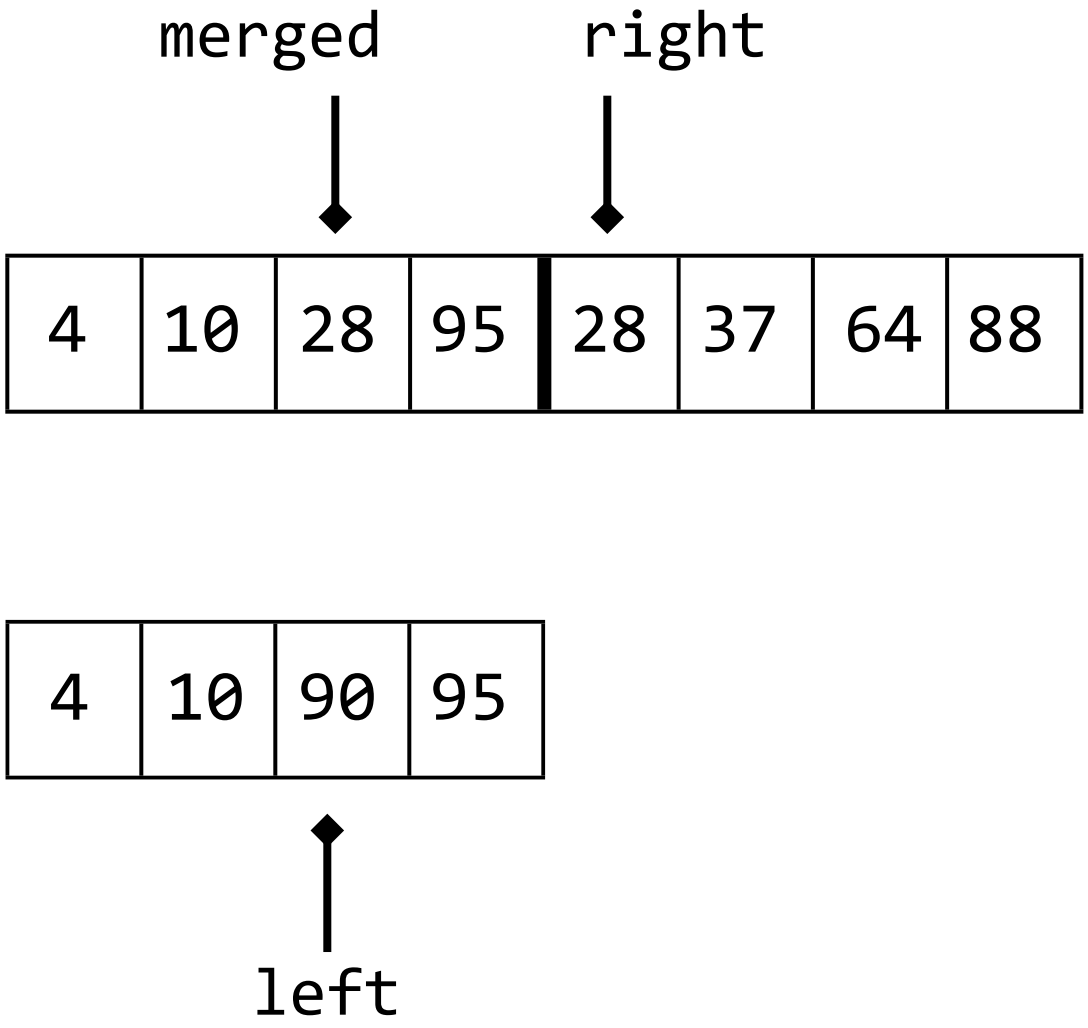

merged
right
4
10
90
95
28
37
64
88
4
10
90
95
left

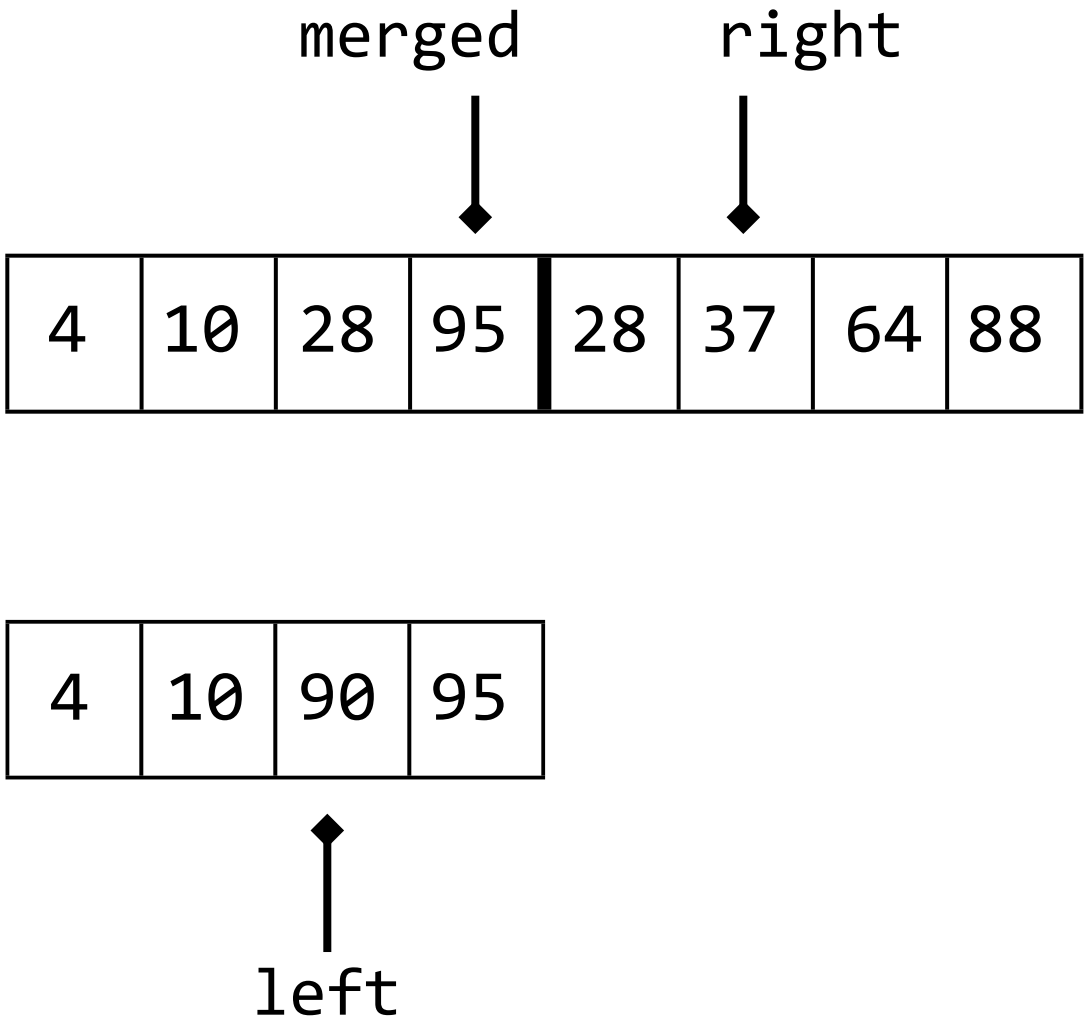

merged
right
4
90
95
28
37
64
88
4
10
90
95
left

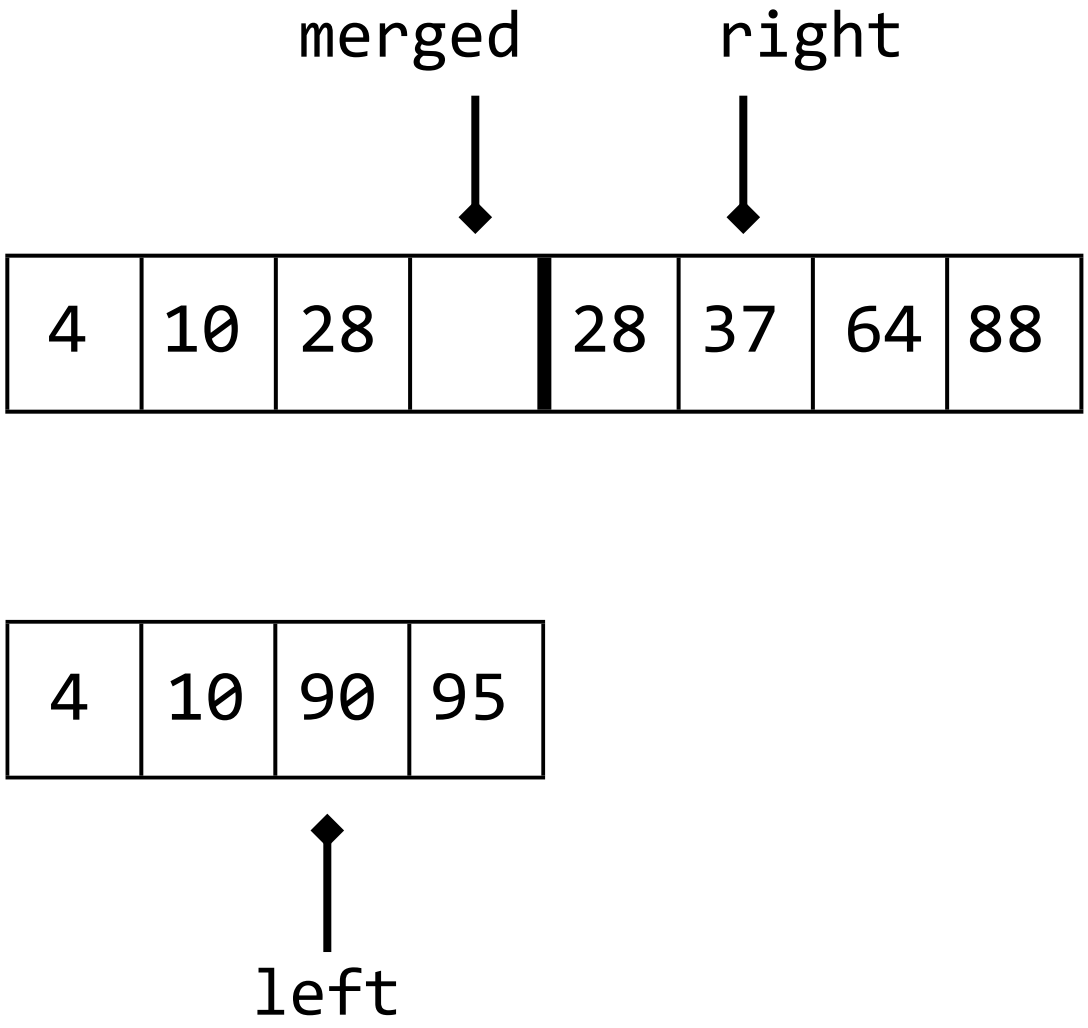

merged
right
4
10
90
95
28
37
64
88
4
10
90
95
left

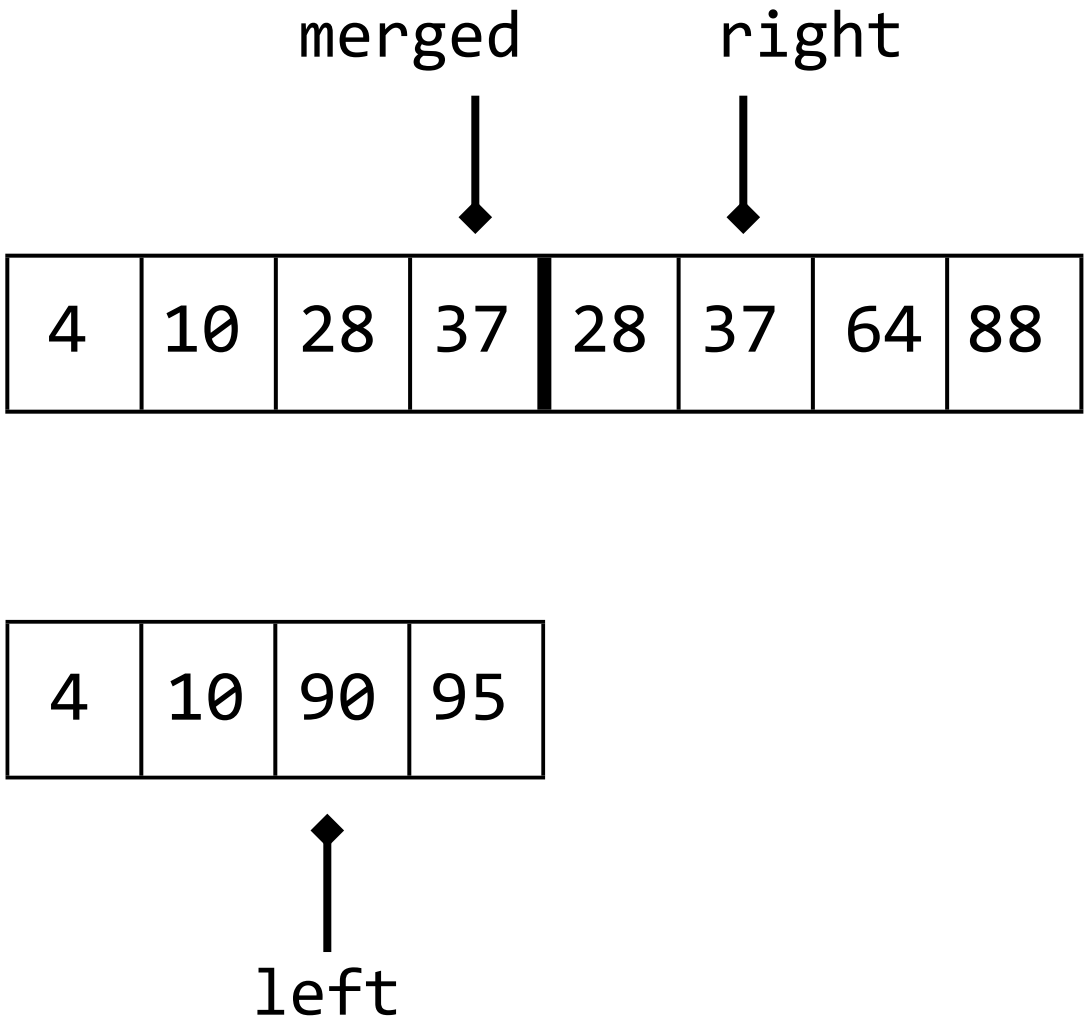

merged
right
4
10
90
95
28
37
64
88
4
10
90
95
left

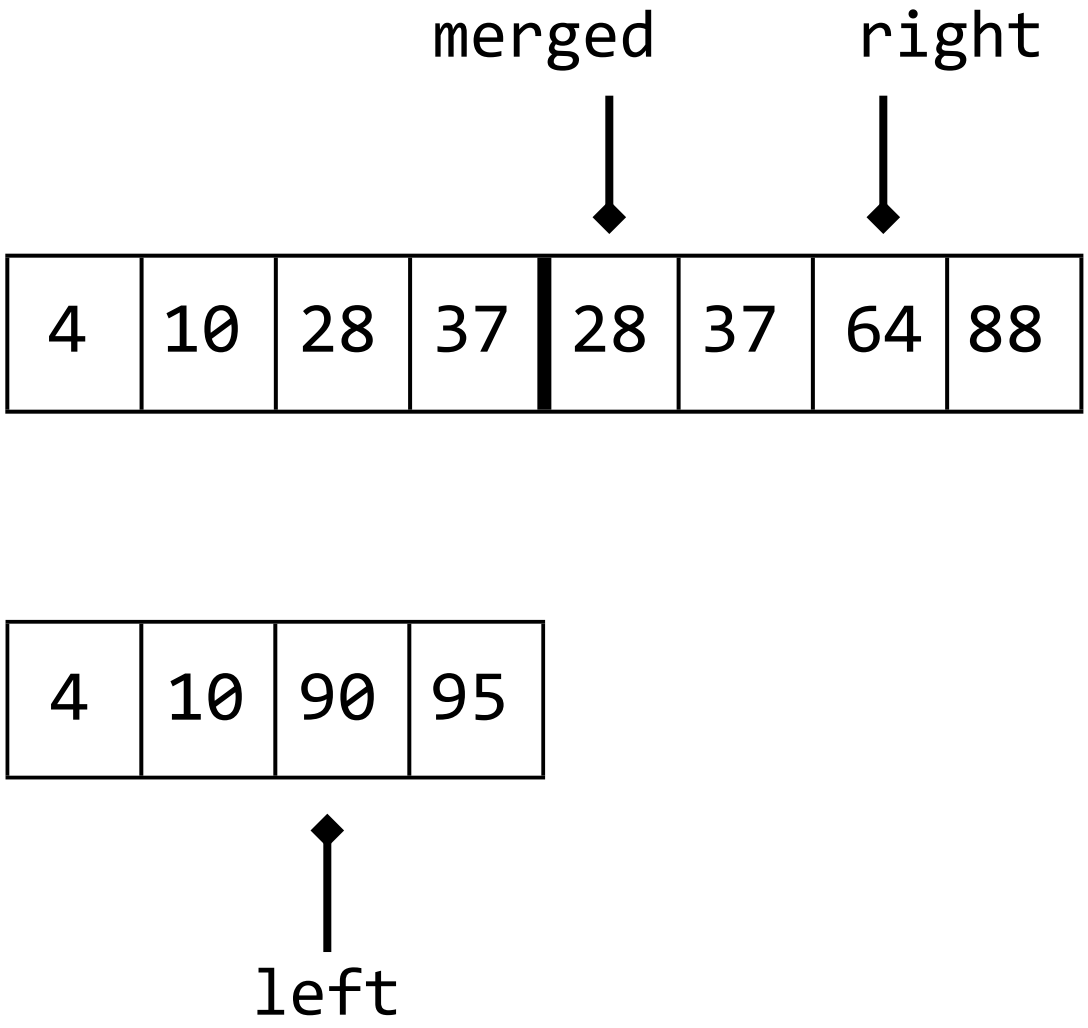

merged
right
4
10
95
28
37
64
88
4
10
90
95
left

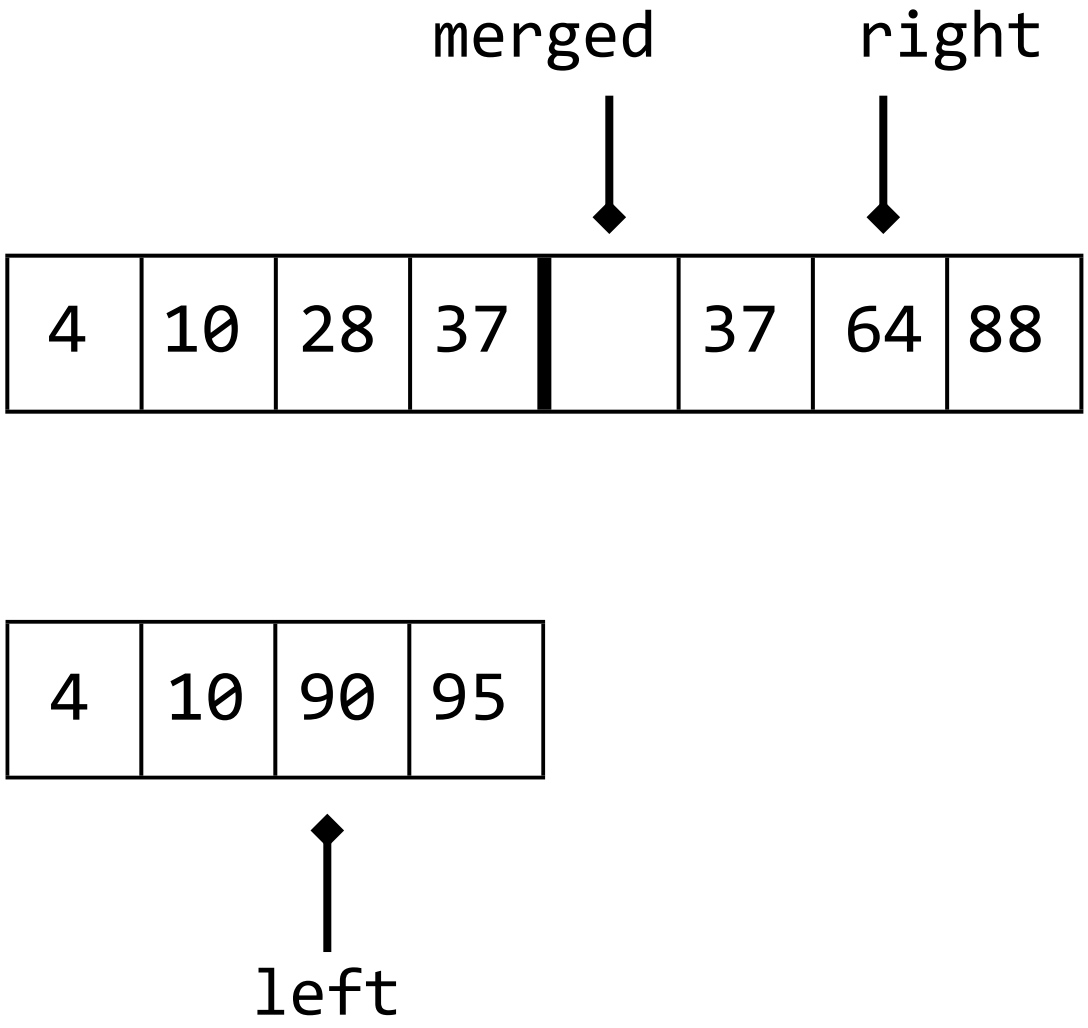

merged
right
4
10
28
95
28
37
64
88
4
10
90
95
left

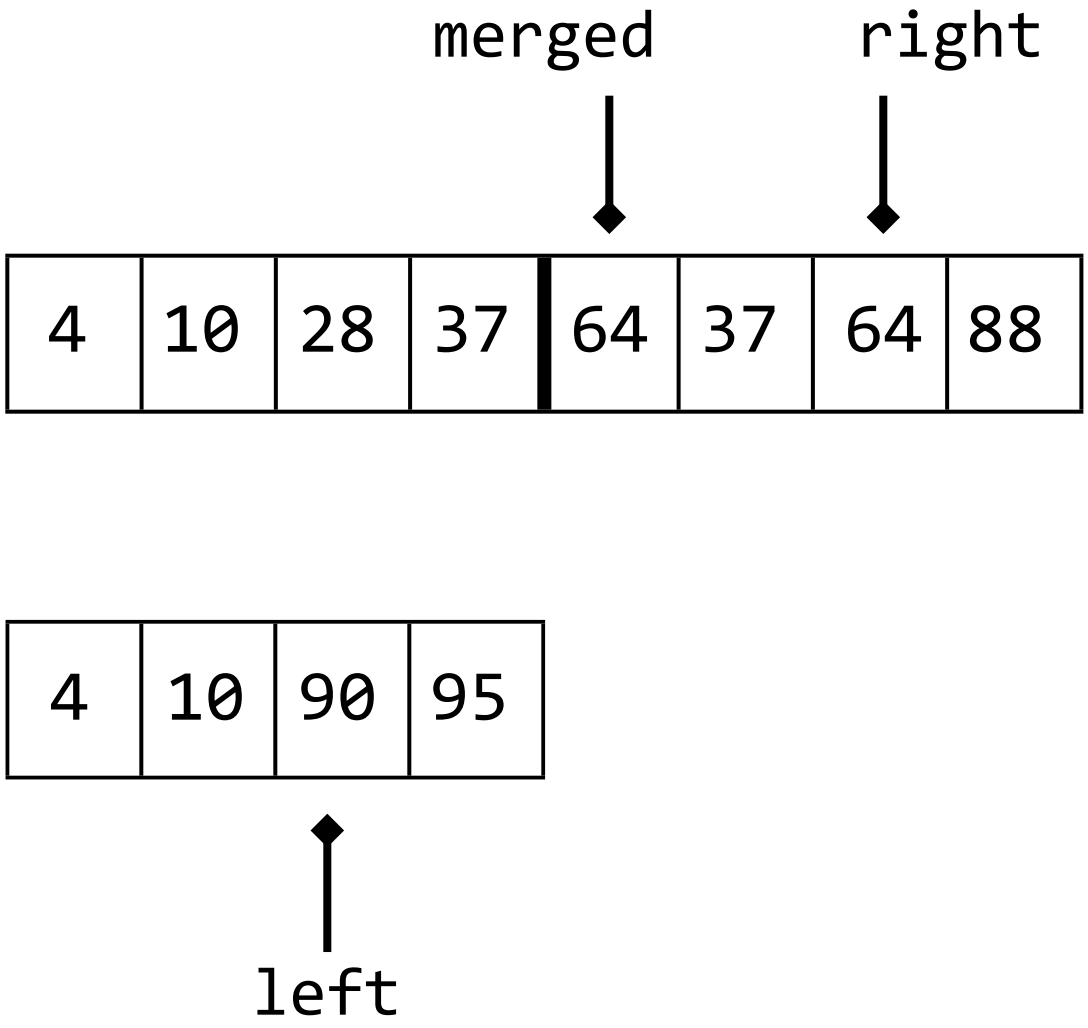

merged
right
4
10
28
95
28
37
64
88
4
10
90
95
left

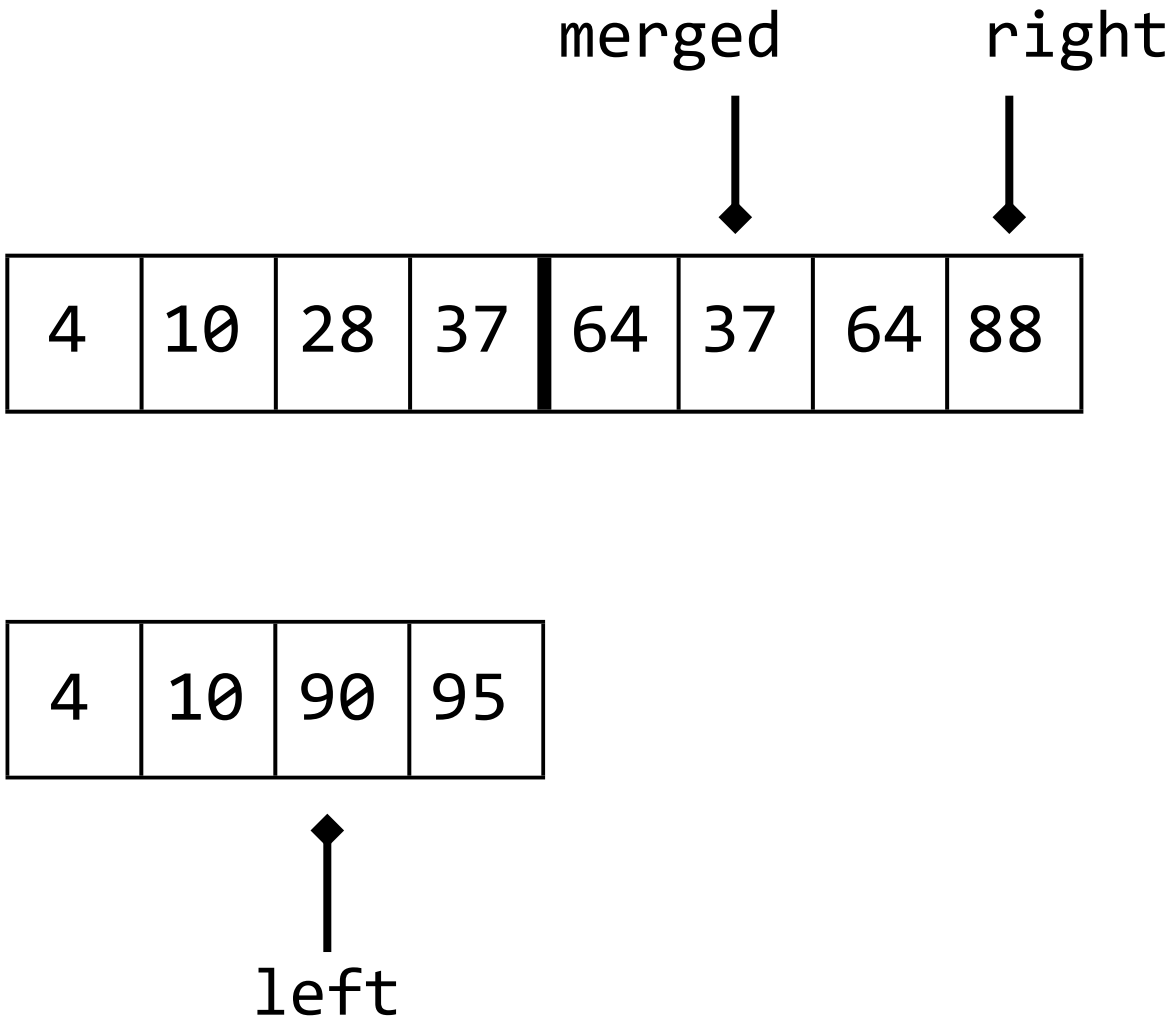

merged
right
4
10
28
28
37
64
88
4
10
90
95
left

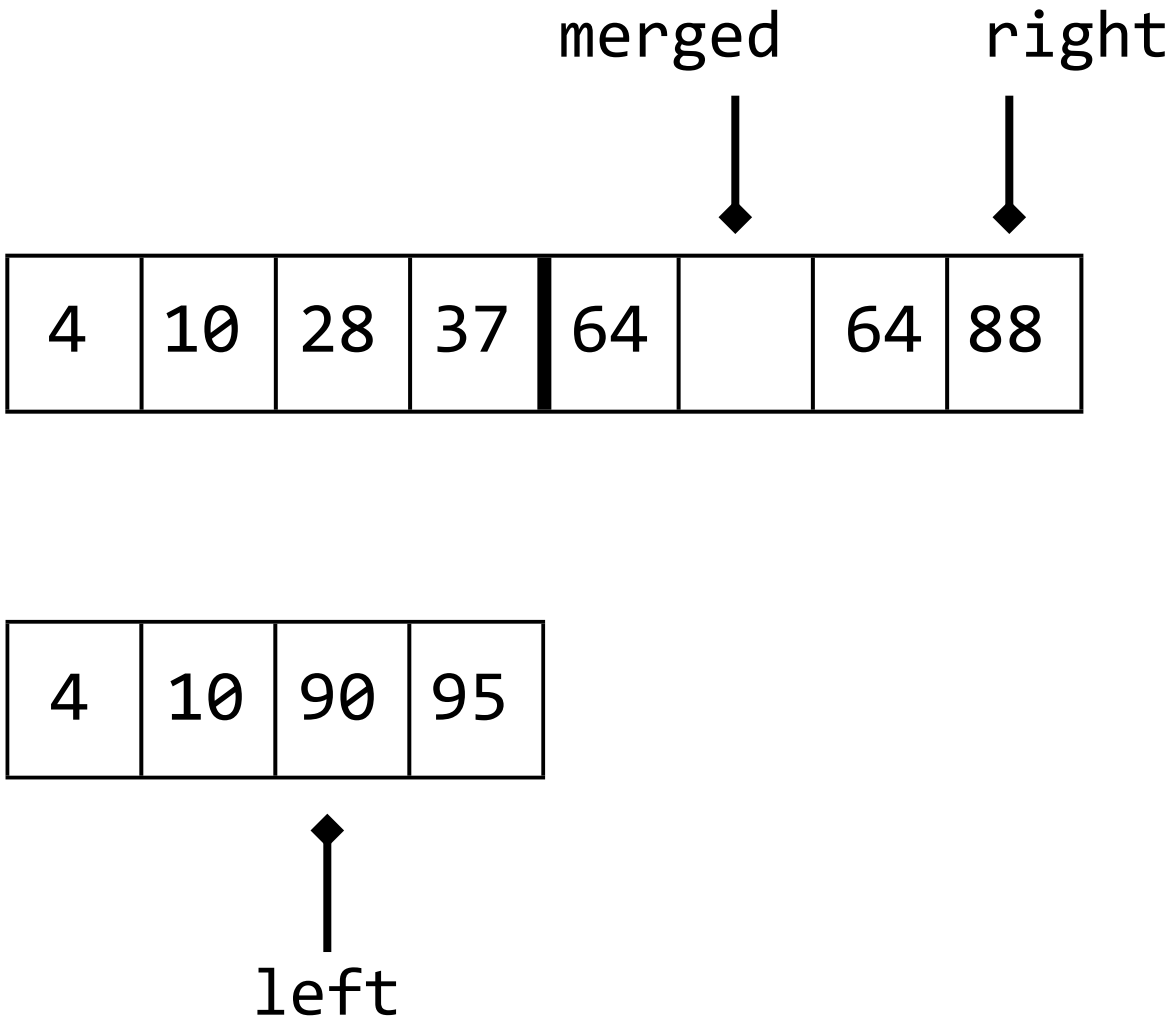

merged
right
4
10
28
37
28
37
64
88
4
10
90
95
left

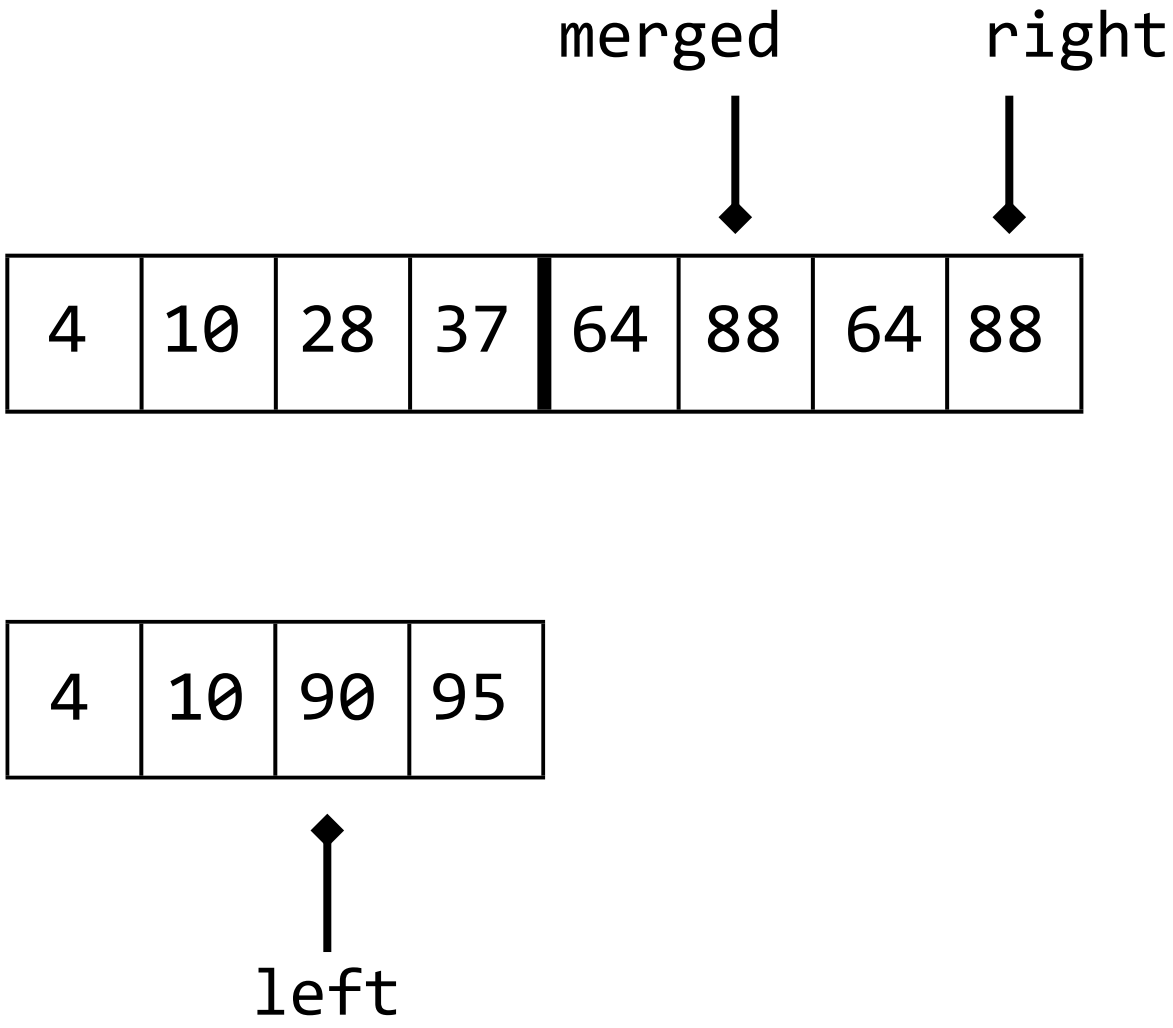

merged
right
4
10
28
37
28
37
64
88
4
10
90
95
left

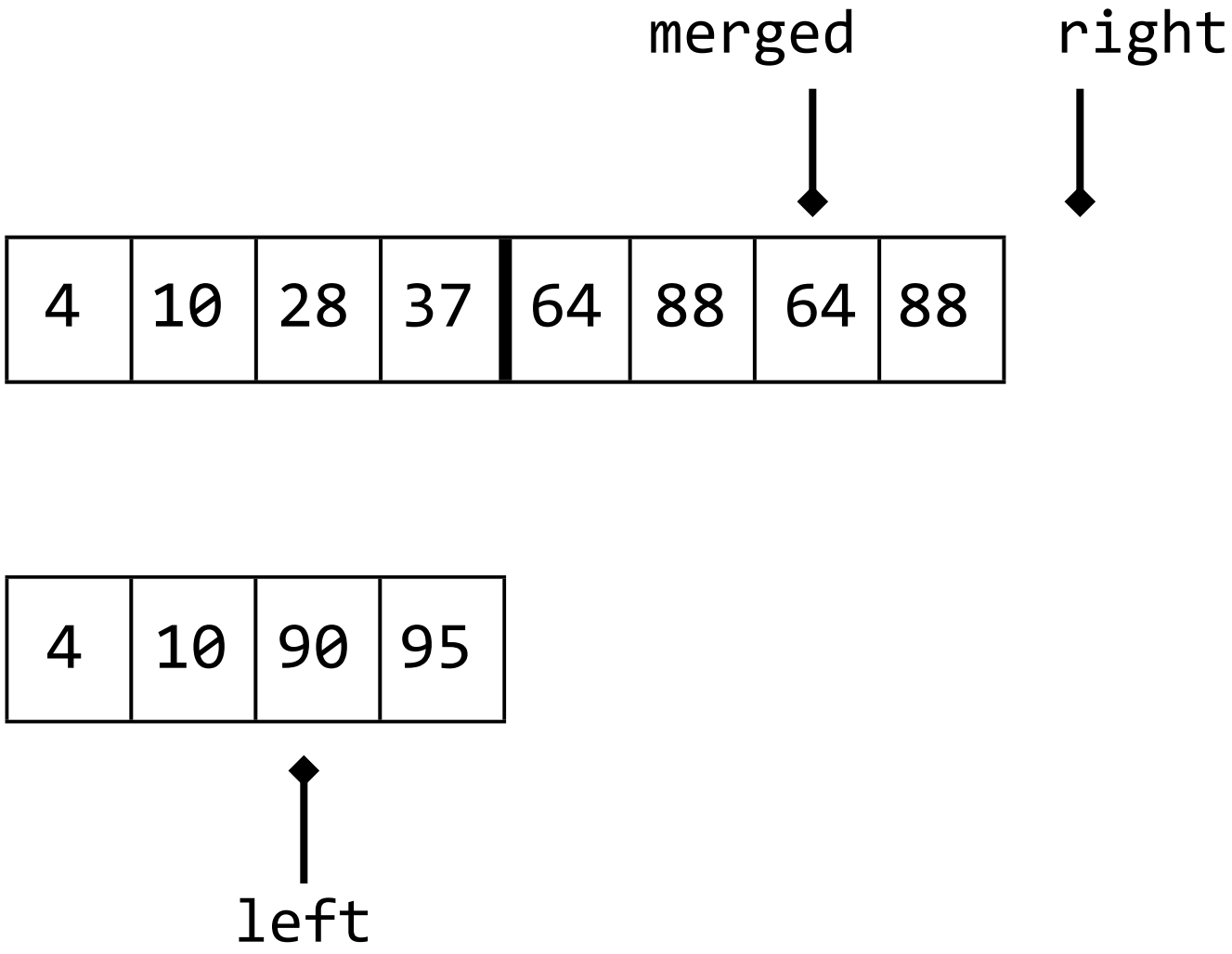

merged
right
4
10
28
37
37
64
88
4
10
90
95
left

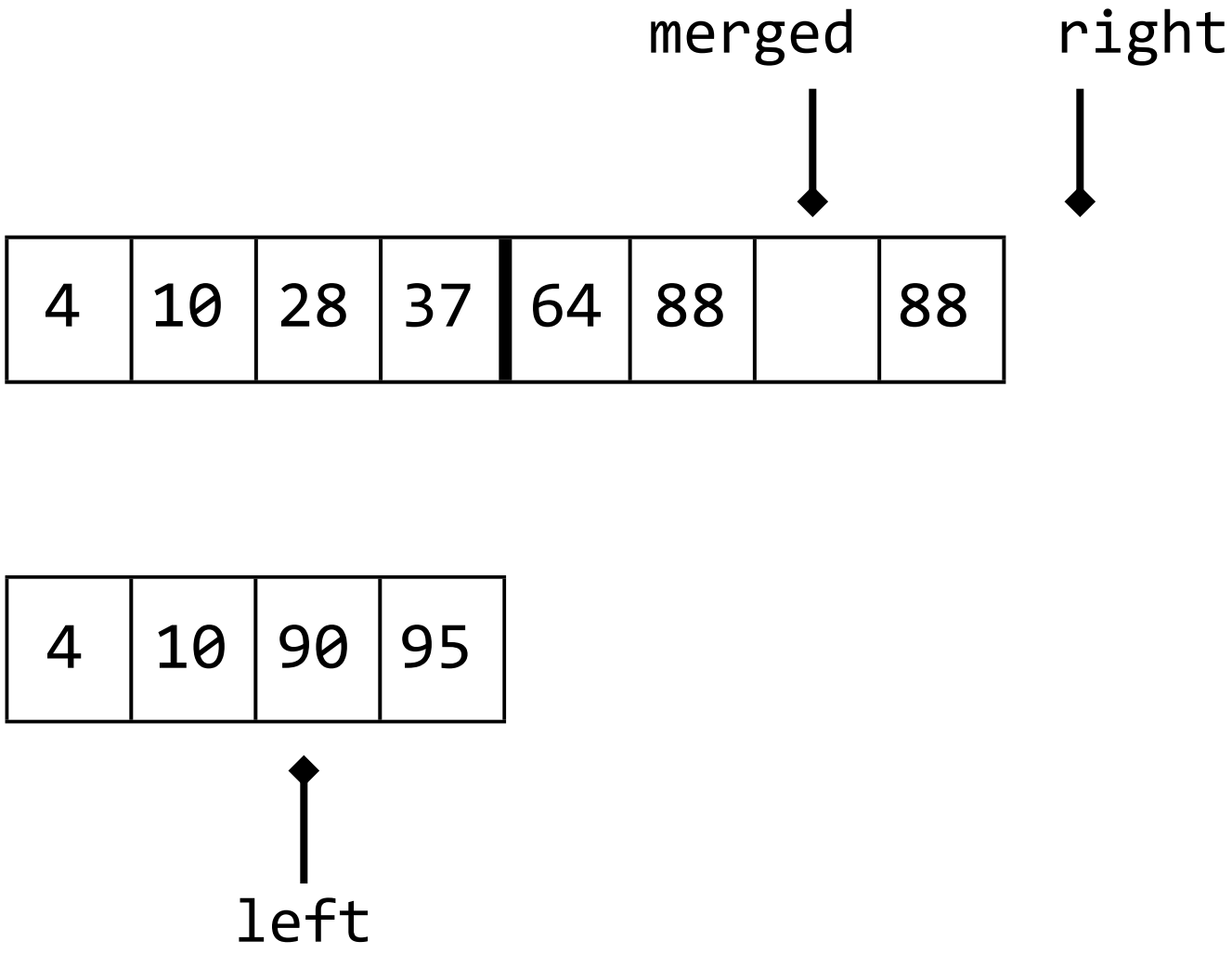

merged
right
4
10
28
37
64
37
64
88
4
10
90
95
left

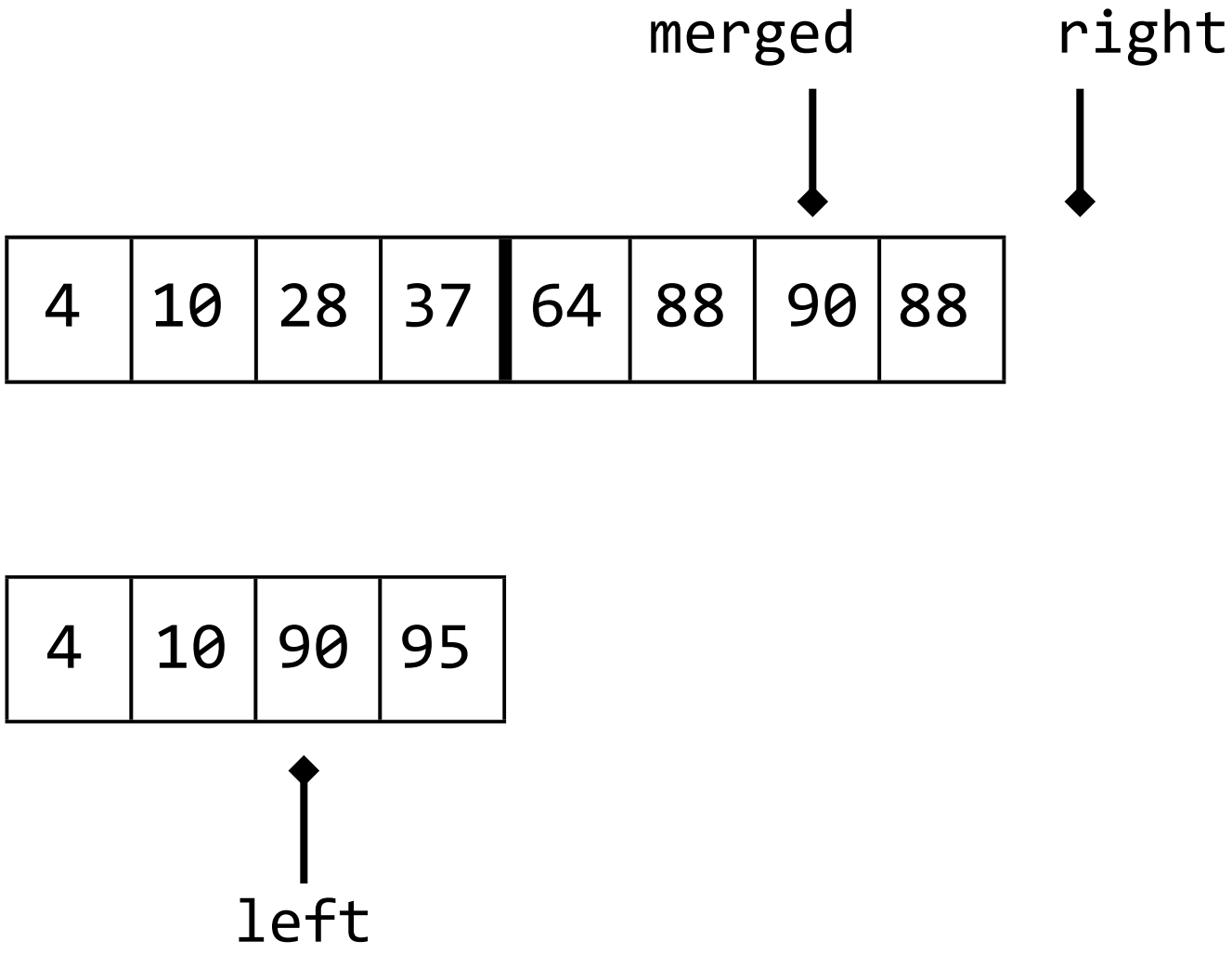

merged
right
4
10
28
37
64
37
64
88
4
10
90
95
left

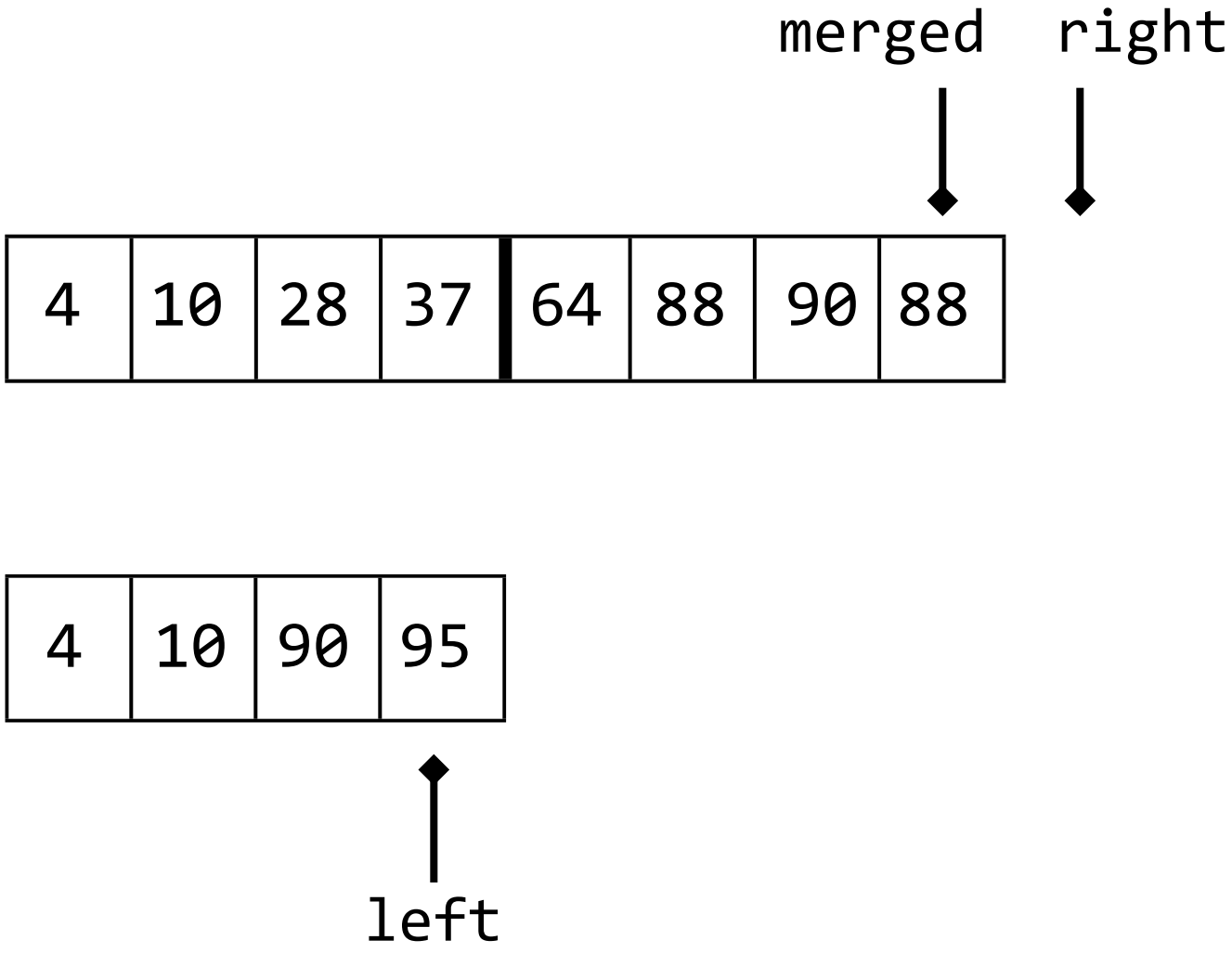

merged
right
4
10
28
37
64
64
88
4
10
90
95
left

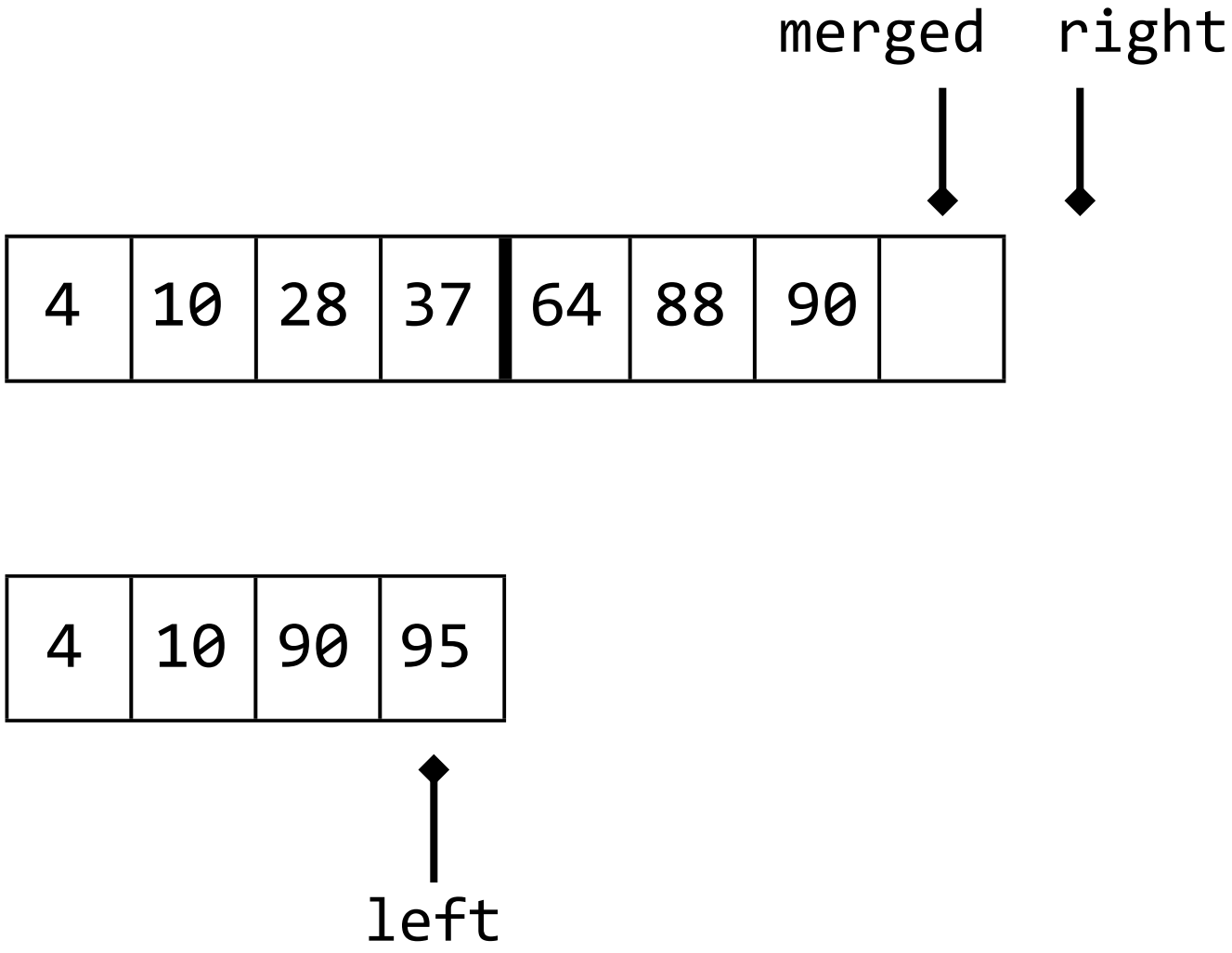

merged
right
4
10
28
37
64
88
64
88
4
10
90
95
left

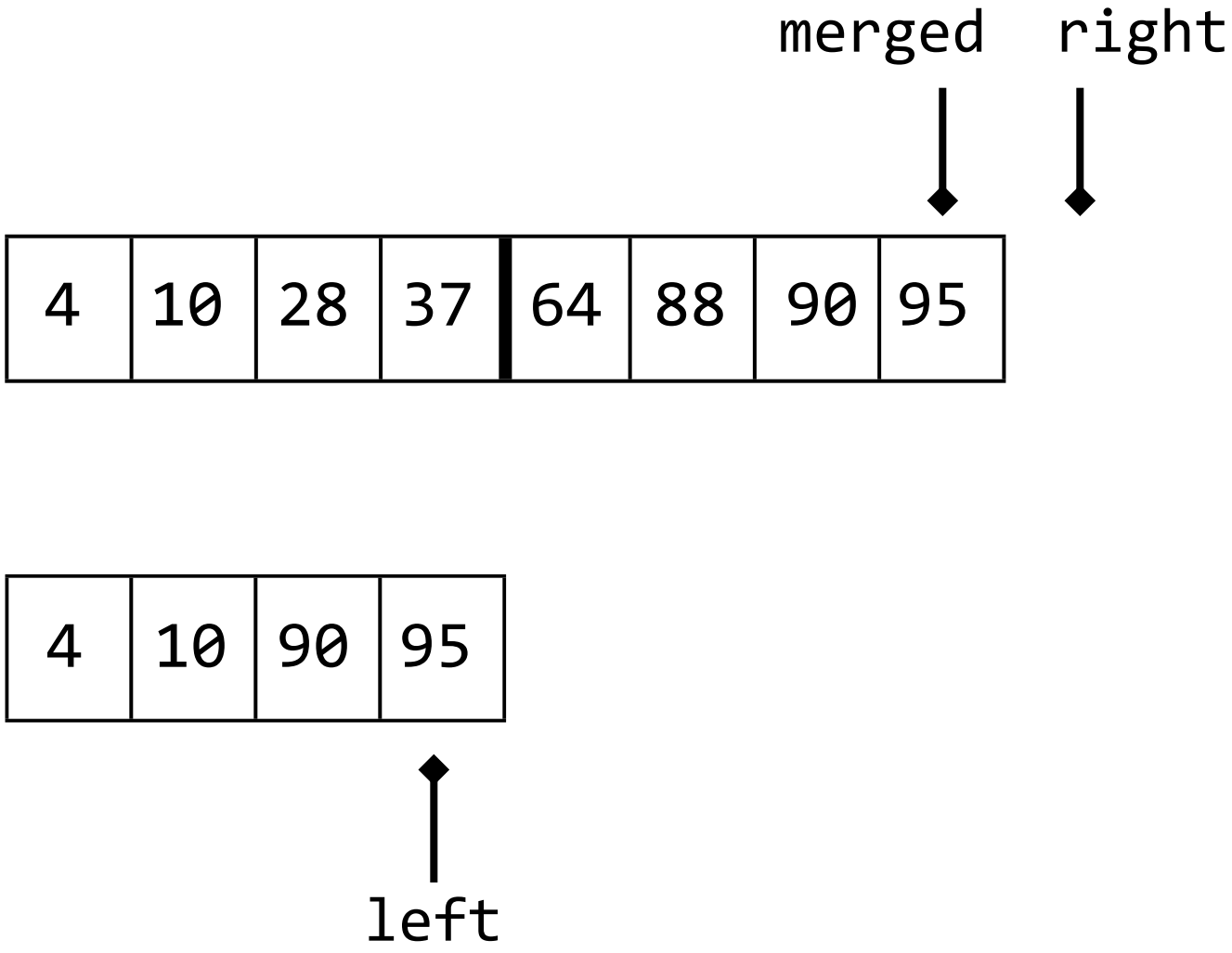

merged
right
4
10
28
37
64
88
64
88
4
10
90
95
left

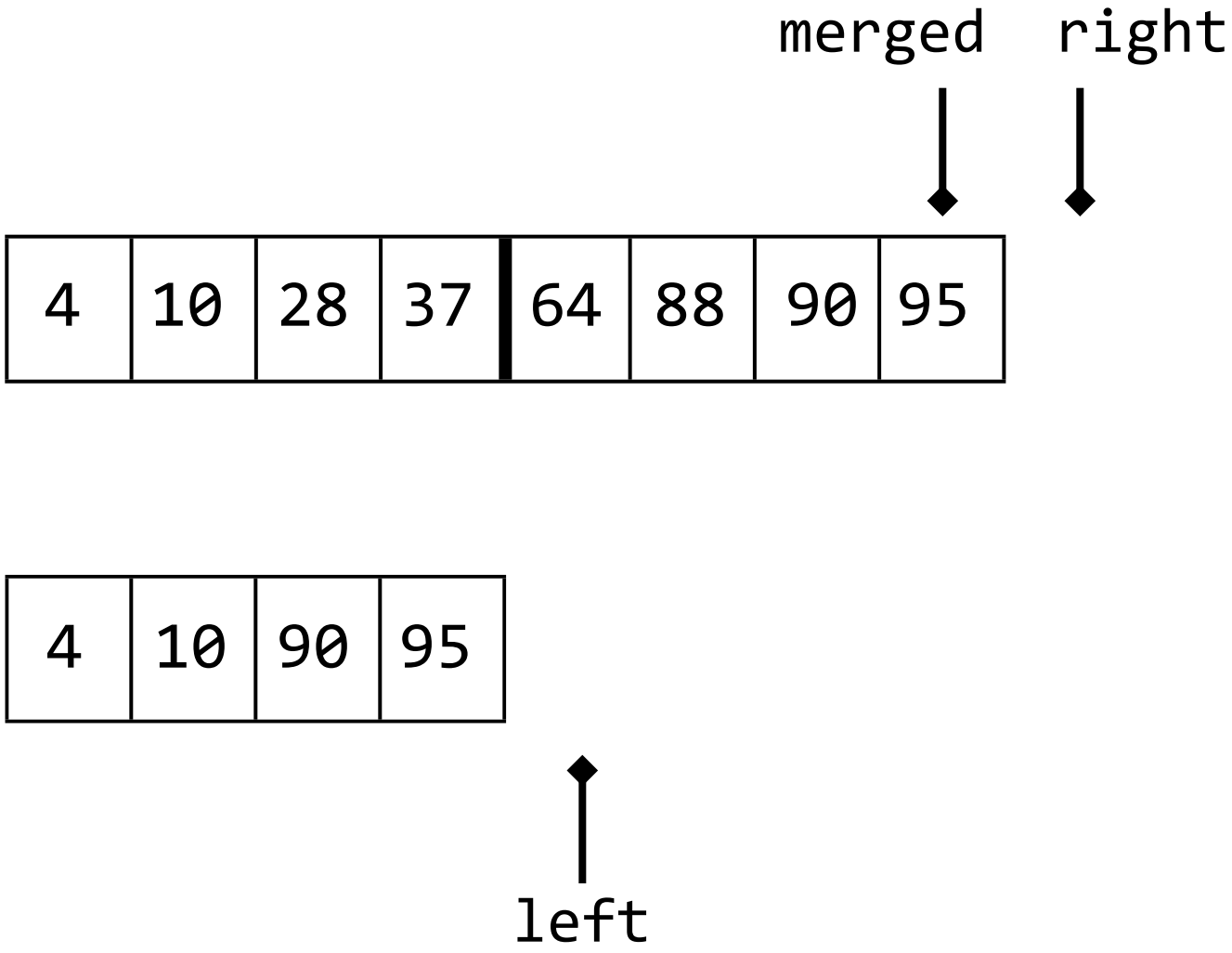

merged
right
4
10
28
37
64
88
88
4
10
90
95
left


merged
right
4
10
28
37
64
88
90
88
4
10
90
95
left


merged
right
4
10
28
37
64
88
90
88
4
10
90
95
left


merged
right
4
10
28
37
64
88
90
4
10
90
95
left


merged
right
4
10
28
37
64
88
90
95
4
10
90
95
left


merged
right
4
10
28
37
64
88
90
95
4
10
90
95
left

```
while (left <= mid && right <= high)
{
    ...
}


while (left <= mid)
{
    list[merged] = temp[left];
    merged++;
    left++;
}
```

```
void mergeSort(int list[], int n, int low, int high)
{
    if (high - low >= 1)
    {
        int mid = (high + low) / 2;

        mergeSort(list, n, low, mid);
        mergeSort(list, n, mid + 1, high);

        int *temp = malloc( n * sizeof(int) );

        for (int i = low; i <= mid; i++)
        {
            temp[i] = list[i];
        }

        int left = low;
        int right = mid + 1;
        int merged = low;
        while (left <= mid && right <= high)
        {
            if (list[right] < temp[left])
            {
```

```
int left = low;
int right = mid + 1;
int merged = low;
while (left <= mid && right <= high)
{
    if (list[right] < temp[left])
    {
        list[merged] = list[right];
        right++;
    }
    else
    {
        list[merged] = temp[left];
        left++;
    }
    merged++;
}
while (left <= mid)
{
    list[merged] = temp[left];
    merged++;
    left++;
}

free(temp);
```

```
                {
                    list[merged] = list[right];
                    right++;
                }
                else
                {
                    list[merged] = temp[left];
                    left++;
                }
                merged++;
            }
            while (left <= mid)
            {
                list[merged] = temp[left];
                merged++;
                left++;
            }

            free(temp);
        }
    }
```

Merge Sort Performance

- This is not an optimal merge sort
  - For example, we're allocating a new (unnecessarily large) array each iteration

Merge Sort Performance

- Merging the lists is the time consuming part
- To merge two lists of a total of n elements, it's at most n comparisons
- Merge sort splits the list in half each time, so it does approximately $\log_2(n)$ splits
- So merge sort does ~ $\log(n)$ repetitions of a task that takes an n amount of time
- Which makes the running time ~ n * $\log(n)$

n * log(n)

- Our simple sorts were all n^2
- This is n * log(n)

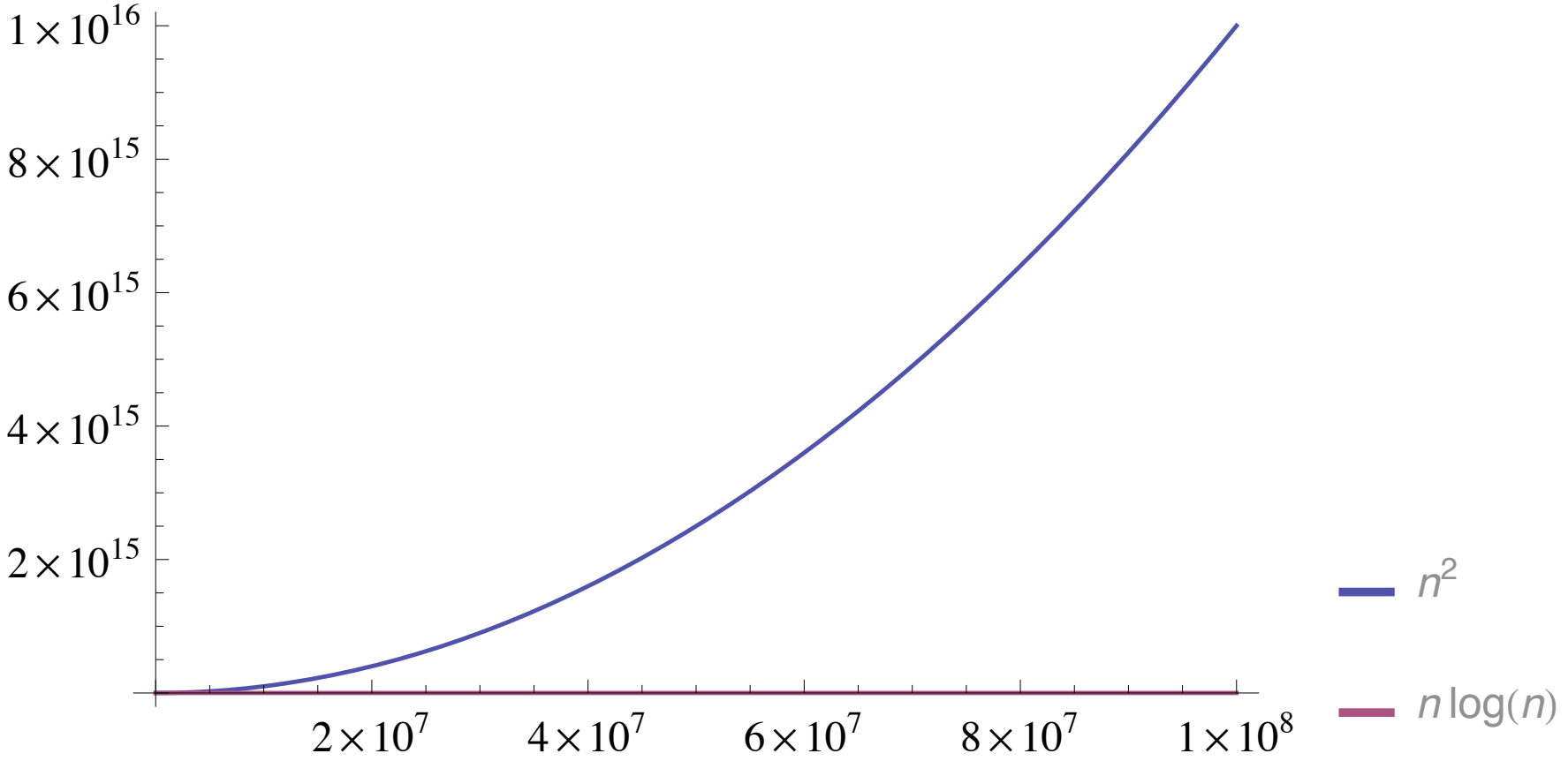
n	*n * log(n)*	n^2
10	~30	100
100	~700	10,000
1,000	~10,000	1,000,000
10,000	~130,000	100,000,000
100,000	~1,660,000	10,000,000,000
100,000,000	~2,700,000,000	10,000,000,000,000,000
	~ 3 seconds	~ 12 days

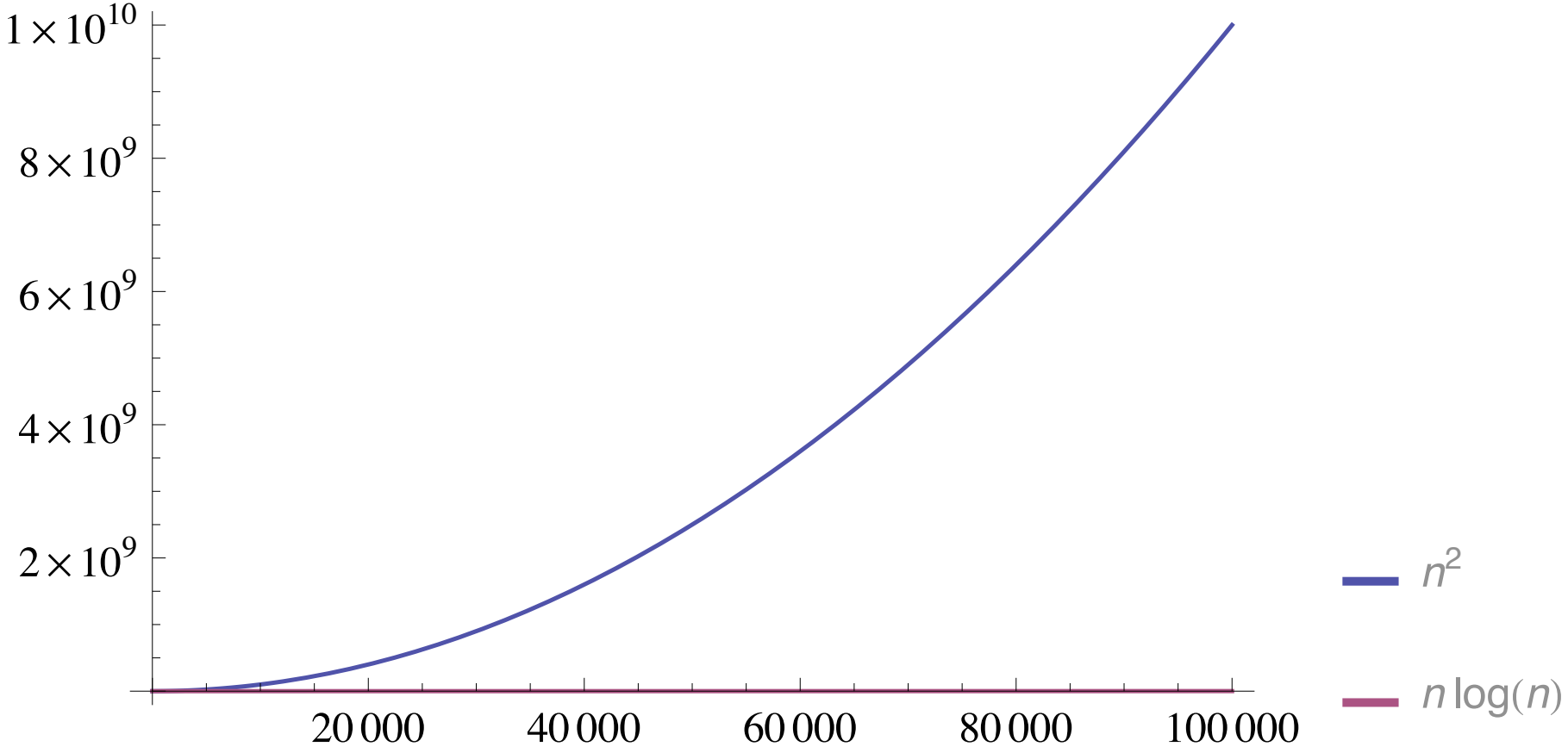
n * log(n)

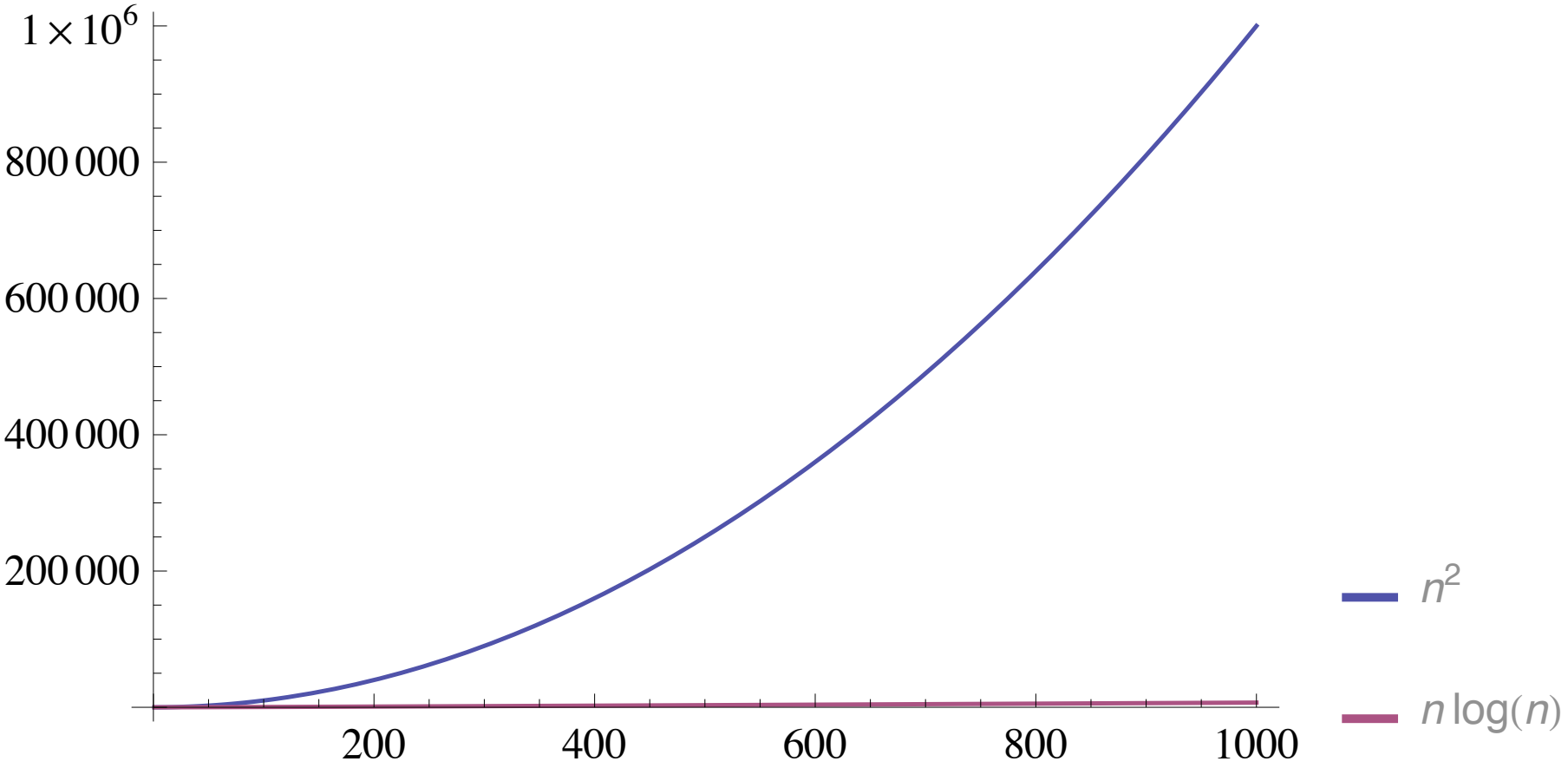
- Our simple sorts were all n^2
- This is n * log(n)

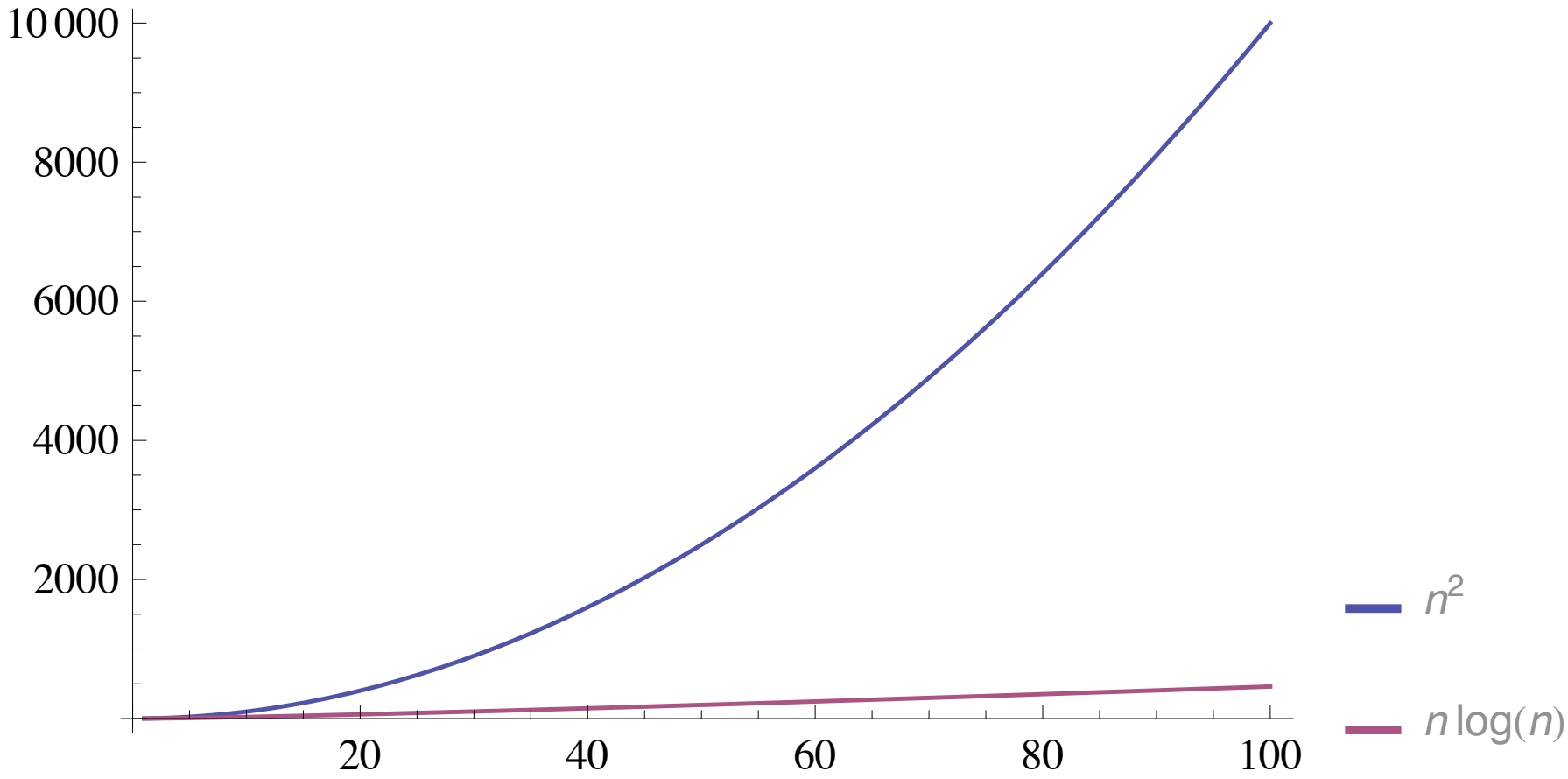
$$n * n - \sum_{i=1}^{n-1} i$$

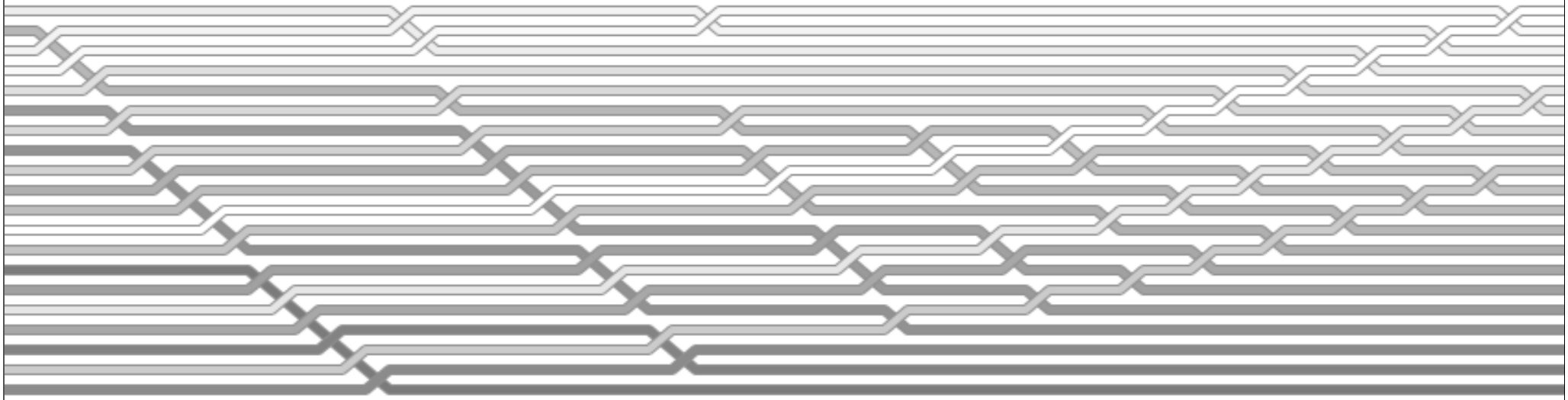
n	*n * log(n)*	n^2	$n * n - \sum_{i=1}^{n-1} i$
10	~30	100	~60
100	~700	10,000	~5,000
1,000	~10,000	1,000,000	~500,000
10,000	~130,000	100,000,000	~50,000,000
100,000	~1,660,000	10,000,000,000	~5,000,000,000
100,000,000	~2,700,000,000	10,000,000,000,000,000	~5,000,000,000,000,000
	~ 3 seconds	~ 12 days	~ 6 days


1×10^{16}
8×10^{15}
6×10^{15}
4×10^{15}
2×10^{15}
2×10^{7}
4×10^{7}
6×10^{7}
8×10^{7}
1×10^{8}
n^2
$n\log(n)$


1×10^{10}
8×10^{9}
6×10^{9}
4×10^{9}
2×10^{9}
20 000
40 000
60 000
80 000
100 000
n^2
$n\log(n)$


1×10^6
800 000
600 000
400 000
200 000
200
400
600
800
1000
n^2
$n \log(n)$

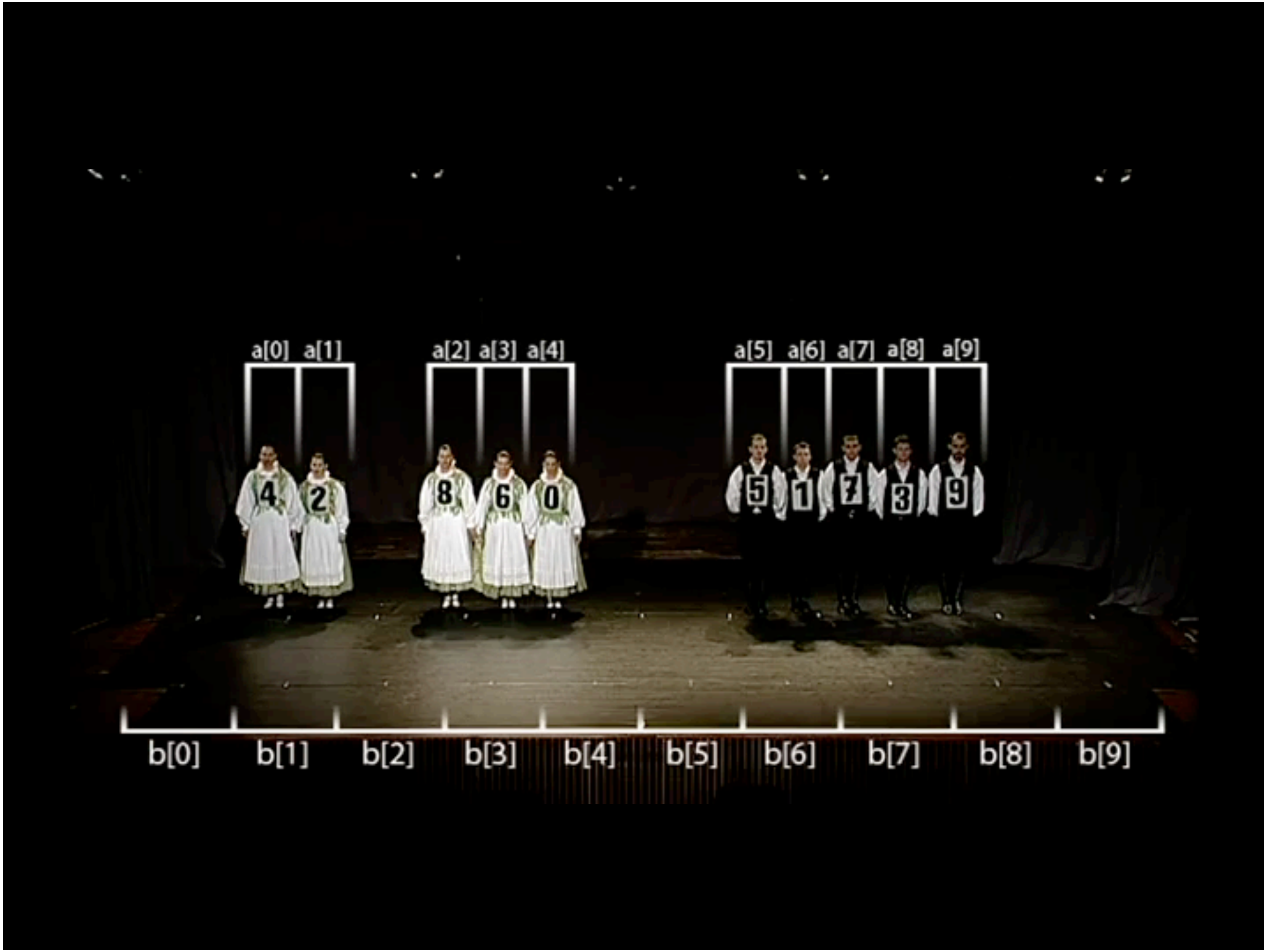

10 000
8000
6000
4000
2000
20
40
60
80
100
n^2
$n \log(n)$



bubble sort

merge sort

http://sortvis.org/


a[0] a[1]
a[2] a[3] a[4]
a[5] a[6] a[7] a[8] a[9]
4 2
8 6 0
5 1 7 3 9
b[0] b[1] b[2] b[3] b[4] b[5] b[6] b[7] b[8] b[9]

Sorting Websites

The animations I've shown are available at:

http://www.sorting-algorithms.com/

Dances for several algorithms are available at:

http://www.youtube.com/user/AlgoRythmics

Weave visualizations are available at:

http://sortvis.org/