ETH zürich

Departement Informatik

Programming and Problem-Solving

Sorting 2

Dennis Komm

Spring 2021 – April 15, 2021

Stacks and Queues

Stacks and Queues

So far access to arbitrary elements in lists by brackets

Stacks and Queues

So far access to arbitrary elements in lists by brackets

Stack

- Last-In First-Out
- Elements can be inserted at the end
- Elements can be extracted from the same end

Stacks and Queues

So far access to arbitrary elements in lists by brackets

Stack

- Last-In First-Out
- Elements can be inserted at the end
- Elements can be extracted from the same end

Queue

- First-In First-Out
- Elements can be inserted at the end
- Elements can be extracted from the front

Queues

Queue – Two Operations

- `append(x)` inserts element `x` at last position
- `pop(0)` removes first element and returns it
- In Python, lists can be used like queues

Queues

Queue – Two Operations

- `append(x)` inserts element `x` at last position
- `pop(0)` removes first element and returns it
- In Python, lists can be used like queues

```
data = [1, 4, 5]
data.append(8)
data.pop(0)
data.pop(0)
```

Queues

Queue – Two Operations

- `append(x)` inserts element `x` at last position
- `pop(0)` removes first element and returns it
- In Python, lists can be used like queues

```
data = [1, 4, 5]
data.append(8)     <------------   data = [1, 4, 5, 8]
data.pop(0)
data.pop(0)
```

Queues

Queue – Two Operations

- `append(x)` inserts element `x` at last position
- `pop(0)` removes first element and returns it
- In Python, lists can be used like queues

```
data = [1, 4, 5]
data.append(8)   ←—— data = [1, 4, 5, 8]
data.pop(0)
data.pop(0)      ←—— data = [5, 8]
```

Stacks

Stack – Two Operations

- `append(x)` inserts element `x` at last position
- `pop()` removes last element and returns it
- In Python, lists can also be used like stacks

Stacks

Stack – Two Operations

- `append(x)` inserts element `x` at last position
- `pop()` removes last element and returns it
- In Python, lists can also be used like stacks

```
data = [1, 4, 5]
data.append(8)
data.pop()
data.pop()
```

Stacks

Stack – Two Operations

- `append(x)` inserts element `x` at last position
- `pop()` removes last element and returns it
- In Python, lists can also be used like stacks

```
data = [1, 4, 5]
data.append(8)    <---  data = [1, 4, 5, 8]
data.pop()
data.pop()
```

Stacks

Stack – Two Operations

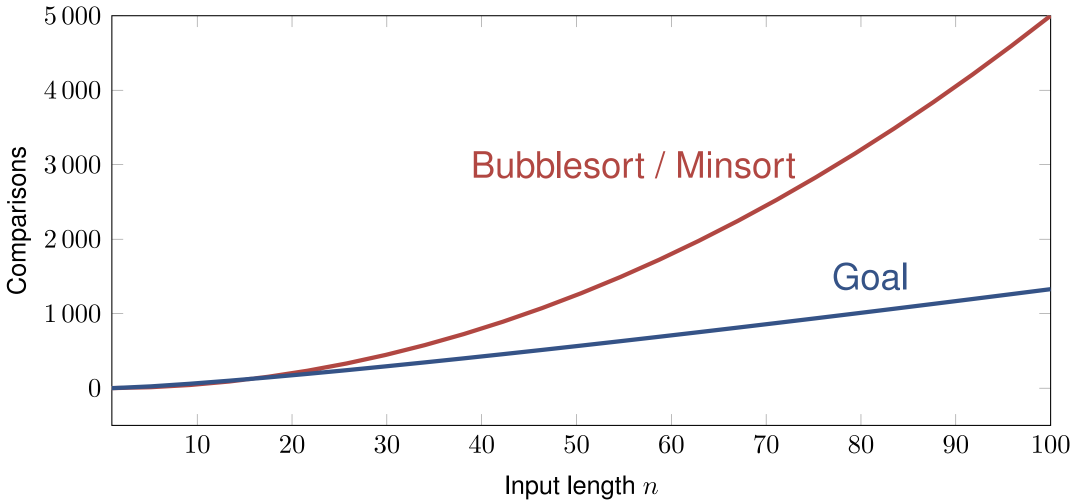
- `append(x)` inserts element `x` at last position
- `pop()` removes last element and returns it
- In Python, lists can also be used like stacks

```
data = [1, 4, 5]
data.append(8)   ←  data = [1, 4, 5, 8]
data.pop()
data.pop()       ←  data = [1, 4]
```

Sorting 2

Mergesort

Time Complexity of Bubblesort



How Fast Can We Sort?

Idea

Merging two sorted list is simple

How Fast Can We Sort?

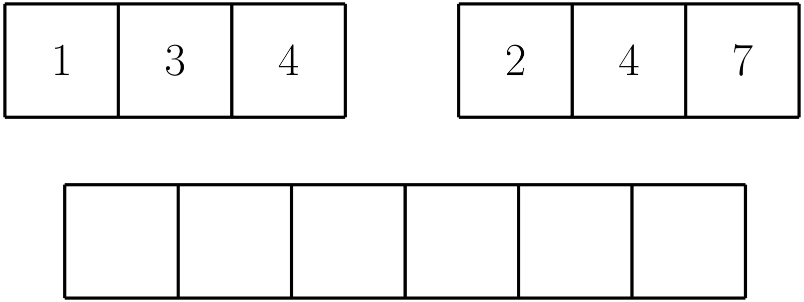
Idea

Merging two sorted list is simple

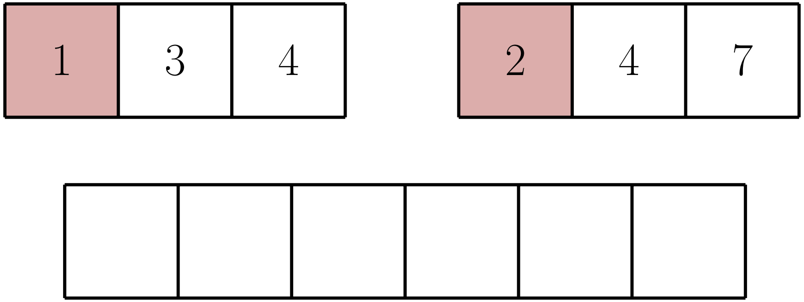
- First sort small lists
- Merge them
- Repeat

⇨ **Divide and Conquer**

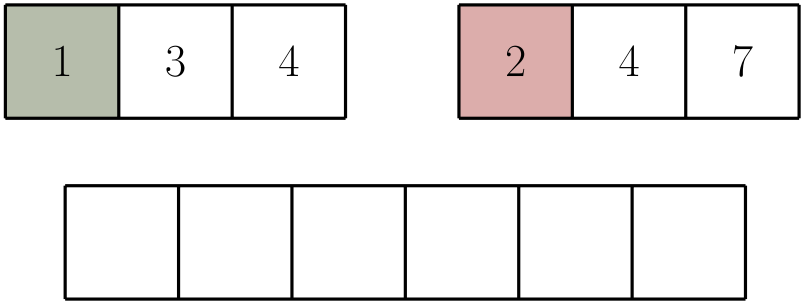
Merging of Sorted Lists



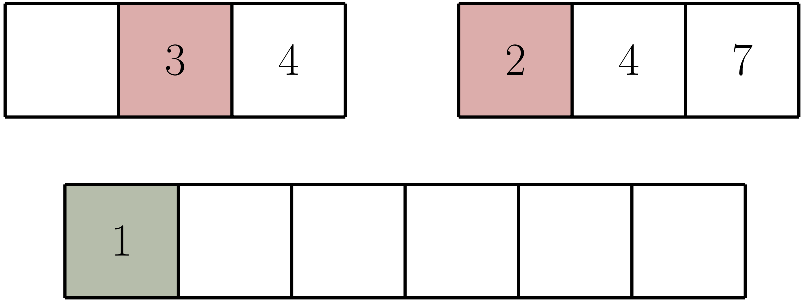
Merging of Sorted Lists



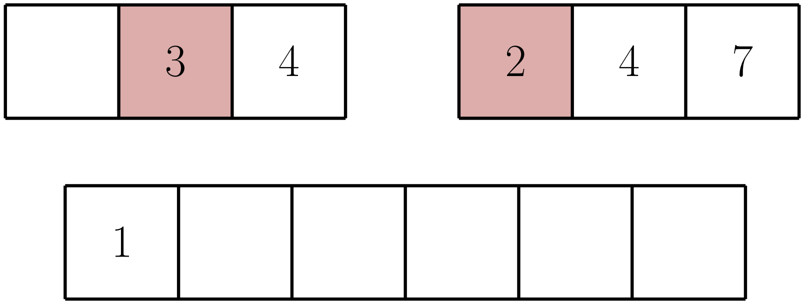
Merging of Sorted Lists



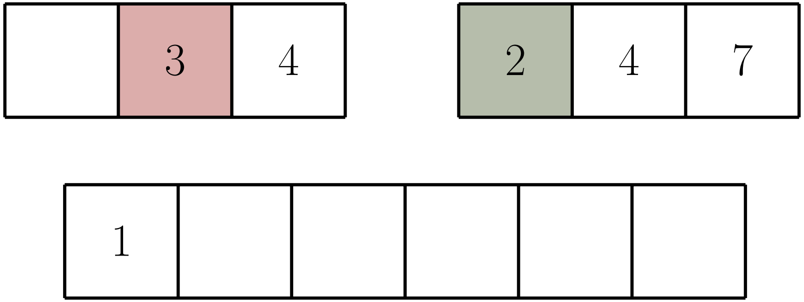
Merging of Sorted Lists



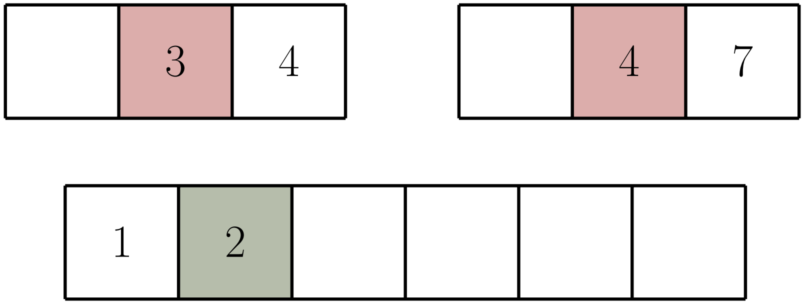
Merging of Sorted Lists



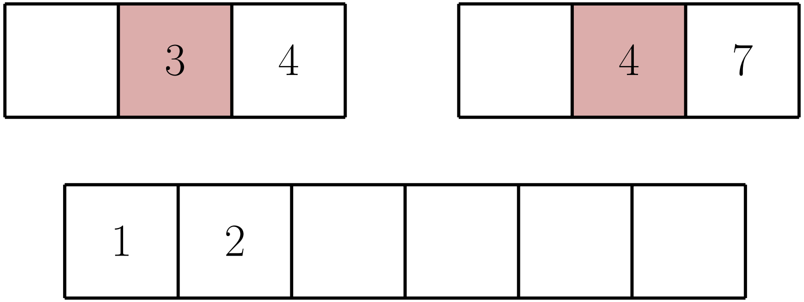
Merging of Sorted Lists



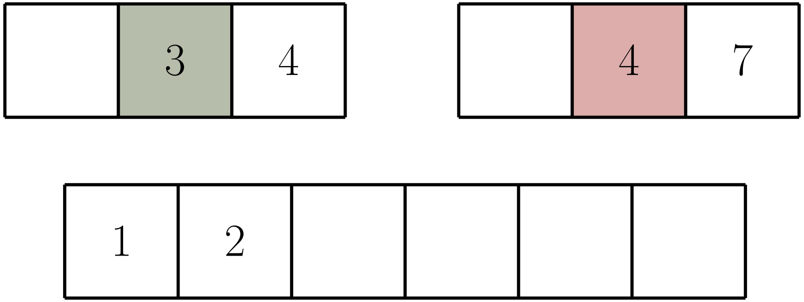
Merging of Sorted Lists



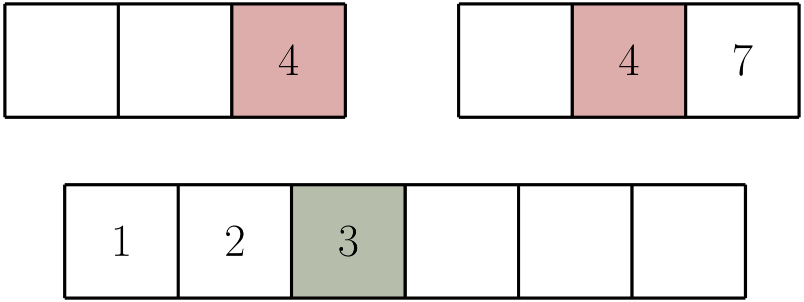
Merging of Sorted Lists

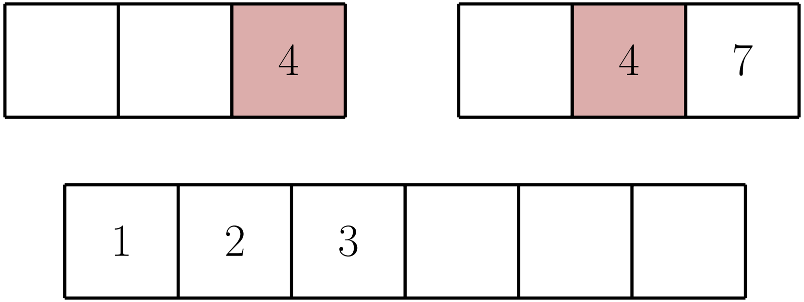


Merging of Sorted Lists

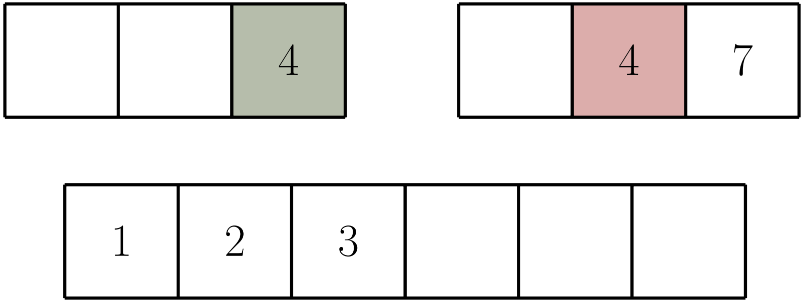


Merging of Sorted Lists

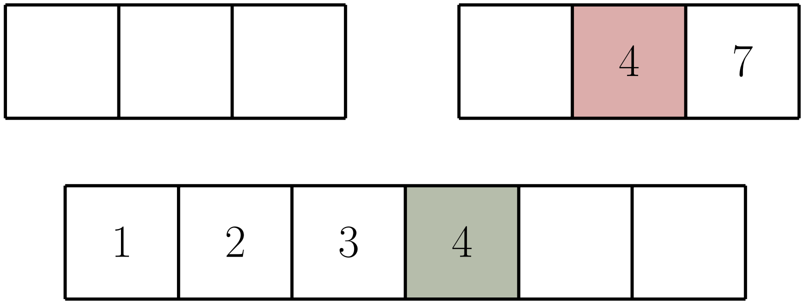



4
4
7
1
2
3

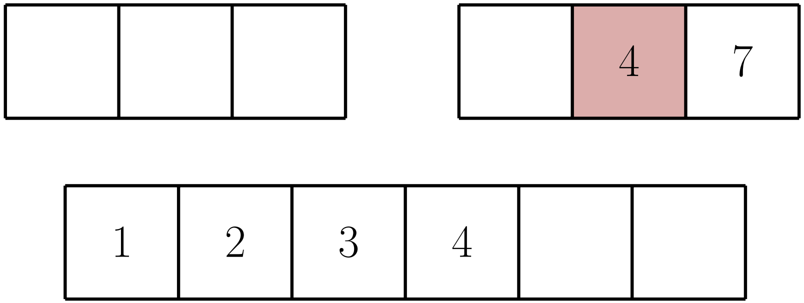
Merging of Sorted Lists



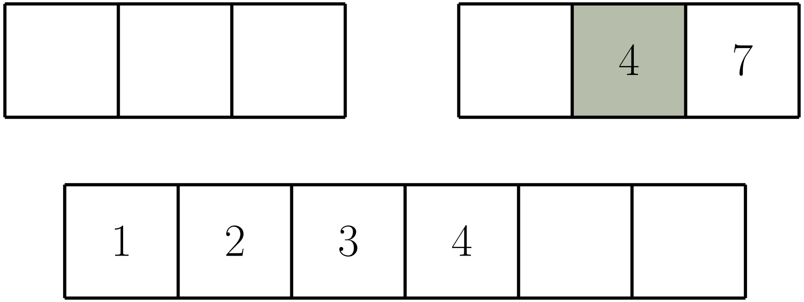
Merging of Sorted Lists



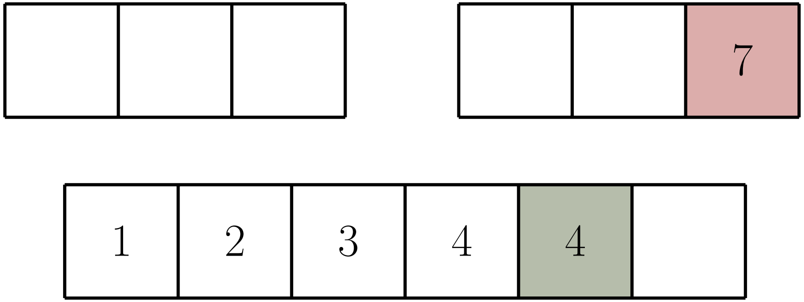
Merging of Sorted Lists



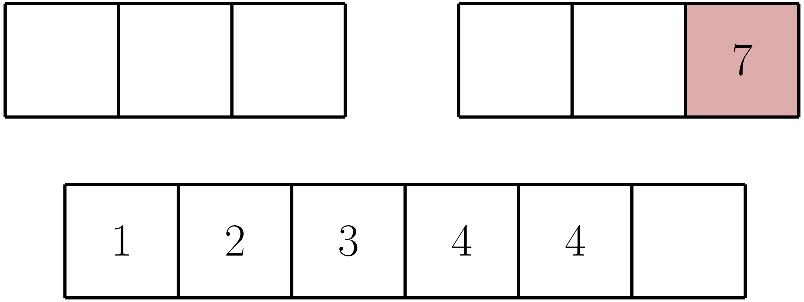
Merging of Sorted Lists



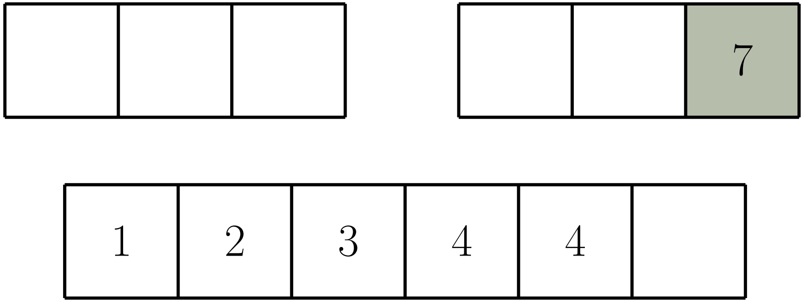
Merging of Sorted Lists



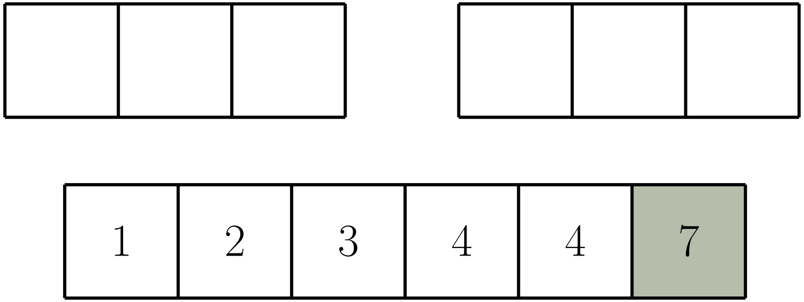
Merging of Sorted Lists



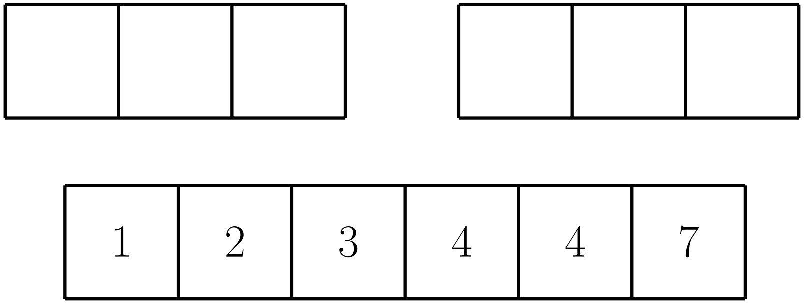
Merging of Sorted Lists



Merging of Sorted Lists



Merging of Sorted Lists



Exercise – Merging of Sorted Lists

Design a function that

- gets two sorted lists
- returns sorted list

Use the functions `pop(0)` and `append()`

Merging of Sorted Lists

```
def merge(left, right):
    result = []
    while len(left) > 0 and len(right) > 0:
        if left[0] > right[0]:
            result.append(right.pop(0))
        else:
            result.append(left.pop(0))
    return result + left + right
```

Merging of Sorted Lists

```
def merge(left, right):
    result = []
    while len(left) > 0 and len(right) > 0:
        if left[0] > right[0]:
            result.append(right.pop(0))
        else:
            result.append(left.pop(0))
    return result + left + right
```

While not both lists are empty

Merging of Sorted Lists

```
def merge(left, right):
    result = []
    while len(left) > 0 and len(right) > 0:
        if left[0] > right[0]:
            result.append(right.pop(0))
        else:
            result.append(left.pop(0))
    return result + left + right
```

Append the smaller of both elements

Merging of Sorted Lists

```
def merge(left, right):
    result = []
    while len(left) > 0 and len(right) > 0:
        if left[0] > right[0]:
            result.append(right.pop(0))
        else:
            result.append(left.pop(0))
    return result + left + right
```

One of the two given sorted lists may still contain elements

Mergesort

Divide and Conquer

Iteratively merge sorted lists

Mergesort

Divide and Conquer

Iteratively merge sorted lists

- First merge “lists” of length 1 to lists of length 2

Mergesort

Divide and Conquer

Iteratively merge sorted lists

- First merge “lists” of length 1 to lists of length 2
- Merge lists of length 2 to lists of length 4

Mergesort

Divide and Conquer

Iteratively merge sorted lists

- First merge “lists” of length 1 to lists of length 2
- Merge lists of length 2 to lists of length 4
- Merge lists of length 4 to lists of length 8

Mergesort

Divide and Conquer

Iteratively merge sorted lists

- First merge “lists” of length 1 to lists of length 2
- Merge lists of length 2 to lists of length 4
- Merge lists of length 4 to lists of length 8
- Merge lists of length 8 to lists of length 16

Mergesort

Divide and Conquer

Iteratively merge sorted lists

- First merge “lists” of length 1 to lists of length 2
- Merge lists of length 2 to lists of length 4
- Merge lists of length 4 to lists of length 8
- Merge lists of length 8 to lists of length 16
- $\ldots$

Mergesort

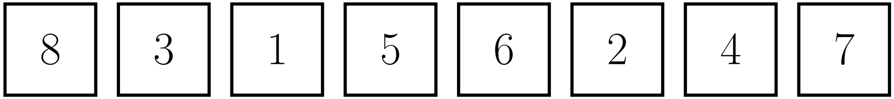
8	3	1	5	6	2	4	7

Mergesort

8	3	1	5	6	2	4	7

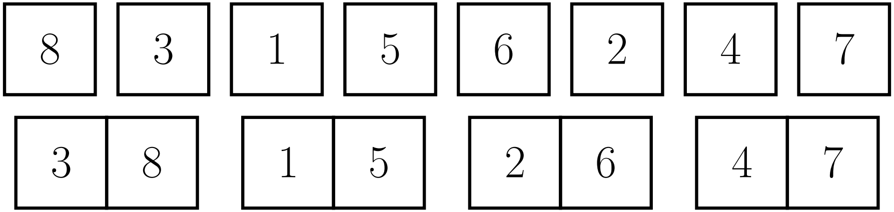
```
[8, 3, 1, 5, 6, 2, 4, 7]
```

Mergesort



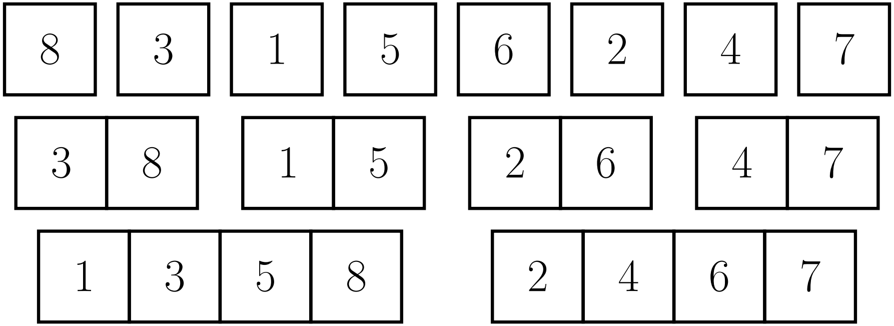
```
[[8], [3], [1], [5], [6], [2], [4], [7]]
```

Mergesort



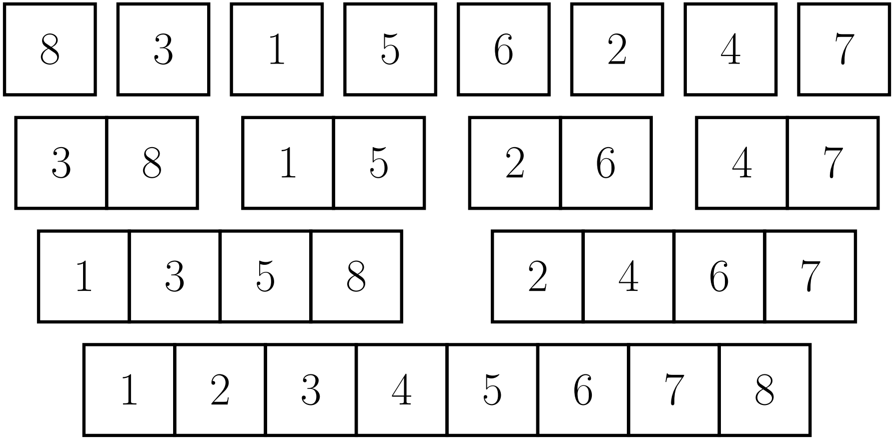
```
[[3, 8], [1, 5], [2, 6], [4, 7]]
```

Mergesort



```
[[1, 3, 5, 8], [2, 4, 6, 7]]
```

Mergesort



```
[[1, 2, 3, 4, 5, 6, 7, 8]]
```

Merge Step

Single Merge Step

- Get a 2-dimensional list, i.e., list that contains lists
- Each two successive lists are merged using the function `merge()`
- The last list is simply appended if there is an odd number of lists
- The result is again a 2-dimensional list that contains the merged lists

Merge Step

```python
def mergestep(data):
    result = []
    while len(data) > 1:
        left = data.pop(0)
        right = data.pop(0)
        result.append(merge(left, right))
    return result + data
```

Merge Step

```
def mergestep(data):
    result = []
    while len(data) > 1:
        left = data.pop(0)
        right = data.pop(0)
        result.append(merge(left, right))
    return result + data
```

While there are still
at least two lists

Merge Step

```
def mergestep(data):
    result = []
    while len(data) > 1:
        left = data.pop(0)
        right = data.pop(0)
        result.append(merge(left, right))
    return result + data
```

Merge the first two lists

Merge Step

```
def mergestep(data):
    result = []
    while len(data) > 1:
        left = data.pop(0)
        right = data.pop(0)
        result.append(merge(left, right))
    return result + data
```

If there is a list left at the end, append it

Mergesort – Complete Algorithm

Complete Algorithm

- Input is given as list `data`
- Convert every element into a list with one element
- This way get 2-dimensional list
- Apply `mergestep()` repeatedly to this list
- At the end, there will only be one element in the list
- This element corresponds to a sorted list

Mergesort – Complete Algorithm

```python
def mergesort(data):
    result = []
    for item in data:
        result.append([item])
    while len(result) > 1:
        result = mergestep(result)
    return result[0]
```

Sorting 2

Time Complexity of Mergesort

Time Complexity of Mergesort

Time complexity of Mergesort is proportional to

Number of merge steps $\times$ Comparisons per merge step

Time Complexity of Mergesort

Time complexity of Mergesort is proportional to
Number of merge steps $\times$ Comparisons per merge step

- Length of sorted lists doubles with each merge step

Time Complexity of Mergesort

Time complexity of Mergesort is proportional to
Number of merge steps $\times$ Comparisons per merge step

- Length of sorted lists doubles with each merge step

⇨ Roughly $\log_2 n$ merge steps for n elements

Time Complexity of Mergesort

Time complexity of Mergesort is proportional to
Number of merge steps $\times$ Comparisons per merge step

- Length of sorted lists doubles with each merge step

⇨ Roughly $\log_2 n$ merge steps for n elements

- In a merge step, one element is written into `result` with every comparison

Time Complexity of Mergesort

Time complexity of Mergesort is proportional to
Number of merge steps $\times$ Comparisons per merge step

- Length of sorted lists doubles with each merge step

⇨ Roughly $\log_2 n$ merge steps for n elements

- In a merge step, one element is written into `result` with every comparison

⇨ At most n comparisons per merge step

Time Complexity of Mergesort

Time complexity of Mergesort is proportional to
Number of merge steps $\times$ Comparisons per merge step

- Length of sorted lists doubles with each merge step

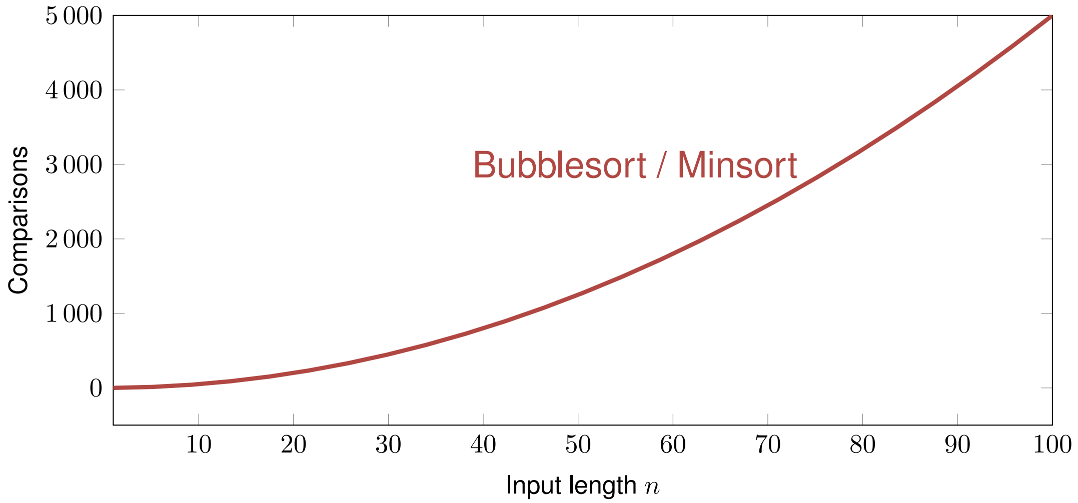
⇨ Roughly $\log_2 n$ merge steps for n elements

- In a merge step, one element is written into `result` with every comparison

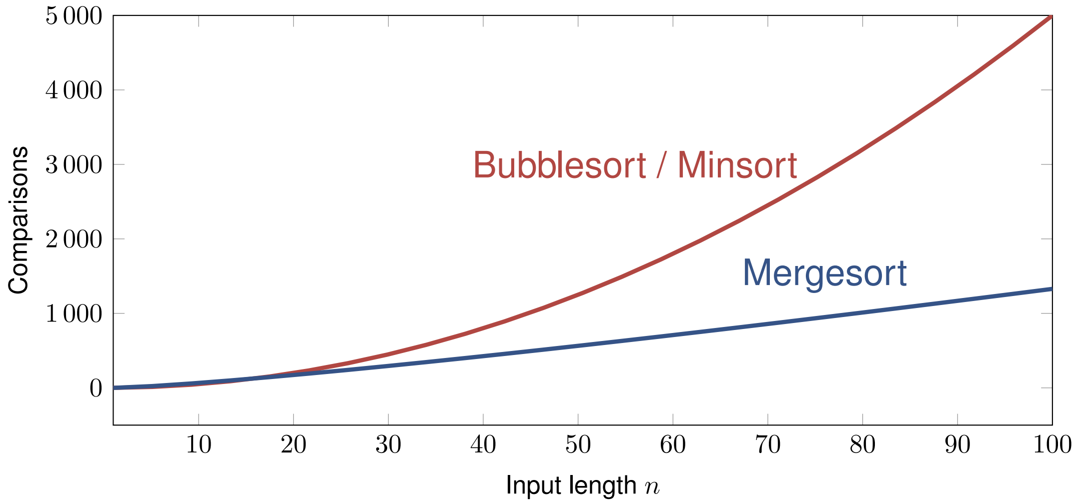
⇨ At most n comparisons per merge step

Time complexity of Mergesort is in $\mathcal{O}(n \log_2 n)$

Time Complexity of Mergesort



Time Complexity of Mergesort



Sorting 2

Complexity of Sorting

Complexity of Sorting

How does the running time change for specific inputs?

- Already sorted
- Sorted in reverse
- Randomly chosen

Complexity of Sorting

How does the running time change for specific inputs?

- Already sorted
- Sorted in reverse
- Randomly chosen

For Mergesort (and also Bubble- and Minsort),
the number of comparisons is always the same for a fixed n

Complexity of Sorting

How does the running time change for specific inputs?

- Already sorted
- Sorted in reverse
- Randomly chosen

For Mergesort (and also Bubble- and Minsort),
the number of comparisons is always the same for a fixed n

- This is not always the case

Complexity of Sorting

How does the running time change for specific inputs?

- Already sorted
- Sorted in reverse
- Randomly chosen

For Mergesort (and also Bubble- and Minsort),
the number of comparisons is always the same for a fixed n

- This is not always the case
- Different best, average, and worst cases

Complexity of Sorting

How does the running time change for specific inputs?

- Already sorted
- Sorted in reverse
- Randomly chosen

For Mergesort (and also Bubble- and Minsort),
the number of comparisons is always the same for a fixed n

- This is not always the case
- Different best, average, and worst cases
- **Timsort**, for instance, makes use of already sorted sub lists

Sorting 2

Bucketsort

Sorting of Few Elements

Sorting of data sets with respect to **one attribute**

Sorting of Few Elements

Sorting of data sets with respect to **one attribute**

Stable sorting: Elements with same attribute maintain order

Sorting of Few Elements

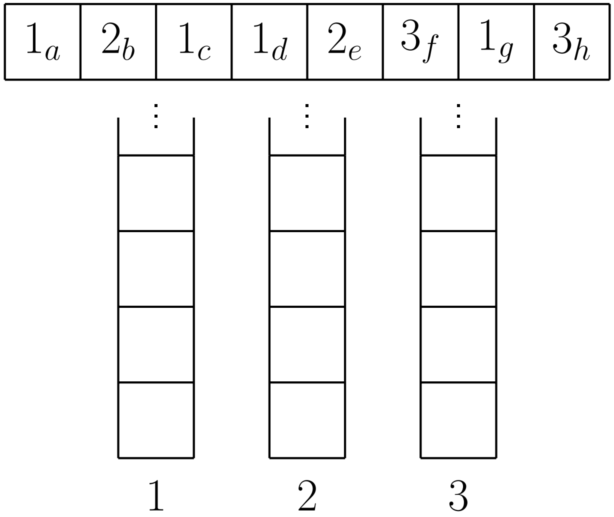
Sorting of data sets with respect to **one attribute**

Stable sorting: Elements with same attribute maintain order

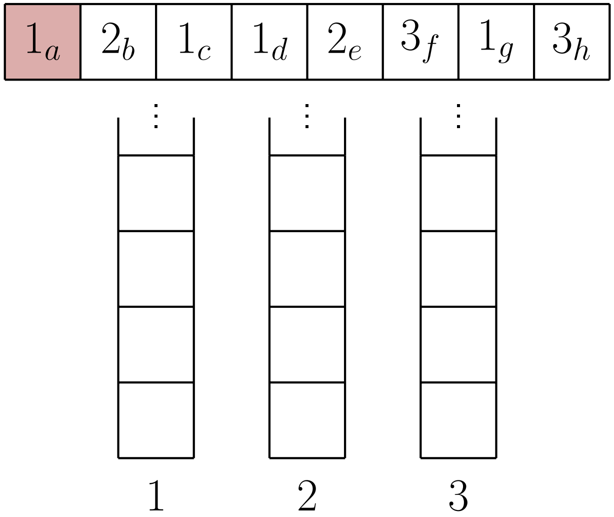
Example

Name	First name	Grade
Adleman	Leonard	6
Caesar	Gaius Julius	3
de Vigenère	Blaise	5
Rivest	Ronald	6
Shamir	Adi	6

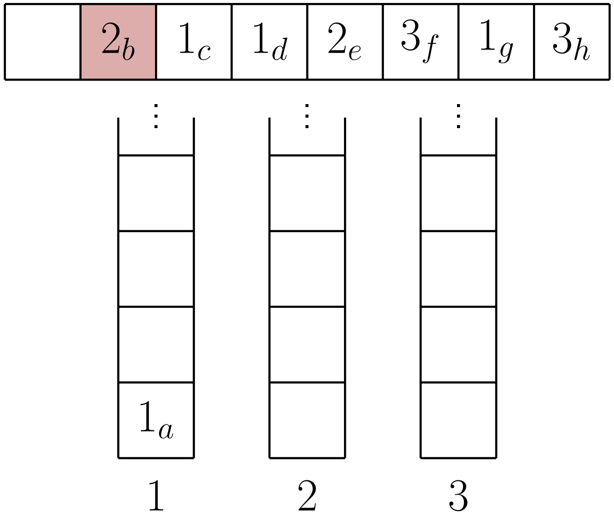
Bucketsort



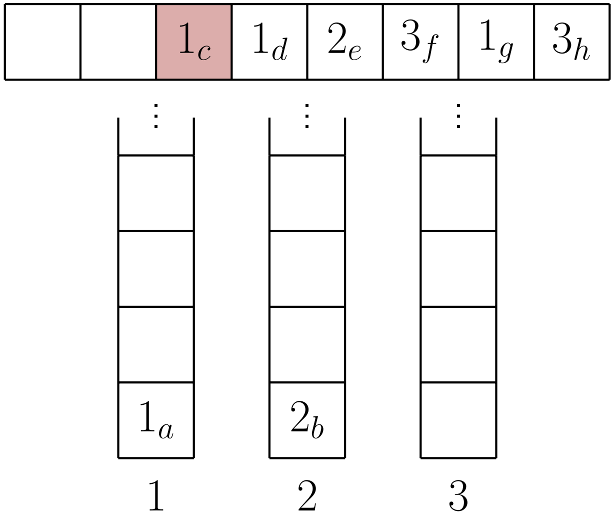
Bucketsort



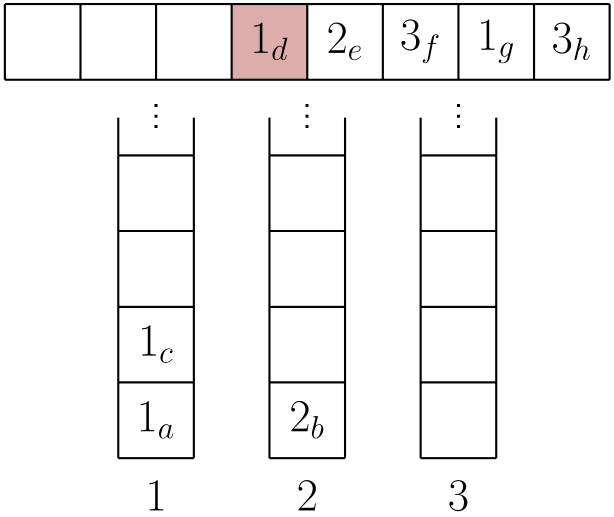
Bucketsort



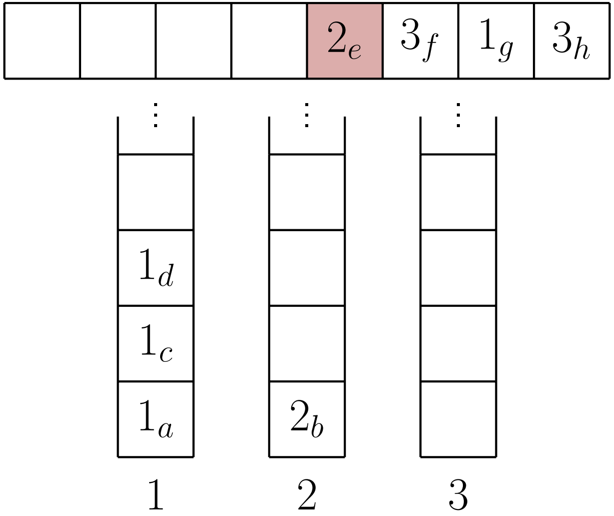
Bucketsort



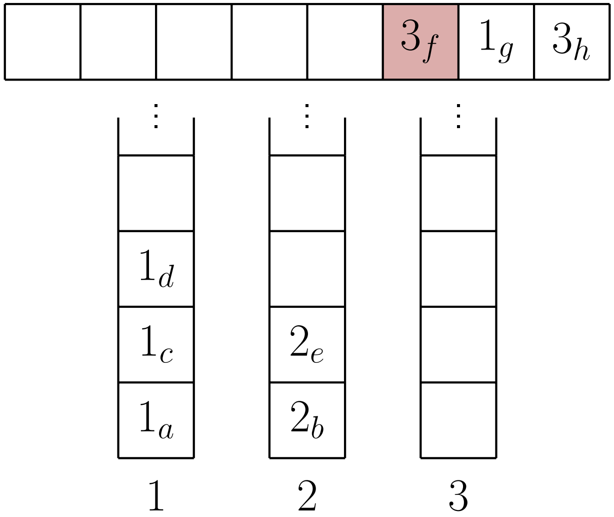
Bucketsort



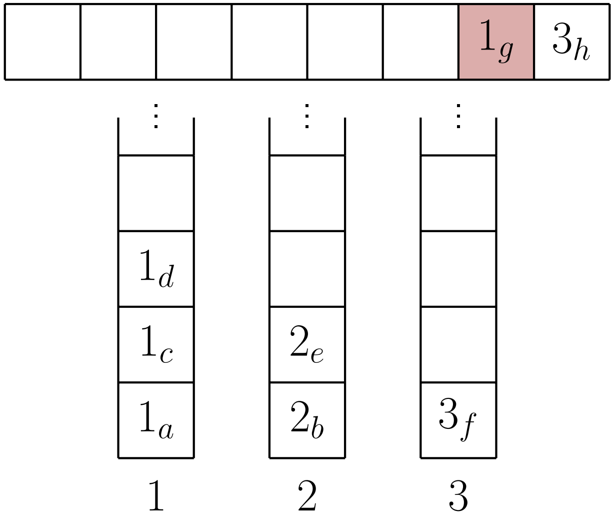
Bucketsort



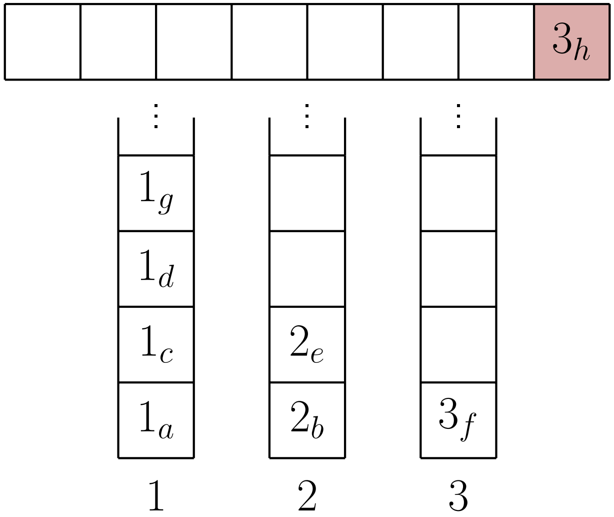
Bucketsort



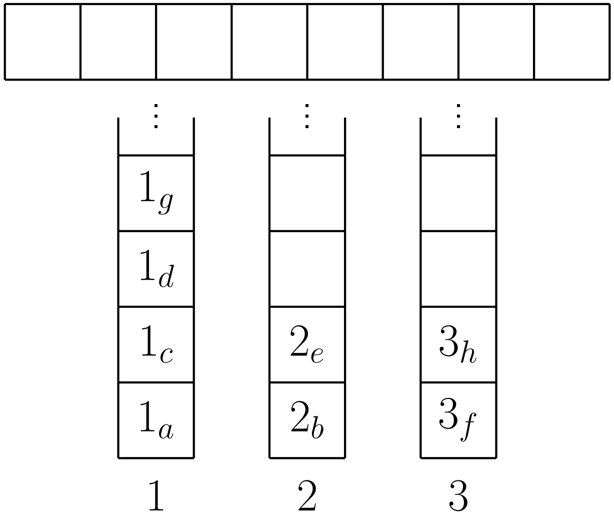
Bucketsort



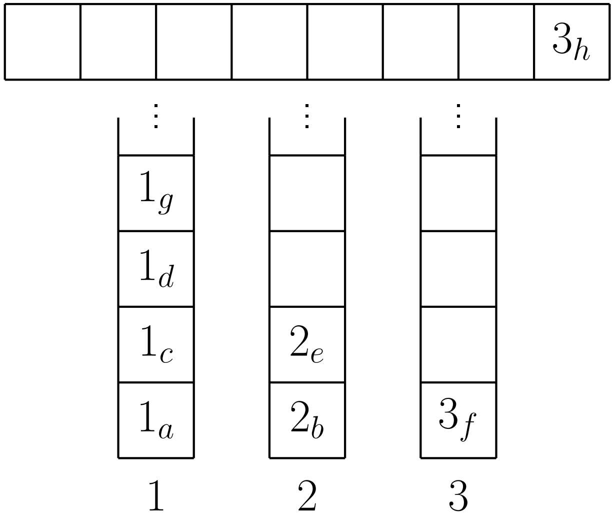
Bucketsort



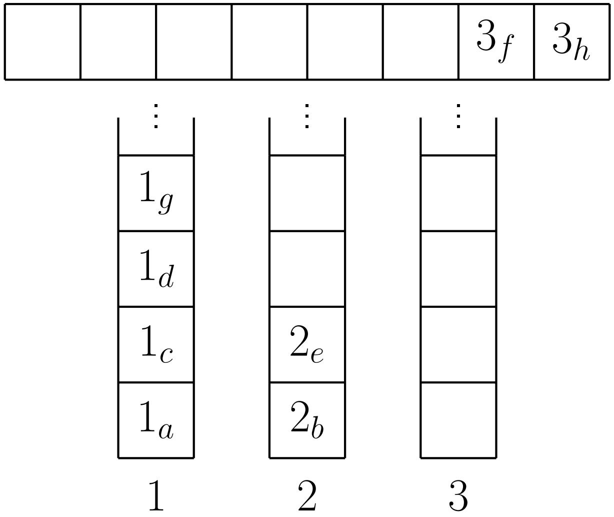
Bucketsort



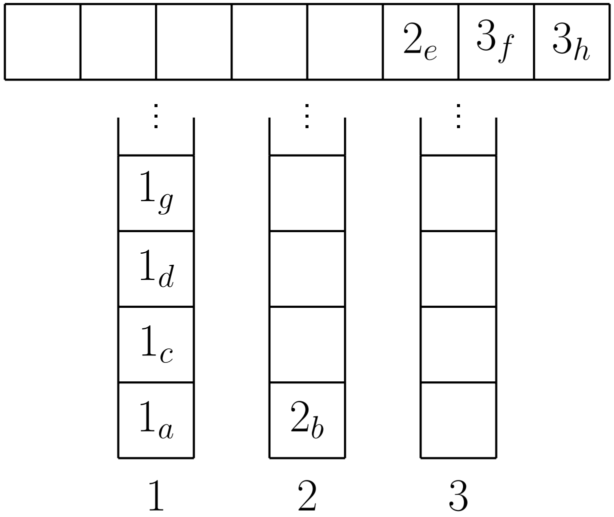
Bucketsort



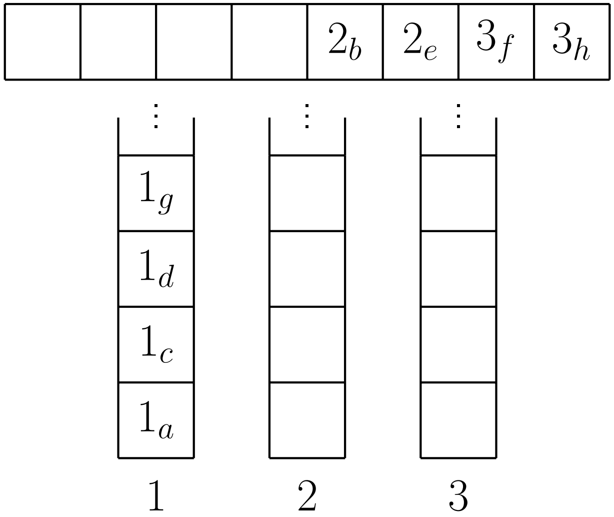
Bucketsort



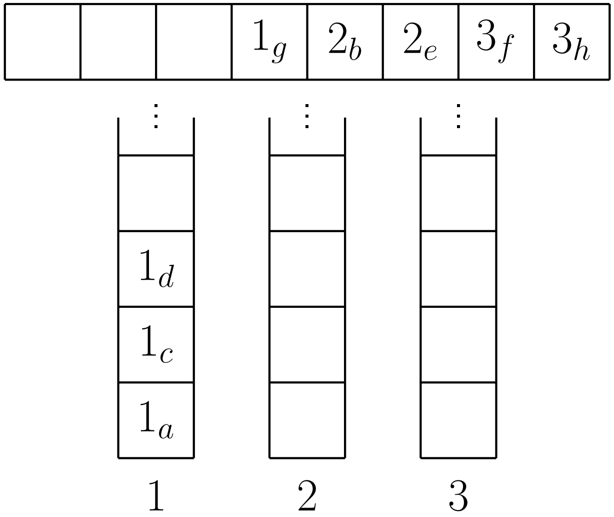
Bucketsort



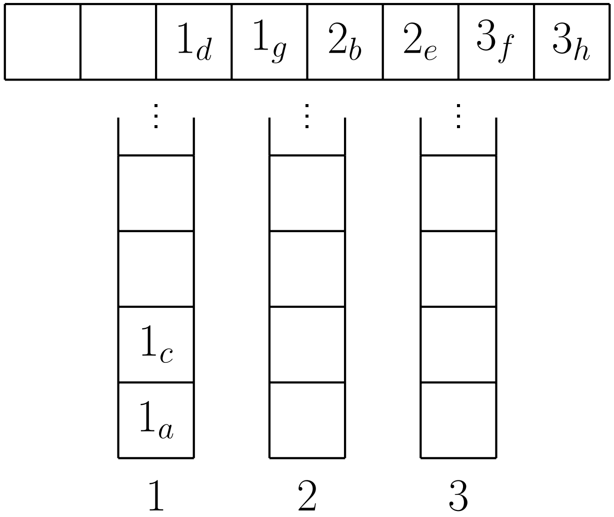
Bucketsort



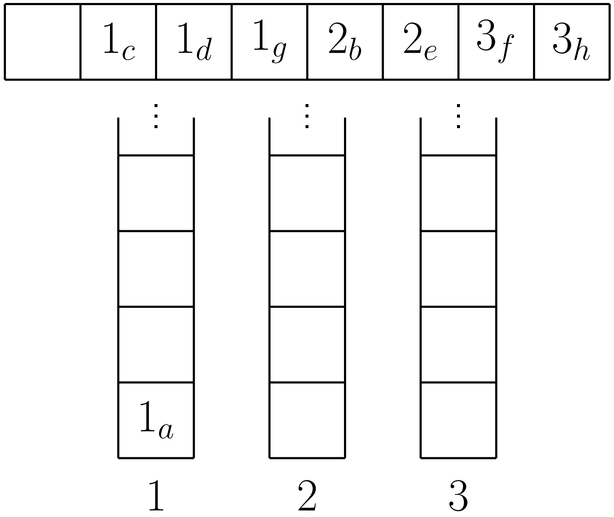
Bucketsort



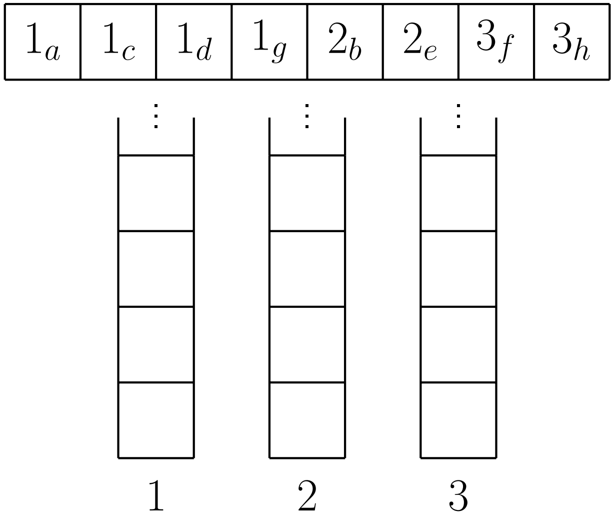
Bucketsort



Bucketsort



Bucketsort



Exercise – Bucketsort

Implement Bucketsort

- as Python function
- using three **stacks** `one`, `two`, and `three` for the possible values 1, 2, and 3
- filling the stacks according to numbers in the list
- concatenating the stacks at the end (this is quite simple in Python using the + operator)

Bucketsort

```python
def bucketsort(data):
    one = []
    two = []
    three = []
    for item in data:
        if item == 1:
            one.append(item)
        else:
            if item == 2:
                two.append(item)
            else:
                if item == 3:
                    three.append(item)
    return one + two + three
```

Bucketsort

```
def bucketsort(data):
    one = []
    two = []
    three = []
    for item in data:
        if item == 1:
            one.append(item)
        else:
            if item == 2:
                two.append(item)
            else:
                if item == 3:
                    three.append(item)
    return one + two + three
```

```
if item == 1:
    one.append(item)
elif item == 2:
    two.append(item)
elif item == 3:
    three.append(item)
```

Sorting 2

Time Complexity of Bucketsort

Time Complexity of Bucketsort

- Let n denote the input length
- Let k denote the number of distinct values

Time Complexity of Bucketsort

- Let n denote the input length
- Let k denote the number of distinct values
- When filling the buckets, at most $k - 1$ comparisons per element

Time Complexity of Bucketsort

- Let n denote the input length
- Let k denote the number of distinct values
- When filling the buckets, at most $k - 1$ comparisons per element

⇨ Total number of comparisons: roughly $k \cdot n$

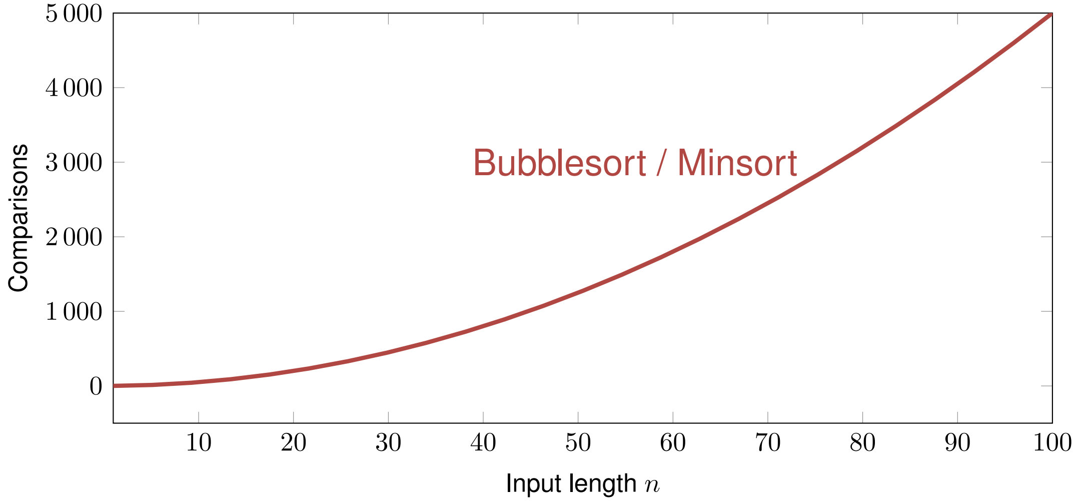
Time Complexity of Bucketsort

- Let n denote the input length
- Let k denote the number of distinct values
- When filling the buckets, at most $k - 1$ comparisons per element

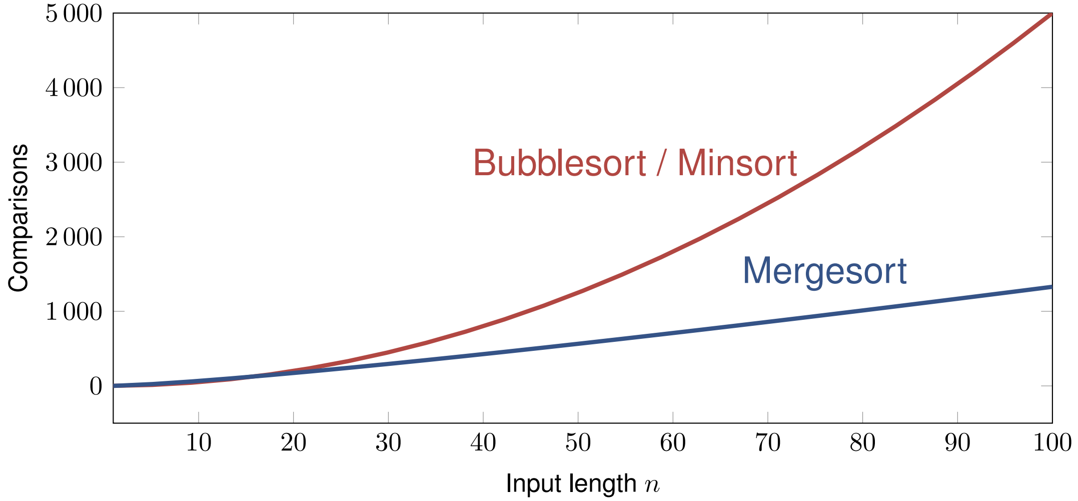
⇨ Total number of comparisons: roughly $k \cdot n$

The time complexity of Bucketsort is in $\mathcal{O}(n)$ if there is a constant number of different values

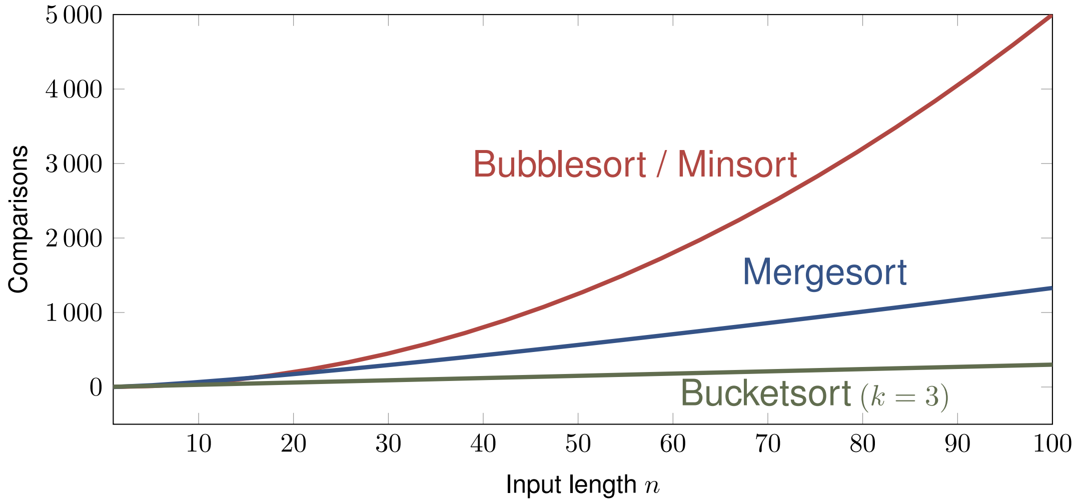
Time Complexity of Bucketsort



Time Complexity of Bucketsort



Time Complexity of Bucketsort



Thanks for your attention