

# Dynamic Data Structures and Algorithms.

## ENGF0002: Design and Professional Skills

Prof. Mark Handley, University College London, UK  
Partly based on slides from George Danezis

Term 1, 2018

# Sorting.

Binary search, and a number of other efficient algorithms, rely on sorted sequences.

We next study:

- A simple sorting algorithm.
- A better sorting algorithm, **merge sort**.
- The **time complexity** of sorting.
- How to show sorting is **correct**.
- How to write **generic** sorting and searching algorithms.

96

92

109

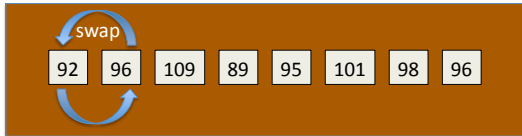
89

95

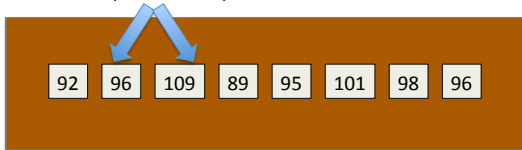
101

98

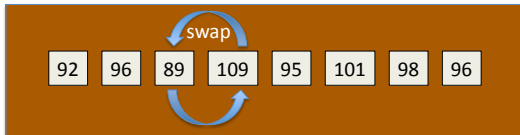
96



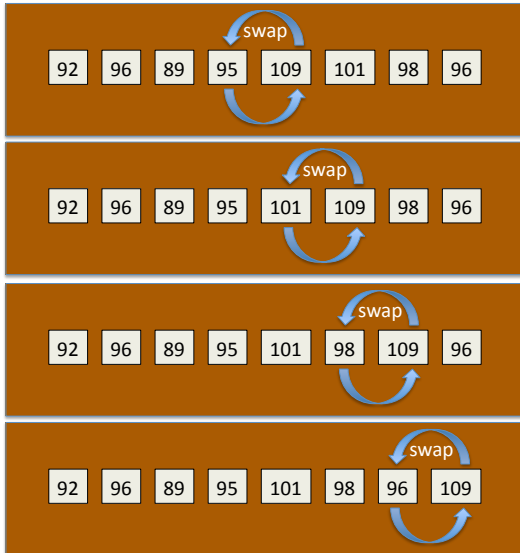
compare, no swap



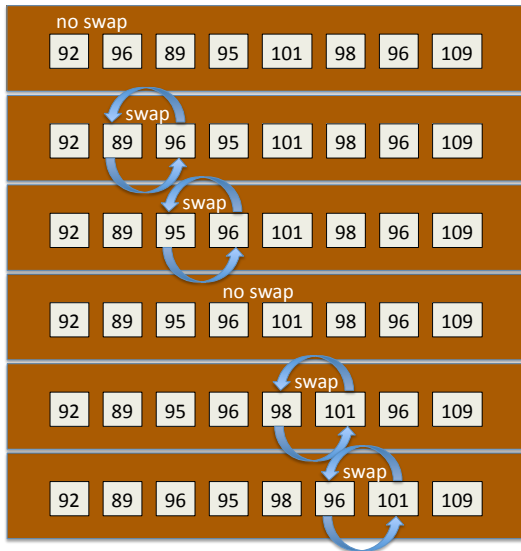
# Bubble Sort



# Bubble Sort

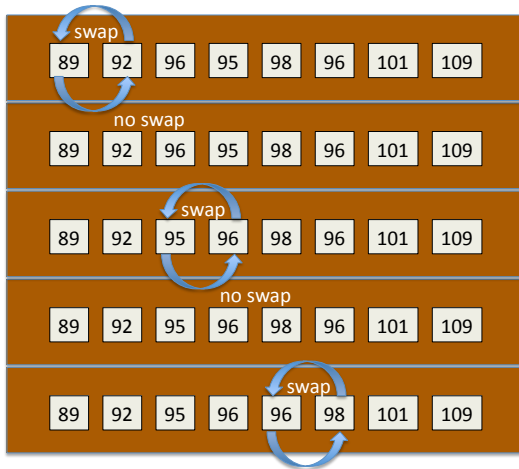


# Bubble Sort

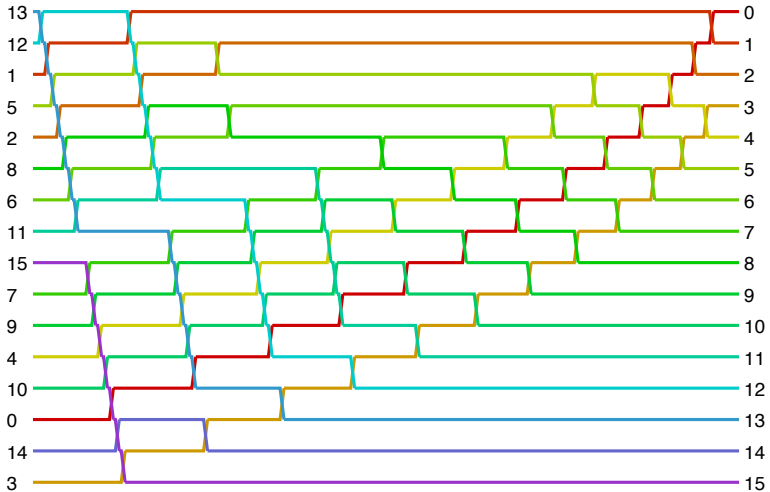




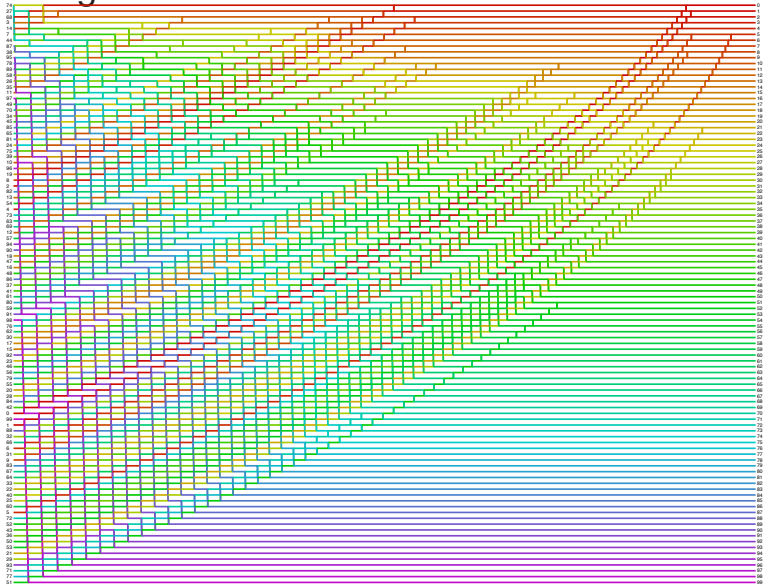
# Bubble Sort



# Visualizing Bubble Sort



# Visualizing Bubble Sort



## Complexity of Bubble Sort

What is the computational complexity of Bubblesort with  $n$  items?

- Each pass through the data involves  $n - 1$  comparisons.
- Each pass moves one more item to the end of the list.
  - After 1 pass, the largest item is at the end.
  - After 2 passes, the largest two items are at the end and sorted.
  - After  $k$  passes, the largest  $k$  items are at the end and sorted.
- After  $n$  passes, the whole list is sorted.

Requires  $n(n - 1)$  comparisons.

$$n(n - 1) = n^2 - n = \mathcal{O}(n^2)$$

# Optimization of Bubblesort

Observe we don't need to compare all  $n$  items in each pass.

- In pass  $k$ , the last  $k - 1$  items are already sorted, so we can stop the pass early.
- Cuts the total number of comparisons in half.

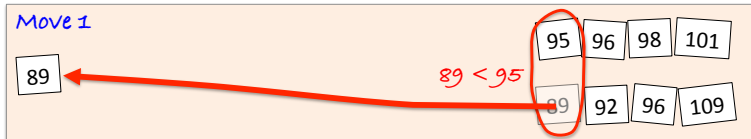
Does this change the complexity?

# Merging lists

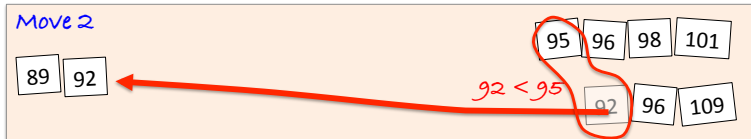
sorted list 1 95 96 98 101

sorted list 2 89 92 96 109

# Merging lists

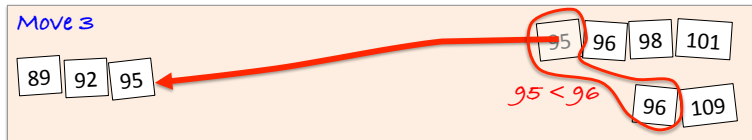


## Merging lists

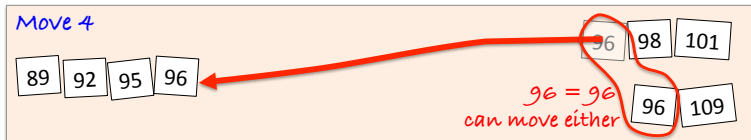




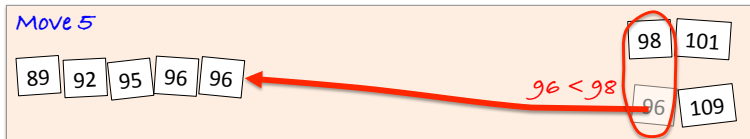
## Merging lists



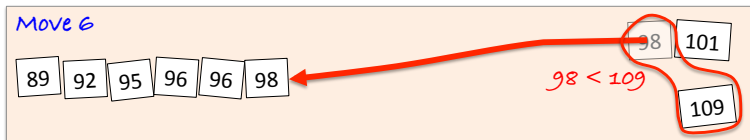
# Merging lists



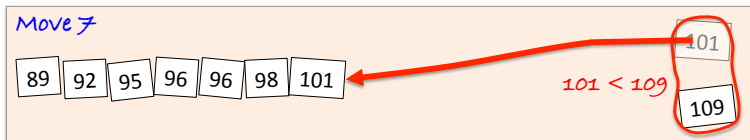
## Merging lists



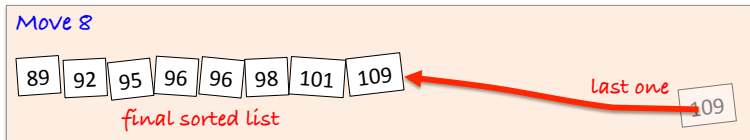
# Merging lists



# Merging lists



## Merging lists



### Observation

Merging two **already sorted** lists containing  $n$  items only requires  $n - 1$  comparisons and  $n$  moves.

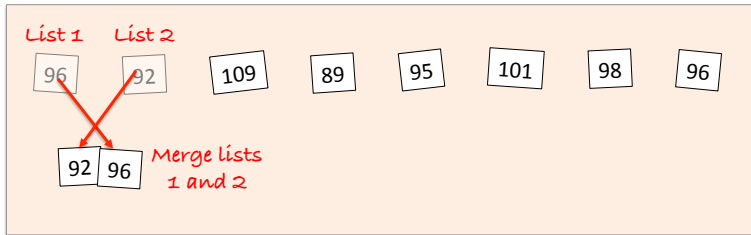
## Little lists



### Observation

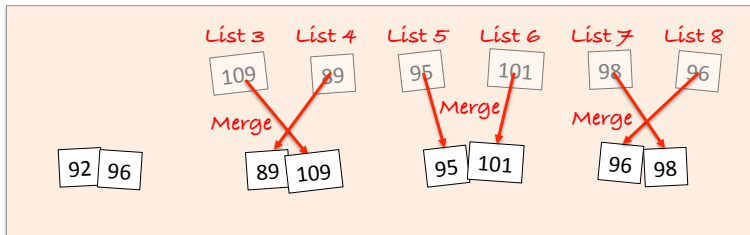
Lists of length 1 are **already sorted!**.

## Merging lists (round 1)

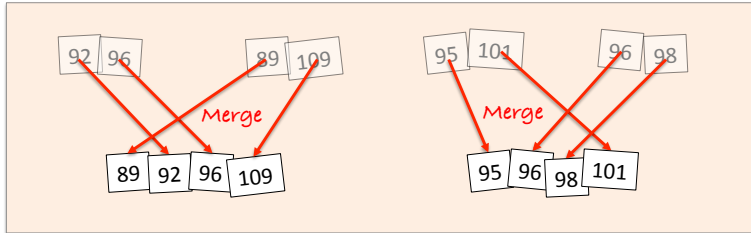




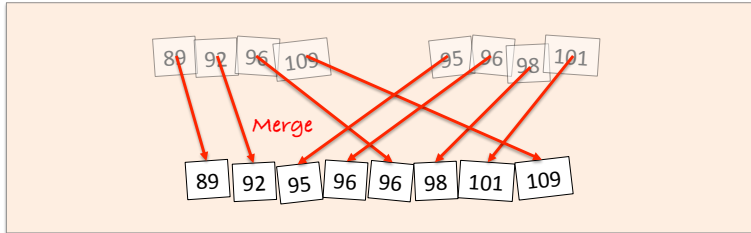
## Merging lists (round 1 continued)



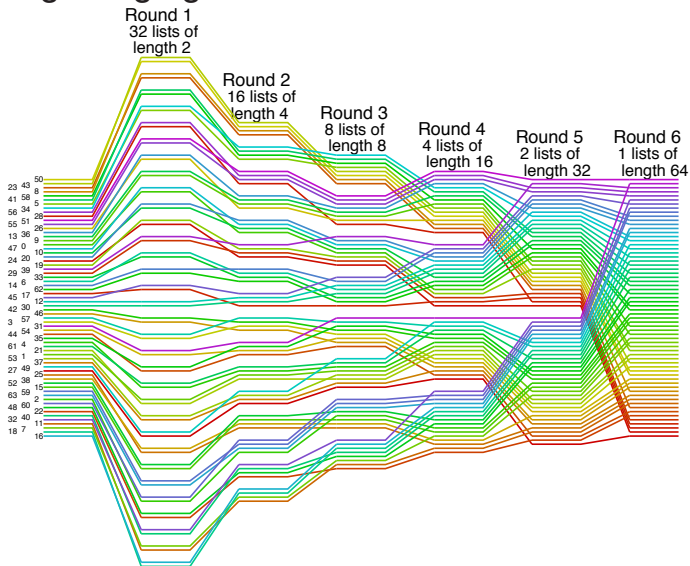
## Merging lists (round 2)



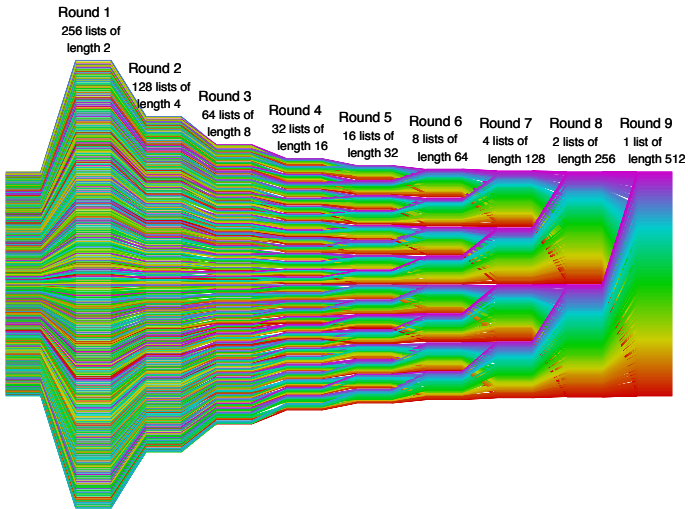
## Merging lists (round 3)



# Visualizing merging lists: 64 items



# Visualizing merging lists: 64 items



# Merge sort

Mergesort was proposed by John von Neumann in 1945.

Essentially two algorithms:

Algorithm 1. Merge two sorted lists into one sorted list:

- While both lists are not empty, consider the first unprocessed item from both.
- Add to the front of the new list the smaller or available one, and remove it from its list. Repeat.
- Append remaining elements of one of the lists.

# Merge sort

Algorithm 2. Merge Sort.

- If the list has length 1, we're done.
- Otherwise, divide into two sub-lists of equal length.
- Merge Sort both lists separately.
- Merge the two sorted lists using Algorithm 1.

## Tests for sorting.

```
def test_simple_sort():
    assert mergesort([2, 3, 1, 4, 0]) == [0, 1, 2, 3, 4]

def test_simple_sort10(): # Test edge cases, for 1 or no elements.
    assert mergesort([190]) == [190]
    assert mergesort([]) == []

def test_simple_longlist():
    import random
    n = 1000
    lst = list(range(n))
    random.shuffle(lst) # Shuffle at random in-place
    sorted_lst = mergesort(lst)
    assert sorted_lst == list(range(n))
```



## Merge two sorted lists (algorithm 1).

```
def mergelists(lst1, lst2):  
    ''' Merge two sorted lists into a sorted list.'''  
    merged_lst = []  
    len1 = len(lst1)  
    len2 = len(lst2)  
    index1 = 0  
    index2 = 0  
    # while either list has any items left  
    while index1 < len1 or index2 < len2:  
        if (index2 == len2      #either we've used all of lst2  
            or (index1 != len1   #or both lists have elements left  
                and lst1[index1] <= lst2[index2])): #and lst1 is smaller  
            merged_lst.append(lst1[index1])  
            index1 += 1  
        else:  
            merged_lst.append(lst2[index2])  
            index2 += 1  
    return merged_lst
```

## A recursive implementation of mergesort (algorithm 2).

```
def mergesort(lst):  
    if len(lst) <= 1:  
        return lst.copy()  
    pivot = len(lst) // 2  
    lst1 = mergesort(lst[:pivot])  
    lst2 = mergesort(lst[pivot:])  
    return mergelists(lst1, lst2)
```

- Python **slicing**: `lst[start:end]` returns the list from index `start` up to, but not including, index `end`. If omitted from zero to the length of the list.
- Use `lst.copy()` to get a new **copy** of the list (otherwise we'll just modify the original!).

# How to argue mergesort is correct.

Structural induction (verbal reasoning):

- Argue that the merge operation is correct: given two sorted lists it produces a sorted list.
- Base case of mergesort: a single element list is sorted.
- Inductive case of mergesort: assume mergesort for a previous step works, prove that the result is sorted. Follows from the properties of merge.

Note that the recursive structure of the program supports the argument of correctness through induction.

## What is the cost of mergesort.

The time complexity of merge (algorithm 1) is  $\mathcal{O}(n)$ , where  $n$  is the sum of the lengths of both lists.

Mergesort (algorithm 2) can only divide an array of  $n$  elements  $\mathcal{O}(\log n)$  times. At each level of subdivision the overall costs of the merge operations will be  $\mathcal{O}(n)$  (total from algorithm 1).

Therefore the overall time complexity of mergesort is of the **order**  $\mathcal{O}(n \log n)$ .

This is **optimal** for comparison based algorithms (but note that **constant factors may vary**.) Space complexity may also vary.

## More sequences and generic programming

# Strings and byte sequences.

Besides **lists**, Python supports a number of other sequence types:

- **tuples**: like a list, but content is immutable once assigned.
- **strings** (`str`): represent sequences of unicode characters.
- **bytes** (`bytes`): represent a sequences of raw bytes (ie. values from 0 to 255).
- Strings are transformed into and from sequences of bytes through an **encoding** such as utf8 or ASCII.

```
>>> name = "École"           # A unicode string
>>> name.encode('utf-8')     # A byte sequence
b'\xc3\x89cole'
```

## Internationalization (i18n).

Old encodings such as ASCII only map latin characters to bytes. Others cannot exist!

Software and services need to be **accessible to all people in their native languages**. Always use string representations that can represent all languages (and emoji!) such as **unicode**, and encodings into bytes such as **utf-8** to represent those strings as bytes.

- Don't assume English is the only possible language.
- Plan for internationalization: do not hard code english strings into software, but rather load them from a file that can be localized.
- Dates, currency, number representations also vary per locale.

# Operations on string.

Strings support a number of operations:

```
>>> s = "All quiet on the western front"
>>> s[10]          # Indexing
'o'
>>> s[4:9]         # Slicing
'quiet'
>>> s[4:9] + s[-6:] # Concatenation & neg indexing
'quiet front'
>>> s.split(' ')    # Split to str list
['All', 'quiet', 'on', 'the', 'western', 'front']
>>> "hello" < "world" # Comparison
True
```

For full reference see:

<https://docs.python.org/3.7/library/string.html>



## String literals & formatting.

- String constants in programs can be enclosed in single or double quotes.
- You can represent special characters in strings using the ‘\’ symbol, such as new line as `\n` and tab as `\t`.
- The `format` method substitutes into the string:

```
>>> "My name is {name}".format(name = "Alice")
'My name is Alice'
>>> "Dec: {num:d} | Hex: {num:x} | Oct: {num:o}".format(num=14)
'Dec: 14 | Hex: e | Oct: 16'
>>> template = "Float: {num:.2e} Fixed: {num:.2f} Perc: {num:.2%}"
>>> template.format(num=0.12345)
'Float: 1.23e-01 Fixed: 0.12 Perc: 12.35%'
```

## Generic programming: how to sort a list of strings.

Do you need to **re-implement mergesort** for lists of strings? Remember the **abstraction principle**.

However our implementation of mergesort seems to just work:

```
def test_str_sort():  
    from mergesort import mergesort  
    items = ["BB", "A", "C", "DAAX", "BA"]  
    sorted_items = ["A", "BA", "BB", "C", "DAAX"]  
    assert mergesort(items) == sorted_items
```

**Generic programming** ensures that algorithms are independent of and agnostic about the exact types they are processing.

## Why generic programming works.

Merge (algorithm 1) only uses ' $\leq$ ' on items within the list to sort:

```
# while either list has any items left
while index1 < len1 or index2 < len2:
    if (index2 == len2      #either we've used all of lst2
        or (index1 != len1  #or both lists have elements left
            and lst1[index1] <= lst2[index2])): #and lst1 is smaller
        merged_lst.append(lst1[index1])
        index1 += 1
    else:
        merged_lst.append(lst2[index2])
        index2 += 1
```

If the operation less-than-or-equal (compare) is supported correctly by the type of object in the list, mergesort will return the correct result.

## Sorting strings by length, rather than lexicographically.

Strategy: allow the programmer to specify a function that performs a comparison. Parametrize mergesort by the comparison function.

```
def mergesort(lst, compare):  
    if len(lst) <= 1:  
        return lst.copy()  
    pivot = len(lst) // 2  
    lst1 = mergesort(lst[:pivot], compare)  
    lst2 = mergesort(lst[pivot:], compare)  
    return mergelists(lst1, lst2, compare)
```

```
def mergelists(lst1, lst2, compare):  
    ''' Merge two sorted lists into a sorted list.'''  
    merged_lst = []  
    len1 = len(lst1)  
    len2 = len(lst2)  
    index1 = 0  
    index2 = 0  
    # while either list has any items left  
    while index1 < len1 or index2 < len2:  
        if (index2 == len2 #either we've used all of lst2  
            or (index1 != len1 #or both lists have elements left  
                and compare(lst1[index1], lst2[index2]))):  
            merged_lst.append(lst1[index1])  
            index1 += 1  
        else:  
            merged_lst.append(lst2[index2])  
            index2 += 1
```

## Example of different comparison functions for strings.

```
def test_order():  
    items = ["A", "BB", "CCC", "DD", "E"]  
  
    def compare_lex(l1, l2):  
        return l1 <= l2  
    assert mergesort(items, compare_lex) == items  
  
    def compare_len(l1, l2):  
        return len(l1) <= len(l2)  
    assert mergesort(items, compare_len) \  
        == ["A", "E", "BB", "DD", "CCC"]
```

# Functions are just variables.

## Functions are first-class objects.

In modern programming languages functions are first-class objects (of type function). They are assigned to names, and they can be passed as arguments to function calls; stored in data structures; etc.

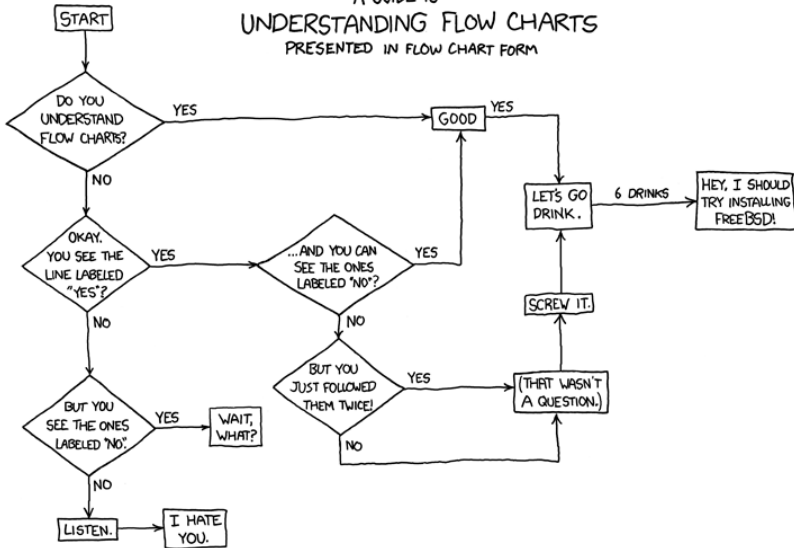
The shorthand `lambda` notation can be used to define functions within expressions.

```
def test_order_lambda():  
    items = ["A", "BB", "CCC", "DD", "E"]  
  
    assert mergesort(items, lambda x,y: x <= y) == items  
    assert mergesort(items, lambda x,y: len(x) <= len(y)) \  
        == ["A", "E", "BB", "DD", "CCC"]
```

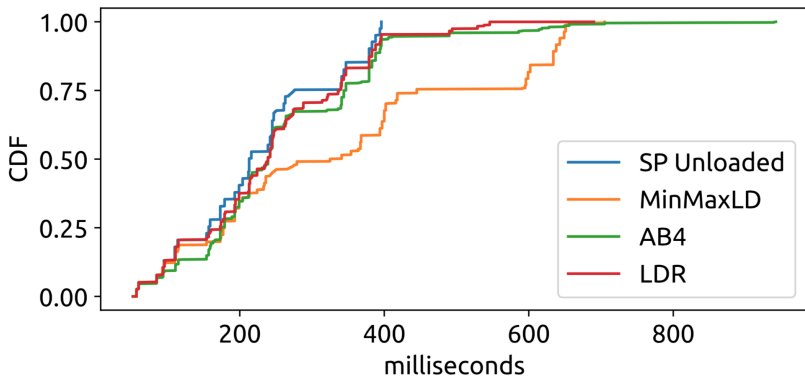
## Visualization Diversion



A GUIDE TO  
UNDERSTANDING FLOW CHARTS  
PRESENTED IN FLOW CHART FORM



## Cumulative Distribution Functions (CDFs)



y value shows the cumulative fraction of data values less than the x value