Designing Efficient Algorithms

Learning Objectives

- Recognize the requirements for building a good hash function and a good hashtable that lead to constant-time search
- Recognize and trace the algorithms for selection sort and merge sort
- Compare and contrast the efficiency and Big-O runtimes of selection sort and merge sort

Increase Efficiency by Cutting Extra Work

We've talked about how to determine the efficiency of an algorithm, but we haven't addressed a more important question. How can we design algorithms to make them more efficient?

Sometimes making a program more efficient is easy; you just need to look for unnecessary actions (statements that aren't used, loops that repeat work already done) and cut them.

```
def findLargest(lst):
        largest = lst[0]
        for i in range(len(lst)):
                for j in range(len(lst)):
                        if lst[i] > largest and lst[i] > lst[j]:
                                largest = lst[i]
        return largest


# could be

def findLargest(lst):
        largest = lst[0]
        for i in range(1, len(lst)):
                if lst[i] > largest:
                        largest = lst[i]
        return largest
```

Increase Efficiency by Thinking Differently

More often we increase the efficiency of an algorithm by thinking about the problem in a different way.

The obvious solution to a problem isn't always the most efficient. We can often make a faster solution by using a different data structure or an entirely different algorithmic approach.

We'll look at two case studies of this today, with search and sorting.

Note: we won't ask you to make your algorithms more efficient in this class as a primary learning goal, but it's still useful to know about!

Optimizing Search

Improving Search

We've discussed linear search (which runs in O(n)), and binary search (which runs in O(log n)).

We use search all the time, so we want to search as quickly as possible.
Can we search for an item in O(1) time?

We can't always search for things in constant time, but there are certain circumstances where we can.

Search in Real Life – Post Boxes

Consider how you receive mail. Your mail is sent to the post boxes at the lower level of the UC. Do you have to check every box to find your mail?

No-- just check the one assigned to you.

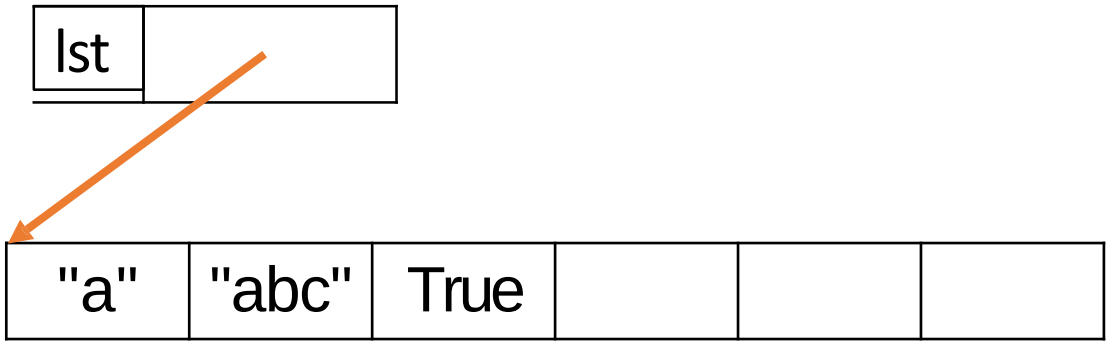
This is possible because your mail has an address on the front that includes your mailbox number. Your mail will only be put into a box that has the same number as that address, not other random boxes.

Picking up your mail is a O(1) operation!

Search in Programming – List Indexes

We can't search a list for an item in constant time, but we can look up an item based on an index in constant time.

Reminder: Python stores lists in memory as a series of adjacent parts. Each part holds a single value in the list, and all these parts use the same amount of space.



Example:

```
lst = ["a", "abc", True]
```

Search in Programming – List Indexes

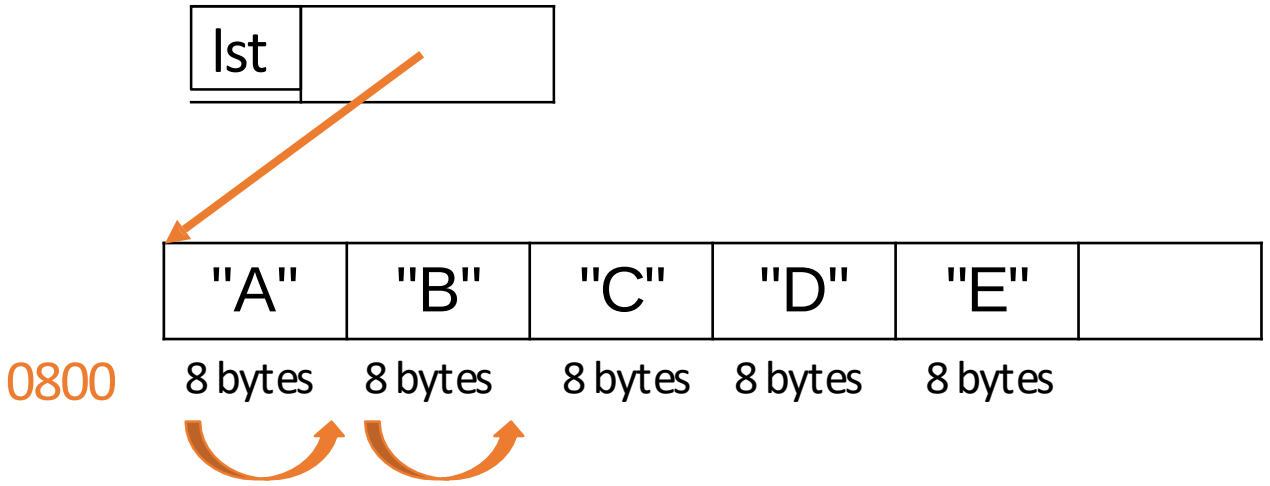
We can calculate the exact starting location of a list index's memory address based on the first address where lst is stored. If the size of a part is N, we can find an index's address with the formula:

start + N * index

Example: in the list to the right, each part is 8 bytes in size and the memory values start at 0800. To access lst[2], compute:

0800 + 8 * 2 = 0816

Given a memory address, we can get the value from that address in constant time. Looking up an index in a list is O(1)!



Combine the Concepts

To implement constant-time search, we want to combine the ideas of post boxes and list index lookup. Specifically, we want to determine which index a value is stored in based on the value itself.

If we can calculate the index based on the value, we can retrieve the value in constant time.

Hash Functions Map Values to Integers

In order to determine which list index should be used based on the value itself we'll need to map values to indexes (integers).

We call a function that maps values to integers a hash function. This function must follow two rules:

- Given a specific value x, hash(x) must always return the same output i
- Given two different values x and y, hash(x) and hash(y) should usually return two different outputs, i and j

Built-in Hash Function

We don't need to write our own hash function most of the time- Python already has one!

```
x = "abc"
hash(x) # some giant number
```

hash() works on integers, floats, Booleans, strings, and some other types as well.

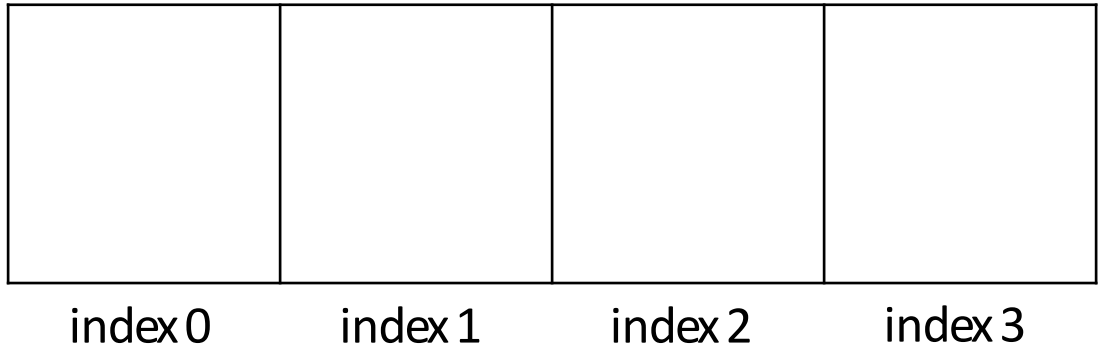
Optimizing Search: Hashtables

Hashtables Organize Values

Now that we have a hash function, we can use it to organize values in a special data structure.

A hashtable is a list with a fixed number of indexes. When we place a value in the list, we put it into an index based on its hash value instead of placing it at the end of the list.

We often call these indexes 'buckets'. For example, the hashtable to the right has four buckets. Important: actual hashtables have far more buckets than this.



Adding Values to a Hashtable

For simplicity, let's say this hashtable uses a hash function that maps strings to indexes using the first letter of the string, as shown to the right. (This is not a good hash function, but it will serve as an example).

```
def hash(s):
    return ord(s[0]) - ord('a')
```

First, add "book" to the table. hash("book") is 1, so we'll put the value in bucket 1.

Next, add "yay". hash("yay") is 24, which is outside the range of our table. How do we assign it?

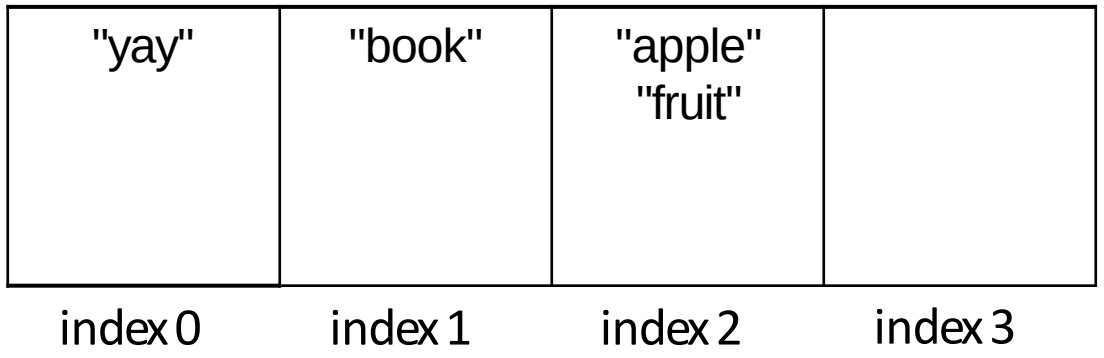
Use value % tableSize to map integers larger than the size of the table to an index. 24 % 4 = 0, so we put "yay" in bucket 0.

Dealing with Collisions

When you add lots of values to a hashtable, two elements may collide. This happens if they are assigned to the same index. For example, if we try to add both "apple" and "fruit" to our table, they will collide.

```
def hash(s):
        return ord(s[0]) - ord('a')
```

Hashtables are designed to handle collisions. One algorithm for handling collisions is to put the collided values in a list and put that list in the bucket. If your table size is reasonably big and the indexes returned by the hash function are reasonably spread out, each bucket will normally hold a constant number of values.



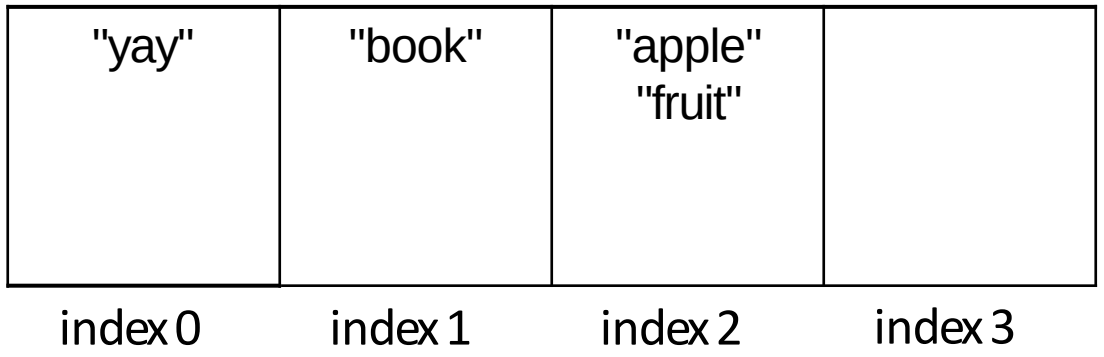
Our example hash function is not good because it only looks at the first letter. A function that uses all the letters would be better.

You Do: Search a Hashtable

Let's say that we want to algorithmically check whether the string "friday" is in our hashtable.

```
def hash(s):
    return ord(s[0]) - ord('a')
```

You do: Which buckets does the algorithm need to check?



Searching a Hashtable is O(1)!

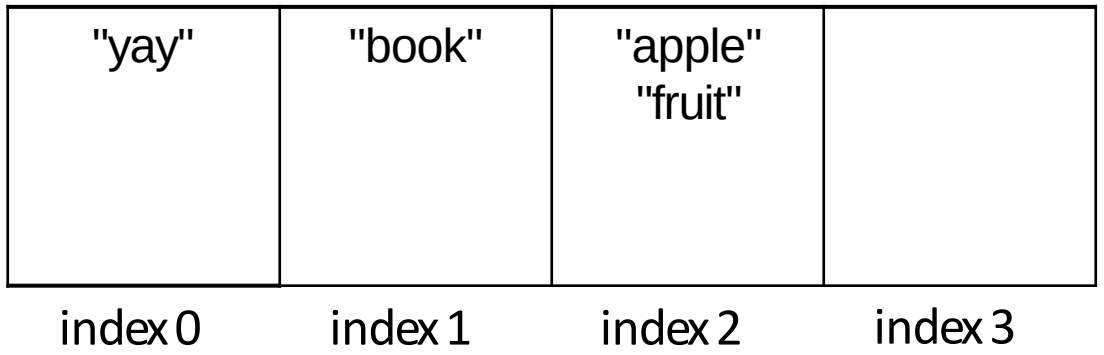
To search for a value, call the hash function on it and mod the result by the table size. The index produced is the only index you need to check!

For example, we can check if "book" is in the table just by checking bucket 1.

If the value is in the table, it will be at that index. If it isn't, it won't be anywhere else either. To check for "stella" just look in in bucket 2.

Because we only need to check one index and each index holds a constant number of items, finding a value is O(1).

```
def hash(s):
        return ord(s[0]) - ord('a')
```



Caveat: Don't Hash Mutable Values!

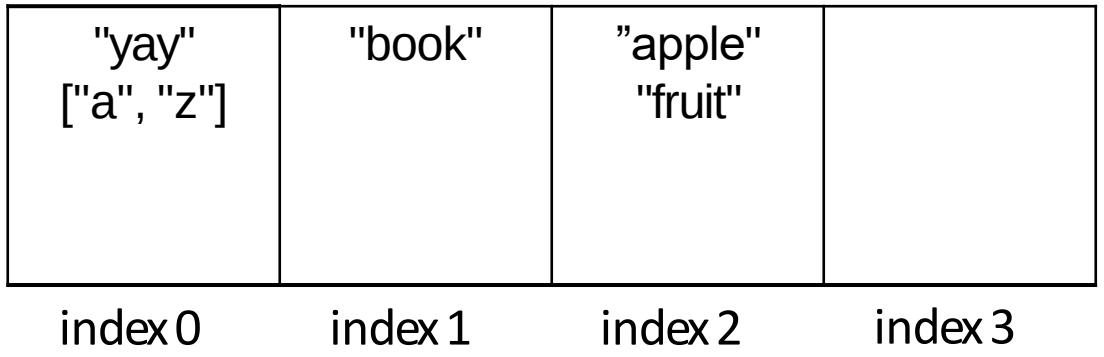
What happens if you try to put a list in a hashtable? Let's set `lst = ["a", "z"]` and use the given hash to add `lst`.

This might seem fine at first, but it will become a problem if you change the list before searching. Let's say we set `lst[0] = "d"`.

When we hash the list again, the hashed value is 3, not 0. But the list isn't stored in bucket 3! We can't find it reliably.

For this reason, we don't put mutable values into hashtables. If you try to run the built-in `hash()` on a list, it will crash.

```
def hash(s):
        return ord(s[0]) - ord('a')
```



Dictionaries Use Hashed Search

Because hashed search requires immutable search values and a hashtable, it isn't used in lists or strings. However, it is used to implement dictionary search.

Recall that the keys of a dictionary must be immutable. This is because those keys are all stored in a hashtable. Each key points to its own value; that's how values can still be accessed.

This means that searching for a key in a dictionary takes O(1) time! Dictionaries are super efficient for basic lookup tasks.

Searching Dictionaries vs. Lists

Recall the built-in operator `in`, which checks for membership in a data structure.

`item in lst` runs in linear time if `lst` is a list, because Python can't guarantee that the list is sorted. It uses linear search.

`item in dict` runs in constant time if `dict` is a dictionary due to hashing.

If you know that you'll need to do a lot of searching for specific values, it's better to store your data in a dictionary than a list, even if its a sorted list.

Coding Efficiently With Dictionaries

Here's an example of how to increase a function's efficiency with dictionary search. Say you want to check whether there are any duplicates in a dataset. This is commonly needed in data analysis to make sure datapoints aren't double-counted.

If we try to check every element in a list using in, it will take $O(n^2)$ time (n-1 actions * n items checked).

If we instead move the items to a dictionary, it takes O(n) time (constant actions * n items checked).

```
def hasDuplicates(studentIDs):
    for i in range(len(studentIDs)):
        others = studentIDs[:i] + studentIDs[i+1:]
        if studentIDs[i] in others:
            return True
    return False
# vs
def hasDuplicates(studentIDs):
    studentDict = { }
    for student in studentIDs:
        if student in studentDict:
            return True
        else:
            studentDict[student] = 1
    return False
```

Optimizing Sorting: Selection Sort

Many Ways of Sorting

Sorting items (putting them in order based on a comparison rule) is as prevalent in computer science and algorithmic thinking as searching is.

Computer scientists have designed dozens of algorithms for how to sort a list under different circumstances as a result. Some of these algorithms are generally more efficient than others.

We'll start with a straightforward sorting algorithm, then show how to sort more efficiently by taking an entirely different approach.

Selection Sort Sorts from Smallest to Largest

If you were asked to sort a large stack of books by title, you might do so by looking for the last title and putting it at the bottom, then look for the next-to-last-title and put it second-to-last, etc. This is called selection sort – you're selecting which item to sort next.

The core idea of selection sort is that you sort from smallest to largest (or largest to smallest). This is a very intuitive way to sort.

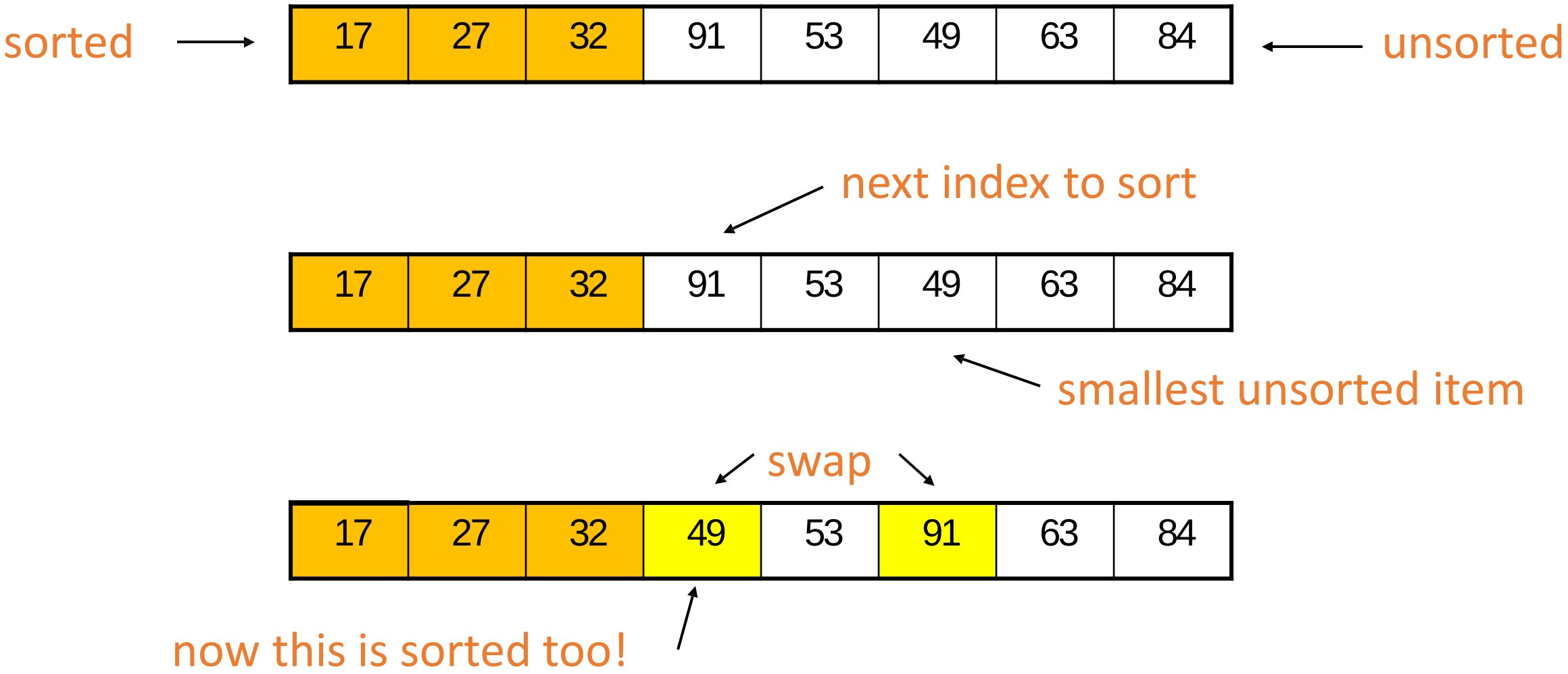
See an example here: https://visualgo.net/en/sorting

Selection Sort Algorithm

1. Start with none of the list sorted
2. Repeat the following steps until the whole list is sorted:
   a) Search the unsorted part of the list to find the smallest element
   b) Swap the found element with the first unsorted element
   c) Increment the size of the 'sorted' part of the list by one

Note: in Python, swapping the element currently in the front position with the smallest element is faster than sliding all of the numbers down in the list.

Selection Sort Process



Swapping Elements in a List

It's common to swap elements in lists as we sort them. Let's implement swapping separately from the rest so we can count it as a single action.

To swap two elements, you need to create a temporary variable to hold one of them. This keeps the first element from getting overwritten.

```
def swap(lst, i, j):
    tmp = lst[i]
    lst[i] = lst[j]
    lst[j] = tmp
```

Selection Sort Code

```
def selectionSort(lst):
        # i is the index of the first unsorted element
        # everything before it is sorted
        for i in range(len(lst)-1):
                # find the smallest element
                minIndex = i
                for j in range(minIndex + 1, len(lst)):
                        if lst[j] < lst[minIndex]:
                                minIndex = j # the element at this index is smaller
                swap(lst, i, minIndex)
        return lst

lst = [2, 4, 1, 5, 10, 8, 3, 6, 7, 9]
sortedLst = selectionSort(lst)
print(sortedLst)
```

Selection Sort – Efficiency Analysis

Sort algorithms do lots of different operations. For now, let's just consider comparisons and swaps.

We'll also refer to individual passes of the sorting algorithm. A pass is a single iteration of the outer loop (or putting a single element into its sorted location).

Selection Sort Code – Comparisons and Swaps

```
def selectionSort(lst):
    # i is the index of the first unsorted element
    # everything before it is sorted
    for i in range(len(lst)-1):
        # find the smallest element
        minIndex = i
        for j in range(minIndex + 1, len(lst)):
            if lst[j] < lst[minIndex]:
                minIndex = j # the element at this index is smaller
        swap(lst, i, minIndex)
    return lst

lst = [2, 4, 1, 5, 10, 8, 3, 6, 7, 9]
lst = selectionSort(lst)
print(lst)
```

A single iteration of this is a pass

Comparison

Swap

Selection Sort – Comparisons and Swaps

What's the worst case input for Selection Sort?

Answer: Any list, really. The list doesn't affect the actions taken.

How many comparisons does Selection Sort do in the worst case if the input list has n elements?

Search for 1st smallest: 1 comparisons
Search for 2nd smallest: 2 comparisons
...
Search for 2nd-to-last smallest: 1 comparison

Total comparisons: $(n-1) + (n-2) + \ldots + 2 + 1 = n * (n-1) / 2 = n^2/2 - n/2$

You do: how many swaps happen per pass?

Selection Sort – Efficiency

The algorithm does a single swap at the end of each pass, and there are n-1 passes, so there are n-1 swaps.

Overall, we do $n^2/2 - n/2 + n-1$ actions. However, we don't care about the lower-order terms or constants.

Selection sort is $O(n^2)$. That's not bad, but we can do better!

Optimizing Sorting: Merge Sort

Improve Efficiency with a Drastic Change

If we want to do better than $O(n^2)$, we need to make a drastic change in our algorithm.

Instead of thinking iteratively, what if we think recursively? Can we solve this using delegation?

One common strategy is Divide and Conquer:

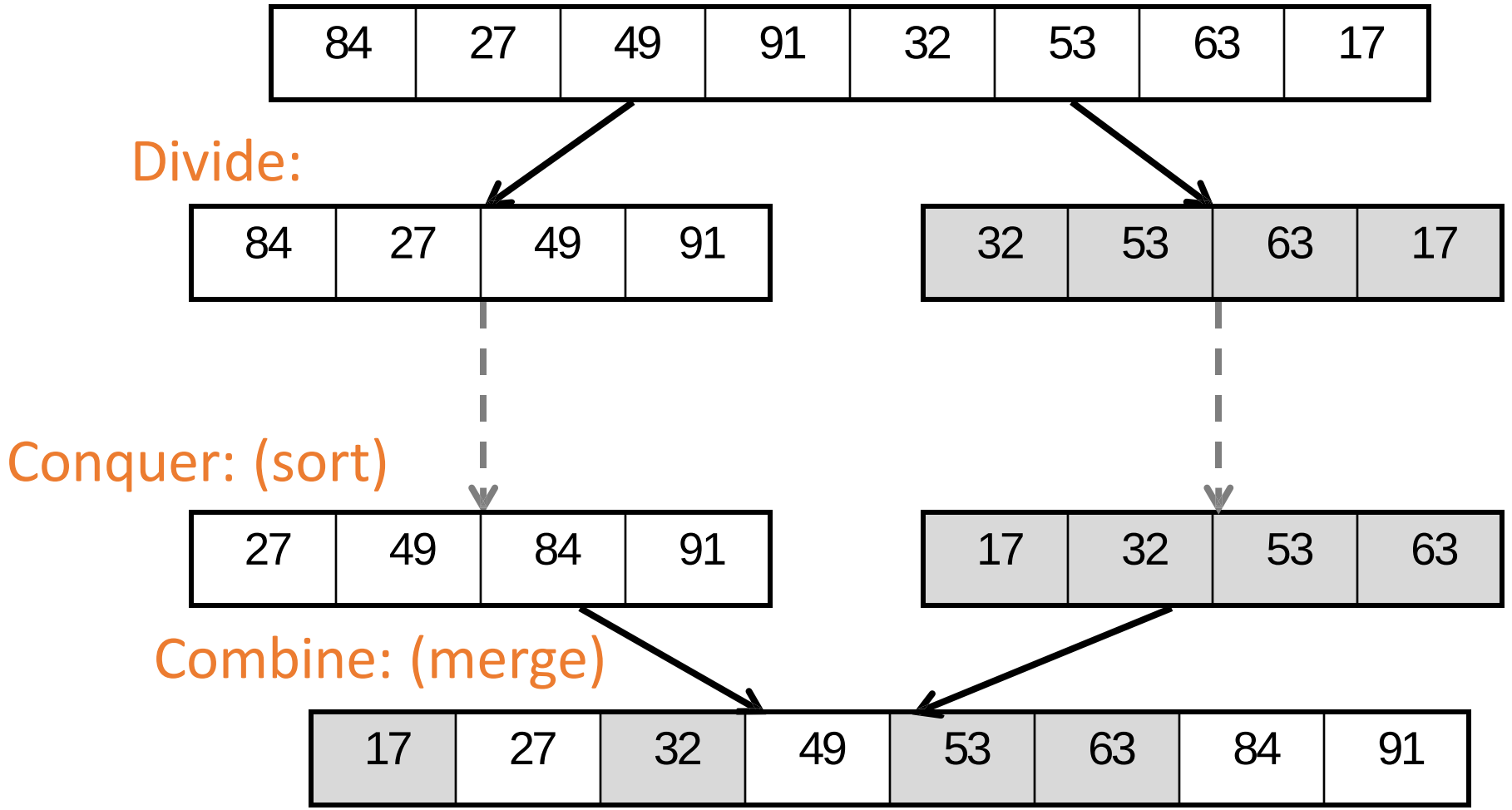
1. Divide the problem into "simpler" versions of itself (usually in two halves).
2. Conquer each problem using the same process (usually recursively).
3. Combine the results of the "simpler" solutions to form the final solution.

Merge Sort Delegates, Then Merges

Merge sort is an algorithm that sorts using divide and conquer. The core idea of the Merge Sort algorithm is that you sort by merging. It's a little unintuitive but will help us improve efficiency.

1. If there are 0 or 1 elements, return the list itself (already sorted)
2. Otherwise…
    1. Delegate sorting the front half of the list (recursion!)
    2. Delegate sorting the back half of the list (recursion!)
    3. Merge the two sorted halves into a new sorted list.

Merge Sort Process



Merge Sort Code

```
def mergeSort(lst):
    # base case: 0-1 elements are already sorted
    if len(lst) < 2:
        return lst
    # divide
    mid = len(lst) // 2
    front = lst[:mid]
    back = lst[mid:]
    # conquer by sorting
    front = mergeSort(front)
    back  =  mergeSort(back)
    # combine sorted halves
    return merge(front, back)
```

Merge By Checking the Front of the Lists

How do we merge two sorted lists?

Visualize it here: https://visualgo.net/en/sorting

1. Create a new empty 'result' list
2. Keep track of two pointers to the two lists, each starting at the first element
3. Repeat the following until we've added all the elements of one of the lists:
   a) Compare the pointed-to elements in each of the two lists
   b) Copy the smaller element to the end of the result list
   c) Move the pointer from the smaller element to the next one in that list
4. Move the rest of the unfinished list to the end of the result list

Merge Code

```
def merge(half1, half2):
        result = [ ]
        i = 0
        j = 0
        while i < len(half1) and j < len(half2):
                # only compare first two (guaranteed to be smallest due to sorting)
                 if half1[i] < half2[j]:
                        result.append(half1[i])
                        i = i + 1
                else:
                        result.append(half2[j])
                        j = j + 1
        # add remaining elements (only one of the halves still has values)
        result = result + half1[i:] + half2[j:]
        return result
```

Merge Sort – Efficiency Analysis

Merge Sort doesn't have swaps; it has copies. We'll consider the number of comparisons and copies that are performed.

Merge sort will have two kinds of passes: split-passes (splitting a list into two halves) and merge-passes (merging two lists into one).

What's the worst case input?

Any list; it doesn't matter.

Merge Sort Code

```
def mergeSort(lst):
    if len(lst) < 2:
        return lst
    mid = len(lst) // 2
    front = lst[:mid]
    back = lst[mid:]
    front = mergeSort(front)
    back = mergeSort(back)  return
    merge(front, back)

lst = [2, 4, 1, 5, 10, 8, 3, 6, 7, 9]
sortedLst = mergeSort(lst)
print(sortedLst)
```

Copy

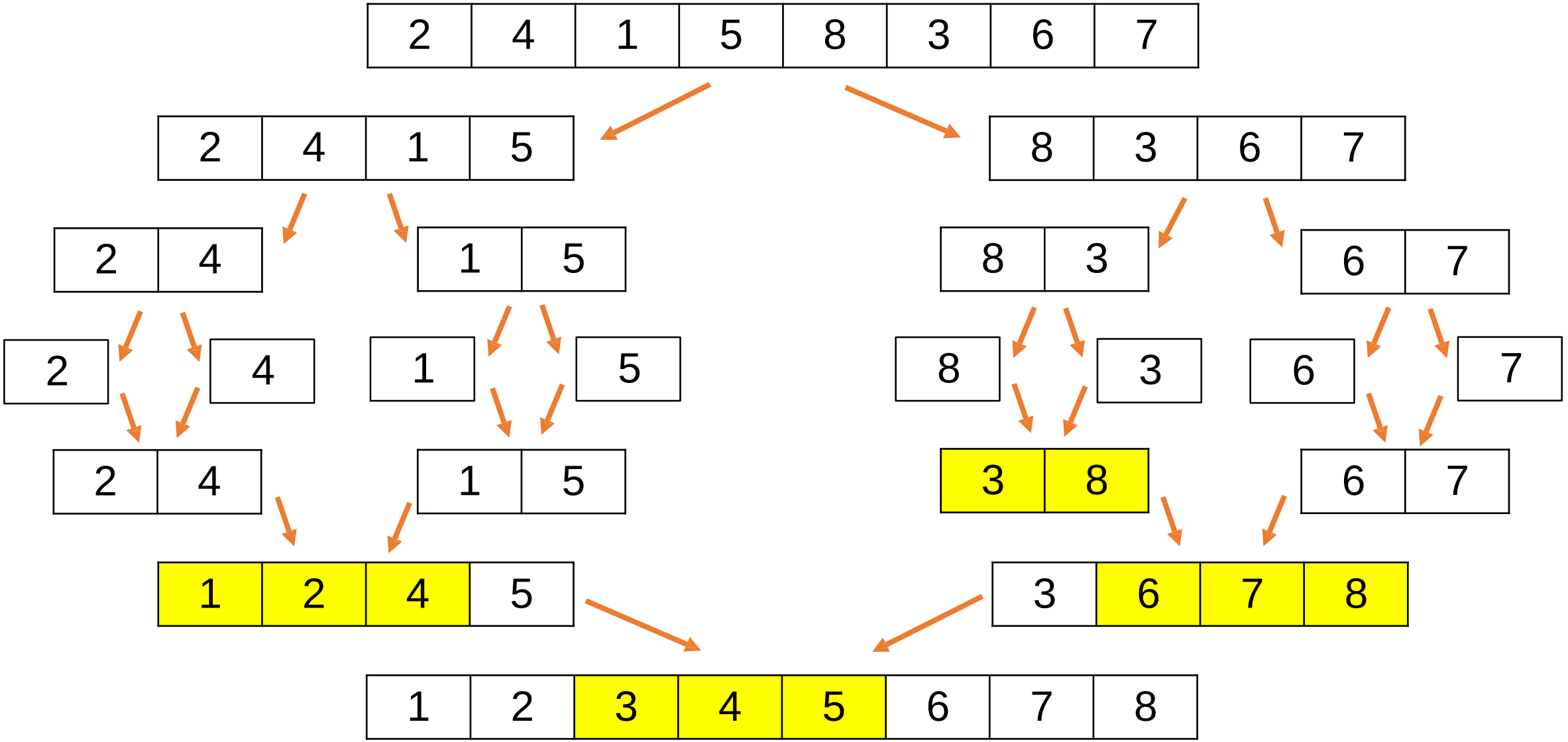
```
def merge(half1, half2):
    result = [ ]  i = 0
    j = 0
    while i < len(half1) and j < len(half2):
        if half1[i] < half2[j]:
            result.append(half1[i])
            i = i + 1
        else:
            result.append(half2[j])
            j = j + 1
    result = result +  half1[i:]  +  half2[j:]
    return result
```

Comparison

Copy

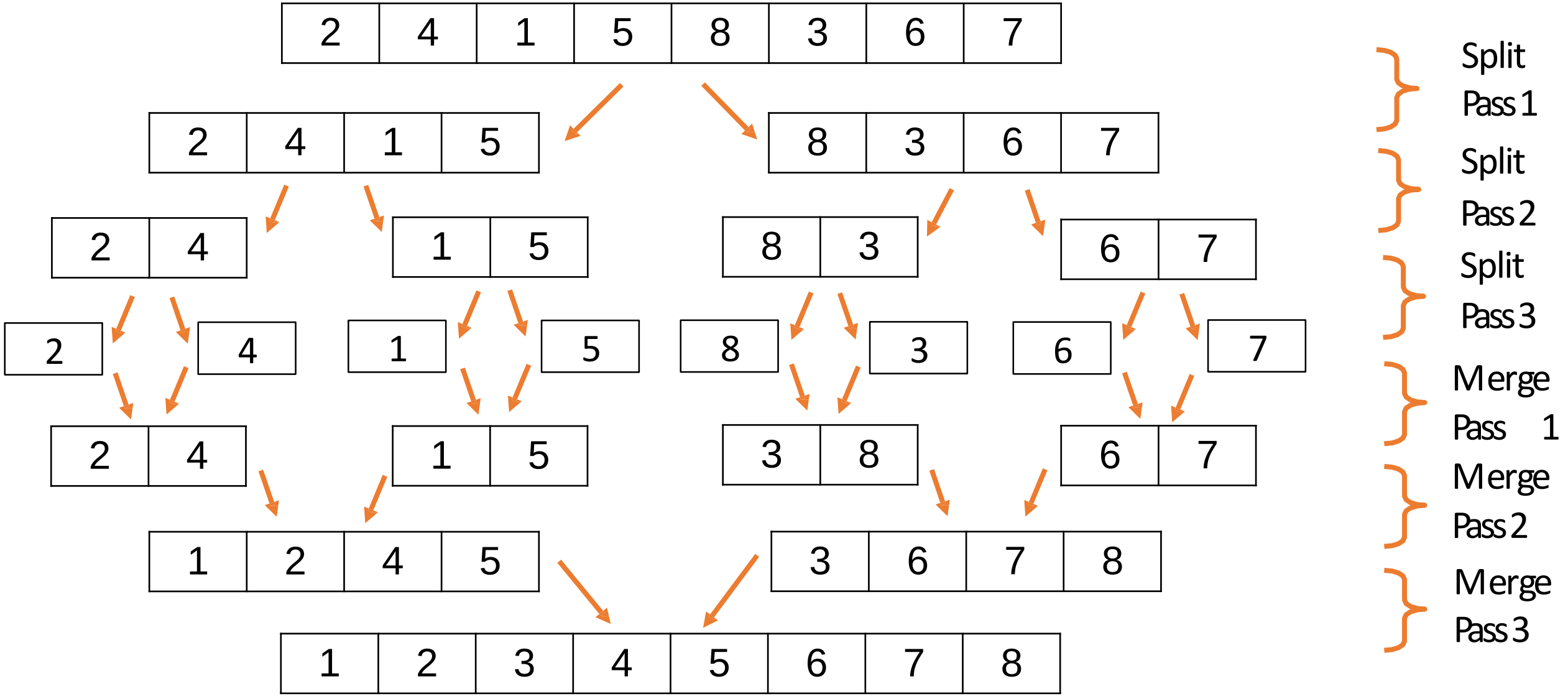
Copy

Merge Sort Call Breakdown



Merge Sort Call Breakdown

n copies in each split-pass
n copies + ~n comparisons in each merge-pass



Merge Sort Efficiency

How many split-passes and merge-passes occur?

Every time a split-pass occurs, we cut the number of elements being sorted in half. The number of split-passes is the number of times we can divide the list in half.

We have one merge-pass for each split-pass, so that same number is used for merge-passes.

That means there are $\log_2 n$ split-passes and $\log_2 n$ merge-passes. Overall

work: $n \log n + 2 * (n \log n) = 3 * (n \log n) = O(n \log n)$

Efficiency of Selection vs. Merge Sort

O(n log n) [or O(n * log n)] may not seem a lot better than $O(n^2)$ [or O(n*n)], but the difference shows when you get up to large datasets!

n	selection sort $n^2/2 + n/2 - 1$	merge sort $3 n \log_2 n$	Ratio (selection / merge)
8 (2^3)	35	72	0.49
16 (2^4)	135	192	0.70
32 (2^5)	527	480	1.1
1024 (2^{10})	524,799	30,720	17.1
1,048,576 (2^{20})	549,756,338,175	62,914,560	**8738.1**

Discuss: how might the efficiency change if we split into three parts instead of two?