# Introduction to Algorithms

## Review

How does **linear search** work?

How does **binary search** work?

What is the **key instruction** usually used to compare their performance?

Which is usually more efficient? By how much?

## The Sorting Problem

What is the statement of the **sorting problem**?

How can you re-arrange the values in a list so that they are in increasing order (according to <=) ?

Examples of Python’s **built-in list sort**:

>>> lst = [9, 3, 1, 4, 3]

>>> lst.sort()

>>> lst

[1, 3, 3, 4, 9]

>>> words = ['shell', 'nose', 'apple', 'tree', 'shoe']

>>> words.sort()

>>> words

['apple', 'nose', 'shell', 'shoe', 'tree']

Does Python’s built-in list sort modify the original list? Or does it leave the original unchanged and return a sorted copy?

## Selection Sort

What is the intuitive idea of **selection sort**? How could you use it to sort things by hand?

How can you implement **selection sort in Python**?

def selection\_sort(lst):

result = []

while len(lst) > 0:

smallest = min(lst)

result.append(smallest)

lst.remove(smallest)

return result

Examples of running selection sort:

>>> selection\_sort([9, 3, 1, 4, 3])

[1, 3, 3, 4, 9]

>>> selection\_sort(['shell', 'nose', 'apple', 'tree', 'shoe'])

['apple', 'nose', 'shell', 'shoe', 'tree']

Does this implementation of selection\_sort **modify** the original list? Or does it leave the original unchanged and return a sorted copy?

How does the **performance of selection sort** compare to Python’s built-in sort?

## Mergesort

What does it mean to **merge** two lists?

[1, 4, 5, 8, 10]

[2, 3, 6, 8]

Does Python provide a merging function in one of its standard modules?

import heapq

def merge(lst1, lst2):

"""Combines lst1 and lst2 into a new sorted list.

lst1 and lst2 must both be in ascending sorted order.

"""

return list(heapq.merge(lst1, lst2))

How can you use merging to sort a list?

[5, 9, 1, 4, 3, 2, 6]

How can you implement **mergesort** in Python?

def mergesort(lst):

"""Returns a copy of lst with its items re-arranged into   
 ascending order.

Uses mergesort.

"""

n = len(lst)

if n < 2:

return lst[:] # already sorted

else:

mid = n // 2

left = lst[:mid]

right = lst[mid:]

left\_sorted = mergesort(left) # recursive call

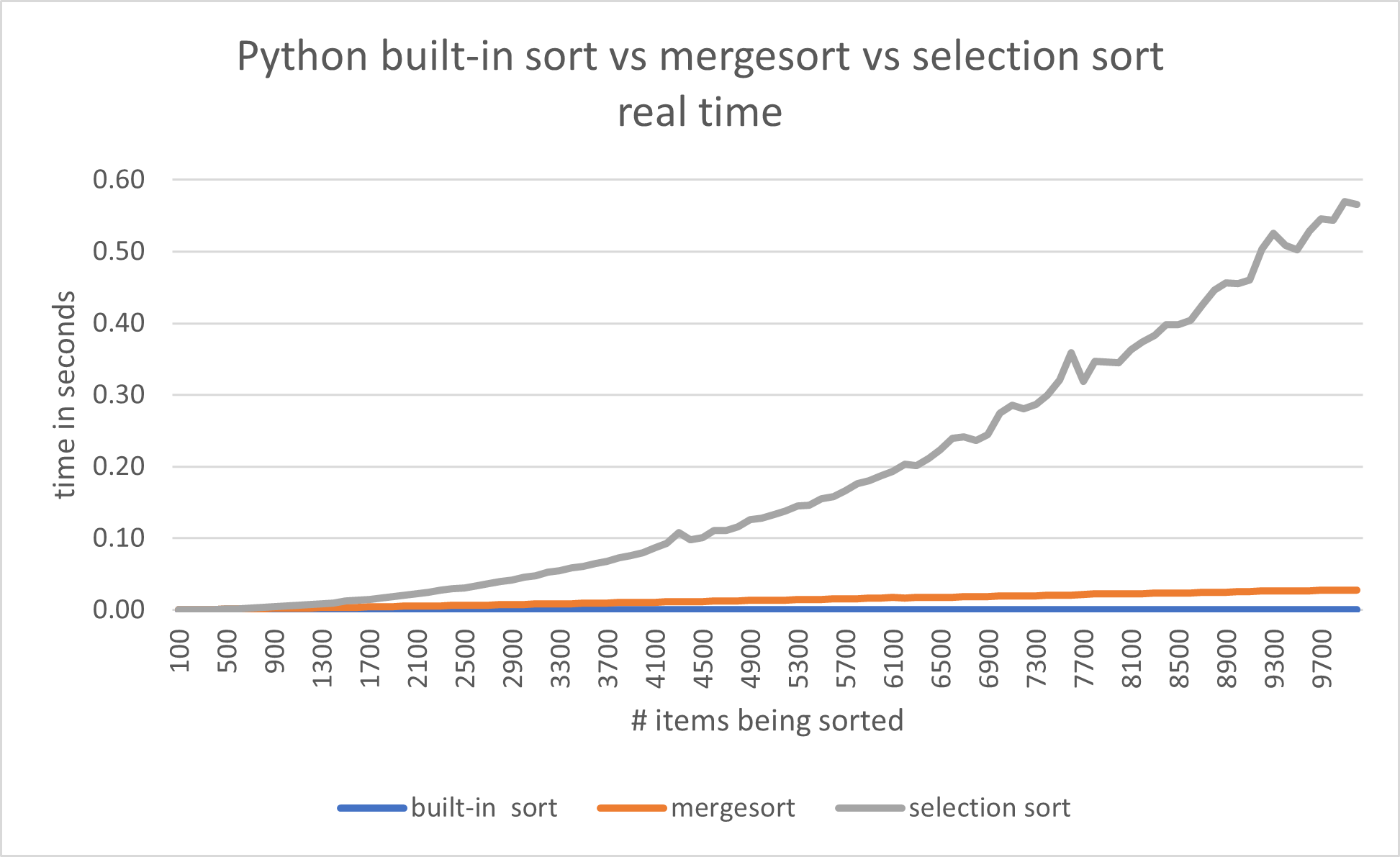
right\_sorted = mergesort(right) # recursive call

return merge(left\_sorted, right\_sorted)

What does it mean that mergesort is **recursive**?

Does this implementation of mergesort **modify** the original list? Or does it leave the original unchanged and return a sorted copy?

How does the **performance of mergesort** compare to selection sort and Python’s built-in sort?



What is the shape of the performance curve for **selection sort** (ignoring the “noise”)?

What is the shape of the performance curve for **mergesort** and **Python’s built-in sort**?

|  |  |
| --- | --- |
| Algorithm | Worst-case running time |
| Selection sort |  |
| Mergsort |  |

Suppose it takes **selection sort** 5 seconds to sort 10,000 values. About how long would you expect **selection sort** to take to sort 20,000 values?