**Benchmark Analysis**

**Linked Lists vs Python Lists  
  
A screenshot of a computer program

AI-generated content may be incorrect.**

**Find Max Sorted List, Find Max Unsorted List**

**A screenshot of a computer

AI-generated content may be incorrect.**

**HTML Search with Regex & Stack**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Time Complexity Analysis**

*html\_search\_regex\_ver.py – O(n).*

The time complexity for reading from the file is O(n) where n is the characters of the file. The time complexity of the regex.findall() regular expression would also be O(n), since the regex expression scans the entire file making n the characters of the file. Then there’s processing tags, which are O(m), where m is the number of tags in the HTML content that is processed once in the loop. Finally the last return check is O(1), because it checks the status if the stack is empty. There for I believe the overall is O(n).

*html\_search\_stack\_ver.py – O(n)*

The time complexity for reading from the file is O(n) where n is the characters of the file. The time complexity of content for the while loop is O(n) as it’s going over each character, still n is the characters in the file. Finding closing tags is O(n) in the worst case but it’s in the while loop so it’s still O(n) anyway. Pushing and popping items from the stack is O(1) and the final check is also O(1) as it’s just checking if the stack is empty. There fore the overall is O(n).

*recursion\_max\_sorting.py – O(n log n)*

The time complexity of checking the list is empty is O(1), because it checks the length of the list if it’s empty. The time complexity of checking the list has only one element is O(1), because it checks the length of the list and if is only one element, it returns that element. The time complexity for sorting the list is O(n log n), where n is elements in the list and it has to iterate over elements within elements of the list. Finally returning the result is just O(1). I’d say the overall is O(n log n) because it’s the most time consuming of the overall process.

*recursion\_max\_unsorting.py – O(n)*

The time complexity of checking the list is empty is O(1), because it checks the length of the list if it’s empty. The time complexity of checking the list has only one element is O(1), because it checks the length of the list and if is only one element, it returns that element. The time complexity for the return, recursive call checks each element in the list, which is n for each element in the list. Each call though involves finding the elements between two elements, which is just O(1). But because the process goes through each element (n) in the list and it’s reduced to O(1) complexity then I believe the overall is O(n).

**Discussion**

**Code**

custom\_linkedlist.py

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

# Linked list

class LinkedList:

    # \_\_init\_\_() method to initialize the linked list

    def \_\_init\_\_(self):

        self.head = None

        self.tail = None

    # \_\_str\_\_() method to return a string representation of the linked list

    def \_\_getitem\_\_(self, index):

        if index < 0 or index >= len(self):

            raise IndexError("list index out of range")

        current = self.head

        for \_ in range(index):

            current = current.next

        return current.data

    # \_\_setitem\_\_() method to set the value of an element at a specific index in the linked list

    def \_\_setitem\_\_(self, index, value):

        if index < 0 or index >= len(self):

            raise IndexError("list index out of range")

        current = self.head

        for \_ in range(index):

            current = current.next

        current.data = value

    # \_\_delitem\_\_() method to delete an element at a specific index in the linked list

    def \_\_delitem\_\_(self, index):

        current = self.head

        for \_ in range(index - 1):

            current = current.next

        current.next = current.next

    # \_\_iter\_\_() method to iterate through the linked list

    def \_\_iter\_\_(self):

        current = self.head

        while current:

            yield current.data

            current = current.next

    # \_\_contains\_\_() method to check if an element is present in the linked list

    def \_\_contains\_\_(self, item):

        current = self.head

        while current:

            if current.data == item:

                return True

            current = current.next

        return False

    # \_\_len\_\_() method to get the length of the linked list

    def \_\_len\_\_(self):

        current = self.head

        count = 0

        while current:

            count += 1

            current = current.next

        return count

    # \_\_getitem\_\_() method to get the element at a specific index in the linked list

    def \_\_getitem\_\_(self, index):

        current = self.head

        for \_ in range(index):

            current = current.next

        return current.data

    def append(self, data):

        new\_node = Node(data)

        if self.head is None:

            self.head = self.tail = new\_node

        else:

            self.tail.next = new\_node

            self.tail = new\_node

    # pop() method to remove the last element from the linked list

    def pop(self):

        if self.head is None:

            raise IndexError("pop from empty list")

        if self.head == self.tail:

            self.head = self.tail = None

        else:

            current = self.head

            prev = None

            while current.next:

                prev = current

                current = current.next

            if prev:

                prev.next = None

            self.tail = prev

    # pop\_at\_index() method to remove the element at a specific index from the linked list

    def pop\_at\_index(self, index):

        current = self.head

        for \_ in range(index - 1):

            current = current.next

        current.next = current.next

    # insert() method to insert an element at a specific index in the linked list

    def insert(self, index, data):

        new\_node = Node(data)

        current = self.head

        for \_ in range(index - 1):

            current = current.next

        new\_node.next = current.next

        current.next = new\_node

    # reverse() method to reverse the linked list

    def reverse(self):

        prev = None

        current = self.head

        while current:

            next\_node = current.next

            current.next = prev

            prev = current

            current = next\_node

        self.head = prev

    # extend() method to extend the linked list with another linked list

    def extend(self, other):

        self.tail.next = other.head

        self.tail = other.tail

    # clear() method to clear the linked list

    def clear(self):

        self.head = None

        self.tail = None

        self.data = None

        self.next = None

**html\_search\_regex\_ver.py**

import re

import timeit

def check\_html(file\_path):

    with open(file\_path, 'r') as f:

        content = f.read()

    stack = []

    tags = re.findall(r'<(\w+)|<\/(\w+)>', content)

    for opening, closing in tags:

        if opening:

            stack.append(opening)

        elif closing:

            if not stack or stack.pop() != closing:

                return "Invalid HTML"

    return "Valid HTML" if not stack else "Invalid HTML"

def benchmark():

    test\_file = "C://Users/<Username>/Path/To/index.html"  # Replace with the path to your test HTML file

    timings = []

    for \_ in range(5):  # Run 5 times

        start\_time = timeit.default\_timer()

        vaild\_check = check\_html(test\_file)

        end\_time = timeit.default\_timer()

        timings.append(end\_time - start\_time)

    for i, timing in enumerate(timings, 1):

        print(f"Run {i}: {timing:.6f} seconds | Result: {vaild\_check}")

    average\_time = sum(timings) / len(timings)

    print(f"\nAverage Time: {average\_time:.6f} seconds")

if \_\_name\_\_ == "\_\_main\_\_":

    benchmark()

**html\_search\_stack\_ver.py**

import timeit

def check\_html\_tags(file\_path):

    with open(file\_path, 'r') as f:

        content = f.read()

    stack = []

    i = 0

    while i < len(content):

        if content[i:i+2] == '</':

            j = content.find('>', i)

            if j == -1 or stack.pop() != content[i+2:j]:

                return "Invalid HTML"

            i = j + 1

        elif content[i] == '<':

            j = content.find('>', i)

            if j == -1:

                return "Invalid HTML"

            stack.append(content[i+1:j])

            i = j + 1

        else:

            i += 1

    return "Valid HTML" if not stack else "Invalid HTML"

def benchmark():

    test\_file = "C://Users/Patrick/Desktop/School/COP4533-Algorithmic-Design-and-Development/Assignment\_2\_Algorithms/index.html"  # Replace with the path to your test HTML file

    timings = []

    for \_ in range(5):  # Run 5 times

        start\_time = timeit.default\_timer()

        vaild\_check = check\_html\_tags(test\_file)

        end\_time = timeit.default\_timer()

        timings.append(end\_time - start\_time)

    for i, timing in enumerate(timings, 1):

        print(f"Run {i}: {timing:.6f} seconds | Result: {vaild\_check}")

    average\_time = sum(timings) / len(timings)

    print(f"\nAverage Time: {average\_time:.6f} seconds")

if \_\_name\_\_ == "\_\_main\_\_":

    benchmark()

**recursion\_max\_sorting.py**

**def find\_max(lst):**

**if len(lst) == 0:**

**raise ValueError("Empty list")**

**if len(lst) == 1:**

**return lst[0]**

**else:**

**lst.sort()**

**return lst[-1]**

**def main():**

**test\_cases = [**

**[1, 2, 3, 4, 5],**

**[-5, -2, -1, -3],**

**[10, 5, 15, 20, 25],**

**[42],**

**[]**

**]**

**for case in test\_cases:**

**try:**

**max\_val = find\_max(case)**

**print(f"List: {case}, Maximum: {max\_val}")**

**except ValueError:**

**print(f"List: {case}, Error: Empty list")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**recursion\_max\_unsorted\_cases.py**

def find\_max(lst):

    if len(lst) == 0:

        raise ValueError("Empty list")

    if len(lst) == 1:

        return lst[0]

    else:

        # find the maximum of the rest of the list using a recursive call and max() function

        return max(lst[0], find\_max(lst[1:]))

def main():

    test\_cases = [

        [1, 2, 3, 4, 5],

        [-5, -2, -1, -3],

        [10, 5, 15, 20, 25],

        [42],

        []

    ]

    for case in test\_cases:

        try:

            max\_val = find\_max(case)

            print(f"List: {case}, Maximum: {max\_val}")

        except ValueError:

            print(f"List: {case}, Error: Empty list")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**test\_html\_search.py**

**import os**

**import timeit**

**import unittest**

**from html\_search\_regex\_ver import check\_html**

**from html\_search\_stack\_ver import check\_html\_tags**

**class TestHtmlSearch(unittest.TestCase):**

**def setUp(self):**

**self.test\_file = os.path.join(os.path.dirname(\_\_file\_\_), "index.html")**

**def test\_html\_search\_with\_regex(self):**

**timings = []**

**for \_ in range(5):  # Run 5 times**

**start\_time = timeit.default\_timer()**

**valid\_check = check\_html(self.test\_file)**

**end\_time = timeit.default\_timer()**

**timings.append(end\_time - start\_time)**

**for i, timing in enumerate(timings, 1):**

**print(f"Run {i}: {timing:.6f} seconds | Result: {valid\_check}")**

**average\_time = sum(timings) / len(timings)**

**print(f"\nAverage Time: {average\_time:.6f} seconds")**

**def test\_html\_search\_with\_stack(self):**

**timings = []**

**for \_ in range(5):  # Run 5 times**

**start\_time = timeit.default\_timer()**

**valid\_check = check\_html\_tags(self.test\_file)**

**end\_time = timeit.default\_timer()**

**timings.append(end\_time - start\_time)**

**for i, timing in enumerate(timings, 1):**

**print(f"Run {i}: {timing:.6f} seconds | Result: {valid\_check}")**

**average\_time = sum(timings) / len(timings)**

**print(f"\nAverage Time: {average\_time:.6f} seconds")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**unittest.main()**

**test\_linked\_list.py**

**import unittest**

**import timeit**

**from custom\_linkedlist import LinkedList  # Adjusted import statement**

**class TestLinkedList(unittest.TestCase):**

**ll\_timings = {}**

**def setUp(self):**

**self.ll = LinkedList()**

**self.ll.append(1)**

**self.ll.append(2)**

**self.ll.append(3)**

**def benchmark(self, func, number=1000):**

**timings = []**

**for \_ in range(5):  # Run 5 times**

**self.setUp()**

**time\_taken = timeit.timeit(func, globals=globals(), number=number)**

**timings.append(time\_taken)**

**average\_time = sum(timings) / len(timings)**

**return timings, average\_time**

**def test\_ll\_append(self):**

**timings, average\_time = self.benchmark(lambda: self.ll.append(4))**

**self.ll\_timings['test\_ll\_append'] = (timings, average\_time)**

**self.ll.append(4)**

**self.assertEqual(self.ll.tail.data, 4)**

**def test\_ll\_getitem(self):**

**timings, average\_time = self.benchmark(lambda: self.ll[0])**

**self.ll\_timings['test\_ll\_getitem'] = (timings, average\_time)**

**self.assertEqual(self.ll[0], 1)**

**self.assertEqual(self.ll[1], 2)**

**self.assertEqual(self.ll[2], 3)**

**def test\_ll\_setitem(self):**

**timings, average\_time = self.benchmark(lambda: setattr(self.ll, '1', 5))**

**self.ll\_timings['test\_ll\_setitem'] = (timings, average\_time)**

**self.ll[1] = 5**

**self.assertEqual(self.ll[1], 5)**

**def test\_ll\_delitem(self):**

**def del\_item():**

**print(len(self.ll))**

**del self.ll[1]**

**timings, average\_time = self.benchmark(del\_item, number=1)**

**self.ll\_timings['test\_ll\_delitem'] = (timings, average\_time)**

**self.assertEqual(len(self.ll), 3)**

**def test\_ll\_iter(self):**

**timings, average\_time = self.benchmark(lambda: [item for item in self.ll])**

**self.ll\_timings['test\_ll\_iter'] = (timings, average\_time)**

**items = [item for item in self.ll]**

**self.assertEqual(items, [1, 2, 3])**

**def test\_ll\_contains(self):**

**timings, average\_time = self.benchmark(lambda: 1 in self.ll)**

**self.ll\_timings['test\_ll\_contains (1 in ll)'] = (timings, average\_time)**

**self.assertTrue(1 in self.ll)**

**timings, average\_time = self.benchmark(lambda: 4 in self.ll)**

**self.ll\_timings['test\_ll\_contains (4 not in ll)'] = (timings, average\_time)**

**self.assertFalse(4 in self.ll)**

**def test\_ll\_pop(self):**

**for i in range(1000):**

**self.ll.append(i)**

**timings, average\_time = self.benchmark(lambda: self.ll.pop(), 1)**

**self.ll\_timings['test\_ll\_pop'] = (timings, average\_time)**

**self.ll.pop()**

**self.assertEqual(self.ll.tail.data, 1)**

**def test\_ll\_pop\_at\_index(self):**

**timings, average\_time = self.benchmark(lambda: self.ll.pop\_at\_index(1))**

**self.ll\_timings['test\_ll\_pop\_at\_index'] = (timings, average\_time)**

**self.ll.pop\_at\_index(1)**

**self.assertEqual(self.ll[1], 2)**

**def test\_ll\_insert(self):**

**timings, average\_time = self.benchmark(lambda: self.ll.insert(1, 4))**

**self.ll\_timings['test\_ll\_insert'] = (timings, average\_time)**

**self.ll.insert(1, 4)**

**self.assertEqual(self.ll[1], 4)**

**def test\_ll\_reverse(self):**

**timings, average\_time = self.benchmark(lambda: self.ll.reverse())**

**self.ll\_timings['test\_ll\_reverse'] = (timings, average\_time)**

**self.ll.reverse()**

**self.assertEqual(self.ll[0], 3)**

**self.assertEqual(self.ll[1], 2)**

**self.assertEqual(self.ll[2], 1)**

**def test\_ll\_extend(self):**

**other = LinkedList()**

**other.append(4)**

**other.append(5)**

**timings, average\_time = self.benchmark(lambda: self.ll.extend(other))**

**self.ll\_timings['test\_ll\_extend'] = (timings, average\_time)**

**self.ll.extend(other)**

**self.assertEqual(self.ll[3], 4)**

**self.assertEqual(self.ll[4], 5)**

**def test\_ll\_clear(self):**

**timings, average\_time = self.benchmark(lambda: self.ll.clear())**

**self.ll\_timings['test\_ll\_clear'] = (timings, average\_time)**

**self.ll.clear()**

**self.assertIsNone(self.ll.head)**

**self.assertIsNone(self.ll.tail)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**unittest.main()**

**test\_lists\_combined.py**

**from test\_linked\_lists import TestLinkedList**

**from test\_python\_lists import TestPythonListFunctions**

**def print\_timings():**

**python\_list = TestPythonListFunctions()**

**python\_list.setUp()**

**# Run all python\_list tests**

**python\_list.test\_append()**

**python\_list.test\_clear()**

**python\_list.test\_count()**

**python\_list.test\_index()**

**python\_list.test\_pop()**

**python\_list.test\_remove()**

**python\_list.test\_sort()**

**python\_list.test\_extend()**

**python\_list.test\_insert()**

**python\_list.test\_reverse()**

**python\_list.test\_copy()**

**python\_list.test\_extend()**

**linked\_list = TestLinkedList()**

**linked\_list.setUp()**

**linked\_list.test\_ll\_append()**

**print(linked\_list.ll\_timings.items())**

**linked\_list.test\_ll\_getitem()**

**print(linked\_list.ll\_timings.items())**

**linked\_list.test\_ll\_setitem()**

**print(linked\_list.ll\_timings.items())**

**linked\_list.test\_ll\_delitem()**

**linked\_list.test\_ll\_iter()**

**linked\_list.test\_ll\_contains()**

**linked\_list.test\_ll\_pop()**

**linked\_list.test\_ll\_pop\_at\_index()**

**linked\_list.test\_ll\_insert()**

**linked\_list.test\_ll\_reverse()**

**linked\_list.test\_ll\_extend()**

**linked\_list.test\_ll\_clear()**

**print("\nTimings:")**

**print(f"{'Test':<30} {'Python List Timings':<50} {'Linked List Timings':<30}")**

**print("="\*110)**

**all\_keys = set(python\_list.py\_list\_timings.keys()).union(set(linked\_list.ll\_timings.keys()))**

**total\_python\_time = 0**

**total\_linked\_time = 0**

**count\_python = 0**

**count\_linked = 0**

**for test\_name in all\_keys:**

**python\_time = python\_list.py\_list\_timings.get(test\_name, (None, 0))[1]**

**linked\_time = linked\_list.ll\_timings.get(test\_name, (None, 0))[1]**

**if python\_time:**

**total\_python\_time += python\_time**

**count\_python += 1**

**if linked\_time:**

**total\_linked\_time += linked\_time**

**count\_linked += 1**

**print(f"{test\_name:<30} {python\_time:.10f} seconds{'':<30} {linked\_time:.10f} seconds")**

**avg\_python\_time = total\_python\_time / count\_python if count\_python else 0**

**avg\_linked\_time = total\_linked\_time / count\_linked if count\_linked else 0**

**print("="\*110)**

**print(f"{'Average':<30} {avg\_python\_time:.10f} seconds{'':<30} {avg\_linked\_time:.10f} seconds")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**print\_timings()**

**test\_python\_lists.py**

**import unittest**

**import timeit**

**# Test all functions of a python list**

**class TestPythonListFunctions(unittest.TestCase):**

**py\_list\_timings = {}**

**def setUp(self):**

**self.test\_list = [1, 2, 3, 4, 5]**

**def benchmark(self, func, number=1000):**

**timings = []**

**for \_ in range(5):  # Run 5 times**

**self.setUp()**

**time\_taken = timeit.timeit(func, globals=globals(), number=number)**

**timings.append(time\_taken)**

**average\_time = sum(timings) / len(timings)**

**return timings, average\_time**

**def test\_append(self):**

**timings, average\_time = self.benchmark(lambda: self.test\_list.append(6), number=995)**

**self.py\_list\_timings['test\_append'] = (timings, average\_time)**

**self.assertEqual(len(self.test\_list), 1000)**

**def test\_extend(self):**

**timings, average\_time = self.benchmark(lambda: self.test\_list.extend([6, 7]), number=1)**

**self.py\_list\_timings['test\_extend'] = (timings, average\_time)**

**self.assertEqual(self.test\_list, [1, 2, 3, 4, 5, 6, 7])**

**def test\_insert(self):**

**timings, average\_time = self.benchmark(lambda: self.test\_list.insert(2, 99), number=1)**

**self.py\_list\_timings['test\_insert'] = (timings, average\_time)**

**self.assertEqual(self.test\_list, [1, 2, 99, 3, 4, 5])**

**def test\_remove(self):**

**timings, average\_time = self.benchmark(lambda: self.test\_list.remove(3), number=1)**

**self.py\_list\_timings['test\_remove'] = (timings, average\_time)**

**self.assertEqual(self.test\_list, [1, 2, 4, 5])**

**def test\_pop(self):**

**popped = self.test\_list.pop()**

**timings, average\_time = self.benchmark(lambda: self.test\_list.pop(), number=1)**

**self.py\_list\_timings['test\_pop'] = (timings, average\_time)**

**self.assertEqual(popped, 5)**

**self.assertEqual(self.test\_list, [1, 2, 3, 4])**

**def test\_clear(self):**

**timings, average\_time = self.benchmark(lambda: self.test\_list.clear(), number=1)**

**self.py\_list\_timings['test\_clear'] = (timings, average\_time)**

**self.assertEqual(self.test\_list, [])**

**def test\_index(self):**

**timings, average\_time = self.benchmark(lambda: self.test\_list.index(3), number=1)**

**self.py\_list\_timings['test\_index'] = (timings, average\_time)**

**index = self.test\_list.index(3)**

**self.assertEqual(index, 2)**

**def test\_count(self):**

**timings, average\_time = self.benchmark(lambda: self.test\_list.count(3), number=1)**

**self.py\_list\_timings['test\_count'] = (timings, average\_time)**

**count = self.test\_list.count(3)**

**self.assertEqual(count, 1)**

**def test\_sort(self):**

**unsorted\_list = [3, 1, 4, 2, 5]**

**timings, average\_time = self.benchmark(lambda: unsorted\_list.sort(), number=1)**

**self.py\_list\_timings['test\_sort'] = (timings, average\_time)**

**self.assertEqual(unsorted\_list, [1, 2, 3, 4, 5])**

**def test\_reverse(self):**

**timings, average\_time = self.benchmark(lambda: self.test\_list.reverse(), number=1)**

**self.py\_list\_timings['test\_reverse'] = (timings, average\_time)**

**self.assertEqual(self.test\_list, [5, 4, 3, 2, 1])**

**def test\_copy(self):**

**timings, average\_time = self.benchmark(lambda: self.test\_list.copy(), number=1)**

**self.py\_list\_timings['test\_copy'] = (timings, average\_time)**

**copied\_list = self.test\_list.copy()**

**self.assertEqual(copied\_list, [1, 2, 3, 4, 5])**

**self.assertIsNot(copied\_list, self.test\_list)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**unittest.main()**

**test\_recursion\_max.py**

**import unittest**

**import timeit**

**from recursion\_max\_sorting import find\_max as find\_max\_sorting**

**from recursion\_max\_unsorted\_cases import find\_max as find\_max\_unsorted\_cases**

**class TestFinxMax(unittest.TestCase):**

**def test\_find\_max\_sorting(self):**

**test\_cases = [**

**[1, 2, 3, 4, 5],**

**[-5, -2, -1, -3],**

**[10, 5, 15, 20, 25],**

**[42],**

**[]**

**]**

**timings = []**

**for \_ in range(5):  # Run 5 times for each test case**

**start\_time = timeit.default\_timer()**

**for case in test\_cases:**

**try:**

**find\_max\_sorting(case)**

**except ValueError:**

**pass**

**end\_time = timeit.default\_timer()**

**timings.append(end\_time - start\_time)**

**print("Find Max Sorting")**

**for i, timing in enumerate(timings, 1):**

**print(f"Run {i}: {timing:.6f} seconds")**

**average\_time = sum(timings) / len(timings)**

**print(f"\nAverage Time: {average\_time:.6f} seconds")**

**def test\_find\_max\_unsorted\_cases(self):**

**test\_cases = [**

**[1, 2, 3, 4, 5],**

**[-5, -2, -1, -3],**

**[10, 5, 15, 20, 25],**

**[42],**

**[]**

**]**

**timings = []**

**for \_ in range(5):  # Run 5 times for each test case**

**start\_time = timeit.default\_timer()**

**for case in test\_cases:**

**try:**

**find\_max\_unsorted\_cases(case)**

**except ValueError:**

**pass**

**end\_time = timeit.default\_timer()**

**timings.append(end\_time - start\_time)**

**print("Find Max Unsorted Cases")**

**for i, timing in enumerate(timings, 1):**

**print(f"Run {i}: {timing:.6f} seconds")**

**average\_time = sum(timings) / len(timings)**

**print(f"\nAverage Time: {average\_time:.6f} seconds")**