HOMEWORK 1

SUBMITTED BY

Net ID: sm11326
NYU ID: N19867031
Student Name: Sai Nishanth Mettu.

Net ID: pt2427
NYU ID: N11088414
Student Name: Praneeth Kumar Thummalapalli.

NetID: mj3184
NYU ID: N19881140
Student Name: Mohammed Adnan Jakati,

Question 1) Answer with True or False

(a) A function f(n) must be in O(g(n)) if it is in o(g(n)).
Answer:- True. For o(g(n)), for any constant c>0, there exists an n_0 such that $f(n) < c * g(n)$ for all $n >= n_0$. For O(g(n)), there exists constants c and n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$. O(g(n)) is the upper bound and o(g(n)) is the strict upper bound, therefore the resultant intersection of O(g(n)) and o(g(n)) will be within O(g(n)).

(b) A function f(n) must be in Ω(g(n)) if it is in ω(g(n)).
Answer:- True. in the case of Ω(g(n)), for any positive constant c>0, there exists a constant n_0 such that $f(n) >= c*g(n)$ for all $n \geq n_0$. Whereas in the case of ω(g(n)), for any constant $c > 0$, there exists an n_0 such that $f(n) > c * g(n)$ for all $n \geq n_0$. Therefore, the intersection of the asymptotic notations would satisfy the given statement.

Question 2) For the following questions, give an example or state such a function does not exist.

(a) A function that is both in O(g(n)) and in ω(g(n)).

Answer:- Such a function does not exist. In case of O(g(n)), there exists constants c and n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$. In the case of ω(g(n)), for any constant $c > 0$, there exists an n_0 such that $f(n) > c * g(n)$ for all $n \geq n_0$.

These notations have conflicting requirements, a function cannot be upper-bounded and also simultaneously be strictly faster-growing than another function.

(b) A function that is both in o(g(n)) and in Ω(g(n)).

Answer:- Such a function does not exist. In the case of o(g(n)), for any positive constant $c > 0$, there exists a constant n_0 such that $f(n) < c * g(n)$ for all $n \geq n_0$. Whereas in the case of Ω(g(n)), for any positive constant $c>0$, there exists a constant n_0 such that $f(n) >= c*g(n)$ for all $n \geq n_0$.

These two conditions aren't possible at the same time, hence such function does not exist.

(c) A function can be in Θ(g(n)) if it is both in O(g(n)) and in o(g(n)).

Answer :- Such a function does not exist. O(g(n)) represents an upper bound on a function. If a function f(n) is in O(g(n)), it means that f(n) grows at most as fast as g(n) asymptotically, i.e. $f(n) <= c*g(n)$. o(g(n)) represents a strict upper bound, $f(n) < c*g(n)$. Whereas, Θ(g(n)) represents a tight bound on a function. $0<c1*g(n)<=f(n)<=c2*g(n)$.

For a function to be in Θ(g(n)), it needs to satisfy both the upper bound O(g(n)) and lower bound (Ω(g(n))) conditions, instead of O(g(n)) and o(g(n)).

"Can Be" But does not necessarily imply (condition):-

If we assume $f(n) = (1/2) * n^2$ and $g(n) = n^2$ as the functions,

f(n) is in O(g(n)):

- We can choose a constant $c = 1$, and for all $n \geq 1$, we have: $f(n) = (1/2) * n^2 \leq c * g(n) = 1 * n^2$. Therefore, $(1/2) * n^2 <= n^2$.

f(n) is in o(g(n)):

- For any positive constant $c > 0$, let's say $c=1$, we can choose $n_0 = 1$, and for all $n \geq 1$, we have: $f(n) = (1/2) * n^2 < 1 * g(n) = n^2$, Therefore $(1/2)*n^2 < n^2$.

f(n) is in Θ(g(n)):

- We can choose constants $c_1 = 1/4$ and $c_2 = 1$. For all $n \geq 1$, we have: $(1/4) * n^2 \leq (1/2) * n^2 <= n^2$.

Here we can see, the function "can be" for some arbitrary constants, but it doesn't necessarily satisfy the definition.

(d) A function can be in Θ(g(n)) if it is in ω(g(n)).

Answer:- Such a function doesn't exist. If it is in ω(g(n)), it means For every positive constant $c > 0$, there exists a positive integer n_0 such that for all $n \geq n_0$, $f(n) > c * g(n)$. It grows strictly faster than any constant multiple of g(n). If it is in Θ(g(n)) There exist positive constants c_1 and c_2 such that for all $n \geq n_0$, $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

These two conditions are contradictory because Θ(g(n)) implies a consistent growth rate with g(n), while ω(g(n)) implies a strictly faster growth rate.

To summarize, we can say a function cannot "necessarily" be in Θ(g(n)) if it is in ω(g(n)) for the same function g(n).

But for some arbitrary constants and functions we can prove that it satisfies the formulas.

Question 3) Give two asymptotically different functions that are both in O(n) and in ω(log n).

Answer:-

Function 1:- $f(n) = n^{1/2}$

- f(n) is in O(n) as for any c > 0, we can set $n_0 = 1$, and for all $n \geq n_0$, we have: $f(n) = n^{1/2} \leq c * n$, where g(n) =n. Put c=1, we get $n^{1/2} <= n$.
- This function is also in ω(log n) Taking c = 1 and $n_0 = 1$, For all $n \geq 1$, $f(n) = n^{1/2} > 1 * \log(n)$.

Function 2:- $f(n) = n^{1/3}$

- f(n) is in O(n) because for any positive constant c > 0, we can set $n_0 = c$, and for all $n \geq n_0$, we have: $f(n) = n^{1/3} \leq c * n$, where g(n) = n. Let c=1, we get $n^{1/2} <= n$.
- This function is in ω(log n) because for you can choose c=1, $n_1 = 1$, and for all $n \geq n_1$, we have: $g(n) = n^{1/3} > c * \log(n)$.

Both have different asymptotic growth rates, and grow faster than the logarithmic growth rates,
The growth rate of log(n) is much slower than linear, quadratic, or exponential growth.

Question 4) Consider the function f that assigns to any natural number n > 1, the largest prime number that divides n. For example, f(10) = 5, f(17) = 17, f(18) = 3.

(a) Compute f(100), f(74), f(201010) by hand.

(b) Consider the following classes of asymptotic growth: O(n), ω(n), Ω(n), o(n), Θ(n), o(log n), O(log n), Θ(log n), ω(log n), Ω(log n), O(n 2), o(n 2), Ω(n 2), Θ(n 2), ω(n 2), O(1), o(1), ω(1), Ω(1), Θ(1).

Which of them does f belong to?

Answer a)

Question 4)

a) $f(100) = 5$

→ Prime factorization of 100 = $2 \times 2 \times 5 \times 5$

$\Rightarrow 2^2 \times 5^2$

∴ Largest prime factor which divides 100 is 5.

Ans = 5

b) $f(74) = 37$.

→ Prime factorization of 74 is ⇒ $2 * 37$

∴ Largest prime factor which divides 74 is 37.

Ans = 37

c) $f(20^{10^{10}}) = 5$

We can rewrite → $20 \rightarrow (2^2 \times 5)$

→ 10^{10} as 10000000000

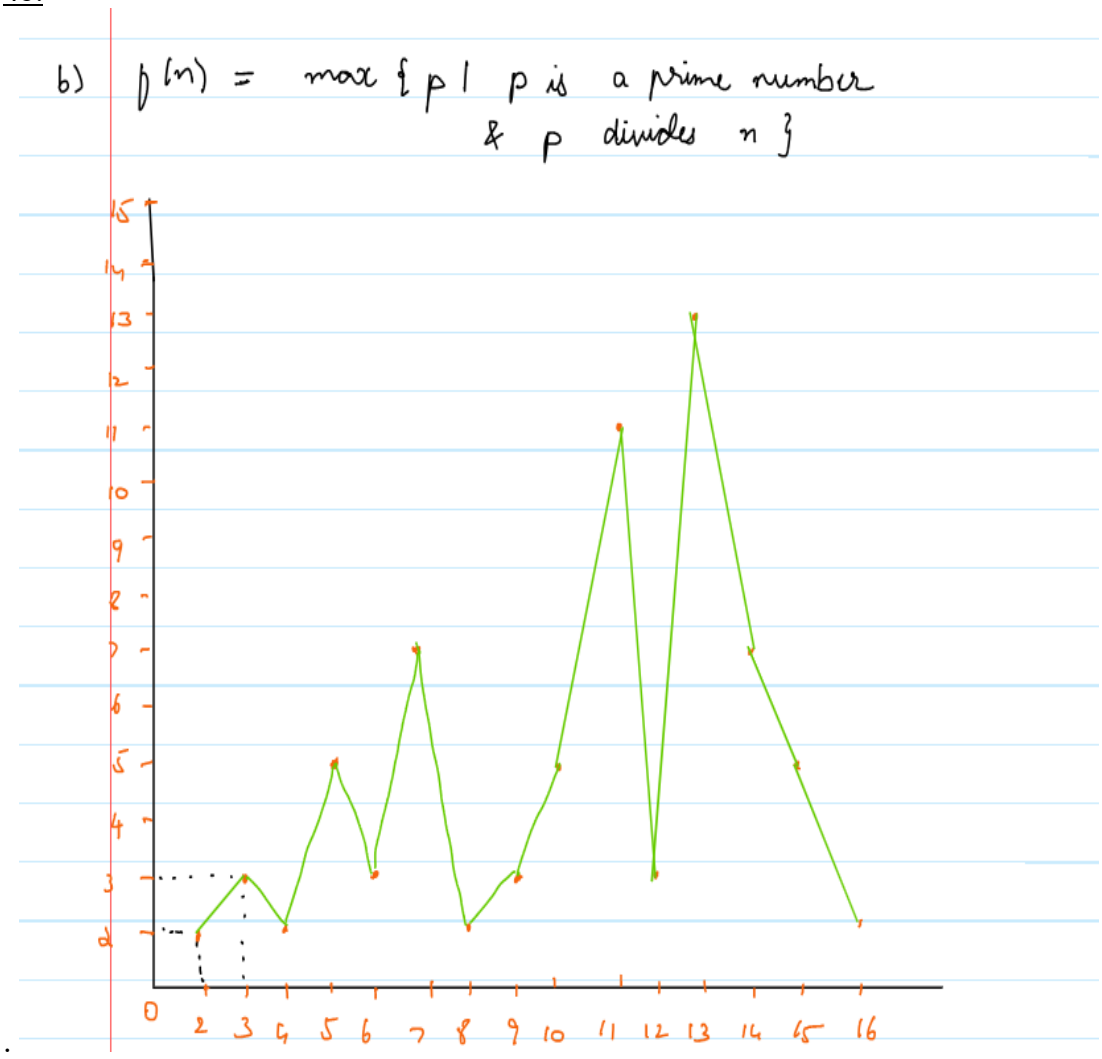
∴ The resultant Expression is $2^{20000000000} \times 5^{10000000000}$

Here we can see the highest "prime factor" would be "5" which would completely divide the function.

Ans = 5

Answer 4b)

4b.



.

1. O(n): Yes, because the largest prime divisor of n cannot be larger than n itself.

2. ω(n): No, as f(n) will never grow faster than linearly with n. It will not be strictly greater than any constant times n.

3. Ω(n): No, because f(n) does not have a constant lower bound that grows with n.

4. o(n): No, the f(n) can be n itself, cannot be strictly less than n.

5. Θ(n): No, because f(n) does not have a constant upper and lower bound that grows linearly with n.

6. o(log n): No, because the growth of f(n) is slower than logarithmic.

7. O(log n): No, the f(n) can be n itself, cannot be strictly less than n.

8. Θ(log n): No, because f(n) does not have a constant upper and lower bound that grows logarithmically with n.

9. ω(log n): No, because f(n) will not grow faster than logarithmically with n.

10. Ω(log n): No, because f(n) does not have a constant lower bound that grows logarithmically with n.

11. O(n^2): Yes, because the growth of f(n) is much slower than quadratic.

12. o(n^2): Yes, because the growth of f(n) is much slower than quadratic.

13. Ω(n^2): No, because f(n) is not bounded from below by a constant times n^2.

14. Θ(n^2): No, because f(n) does not have a constant upper and lower bound that grows quadratically with n.

15. ω(n^2): No, because f(n) will not grow faster than quadratic with n.

16. O(1): No, because f(n) is not a constant function.

17. o(1): No, because f(n) is not a constant function.

18. ω(1): No, because f(n) is always greater than 1.

19. Ω(1): Yes, because f(n) is always greater than or equal to 1.

20. Θ(1): No, because f(n) does not have a constant upper and lower bound.

Question 5)

5]

ia] $result = \sum_{i=1}^{N} 10 \cdot 3i$

$$result = 10 \cdot 3 \cdot \frac{N \cdot (N+1)}{2}$$

$$\boxed{result = 15N^2 + 15N}$$

ib] $f(N) = 15N^2 + 15N$

$$10N^2 \leq 15N^2 + 15N \leq 30N^2$$
$$c_1 g(N) \leq f(N) \leq c_2 g(N)$$

$$\therefore \boxed{f(N) \in \theta(N^2)}$$

iia] $\sum_{i=1}^{N} \sum_{j=1}^{i-1} (i+j) = result$

$$result = \sum_{i=1}^{N} \left[\sum_{j=1}^{i-1} i + \sum_{j=1}^{i-1} j \right]$$

$$= \sum_{i=1}^{N} \left[(i-1)i + \frac{(i-1)(i)}{2} \right]$$

$$= \sum_{i=1}^{N} \left[\frac{3}{2} (i^2 - i) \right]$$

$$\text{result} = \frac{3}{2}\left[\sum_{i=1}^{N} i^2 - \sum_{i=1}^{N} i\right]$$

$\Rightarrow$ using Sum of Sq. & Sum of natural n

$$\text{result} = \frac{3}{2}\left[\frac{N(N+1)(2N+1)}{6} - \frac{N(N+1)}{2}\right]$$

$$= \frac{3}{2} N(N+1)\left[\frac{2N+1}{6} - \frac{1}{2}\right]$$

$$= N(N+1)\left[N - \frac{5}{2}\right]$$

$$= N^3 + \frac{3}{2}N^2 - \frac{5}{2}N.$$

iib] $f(N) = N^3 + \frac{3}{2}N^2 - \frac{5}{2}N.$

$f(N) \in \Theta(N)$ iff $\exists\ c_1, c_2, n_0$

such that: $c_1 g(n) \leq f(N) \leq c_2 g(N)\ \forall\ n$

$$\frac{1}{2}n^3 \leq N^3 + \frac{3}{2}N^2 - \frac{5}{2}N \leq 5N^3$$

$\therefore \boxed{f(N) \in \Theta(N^3)}$

iii a] $\text{result} = \sum_{i=1}^{51} \sum_{j=1}^{N} 1.$

$$\text{result} = \sum_{i=1}^{51} N = 51N.$$

$$\boxed{\text{result} = 51 \cdot N}$$

iii b] $f(N) = 51 \cdot N$

$f(N) \in \theta(N)$ iff $\exists\ c_1, c_2, n_0$

such that $c_1 g(N) \leq f(N) \leq c_2 g(N)\ \forall$

$50N \leq 51N \leq 52N \quad \forall N \geq 1$

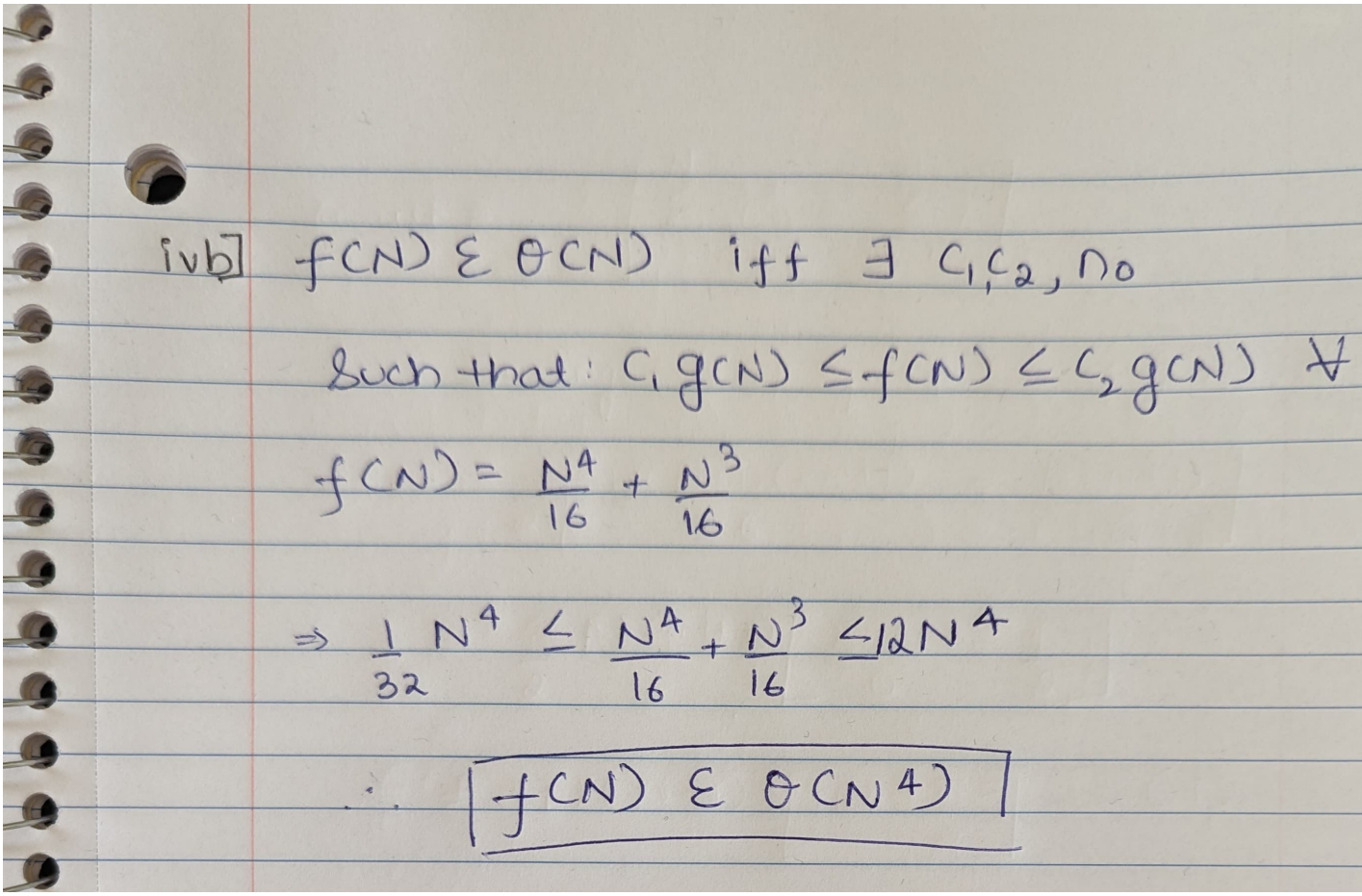
$\therefore \boxed{f(N) \in \theta(N)}$

iv a] $\sum_{i=1}^{N/2} \sum_{j=1}^{N/4} \sum_{K=1}^{N} K = \text{result}$

$$\text{result} = \sum_{i=1}^{N/2} \sum_{j=1}^{N/4} \frac{N(N+1)}{2}.$$

$$\text{result} = \frac{N(N+1)}{2} \cdot \frac{N}{4} \cdot \frac{N}{2}.$$

$$= \frac{N^3(N+1)}{16} = \frac{N^4 + N^3}{16}$$



Question 6)

I. **Brute force Algorithm when given an unsorted array A storing n distinct numbers. For each element A[i] of A, to find the index j ≠ i such that the element A[j] is closest to A[i] in value.**

Answer: -

This is $O(n^2)$

```
1.  n = len(arr)
2. closest = []
3. for i in range(n):
4.     distance = float('inf')
5.     closest_index = -1
6.     for j in range(n):
7.         if i!= j:
8.             abs_difference= abs(arr[i] - arr[j])

9.         if abs_difference < distance:
10.            distance = abs_diff
11.            closest_index = j

12.    closest_indices.append(closest_index)
13. return closest_indices
```

II. If you are allowed to modify or rearrange the data in the arrays, can you think of a way of speeding up your solution? (Recall that one can sort in O(n log n) time, using, for example MergeSort.)

Answer :- Yes we can significantly speed up the algorithm by sorting the array.

```
1.  n = len(arr)
2.  sorted_vals = sorted(range(n), key=lambda i: arr[i])
3.  closest = [0] * n
4.  for i in range(n):
5.      if i == 0:
6.          closest [sorted_vals[i]] = sorted_vals[i + 1]
7.      elif i == n - 1:
8.          closest[sorted_vals[i]] = sorted_vals[i - 1]
9.      else:
            left = sorted_vals[i - 1]
            right = sorted_vals[i + 1]
            current = sorted_vals[i]
            left_diff = abs(arr[left] - arr[current])
            right_diff = abs(arr[right] - arr[current])
10.         if left_diff < right_diff:
                closest [current_index] = left_index
11.         else:
                closest_indices[current_index] = right_index
```

return closest_indices

Here the plan of action is to first sort the indices based on the array values, which will sort the array without modifying its original order. Then, we iterate through the sorted indexes and calculate the closest indices based on neighboring elements in the sorted order.

This is O(n log n) due to sorting.

III. How would your answers in (i) and (ii) change if you replace "closest" by "farthest" in the problem statement?

Answer :- For closest we have to find the smallest absolute difference. For Farthest we have to find the element with Largest Absolute Difference. The overall time complexities and approaches for both brute-force and sorting methods remain the same, i.e. $O(n^2)$ for Brute Force and O(n*logn) for Sorted Arrays but for "farthest" we have to find maximum difference instead of the minimum difference, that's it.

```
1.  n = len(arr)
2. closest = []
3. for i in range(n):
4.     distance = float('inf')
5.     closest_index = -1
6.     for j in range(n):
7.         if i!= j:
8.             abs_difference= abs(arr[i] - arr[j])

9.         if abs_difference > distance:
10.            distance = abs_diff
11.            closest_index = j

12.    closest_indices.append(closest_index)
13. return closest_indices
```

This is $O(n^2)$, here we just change the sign.

```
1.  n = len(arr)
2.  sorted_vals = sorted(range(n), key=lambda i: arr[i])
3.  closest = [0] * n
4.  for i in range(n):
5.  if i == 0:
6.  closest [sorted_vals[i]] = sorted_vals[i + 1]
7.  elif i == n - 1:
8.  closest[sorted_vals[i]] = sorted_vals[i - 1]
9.  else:
        left = sorted_vals[i - 1]
        right = sorted_vals[i + 1]
        current = sorted_vals[i]
        left_diff = abs(arr[left] - arr[current])
        right_diff = abs(arr[right] - arr[current])
10. if left_diff > right_diff:
        closest [current_index] = left_index
11. else:
        closest_indices[current_index] = right_index

        return closest_indices
```

Here, we the time Complexity is O(n*logn), we just change the sign to find the farthest element.

Question 7)

a) Assuming 100 % accuracy of the web service, outline an algorithm to sort pages of a manuscript using are consecutive(.). The algorithm takes the array of the manuscript's image filenames as the input such that each image contains a different page of the manuscript. The algorithm sorts the input array so that i-th element of the output array is the filename of the i-th page of the manuscript.

Answer :- sort (manuscript):

```
page_relationships = {}

    # Iterate through page filenames to build relationships
    for i in range(len(page_filenames)):
        for j in range(i + 1, len(page_filenames)):
            if are_consecutive(page_filenames[i], page_filenames[j]):
                # page_filenames[i] comes before page_filenames[j]
```

```
            if page_filenames[i] not in page_relationships:
                page_relationships[page_filenames[i]] = []
            page_relationships[page_filenames[i]].append(page_filenames[j])


    # Perform DFS to get the correct order
    sorted_pages = []

    for page in page_filenames:
        if page not in page_relationships:
            dfs(page)

    return sorted_pages
```

b) Let's change our assumption in (i) so that are consecutive(.) returns at most one incorrect output per manuscript. Is it possible to design an algorithm for the problem relying solely on the provided web service under our changed assumptions? If it is possible, outline the algorithm. If it is not possible, argue why.

Answer :- We can do this by using Merge Sort with Error Handling to tackle this problem.

Sort_with_errors(manuscript, are_consecutive, subarray_size=10):

➔ *DIVIDE INTO SUB ARRAYS*

1. subarrays = [manuscript [i:i+subarray_size] for i in range(0, len(manuscript), subarray_size)]

➔ *MERGE SORT ALGORITHM ON EACH SUB ARRAY*

merge_sort(subarray):

1. if len(subarray) == 1:
2. return subarray

3. mid_value = len(subarray) // 2
4. left = merge_sort(subarray[:mid-value])
5. right = merge_sort(subarray[mid_value])
6. return merge(left, right)

➔ *MERGE ALGORITHM ON LEFT AND RIGHT SUB ARRAYS*

merge(left, right):

1. merged = []

```
2. left_index, right_index = 0, 0

3. while left_index < len(left) and right_index< len(right):
4. page1, page2 = left[left_index], right[right_index]
5. if are_consecutive (page1, page2):
6.     merged.append(page1)
7.     merged.append(page2)
8.     left_index += 1
9.     right_index += 1
10.else:
11.    merged.append(page1)
12.    right_idx += 1
13.
    merged.extend(left[left_index:])
    merged.extend(right[right_index:])
14. return merged.
```

➔ *SORT EACH SUBARRAY*

```
sorted_subarrays = [merge_sort(subarray) if len(subarray) > 1 else subarray for subarray in subarrays]
```

➔ *MERGE SORT SUBARRAYS USING HEAPS AND ERROR HANDLING*

```
1) sorted_pages_array = []
2) min_heap = []

3) for i in range(len(sorted_subarrays)):
4) if len(sorted_subarrays[i]) > 0:
5) heapq.heappush(min_heap, (sorted_subarrays[i][0], i, 0))

6) while min_heap:
7) page, subarray_index, page_index = heapq.heappop(min_heap)
8) sorted_pages.append(page)

9) if page_idx + 1 < len(sorted_subarrays[subarray_index]):
        next_page = sorted_subarrays[subarray_index][page_index + 1]
        heapq.heappush(min_heap, (next_page, subarray_index, page_index + 1))

10)     return sorted_pages
```

Here we are using Priority Queue/ Min Heap to utilize Error Handling.
If the pages are consecutive, they are added to the **sorted_pages** list. If not, the algorithm advances by adding only **page1** to **sorted_pages** and pushing the pair **(page2)** back into the priority queue to handle potential errors.

c) What is the maximum cost of using the web service, when running your algorithm on a manuscript with n pages provided one call of are consecutive(.) costs $0.01?

In the worst case, each pair of pages in the manuscript needs to be checked to establish their order. This means you would need to make nC2 calls to the **are_consecutive()** function,

So, the maximum cost in dollars for using the web service would be:
Maximum Cost = Number of Calls×Cost per Call
Maximum Cost= (n * (n-1))/2 * 0.01

d) Provide the worst case cost estimate of using the web service, when running your algorithm on 1000 manuscripts with about 100 pages per document = 1,000 manuscripts

Number of Calls per Manuscript= $(100 \cdot (100-1))/2 = 4{,}950$ calls per manuscript

Now, for 1,000 manuscripts:
Total Number of Calls for 1,000 Manuscripts=1,000×Number of Calls per Manuscr ipt=
1,000×4,950=4,950,000 calls
Total Number of Calls for 1,000 Manuscripts=1,000×Number of Calls per Manuscr ipt=
1,000×4,950=4,950,000 calls

Given that each call to the **are_consecutive()** function costs $0.0, the worst-case cost estimate for using the web service when running your algorithm on 1,000 manuscripts, each with approximately 100 pages, would be $49,500.

The End.