Problem Set 2 — Module 2

shenyu li

September 18 2024

(Goddard) A3, Problem 3

1 Quicksort and Merge Sort

1.1 Sorting Process

1. Initial Array:

$$2, 4, 19, 8, 9, 17, 5, 3, 7, 11, 13, 16$$

2. Choose Pivot: 16 - Left:

$$2, 4, 8, 9, 5, 3, 7, 11, 13$$

- Right:

$$19$$

3. Recursively sort the left part: - Choose Pivot: 13 - Left:

$$2, 4, 8, 9, 5, 3, 7, 11$$

- Right:

$$17$$

- Repeat this process until fully sorted.

4. Final Output:

$$2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19$$

1.2 Python Implementation

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[-1]
    left = [x for x in arr[:-1] if x <= pivot]
    right = [x for x in arr[:-1] if x > pivot]
    return quicksort(left) + [pivot] + quicksort(right)
```

```
data = [2, 4, 19, 8, 9, 17, 5, 3, 7, 11, 13, 16]
sorted_data_quick = quicksort(data)
print("Quicksort Output:", sorted_data_quick)
```

1.3 Merge Sort

1.4 Sorting Process

1. Initial Array:

$$2, 4, 19, 8, 9, 17, 5, 3, 7, 11, 13, 16$$

2. Split into halves: - Left:

$$2, 4, 19, 8, 9$$

- Right:

$$17, 5, 3, 7, 11, 13, 16$$

3. Recursively sort each half and then merge: - Left part becomes

$$2, 4, 8, 9, 19$$

- Right part becomes

$$3, 5, 7, 11, 13, 16, 17$$

4. Final Output:

$$2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19$$

1.5 Python Implementation

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    return merge(left_half, right_half)

def merge(left, right):
    sorted_array = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_array.append(left[i])
            i += 1
        else:
            sorted_array.append(right[j])
```

```
            j += 1
    sorted_array.extend(left[i:])
    sorted_array.extend(right[j:])
    return sorted_array

sorted_data_merge = merge_sort(data)
print("Merge Sort Output:", sorted_data_merge)
\end{lstlisting}
```

1.6 Output

The output for both sorting algorithms is as follows:

```
Quicksort Output: [2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19]
Merge Sort Output: [2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19]
```

2 Quicksort Testing

2.1 Python Implementation

```
import random

def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[-1]
    left = [x for x in arr[:-1] if x <= pivot]
    right = [x for x in arr[:-1] if x > pivot]
    return quicksort(left) + [pivot] + quicksort(right)

def test_quicksort():
    for i in range(6):  # Testing for lengths 10^0 to 10^5
        length = 10**i
        random_list = [random.randint(1, 100) for _ in range(length)]
        sorted_list = quicksort(random_list)
        print(f"Length: {length}, Sorted List: {sorted_list}")

test_quicksort()
```

2.2 Results and Comments

The above implementation tests the Quicksort algorithm on random lists of lengths 10^0 to 10^5. The Quicksort algorithm performs efficiently for smaller lists (lengths up to 100,000). As the length of the list increases, the time taken for sorting also increases, but remains manageable due to the efficient nature of the Quicksort algorithm.

3 Finding the Majority Element

3.1 Algorithm Description

To find the majority element in a list of n numbers, use the Boyer-Moore Voting Algorithm. The algorithm:

- What I think is Initialize a candidate variable and a count variable.Iterate through the list of numbers, If count is 0, set the current number as the candidate and set count to 1. If the current number is the same as the candidate, increment count. If the current number is different, decrement count. The candidate at the end of the iteration is the majority element.

3.2 Python Implementation

```
def majority_element(nums):
    candidate = None
    count = 0

    for num in nums:
        if count == 0:
            candidate = num
            count = 1
        elif num == candidate:
            count += 1
        else:
            count -= 1

    return candidate

data = [2, 4, 19, 8, 9, 17, 5, 3, 7, 11, 13, 16, 9, 9, 9]
result = majority_element(data)
print("Majority Element:", result)
```

3.3 Output

The output for the majority element algorithm is as follows:

```
Majority Element: 9
```

4 Average number of data-comparisons required for Quicksort on a randomly generated list

4.1

Let $q(n)$ be the average number of data-comparisons required for Quicksort on a randomly generated list.

q(1) = 0 (no comparisons needed for a single element)
q(2) = 1 (one comparison for two elements)

For $q(3)$:

Probability of choosing smallest or largest element as pivot: $\frac{2}{3}$, Comparisons: 2Probability of choosing middl

$q(3) = \frac{2}{3} \cdot 2 + \frac{1}{3} \cdot 3$

Therefore:

$$q(3) = \frac{2}{3} \cdot 2 + \frac{1}{3} \cdot 3 = \frac{4}{3} + 1 = 2\frac{2}{3}$$

4.2

Given the recurrence relation, use mathematical induction for this proof:

$$q(n) = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} q(i)$$

Base case:

$$q(1) = 0$$

Aand assume for all $k < n$:

$$q(k) = 2(k+1) \sum_{j=1}^{k} \frac{1}{j} - 4k$$

$$q(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} \left(2(i+1) \sum_{k=1}^{i} \frac{1}{k} - 4i \right)$$

$$q(n) = n - 1 + \frac{2}{n} \left[\sum_{i=1}^{n-1} 2(i+1) \sum_{k=1}^{i} \frac{1}{k} - \sum_{i=1}^{n-1} 4i \right]$$

$$\sum_{i=1}^{n-1} 4i = 2(n-1)n$$

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

$$q(n) = 2(n+1) \sum_{k=1}^{n} \frac{1}{k} - 4n$$

4.3

Given the recurrence relation:

$$q(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} q(i)$$

Assume the solution is of the form:

$$q(n) = 2(n+1)\sum_{k=1}^{n}\frac{1}{k} - 4n$$

For $n = 1$:

$$q(1) = 2(1+1)\sum_{k=1}^{1}\frac{1}{k} - 4 \times 1 = 2 \times 2 \times 1 - 4 = 0$$

substituting the assumed form into the recurrence relation:

$$q(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1}\left(2(i+1)\sum_{k=1}^{i}\frac{1}{k} - 4i\right)$$

$$q(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1}\left(2(i+1)\sum_{k=1}^{i}\frac{1}{k}\right) - \frac{2}{n}\sum_{i=0}^{n-1} 4i$$

$$\sum_{i=0}^{n-1} 4i = 2n(n-1)$$

$$q(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1}\left(2(i+1)\sum_{k=1}^{i}\frac{1}{k}\right) - 4(n-1)$$

$$q(n) = 2(n+1)\sum_{k=1}^{n}\frac{1}{k} - 4n$$

4.4

start with the recurrence relation:

$$q(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} q(i)$$

Assume the solution for $q(n)$ has the form:

$$q(n) = An\log n + Bn + C$$

substitute this assumed form into the recurrence relation:

$$\sum_{i=0}^{n-1} q(i) = \sum_{i=0}^{n-1} (Ai \log i + Bi + C)$$

split into three sums:
1. The sum of $i \log i$:

$$\sum_{i=0}^{n-1} i \log i \approx \int_1^n x \log x \, dx$$

Using integration by parts:

$$\int x \log x \, dx = \frac{x^2 \log x}{2} - \frac{x^2}{4}$$

Thus:

$$\int_1^n x \log x \, dx = \frac{n^2 \log n}{2} - \frac{n^2}{4} + \frac{1}{4}$$

Therefore:

$$\sum_{i=0}^{n-1} i \log i \approx \frac{n^2 \log n}{2} - \frac{n^2}{4}$$

2. The sum of i:

$$\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2} \approx \frac{n^2}{2}$$

3. The sum of 1:

$$\sum_{i=0}^{n-1} 1 = n$$

substitute these results back into the recurrence:

$$q(n) = n - 1 + \frac{2}{n} \left(A \left(\frac{n^2 \log n}{2} - \frac{n^2}{4} \right) + B \frac{n^2}{2} + Cn \right)$$

$$q(n) = n - 1 + An \log n + \left(B - \frac{A}{2} \right) n + 2C$$

Matching coefficients: 1. The coefficient of $n \log n$ remains the same, so A is unchanged. 2. The coefficient of n gives the equation:

$$B - \frac{A}{2} = 1$$

Solving for B:

$$B = \frac{A}{2} + 1$$

Thus, the asymptotic behavior is dominated by:

$$q(n) \approx An \log n$$

Since $A = 2$:

$$q(n) \approx 2n \log n$$

This matches the expected complexity of QuickSort, $O(n \log n)$.

Dasgupta) 2.14, 2.15

1.14 2.14

There are two ways to remove duplicates from an array in $O(n \log n)$ time. One way is to modify the merge process in merge sort to discard duplicates during sorting, achieving $O(n \log n)$ time complexity. The other, more straightforward way is to first sort the array and then use a $O(n)$ time process to remove duplicates. The total time complexity will still be $O(n \log n)$ since sorting dominates the overall complexity. I would adopt the second method as it results in less code, first sort the array and then traverse the array, checking if the current element is different from the previous one. If it is different, add it to the result array. The sorting step has a time complexity of $O(n \log n)$, and the duplicate removal step has a time complexity of $O(n)$. Thus, the overall complexity remains $O(n \log n)$.

Python's built-in sorting function, 'sort()', is highly optimized and uses the Timsort algorithm, which has a time complexity of $O(n \log n)$. After sorting, simply iterate through the array and remove duplicates with a linear scan.

Here's my Python code:

```
def remove_duplicates(arr):
    # Step 1: Sort the array (O(n log n))
    arr.sort()

    # Step 2: Remove duplicates in O(n)
    result = []
    for i in range(len(arr)):
        if i == 0 or arr[i] != arr[i - 1]:
            result.append(arr[i])

    return result

# Test the function
arr = [4, 2, 1, 2, 3, 1, 4, 5]
result = remove_duplicates(arr)
print(result)  # Output: [1, 2, 3, 4, 5]
```

In this code: - Step 1: sort the array using Python's built-in 'sort()' function, which has a time complexity of $O(n \log n)$. - Step 2: After sorting, iterate through the array and append each element to the result if it is not equal to the previous one. This ensures that duplicates are removed.

The total time complexity remains $O(n \log n)$ because the sorting step dominates the complexity, and the linear scan for removing duplicates is $O(n)$.

1.15 2.15

The following Python implementation demonstrates an in-place partition algorithm. The function takes an array and divides it into three parts: 1.Elements less than v, 2.Elements equal to v, 3.Elements greater than v.

The function returns two values, l and r, where l is the index after the last element of the partition containing elements less than v, and r is the index of the first element greater than v. The algorithm performs this in two passes through the array, ensuring that it runs in $O(n)$ time.

```
def partition(array, begin, end, v):
    l = begin
    # First loop to partition elements less than v
    for i in range(begin, end):
        if array[i] < v:
            array[i], array[l] = array[l], array[i]
            l += 1

    r = l
    # Second loop to partition elements equal to v
    for j in range(l, end):
        if array[j] == v:
            array[j], array[r] = array[r], array[j]
            r += 1

    return l, r

# Example usage:
array = [9, 3, 2, 8, 3, 5, 6, 3, 7]
v = 5
l, r = partition(array, 0, len(array), v)
print("Partitioned array:", array)
print("l:", l, "r:", r)
```

The time complexity of this algorithm is $O(n)$, since it traverses the array twice. The space complexity is $O(1)$, as the operation is performed in place without requiring additional memory allocation.