1

1. [16 pts] Describe a recursive algorithm that prints the digits of an integer n > 0 vertically on a screen. For example, given the integer n=123, the output displayed on the screen is

```
1
2
3
```

Iterations of any types are not allowed. Recall, when **describing**, you need to answer parts a-d above the “problems” section of this document.

a. Main Concept

The main concept of my algorithm is to recursively extract and display all the digitals of a positive integer 'n'. It breaks down the problem into smaller steps, extracting each digit from right to left adn storing them in reverse order in an array 'A'.

b. Pseudo-Code

Algorithm AllDigital_Display(n, A, d):

```
Input: The integer n > 0, A is a empty integer array, and d is the digit number of n

Output: Print all digitals of n on the screen

if d = 0 then

    length ← len(A)    #### get the array A's length

    for i ← length - 1 to 0 do

        print(A[i])

    return

if d > 0 then

    remainder ← n % 10

    add remainder into the end of array A

    n ← n // 10

    d ← d - 1

    return AllDigital_Display(n, A, d)
```

c. Running the Algorithm

Example 1: AllDigital_Display(n = 123, A = [ ], d = 3)

Let's analyze the execution of the AllDigital_Display algorithm with n = 123, an empty array A, and d = 3:

step 1:

n = 123, A = [], d = 3

Since d is not 0, it calculates remainder = n % 10, which is 3.

It adds remainder to the end of A, so A = [3].

It updates n by performing an integer division by 10, so n = 12.

It decreases d by 1, so d = 2.

The function is recursively called with the updated values: AllDigital_Display(12, [3], 2).

step 2:

remainder = 12 % 10, which is 2.

Add remainder to A, so A = [3, 2].

Update n to 1.

Decrease d to 1.

Recursively call AllDigital_Display(1, [3, 2], 1).

step 3:

remainder = 1 % 10, which is 1.

Add remainder to A, so A = [3, 2, 1].

Update n to 0.

Decrease d to 0.

d = 0

Since d is now 0, it enters the first condition.

It iterates through A in reverse order and prints each digit:

```
1
2
3
```

d. Prove/justify its correctness and its running time

Correctness Proof:

Base Case (d = 0):

When d is initially 0, the algorithm prints the digits stored in array A. Since we've ensured that it correctly stores the digits in reverse order during the recursive calls, this step is correct.

Inductive Step:

Assume that the algorithm correctly prints all the digits of a positive integer n when d is greater than 0, for some arbitrary positive integer d.

Now, consider the case where d is d - 1, and the algorithm processes the last digit of n (the least significant digit).

The algorithm extracts the last digit of n (remainder ← n % 10) and adds it to the end of array A.

This step correctly stores the digit in A.

It then updates n by removing the last digit (n ← n // 10) and decreases d by 1 (d ← d - 1).

The algorithm is then recursively called with the updated values.

By the inductive hypothesis, we assume that the algorithm correctly prints all the remaining digits of n.

After the recursive call returns, the algorithm prints the last digit in A, which is the digit that was added to A earlier.

Since the digits are printed in reverse order, this step effectively prints the last digit of n.

Therefore, the algorithm correctly prints all the digits when d is greater than 0.

Running time:

The number of recursive calls is equal to the number of digits in n. For example, if n has 4 digits, it will make 4 recursive calls, and d is the number of digits in n, recursive time = O(d).

When d = 0, this Algo needs to print d times, it spends O(d).

The time complexity of the algorithm is `O(2d) = O(d)`.

$d = \lceil \log_{10} n \rceil$

$O(\lceil \log_{10} n \rceil) <= O(\log_{10} n + 1)$

Therefore, the running time is $O(\log_{10} n)$

2

2. [16 pts] Suppose that we are given an unsorted array A with n elements and whose entries are integers between 1 and n. Describe a recursive algorithm for finding a repeated element in A (if any). Iterations of any types are not allowed. Recall, when describing, you need to answer parts a-d above the "problems" section of this document.

a. Main Concept

The main concept of the "Repeat_Selection" algorithm is to find and print repeated elements within an unsorted array, A, containing n elements. It uses a divide-and-conquer approach and a pivot selection strategy (Median of Medians or MOM) to efficiently identify and process repeated elements in the array.

b. Pseudo-Code

Algorithm Repeat_Selection(A, n):

```
Input: an unsorted array A with n elements and whose entries are integers between 1 and n.

Output: repeated elements in A.

if n <= 1 then

    return

pivot ← use Median of Medians(MOM) to get Array 'A' Median

remove all the elements from Array 'A' and put them into three sequences:

1. L, storing the elements in 'A' less than pivot

2. E, storing the elements in 'A' equal to pivot

3. G, storing the elements in 'A' greater than pivot

if len(E) > 1 then

   print("Repeated elements:", E[0])

Repeat_Selection(L, len(L))

Repeat_Selection(G, len(G))
```

c. Running the Algorithm

Let's walk through each step of the Repeat_Selection algorithm with the provided input array A = [2, 9, 7, 10, 7, 3, 11, 5, 3, 7, 9, 20, 88, 13] and n = 14. The process is just like binary tree.

Step 1:

Repeat_Selection(A = [2, 9, 7, 10, 7, 3, 11, 5, 3, 7, 9, 20, 88, 13], n = 14)

n = 14 > 1

pivot ← 7

L ← [2, 3, 3, 5]

E ← [7, 7, 7]

G ← [9, 10, 11, 9, 20, 88, 13]

len(E) = 3 > 1

print("Repeated elements:", 7)

Step 1.1:

Repeat_Selection(L = [2, 3, 3, 5], len(L) = 4)

n = len(L) = 4 > 1

pivot ← 3

L ← [2]

E ← [3, 3]

G ← [5]

len(E) = 2 > 1

print("Repeated elements:", 3)

Step 1.1.1:

Repeat_Selection(L = [2], len(L) = 1)

n = len(L) = 1 <= 1

return

Step 1.1.2:

Repeat_Selection(G = [5], len(G) = 1)

n = len(G) = 1 <= 1

return

Step 1.2:

Repeat_Selection(G = [9, 10, 11, 9, 20, 88, 13], len(G) = 7)

n = len(G) = 7 > 1

pivot ← 11

L ← [9, 10, 9]

E ← [11]

G ← [20, 88, 13]

len(E) = 1 <= 1

Step 1.2.1:

Repeat_Selection(L = [9, 10, 9], len(L) = 3)

n = len(G) = 3 > 1

pivot ← 9

L ← []

E ← [9, 9]

G ← [10]

len(E) = 2 > 1

print("Repeated elements:", 9)

Step 1.2.1.1:

Repeat_Selection(L = [], len(L) = 0)

n = len(L) = 0 <= 1

return

Step 1.2.1.2:

Repeat_Selection(G = [10], len(G) = 1)

n = len(G) = 1 <= 1

return

Finally

Algo has printed all repeated elements 7， 3， 9

d. Prove/justify its correctness and its running time

Correctness Proof:

Base Case (n <= 1):

If the input array A has fewer than or equal to 1 element, there are no repeated elements to find, and the algorithm correctly returns without any actions. This satisfies the base case condition.

Inductive Hypothesis:

Assume that the algorithm correctly identifies and prints repeated elements for arrays of size less than or equal to k.

Inductive Step:

We want to show that the algorithm correctly identifies and prints repeated elements for an array of size k + 1.

The algorithm proceeds as follows for an array of size k + 1:

It selects a pivot element using the Median of Medians (MOM) algorithm, which ensures that the pivot is chosen to reasonably balance the partitions.

It partitions the array into three sequences: L, E, and G, where:

- L contains elements less than the pivot,
- E contains elements equal to the pivot, and
- G contains elements greater than the pivot.

If there are repeated elements in the array, they will be present in sequence E (elements equal to the pivot). The algorithm prints these repeated elements correctly by checking if the length of E is greater than 1.

The algorithm then recursively applies itself to sequences L and G. By the inductive hypothesis, we assume that it correctly identifies and prints repeated elements within these sequences.

The algorithm continues to make progress toward the base case by repeatedly partitioning the array into smaller subarrays, ensuring that it explores all elements. Since it correctly identifies and prints repeated elements at each step, it will eventually identify all repeated elements in the original array.

Therefore, by induction, we can conclude that the "Repeat_Selection" algorithm correctly identifies and prints repeated elements in the input array A.

Running time:

The selection of a pivot using the Median of Medians (MOM) algorithm takes `O(n)` time in the worst case.

In each level of recursion, the array is divided in half, and the counting step takes `(n)` time at that level.

At each level, the counting step takes `O(n)` time, and there are `og(n)` levels.

So, the runtime complexity of the algorithm is `O(n * log(n)) + O(n) = O(nlog(n))`

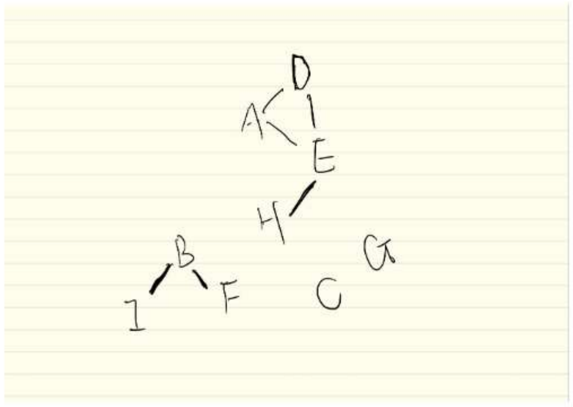
3

3. [8 pts] Suppose we have a social network with members A, B, C, D, E, F, G, H and I and the set of friendship ties

```
{(A, D), (A, E), (D, E), (F, B), (B, I), (E, H)}
```

What are the connected components?

Solution



4

4. [3 pts] Suppose we represent a graph G having n vertices with an adjacency matrix. Why, in this case, would inserting an undirected edge in G run in $O(1)$ time while inserting a new vertex would take $O(n$^2) time?

Solution:

1. Inserting an Undirected Edge (O(1) time):

- To insert an undirected edge between two vertices i and j, we simply need to update the cells (i, j) and (j, i) in the adjacency matrix to indicate the existence of the edge. Since the adjacency matrix is a constant size (n x n), updating two specific cells (i, j) and (j, i) takes constant time, which is O(1).

2. Inserting a New Vertex (O(n^2) time):

- When we insert a new vertex into the graph represented as an adjacency matrix, we need to expand the matrix to accommodate the new vertex. This typically involves creating a new matrix of size (n+1) x (n+1) and copying all the elements from the old matrix to the new one. This copying process takes O(n^2) time because we have to copy n^2 elements from the old matrix to the new one.
- After copying, we can add the edges associated with the new vertex. If we have to establish connections between the new vertex and existing vertices, we may need to update multiple cells, which can be done in O(n) time in the worst case if we need to update all the existing vertices. However, the dominant factor in the time complexity is still the initial copying of the matrix, which is O(n^2).

5

5. [3 pts] One additional feature of the list-based implementation of the union-find structure is that it allows for the contents of any set in a partition to be listed in time proportional to the size of the set. Explain how this can be done.

solution:

The list-based implementation of the union-find data structure, also known as disjoint-set data structure, allows for efficient listing of the contents of any set in a partition in time proportional to the size of the set. This feature is particularly useful for various applications where we need to retrieve or iterate through the elements of specific sets.

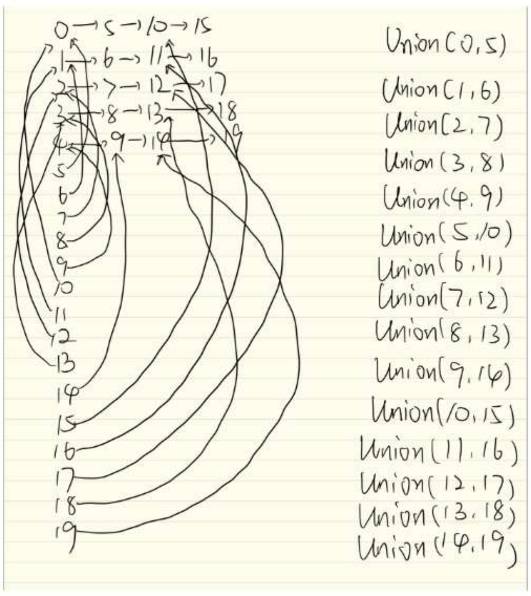
In a list-based implementation of the union-find (disjoint-set) data structure, we can efficiently list the contents of any set in a partition in time proportional to the size of the set. This is achieved by representing each set as a list or array, with each element in the list corresponding to a member of the set. The key steps are:

Maintain representative elements (leaders) for each set. Use the find operation to find the representative element of the desired set efficiently. The amortized time complexity of this operation is nearly constant. Once we have the representative element, access the associated list directly and iterate through it. Listing the contents of the set takes O(k) time, where k is the size of the set. This feature is valuable for various applications where we need to retrieve or iterate through the elements of specific sets, and it ensures efficient access to the elements within a set while maintaining the benefits of the union-find data structure for set union and element connectivity operations.

6

6. [8 pts] Suppose we have 20 singleton sets, numbered 0 through 19, and we call the operation union(find(i), find(i+5)), for i=0,1,2,…,14. Draw a picture of a list-based representation of the sets that result.

Solution:



Results:

0→5→10→15

1→6→11→16

2→7→12→17

3→8→13→18

4→9→14→19

7

7. [6 pts] Consider a method, remove, which removes e from whichever list it belongs to, in a list-based implementation of a union-find structure. Explain how to modify the list-based implementation so that this method runs in O(1) time.

Solution

Make the linked lists into double linked lists by changing their representation. Removals will be possible in O(1) time as a result.

8

8. [6 pts] Let G be a simple connected graph with n vertices and m edges. Explain why O(log m) is O(log n). Note that a simple graph is a graph without parallel edges or self-loops. Problem Hint: Review the definition of Big-Oh and the combinatorial problem of how large m can be.

Solution

Because m <= n*(n-1)/2, O(m) = O(n^2).

O(log m) = O(log n^2) = O(2log n) = O(log n).

9

9. [10 pts] Suppose G is a graph with n vertices and m edges. Explain a way to represent G using O(n+m) space so as to support in O(log n) time an operation that can test, for any two vertices v and w, whether v and w are adjacent. Additionally, the representation of G should support the deletion of an edge in O(log n) time. Explain why your run time for these operations are O(log n).

Solution

I can implement the adjacency list for each vertex using a balanced binary search tree (BBST) to achieve the desired time complexities.

1. Representation:

- Maintain an array of size n, where each element corresponds to a vertex in the graph.
- For each vertex v, store its neighbors in a BBST. The BBST should contain the neighboring vertices sorted by their indices.

2. Testing Adjacency (v, w) in O(log n):

- To test whether two vertices v and w are adjacent, follow these steps:
- Find vertex v in the array in O(1) time.
- Search for vertex w in the BBST of vertex v. This search takes O(log n) time in the worst case.
- If w is found in the BBST, then v and w are adjacent; otherwise, they are not.

3. Deleting an Edge (v, w) in O(log n):

- To delete an edge (v, w), follow these steps:
- Locate vertex v in the array in O(1) time.
- Remove vertex w from the BBST of vertex v. This operation takes O(log n) time because we are deleting from a BBST.
- If needed, perform a symmetric operation to remove vertex v from the BBST of vertex w. This step also takes O(log n) time.

4. Space Complexity:

- The space complexity of this representation is O(n + m), where n is the number of vertices and m is the number of edges. Each vertex requires space for its BBST, and the adjacency lists stored in the BBSTs represent the edges.

10

10. [8 pts] Assume that the Radix-Sorting algorithm (see Canvas, Module 2, M2L3-Bucket-Radix-Sort.pdf, Slide 11 “Radix-Sort”) is revised to sort via the most significant digit (MSD) to the least significant digit (LSD). That is, the for loop is revised to

```
for i ← MSD to LSD
        bucketSort(S, N)
```

Give an example that does not properly sort digits of length 3. To do this, give your initial sequence of integers, then show the sequence after each pass of the for loop.

Solution

Let's consider the initial sequence of integers:

Initial sequence: [32, 15, 1, 201, 43, 9, 1001, 72, 5, 41]

Now, let's go through each pass of the for loop from MSD to LSD:

Pass 1 (Most Significant Digit):

Bucket sort based on the most significant digit (leftmost digit):

[32, 15, 1, 201, 43, 9, 1001, 72, 5, 41]

1, 1001

15

201

32, 43, 41

5

72

9 After the first pass, the sequence becomes: [1, 1001, 15, 201, 32, 43, 41, 5, 72, 9]

Pass 2 (Middle Digit - Middle Significant Digit):

Bucket sort based on the middle digit (the second digit from the left):

[1, 1001, 15, 201, 32, 43, 41, 5, 72, 9]

1, 15, 5, 9

1001

201

32

43

41

72

After the second pass, the sequence remains the same, as all numbers have the same middle digit: [1, 1001, 15, 201, 32, 43, 41, 5, 72, 9]

Pass 3 (Least Significant Digit):

Bucket sort based on the least significant digit (rightmost digit):

[1, 1001, 15, 201, 32, 43, 41, 5, 72, 9]

1, 5, 9, 15

1001

201

32

43

41

72

After the third pass, the sequence remains the same because all numbers have the same least significant digit: [1, 1001, 15, 201, 32, 43, 41, 5, 72, 9]

In conclusion, the radix sort with the MSD-to-LSD approach didn't properly sort the integers in this example because it didn't consider the different lengths of the integers. In this case, the numbers with different lengths remained in the same order as they were initially, resulting in incorrect sorting.

11

11. [16 pts] Consider an array of n distinct comparable elements. Describe a variant of the Selection algorithm to return an array of the first k-th largest elements. Your algorithm must run in O(n) time. Note, if k = 2, then the maximum element and the next to the maximum element will be returned. Recall, when describing, you need to answer parts a-d above the "problems" section of this document.

Solution

a. Explain the main concept of your algorithm

The main concept of the algorithm is to efficiently find and return the first k largest elements from an array of n distinct comparable elements, where k is an integer between 1 and n, while running in O(n) time. The algorithm achieves this by using a modified version of the quick-select algorithm to find the k-th largest element and then selecting the first k elements from the sorted portion of the array.

b. Pseudo-code

Algorithm FindKthLargestElements(arr, k):

```
Input: An array arr of n distinct comparable elements, and an integer k ∈ [1, n].

Output: An array of the first k-th largest elements from arr.

if k <= 0 or k > n:
    return []  # Invalid input

Algorithm partition(arr, low, high):

    pivot ← arr[high]

    i ← low - 1

    for j in range(low, high):
        if arr[j] >= pivot:
            i ← i + 1
            arr[i], arr[j] ← arr[j], arr[i]

    arr[i + 1], arr[high] ← arr[high], arr[i + 1]

    return i + 1

Algorithm quickSelect(arr, low, high, k):

    if low < high:

        pivotIndex ← partition(arr, low, high)

        if pivotIndex = k - 1:

            return  # k-th largest element found

        elif pivotIndex < k - 1:

            quickSelect(arr, pivotIndex + 1, high, k)

        else:

            quickSelect(arr, low, pivotIndex - 1, k)

quickSelect(arr, 0, len(arr) - 1, k)  # Find the k-th largest element

return arr[:k]
```

c. Present an example of running your algorithm

The input array is [7, 10, 3, 2, 14, 1, 9].

We want to find the first 2 largest elements (maximum and next-to-maximum), so k = 2.

We call the FindKthLargestElements algorithm with these inputs.

The algorithm finds and returns the first 2 largest elements from the array, which are [14, 10].

The output of the algorithm is [14, 10], which consists of the maximum element 14 and the next-to-maximum element 10, as expected.

d. Prove/justify its correctness and its running time.

Prove correctness

To prove the correctness of the algorithm for finding the first k largest elements from an array, we need to demonstrate that it produces the correct output for all valid inputs. We'll do this by analyzing the key components of the algorithm:

Valid Input Check: The algorithm begins by checking whether the input values are valid. It verifies that k is within the range [1, n], where n is the size of the input array. This step ensures that the algorithm handles only valid inputs.

Quick-Select Algorithm: The core of the algorithm relies on the quick-select algorithm to efficiently find the k-th largest element in the array. The quick-select algorithm is a well-known and widely accepted method for finding the k-th order statistic in linear time. It works by repeatedly partitioning the array around a pivot element until the k-th largest element is found.

Return First k Largest Elements: After finding the k-th largest element, the algorithm returns the first k elements from the sorted portion of the array. These k elements are guaranteed to be the k largest elements in the original array because of the way the quick-select algorithm partitions the array

Running time

The running time of the algorithm for finding the first k largest elements in an array is O(n), where n is the number of elements in the array. This linear time complexity is achieved by using the quick-select algorithm.

1. Quick-Select Algorithm: The most significant part of the algorithm is the quick-select step, which is used to find the k-th largest element in the array. Quick-select has an average-case time complexity of O(n). In the worst case, it may degrade to O(n^2), but it can be optimized to have an expected O(n) time complexity through techniques like choosing a good pivot.
2. Copying k Elements: After finding the k-th largest element, the algorithm copies the first k elements from the sorted portion of the array. This copy operation has a time complexity of O(k), but since k is typically much smaller than n (e.g., when you want the top k largest elements), it is considered a constant factor and does not affect the overall time complexity.

Therefore, the dominant factor in the running time is the quick-select step, which is O(n) on average. This makes the overall time complexity of the algorithm `O(n)` , making it efficient for finding the first k largest elements in an array of n elements.