Solution Set 3

Solutions to problem set 3

Solution 1

a-

We can implement merge sort by dividing the initial input array into 3 subproblems instead of 2. The only caution we need to exercise is to carefully pass the boundaries for the three parts in the subsequent recursive portions. Combining the three arrays is equivalent of solving the problem of merging n number of sorted arrays, using a priority queue. One way the algorithm can be implemented is shown below.

```
import java.util.Random;
import java.util.PriorityQueue;

class Demonstration {
    public static void main( String args[] ) {
        createTestData();
        long start = System.currentTimeMillis();
        mergeSort(0, input.length - 1, input);
        long end = System.currentTimeMillis();
        System.out.println("Time taken = " + (end - start));
        printArray(input);
    }

    private static int SIZE = 100;
    private static Random random = new Random(System.currentTimeMillis());
    static private int[] input = new int[SIZE];
    static PriorityQueue<Integer> q = new PriorityQueue<>(SIZE);

    private static void printArray(int[] input) {
        System.out.println();
        for (int i = 0; i < input.length; i++)
            System.out.print(" " + input[i] + " ");
        System.out.println();
    }

    private static void createTestData() {

        for (int i = 0; i < SIZE; i++) {
            input[i] = random.nextInt(10000);
        }
    }
```

```
    private static void mergeSort(int start, int end, int[] input) {

        if (start >= end) {

            return;
        } else if (start + 1 == end) {
            if (input[start] > input[end]) {
                int temp = input[start];
                input[start] = input[end];
                input[end] = temp;
            }

            return;
        }

        int oneThird = (end - start) / 3;

        // sort first half
        mergeSort(start, start + oneThird, input);

        // sort second half
        mergeSort(start + oneThird + 1, start + 1 + (2 * oneThird), input);

        // sort third half
        mergeSort(start + 2 + (2 * oneThird), end, input);

        // merge the three sorted arrays using a priority queue
        int k;

        for (k = start; k <= end; k++) {
            q.add(input[k]);
        }

        k = start;
        while (!q.isEmpty()) {
            input[k] = q.poll();
            k++;
        }
    }
}
```

Merge Sort with 3 Subproblems

b-

How many recursion levels will the be generated?

Say we are provided with an array of size n. The question we need to ask ourselves is how many times do we need to successively divide n by 3 to get to 1? If there are 9 elements, we'll divide once to get 3 and then one more time to get 1. This can be expressed as a logarithmic equation as follow:

$$log_3 n = x$$

$$log_3 n = x$$

The number of levels of the recursion tree will be 1 more than $\log_3 n$ so the correct answer would be $\log_3 n + 1$.

c-

The recurrence equation will be given as:

$$T(n) = Cost\ to\ divide\ into\ 3\ subproblems + 3 * T(\frac{n}{3}) + Cost\ to\ merge\ 3\ subproblems$$

We determined in the previous question the number of recursion levels for the 3-way merge sort will be $\boldsymbol{log_3 n+1}$. Next, we need to determine the cost to merge the three subproblems. Unlike in traditional merge sort, the 3-way merge sort uses a priority queue to create a min-heap before attempting a merge of the three subproblems. Insertion into a priority queue takes $\boldsymbol{lgn}$ time and since we insert all the n elements into the queue, the total time to create the heap comes out to be $\boldsymbol{nlgn}$. Finally, the cost to divide the problem into three subproblems is constant and can be represented by $\boldsymbol{d}$. Therefore, the time complexity will be:

$$T(n) = (log_3 n + 1) * (d + nlgn)$$

$$T(n) = dlog_3 n + d + nlgnlog_3 n + nlgn$$

$$T(n) = O(nlgnlog_3 n)$$

Since lgn > $\log_3 n$, we can simplify to:

$$T(n) = O(n(lgn)^2)$$

d-

Had we not used a priority queue while implementing the 3-way merge sort and instead written a merge of three arrays in O(n) then the complexity of the

3-way merge would have come out to be ***O(nlog$_3$n)***. We used priority queue to

simplify the code at the expense of efficiency. In professional settings, it is almost always preferable to write readable and maintainable code than to optimize for running time. It would appear that ***log$_3$n*** is less than ***lgn*** for large values of n, however 3-way merge may not end up being faster than 2-way merge sort for reasons discussed below.

Subdividing a problem into more parts may not necessarily improve runtime of the algorithm. In the case of merge sort if we divide the array into three parts we end up with lesser number of recursion levels since $\log_3 n$ is less than $\log_2 n$. However, the number of comparisons performed in the combine-step increases in case of 3-way merge sort and the running time will come out to be greater than that of traditional merge sort for large inputs.

Similar argument can be made about binary search. Dividing the search space into three parts may not necessarily speed up the running time for ternary search.

Solution 2

The optimized code for printing permutations uses a slightly different loop for the regular case as depicted below.

```
        // regular case
        for (int i = index; i < str.length; i++) {
            swap(str, index, i);
            permutate(str, index + 1);
            swap(str, index, i);
        }
```

In the unoptimized version, we ran the for loop for the entire length of the array. However, in the optimized version, we can see that the loop would run for:

$$n$$

$$n-1$$

$$n-2$$

$$.$$

.

1

The total number of invocations of the permute method would still remain unchanged, only the cost of each permute method would change. If you were to determine the exact expression representing the cost then it would come out to be smaller in value than the exact expression for the unoptimized version. However, the big O will remain unchanged as we already have seen that the series

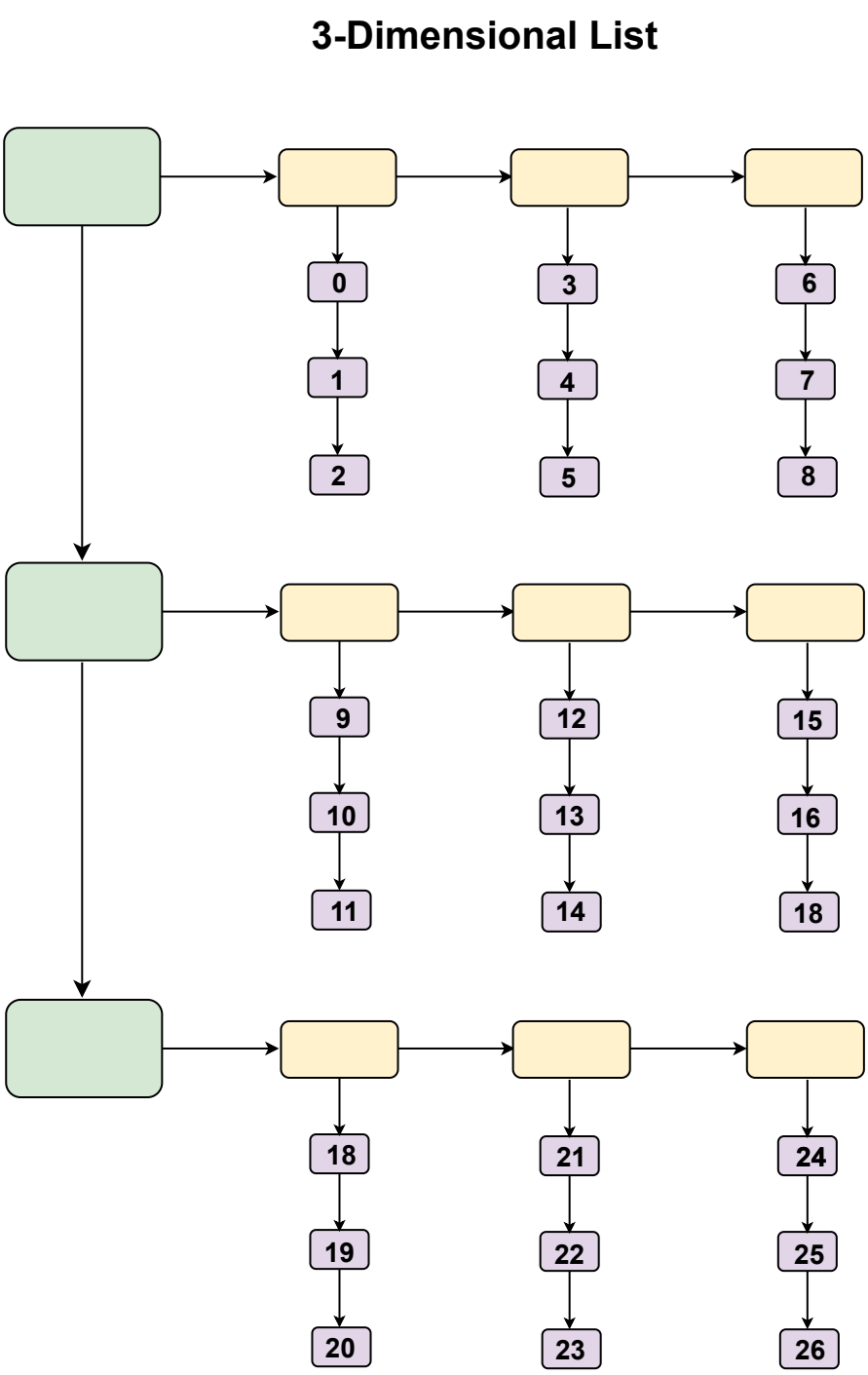
$$1 + 2 + 3...n - 2 + n - 1 + n$$

sums upto $^{n(n+1)}/_{2}$ which is again O(n).

Solution 3

a- The below picture will help you visualize the resulting data-structure when n=3. The problem can be easily solved recursively. From the diagram below, it should be clear that the total number of nodes in the leaf-lists would be:

$$Total\ Leaf\ Nodes = n^n = 3^3 = 27$$



We are essentially creating a tree of Linked Lists with the integers being stored at the leaf nodes. The root is a linked list of three objects. Each object at the root level points to another linked list of three objects. In turn, objects at the second level point to yet another linked list of three objects. The objects at the last level point to a linked list of a single element in which integers are stored.

You can see how we can recursively do the allocation given the value of *n*. The following code is one way to solve the problem. The tricky part to the solution is given an index, we need to find the appropriate node it points to. Say, if we are given index=25, then we'll divide it by *n* for *n* times and at each iteration

note the remainder.

index = 25 and n = 3

25 / 3 = 8 and remainder is 1
8 / 3 = 2 and remainder is 2
2 / 3 = 0 and remainder is 2

You can verify that to get to index 25, we'll start with the third linked list at the root level, then the third linked list at the second level and finally, the second list at the leaf level. Look at the `getIndices` method to see how we compute the right linked lists to go to.

```
import java.util.LinkedList;
import java.util.List;
import java.util.Stack;

class Demonstration {
    public static void main( String args[] ) throws Exception{
        // 1D list
        NDimensionalList list = new NDimensionalList(1);
        list.put(0, -10);
        System.out.println(list.get(0));
        System.out.println();

        // 2D list
        list = new NDimensionalList(2);
        list.put(0, -3);
        System.out.println(list.get(0));
        list.put(3, 5);
        System.out.println(list.get(3));
        System.out.println();

        // 3D list
        list = new NDimensionalList(3);
        list.put(25, -100);
        System.out.println(list.get(25));
        list.put(0, 21);
        System.out.println(list.get(0));
        list.put(26, 211);
        System.out.println(list.get(26));
        System.out.println();

        // 5D list
        list = new NDimensionalList(5);
        list.put(124, 313);
        System.out.println(list.get(124));
        System.out.println();

    }
}

class NDimensionalList {
```

```
int n;
int maxIndex;

List<Object> superList;

public NDimensionalList(int n) {
    if (n <= 0)
        throw new IllegalArgumentException();

    this.n = n;
    this.maxIndex = (int) Math.pow(n, n);
    allocate(n, superList);
}

// Recursive method that initializes the resulting N-Dimensional List
private void allocate(int rem, List<Object> list) {

    if (rem == -1)
        return;

    if (superList == null) {
        superList = list = new LinkedList<Object>();
        allocate(rem - 1, list);
    } else {
        for (int i = 0; i < n; i++) {
            List<Object> subList = new LinkedList<Object>();
            list.add(i, subList);
            allocate(rem - 1, subList);
        }
    }
}

// Calculates the index to the right node
private Stack<Integer> getIndices(int index) {
    Stack<Integer> stack = new Stack<Integer>();

    for (int i = 0; i < n; i++) {
        stack.push(index % n);
        index = index / n;
    }

    return stack;
}

@SuppressWarnings("unchecked")
public Integer get(int i) throws Exception {

    if (i < 0 || i >= maxIndex)
        throw new IndexOutOfBoundsException();

    Stack<Integer> stack = getIndices(i);
    List<Object> temp = superList;

    while (!stack.isEmpty()) {
        temp = (List<Object>) temp.get(stack.pop());
    }

    return (Integer) (temp.get(0));
}

@SuppressWarnings("unchecked")
public void put(int i, int num) {
```

```
        if (i < 0 || i >= maxIndex)
            throw new IndexOutOfBoundsException();

        Stack<Integer> stack = getIndices(i);
        List<Object> temp = superList;

        while (!stack.isEmpty()) {
            temp = (List<Object>) temp.get(stack.pop());
        }

        temp.add(0, num);
    }
}
```

d- Before we determine the complexity of initialization, let's find out the space complexity, which will also help us figure out the former. Looking at the diagram, you can count all the nodes and verify they sum up to:

$$Total\ Nodes = 3^1 + 3^2 + 3^3$$

$$Total\ Nodes = 3 + 9 + 27$$

$$Total\ Nodes = 39$$

The above is a geometric series and the sum of powers of any integer ***x*** can be represented as:

$$Sum = x^0 + x^1 + x^2 + \cdots + x^k$$

$$Sum = 1 + x^1 + x^2 + \cdots + x^k$$

$$Sum = \frac{x^{k+1} - 1}{x - 1}$$

We'll plug in the value of the n equal to 3 in the above formula and also subtract one since our sum doesn't include $3^0 = 1$.

$$Sum = \frac{3^{3+1} - 1}{}$$

$$Sum = \frac{3^4 - 1}{3 - 1}$$

$$Sum = \frac{81 - 1}{2}$$

$$Sum = \frac{80}{2}$$

$$Sum = 40$$

But we also need to subtract 1.

$$Total\ Nodes = 40 - 1 = 39$$

Therefore for an n-dimensional array the total number of nodes will be:

$$Total\ Nodes\ in\ NDimensional\ List = \frac{n^{n+1} - 1}{n - 1} - 1$$

We can rewrite the above equation as:

$$Total\ Nodes\ in\ NDimensional\ List = \frac{n^{n+1}}{n - 1} - \frac{1}{n - 1} - 1$$

For big O we can ignore the last term, which is a constant. Similarly, the second term will tend to zero as *n* becomes larger and larger

$$\frac{1}{n - 1} \to 0\ as\ n\ \to \infty$$

So both the second and the last terms will not affect big O as the input size *n* becomes larger and larger. For very large n, total nodes are roughly equal to:

$$\frac{n^{n+1}}{n - 1}$$

$$Total\ Nodes\ in\ NDimensional\ List \approx \frac{n^{n+1}}{n-1}$$

From the denominator, we can also ignore the constant 1 as for very large values of *n*, the values *n-1* and *n* are roughly equal.

$$n - 1 \approx n$$

We can now restate the sum of nodes as:

$$Total\ Nodes\ in\ NDimensional\ List \approx \frac{n^{n+1}}{n}$$

$$Total\ Nodes\ in\ NDimensional\ List \approx \frac{n^n * n}{n}$$

$$Total\ Nodes\ in\ NDimensional\ List \approx n^n$$

$$Total\ Nodes\ in\ NDimensional\ List\ is\ O(n^n)$$

There, we have worked out the space complexity to be ***O(n^n)***.

b- In order to determine the time to initialize the n-dimensional list, we need to realize that each node of the tree has to be visited and initialized. In fact, in our given solution, the leaf node is also a linked list of length one. We already know the big O for the total number of nodes in the list, therefore the big O for initialization would be proportional to the number of nodes that need to be visited which is **O(n^n)**.

c- To determine the complexity of the `get` and `put` method realize that both the methods call `getIndices` which runs a loop for *n*. Therefore to get the indices of the linked lists to traverse, we need to run a loop for **O(n)**. Next, once we have determined the indices, we need to traverse to the right leaf node.

Look at the diagram above and note that to get to the last leaf node with **index = 27**, we need to hop over all the 3 green nodes, then we hop over 3 yellow nodes and lastly we hop over 3 purple nodes for a total of **3 * 3 = 9 nodes**. So

can the complexity to reach a leaf node be **n * n= n^2 ?** If you think about it, it

makes sense. The n-dimensional list will have exaclty *n* levels. At each level, in the worst case, we'll need to hop over *n* nodes to get to the next linked list to traverse or hit a leaf node. Hence the complexity will indeed be **O(n^2)**.

But Don't forget the `getIndices` method ! The total complexity is:

> $$O(n) + O(n^2) = O(n^2)$$