Q1: 35/35

Q2: 25/25

Q3: 37/40

97/100

ST340 Assignment 1

Anojan Mariathas (2244070), Jude Thomas (2210732), Mathuyan Sivapalan (2242525)

Contributions:

- Question 1: Jude took the lead in writing most of the code for the algorithms after thorough discussions with Mathuyan and Anojan. Meanwhile, Mathuyan and Anojan contributed by working on the explanations and proofs of the question.
- Question 2: Mathuyan took the lead in writing majority of the code for the majority_element algorithm after an in-depth discussion with Anojan and Jude. Jude and Anojan constructed the proof in **2b** after discussing a sketch proof with Mathuyan as well as being responsible for proving why the algorithm works.
- Question 3: Anojan took the lead in sourcing and modifying the code for this question, from the one provided in Lab 2. Jude and Mathuyan both provided peer reviews pointing out any places for improvement and contributed thoroughly to the discussion of why the solution minimises the objective function.

Question 1 35/35

(a) 10/10

The `merge` algorithm is designed to combine two pre-sorted arrays into a single sorted array in linear time. It achieves this by maintaining two pointers, one for each input array, and iteratively comparing the elements at these positions. The smaller element is appended to the result array, and the corresponding pointer is advanced. This process continues until all elements from one of the arrays are exhausted. Any remaining elements from the other array are then appended directly to the result array. The algorithm ensures correctness by comparing and processing each element only once, resulting in a time complexity of $O(n + m)$, where n and m are the lengths of the two input arrays.

```
merge <- function(arr1, arr2) {
  i <- 1
  j <- 1
  merged_array <- c()

  # Compare elements from both arrays and the smaller one to the merged array
  while (i <= length(arr1) && j <= length(arr2)) {
    if (arr1[i] < arr2[j]) {
      merged_array <- c(merged_array, arr1[i])
      i <- i + 1
    } else {
      merged_array <- c(merged_array, arr2[j])
      j <- j + 1
    }
  }


  if (i <= length(arr1)) {
```

```
    merged_array <- c(merged_array, arr1[i:length(arr1)])
  }


  if (j <= length(arr2)) {
    merged_array <- c(merged_array, arr2[j:length(arr2)])
  }

  return(merged_array)
}

# Test Example
arr1 <- c(1,3,3,6,7)
arr2 <- c(2, 4, 5)
result <- merge(arr1, arr2)
print(result)
```

```
## [1] 1 2 3 3 4 5 6 7
```

(b) 8/8

The following algorithm is a divide-and-conquer sorting technique that recursively divides an array into smaller sub-arrays, sorts them, and merges the results to produce a fully sorted array. The algorithm works as follows:

1. **Base Case**: If the array has one or zero elements, it is already sorted and is returned directly.
2. **Divide Step**: The array is split into two halves. The midpoint is calculated, and the array is divided into `left` and `right` sub-arrays.
3. **Recursive Step**: The `mergesort` function is called recursively on both the `left` and `right` sub-arrays until they are reduced to base cases.
4. **Merge Step**: The sorted sub-arrays are combined using the `merge` function to produce a single sorted array.

The algorithm ensures correctness by sorting smaller sub-arrays before merging them, leveraging the pre-sorted property of the sub-arrays. The time complexity of `mergesort` is $n \log_2(n)$, as the array is divided into halves at every recursion level ($\log n$) and merging takes $O(n)$ at each level. This makes `mergesort` a highly reliable sorting algorithm for large data sets, with consistent performance regardless of the input's initial order.

```
mergesort <- function(arr) {
  # Base case
  if (length(arr) <= 1) {
    return(arr)
  }

  # Divide the array into two halves
  mid <- floor(length(arr) / 2)
  left <- arr[1:mid]
  right <- arr[(mid + 1):length(arr)]

  # Recursively sort both halves
  sorted_left <- mergesort(left)
  sorted_right <- mergesort(right)

  # Use the pre-defined merge function to combine sorted halves
```

```r
  return(merge(sorted_left, sorted_right))
}

# Example
unsorted_array <- c(6, 3, 5, 1, 8, 7)
unsorted_array2 <- c(1,5,3,6,2)
sorted_array <- mergesort(unsorted_array)
sorted_array2<- mergesort(unsorted_array2)
print(sorted_array)
```

```
## [1] 1 3 5 6 7 8
```

```r
print(sorted_array2)
```

```
## [1] 1 2 3 5 6
```

(c) 6/6

Proof by induction of Mergesort

We claim that for any input array of length $n \in \mathbb{N}$, mergesort will output the array in sorted order. We will prove this by induction, assuming that the merge function correctly combines two sorted arrays into a single sorted array.

Base Case: For $n = 1$, the input array is a single element $x = (x_1)$, which is inherently sorted. In this case, mergesort directly returns the input array. Thus, the claim holds for $n = 1$.

Inductive Hypothesis: Assume that for all arrays of size $1, 2, \ldots, k$, mergesort correctly outputs the array in sorted order. That is, for any array A of size up to k:

$$\text{mergesort}(A) \text{ outputs a sorted permutation of } A.$$

*__Inductive Step__: We must show that `mergesort` correctly sorts any array A of size $n = k + 1$.

1. **Splitting**: Mergesort splits A into two sub-arrays:

$$A_{\text{left}} = A[1 : \lfloor n/2 \rfloor], \quad A_{\text{right}} = A[\lfloor n/2 \rfloor + 1 : n].$$

   The lengths of A_{left} and A_{right} are strictly less than n. Specifically:

$$|A_{\text{left}}| \leq \lfloor n/2 \rfloor, \quad |A_{\text{right}}| \leq \lceil n/2 \rceil.$$

2. **Recursive Sorting**: By the inductive hypothesis, mergesort correctly sorts A_{left} and A_{right}. Therefore, the recursive calls return:

$$\tilde{A}_{\text{left}} = \text{sorted version of } A_{\text{left}}, \quad \tilde{A}_{\text{right}} = \text{sorted version of } A_{\text{right}}.$$

3. **Merging**: By assumption, the merge function correctly combines two sorted arrays into a single sorted array. Applying merge to $\tilde{A}_{\text{left}}$ and $\tilde{A}_{\text{right}}$ produces a sorted array $\tilde{A}$ that is a permutation of A.

4. **Result**: The output $\tilde{A}$ is sorted and contains all elements of A. Thus:

$$\text{mergesort}(A) = \tilde{A}.$$

Conclusion: By the principle of mathematical induction, mergesort correctly sorts any input array A of size $n \in \mathbb{N}$.

(d)

Let $T(n)$ be the total number of comparisons made by mergesort on an input of size n. Assume $n = 2^k$ for $k \in \mathbb{N}$. We will prove by induction that:

$$T(n) \leq n \log_2(n)$$

Base Case ($n = 2$):

For $n = 2$, `mergesort` requires exactly 1 comparison: $T(2) = 1$.

The claim is:

$$T(2) \leq 2 \log_2(2)$$

Since $\log_2(2) = 1$, this holds: $1 \leq 2 \times 1$

Inductive Hypothesis:

Assume the claim holds for $n = 2^k$, i.e.,

$$T(2^k) \leq 2^k \log_2(2^k)$$

Inductive Step:

We must show that the claim holds for $n = 2^{k+1}$:

$$T(2^{k+1}) \leq 2^{k+1} \log_2(2^{k+1})$$

From the recurrence relation for mergesort:

$$T(2^{k+1}) \leq 2T(2^k) + 2^{k+1}$$

Substitute the inductive hypothesis $T(2^k) \leq 2^k \log_2(2^k)$:

$$T(2^{k+1}) \leq 2 \cdot (2^k \log_2(2^k)) + 2^{k+1}$$

Simplify:

$$T(2^{k+1}) \leq 2^{k+1} \log_2(2^k) + 2^{k+1}$$

Since $\log_2(2^k) = k$, substitute:

$$T(2^{k+1}) \leq 2^{k+1} k + 2^{k+1}$$

Factor out 2^{k+1}:

$$T(2^{k+1}) \leq 2^{k+1}(k + 1)$$

Finally, recall that $\log_2(2^{k+1}) = k + 1$, so we can write:

$$T(2^{k+1}) \leq 2^{k+1} \log_2(2^{k+1})$$

Thus, by induction, we have proved that:

$$T(n) \leq n \log_2(n)$$

for all $n = 2^k$.

(e)

Both **mergesort** and **quicksort** (**with median pivot selection**) use a **divide-and-conquer** strategy to break the problem into smaller parts before recombining them into a sorted array. Both have an average and worst-case time complexity of $O(n \log_2 n)$, as choosing the median pivot in quicksort ensures balanced partitions, avoiding the typical worst-case $O(n^2)$ scenario.

Mergesort always divides the array into equal halves, making it consistent regardless of input order, while quicksort's partitioning is data-dependent but remains balanced with a median pivot. In terms of memory efficiency, quicksort is more space-efficient, as it manages data more effectively during sorting. This makes quicksort a reliable choice for handling large data-sets while maintaining optimal time complexity.

Question 2 25/25

(a) 18/18

1. Define FUNCTION majority_element(A):
   - 1.1 Set N ← length(A).
   - 1.2 If N == 1, then:
     - 1.2.1 Return A[1].
   - 1.3 Set MID ← floor(N / 2).
   - 1.4 Set LEFT ← A[1:MID].
   - 1.5 Set RIGHT ← A[MID+1:N].
   - 1.6 Set LEFT_MAJORITY ← majority_element(LEFT).
   - 1.7 Set RIGHT_MAJORITY ← majority_element(RIGHT).
   - 1.8 If LEFT_MAJORITY == RIGHT_MAJORITY, then:
     - 1.8.1 Return LEFT_MAJORITY.
   - 1.9 Set LEFT_COUNT ← count_equivalence(A, LEFT_MAJORITY).
   - 1.10 Set RIGHT_COUNT ← count_equivalence(A, RIGHT_MAJORITY).
   - 1.11 If LEFT_COUNT > N / 2, then:
     - 1.11.1 Return LEFT_MAJORITY.
   - 1.12 Else if RIGHT_COUNT > N / 2, then:
     - 1.12.1 Return RIGHT_MAJORITY.
   - 1.13 Return "no majority".
2. Define FUNCTION count_equivalence(A, candidate):
   - 2.1 Set COUNT ← 0.
   - 2.2 For each ELEMENT in A:
     - 2.2.1 If ELEMENT == candidate, then:
       - 2.2.1.1 Set COUNT ← COUNT + 1.
   - 2.3 Return COUNT.

```r
# Function to find the majority element using divide and conquer
majority_element <- function(a) {
  n <- length(a)

  # Base case: if only one element, return it as the majority
  if (n == 1) {
    return(a[1])
```

```
  }

  # Divide the array into two halves
  mid <- floor(n / 2)
  left <- a[1:mid]
  right <- a[(mid + 1):n]

  # Recursively find majority elements in both halves
  left_majority <- majority_element(left)
  right_majority <- majority_element(right)

  # If both halves have the same majority element, return it
  if (left_majority == right_majority) {
    return(left_majority)
  }

  # Count the occurrences of each candidate in the entire array
  left_count <- count_equivalence(a, left_majority)
  right_count <- count_equivalence(a, right_majority)

  # If either majority element appears more than n/2 times, return it
  if (left_count > n / 2) {
    return(left_majority)
  } else if (right_count > n / 2) {
    return(right_majority)
  }

  # If neither majority element appears more than n/2 times, return "no majority"
  return("no majority")
}

# Helper function to count equivalence of candidate in array
count_equivalence <- function(a, candidate) {
  count <- 0
  for (elem in a) {
    if (identical(elem, candidate)) {  # Check equivalence relation
      count <- count + 1
    }
  }
  return(count)
}
```

```
# Example usage
a <- c(1, 2, 2, 2, 3, 3, 2, 2)
majority_element(a)
```

```
## [1] 2
```

```
majority_element(c(1, 1, 2, 3, 3, 3, 3))
```

```
## [1] 3
```

```
# Example where there is no majority
array<- c(2,2,4,4,6,6,7,7,8,8,9,9,10,10,12,12)
majority_element(array)
```

```
## [1] "no majority"
```

Case 1: A Majority Element t Exists

If there is a majority element t, it appears in more than $\frac{n}{2}$ positions in the full array. When the array is split into two halves, each of size $\frac{n}{2}$:

- Since t appears in more than $\frac{n}{2}$ positions, at least one subarray must contain t as a majority element.
- It is possible that both subarrays independently identify t as their majority.
- In either case, during the combination step:
  - If both halves return t, then t is returned as the majority element.
  - If one half returns t, its frequency is counted in the full array, and since t appears in more than $\frac{n}{2}$ positions, it is correctly identified as the majority element. Therefore, the array will correctly return t as its majority element.

Case 2: No Majority Element Exists

If there is no majority element in the full array:

- When the array is split into two subarrays, each of size $\frac{n}{2}$, there are two possible outcomes:
  1. **Neither subarray has a majority element:**
     - In this case, the recursive calls will return `"no majority"`, and the algorithm correctly outputs `"no majority"`.
  2. **One subarray has a majority element t, but the other does not:**
     - The algorithm then checks whether t appears more than $\frac{n}{2}$ times in the full array.
     - Since there is no true majority, t will appear at most $\frac{n}{2}$ times, meaning it is not a majority.
     - The algorithm then correctly outputs `"no majority"`.

(b) 7/7

Let $T(n)$ denote the running time. In this algorithm, we split the array into two halves, giving us two subarrays. There are four possible cases:

- **Case 1:** Both sub arrays return **no majority**. Neither sub array has a majority element, whic would mean that the full array does not have a majority element overall. The recurrence relation is given by $T(n) = 2T(\frac{n}{2})$.
- **Case 2:** One sub array has a majority element, while the other does not. The only possible majority for the full array is the element that was the majority in one of the sub arrays. To verify whether it remains the majority in the full array, we count its occurrences in the full array. This requires an additional $O(n)$ comparisons: $T(n) = 2T(\frac{n}{2}) + n$
- **Case 3:** Both sub arrays have a majority, and the majority element is the same in both halves. Then, the full array has the same majority element, requiring no additional work: $T(n) = 2T(\frac{n}{2})$
- **Case 4:** Both sub arrays have a majority, but the majority elements are **different**. To determine if one of them is the majority in the full array, we count the number of occurrences of both arrays, and so we have: $T(n) = 2T(\frac{n}{2}) + 2n$

Since the worst-case scenario occurs in **Case 4**, the upper bound for the running time is given by: $T(n) = 2T(\frac{n}{2}) + 2n$

We now prove an upper bound for $T(n)$.

Base Case $n = 2$

In the context of the **majority element algorithm**, when the input size is **1**, the algorithm **immediately returns the single element** without performing any additional operations hence $T(1) = 0$.

$$T(2) \leq 2T(1) + 4 = 4$$

Also, since: $(2n)\log_2(n) = (2 \times 2)\log_2(2) = 4$ the base case holds.

Inductive Hypothesis

Assume that for $n = 2^k$:

$$T(2^k) \leq 2^{k+1}\log_2(2^k)$$

Inductive Step

We want to prove that it holds for $n = 2^{k+1}$, ie.

$$T(2^{k+1}) \leq 2(2^{k+1})\log_2(2^{k+1})$$

We use the bound from *1(d)*, $T(2^{k+1}) \leq 2T(2^k) + 2(2^{k+1})$, we apply it to the inductive hypothesis $T(2^k) \leq 2^{k+1}\log_2(2^k)$ to get

$$\begin{aligned} T(2^{k+1}) &\leq 2(2^{k+1}\log_2(2^k)) + 2^{k+2} \\ &= 2^{k+2}\log_2(2^k) + 2^{k+2} \end{aligned}$$

Using the fact that $\log_2(2^{k+1}) = \log_2(2^k) + 1$, we get:

$$\begin{aligned} T(2^{k+1}) &\leq 2^{k+2}(\log_2(2^k) + 1) \\ &= 2^{k+2}\log_2(2^{k+1}) \\ &= 2 \times 2^{k+1}\log_2(2^{k+1}) \end{aligned}$$

Thus, the induction step holds when $n = 2^{k+1}$, proving that:

$$T(n) \leq (2n)\log_2(n)$$

for all $n = 2^k$.

Question 3 37/40

```
load("vonNeumann.rdata")
```

Note that some of the code from Lab 2 has been reused.

We want to find the value of k such that the following objective function is minimised:

$f(k, b_1, \ldots, b_k, d_1, \ldots, d_k, c_1, \ldots c_k) = \exp(||\mathbf{A} - \mathbf{B}||_F) + \lambda k(m + n + 1)$

```
# Define helper functions (sourced from Lab 2 code)
viewImage <- function(x) {
  plot(1:2, axes=FALSE, xlab="", ylab="", type='n')
  rasterImage(x, xleft=1, ybottom=1, xright=2, ytop=2)
}

# Main compression function
compressImage <- function(matrixA, lambda) {
  # Perform SVD
  svdResult <- svd(matrixA)
  U <- svdResult$u
  D <- svdResult$d
  V <- svdResult$v
```

Some explanations of why you use SVD are needed.

```r
  # Dimensions of the matrix
  m <- nrow(matrixA)
  n <- ncol(matrixA)
  p <- min(m, n)

  # Initialise storage for results
  # Create a numeric vector of length p
  objectiveValues <- numeric(p)
  # Empty list with p elements, each element holds the rank k approximation matrix
  compressedMatrices <- vector("list", p)

  # Iterate over possible k values
  for (k in 1:p) {
    # Construct the rank-k approximation of the matrix A using the first k columns of U,
    #the first k singular values from D and the first k columns of V
    B_k <- U[, 1:k] %*% diag(D[1:k], nrow=k, ncol=k) %*% t(V[, 1:k])

    # Calculate the Frobenius norm of the difference
    frobeniusNorm <- norm(matrixA - B_k, type = "F")

    # Compute the objective function
    objectiveValues[k] <- exp(frobeniusNorm) + lambda * k * (m + n + 1)

    # Store the compressed matrix in the list created prior
    compressedMatrices[[k]] <- B_k
  }

  # Find the optimal k (index) that minimises the objective function
  optimalK <- which.min(objectiveValues)

  # Get the compressed image and normalise the compressed image to be between 0 and 1
  compressedImage <- compressedMatrices[[optimalK]]
  compressedImage <- (compressedImage - min(compressedImage)) /
                                  (max(compressedImage) - min(compressedImage))

  # Return the normalised compressed image and optimal value k
  return(list(compressed = compressedImage, optimalK = optimalK))
}

# Load the greyscale von Neumann matrix
matrixA <- img
#if (length(dim(matrixA)) > 2) {
 # matrixA <- apply(matrixA, c(1, 2), mean)  # Convert to greyscale
#}

dims <- dim(matrixA)
if (length(dims) > 2) {
  # convert the image into greyscale manually
  m <- dims[1]
  n <- dims[2]
  mtx <- matrix(0, m, n)
  for (i in 1:m) {
    for (j in 1:n) {
```

```
      mtx[i, j] <- sum(matrixA[i, j, ]) / 3
    }
  }
  matrixA <- mtx  # Assign greyscale matrix to matrixA
}

# Compare with lambda = 1 vs lambda = 2
result1 <- compressImage(matrixA, lambda=1)
result2 <- compressImage(matrixA, lambda=2)

# Plot original and compressed images
par(mfrow = c(1, 3), mar = c(1, 1, 3, 1))
viewImage(matrixA)
title("Original Image", cex.main = 5)
viewImage(result1$compressed)
title(paste("Compressed (Lambda=1, k=", result1$optimalK, ")", sep=""), cex.main = 5)
viewImage(result2$compressed)
title(paste("Compressed (Lambda=2, k=", result2$optimalK, ")", sep=""), cex.main = 5)
```

The rank-k matrix approximation ensures that B_k captures the most important k-dimensional features of the image, as the first k singular values D_k represent the majority of the image's energy. The Frobenius norm measures the error when using this approximation, where a lower k reduces the complexity but increases the error while a higher k reduces the error but increases complexity. The penalty term $\lambda k(m+n+1)$ ensures that we find a balance between these competing factors.

By computing $f(k)$ for all $k \in [1, \min(m,n)]$, we observe that the optimal k minimises the trade-off, giving a balance between compression and quality.

From the output, we see that the optimal value of k when $\lambda = 1$ is 67 and the optimal value of k when $\lambda = 2$ is 62. We expect that the value of k decreases as λ increases from 1 to 2 as for a higher λ, there is a penalty for choosing a larger k which pushes the optimisation to selecting a smaller value of k.

In other words, as the cost of complexity increases, the chosen rank decreases, resulting in a more compressed image at the cost of a slight approximation error.