Assignment 1

Stephen Brownsey (1619685), Robin La (1602772) and Niraj Shah (1601955)

Q1.

(a)

In this question we are required to merge two sorted lists into one sorted list. To do this we take our two sorted lists and iterate through them, comparing each element as we and choosing which goes in each index. To test our merge function, we'll use the values in the assignment and randomly generate two sorted lists and apply the function to them to ensure it works as expected.

```
merge <- function(a, b) {
    #Defining a numeric variable of length n in advance
    output <- numeric(length(a) + length(b))
    #defining the variables
    ai <- 1; bi <- 1; j <- 1;
    #Looping through the variables to be merged
    for(j in 1:length(output)) {

        if((ai <= length(a) && a[ai] < b[bi]) || bi > length(b)) {
            output[j] <- a[ai]
            ai <- ai + 1
        } else {
            output[j] <- b[bi]
            bi <- bi + 1
        }
    }
    output
}
merge(c(1,3,3,6,7), c(2,4,5))
```

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

```
a <- sort(c(round(runif(25, min = 0, max = 10),0)), decreasing = FALSE)
b <- sort(c(round(runif(25, min = 0, max = 10),0)), decreasing = FALSE)
a
```

```
##  [1] 1 1 1 1 2 2 2 3 3 3 3 3 4 4 4 5 5 5 7 8 8 8 8 8 9
```

```
b
```

```
##  [1]  0  1  1  2  2  2  2  3  3  3  3  5  6  6  6  7  7  7  7  8  8 10 10
## [24] 10 10
```

```
merge(a,b)
```

```
##  [1]  0  1  1  1  1  1  1  2  2  2  2  2  2  2  3  3  3  3  3  3  3  3  3
## [24]  4  4  4  5  5  5  5  6  6  6  7  7  7  7  7  8  8  8  8  8  8  8  9
## [47] 10 10 10 10
```

(b)

```
mergesort <- function(input) {
    #Recursive calls till they are of length one
    if(length(input) > 1) {
        #Finding the midpoint
        q <- ceiling(length(input)/2)
        #Recursive merch call on left hand side
        a <- mergesort(input[1:q])
        #Recursive merch call on right hand side
        b <- mergesort(input[(q+1):length(input)])
        #Merging together
        merge(a,b)
    } else {
        #Outputting the input in sorted form once completed
        input
    }
}
#Testing the code works
list_test <- sample(0:10, 25, replace = TRUE)
list_test1 <- sample(0:10, 30, replace = TRUE)
list_test
```

```
##  [1] 9 3 0 1 9 8 1 0 9 3 4 5 2 1 5 5 3 7 8 9 9 7 1 6 4
```

```
mergesort(list_test)
```

```
##  [1] 0 0 1 1 1 1 2 3 3 3 4 4 5 5 5 6 7 7 8 8 9 9 9 9 9
```

```
list_test1
```

```
##  [1]  7  2  6  3 10  8  0  2  9  0  7  9  4  8  3  2  3 10  7  7  8  5  3
## [24]  3  5  3  8  1  4  8
```

```
mergesort(list_test1)
```

```
##  [1]  0  0  1  2  2  2  3  3  3  3  3  3  4  4  5  5  6  7  7  7  7  8  8
## [24]  8  8  8  9  9 10 10
```

(c)

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

Base case - For $n = 1$, the input array is a single element $x = (x_1)$, hence already sorted.

Assumption - Suppose, for input array of length $n = 1, 2, ..., k$, our claim holds, that is, for any input array up to length k, mergesort will correctly output this in sorted order.

Inductive step - We will prove that for any array of length $n = k + 1$, e.g. $x = (x_1, x_2, ..., x_{k+1})$, mergesort will return the array in sorted order too. The mergesort algorithm firstly splits the array into two subarrays x_{left} (the elements on the left) and x_r (the elements on the right) about the center (We will take the ceiling function of it and include in x_{left}, to avoid issues with non-integer values).

Note that both x_{left} and x_{right} are less than k elements long. From our inductive step we know that mergesort works for arrays of ANY length up to k. Then mergesort will return both subarrays in sorted order - $\tilde{x}_{left}$ and $\tilde{x}_{right}$.

We have assumed that the merge function correctly sorts two sorted arrays, so passing $\tilde{x}_{left}$ and $\tilde{x}_{right}$ through the merge function will give us the sorted output array of length $k+1$, which is exactly what we wanted. As we have proved this for any general $n = k+1 \in \mathbb{N}$, we can conclude that mergesort does indeed output its input array in sorted order.

(d)

After splitting the initial array into two parts, the algorithm will then make comparisons on the two subarrays, so we have $2T(\frac{n}{2})$ comparisons. At the end of the mergesort, we use the merge algorithm to merge the two arrays into sorted order, which takes n comparisons at most. So the required recurrence inequality for T(n) is: $T(n) \leq 2T(\frac{n}{2}) + n$

We now use this to prove the inequality $T(n) \leq nlog_2 n$ for all $n \in \{2^k : k \in \mathbb{N}\}$, via induction.

Base Case: Let $n = 2$. Then $T(2) = 2T(1) + 2 = 2$. Also $2log_2 2 = 2$, so the required inequality holds for the base case.

Assumption Case: Assume it holds for $n = 2^k$, so $T(2^k) \leq 2^k log_2 2^k$, we want to show that it holds for $n = 2^{k+1}$

Inductive step:

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

So it holds for $n = 2^{k+1}$, hence the statement holds for all $n \in \{2^k : k \in \mathbb{N}\}$

(e)

Both quicksort and mergesort are divide and conquer algorithms where you break the original problem down and solve these smaller problems iteratively before recombining the list to give the final output. Both have average running times $O(nlog(n))$, however quicksort has a worst case running time of $O(n^2)$ - though this can be overcome by good pivot choice. One such method would be randomly choosing the pivot.

Note that if we pass an already sorted array into mergesort, it will still make the same number of computations. For quicksort, if we choose the pivot to be the first element, the worst case is achieved when the array is in reverse-sorted order.

From a space perspective, quicksort is more efficient as it is an internal sorting algorithm, in other words doesn't require any additional memory as it is done in main memory. On the other hand, mergesort requires additional memory space for the auxiliary to be stored.

Q2. Majority Element

(a)

In this scenario, it is simplest if we break our majority element function down into smaller substeps. The first, is_majority, returns *NULL* if the chosen element, x, is not a majority element and x if it is a majority element. Then, our is_majority_element function, tells us whether a majority element exists in the whole

input array. It returns the element if it does exist and NULL if it doesn't. Finally, to get the output "no majority" or the majority element we define a 3rd function, majority_element, to do this.

```
is_majority <- function(array, x){
#Defining a count variable
    count <- 0
    if(is.null(x)){
    #if there aren't any x's return NULL
        return(NULL)
    }
    for(i in 1:length(array)){
    #Counting how many x's are there.
        if(array[i] == x){
            count <- count + 1
        }
    }
    #Seeing whether x is a majority element or not
    if(count > ceiling(length(array)/2)){
        return(x)
    }else {
        return(NULL)
    }
}
is_majority_element <- function(array){
    n <- length(array)
    #trivial case
    if(n == 1){
        return(array[1])
    }
    mid = ceiling(n/2) # ceiling not necessary as we know even number but may as well include
    left <- array[1:mid]
    right <- array[(mid + 1):n]

    left_maj <- is_majority_element(left)
    right_maj <- is_majority_element(right)
    count <- 0

    is_left_maj <- is_majority(array, left_maj)
    is_right_maj <- is_majority(array, right_maj)
    #if_else section
    if(is.null(is_left_maj) & is.null(is_right_maj)){
        return(NULL)
    }else if (!is.null(is_left_maj)){
        return(left_maj)
    }else{
        return(right_maj)
    }

}
majority_element <- function(array){
    x <- is_majority_element(array)
    if(is.null(x)){
        return("no majority")
    }else {
```

```
        return(x)
    }
}
array1 <- c(1,2,3,4,5,6,7,8)
array <- c(5,5,5,5,6,6,6,6,5,5,5,5,5,1,23,3,4,5,6,7,8,55,5,5,5,5,5,5,5,5,5)
majority_element(array)
```

```
## [1] 5
```

```
majority_element(array1)
```

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

```
#Testing edge case to confirm algorithm works
majority_element(c(3,3,3,3,4,4,4,4))
```

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

In this algorithm, which splits the array into two smaller sub arrays, we have 2 cases that can occur:

Case 1 - Suppose the majority element t exists. Then, when splitting the full array into two subarrays, each of length $\frac{n}{2}$, (assuming n is even), at least one or even both subarrays will return t as its majority element. Hence, the full array will return t as its majority element.

Case 2 - Suppose the majority element t doesn't exist. Then, when splitting the full array into two subarrays, each of length $\frac{n}{2}$ (assuming n is even), we have two possible outcomes. Either both subarrays will not have a majority element in which case we exit and display "no majority" or one subarray has a majority element t and the other subarray doesn't have a majority element. In this case, the algorithm will look at the full array and see that t is not a majority element, as it will appear less than $\frac{n}{2} + 1$ times in the full array of size n.

(b)

Let $T(n)$ denote the running time. In this algorithm we split the array into half giving us 2 arrays. There are 4 possible scenarios that can happen with this algorithm:

- Both sub arrays return no majority. In this case, neither sub array has a majority element, and thus the whole array has no majority element. Here, $T(n) = 2T(\frac{n}{2})$
- One of the sub arrays has a majority element, but the other sub array doesn't have a majority element. The only possible majority for the whole array is with the element that formed a majority in one of the sub arrays. Here, we just compare every element in the full array and count the number of elements that are equal to this value. This tells us whether this element is the majority for the whole array. Here, $T(n) = 2T(\frac{n}{2}) + n$
- Both sub arrays have a majority, and the majority is the same for both arrays. Then overall we have a majority element, which is the same element that was a majority, in both the arrays. Here $T(n) = 2T(\frac{n}{2})$
- Both sub arrays have a majority, but the majority element is different for both arrays. Then we count the number of elements equal to both of the candidates for majority element in the full array. Either it could be that one of the majority element candidate is a majority element in the full array, or there is no majority element. Here, $T(n) = 2T(\frac{n}{2}) + 2n$

So *upper bound* is given by: $T(n) = 2T(\frac{n}{2}) + 2n$.

Using the result to 1d, we find that $T(n) \leq (2n)log_2(n)$ is a suitable upper bound, where n can be written in the form $n = 2^k$, where k is an integer.

Proof: Induction

The base case is n = 2. Here, $T(2) \leq 2T(1) + 4 = 4$. Also, when n = 2, $(2n)log_2(n) = 4$. So it holds for the base case.

Assume it holds for $n = 2^k$, so $T(2^k) <= 2^{k+1}log_2 2^k$. Want to show it works for $n = 2^{k+1}$

For n = 2^{k+1}

$$
\begin{aligned}
T(2^{k+1}) &\leq 2T(2^k) + 2 \times 2^{k+1} && \text{Upper bound} \\
&\leq 2 \times 2^{k+1} log_2 2^k + 2^{k+2} && \text{Inductive hypothesis} \\
&= 2^{k+2} \times k + 2^{k+2} \\
&= 2^{k+2}(k+1) \\
&= 2 \times 2^{k+1}(k+1) \\
&= 2 \times (2^{k+1}) log_2(2^{k+1}) \\
&= (2n) log_2(n)
\end{aligned}
$$

So it holds for $n = 2^{(k+1)}$

Q3.

```
load("pictures.rdata")
```

Code from lab 2

```
image.compress.param <- function(pic) {
  img <- images[[pic]]

  # find the size of the image
  dims <- dim(img); m <- dims[1]; n <- dims[2]

  if (length(dims) > 2) {
    # convert the image into greyscale
    mtx <- matrix(0,m,n)
    for (i in 1:m) {
      for (j in 1:n) {
        mtx[i,j] <- sum(img[i,j,])/3
      }
    }
  } else {
    mtx <- img
  }

  p <- min(m,n)

  # perform the decomposition
  decomposition <- svd(mtx)

  return(list(img=img, mtx=mtx, p=p, svd=decomposition))
}
compute.compression <- function(k, p, mtx, decomposition) {
  if (1 <= k && k <= p) {

    # compute the k-rank approximation
    if (k == 1) {
      approximation <- decomposition$d[1]*decomposition$u[,1]%*%t(decomposition$v[,1])
```

```
    } else {
      approximation <- decomposition$u[,1:k]%*%diag(decomposition$d[1:k])%*%t(decomposition$v[,1:k])
    }

    approximation.error <- norm(mtx-approximation,type="F")
    approximation.error.theory <- sqrt(sum(decomposition$d[(k+1):p]^2))

    # rescale the approximation so the values of the image matrix are in [0,1]
    maxval <- max(approximation); minval <- min(approximation)
    compressedImage <- (approximation - minval)/(maxval - minval)
    return(compressedImage)
  } else {
    return(NULL)
  }
}
```

Suppose we wish to approximate mxn matrix A, by Schmidt's theorem on low-rank matrix approximations, we know that the rank k matrix B that minimises $||A - B||_F$ is

$$\tilde{A}_k = \sum_{i=1}^{k} \sigma_i u_i v_i$$

where $\tilde{A}_k$ is the k-rank truncated singular value decomposition of original matrix A.

Then, to find the values which minimise function

$$f(k, b_1, ..., b_k, d_1, ...d_k, c_1, ..., c_k) = exp(||A - B||_F) + k(m + n + 1)$$

we should choose b_i, d_i and c_i so that b_i corresponds to u_i, the left singular vectors of A, d_i corresponds to v_i, the right singular vectors of A, and the c_i that corresponds to σ_i, the singular values of A, all obtained from the singular value decomposition of A.

By Eckart-Young-Mirsky theorem, we know that

$$||A - A_k||_F \leq ||A - C||_F$$

for any matrix C such that $rank(C) \leq k$, so increasing k will reduce the value of the frobenius norm - but will increase the space complexity term. So we wish to find a value of k for which the k-rank approximation matrix is sufficiently "close" to the original matrix, whilst keeping the space complexity term $k(m + n + 1)$ low as well. We will use the theorem covered in lectures which states that

$$||A - \tilde{A}_k||_F = \sqrt{\sum_{j=k+1}^{min\{m,n\}} \sigma_j^2}$$

```
res<-image.compress.param(4)
# note that the file pictures.rdata must be loaded onto global environment
decomp<-res$svd
Sigma<-decomp$d
min<-res$p
m<-dim(res$img)[1]
n<-dim(res$img)[2]
list<-c()
for (k in 1:min-1){
  frobeniusNorm = sqrt(sum((Sigma[(k+1):min])^2))
  #summing the singular values from k+1 to p, where p=min{m,n} defined previously
```

```
  value = exp(frobeniusNorm) + k*(m+n+1)
  list[k] = value
}
min(list)
```

```
## [1] 371027
```

```
which.min(list)
```

```
## [1] 257
```