ST340 Assignment 1

Xintian Han 1909780, Runze Wang 1907544, Jingyuan Chen 2029628

Question 1

(a) the merge algorithm is

```
     merge<-function(a,b){
       n_1<-length(a)
       n_2<-length(b)
       n<-n_1+n_2
     c<-rep(0,n)
     i=1
     j=1
     for(k in 1:n){
    if(j > length(b)){
    c[k]<-a[i]
    i=i+1
     }else if
       (i > length(a)){
         c[k]<-b[j]
         j=j+1
       }else if
       (a[i]< b[j]){
         c[k]<-a[i]
         i=i+1
       }else{
         c[k]<-b[j]
         j=j+1
       }
       }
return(c) }
```

ie : we create the merge algorithm by taking two sorted arrays a,b in to one array c with increasing order. For array a, we have length of n_1; similarly for array b, we have length of n_2; for array c, we have length of n=n_1+n_2. We denote the elements in array a,b and c to be a_i,b_j and c_k,with i=1; j=1 at the beginning,then we create the for loop and if else loop to rank these elements in increasing order to create c. For k=1:n,if j is larger than n_2, this means that all elements of array b has been moved to the array c,then the rest positions of the array c have the same elements as the rest element of array a.Therefore,array c equals to array a.If i is larger than n_1,this means that there is no elements of arrays a.Then the rest elements of array c equal to the rest elements of array b.Therefore,array c equals to array b.Otherwise, for j smaller than n_2 and i smaller than n_1;if a_i is smaller than b_j, then c_k has the k^{th} element as a_i or if a_i is bigger than b_j it has the k^{th} element b_j for i=i+1 and j=j+1.Array c is a combination of array a and b in increasing order.

We could use the following to test the merge function is actually working:

```
print(merge(c(1,2,3,4,5),c(1,2)))
```

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

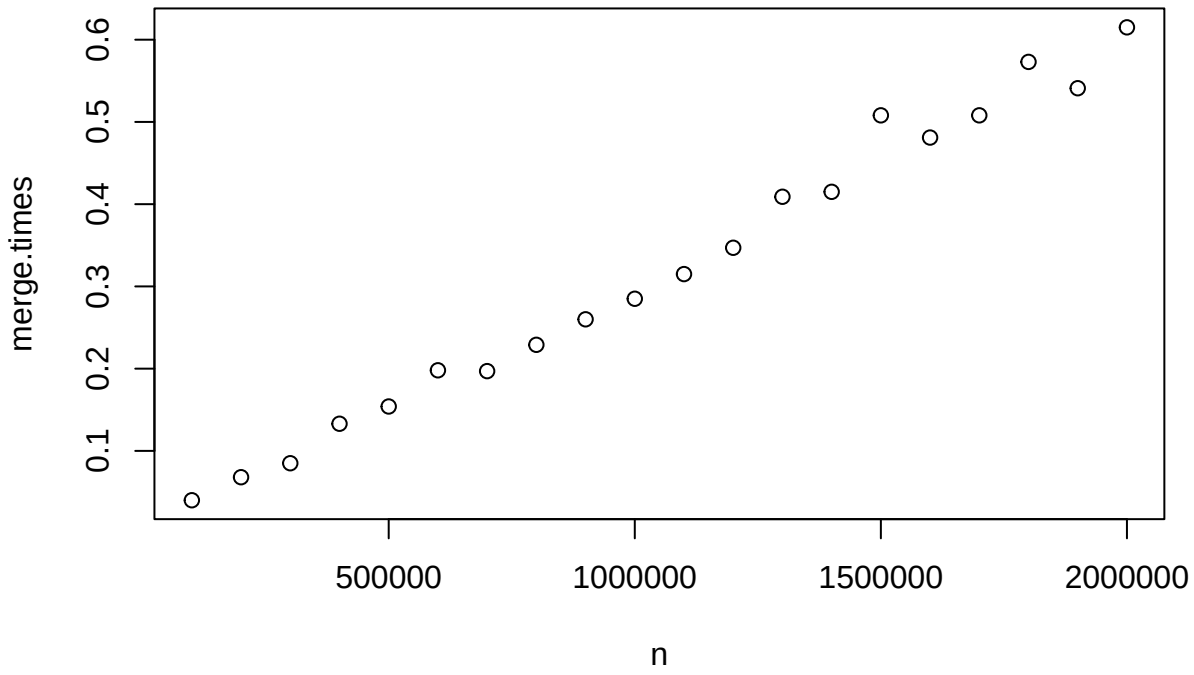
```
print(merge(c(1,3,3,6,7),c(2,4,5)))
```

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

Then we want to find out the time complexity of merge is $O(n_1+n_2)$ where n_1=length(a) and n_2=length(b). To describe the time complexity we need to find the worst case , that is when we compare each element in arrays a and b. We must first pick the smallest element from each array :that is compare a_1 and b_1, then assign the smaller one among these two to array c as c_1. Next, we consider the same issue and take the same action, one comparison in each step for i=2,. . . ,n_1 and j=1,. . . ,n_2 in order to obtain all of the entries from arrays a and b and assign them to array c,that is at most n_1+n_2 steps.We have $O(1)$=0,therefore, the worst case should be the number of steps of multiple one, which is $O(n_1+n_2)$=n_1+n_2-1

We set n=a+b,it also can be seen from the plot with the worst case merge:

```
n<-1e5*(1:20)
a<-1e2
merge.times<- rep(0,length(n))
for (i in 1:length(n)){
  b <-1:n[i]-1
  merge.times[i] <- system.time(merge(a,b))[3]
}
plot(n,merge.times)
```



The graph above shows a straight line which indicates that the merge.times has a linear relationship with the length of a+b. That is as the length of a and b increase, the system time increases linearly.Hence the time complexity of merge alogorithm is $O(n_1 + n_2)$

(b)

```
  mergesort<-function(A) {
 if(length(A) ==1){return(A)}
else{
   pivot <- ceiling(length(A)/2)
  a <- mergesort(A[1:pivot])
  b <- mergesort(A[(pivot+1):length(A)])
  merge(a,b)
     }
  }
```

We can test that the algorithm is actually working:

```
print(mergesort(c(1,54,8,8,9,10,8,3,2,1,2)))
```

```
##  [1]  1  1  2  2  3  8  8  8  9 10 54
```

```
print(mergesort(c(9,7,9,2,16)))
```

```
## [1]  2  7  9  9 16
```

ie : we create the mergesort algorithm with the input array A of length n. if n equals to 1, we return A. If the length n is not 1, we could set a pivot point of ceiling ($length(A)/2$) with first half of the array has elements from the 1^{st} of A to the $ceiling(length(A)/2)^{th}$ of A and has length of ceiling($length(A)/2$) and second half of the array has element from the $(ceiling(length(A)/2) + 1)^{th}$ of A to the $length(A)^{th}$ of A and has length $length(A)$-ceiling($length(A)/2$). We put the first half and second half of the array into the mergesort algorithm with the same process as before to split them further until they both have length of 1.After that,the merge algorithm could be used to produce the array with increased order.

(c)

To prove mergesort by induction,we need to prove that the mergesort and merge are both true.

We first have to prove merge is correct.

We create the for loop and if else loop to rank theses elements in increasing order. In the first iteration, we get the first element in c is the smallest element between a_1 and b_1 ($a_1 \neq b_1$) or one of a_1 and b_1 if ($a_1 \neq b_1$). As a_1 and b_1 are the smallest element in each array, c_1 is the smallest element in array c. In the following k iteration, $c_k \leq a_k$ and $c_k \leq b_k$. so in array c, $c_1 \leq c_2 \leq \ldots\ldots \leq c_k$ The merge is correct.

Then we want to prove the mergesort algorithm:

Proof: By induction on n

Base case : $n = 1$,we can get the number by it self.

Inductive hypothesis: mergesort outputs its input array in sorted order for n$\leq k$

Inductive step: : mergesort outputs its input array in sorted order holds for n=$k + 1 \geq 2$

First recursive M=celling($n/2$)<n, By using induction subarray A[1,2,...M] is sorted

Second recursive n-M<n, using induction subarray A[M+1,....n] is sorted

The Merge can sort A[1,2,...M] and A[M+1,....n] into A[1,2,...n] in the sorted array. We can get that mergesort algorithm outputs its input array in sorted order.

(d)

We can describe the mergesort algorithm as below:

T(n) is the total number of comparison in mergesort with input A of size n that is even. ie mergesort(A)=T(n).If n=1,we return A.Otherwise,we divide the array A into two parts A_1 and A_2, with length n/2 respectively.Then we have mergesort(A_1)=T(n/2) and mergesort(A_2)=T(n/2).Applying the merge(mergesort(A_1), mergesort(A_2)), find the number of comparison equals to $O(n)$,which is obtained from previous part.

The recurrence inequality for $T(n) \leq 2T(n/2) + n$

Prove by induction that $T(n) \leq nlog_2 n$:

Base case:n=1 ,$T(1) = log_2 1 = 0$

Assume n/2 is true, then $T(n/2) \leq (n/2)log_2(n/2)$

Then we consider the case n which is:

$T(n) \leq 2(n/2)log_2(n/2) + n = n(log_2 n + log_2(1/2)) + n = n((log_2 n) - 1) + n = nlog_2 n - n + n = nlog_2 n$

(e)

In quicksort, we define the first element in the array to be the pivot and compare other elements in the array with the pivot, find the smaller,equal and bigger ones. Then continue this same process we will get an array in sorted order. In mergesort, the input array is divided into two parts with the pivot ceiling(length(A)/2), then we continue doing this process until it has length 1 and use merge algorithm to sorted them in ascending order. The time complexity for quicksort is $O(n^2)$ and the time complexity of the mergesort is $O(nlog_2 n)$. Comparing the time complexity we can easily see that mergesort perform better than quicksort.

Question 2

(a)

Pseudo code

Find_Majority(A):

1.majority_element(A):

1.1 If length(A)=1, a ← 1

1.2 if length(A)>1

1.2.1 set c ← A[1:n/2],d ← A[(n/2+1):n]

1.2.2 set c_left← majority_element,d_right← majority_element

1.2.3 set c_leftcount← sum(A==c_left),d_rightcount← sum(A==d_right)

1.2.4 If c_left=d_right, then a← c_left

1.2.5 else if c_leftcount > d_rightcount, then a ← c_left

1.2.6 else if c_leftcount < d_rightcount, then a← d_right

1.2.7 else a← "no majority"

1.3.Output a

2.majority_count(A):

2.1.majority← majority_element(A)

2.2.c← sum(A==majority)

2.3.Output c

3.C ← majority_count(A)

3.1 If C >Length(A/2), then

Outoput majority_element(A)

3.2 else

Output "no majority".

To produce the code, we have input a that is an array of n=2^k elements, which means it has length(a)=n

```
majority_element<-function(a){
  if (length(a)==1) {return(a)}
 else {
#we split the array a into a left and right array has length n/2, n is even
    #called c and d respetively
    n<-length(a)
    c<-a[1:n/2]
    d<-a[(n/2+1):n]

  c_left<-majority_element(c)
  d_right<-majority_element(d)

      if(c_left==d_right){
        return(c_left)}
  else{
    c_leftcount<-sum(a==c_left)
      d_rightcount<-sum(a==d_right)
      if (c_leftcount > d_rightcount ){
        return(c_left)}
      else if(c_leftcount < d_rightcount ){
        return(d_right)}
        else{
          return("no majority")
   }
      }
    }
}

majority_count<-function(a){

  majority<-majority_element(a)
  m_count<-sum(a==majority)
```

```
  return(m_count)
}

Find_Majority<-function(a){
  n<-length(a)
  Count<-majority_count(a)
  if(Count>n/2){
    return(majority_element(a))
  }else{
    return("no majority")
  }
}
```

Test the algorithm actually works:

```
print(Find_Majority(c(1,2,3,3,3,3,3,2)))
```

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

```
print(Find_Majority(c(2,2,2,3)))
```

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

```
print(Find_Majority(c(1,1,2,2,3,3)))
```

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

The explaination of the algorithm is as following:

We assume the input a has length n where n=2^k. Then in order to find the majority element we have to consider the array with length n. When n is 1 the majority element is just a itself. When n>2, we divide the array into two parts with the median n/2 as the pivot, where left array c has length n/2 from 1:n/2 and right array d has length n/2 from (n/2+1):n.Then we apply the same process for several times to find the majority element of c and d. If array c and d has the same majority element, we output the array c. If they don't , then we find the sum of the elements in array a that equal to the majority array of c called it c_leftcount and sum of the elements in array a that equal to the majority array of d called it d_rightcount. If c_leftcount>d_rightcount, output left array c; if c_leftcount<d_rightcount,output d. Otherwise return no majority.We also need to make sure that the occurence of the element in array a equal to majority element of a is bigger than n/2.

(b)

We first solve array size n by recursively solving two smaller arrays (the left array c and the right array d of size n/2 and comparing two smaller groups by counting the number of the majority element of left array and the number of the majority of right array.

We can get the following time complexities:

Time complexity of divide part=$O(1)$

Time complexity of two sub-arrays= $2T(n/2)$

Time complexity of combine part = Time complexity of counting the number of two sub-arrays=$2O(n)$

Therefore, $T(n) = O(1) + 2T(n/2) + 2O(n) = 2T(n/2) + 2O(n)$ for n >1 and $T(n) = O(1)$ for n=1. $T(n) = 2T(n/2) + O(n) \leq 2T(n/2) + n$

This is the same as the Q1(d), we can get $T(n) \leq nlog_2(n)$

The recurrence inequality for $T(n) \leq 2T(n/2) + n$

Prove by induction that $T(n) \leq nlog_2 n$:

Base case:n=1 ,$T(1) = log_2 1 = 0$

Assume n/2 is true, then $T(n/2) \leq (n/2)log_2(n/2)$

Then we consider the case n which is:

$T(n) \leq 2(n/2)log_2(n/2) + n = n(log_2 n + log_2(1/2)) + n = n((log_2 n) - 1) + n = nlog_2 n - n + n = nlog_2 n$

Question 3

From Lab2, we generated the code as below:

```
load("~/Downloads/pictures.rdata")
# helper function to view an image
viewImage <- function(x) {
  plot(1:2,axes=FALSE,xlab="",ylab="",type='n')
  rasterImage(x,xleft=1,ybottom=1,xright=2,ytop=2)
}

image.compression <- function() {
  # choose a picture
  pic <- as.integer(readline(prompt="Choose a picture (1-5) from
  Gauss, Cox, von Neumann, Nightingale or checkerboard: "))
  res <- image.compress.param(pic)

  # plot side-by-side the original image and the singular values
  par(mfrow=c(1,2))
  viewImage(res$mtx)
  plot(res$svd$d)
  abline(h=0,lty=2)

  # until you choose to quit...
  k <- Inf
  while (TRUE) {
    # choose the rank of the approximation
    k <- as.integer(readline(prompt=paste("choose a number of k between 1 and
    ",res$p," (anything else to exit): ",sep="")))
    compressedImage <- compute.compression(k, res$p, res$mtx, res$svd)

    # view the original and compressed image side-by-side
    if (!is.null(compressedImage)) {
      viewImage(res$mtx)
      viewImage(compressedImage)
    } else {
      break;
    }
  }
```

```
}

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))

  print(paste("approximation.error = ",approximation.error,sep=""))
  print(paste("approximation.error.theory = ",approximation.error.theory,sep=""))

  # 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)
  }
}
```

With the code above we could use it to find k that minimise f:

```
#Nightingale has pic=4
Nightingale<-image.compress.param(4)
A<-Nightingale$mtx

#decomposition the matrix A
tmp<- svd(A)
U <- tmp$u
d<- tmp$d
V <- tmp$v

m<-767
n<-584

#find p
p<-min(m,n)
print(p)
```

```
## [1] 584
```

```
#we want to find k minimise the exponential of Frobenius norm
minf <- rep(0,584)
for (k in 1:(583)) {
 minf[k] <- exp(sqrt(sum((d[(k+1):584])^2)))
}

#for k=584
print(minf[584])
```

```
## [1] 0
```

```
#find k that minimises f
f<-minf+(1:584)*(767+584+1)
which.min(f)
```

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

Hence k=257

Stating in another format:in this question,we are going to minimise the function:

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

With matrix B of rank k:

$$B = U_k \Sigma_k V_k^T = \sum_{i=1}^{k} c_i b_i d_i^T$$

ie: U_k is the orthogonal matrix consist of column vectors $b_1, ..., b_k$; V_k is the orthogonal matrix consist of column vectors $d_1, ..., d_k$ and Σ_k is diagonal matrix with eigenvalues $c_1, ...c_k$ in the diagonal entries.

We could determine the value k in order to obtain the minimum value of f. That is the same as minimise the Frobenius norm by choosing the correct value k. We know that applying the Low-rank approximation theorem,we could get the following form of the Frobenius norm:

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

Therefore,we can conclude that when k=257, the minimum value of f is reached.This also indicates that the matrix B has rank 257.