# ST340 Assignment 1

Dylan Dijk 1802183

06/02/2021

## Contents

# 1

## A

**Pseudo code for merge Algorithm**

1. Let $a_1$ and $a_2$ be the two input vectors. $n_1$ and $n_2$ their respective lengths.
2. Create vector `c` of length $n_1 + n_2$
3. Set `i`, `j`, `k` as 1
4. **While** $i \leq n_1$ and $j \leq n_2$
    1. **If** $a_1[i] \leq a_2[j]$ set $c[k] = a_1[i]$
    2. And set $i = i + 1$, $k = k + 1$
    3. **Else** set $c[k] = a_2[j]$
    4. And set $j = j + 1$, $k = k + 1$
5. **While** $i \leq n_1$
    1. Set $c[k, k+1, ..., n_1 + n_2] = a_1[i, ..., n_1]$
6. **While** $i \leq n_2$
    1. Set $c[k, k+1, ..., n_1 + n_2] = a_2[j, ..., n_2]$
7. Output `c`

**General description:**

In this algorithm we traverse both sorted vectors simultaneously, comparing elements termwise and selecting the smallest elements.
The final two **while** statements are used to fill the vector `c` when remaining elements are left from a single vector.

### R Code merge Algorithm

```
merge = function(a1, a2){

  n1 = length(a1)
  n2 = length(a2)

  c = rep(0, (n1 + n2))

  i = 1
  j = 1
  k = 1

  while (i <= n1 && j <= n2) {
    if (a1[i] <= a2[j]){
    c[k] = a1[i]
    i = i+1
    k = k+1

  } else {
    c[k] = a2[j]
    j = j+1
    k = k+1
  }
  }

  while(i <= n1){
    c[k:(n1 + n2)] = a1[i:n1]
    i = n1 + 1
  }

  while(j <= n2){
    c[k:(n1 + n2)] = a2[j:n2]
    j = n2 +1
  }

  return(c)

}
```

### Testing merge

```
merge(a1 = c(1, 3, 3, 6, 7), a2 = c(2, 4, 5))
```

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

## B

**Pseudo code for mergesort Algorithm**

**mergesort(a)**:

1. Let `a` be the input vector
2. set `n` equal to the length of `a`
3. **If** $n > 1$
    1. pivot = $\lfloor n/2 \rfloor$
    2. $L = a[a < pivot]$ , $R = a[a > pivot + 1]$
4. **If** $n = 1$
    1. `mergesort(a)` = a
5. c = `mergesort(L)`, d = `mergesort(R)`
6. merge(c,d)

**R code for mergesort Algorithm**

```
mergesort = function(a){
  n = length(a)
  if(n>1){
    L = a[1:(n%/%2)]
    R = a[((n%/%2)+1):n]

    c = mergesort(L)
    d = mergesort(R)
    merge(c,d)
  } else {
  a
  }

}
```

**Testing mergesort Algorithm**

```
mergesort(a = 5:1)
```

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

## C

Want to prove by induction that `mergesort` outputs its array in sorted order.

### Base case

For $n = 1$ `mergesort(a)` = a
Therefore true for n = 1

### Assumption

Now let $k \in \mathbb{N}$ and assume `mergesort(a)` outputs `a` in sorted order $\forall\ n \leq k$
We will also assume that the `merge` algorithm works $\forall n \in \mathbb{N}$

### Inductive step

Now suppose `a` is of length k + 1
$\Rightarrow$ `length(L)` $\leq k$ and `Length(R)` $\leq k$
$\Rightarrow$ `mergesort(L)` and `mergesort(R)` are in increasing order by assumption.
$\Rightarrow$ `merge(mergesort(L), mergesort(R))` is in increasing order.
Therefore have proved by induction that `mergesort` outputs the array in sorted order.

## D

First of all the `merge` algorithm with arrays of equal length ($n$) has worst case number of comparisons equal to $(2n - 1)$. Therefore $T_{merge}(m, m) < 2m$ where $T_{merge}$ denotes the number of comparisons of the `merge` algorithm.
Now for the `mergesort` algorithm we have the recurrence equation for comparisons:

$$T(n) = 2T\left(\frac{n}{2}\right) + T_{merge}(\frac{n}{2}, \frac{n}{2}) \implies \quad T(n) \leq 2T(\frac{n}{2}) + n$$

With $T(1) = 0$

Now want to prove by **induction** that $T(n) \leq n \log_2 n \ \ \forall n \in \{2^k : k \in \mathbb{N}\}$

### Base case

For k = 1
$T(2) \leq 2T(1) + 1 = 1 \leq 2log_2 2$
Therefore true for $k = 1$

### Assumption

Now assume inequality holds for some $k \in \mathbb{N}$

### Inductive step

Want to show true for $k + 1$
$T(2^{k+1}) \leq 2T(2^k) + (2^{k+1})$ ,by our recurrence inequality given at the start of **D**
$\leq 2^{k+1} log_2 2^k + 2^{k+1} = k(2^{k+1}) + 2^{k+1}$ ,here we use the assumption
$= 2^{k+1}(k + 1) = nlog_2 n$

Therefore have proved this inequality by induction.

## E

`mergesort` works by being recursively applied to the two halves of the input, then using the `merge` algorithm to combine the paired output.

`quicksort` also works by being recursively applied to two sections of the input. But now the pivot is the first element of the input. The pivot is used to compare the size of the elements to determine the splitting of the array, and therefore does not require the `merge` algorithm.

In lectures we saw that worst case number of comparisons for `quicksort` was $\Theta(n^2)$ so is **worse** performing than `mergesort` where we showed the number of comparisons is bounded by $n \log_2 n$.

# 2

## A

**Pseudo code for majority element Algorithm**

**maj_element(x)**:

1. Let `x` be the input vector and `n` the respective length (Where $n = 2^k$)
2. **If** $n > 1$
    1. pivot $= \frac{n}{2}$
    2. Assign $L =$ `maj_element(x[x<(pivot+1)])`
    3. Assign $R =$ `maj_element(x[x>pivot])`
3. **If** $n = 1$ `return(x)`
4. **If** $L == R$ `return(L)`
5. **else If** `sum(x == L) > pivot` then `return(L)`
6. **else If** `sum(x == R) > pivot` then `return(R)`
7. **else** `return('no majority')`

**R code for majority element Algorithm**

```
maj_element = function(x){
  n = length(x)
  if(n == 1){
    return(x)
  }
  if(n > 1){
    m = (n/2)
    L = maj_element(x[1:m])
    R = maj_element(x[(m+1):n])
  }
  if(L == R){
    return(L)
  } else if(sum(x == L) > (m)){
    return(L)
  } else if(sum(x == R) > (m)){
    return(R)
  } else{
    return('no majority')
  }
}
```

This algorithm works by recursively splitting the array into two and applying the algorithm on each returning the values L and R. When the input is just a single number it prints this number.
Then if L is equal to R we return this value.
If they are different we count the number of times L appears in the original input and then R, if one of these is strictly greater than n/2 we return this number if not we print 'no majority element'.

*Now when I refer to L and R it is the values specifically we get from the first recursive call.*

The reason the algorithm works:

- If there is a majority element it will exist in both halves and therefore by force L and R will take this value.
  - Therefore L will be equal to R and will be returned as the majority element.
- If there isn't a majority element then either: both L and R are 'no majority element' or they are different numbers. This second scenario is possible if the array is for example `(2,2,2,2,3,3,3,3)`, in which case L and R will be equal to 2 and 3 respectively.
  - In both cases the algorithm will print 'no majority element'.

## B

In the worst case the number of equivalence checks $T(n)$ is $2T(\frac{n}{2}) + 2n$.
This the case when L is equal to R, as then we have to count how many times each appears in the array which requires 2n equivalence checks. So we have the bound:

$$T(n) \leq 2T(\frac{n}{2}) + 2n$$

With $T(1) = 0$
An upper bound on the worst case number of comparisons is $2nlog_2 n$. I got to this by expanding this recurrence inequality.

**Proof by induction**

For k = 1. **Base case**

$T(2) \leq 2T(1) + 2 = 2 \leq 2 \times 2 \times \log_2 2$

Now **assume** the upper bound holds for some $k \in \mathbb{N}$
Now want to show bound holds for $k + 1$
$T(2^{k+1}) \leq 2T(2^k) + 2^{k+2}$
$\leq 2[2^{k+1}log_2(2^k)] + 2^{k+2}$
$\leq 2[k2^{k+1}] + 2^{k+2}$
$\leq (k+1)2^{k+2} \quad = \quad 2(2^{k+1})log_2 2^{k+1}$

Therefore have proved the bound holds $\forall k \in \mathbb{N}$ by induction.

# 3

The function `image.compress.param()`calculates the greyscale version of the 3 slice matrix by taking the average of the 3 elements for a single entry to form the greyscale matrix `mtx`
It also calculates the SVD decomposition of `mtx`

```
nightingale_data = image.compress.param(4)
```

Entering `4` into this function will select the nightingale image and create the matrix for the greyscale image and will perform the singular value decomposition .

We are interested in minimizing the function $\exp(\|A - B\|_F) + k(m + n + 1)$
I will use the fact that:

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

Where $\tilde{A}_k$ is the kth rank approximation of the matrix A. e.g:

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

I am going to create a vector of length $\min\{m, n\} = 584$
which I will then fill with all the values from equation (1)

```
function_values = vector(length = 584)
```

Here `sum_of_d` is the sum of all the squared singular values
`cumsum_of_d` is the cumulative summation of the squared singular values

```
sum_of_d    = sum((nightingale_data$svd$d)^2)
cumsum_of_d = cumsum((nightingale_data$svd$d)^2)
```

Here I am assigning the correct values to `m` and `n` using the dimensions of the image matrix.

```
m = dim(images[[4]])[1]
n = dim(images[[4]])[2]

m_n_1 = m + n + 1
```

Here I am calculating all of the function values and entering them into the vector

```
for(i in 1:584){

  function_values[i] =  exp(sqrt(sum_of_d - cumsum_of_d[i])) + ((i)*(m_n_1))

}
```

Now I can find the minimum value that the function takes and can find the value of `k` that gave this minimum value.

```
min(function_values)
```

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

```
(1:n)[function_values == min(function_values)]
```

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

Therefore the **optimal value of k is 257**.

And referring to notation given in the assignment:
$c_1, ..., c_k$ will be the first k singular values (when ordered in decreasing order).
$d_1, ..., d_k$ are the first k columns of the right singular matrix.
$b_1, ..., b_k$ are the first k columns of the left singular matrix.