Quantum Computing and Quantum Information

Pradipta Parag Bora

April 2020

Contents

1 Introduction **1**

2 Mathematical Preliminaries **1**
2.1 Vectors and Vector Spaces 2
2.2 Linear Operators and Matrices 2
2.3 Inner Products 2
2.4 Eigenvalues and Eigenvectors 3
2.5 Adjoints and Hilbert Spaces 4
2.6 Operator Functions 5
2.7 Polar and Singular Value Decompositions 5

3 Introduction to Computer Science **7**
3.1 Turing Machines 7
3.2 Circuit Model of Computation 9
3.3 Analysis of Computational Problems 9

1 Introduction

Through this Summer of Science report, We attempt to summarise everything we covered related to Quantum Computing and Quantum Information Theory. We will mainly follow the book **Quantum Computation and Quantum Information** by **Nielsen and Chuang**. We will begin by going through the mathematical preliminaries, including an introduction to the linear algebraic treatment of Quantum Mechanics through which we will explore Quantum Computation. Then we will move onto qubits, their representation and on the way we will study and go through various fascinating Quantum algorithms and their uses. Hopefully through this report one will be able to gain a decent understanding of Quantum Computation.
The first two sections do not deal with quantum computation and quantum mechanics and instead provide a brief overview of the mathematics we will require and the basic ideas of computer science. We then move into the postulates of quantum mechanics and then we start with basic quantum circuits.

2 Mathematical Preliminaries

We assume that the reader is familiar with basic linear algebra that is covered during the first year. As our treatment of Quantum Computation heavily uses this, we will summarise some key theorems and notations that we will be using. The reader is free to skip this section as it merely summarises the mathematical tools that we will use. Also note that the following section is very direct and rarely provides any motivation behind the definitions and proofs to keep it concise and simply provides a list of the preliminaries. Readers are advised to refer to various references of linear algebra to get a better understanding.

2.1 Vectors and Vector Spaces

The most important structure in linear algebra is a vector which we will represent as $|\psi\rangle$. This vector resides in a vector space that satisfies some axioms that give this structure it's core defining properties. This is famously known as the *bra-ket* notation and the $|.\rangle$ is used to represent a vector. The same vector in the vector space $\mathbb{C}^n$ is conveniently represented as a column vector.
We will be generally working in the vector space $\mathbb{C}^n$ over the field $\mathbb{C}$. A set of vectors $|\psi_1\rangle, |\psi_2\rangle \cdots |\psi_n\rangle$ is said to be *linearly-independent* if for any set of coefficients a_i,

$$\sum_{i=1}^{n} a_i |\psi_i\rangle = 0 \implies a_i = 0 \forall 1 \leq i \leq n$$

A set of vectors $|\psi_i\rangle$ is said to be a spanning set of a vector space $\mathbf{W}$ if any vector in $\mathbf{W}$ can be represented as a linear combination of that set. Note that such a set may be non-finite however we will be dealing with finite dimensional vector spaces only. If this set is linearly independent then this set is called a *basis* of that vector space. It is left as an exercise to show that the cardinality of all bases are same and this value is termed as the *dimension* of that vector space.

2.2 Linear Operators and Matrices

A linear operator A from vector space $\mathbf{V}$ to $\mathbf{W}$ is a function mapping vectors from $\mathbf{V}$ to $\mathbf{W}$ satisfying linearity, that is:

$$A(\sum_i a_i |\psi_i\rangle) = \sum_i a_i A |\psi_i\rangle$$

We can then define compositions of operators, BA as function that first operates A and then operates B.
It is easy to see that matrix multiplication is a linear operator. Conversely any linear operator has a matrix representation. To see this, let $A : V \to W$ be a linear operator. Let $|v_i\rangle$ be an ordered basis (basis formed by fixing order of it's elements, in this case by enforcing an order restriction on $|v_i\rangle$.) of V and $|w_j\rangle$ be an ordered basis of W.
Then there exists complex numbers $A_{i,j}$ such that:

$$A |v_i\rangle = \sum_j A_{i,j} |w_j\rangle$$

If we represent any vector in V by it's coordinate vector representation, that is form a column vector whose elements with respect to an ordered basis are the scalar coefficients along the basis elements and multiply it by the matrix formed by $A_{i,j}$ we get the coordinate vector representation of the output vector in the ordered basis $|w_j\rangle$.

2.3 Inner Products

An inner product of a vector space $\mathbf{V}$ is defined as a function from $\mathbf{V} \times \mathbf{V}$ to $\mathbf{C}$. We will denote the inner product of $|v\rangle$ and $|w\rangle$ as $(|v\rangle, |w\rangle)$ or as $\langle v|w\rangle$. The inner product satisfies the following properties:

1. $\langle v|w\rangle = (\langle w|v\rangle)^*$
2. $\left\langle v \middle| \sum_j a_j |w_j\rangle \right\rangle = \sum_j a_j \langle v|w_j\rangle$
3. $\langle v|v\rangle \geq 0$ with equality *iff* $v = 0$

We can now define the *length* or *norm* of a vector $\||v\rangle\| = \sqrt{\langle v|v\rangle}$. Two vectors are said to be orthogonal if their inner product is 0. We now define an *orthonormal* basis as a basis $|v_i\rangle$ whose elements have unit norm and they are pairwise orthogonal, that is for $i \neq j \langle v_i|v_j\rangle = 0$. It can be proven that every finite dimensional vector space has an orthonormal basis. This is left as an exercise. The procedure for determining this orthonormal set from any basis is termed as the *Gram-Schmidt* orthogonalisation process.

This leads to a simple representation of the inner products of two vectors in matrix form. Let $|w\rangle = \sum_i w_i |i\rangle$ and $|v\rangle = |v\rangle = \sum_j v_j |j\rangle$ be two vectors written with the same orthogonal basis. Then $\langle w|v\rangle = (\sum_i w_i |i\rangle, \sum_j v_j |j\rangle) = \sum_i w_i^* v_i$ which we can nicely write in matrix form as

$$\begin{bmatrix} v_1^* & v_2^* & \cdots & v_n^* \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

We now introduce the outer product notation. Take two vectors $|w\rangle$ and $|v\rangle$ belonging to inner product spaces. Define the outer product between them to be the linear operator $|w\rangle\langle v|$ which operates on $\left|v'\right\rangle$ belonging to the same vector space as $|v\rangle$ yielding:

$$(|w\rangle\langle v|)\left|v'\right\rangle = \left\langle v\middle|v'\right\rangle |w\rangle$$

One use of this can be seen as follows:
Let $|i\rangle$ be an orthonormal basis of a vector space $\mathbf{V}$. Then for any vector $|v\rangle$ in this vector space, it is easy to see that

$$(\sum_i |i\rangle\langle i|)|v\rangle = |v\rangle$$

which in turn implies that

$$\sum_i |i\rangle\langle i| = I_v$$

This is termed as the completeness relation.
Using this we can easily get the matrix representation of an operator $A : \mathbf{V} \to \mathbf{W}$. Let $|v_i\rangle$ be an orthonormal basis for $\mathbf{V}$ and $|w_j\rangle$ be an orthonormal basis for $\mathbf{W}$

$$A = I_W A I_V = \sum_{i,j} |w_j\rangle\langle w_j| A |v_i\rangle\langle v_i|$$

The terms $\langle w_j| A |v_i\rangle$ are nothing but the entries of the matrix corresponding to this transformation written in coordinate vector form.

2.4 Eigenvalues and Eigenvectors

An eigenvector for a linear operator A is a non-zero vector $|v\rangle$ such that $A|v\rangle = v|v\rangle$ for some complex number v. This number is called the eigenvalue corresponding to this eigenvector and we will refer to both the eigenvalue and the eigenvalue by the same letter.
A sufficient and necessary condition for a number λ to be an eigenvalue is

$$det|A - \lambda I| = 0$$

We will term the subspace spanned by eigenvectors corresponding to the same eigenvalue as the eigenspace corresponding to that eigenvalue. Now we introduce diagonalizable operators. An operator is said to be diagonalizable on a vector space if there exists an orthonormal basis composed of eigenvectors of that operator. It is left as an exercise to show that such an operator A can be represented as:

$$A = \sum_i \lambda_i |i\rangle\langle i|$$

where $|i\rangle$ corresponds to the orthonormal eigenvectors and λ_i are the corresponding eigenvalues. We will now look at some conditions under which operators are diagonalizable. This is of importance to us as we will be using the diagonal representation of the operator many times later on.

2.5 Adjoints and Hilbert Spaces

For any linear operator A it can be proven that there is a unique linear operator $A^\dagger$ satisfying

$$(|w\rangle, A|v\rangle) = (A^\dagger|w\rangle, |v\rangle)$$

This operator is termed as the adjoint of A. It can be shown that the matrix representation of $A^\dagger$ is obtained by taking the complex conjugate and then the transpose of A or $A^\dagger = (A^*)^T$

Definition 2.1. An operator is said to be **Hermitian** if $A^\dagger = A$.

We proceed to show an example of an useful hermitian operator, the Projector. Let **W** be a d dimensional subspace of **V** which is k dimensional. Let $|v_1\rangle \cdots |v_d\rangle$ be an orthonormal basis of **W**. By Gram Schmidt process we can extend this to an orthonormal basis of **V**. The Projector $P : \mathbf{V} \to \mathbf{W}$ is defined as:

$$P = \sum_{i=1}^{d} |v_i\rangle \langle v_i|$$

Informally this projector projects vectors from **V** to **W** as it only leaves those components of the vector that are along the basis vectors which are in **W**. We leave some trivial exercises corresponding to the Projector below that can be done by following the definitions.

Exercise 2.1. Prove that P is Hermitian

Exercise 2.2. Prove that $P^2 = P$ and thus $P^\dagger P = P$

Definition 2.2. An operator is A said to be **normal** if $A^\dagger A = AA^\dagger$.

The following theorem is incredibly useful and we omit it's proof as it is rather involved:

Theorem 2.1. Spectral Theorem: An operator A is diagonalizable if and only if it is normal.

Now we talk about a special type of normal operators, Unitary operators. These operators are of particular interest to us as all single quantum gates can be represented as unitary operators.

Definition 2.3. An operator is A said to be **unitary** if $A^\dagger A = I$.

A special class of hermitian operators is of particular interest to us. These operators called Positive operators are useful as they come up in the polar and singular decompositions which we will discuss shortly.

Definition 2.4. An operator is A said to be **positive** if $(|v\rangle, A|v\rangle) \in \mathbb{R}^+$ for all $|v\rangle$

Please go through the following exercises related to positive operators.

Exercise 2.3. Prove that for any operator A, $AA^\dagger$ is positive.

Exercise 2.4. Prove that any positive operator is necessarily hermitian.

Solution. Let A be a positive operator. Observe that we can write A as:

$$A = \frac{A + A^\dagger}{2} + i\frac{A - A^\dagger}{2i}$$

Observe that $\frac{A+A^\dagger}{2}$ and $\frac{A-A^\dagger}{2i}$ are both hermitian operators. We will call them B and C from now on.

$$(|v\rangle, A|v\rangle) = (|v\rangle, B|v\rangle) + i(|v\rangle, C|v\rangle)$$

Observe that for any hermitian operator B,

$$(|v\rangle, B|v\rangle) \in \mathbb{R}$$

Thus it follows that $(|v\rangle, C|v\rangle) = 0$ for all $|v\rangle$ which implies $C = 0$. Thus $A = B$ and so A is hermitian.

2.6 Operator Functions

It becomes useful for us to define functions on operators that are normal. Formally if A is a normal operator and it's diagonal representation is as follows:

$$A = \sum_i v_i \left|v_i\right\rangle \left\langle v_i\right|$$

then we define

$$f(A) = \sum f(v_i) \left|v_i\right\rangle \left\langle v_i\right|$$

It is easy to see that $f(A)$ is uniquely determined which allows us to define things like square roots of positive operators, exponentials of normal operators etc.

Exercise 2.5. If A is a normal operator satisfying $A^2 = I$ then prove that

$$\exp(i\theta A) = \cos\theta I + i \sin\theta A$$

Solution. Observe that the possible eigenvalues are 1 and -1. Let $|v_j\rangle$ be an orthonormal basis of the eigenspace corresponding to the eigenvalue 1 and $|w_k\rangle$ be the orthonormal basis for the eigenspace corresponding to -1. Then,

$$A = \sum_j \left|v_j\right\rangle \left\langle v_j\right| - \sum_k \left|w_k\right\rangle \left\langle w_k\right|$$

From the definition of the exponential for normal matrices,

$$\exp\{i\theta A\} = \sum_j \exp\{i\theta\} \left|v_j\right\rangle \left\langle v_j\right| + \sum_k \exp\{-i\theta\} \left|w_k\right\rangle \left\langle w_k\right|$$

$$= \cos\theta(\sum_j \left|v_j\right\rangle \left\langle v_j\right| + \sum_k \left|w_k\right\rangle \left\langle w_k\right|) + i\sin\theta(\sum_j \left|v_j\right\rangle \left\langle v_j\right| - \sum_k \left|w_k\right\rangle \left\langle w_k\right|) = \cos\theta I + i\sin\theta A$$

where we have used the completeness relation and the diagonal decomposition of A

We briefly mention that the notion of trace for a matrix can be extended for operators. This is because the trace satisfies

$$tr(AB) = tr(BA)$$

for matrices A, B. Suppose $A^{'}$ is one matrix representation of A. Then it is related to another matrix representation by $A^{''}$ by

$$A^{'} = PA^{''}P^{-1}$$

for some change of basis matrix P. Clearly by the previous result $tr(A^{'}) = tr(A^{''})$ and so trace of an operator is independent of it's matrix representation.

2.7 Polar and Singular Value Decompositions

We omit the proof of the Polar decomposition as it is rather involved. The singular value decomposition follows from the polar decomposition.

Theorem 2.2. Polar Decomposition For any linear operator A on a vector space $\mathbf{V}$ there exists unitary U and positive J, K such that

$$A = UJ = KU$$

where

$$J = \sqrt{A^\dagger A}, K = \sqrt{AA^\dagger}$$

Theorem 2.3. Singular Decomposition For any square matrix A there exists unitary matrices M, N and diagonal matrix D with positive entries such that

$$A = MDN$$

Proof. By the polar decomposition, there exists unitary U and positive J such that

$$A = UJ$$

As J is positive, it is diagonalisable, so

$$J = T^\dagger DT$$

where the entries of D are the eigenvalues of J which are positive and T is unitary. Setting $M = UT^\dagger$ and $N = T$ completes the proof. □

We end the mathematical preliminaries at this stage and move onto the postulates of Quantum mechanics which will provide an axiomatic approach to quantum mechanics.

3 Introduction to Computer Science

Our main motivation behind using quantum computation is to provide an improvement over the classical model of computation. In order to do so we must have a proper notion of how we compare two different models of computation. Many classically difficult to solve problems have in principle quantum algorithms that solve these problems with ease. For instance the factorisation problem of finding the prime factors of a number N, takes exponentially more time classically than via the quantum shor's algorithm. But how do we quantify which algorithm is better or efficient? We will deal with this at the end of this section. We begin by first briefly discussing a model of computation that is deemed to be universal, the Turing model. We then provide a circuit equivalent of the same which is more feasible to us. We end by looking at energy requirements of computation and then we look at how we can have a fundamentally reversible model of computation and investigate the benefits of having so.

3.1 Turing Machines

The field of computer science arose back in the 1930s when the pioneers of the field Alan Turing and Alonzo Church formalized the notion of an algorithm used to solve mathematical problems by providing a model of computation. Turing's model, called the Turing Machine provides an abstract notion of a machine, that in principle can solve all problems associated with an algorithmic process. We will discuss it's construction now.
A Turing machine contains four main elements:

1. a program, rather like an ordinary computer which contains the instructions corresponding to the algorithm
2. a finite state control, which acts like a stripped-down microprocessor working with the other components and containing the states of the machine
3. a tape which is the memory/output of the machine
4. a read write tape head, which points to the current position on the tape which is being accessed.

The main component of the turing machine is the finite state control, which contains a collection of states of the machine and the machine works by trying to look for the instruction in it's collection that corresponds to the current state the Turing machine is in. Let the states of the machine be $q_1, q_2 \cdots q_m$ and two other special states q_s and q_h which correspond to the starting and halt state respectively. The machine stops operating whenever q_h is obtained.
The tape can be thought of as a linear sequence of squares where each element is either $0, 1, b$ or s where b corresponds to a blank space and s refers to the left most starting square on the tape. The tape will start with s and then contain some zeros and ones and finally have blanks on it. The read write tape head is responsible for pointing at the current square and overwriting it with the new value on the square as the machine moves onto the next square. This can be regarded as both the input and the output of the device.
Let us see how the turing machine works. We will illustrate this via an example. We'll show an explicit construction of a turing machine that outputs the function $f(x) = 1$. Firstly let us see how the program of the machine is defined. Formally a *program* in a turing machine is a list of tuples (which correspond to commands) of the form

$$\left(q_i, x, q_j, x^{'}, d\right)$$

where q_i is the current state in which the machine is supposed to be, x is the value on the current square in the tape head, q_j is the state the machine goes into after executing the current command, x' is the value that the machine will write on the current square after this command is executed and d is either $0, +1, -1$ where a value of 0 instructs the machine to not move the tape head, $+1$ means move the pointer to the right and -1 means move the tape to the left. The only exception to this rule is when the current value on the tape is s which would ignore the -1 instruction as we are at the beginning of the tape.
The machine will obtain it's current working condition by reading the value on the tape square and scan for the program line that has it's current state and this value as the first two elements of the tuple. If such a

line does not exist it automatically goes to q_h and halts execution.
Now we will list an explcit program for computing $f(x) = 1$ Consider the following program

1. $(q_s, s, q_1, s, 1)$
2. $(q_1, 0, q_1, b, 1)$
3. $(q_1, 1, q_1, b, 1)$
4. $(q_1, b, q_2, b, -1)$
5. $(q_2, b, q_2, b, -1)$
6. $(q_2, s, q_3, s, 1)$
7. $(q_3, b, q_h, 1, 0)$

We can see that this program outputs $f(x) = 1$ followed by a series of blanks. It is upto you to verify this by tracing the working of the machine line by line. Although it seems that we certainly did not require such a complex setup to simply compute $f(x) = 1$ the beauty of turing machines lie in the fact that this process can be used to compute functions ranging from addition, polynomial evaluation and arithmetic. Infact all operations that you can do on your current computer can be done on a turing machine. The efficiencies may be different but we are more interested in knowing which problems can indeed be solved on a turing machine. Church and Turing came up with their thesis, the Church Turing thesis that answers exactly this question.

Thesis. Church Turing Thesis: The class of functions computable by a Turing machine corresponds exactly to the class of functions which we would naturally regard as being computable by an algorithm.

It is to note that this is a hypothesis that has not been proven, however over the last 60 years no counter evidence for the same has been found. It is interesting to note that quantum computers, and in general, the quantum model of computation does not change the class of functions computable by the model, it merely provides a more efficient way to do so.
Showing that the turing model of computation universally reflects what it means to have an algorithm compute a function is beyond the scope of this report. We will now go through some variations of turing machines. For instance, instead of having just one single tape, the machine could contain two tapes wherein you can use one for inputs and other for outputs. This variation is identical to the simple turing machine in the sense that both can compute the same set of functions. However for explicitly writing programs for a turing machine it is indeed more convenient to work with the two tape version. A program line of this turing machine will be of the form $\left(q, x_1, x_2, q^{'}, x_1^{'}, x_2^{'}, d_1, d_2\right)$ where we will use subscript 1 for the first tape and 2 for the other tape.

Exercise 3.1. Write the program lines for a Turing Machine to compute the binary NOT of a binary number provided to you as input in a tape. (Hint: for such problems it makes more sense to use multiple taped turing machines and more symbols than just $0, 1, b, s$)

Exercise 3.2. Describe a Turing program to reverse a binary number provided to you as input in the tape.

Another version of a turing machine is a probabilistic turing machine. This machine will transition from one state to another by randomly choosing a state from various possibilities. Note that a deterministic turing maching can in essence simulate a probabilistic machine by going through all possibilites (searching through the search space). Thus they both can compute the exact same class of functions. The probabilistic machine may be more efficient but right now we are dealing with the classes of solvable problems.
Now we describe the circuit model of computation in brief.

3.2 Circuit Model of Computation

The circuit model of computation is essentially the model we are most familiar with, wires and gates that compute binary functions. Taking multiple inputs we can actually compute all kinds of functions which is equivalent to those which can be simulated on a turing machine. In general our circuits will have gates, which are machines that compute functions of the form $f : \{0,1\}^k \to \{0,1\}^l$. However we would like to reduce all our circuits to those made up of some universal blocks. There are a few gates called the universal gates in classical computation that can build up any gate. One such gate is the NAND gate which takes two binary inputs and computes their binary and.
There are other universal gates as well and some common gates like OR, AND, XOR which are self explanatory. Two other gates are the FANOUT and Crossover gates. The FANOUT gate takes an input and copies it out. We will later see that the FANOUT gate has no quantum equivalent because of the *no-cloning* theorem. The Crossover gate takes two inputs and simply swaps them.
Later on when we get into quantum circuits, we will show quantum equivalents of these gates and also look at universal quantum gates. It is important to note that quantum mechanics is reversible, infact all of quantum mechanics can be represented as unitary transformations which we will look into in the next section. As a result it is not possible to have irreversible gates in our quantum computational model. So a quantum variant of an AND gate is not possible as AND is irreversible which means that given the output of the AND gate we cannot. We reserve this discussion for later when we start dealing with quantum circuits.

3.3 Analysis of Computational Problems

So far we have said that quantum computers provide a more efficient solution of various classical problems. How do we deem which algorithm is better? This section deals with that.
First we talk about orders of functions. Let us take an algorithm which scans a list of n numbers and outputs the maximum among them. Clearly such an algorithm takes n operations to finish. In general lesser the number of steps the better the algorithm. But we cannot really talk about the exact number of steps and instead we need an estimate on the number of steps as this value can be variable. Suppose we have an algorithm that scans a list of n numbers to check if 1 is present or not and if it finds 1 it terminates. Clearly this can take any number of steps ranging from 1 to n. What we know is that at worst it takes n steps (the upper bound) and in the best case it takes 1 step (lower bound). So the ideas of upper bound and lower bound on the number of steps provide a rough estimate of how good the algorithm is with average case analysis provide a more better picture. To talk about such cases in a more formalised manner we use the big-O, big-Omega and big-theta notation.

Definition 3.1. Big O A function $f(n)$ is said to be $O(g(n))$ if there exists some constant c and some n_0 such that for all

$$n \geq n_0, f(n) \leq cg(n)$$

An algorithm A is said to have (upper) time complexity $O(g(n))$ is the function $\mathbf{TIME}(A(n)) = f(n)$ is $O(g(n))$ where $f(n)$ is the number of steps the algorithm takes on that input. For example the previous algorithm is $O(n)$ as for $c = 1$ and $n_0 = 1$ $f(n) \leq n$ is satisfied where n is the size of our input.
Similarly we have big theta and big O notations that deal with different bounds.

Definition 3.2. Big Theta A function $f(n)$ is said to be $\Theta(g(n))$ if there exists some constant c_1, c_2 and some n_0 such that for all

$$n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Definition 3.3. Big Omega A function $f(n)$ is said to be $\Omega(g(n))$ if there exists some constant c and some n_0 such that for all

$$n \geq n_0, cg(n) \leq f(n)$$

So for instance the previous algorithm is $\Omega(1)$ and $O(n)$. It is upto you to prove that we cannot have any Θ bound for this algorithm. We can also have space complexity where instead of talking about the number of steps, we refer to the number of memory units the program requires. So for instance the previous algorithm has $\Theta(n)$ space complexity as it is using roughly n memory units to store all values (measured as multiples of memory units used to store one number).

Now let us compare the algorithms for prime factorisation. If a number N is given the size of it's input is $\log_2 N$. Remember that computer scientists always deal with logarithms to the base 2 as values are stored in binary. So the implicit base is 2. Now Shor's algorithm is $O(\log N^3)$ which is a massive improvement over classical algorithms. A simple trial algorithm is $O(N)$ which becomes exponential in the input size $\log_2 N$. Notice that we are only concerned with the behaviour of the function at large values of the input as computation becomes infeasible only at those limits.

Exercise 3.3. Consider the Merge Sort algorithm. This algorithm can be defined in psuedocode as follows in recursive form (self calling):

```
MERGESORT(List A, SIZE N):
    List B = List A[0, N/2]
    List C = List A[N/2 + 1, N]
    B = MERGESORT(B, N/2)
    C = MERGESORT(C, N/2)
    A = MERGE(B, C, N/2, N/2)
    return A
```

The merge function combines the two lists B and C which have half the number of elements that are already sorted and outputs the sorted list formed by combining B and C.
1. Give an $O(N_1 + N_2)$ complexity algorithm for MERGE(List A, List B, Size $\mathrm{N}_1, SizeN_2$).
2. Using the above algorithm write the recursive definition for **TIME**(List A, Size N) in terms of N. Use this to compute the time complexity of Merge Sort.