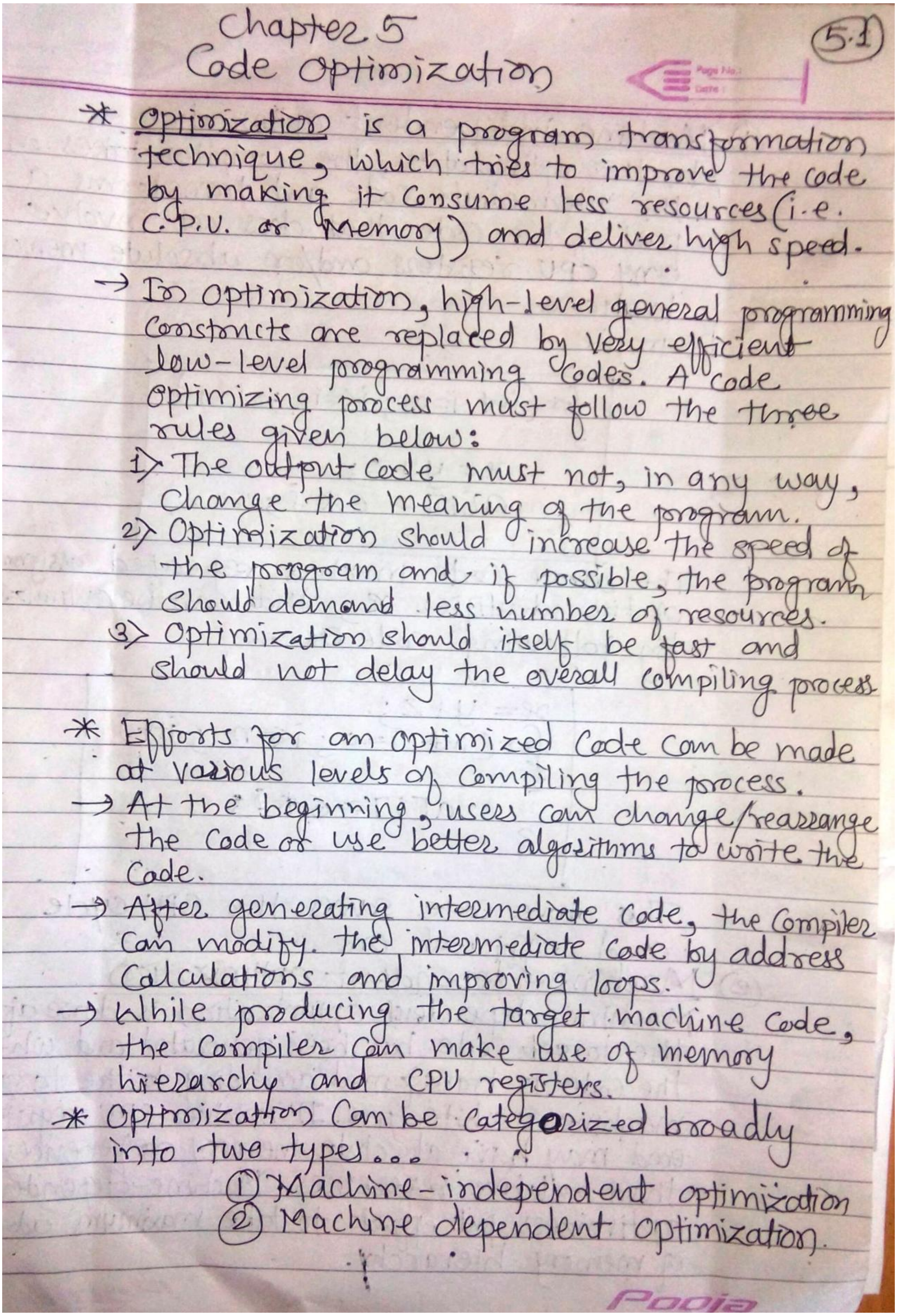
Chapter 5 Code Optimization……………………………………………………..



Chapter 5
Code optimization

5.1

* Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. C.P.U. or memory) and deliver high speed.

→ In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

1> The output code must not, in any way, change the meaning of the program.

2> Optimization should increase the speed of the program and, if possible, the program should demand less number of resources.

3> Optimization should itself be fast and should not delay the overall compiling process

* Efforts for an optimized code can be made at various levels of compiling the process.

→ At the beginning, users can change/rearrange the code or use better algorithms to write the code.

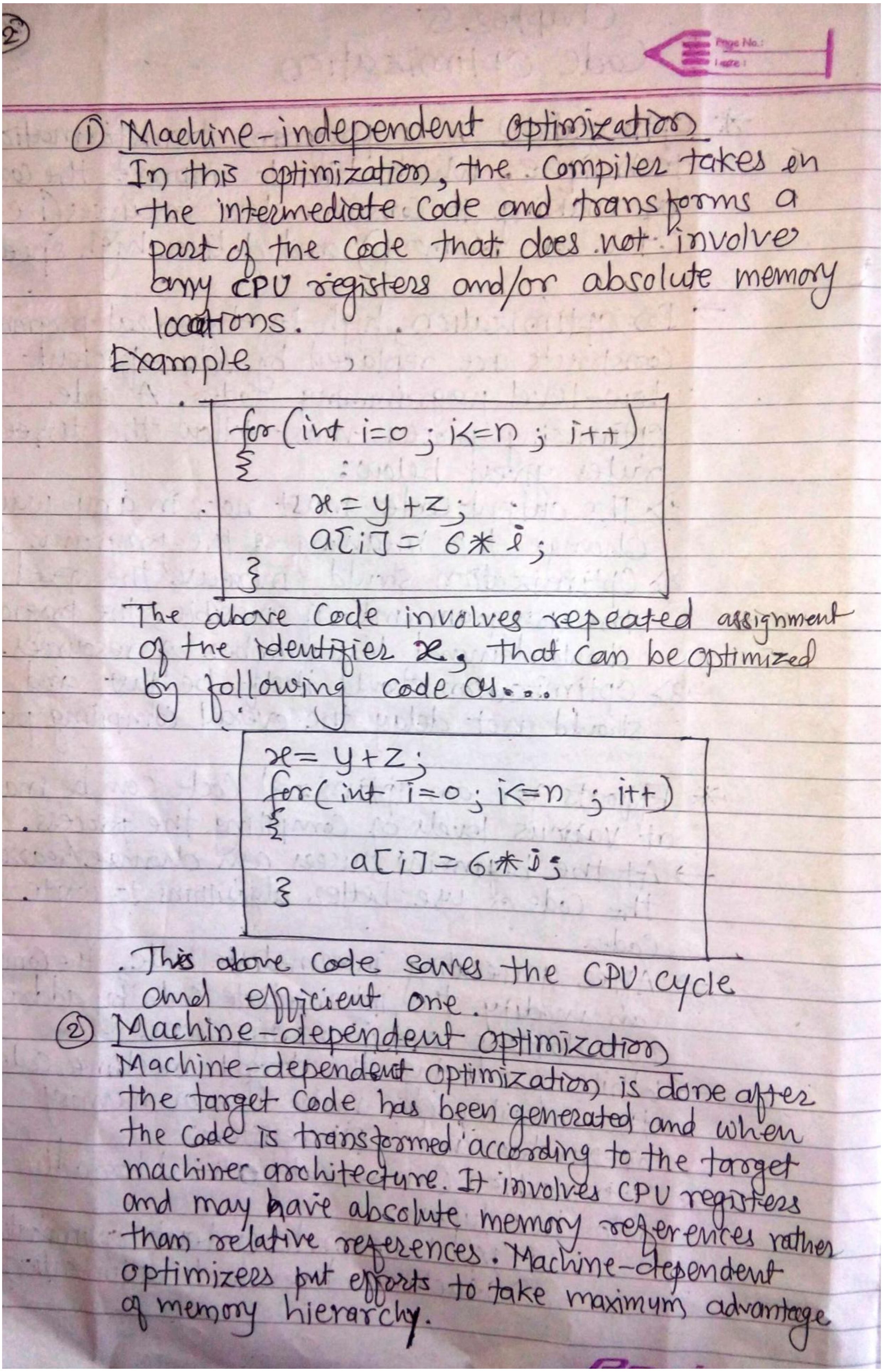
→ After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.

→ While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

* Optimization can be categorized broadly into two types...

① Machine-independent optimization

② Machine dependent optimization.



2

Page No.:
Date:

① Machine-independent optimization

In this optimization, the compiler takes en the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations.

Example

```
for(int i=0 ; i<=n ; i++)
{
      :
    x = y+z ;
    a[i] = 6*i ;
}
```

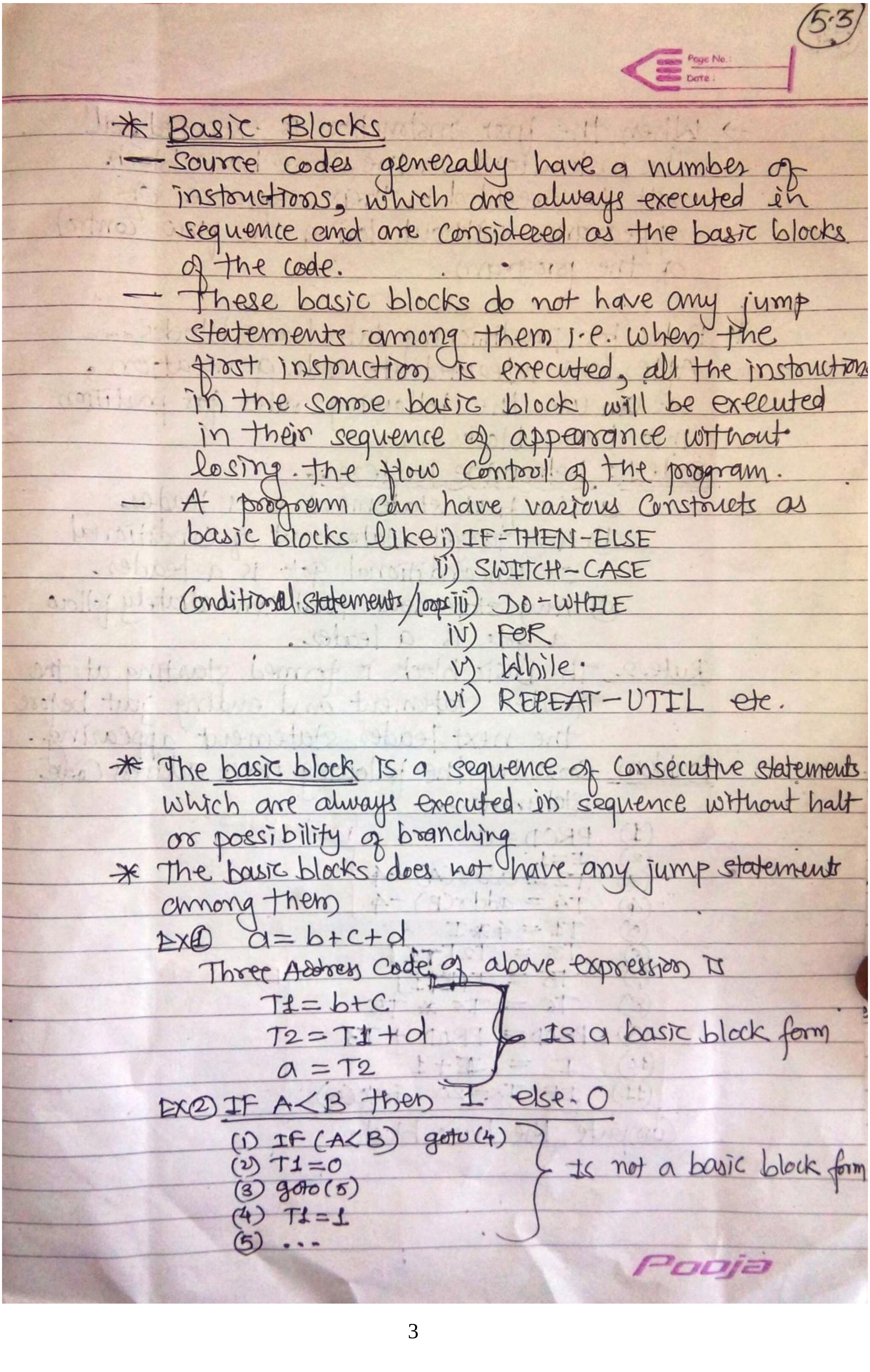
The above code involves repeated assignment of the identifier x, that can be optimized by following code as....

```
x= y+z;
for(int i=0 ; i<=n ; i++)
{
      a[i]=6*i;
}
```

This above code saves the CPU cycle and efficient one.

② Machine-dependent optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machiner architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.



* Basic Blocks

— Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code.

— These basic blocks do not have any jump statements among them i.e. when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

— A program can have various constructs as basic blocks like i) IF-THEN-ELSE
ii) SWITCH-CASE
Conditional statements/loops iii) DO-WHILE
iv) FOR
v) While.
vi) REPEAT-UTIL etc.

* The basic block is a sequence of consecutive statements which are always executed in sequence without halt or possibility of branching

* The basic blocks does not have any jump statements among them

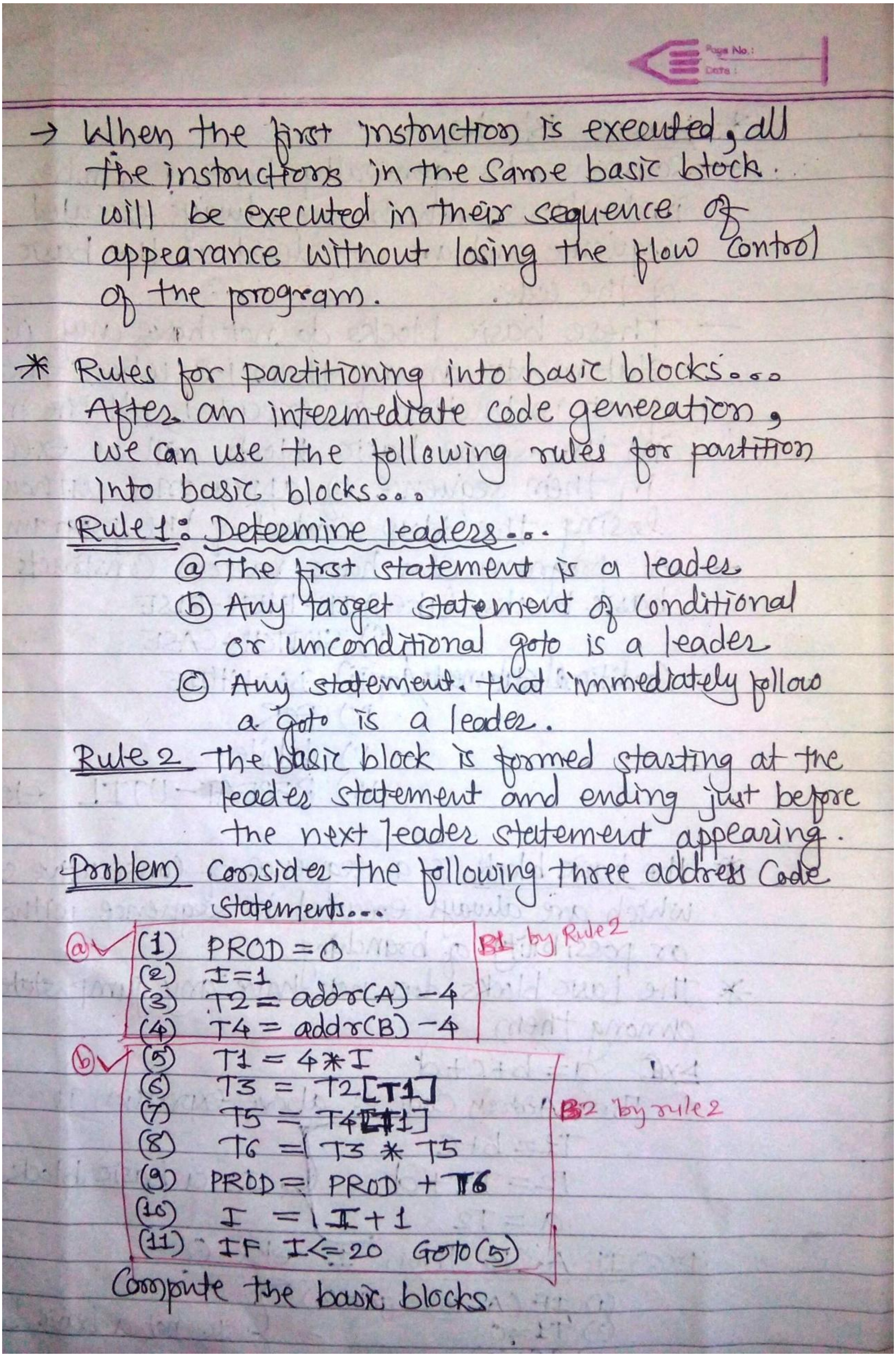
Ex① a = b+c+d

Three Address Code of above expression is

```
T1 = b+c
T2 = T1 + d     } Is a basic block form
a = T2
```

Ex② IF A<B then 1 else 0

```
(1) IF (A<B) goto(4)
(2) T1=0
(3) goto(5)          } is not a basic block form
(4) T1=1
(5) ...
```



→ When the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

* Rules for partitioning into basic blocks...
After an intermediate code generation, we can use the following rules for partition into basic blocks...

Rule 1: Determine leaders...

ⓐ The first statement is a leader

ⓑ Any target statement of conditional or unconditional goto is a leader

ⓒ Any statement that immediately follow a goto is a leader.

Rule 2 The basic block is formed starting at the leader statement and ending just before the next leader statement appearing.

Problem Consider the following three address Code statements...

```
ⓐ✓ (1)  PROD = 0               B1 by Rule2
    (2)  I = 1
    (3)  T2 = addr(A) - 4
    (4)  T4 = addr(B) - 4
ⓑ✓ (5)  T1 = 4 * I
    (6)  T3 = T2[T1]
    (7)  T5 = T4[T1]           B2 by rule2
    (8)  T6 = T3 * T5
    (9)  PROD = PROD + T6
    (10) I = I + 1
    (11) IF I<=20 GOTO(5)
```

Compute the basic blocks.

Solution • Because first statement is a leader, so—
PROD=0 Is a leader

• Because the target statement of conditional or unconditional goto is a leader, so—
T1= 4 * I is also a leader

So, the given Code Can be partitioned into 2 blocks as

```
PROD=0
 I=1
 T2 =addr(A) -4
 T4 = addr(B) -4          B1

T1 = 4 * I
T3 = T2[T1]
T5 = T4[T1]
 T6 = T3 * T5              B2
PROD = PROD + T6
  I = I+1
IF I<=20  Goto B2
```

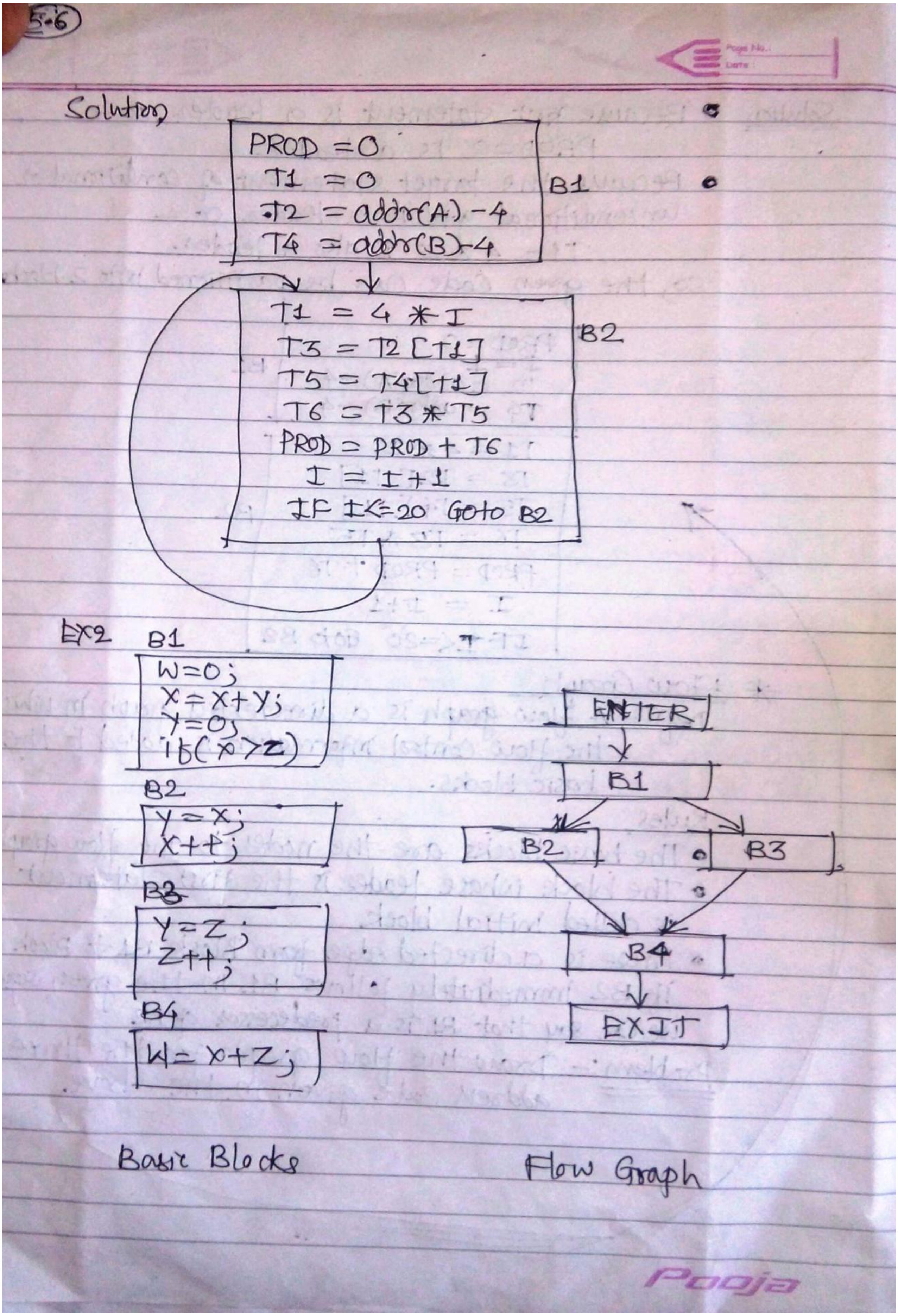
* **Flow Graph**

Defn :- A flow graph is a directected graph in which the flow control information is added to the basic blocks.

Rules

- The basic blocks are the nodes to the flow graph
- The block whose leader is the first statement is called initial block.
- There is a directed edge from Block B1 to Block B2 if B2 Immediately follows B1 in the given sequence, we Can say that B1 is a predecessor of B2

Problem :- Draw the flow graph for the three address Code given in the above.



5-6

Solution

```
PROD = 0
 T1  = 0
 T2  = addr(A) - 4
 T4  = addr(B)-4
```
B1

```
T1 = 4 * I
T3 = T2 [T1]
T5 = T4[T1]
T6 = T3 * T5
PROD = PROD + T6
  I = I+1
IF I<=20 Goto B2
```
B2

EX2

B1
```
W=0;
X=X+Y;
Y=0;
if(X>Z)
```

B2
```
Y=X;
X++;
```

B3
```
Y=Z;
Z++;
```

B4
```
W=X+Z;
```

Basic Blocks

Flow Graph

Code optimization is a technique which tries to improve the code by eliminating unnecessory Code lines and arranging the statements in such a sequence that speed up the program execution without wasting the resources.

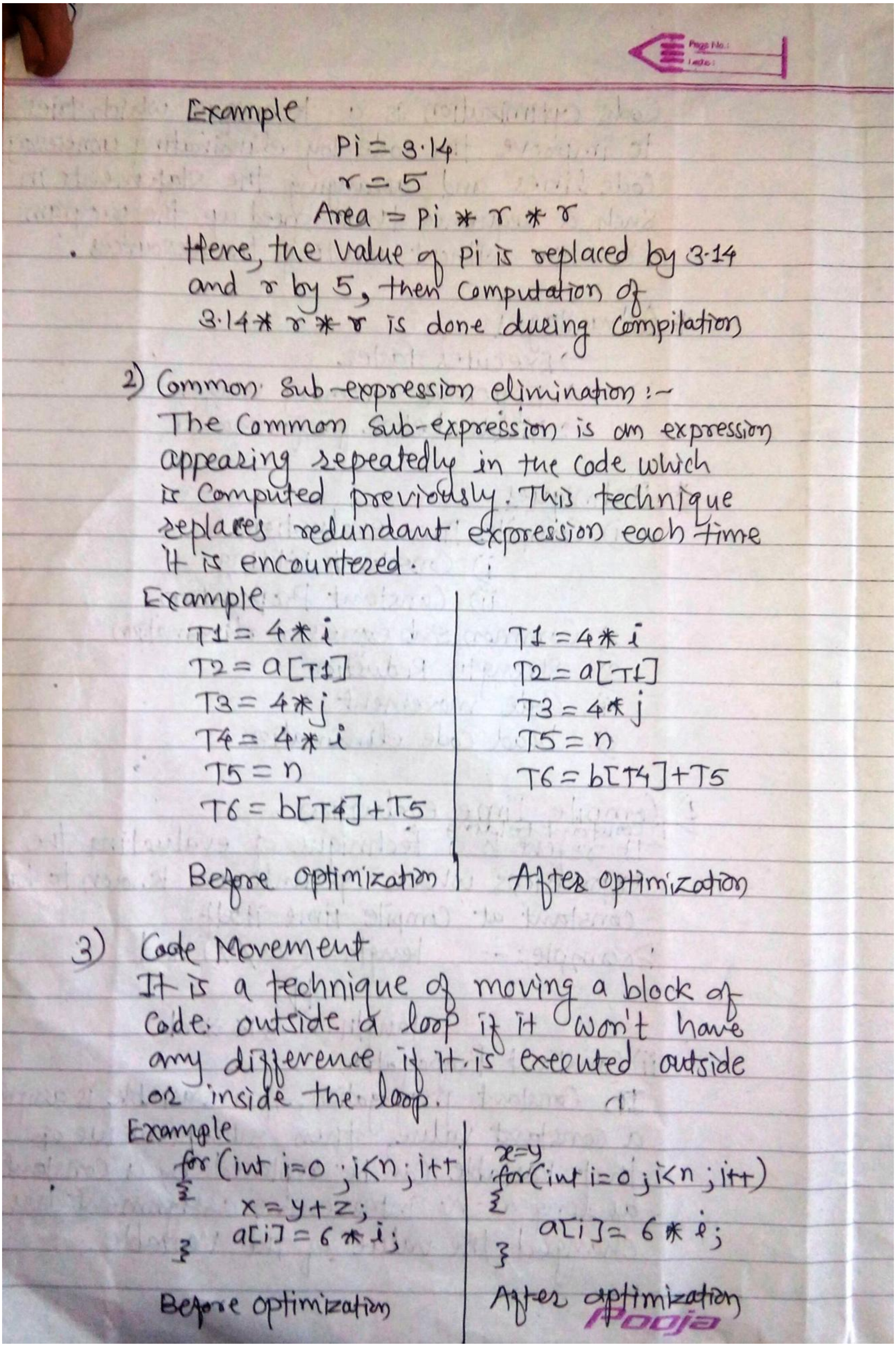
* Advantages
→ Executes faster
→ Efficient memory usage
→ Yields better performance

* Techniques
1) Compile Time evaluation
i) Constant folding
ii) Constant Propogation
2) Common sub expression elimination
3) Strength Reduction
4) Code movement
5) Dead code elimination

1) Compile Time evaluation
i) Constant folding → It refers to a technique of evaluating the expressions whose opeands are known to be constant at compile time itself.

Example :- length = (22/7) *d
⇓
length = 3.14 * d

ii) Constant Propogation
~~In~~ Constant propogation, if a variable is assigned a constant value, then subsequent use of that variable can be replaced by a constant as long as no intervening assignment has changed the value of the variable



Example

Pi = 3.14
r = 5
Area = pi * r * r

- Here, the value of pi is replaced by 3.14 and r by 5, then computation of 3.14 * r * r is done dueing compilation

2) Common sub-expression elimination :-

The Common sub-expression is an expression appearing repeatedly in the code which is computed previously. This technique replaces redundant expression each time it is encountered.

Example

```
T1 = 4*i
T2 = a[T1]
T3 = 4*j
T4 = 4*i
T5 = n
T6 = b[T4]+T5
```

Before optimization

```
T1 = 4*i
T2 = a[T1]
T3 = 4*j
T5 = n
T6 = b[T4]+T5
```

After optimization

3) Code Movement

It is a technique of moving a block of code outside a loop if it won't have any difference if it is executed outside or inside the loop.

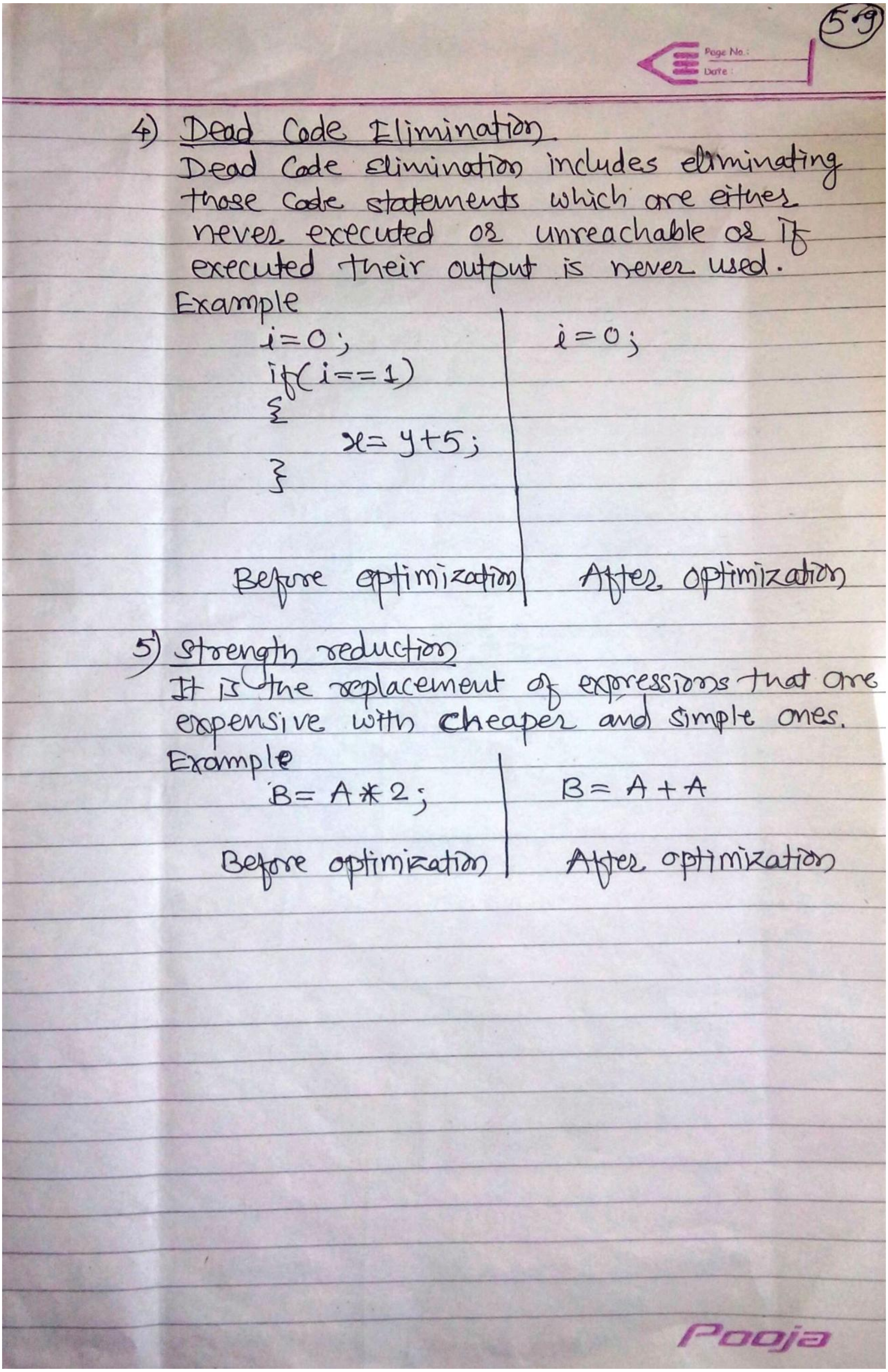
Example

```
for (int i=0 ; i<n ; i++
{
    x = y+z;
    a[i] = 6 * i;
}
```

Before optimization

```
x=y
for(int i=0 ; i<n ; i++)
{
    a[i] = 6 * i;
}
```

After optimization



4) Dead Code Elimination

Dead Code Elimination includes eliminating those code statements which are either never executed or unreachable or if executed their output is never used.

Example

i=0;
if(i==1)
{
x= y+5;
}
	i=0;

Before optimization	After optimization

5) Strength reduction

It is the replacement of expressions that are expensive with cheaper and simple ones.

Example

B= A*2;	B= A+A
Before optimization	After optimization