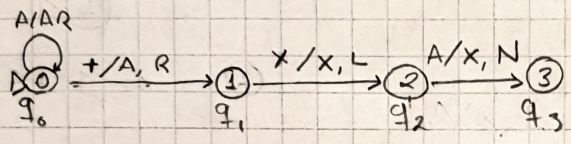
PROBLEM 1.1 TURING MACHINE

1. Two integers n_1, n_2 in unary form. The repeated character: A.
Blank: X and Special Character: +
When $n_1 = 3$ and $n_2 = 4$ the input is
AAA + AAAAXXX... and the result AAAAAAA + XXX...
A explicit TM that retrieves a sum in unary form, of two numbers assuming 'A' as alphabet character

ALGORITHM:

1. Read the symbols of the first input, make no replacements and move to the right.
2. When reaching the '+' special character replace it with 'A' and move to the right.
3. Read the symbols of the right until the rightmost A (left to the first 'X'.
4. Replace the rightmost A with 'X'
5. Stop the machine.

TRANSITIONS DIAGRAM



2. Scaling of maximal number T(n) of execution steps for an input of length n as a function of n: is it $O(n)$, $O(n^2)$ or exponential?

Big-O notation is used to measure the running time of an algorithm in terms of efficiency, considering that, the time it takes to compute a function grows as the size of the input grows.

Given an array of n elements, we want to write a function that takes the array and returns the sum of those elements,

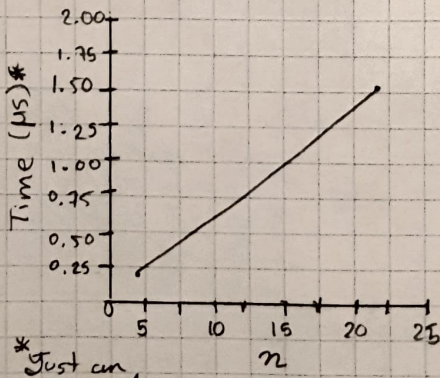
A pseudocode for this would be:

```
def function(array):
    total = 0
    for each i in array:
        total += i
    return total
```

And what we want is how does the runtime of this function grow? And that will depend on the number of elements n.

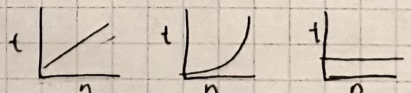
The best way to visualize this, is with a graph.

→ Time it takes to execute depending ← the input size (n)



*Just an example.

A straight line pattern corresponds to the complexity behavior of the question.

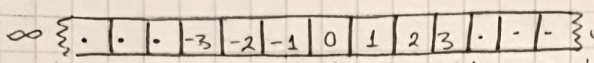


The time it takes for a sum of integers in unary form can be expressed as a linear function where: $T = an + b$; a, b are constants
n is size of input.

The fastest growing term is 'an' since 'b' doesn't change as 'n' increases, the constant 'a' is eliminated and we're left with 'n'. That gives us the big-O notation: $O(n)$

1.2 BI-INFINITE TURING MACHINE

1. To simulate a bi-infinite TM we need a two-tape TM (already proven to be equally powerful as an ordinary TM).
An bi-directional infinite TM tape looks like:

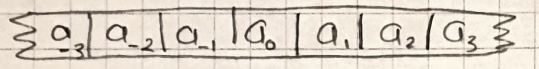


We can simulate it by creating a two-tape TM with a positive and negative index track.

0	1	2	3	...
Δ	-1	-2	-3	...

↳ In which 'Δ' indicates the transition, we can also use a 'turnaround marker' commonly denoted as '#'.

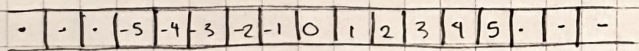
Each tape has an independent head, in one step, the state and symbol determine the next step. This form of simulation is from the theorem:
» FOR EVERY 2-WAY INFINITE TAPE TM M', THERE'S A STANDARD TM M' SUCH THAT L(M)=L(M') «



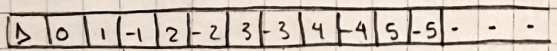
(a) is the symbol scanned.

Δ	a_0	a_1	a_2	a_3	a_4	a_5	a ...
	a_{-1}	a_{-2}	a_{-3}	a_{-4}	...		

Another way of simulation is the folding structure:
Bi-infinite tape TM



Right-infinite tape TM



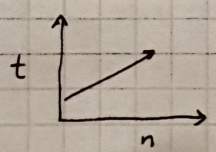
If positive: R → RR & L → LL At 0: R → R & L → L
If Negative: R → LL & L → RR

2. The correspondance in steps is linear, if an algorithm requires $O(p(n))$ steps on bi-infinite TM, with $p(n)$ polynomial, the same algorithm can run in $O(p(n))$ on a right-infinite TM

→ If we consider an algorithm requiring $O(p(n))$ steps on a bi-infinite TM where n=size of input, T=number steps TM takes to complete the algorithm The machine can read up to 2n symbols from the tape since it goes left and right, the total of symbols read by TM is $O(n)$

→ In a right-infinite TM the simulation will require the TM to move the special symbol to the left every time it reaches the left end of the tape and the total number of steps on the right-infinite TM is $T + O(n)$.

→ T is polynomial in n, $O(n)$ is polynomial in n. The number of steps taken by a TM on the right-infinite tape is $O(p(n))$. The correspondance is linear.



1.3 A SATISFIABILITY PROBLEM

Logic Boolean expression ϕ

$$\phi(x_1, x_2, \ldots, x_n) = \bigwedge_{\{i,j = 1,\ldots,n\}} (\tilde{x}_i \vee \tilde{x}_j)$$

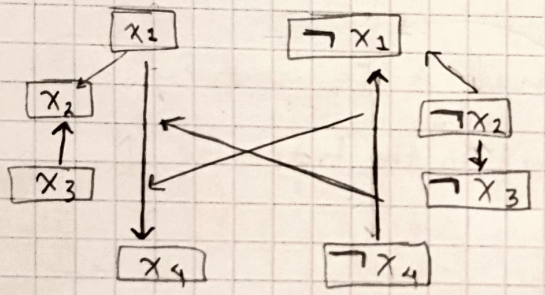
$\tilde{x}_i$ indicates x_i or $\neg x_i$

→ $\phi(x_1, x_2, x_3, x_4) = (\neg x_1 \vee x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_4 \vee \neg x_1) \wedge (x_2 \vee x_3)$

1. Draw a graph $G(\phi)$ for the logical expression ϕ

(α, β) → Directed edge
α and β are connected only if $(\neg\alpha \vee \beta)$ or $(\beta \vee \neg\alpha)$ are presented in ϕ.

$G(\phi)$ vertices correspond to x_i and $\neg x_i$ in ϕ.

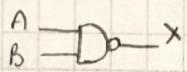


1. NAND is the combination of NOT and AND. If two inputs for NAND are both 1, the output is 0, otherwise, the input is 1

Boolean expression.

$X = (A \cdot B)'$

LOGIC DIAGRAM

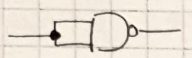


TRUTH TABLE

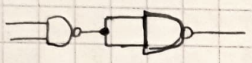
A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

It's universal because all other gates (NOT, AND, OR, NOR) can be created with this gate and thus provide an inverter.

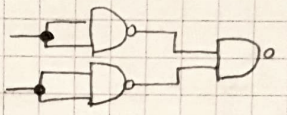
NOT



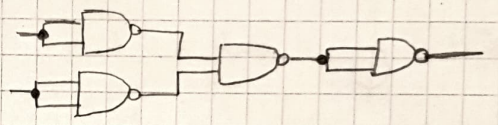
AND



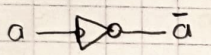
OR



NOR



NOT



input	out
0	1
1	0

input
0
1

0_1	0_2	output
0	0	1
0	1	1
1	0	1
1	1	0

AND

input	input	output (AND)	output (NAND)
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	1

XOR

input	input	output XOR	output (NAND)
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	1

NOR

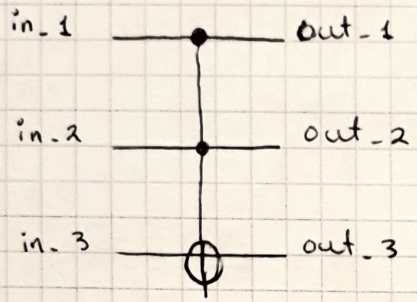
input	input	output NOR	output NAND
1	1	0	1
1	0	0	0
0	1	0	0
0	0	1	1

2.

TOFFOLI GATE.

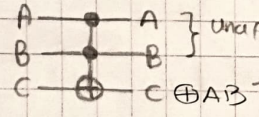
in_1	in_2	in_3	out_1	out_2	out_3
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

→ Reversibility can be observed in the truth table. If the two control inputs are both 1 then the 3rd input is reversed.



→ Show that TOFFOLI is reversible and universal.

TOFFOLI gate takes 3 inputs A, B, C of which it returns A, B uncharged and flips C only if both A and B are true.



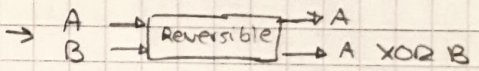
→ REVERSIBILITY.

A and B are preserved.

When we apply the Toffoli gate twice to a set of bits has the effect:

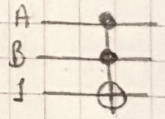
$(a, b, c) \rightarrow (a, b, c \oplus ab) \rightarrow (a, b, c)$

Toffoli gate is reversible because it has an inverse.

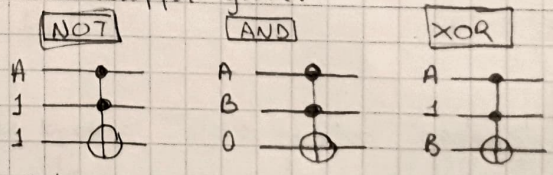


→ UNIVERSALITY

We can make a NAND gate from it

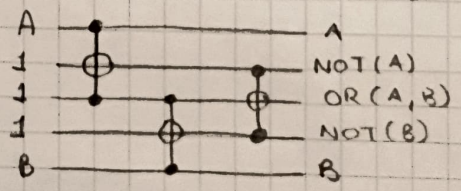


NAND gate helps us implement every conceivable circuit, so we can implement NOT(A), AND(A,B), XOR(A,B) using a single Toffoli gate.

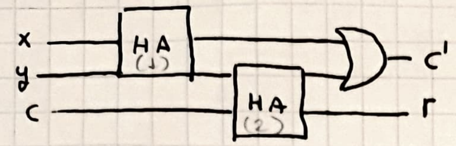


Using these we can make an OR gate from three Toffoli gates, because

$A \vee B = \neg(\neg A \wedge \neg B)$

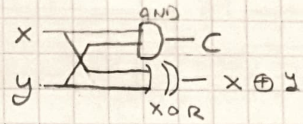


1.6 A CLASSICAL ADDER.



Full-adder circuit.

1. Each half-adder (HA) is :



The truth table for half-adder is

x	y	c	$x \oplus y$
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

'r' represents $x \oplus y \oplus c$, to prove it we need to build a truth-table for the full-adder taking x, y, c as inputs, the HA as other iputs and OR as the last output.

x	y	c	out_1 (HA 1)	out_2 (HA 1)	out_1 (HA 2)	out_2 (HA 2)	out_1 'r' (OR Gate)	out_2 'r' (OR Gate)
1	1	1	1	0	1	0	1	1
1	1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	1	0
1	0	0	0	1	0	0	0	1
0	1	1	0	1	0	1	1	0
0	1	0	0	1	0	0	0	1
0	0	1	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0

$x \oplus y \oplus c$

2. Two numbers x, y and a binary representation of each written:
$x = x_2 x_1 x_0$ $y = y_1 y_0$ $x = \sum_i x_i 2^i$

→ What is the smallest number of bits required to represent: x, y, x+y ?

First we need the binary/decimal representation of x, y.

$x = x_2 x_1 x_0$

x_2	x_1	x_0	Decimal
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

$y = y_1 y_0$

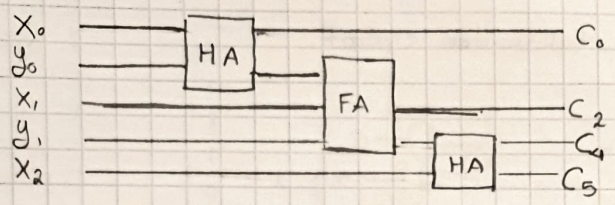
y_1	y_0	Decimal
0	0	0
0	1	1
1	0	2
1	1	3

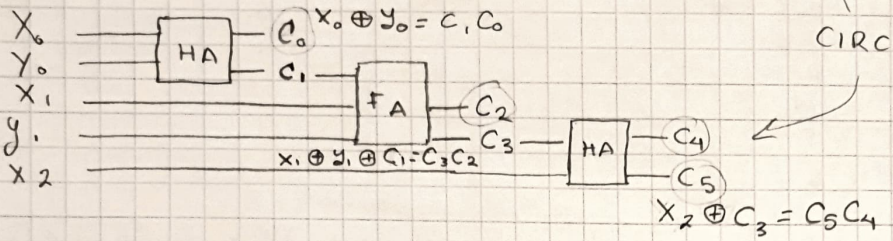
The smallest number of bits to represent x is 1, x has value 000 as binary on the bit 1 and has value 0 on decimal which is the smallest unit, y follows the same logic. For x + y we will have $x_2 x_1 x_0 + y_1 y_0$, therefore:

$x_2 x_1 x_0 + y_1 y_0 = x + y$ → 111 + 11 = 1010 (Binary) = 7 + 3 = 10 (Decimal)

10 is the biggest decimal value we get so now the number of bits required to represent it is 4 (1010), we need at least 4 bits, less bits would make impossible to perform the computation.

3. Circuit for x+y knowing $x = x_2 x_1 x_0$, $y = y_1 y_2$, for this purpose we're gonna establish 'c' as an output.





$X_0 \oplus Y_0 = C_1 C_0$

$X_2 \oplus C_3 = C_5 C_4$

CIRCUIT

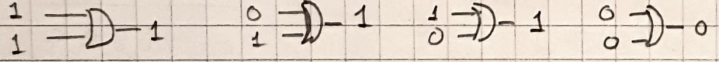
OUTPUT: $C = C_5 C_4 C_2 C_0$

1.7 REVERSIBLE COMPUTING

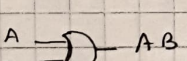
1.

in_1	in 2	output
0	0	0
0	1	1
1	0	1
1	1	1

This is an OR gate, outputs 1 when one of the inputs is 1, otherwise outputs 0



2. A logic gate is called reversible if we can construct the input when knowing the output.



A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

→ For this to be reversible we need:

	output A	output B	input
(1)	0	0	0
(2)	1	0	1
(3)	0	1	1
(4)	1	1	1

↳ Reference numbers

(1) and (4) can be duplicated with Fanout gates but for (2) and (3) we would need ancilla bits, and this don't work as gates.

Since not all inputs have logic gates that we could apply to make it reversible, the entire gate is not reversible.

3. $|a\ b\ c\rangle \mapsto |a\ b\ c \oplus (a+b)\rangle$ where $a, b, c \in \{0,1\}$

a	b	c	a	b	$c \oplus (a+b)$ (XOR, OR)
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	1	0

For $f: \{0,1\}^k \to \{0,1\}^l$ to be reversible we need to find $\tilde{f}: \{0,1\}^{k+l} \to \{0,1\}^{k+l}$ such that $\tilde{f}(x, y) = (x, y \oplus f(x))$

continue...

... 3, 4 ⑦

$|000\rangle \mapsto |000\rangle$
$|001\rangle \mapsto |001\rangle$
$|010\rangle \mapsto |011\rangle$
$|011\rangle \mapsto |010\rangle$
$|100\rangle \mapsto |101\rangle$
$|101\rangle \mapsto |100\rangle$
$|110\rangle \mapsto |111\rangle$
$|111\rangle \mapsto |110\rangle$

OR matrix (A) =

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

In this case the det(A) is different from zero ($\det(A) = -1$) so the inverse of the matrix exists.

The inverse is

$$A^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

The conjugate transpose

$$A^{*} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Since $A = A^{-1} = A^{*}$, we can say that A is unitary

$U_f = U_f^{-1} = U_f^{*}$ → Inverse equal to the transpose.