Contents

1. THE TURING MACHINE

1.1. Brief History

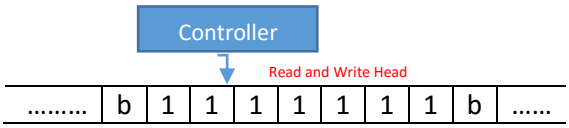
1936 Introduced by Alan M. Turing to solve computational problems. The Turing machine is the foundation of modern computers.

We will consider a highly simplified Turing machine.

1.2. The components of the Turing Machine

A Turing machine has three components: a **tape**, a **controller** and a **read/write head**. The memory of a Turing machine is assumed to be infinite. This contrast with the modern computers which use a Random Access Memory which has finite capacity.

Table 1The Components of the Turing Machine



The tape contains at any time characters from the sequences accepted by the machine. The tape of a simple machine we are considering would only hold either 1's or blanks denoted by **b**.

Table 2The Tape in the Turing Machine

……	b	1	1	1	1	1	1	1	b	……

Tape

The tape starts from the left. An integer is represented by a sequence of 1's and the rest of the tape contains blanks. In the Table 1 the tape holds the number 7 made up of seven 1's.

The Read/Write head points at only one position on the tape at any given time. This position holds the *current symbol.* After reading or writing the head moves to the left or to the right. Reading, writing and moving are done under the instructions of the controller.

The controller is the counterpart of the CPU unit of the modern computers. A machine that has a finite number of states and moves from one state to another based on the input is called a Finite State Automaton (the controller may have more states that the automaton). The move from one state to another is defined by a **'change function'.**

Example 1.1

Consider a FSA with three states, State A, State B and State C. And the following change function. The expression x/y/L, x/y/R and x/y/N means the following for the read/write head: if you read x, write y and wither move to the left (L), or move to the right R or do not move N.

In our example there are only two symbols, hence there are two paths out of each state: either 1 is read or blank is read.

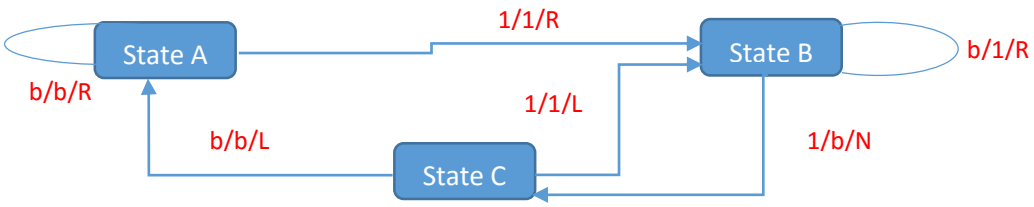


Figure 1 Transition state diagram for the Turing machine

In the transition state diagram, the beginning of the lines between states, called "transition lines", show the current state and the end of the line shows the next state. Corresponding to the diagram we can create a transition table:

Table 3 Transition Table

Current State	Read	Write	Move	New State
A	b	b	R	A
A	1	1	R	B
B	b	1	R	B
B	1	b	N	C
C	b	b	L	A
C	1	1	L	B

We have six instructions: (A,b,b,R,A), (B,b,1,R,B), etc.

1.3. Simulating a Simple Language on a Turing Machine

Consider the following simple language. The only data type is non-negative integers and there are a few symbols '{'} and '}'. We have three loop statements: the **increment** statement, the **decrement** statement, and the **loop** statement. The increment adds 1 to a variable.

incr (X) **decr(X)** **while(X) { decr (X**) body of the loop **}**

A macro is a sequence of one or more instructions that are grouped together under one name, the macro name, and can be called in other simulations/codes by using the macro name without the need to repeat the lines of code that make up the macro.

Macro 1 (Assign 0 to a variable X): X<- 0

```
While (X)
{
decr(X)
}
```

Macro 2 (Set X to be value n): X <- n

```
X <- 0
incr (X)
incr(X)

……
incr (X)
```

//the incr statement is repeated n times.

TASK 1.1: Write the code in the simple language for the macros 3 – 7.

Macro 3 (Set Y to receive the value in the variable X): Y <- X

Macro 4: Y <- Y + X

Macro 5: Y <- Y * X

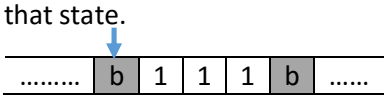
Macro 6: Y <- Y^X

Macro 7: if X then A

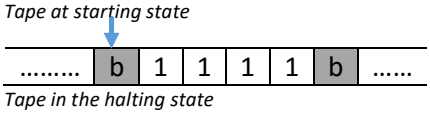
TASK 1.2: Is it possible to encode the statement **read X** in this language? If yes, how. If not, why not?

We will learn how to run the simple language on our Turing machine for **incr (X).**

The controller has 4 states, S1 to S4. S1 is the starting state and S4 the halting state, S2 is the move right state and S3 the move left state. If the machine reaches the halting state, it stops: there is no instruction starting with that state.



Tape at starting state



Tape in the halting state

(S1,b,b,R,S2); (S2,1,1,R,S2); (S2,b,1,L,S3); (S3,1,1,L,S3); (S3,b,b,N,S4)

We would like a Turing machine/ automaton that can implement incr(X) with minimum number of operations.

TASK 1.3: Show how this Turing machine can increment X when X = 3.

We will learn how to run the simple language on our Turing machine for **decr (X).**

It is sufficient to use three states. S1 is the starting state, S2 is the checking statement, which checks if the current symbol is 1 or blank b. S3 is the halting state.

(S1, b,b,S2); (S2,b,b,N,S3); (S2,1,b,N,S3)

Task 1.4: Can you tell in which state will this automaton be when it halts?

Homework 1. Simulate the fourth macro Y <- Y + X. Run the simulation for X = 2 and Y = 3

1.4. The Church-Turing Thesis

The Church-Turing thesis states that a computable function is one that can be run on a Turing machine. No algorithm has been found that cannot be simulated using a Turing machine.

1.5. Gödel Numbers

Assign a number to every program that can be written in a specific language.

Symbol -> Hexadecimal code

incr (X) is $(AF)_{16}$ which is 175.

Conversely, some numbers can be de-coded back to a program.

3058 becomes $(BF2)_{16}$ which is decr (X) 2 = decr (X_2)

The combination of a program and its input defines a Gödel number. These numbers are very useful when one wants to mathematically show properties of programs and programming languages (termination, correctness, confluence).

1.6. The Halting Problem

Programs that can be run on a computer involve some form of repetition – loops or recursive functions. A repetition may never terminate – it may loop forever.

For example

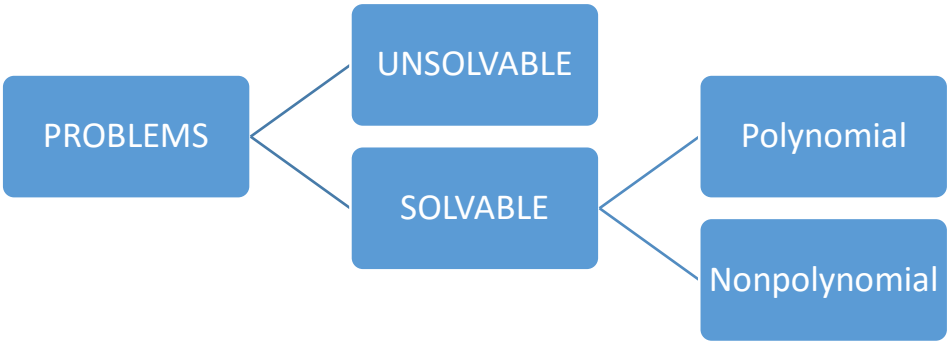
```
X <- 1

While (X)
{
}
```

Can we write a program that tests whether or not any program, represented by its Goedel number, will terminate?

This question is also known under the name **'the halting problem.'**

Unfortunately, the halting problem is not solvable. There does not exist a program that can test and decide whether or not a program terminates.

1.7. The Complexity of Problems



Examples of polynomial algorithms

Searching for an element in an array

Input: A sequence of n numbers A = {a1;a2;…;an} and a value v

Output: An index i such that v = A[i] or the special value NIL if v does not appear in A.

Write pseudocode for linear search, which scans through the sequence, looking for v.

1. How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array?
2. How about in the worst case?

3. What are the average-case and worst-case running times of linear search in O-notation? Justify your answers.

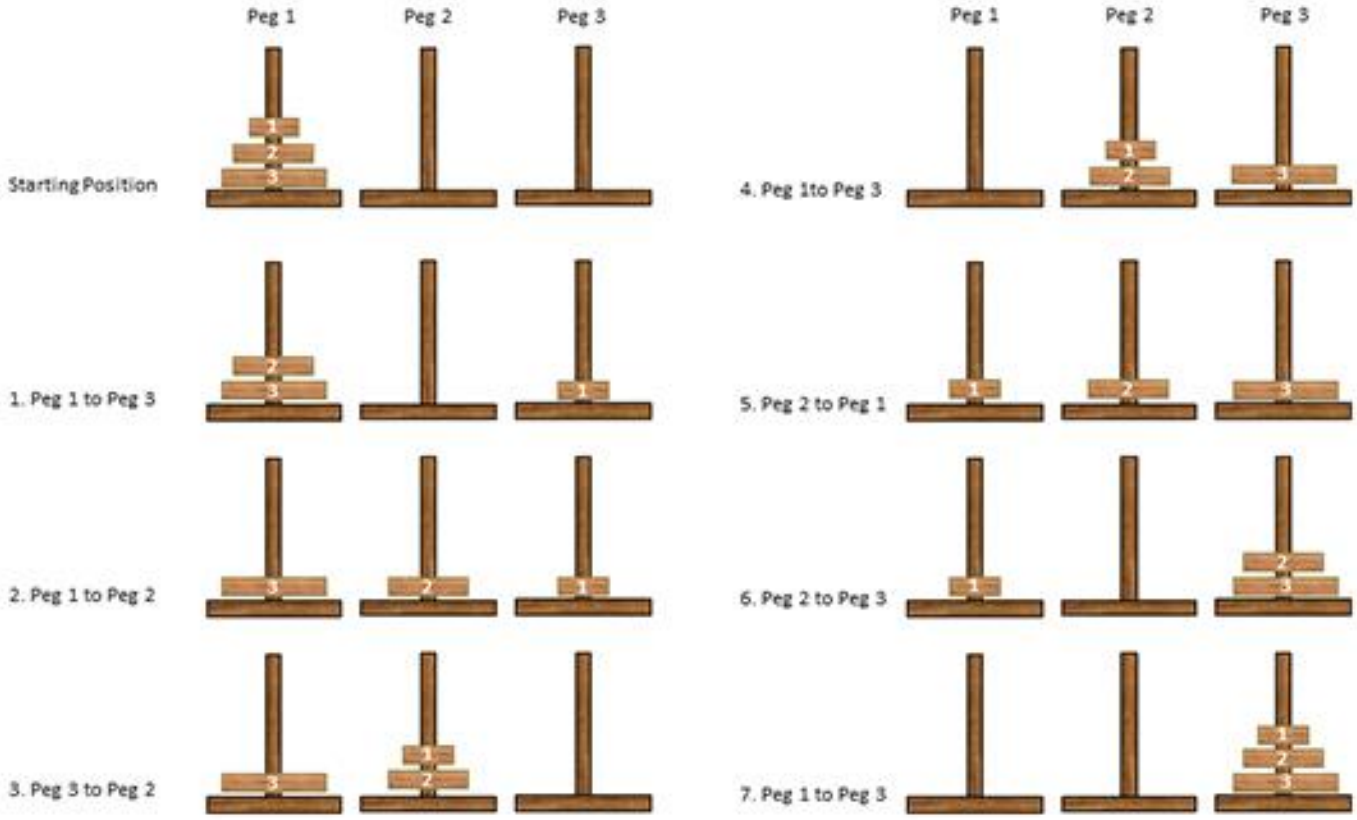
Sorting (see Section 2.1. Insertion Sort)

This typically require a quadratic algorithm, or at least a logarithmic algorithm.

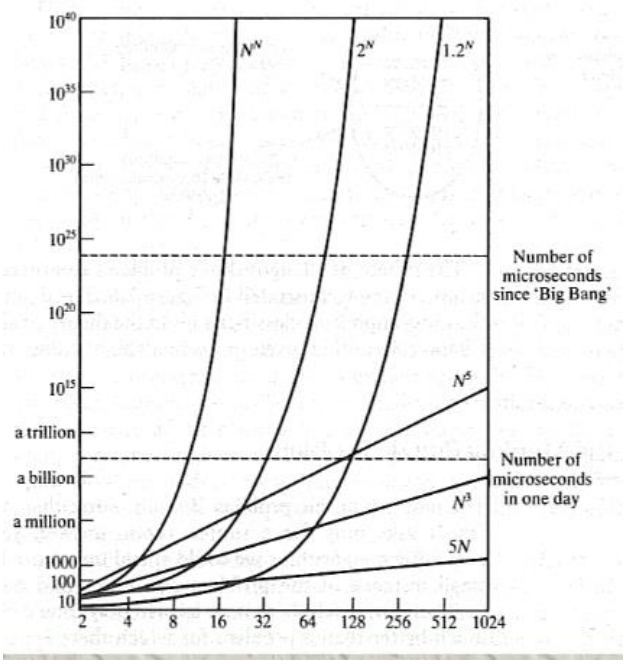
Examples of problems that are solvable but require exponential time

The Towers of Hanoi

The Classical 3-peg: There are 3 pegs and 4 disks of different sizes. At all times the disks are stacks so that we never place a larger disk over smaller ones. You are allowed to move disks one at a time.



There are variants of this problem with n- disks. This problem required 2^{n-1} moves for n disks. Even moving a million rings a second, for 64 rings you need half a million years to complete the puzzle. If you need 10 sec / ring, you need five trillion years! This is probably the lower bound.



The Stable Matching problem

The stable marriage problem has been stated as follows:

Given *n* men and *n* women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. When there are no such pairs of people, the set of marriages is deemed stable.

A brute-force algorithm will try all possible combinations and keep only the once that are correct solutions to this problem. The time complexity is N! * (time required to check if a matching is stable) = N! * (N^2)

```
function stableMatching {
    Initialize all m ∈ M and w ∈ W to free
    while ∃ free man m who still has a woman w to propose to {
       w = first woman on m's list to whom m has not yet proposed
       if w is free
         (m, w) become engaged
       else some pair (m', w) already exists
         if w prefers m to m'
            m' becomes free
           (m, w) become engaged
         else
           (m', w) remain engaged
    }
}
```

1.8. Inefficiency and Intractability (Some definitions)

Definition 1.8.1 (NP-Problem) A problem is assigned to the NP (nondeterministic polynomial time) class if it is solvable in polynomial time by a nondeterministic Turing machine. A P-problem (whose solution time is bounded by a polynomial) is always also NP.

Definition 1.8.2 (NP-Hard) NP-hardness (non-deterministic polynomial-time **hard**), in computational complexity theory, is a class of **problems** that are, informally, "at least as **hard** as the hardest **problems** in **NP**".

Definition 1.8.3 (NP-Complete Problem) In computational complexity theory, a decision **problem** is **NP-complete** when it is both in **NP** and **NP**-hard. The set of **NP-complete problems** is often denoted by **NP**-C or NPC. The abbreviation **NP** refers to "nondeterministic polynomial time".

Definition 1.8.4 (Tractable) Problems that are solvable in polynomial time are called tractable.

Definition 1.8.5 (Tractable) Problems that are solvable in superpolynomial time (more than polynomial but but exponential) are called intractable.

2. The Complexity of Algorithms

2.1. Example (The complexity of the Insertion Sort)

We start with insertion sort, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for acard, wecompare it with each of the cards already in the hand, from right to left, as illustrated in Figure 2.2. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

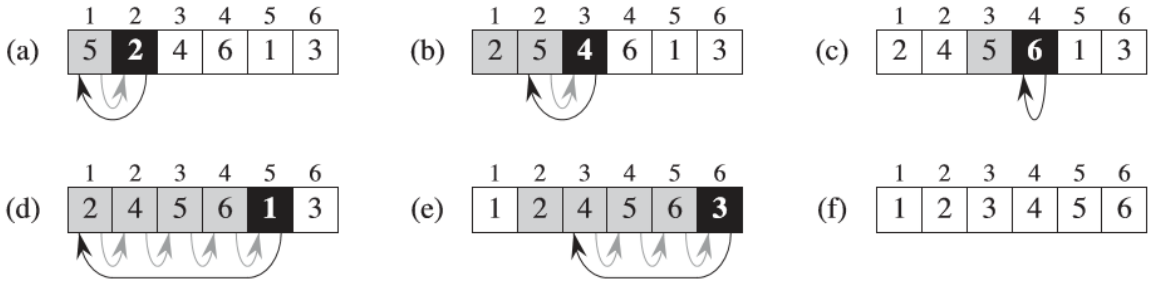


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

Here is the pseudocode of the insertion sort algorithm.

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

The costs c_i are the steps that are executes on each line of the algorithm. This is calculated by adding together all the operations executed on each line.

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute c_in in to the total running time.

To compute T(n), the running time of INSERTION-SORT on an input of n values, we sum the products of the cost and times columns, obtaining after some calculations:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i it is thus a quadratic function of n.

2.2. The Asymptotic Notation

Knowing how to analyse algorithms can help us to improve them so that they solve a problem in a most efficient way. We can analyse algorithms in terms of:

A) **Duration**: we estimate how long a program will run.
B) **Input size:** we can estimate the largest input that can be reasonably given to the program.
C) **Efficiency**: we can compare the number of steps required
D) **Improving Code**: we can analyse parts of code so that we can make them more efficient and remove redundant or inefficient sequences of commands.

The Time Complexity of an algorithm is a function indicating how the running time changes with the size of the input. The usual notation for the time complexity of an algorithm for an input of size n is T(n). We usually measure T(n) in terms of seconds.

Observation: *The running time is also affected by the hardware environment (e.g., the processor, clock rate, memory, disk) and software environment (e.g., the operating system, programming language) in which the algorithm is implemented and executed.*

How to choose which input you should use to calculate T(n):

1) **You should calculate T(n) for a typical input**

2) **You should calculate T(n) for an average or expected case**

3) **You should calculate T(n) for the worst case or for an input for which you suspect your algorithm would run the slowest.**

Example 2.1: Suppose you're your program P1 requires T1(n)=n^3 operations and P2 requires T2(n)=2^n operations. Suppose that your computer can execute 10^6 operations per second.

What is the running time of these programs for an input of size n=1000=10^3?

a) For P1, the number of required operations is (10^3)^3=10^9 and it will require 10^3 seconds about a day and a half.

b) For P2, the number of operations is 2^(10^3) and it will take $\frac{2^{10^3}}{60*60*365*10^6}$ if you take the log of this fraction and approximate you will get a running time of about 10^{289} years.

Example 2.2

(**Summation**) Summing up **n** numbers. This algorithm has T(n).

(**Factorisation**) Finding out all the divisors of a number. An algorithm would be for a given number N to try out and check if it is divisible to the numbers 1,2,3,…,N/2. For example for the number 5917 we get the factors 97 x 61.

The problem in Example 2.2 has application in cryptography and it is considered hard. Can you think of its time complexity?

2.3. Big O Notation (or the asymptotic performance)

Big O notation uses a function to describe how the algorithm's worst-case performance relates to the problem size as the size grows very large.

For example, $O(N^2)$ means an algorithm's runtime (or memory usage or whatever you're measuring) increases as the square of the number of inputs N. If you double the number of inputs, the runtime increases by roughly a factor of 4. Similarly, if you triple the number of inputs, the runtime increases by a factor of 9.

There are five basic rules for calculating an algorithm's Big O notation:

1. If an algorithm performs a certain sequence of steps f(N) times for a mathematical function f, it takes O(f(N)) steps.

2. If an algorithm performs an operation that takes O(f(N)) steps and then performs a second operation that takes O(g(N)) steps for functions f and g, the algorithm's total performance is O(f(N) + g(N)).

3. If an algorithm takes O(f(N) + g(N)) and the function f(N) is greater than g(N) for large N, the algorithm's performance can be simplified to O(f(N)).

4. If an algorithm performs an operation that takes O(f(N)) steps, and for every step in that operation it performs another O(g(N)) steps, the algorithm's total performance is O(f(N) × g(N)).

5. Ignore constant multiples. If C is a constant, O(C × f(N)) is the same as O(f(N)), and O(f(C × N)) is the same as O(f(N)).

2.4. Some complexities you will encounter:

1

An algorithm with O(1) performance takes a constant amount of time no matter how big the problem is

Log N

An algorithm with O(log N) performance typically divides the number of items it must consider by a fixed fraction at every step.
The logarithmic function log(N) grows relatively slowly as N increases, so algorithms with O(log N) performance generally are fast enough to be useful. As we shall see, algorithms that run in trees tend to have this performance.

N

The function N grows more quickly than log(N) but still not too quickly, so most algorithms that have O(N) performance work quite well in practice.

N log N

Suppose an algorithm loops over all the items in its problem set and then, for each loop, performs some sort of O(log N) calculation on that item. In that case, the algorithm has O(N × log N) or O(N log N) performance. Many sorting algorithms that work by comparing items have an O(N log N) runtime. In fact, it can be proven that *any* algorithm that sorts by comparing items must use at least O(N log N) steps, so this is the best you can do, at least in Big O notation.

N^2

An algorithm that loops over all its inputs and then for each input loops over the inputs again has $O(N^2)$ performance. An algorithm is said to have *polynomial runtime* if its runtime involves any polynomial involving N. O(N), $O(N^2)$, $O(N^6)$, and even $O(N^{4000})$ are all polynomial runtimes.

Polynomial runtimes are important because in some sense these problems can still be solved

2^N

Exponential functions such as 2^N grow extremely quickly, so they are practical for only small problems. Typically algorithms with these runtimes look for optimal selection of the inputs.

For problems with exponential runtimes, you often need to use *heuristics*—algorithms that usually produce good results but that you cannot guarantee will produce the best possible results.

$N!$

The factorial function, written N! and pronounced "N factorial," is defined for integers greater than 0 by $N! = 1 \times 2 \times 3 \times ... \times N$. This function grows much more quickly than even the exponential function 2^N. Typically algorithms with factorial runtimes look for an optimal arrangement of the inputs.

For example, in the traveling salesman problem (TSP), you are given a list of cities. The goal is to find a route that visits every city exactly once and returns to the starting point while minimizing the total distance traveled.

3. Exercises

Write an algorithm and a python program for each of the problems below.

Simple Exercises

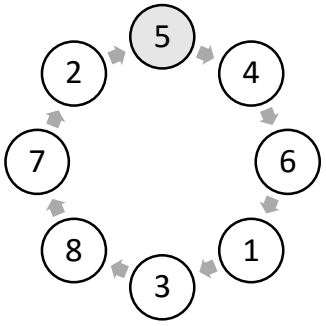
1. Linear Search
2. Insertion Sort
3. Compare two arrays: print common elements

Medium- Hard

1. Write a python program to simulate the Turing machine and the simple language in these notes.

Difficult Exercises

1. **(Pass the parcel)** There are **n** children seated in a circle. The children pass a parcel that has **n** levels under each level there is a sweet. The final level of wrapping contains a bar of chocolate. Starting from a given position in the circle, we count in one direction and the m-th person gets to open a level of the wrapping then exists the circle. As children exit the game, the circle becomes smaller. Write a program that writes the order in which children get a sweet and print who gets the final bar of chocolate. As an input to the algorithm you need: the number n of children, the number m-for counting and the position at which the counting starts.



In the image, we have 8 children numbered from 1 to 8. We start counting from 5. In the circle we have the order in which the children get to open the parcel.

2. ***HARD* (Magic Square).** This is a square of n x n that contains numbers from 1 to n^2 so that the sum of the elements on each row and the sum of the elements on each column to be equal to n(n^2+!)/2. In addition, the sum of the elements on the two diagonals is also equal to the same number.
The square below is the magic square for n=7.

22	**47**	**16**	**41**	**10**	**35**	**04**
05	**23**	**48**	**17**	**42**	**11**	**29**
30	**06**	**24**	**49**	**18**	**36**	**12**
13	**31**	**07**	**25**	**43**	**19**	**37**
38	**14**	**32**	**01**	**26**	**44**	**20**
21	**39**	**08**	**33**	**02**	**27**	**45**
46	**15**	**40**	**09**	**34**	**03**	**28**

Is this problem solvable? If yes, write an algorithm to generate magic square of n size given n as an input. Is your algorithm polynomial?