Running Time of TM:

- Let M be a decider. The **running time of M/the time complexity of M** is function T: N → N s.t. T(n) is the maximum number of steps that M takes on inputs of size n before it halts.
- **Big-O** is used to describe the worst case running time for a given algorithm. It provides an upper bound.
  O(n) means that it takes at most time proportional to n.
- **Big-Ω** is used to describe the best case running time for a given algorithm. It provides a lower bound.
  Ω(n) means that it takes at least time proportional to n.
- **Big-Θ** is used to describe a tight bound for the running time for a given algorithm.
  T(n) is said to be in Θ(f(n)) if it is both in O(f(n)) and in Ω(f(n)).
  Θ(n) means that it is within a constant of n.
- Recall that the different variations of TMs, regular TMs, multi-tape TMs, NTMs, are all equivalent. However, when we are counting time, it does matter which type of TM we use.
- E.g.
  Consider the following language L = $\{0^k1^k \mid k \in N\}$.
  We want to have a recognizer for this language.
  Here is what the recognizer does:
  Given the input on the tape, the recognizer:
  1. Scans the entire input to check if there are any 0's after a 1.
  2. If there isn't:
     a. It goes back to the start of the tape.
     b. It marks the 0.
     c. It scans to the right until it sees a blank symbol, meaning that it has scanned the entire tape, or that it sees an unmarked 1.
     d. If it sees an unmarked 1, it marks it, checks if there are any more unmarked 1's and then scans to the left until it hits the last unmarked 0.
     e. If there are no more unmarked 0's, but there are still unmarked 1's, then reject.
        If there are unmarked 0's but no more unmarked 1's, then reject.
        If there are no more unmarked 0's and 1's, accept.
        Repeat steps b to e until it halts.
  3. If there is, reject.

  The running time of this algorithm, $T_1(n)$, is $\Theta(n^2)$.
  $T_1(n) = \Theta(n^2)$
  If there is a 0 after a 1, we will find it after n moves at most. This is the first sweep.
  If there is no 0 after a 1, to compare each pair of unmarked 0 and 1, will take about n/2 steps.
  There will be about n/2 of these pair matchings.
  $(n/2)*(n/2) = n^2/4$
  Hence, it takes $\Theta(n^2)$ time.

However, we can come up with another recognizer that recognizes this language faster.
Here is what the new recognizer does:
Given the input on the tape, the recognizer:

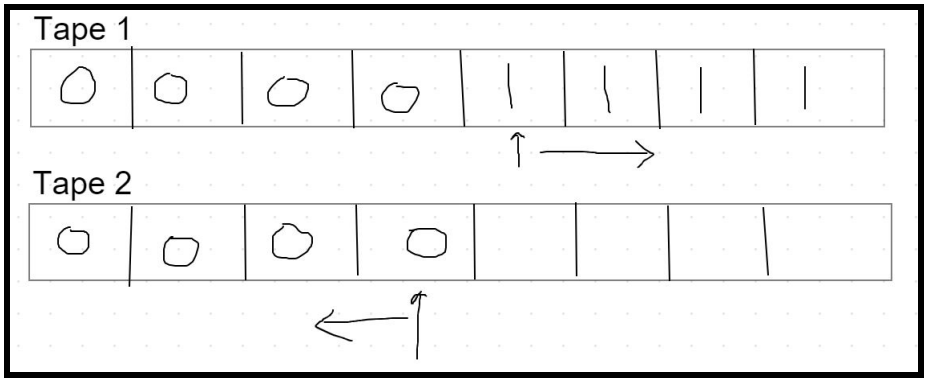
1. Scans the entire input to check if there are any 0's after a 1.
2. If there isn't:
    a. It goes back to the start of the tape.
    b. Starting from the first unmarked 0, it marks every 0 and every other 1.
    c. Repeats steps a & b until it either accepts or rejects. It accepts if there are no more 0's and 1's to mark. It rejects if there are still more 0's or 1's to mark, but no more of the opposite type to mark.
    I.e. There are 0's left but no 1's left or there are 1's left but no 0's left.
3. If there is, reject.

The running time of this algorithm, $T_2(n)$, is $\Theta(n\log(n))$.
Note: There is no standard TM that can recognize the language in fewer than $\Theta(n\log(n))$ steps.

Now, instead of using a standard TM, we will use a multi-tape (2-tape) TM.
Here is what the multi-tape recognizer does:
Given the input on the first tape, the recognizer:

1. Scans the entire input to check if there are any 0's after a 1.
2. If there isn't:
    a. It goes back to the start of the first tape.
    b. It copies all the 0's onto the second tape.
    c. Now, we will move right on tape 1 and move left on tape 2. If tape 1 arrives at the blank symbol at the same time tape 2 arrives at the beginning of the tape, then accept. Otherwise, reject.
3. If there is, reject.

Here's a picture of the 2 tapes.
Tape 1 contains the input.
Tape 2 contains all the 0's copied over from tape 1.
We start at the leftmost 1 in tape 1 and the rightmost 0 in tape 2.
At each step, we move right 1 cell in tape 1 and left 1 cell in tape 2.
If tape 1 arrives at the first blank symbol the same time that tape 2 arrives at the beginning of its tape, then accept. Otherwise, reject.



The running time of this algorithm, $T_3(n)$, is $\Theta(n)$.
We scanned the entire tape to make sure that no 0's came after 1's. (Takes n steps).
We copied all the 0's from the first tape to the second tape. (Takes n/2 steps).
We ran both tapes, tape 1 starting from the first 1 and tape 2 starting from the last 0. (Takes n/2 steps).
In total, it took $\Theta(n)$ steps.

Notice that while both single and multi-tape TMs can recognize the language, the single tape TM can do so in $\Theta(n\log(n))$ steps at best while the multi-tape TM can do so in $\Theta(n)$ steps.

Because of the fact that different variations of TMs have different running times, we want to make a very rough distinction between problems that can be solved in polynomial time and problems that cannot be solved in polynomial time.

- We classify languages/decision problems into 2 categories:
    1. Those that are decidable/solvable in polynomial time, no matter what the exponent is.
       I.e. $O(n^k)$ where k is a constant.
    2. Those that are not decidable/solvable in polynomial time, no matter what the exponent is. These typically require exponential time.
       I.e. $O(2^n)$

Note: $O(n^{\log n})$ is not polynomial, because log n is not a constant, but is still better than $O(2^n)$.

Polynomial Time Thesis:

- **Polynomial Time Thesis:** A problem is efficiently computable/tractable/feasible iff there is a polynomial time algorithm that solves it.

Theoretical Justifications:

1. The running time is immune to models of computation up to polynomials.
   Recall theorem 2.1. It said that if a multi-tape TM takes m steps, then an equivalent regular TM takes $O(m^2)$ steps. Hence, if a multi-tape TM takes polynomial time, then its equivalent regular TM still takes polynomial time.
   Note: There is an important likely exception. If an NTM takes polynomial time, we don't know if its equivalent regular TM takes polynomial time.
2. The running time is immune to the details of encoding numbers up to polynomials, as long as we don't cheat. (The "don't cheat" part will be explained below.)
   I.e. Size of different reasonable encoding differs only by a polynomial amount.
   This is relevant because the running time is a function of the size of the input. If one way of counting the size of the input gets a very different result than another way, then we will get wildly different running times.
   However, if the size of different reasonable encoding differs only by a polynomial amount, it's ok because a polynomial of a polynomial is still a polynomial.
   E.g. 1: We want to represent the number n.
   The binary representation of n has $\text{floor}(\log_2 n)$ bits.
   The base r representation of n has $\text{floor}(\log_2 r)$ digits.
   $\log_2 n = (\log_2 r) * (\log_r n)$ where $\log_2 r$ is a constant.

E.g. 2: Consider a graph G = (V,E)
Let m = |E|
Let n = |V|
One way to calculate the size of the graph is m+n.
Another way to calculate the size of the graph is n^2.
The 2 ways are within a polynomial amount.
A third and more accurate way to calculate the size of the graph is the following:
Each node (vertex) is represented by $\log_2 n$ bits.
Hence, n nodes are represented by $n\log_2 n$ bits.
Furthermore, since each edge has 2 nodes, m edges are represented by $2m\log_2 n$ bits.
In total, we have $n\log_2 n + 2m\log_2 n$ or $\Theta((n+m)\log_2 n)$ bits.
Once again, the difference is within a polynomial amount.

Now, I will explain by "don't cheat". Don't cheat means that we don't put garbage in our encoding.
E.g. 3: Going back to our example 1, "We want to represent the number n", representing a number, x, in base 1 means that we write a symbol x many times.
If we want to represent 1 million in base 1, we have to write 1 million symbols.
Now, the difference between base 1 and base 2 representation of any number is no longer polynomial, it's exponential.

Empirical Justifications:

1. Polytime solvable problems that arise in practice usually have a "reasonable" exponent.
   For example, n^{100} can be really slow even with pretty small inputs, but in reality, most problems can be solved in n^2 or n^3 steps.
2. Furthermore, even when exponents are very high, they are still less than 2^n in most cases.
   For example, $n^{100} \leq 2^n \ \forall \ n \geq 1000$.
3. Algorithms that are not in polynomial time are usually in exponential time and exponential time algorithms are usually brute-force algorithms.

Complexity Class P:

- **P** is the set of languages/decision problems that can be decided by polynomial TMs.
  Note: NP does not mean non-polynomial time.
- Here are some examples of languages in P:
  - **E.g. 1:** SUM
    Instance: The representation/encoding of 3 numbers, x, y and z.
    I.e. Instance: $\langle x, y, z \rangle$ s.t. $x, y, z \in N$
    Question: Is $z = x + y$?
    Running time: Linear
  - **E.g. 2:** REACHABILITY
    Instance: $\langle G, s, t \rangle$ where G = (V,E) is a directed graph and s and t are nodes.
    $s, t \in V$
    Question: Does G have a path from s to t?
    Running time of BFS and DFS: $\Theta(m+n)$ where m = |E| and n = |V|

- **E.g. 3:** WEIGHTED SPANNING TREE
  Instance: ⟨G, wt, b⟩ where G = (V,E) is an undirected graph and wt is a weight function s.t. wt: E → N and b is a budget s.t. b ∈ N.
  Question: Does G have a spanning tree whose weight is no more than b?
- **E.g. 4:** PRIME
  Instance: ⟨n⟩ where n is a natural number.
  Question: Is n prime?

  Here is an algorithm, called PRIME-SOLVER, that solves this question:
  if n < 2 then return no
  for i = 2 to floor($\sqrt{n}$) do
  if n mod i = 0 then return no
  return yes

  Note: This algorithm is not in P.
  Here's the reasoning:
  - The loop runs Θ($\sqrt{n}$) or Θ($n^{1/2}$) times.
  - However, we want our program to be polynomial in respect to |⟨n⟩|. It doesn't take n bits to represent n. It takes $\log_2 n$ bits to represent n. Similarly, it takes $\log_2 n^{1/2}$ bits to represent $\sqrt{n}$.
  - $$\sqrt{n} = 2^{\log_2 \sqrt{n}} = 2^{\frac{1}{2}\log_2 n}$$
    Since $\log_2 n$ is the size of |⟨n⟩|, this algorithm is exponential in respect to |⟨n⟩|. Hence, it's not in P.

  Just because a problem is in P doesn't mean that every decider for that language is polynomial.
  PRIME is in P while PRIME-SOLVER is not in P.

- **Theorem 7.1:** Consider the language EXP = {⟨M,x⟩ | M accepts x in at most $2^{|\langle x \rangle|}$ steps}. EXP is decidable but is not in P.

  Proof that EXP is not in P:
  Suppose by contradiction that EXP is in P.
  Then, let EXP' = {⟨M⟩ | M accepts ⟨M⟩ in at most $2^{|\langle M \rangle|}$ steps}
  EXP' is in P.
  Let D = ¬EXP'. D = {⟨M⟩ | M does not accept ⟨M⟩ in at most $2^{|\langle M \rangle|}$ steps}
  If a problem is in P, then its complement is also in P.
  Hence, D is in P.
  Therefore, there exists a polytime TM M_D that decides D.
  Let P(n) = n^k be a polynomial upper bound on the running time of M_D.
  Let n_0 be a number s.t. ∀ $n \geq n_0$ $n^k \leq 2^n$.
  n_0 exists because every polynomial is less than or equal to an exponential for a sufficiently large n.
  Assume without loss of generality that $|\langle M_D \rangle| \geq n_0$.
  What does M_D do on input $\langle M_D \rangle$?

Case 1: M_D accepts $\langle M_D \rangle$.
→ This means that M_D does not accept $\langle M_D \rangle$ in $\leq 2^{|\langle MD \rangle|}$ steps.
→ But M_D on $\langle M_D \rangle$ halts in $|\langle M_D \rangle|^k$ steps and $|\langle M_D \rangle|^k \leq 2^{|\langle MD \rangle|}$.
→ M_D must reject $\langle M_D \rangle$. This is a contradiction.

Case 2: M_D rejects $\langle M_D \rangle$.
→ This means that M_D does not accept $\langle M_D \rangle$ in $\leq 2^{|\langle MD \rangle|}$ steps, since it does not accept $\langle M_D \rangle$ at all.
→ This means that M_D accepts $\langle M_D \rangle$. This is a contradiction.

This means that our original assumption that EXP is in P is false.
Hence, EXP is not in P.

- **Note:** Undecidable problems are not in P.
- **Note:** A polynomial of a polynomial is still a polynomial.

Decision, Search and Optimization Problems:

- **Travelling Salesman Problem (TSP-OPT):**
  Input: $\langle G, wt \rangle$ where $G = (V,E)$ is an undirected graph and wt is a weight function, $wt: E \rightarrow N$.
  Output: A minimum weight tour of this graph, if one exists.
  A tour means a cycle that visits each node exactly once.
  The weight of a tour is the sum of the weight of its edges.
- I can define a decision counterpart, **TSP-DEC**, that will help us solve TSP-OPT. This is a decision problem.
  Instance: $\langle G, wt, b \rangle$ where $G = (V,E)$ is an undirected graph, wt is a weight function, $wt: E \rightarrow N$, and b is a budget, $b \in N$.
  Question: Does G have a tour of $wt \leq b$?

  Solution:
  Given an oracle, which is a subroutine used as a black box, that solves TSP-DEC in one step, called TSP-DEC-SOLVER, that takes in $\langle G, wt, b \rangle$, construct an algorithm called TSP-OPT Solver, which takes in $\langle G, wt \rangle$, and solves TSP-OPT in polytime.

  TSP-OPT-SOLVER($\langle G, wt \rangle$)
  1. Use TSP-DEC-SOLVER($\langle G, wt, b \rangle$) and binary search to find the weight, b^*, of the optimal tour.

     For the binary search:
     The lower bound is $n*w_{min}$, where n is the number of edges and w_{min} is the minimum weight of any edge.
     The upper bound is $n*w_{max}$, where n is the number of edges and w_{max} is the maximum weight of any edge.
     To get the starting value for the budget, we take the middle value in the range from w_{min} and w_{max}.
     Run TSP-DEC-SOLVER($\langle G, wt, b \rangle$) to see if there's a tour that costs within that budget.
     If yes, we go in the direction of w_{min} to see if we can find a cheaper tour.
     Otherwise, we go in the direction of w_{max} to increase our budget.

2. Remove the edges one at a time from G and use TSP-DEC-SOLVER(⟨G, wt, b*⟩) to see if there exists a tour of wt b* in the remaining graph.
   If yes, continue.
   If no, put the edge back and continue.
3. Output edges left in G.

Proof that TSP-OPT-SOLVER has polynomial run time:

1. $|\langle G, wt\rangle| = O((m+n)\log n + m*\log_2 w_{max}) \rightarrow$ this is polytime w.r.t to n, m, $\log_2 w_{max}$
   $|\langle G\rangle| = O((m+n)\log n)$, as stated before
   $|\langle wt\rangle| = O(m*\log_2 w_{max})$ cause $m*w_{max}$ is the maximum total weight of the edges.

2. Time needed for steps 1 and 2 above =
   Number of calls to TSP-DEC-SOLVER * size of input

   The size of the input is $O((m+n)\log n + m*\log_2 w_{max} + \log_2(n*w_{max}))$.
   The input is ⟨G, wt, b⟩
   The worst b can be is $n*w_{max}$.
   Hence, the size of the input is $O((m+n)\log n + m*\log_2 w_{max} + \log_2(n*w_{max}))$.

   The number of calls to TSP-DEC-SOLVER is $O(\log_2(n*w_{max}) + m)$.
   This is because in step 1 from above, we make $\log_2(n*w_{max} - n*w_{min})$ calls to TSP-DEC-SOLVER.
   Setting $n*w_{min}$ to be 0, we get $\log_2(n*w_{max})$.
   Furthermore, in step 2, we call TSP-DEC-SOLVER on each edge of G.
   Since G has m edges, we call TSP-DEC-SOLVER m times.
   Hence, the number of calls to TSP-DEC-SOLVER is $O(\log_2(n*w_{max}) + m)$.

   In total, the time needed for steps 1 and 2 above =
   $((m+n)\log n + m*\log_2 w_{max} + \log_2(n*w_{max})) * (\log_2(n*w_{max}) + m)$,
   which is still in polynomial time.
3. The time needed for step 3 = $O(n*\log_2 n)$.
   This is because each edge is represented using $\log_2 n$ bits and there are n edges.

Hence, the time for TSP-OPT-SOLVER is polynomial in the size of the input assuming that TSP-DEC-SOLVER takes 1 step.
We assumed that TSP−DEC−SOLVER is constant time, but even if it did take polynomial time it would be fine. This is because a polynomial combined with a polynomial function is still polynomial.

- **Max Independent Set (MAX-IS):**
  Input: ⟨G⟩ where G = (V,E) is an undirected graph.
  Output: V' ⊆ V s.t. V' is an independent set (IS), meaning that there are no edges between any 2 nodes, of maximum size.
- I can define a decision counterpart, **IS-DEC**, that will help us solve MAX-IS. This is a decision problem.
  Instance: ⟨G, b⟩ where G = (V,E) is an undirected graph, and b is a budget, b ∈ N.
  Question: Does G have an IS of size ≥ b?