Hipótese de Church: "São caracterizações tão gerais da noção do efetivamente computável quanto consistentes com o entendimento intuitivo usual" (Church).

A Hipótese de Church é assumida como verdadeira em todos os estudos relacionados com a Teoria da Computação. Note-se que não é demonstrável pois é fundamentada em uma noção intuitiva (não-formal) do que é efetivamente computável

**Alfabeto**: conjunto finito de símbolos ou caracteres

**Cadeia de Símbolos**: sequência de zero ou mais símbolos justapostos

**Palavra**: cadeia de símbolos finita

**|w|** = comprimento ou tamanho da palavra

**Prefixo** = qualquer sequencia inicial de símbolos contígua da palavra

**Subpalavra** = qualquer sequencia de símbolos contígua da palavra

**Sufixo** = qualquer sequencia final de símbolos contígua da palavra

**Linguagem Formal** = conjunto de palavras sobre um alfabeto

Máquinas de Turing descrevem Linguagens Recursivamente Enumeráveis (LRE)

Um **programa** pode ser descrito como um conjunto estruturado de instruções que capacitam uma máquina a aplicar sucessivamente certas operações básicas e testes sobre os dados iniciais fornecidos, com o objetivo de transformar estes dados numa forma desejável.

Um **programa monolítico** é estruturado usando desvios condicionais e incondicionais, não fazendo uso explícito de mecanismos auxiliares de programação que permitam uma melhor estruturação do controle como iteração, subdivisão ou recursão. Portanto, a lógica é distribuída por todo o bloco (monólito) que constitui o programa. Pode ser especificado através de **fluxogramas**.

**Programa Iterativo**: (Programação Estruturada)

Um Programa Iterativo P é indutivamente definido como segue:

a) A operação vazia v constitui um programa iterativo;

b) Cada identificador de operação constitui um programa iterativo;

c) Composição Seqüencial. Se V e W são programas iterativos, então a composição seqüencial denotada por:

V;W

resulta em um programa iterativo cujo efeito é a execução de V e, após, a execução de W;

d) Composição Condicional. Se V e W são programas iterativos e se T é um identificador de teste, então a composição condicional denotada por:

(se T então V senão W)

resulta em um programa iterativo cujo efeito é a execução de V se T é verdadeiro ou W se T é falso;

e) Composição Enquanto. Se V é um programa iterativo e se T é um identificador de teste, então a composição enquanto denotada por:

enquanto T faça (V)

resulta em um programa iterativo que testa T e executa V, repetidamente, enquanto o resultado do teste for o valor verdadeiro. Caso contrário, a iteração termina;

f) Composição Até. Se Vé um programa iterativo e se T é um identificador de teste, então a composição até denotada por:

até T faça (V)

resulta em um programa iterativo que testa T e executa V, repetidamente, enquanto o resultado do teste for o valor falso. Caso contrário, a iteração termina.

Assim, relativamente a composição seqüencial, tem-se que:

P1:P2:P3 :...: Pn

denota o programa cujo efeito é a execução na ordem P1, P2, ... , Pn (da esquerda

para a direita).

**Linguagem recursiva**: é uma linguagem formal para a qual existe uma máquina de Turing que sempre para (decide) quando recebe uma entrada finita de símbolos do alfabeto da linguagem. Isso significa que a máquina de Turing aceita exatamente as palavras que pertencem à linguagem e rejeita todas as outras.

Linguagens recursivas são também chamadas de **decidíveis** ou **Turing-decidíveis**.

**Computação de um programa recursivo**: Intuitivamente, consiste na avaliação da expressão inicial onde cada identificador de sub-rotina referenciado é substituído pela correspondente expressão que o define, e assim sucessivamente (recursivamente), **até que seja substituído pela expressão vazia v, determinando o fim da recursão.**

**Máquina**: formalismo que permite a execução de um programa

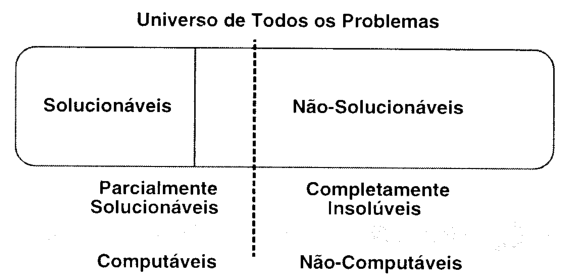
**Função computada**: a computação de um programa deve ser associada a uma entrada e uma saída. Adicionalmente, espera-se que a resposta (saída) seja gerada em um tempo finito.

**Computação**: um histórico do funcionamento da máquina para o programa, considerando um valor inicial.



Equivalência Forte de Programas. Se as correspondentes funções computadas coincidem para qualquer máquina

duas máquinas são consideradas equivalentes, quando a máquina 1 pode simular a máquina 2 e a 2 pode simular a máquina 1.



Problema Decidível (Solucionável): Um problema é decidível se existe um algoritmo que pode resolver todas as instâncias do problema de maneira correta e em tempo finito

Problema Indecidível: Um problema é considerado indecidível se não há algoritmo que possa resolver todas as instâncias do problema de maneira correta e em tempo finito

Problema Decidível Parcialmente: um problema decidível parcialmente (ou semidecidível) é um problema para o qual existe um algoritmo que pode confirmar uma resposta positiva, mas não necessariamente uma resposta negativa

Classes de problemas na complexidade computacional:

1. **P (Polynomial time)**:
   * **Definição**: Problemas que podem ser resolvidos por um algoritmo determinístico em tempo polinomial.
   * **Exemplo**: Ordenar uma lista de números usando o algoritmo de ordenação por inserção.
2. **NP (Nondeterministic Polynomial time)**:
   * **Definição**: Problemas cuja solução pode ser verificada por um algoritmo determinístico em tempo polinomial.
   * **Exemplo**: O problema do caixeiro viajante, onde a verificação de uma solução (um caminho específico) é rápida, mas encontrar essa solução pode não ser.
3. **NP-completo**:
   * **Definição**: Problemas que são tanto NP quanto NP-difícil. Ou seja, são os problemas mais difíceis dentro de NP, e qualquer problema em NP pode ser reduzido a eles em tempo polinomial.
   * [**Exemplo**: O problema da satisfatibilidade booleana (SAT), onde se deve determinar se existe uma atribuição de valores que torna uma fórmula booleana verdadeira1](https://acervolima.com/diferenca-entre-np-dificil-e-np-completo/).
4. **NP-difícil**:
   * **Definição**: Problemas que são pelo menos tão difíceis quanto os problemas mais difíceis em NP. Eles não precisam estar em NP (ou seja, suas soluções não precisam ser verificáveis em tempo polinomial).
   * [**Exemplo**: O problema de otimização do caixeiro viajante, onde se busca o caminho mais curto possível que visita todas as cidades exatamente uma vez2](https://pt.wikipedia.org/wiki/NP-dif%C3%ADcil).
5. **O(1) - Constante**:
   * **Exemplo**: Acesso a um elemento específico em um array. Não importa o tamanho do array, o tempo de acesso é sempre o mesmo.
6. **O(log n) - Logarítmica**:
   * **Exemplo**: Busca binária em uma lista ordenada. [A cada passo, o tamanho do problema é reduzido pela metade1](https://www.iugu.com/blog/analise-complexidade-algoritmos).
7. **O(n) - Linear**:
   * **Exemplo**: Percorrer todos os elementos de um array para encontrar um valor específico. O tempo de execução cresce linearmente com o tamanho do array.
8. **O(n log n) - Quase-linear**:
   * **Exemplo**: Algoritmos de ordenação eficientes como o Merge Sort e o Quick Sort. [Eles dividem o problema e resolvem subproblemas de forma recursiva](https://www.iugu.com/blog/analise-complexidade-algoritmos)[2](https://www.escoladnc.com.br/blog/a-importancia-da-complexidade-algoritmica-na-ciencia-da-computacao/).
9. **O(n²) - Quadrática**:
   * **Exemplo**: Algoritmos de ordenação simples como o Bubble Sort e o Insertion Sort. O tempo de execução cresce quadraticamente com o tamanho do array.
10. **O(n³) - Cúbica**:
    * **Exemplo**: Algoritmos de multiplicação de matrizes ingênuos. O tempo de execução cresce cubicamente com o tamanho das matrizes.
11. **O(2^n) - Exponencial**:
    * **Exemplo**: Algoritmos de força bruta para resolver o problema do caixeiro viajante. O tempo de execução cresce exponencialmente com o número de cidades.
12. **O(n!) - Fatorial**:
    * **Exemplo**: Algoritmos de força bruta para resolver problemas de permutação, como gerar todas as permutações possíveis de um conjunto.