### Introdução

"Alguns problemas computacionais são resolvíveis a princípio, mas as soluções requerem tanto tempo ou espaço a mais que elas não são usadas na prática. Tais problemas são chamados intratáveis." (Sipser M., 1997)

Resumindo, intratabilidade se refere aos problemas computacionais que são possíveis de se resolver na teoria, mas na prática são inviáveis já que, ou levam muito tempo para realizar a computação necessária, ou gastam muito espaço para tal ou ambos, mesmo quando as entradas para o problema são pequenas.

Aqui nesta apresentação, iremos discorrer sobre as bases dos problemas intratáveis.

### Definições

Aqui iremos apresentar algumas definições que serão usadas ao longo deste trabalho. Todas as definições foram retiradas de (Cormen, 2009). As definições retiradas de outras fontes possuem suas citações.

### Alfabeto

Um “alfabeto é um conjunto finito de símbolos.”

### Linguagem

Uma “linguagem sobre é qualquer conjunto de cadeias feitas de símbolos do alfabeto .”

### Aceitação

Um “algoritmo aceita uma cadeia se, dada a entrada , a saída do algoritmo é 1. A linguagem aceita por é o conjunto de cadeias = { : = 1}. Se = 0, é dito que o algoritmo rejeita a cadeia ”;

#### Aceitação em tempo polinomial

Uma “linguagem é dita aceita em tempo polinomial por um algoritmo se é aceita por esse algoritmo e se, em adição, existe uma constante tal que para qualquer cadeia de tamanho como entrada, com , o algoritmo aceita em tempo .” Por hora, basta dizermos que significa que o tempo de execução do algoritmo é limitado superiormente pela função = .

### Decisão

Uma “linguagem é decidida por um algoritmo se qualquer cadeia binária em é aceita por e qualquer cadeia binária não pertencente à é rejeitada por .”

#### Decisão em tempo polinomial

Uma “linguagem é **decidida** **em tempo polinomial** por um algoritmo se existir uma constante que, para qualquer cadeia de tamanho , o algoritmo decide corretamente se em tempo .” Isto é, existir uma constante k para a qual o algoritmo A aceite todas as cadeia binárias pertencentes à L e rejeite todas as cadeias não pertencentes à L, tudo isso com tempo .

### 

### Verificação

Um “problema é verificado se pudermos nos certificar de que a solução está correta em tempo polinomial em relação ao tamanho da entrada do problema.”

### Redutibilidade em tempo polinomial

Uma “linguagem é redutível em tempo polinomial para uma linguagem se existir uma função computável em tempo polinomial tal que para todo se, e somente se, (a cadeia x aplicada na função f tenha um resultado que pertença à L2) .

Essa função é chamada função de redução e um algoritmo em tempo polinomial que computa é chamado algoritmo de redução.”

Uma notação utilizada para representar a redutibilidade é: .

Dizer que uma linguagem é redutível a outra significa de dizer que não é mais do que um fator polinomial mais difícil que .

## **Complexidade computacional**

**PRIMEIRO SLIDE**

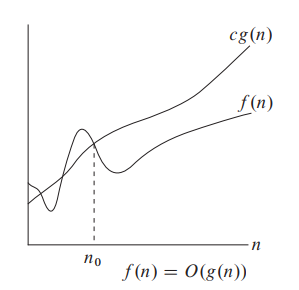
Um ponto que será mencionado algumas vezes ao longo desta apresentação se refere à complexidade computacional dos algoritmos. Ela é um ramo da Teoria da Computação que determina quais são os requisitos de recursos para diferentes problemas a partir de um algoritmo. Um grande foco em relação aos recursos é dado à memória que é gasta ao executar determinado algoritmo, bem como ao tempo despendido na execução.

**SEGUNDO SLIDE**

Ainda dentro desse assunto, temos a complexidade temporal assintótica. Essa é o estudo sobre como o tempo de execução do algoritmo aumenta em relação ao tamanho da entrada permitida, isso quando o tamanho dessa entrada aumenta sem restrições. Cormen, em seu livro Introdução aos Algoritmos diz que um algoritmo assintoticamente eficiente é sempre preferível em todos os casos, exceto quando a entrada é muito pequena.

**TERCEIRO SLIDE**

Na apresentação, algumas vezes serão mencionadas que algum algoritmo possui complexidade O(g(n)), onde é alguma função limitante superior, como vocês podem visualizar na imagem. Basicamente, estamos dizendo que o problema tem uma complexidade temporal assintótica limitada superiormente por uma constante ‘c’ qualquer vezes a função g(n) para qualquer entrada de tamanho maior ou igual à um valor n0 definido.



## **CLASSE P**

Como falamos no início, um problema é intratável se não possui um algoritmo eficiente que possa resolvê-lo. A classe P, que significa classe de complexidade polinomial, foi apresentada primeiramente por Cobham em 1964. Ela é a classe que concentra todos os problemas que podem ser resolvidos em tempo polinomial por algum algoritmo. Mais especificamente, problemas que podem ser resolvidos em tempo polinomial por uma Máquina de Turing DETERMINÍSTICA. Cormen se refere aos problemas da classe P como sendo os **problemas que podem ser resolvidos com uma complexidade de tempo O(n^k), para alguma constante k, onde n é o tamanho da entrada do problema.** Mais a fundo, nós podemos usar as definições de aceitação e linguagem para definir a classe P como sendo o conjunto de todas as linguagens que podem ser aceitas em tempo polinomial.

Problemas como achar o maior divisor comum e determinar se um dado número é primo ou não são exemplos de problemas que estão na classe de complexidade P.

## **CLASSE NP**

**PRIMEIRO SLIDE**

Outra classe de complexidade é a chamada classe NP, que significa polinomial não determinística. Essa classe engloba os problemas que podem ser verificados em tempo polinomial por uma Máquina de Turing Não Determinística e foi apresentada por Edmond em 1965 em seu artigo publicado naquele ano. Uma definição mais rigorosa relacionada ao conceito de linguagem define que, dada uma linguagem L, esta pertence à classe NP se, e somente se, existem duas entradas A, um algoritmo em tempo polinomial, e c, uma constante, tal que o algoritmo A verifique a linguagem L em tempo polinomial, com função de complexidade dada por O(|x|^c)

Cormen afirma que, do conhecimento de que toda Máquina de Turing determinística é,também, uma Máquina de Turing não determinística com, no máximo, um movimento, podemos dizer que P está contido na classe NP.

**SEGUNDO SLIDE**

Ainda parafraseando Cormen, ele dá uma definição informal à um subconjunto da classe de problemas NP. Cormen diz que NP Completo é um problema da classe NP que é tão entre as aspas, difícil, quanto qualquer outro problema em NP. Formalmente, Sipser define como NP Completo uma linguagem L que satisfaz as duas condições mostradas à vocês. A primeira requer que L esteja no conjunto de complexidade NP. Já a segunda, determina que todo algoritmo em NP seja redutível à L em tempo polinomial.

O conceito de NP Completo é muito importante para um assunto que iremos tratar a seguir e que, até hoje, intriga a comunidade científica que busca uma resposta. Esse conceito foi apresentado pela primeira por Cook em 1971 que, em seu artigo, mostrou que qualquer problema de reconhecimento que pode ser **resolvido** em tempo polinomial por uma Máquina de Turing Não Determinística pode ser entre aspas, reduzido, ao problema de satisfação de uma sentença, que testa se uma fórmula booleana é satisfeita ou não.

* **Consequências da NP Completude**

NP completo não só é uma definição muito importante como também nos traz algumas consequências. A primeira é que, se qualquer problema NP Completo puder ser resolvido em tempo polinomial, então qualquer problema em NP tem um algoritmo em tempo polinomial, ou seja a classe de complexidade P é igual a classe de complexidade NP.

Outra consequência vem da equivalência da anterior, ou seja, se algum problema NP não puder ser resolvido em tempo polinomial, então nenhum problema NP Completo pode ser resolvido em tempo polinomial.

* **NP Hard**

Lembrando a definição dada anteriormente, uma linguagem L é NP-Completo se

1. L está em NP;
2. Todo algoritmo em NP é redutível à L em tempo polinomial.

Se uma linguagem L satisfaz a condição 2, mas não necessariamente a primeira, nós falamos que L é NP-Hard.

* **EXEMPLO DE PROBLEMA NP COMPLETO**

O problema de encontrar ciclos hamiltonianos é NP Completo. Basicamente, um ciclo hamiltoniano de um grafo não-direcionado G é um simples ciclo que contém cada vértice de V. Um grafo que contém um ciclo desse tipo é dito hamiltoniano.

Um problema muito famoso, mas que poucos sabem que pertence à classe dos problemas NP Completo, é o problema do Caixeiro Viajante, ou, em inglês, TSP. Basicamente, dado um grafo completo G com N vértices, o objetivo é procurar o trajeto com menor custo possível passando por todos os vértices apenas uma vez e retornando ao vértice de origem.

Iremos apresentar outros problemas a seguir mas, como o TSP é um problema famoso, iremos apresentar a prova presente no livro introdução aos Algoritmos, do Cormen, de que o TSP é um problema NP-Completo.

Primeiro, nós mostraremos que o TSP pertence à classe de complexidade NP. Dada uma instância do problema, nós usamos como certificado a sequência de n vértices no trajeto. O algoritmo de verificação checa que a sequência contém um vértice exatamente uma vez, soma os custos das arestas percorridas e checa se a soma é, ao menos, um valor k. Esse processo pode certamente ser feito em tempo polinomial.

Para provarmos que o TSP é NP-hard, nós mostramos que um ciclo hamiltoniano ( [path](https://en.wikipedia.org/wiki/Path_(graph_theory)) in an undirected or directed graph that visits each [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) exactly once que é comprovadamente um problema NP-Completo, mas que não iremos provar aqui) TSP. Seja G um grafo com um conjunto de arestas E e de vértices V uma instância de um ciclo hamiltoniano. Nós construímos uma instância do TSP como segue. Nós formamos um grafo completo G’ com um conjunto de vértices V e um conjunto de arestas E’, onde E’ = {(i,j) : i,j pertence à V e ‘i’ é diferente de ‘j’} , e definimos a função de custo c como:

c(i,j) = 0, se (i,j) pertence à E e c(i,j) = 1 se (i,j) não pertence à E.

Note que, por G ser não-direcionado, ele não tem ciclos, então c(v,v) =1 pra todos os vértices v pertencentes ao conjunto de vértices V. A instância do TSP é, então, (G’, c, 0), que pode facilmente ser criado em tempo polinomial.

Agora, nós mostramos que o grafo G tem um ciclo hamiltoniano se, e somente se, o grafo G’ tem um trajeto de custo 0, no mínimo. Suponha que o grafo G tem um ciclo hamiltoniano h. Cada aresta em h pertence a E e então tem custo 0 em G’. Então, h é um trajeto em G’ com custo 0. inversamente, suponha que o grafo G’ tem um trajeto h’ de custo no mínimo 0. Como os custos das arestas em E’ são 0 e 1, o custo do trajeto h’ é exatamente 0 e cada aresta no trajeto deve ter custo 0. Portanto, h’ contém apenas arestas em E. Nós concluímos que h’ é um ciclo hamiltoniano em G e, portanto, temos o nosso trajeto, além de mostrar que o TSP é NP-Completo.

**SEGUNDO SLIDE**

Outro problema que também pertence à classe NP Completo é o problema de cobertura de nós. Basicamente, se G é um grafo não direcionado, uma cobertura de nós de G é um subconjunto de nós onde cada aresta de G possui um desses nós. O problema da Cobertura de Nós verifica se um dado grafo G contém uma cobertura de nós com tamanho específico.

**TERCEIRO SLIDE**

Dado um grafo não direcionado G, com um conjunto de vértices V e um conjunto de arestas E, um clique nada mais é do que o subconjunto V’ próprio do conjunto de vértices V, com cada par seu conectado por uma aresta do conjunto E. Em suma, um clique é um subgrafo completo de G. O tamanho de um clique é o número de vértices que ele contém

Basicamente, o problema Clique é o problema de otimização de encontrar um clique com tamanho máximo em um grafo. Como é um problema de decisão, a pergunta feita é se, dado um tamanho , existe um clique desse tamanho no grafo.

**QUARTO SLIDE**

Os problemas NP são a base da criptografia. O algoritmo mais popular de chave pública, o RSA, é longamente baseado no problema de fatoração e na dificuldade de se fatorar números inteiros muito grandes. Entretanto, se em algum momento fatorar números grandes se tornar uma tarefa fácil, então, quebrar o sistema de criptografia RSA se tornará fácil, como dito por (Cormen, 2009).

Fatoração significa dividir um número não-primo entre um certo número de primos. Por exemplo, fatorar 6 resulta em 2 multiplicado por 3. É muito difícil fatorar números muito grandes, mas se você possuir dois fatores, fica mais fácil de checar se eles são os fatores de um número maior se você multiplicar esses fatores.

Apesar de sabermos que a fatoração de números inteiros está na classe NP, não sabemos ainda se é NP Completo.

* **HIPÓTESE P =/= NP**

**PRIMEIRO SLIDE**

Um dos assuntos que mais intriga a comunidade científica é a hipótese de a classe de complexidade P ser, ou não, igual a classe NP. Esse é um dos 7 problemas selecionados pelo Instituto de Matemática Clay que valem um milhão de dólares, os chamados *Millenium Prize Problems*. Com constata Sipser, se uma Linguagem L é NP-Completo e L pertence à classe P, então a classe de complexidade P é igual a classe de complexidade NP.

**SEGUNDO SLIDE**

Se a hipótese de que P é igual à NP for confirmada, isso implicaria que qualquer problema verificado em tempo polinomial poderia ser, também, decidido em tempo polinomial, ou seja, sua solução pode ser encontrada também em tempo polinomial.

A principal relação dessa hipótese com o tema dessa apresentação é que, se a hipótese de que, na verdade, P não é igual a NP seja confirmada, isso significaria que os problemas de complexidade NP-Completo são intratáveis, como constata Sipser.

Sipser também cita que muitos teóricos da ciência da computação acreditam que P não é igual a NP. Cormen relata que, talvez, a razão que leva os teóricos à acreditarem nisso vem da existência de problemas da classe NP-Completo e que, após anos de estudo, nenhum algoritmo em tempo polinomial foi descoberto para algum problema NP-Completo. Mas, como dito anteriormente, até agora esse é um dos maiores problemas no campo científico.

**CURIOSIDADES**

Agora, falaremos sobre algumas curiosidades sobre as possíveis conclusões sobre a hipótese de P ser igual, ou não, à NP.

**CRIPTOGRAFIA**

Já que citamos a criptografia, iremos falar do que aconteceria caso a hipótese de que P = NP fosse comprovada. Muitos acreditam que, no momento que essa hipótese fosse aceita, significaria que a segurança da maioria das aplicações estaria imediatamente comprometida, o que não é verdade. O que acontece, é que, se a hipótese for comprovada, teremos a certeza de que poderíamos resolver os problemas em NP em tempo polinomial, como, por exemplo, a fatoração de números grandes, a qual o sistema de criptografia RSA é fortemente baseado. Mas isso não necessariamente significa que nós já conheceríamos um algoritmo que poderia resolver os problemas em NP (ou, agora, em P) tempo polinomial.

Ainda, ser resolvido em tempo polinomial não implica que esse tempo seja curto. O tempo polinomial apenas indica que a função de complexidade de tempo é polinomial, mas um polinômio não necessariamente é pequeno. Ou seja, poderíamos, ainda, levar muito tempo para podermos resolver algum problema NP, mas ainda estaríamos levando um tempo polinomial para tal.

**E SE P=NP?**

Russell Impagliazzo apresenta em seu artigo *A Personal View of Average-Case Complexity*, de 1995, um mundo chamado Algorithmica, onde é sabido que P = NP. Russel nos apresenta sua visão sobre as implicações dessa igualdade dizendo que, por exemplo, quase qualquer tipo de problema de otimização poderia ser fácil e automático de se resolver, linguagens de programação não iriam necessitar envolver instruções sobre *como* o computador deveria desempenhar, mas, sim, especificar as propriedades que uma saída deveria ter em relação à uma entrada. Abre aspas, se a linguagem de especificação é tal que é fácil avaliar se uma saída está de acordo com a especificação, então o compilador poderia automaticamente abastecer o algoritmo para resolver o problema NP para gerar a saída. Sem nos aprofundarmos, Russel ainda diz que o fato de P ser igual a NP tornaria trivial muitos aspectos de programas da inteligência artificial que são desafiadores na vida real.

Matemática auxiliada por computador se tornaria uma frase redundante, diz Russel, já que os computadores poderiam encontrar provas para qualquer teorema em tempo aproximado ao tamanho da prova. Ele ainda diz que, resumidamente, assim que um algoritmo viável para um problema NP-Completo for encontrado, a capacidade dos computadores se tornará aquela retratada na ficção científica.

Apesar das coisas boas, Russel ainda fala um pouco sobre algumas coisas ruins nesse mundo Algorithmica. Por exemplo, não existiria formas de diferenciar pessoas ou computadores por meio de métodos informacionais. Algoritmos de aprendizado poderiam simplesmente aprender a imitar o comportamento de outra máquina ou pessoa. Qualquer código desenvolvido poderia ser facilmente quebrado. Não faria muita diferença manter em segredo o código o qual um algoritmo se baseia, já que um algoritmo idêntico poderia ser automaticamente gerado a partir de números pequenos de exemplos de mensagens encriptadas e não-encriptadas. Além disso, não haveria maneira de permitir acesso humano à alguma informação sem permitir que esse acesso fosse disponibilizado para todos. Em particular, qualquer arquivo ou informação remotamente acessível através de um canal possivelmente seguro seria, basicamente, disponivel publicamente. Russel deixa claro que essas implicações assumem que nenhuma propriedade física é diretamente observável à distância, ou seja, que ninguém estaria inspecionando e que isso pode não ser verdade.

**E se P =/= NP?**

Outro mundo apresentado por Russel é chamado, por ele, de Heuristica. Nesse mundo, é de conhecimento que os problemas NP são intratáveis no pior caso, ou seja, a classe de complexidade P é comprovada ser diferente de NP, mas são tratáveis em média para qualquer distribuição amostrável.

Ele começa dizendo que esse mundo é paradoxal em algum sentido. Nesse mundo, existem instâncias dificeis de problemas NP, ou seja, os problemas NP-Hard, mas encontrá-los por si só é um problema intratável…

Em relação a segurança das redes e da criptografia, Russel afirma que não haveria muito diferença entre os mundos de Algorithmica, onde P é igual a NP, e Heristica, onde P é diferente de NP. Não ajudaria muito se os usuários legítimos gastassem muito tempo pensando em problemas para garantir a segurança as aplicações, se os interceptadores de comunicações entre dois agentes, isto é, entre remetente e receptor, pudessem resolver os problemas em períodos comparáveis de tempo, ou seja, se qualquer “intruso” no canal de comunicação entre dois agentes pudesse quebrar o código de segurança em períodos comparáveis de tempo.