CLASSIFICAÇÃO DE PROBLEMAS

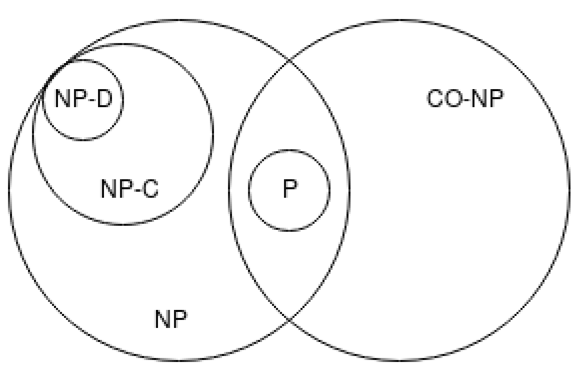
Problemas podem ser classificados quanto ao seu tamanho, medido pelo tamanho da representação de sua entrada. Pode-se também classificá-los quanto à sua complexidade. Por fim, classifica-se eles quanto às suas saídas. Neste último caso, há três classificações possíveis:

- Problemas de decisão
  - Sua resposta é sim ou não.
  - A complexidade de um algoritmo é sempre baseada neste tipo de problema.
  - Dado um problema de decisão, um certificado deste problema para uma resposta é a prova (teorema) desse resultado.
  - A pergunta se um número é primo é um problema de decisão.
  - Saber se uma rede está conectada é um problema de decisão.
- Problemas de localização
  - Buscam localizar alguma estrutura ou propriedade na entrada dada.
  - "Qual o risco?" é um problema de localização.
  - Localizar um ponto congestionado numa rede é um problema de localização.
  - Descobrir o caminho de um ponto A a um ponto B é um problema de localização.
- Problemas de otimização
  - Buscam encontrar estrutura ou propriedade que sejam resposta ótima a um problema.
  - Podem ser transformados em um problema de decisão definindo-se um parâmetro k como base para o valor que queremos otimizar; assim, "qual o melhor caminho entre A e B" torna-se "existe caminho entre A e B com distância máxima igual a k?"
  - Saber o melhor caminho é um problema de otimização.
  - Colorir vértices de um grafo com o menor número de cores possíveis, também.

CLASSE DE PROBLEMAS

- **Classe P**: resolvidos em tempo polinomial.
  - Para uma entrada n, costumam levar os tempos $\log n < \log^k n < \sqrt{n} < n < n \log n < n^2 < n^k < 2^n < n! < n^n$. Os três últimos tempos são completamente não eficientes.
  - São tratáveis.
  - Divisão e conquista e método guloso são técnicas para soluções de problemas P.

- **Classe L:** são resolvidos em tempo logarítmico de memória, ou seja, para uma entrada *n*, costumam levar os tempos log$n$ < log$^k n$.
- **Classe NP**: são não-deterministicamente polinomiais.
  - Um problema de decisão π pertence à classe NP se existe um algoritmo determinístico que o resolva emitindo um certificado para a resposta "SIM" tal que esse certificado possa ser verificado em tempo polinomial.
  - São O(2^n), razão pela qual não são polinomiais.
  - Se um algoritmo para provar está correto, o próprio algoritmo é o certificado NP.
  - São problemas intratáveis, o que significa que o melhor algoritmo exato que existe é inaplicável para problemas deste tipo.
  - Dois problemas puramente NP, não redutíveis à P, são a fatoração e o logaritmo discreto.
- **Classe CO-NP:** Caso exista um algoritmo determinístico que resolva um problema emitindo um certificado para a resposta "NÃO" tal que esse certificado possa ser verificado em tempo polinomial, este problema pertence à classe CO-NP.
- **Classe NP-Completo**: Uma redução polinomial de problemas NP é o NP-Completo.
  - É também intratável, embora sejam contornáveis com algumas técnicas.
  - São O(2^n), razão pela qual não são polinomiais. No caso, porém, ao se utilizar alguma das técnicas previamente mencionadas, podem melhorar.
  - Exemplos de problemas incluem: satisfabilidade ("dada uma fórmula lógica proposicional simples, ela possui uma valoração que resulta em verdadeiro?") para quando a entrada for maior que 3; caixeiro viajante; problema da mochila; coloração de grafos; clique máxima; conjunto independente; cobertura de vértice.
  - Programação dinâmica e *backtracking* são técnicas usadas para resolver problemas NP-Completos.
- **Classe NP-Difícil**: São pelo menos tão difíceis quanto problemas NP-Completos.
  - É dito intratável.



ALGUMAS DEFINIÇÕES MATEMÁTICAS

Seja Φ um problema:

- Para todo Φ pertencente ao conjunto NP, existe um problema π pertencente ao conjunto NP tal que Φ se reduza polinomialmente à π, sendo π pertencente ao conjunto dos problemas NP-Completos.
- Se Φ pertence ao conjunto P, então existe um algoritmo $O(n^k)$, ou seja, um algoritmo em tempo polinomial, para Φ.
- n^k está incluso em $O(2^n)$, mas 2^n não está incluso em $O(n^k)$.

DIVISÃO E CONQUISTA

- Algoritmo que serve para resolver problemas do conjunto P.
- Um exemplo típico é o *merge sort*.
- Consiste em dividir um problema em problemas menores, resolvê-los, então compor a solução do problema original a partir da solução dos problemas menores.
- Pode-se usar o teorema mestre para calcular a expressão geral de recorrência de um algoritmo de divisão e conquista. Com este valor, é fácil saber se vale a pena ou não aplicar o algoritmo, a depender da resposta. Na fórmula abaixo, T(n) é o problema; *a* é a quantidade de operações nos sub-problemas, *n/b* é a fração de problemas e $O(n^d)$ é o tempo gasto para juntar as soluções.

$$T(n) = aT*(\lceil n/b \rceil) + O(n^d)$$

$$T(n) = \left\{ \begin{array}{c} O(n^d) \ se \ d > \log_b a \\ O(n^d \log n) \ se \ d = \log_b a \\ O(n^{\log_b a}) \ se \ d < \log_b a \end{array} \right\}$$

- Em uma multiplicação de matrizes, pode-se fazer 8 (a=8) multiplicações quebradas em duas partes (n/b = n/2, pois cada multiplicação é de uma linha por uma coluna), e o tempo para juntar é O(1), o que faz com que a fórmula abaixo seja n^3, um tempo ruim:

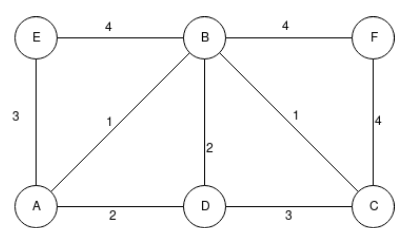
$$T(n) = 8T(\frac{n}{2}) + O(1) = T(n) = n^3$$

MÉTODO GULOSO

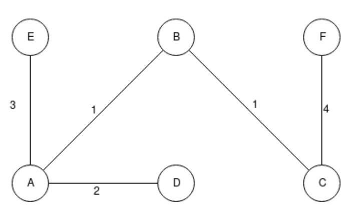
- Algoritmo que serve para resolver problemas P.
- Consiste em, para encontrar a solução de um problema de otimização, escolher o passo que parecer melhor localmente. No caixeiro viajante, por exemplo, ele escolheria o próximo vértice baseado naquele que fosse o de menor distância. No entanto, para percorrer todos os vértices, isto não necessariamente funciona bem, pelo contrário, costuma ser terrível.
- Na maioria das vezes, não leva à solução ótima.
- Em alguns casos, fornece uma aproximação ou limite inferior/superior razoável.
- Quando funciona, costuma ser o melhor algoritmo.
- O método guloso funciona bem para o problema do rolo de moedas. Por exemplo, para dar R$0,73 de troco na menor quantidade possível de moedas, ele pegaria uma de R$0,50; duas de R$0,10 e três de R$0,01.

ÁRVORE GERADORA MÍNIMA

- É uma aplicação do método guloso.
- A árvore geradora mínima é um grafo conexo sem ciclo, cuja soma do peso das arestas é o menor possível. Tome, por exemplo, o grafo abaixo:



- Para a árvore geradora mínima, primeiro pegamos as arestas de menor valor, que pode ser AB ou BC. Depois, pegamos outra de menor valor, e assim sucessivamente, até que tenhamos pisado em todos os vértices sem, porém, ter fechado um ciclo no grafo.
- Assim, a ordem da árvore geradora será: AB, BC, AD, AE e CF:



CÓDIGO DE HUFFMAN

- Outra das aplicações do método guloso.
- Para letras muito repetitivas em textos (como as vogais *a* e *e*), este algoritmo busca dar uma representação em *bits* com o menor número possível, deixando, para as letras menos utilizadas, números maiores.
- Sem paciência pra fazer árvores nem sou obrigado, então quem quiser, recomendo este vídeo: https://www.youtube.com/watch?v=Tl4x219Ri4k.

BACKTRACKING/FORÇA BRUTA (UI!)

- Algoritmo aplicado a problemas NP.
- Em si, o *backtracking* é como uma força bruta, mas ele reduz o conjunto de entradas para que o problema seja mais eficiente.
- Consiste em, para um conjunto de entrada qualquer, tentar resolver o problema pegando o próximo item do conjunto que possa resolver o problema. Caso não resolva, ele volta um item no conjunto de solução, e testa sucessivamente até resolver, e assim vai, retirando ou adicionando itens no conjunto de solução conforme eles façam parte ou não da solução.
- Por exemplo, no problema do percurso do cavalo, ele só testará as posições do quadrado para os quais o cavalo, que anda em L, possa ir.
- Além do algoritmo do cavalo, a permutação caótica é outro exemplo de problema resolvível com *backtracking*.
- A depender da posição inicial do cavalo, o *backtracking* pode se tornar cada vez mais lento para calcular o percurso, por conta da natureza do algoritmo.

PROGRAMAÇÃO DINÂMICA

- Algoritmo aplicado a problemas NP.
- A divisão e conquista tenta resolver problemas um pouco menores para, com eles, aplicar algo como uma fórmula geral para que se chegue ao problema maior. Assim, ela é diferente da divisão e conquista porque não resolve problemas menores e os junta para uma solução; ela resolve problemas menores até poder conseguir chegar ao maior.
- Soluções recursivas para Fibonacci são péssimas, mas a programação dinâmica é ótima: calcula-se problemas menores (fib(n-1), fib(n-2)), guarda-se seus resultados, e estes são usados para compôr o resultado maior.
- Outro exemplo de uso inclui a subsequência crescente mais longa.

DISTÂNCIA DE EDIÇÃO MÍNIMA

- Um exemplo de uso de programação dinâmica, é um algoritmo no qual se deseja transformar uma palavra na outra usando o mínimo possível de operações caractere a caractere. Entre as possíveis, estão adição, remoção, troca ou manutenção da letra. Todas as operações têm peso 1, exceto a última, de peso 0.
- Com as duas palavras, monta-se uma matriz. Supondo que sejam “maltar” e “marcio”, a matriz fica assim:

	?	M	A	R	C	I	O
?	0	1	2	3	4	5	6
M	1						
A	2						
L	3						
T	4						
A	5						
R	6						

- Feito isto, deve-se observar os três primeiros valores (0, 1 e 1). Tanto a operação de descer um número quanto trazê-lo do lado envolve um custo do número trazido + 1. Na diagonal, se as letras forem iguais, não acrescenta nada, apenas repete o número. Se as letras forem diferentes, o custo é 1 como nos casos anteriores.

- Assim, no caso das letras M e M, o zero da diagonal se mantém, e as demais linhas são puxadas do lado. O mesmo ocorrerá quando esbarrarem-se as letras A e A. No mais, o resto é só preencher a tabela, atentando-se para estes movimentos ou para letras iguais, como ocorre com as duas letras R do ma**R** com o ta**R**:

	?	M	A	L	T	A	R
?	0	1	2	3	4	5	6
M	1	0					
A	2						
R	3						
C	4						
I	5						
O	6						

	?	M	A	L	T	A	R
?	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
A	2	1	0	1	2	3	4
R	3						
C	4						
I	5						
O	6						

	?	M	A	L	T	A	R
?	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
A	2	1	0	1	2	3	4
R	3	2	1	1	2	3	3
C	4	3	2	2	2	3	4
I	5	4	3	3	3	3	4
O	6	5	4	4	4	4	4

- Assim, o número de edições mínimos é 4, pois este é o último número da matriz.

PROBLEMA DA MOCHILA

- Há uma mochila com capacidade máxima W, e *n* itens de peso W_1, W_2, … W_n de valor V_1...V_n. Deseja-se carregar o máximo de coisas valiosas no limite possível de capacidade.
- É um problema NP-Difícil. Sua versão de decisão (é possível conseguir um valor maior que k?) é um problema NP-Completo.
- Sua complexidade é O(*n* W), ou seja, exponencial, pois W é a entrada, e não o tamanho desta. O tamanho é log w, ou seja, a verdadeira complexidade é $O(n\ 2^{\log w})$. Daí o fato do problema da mochila ser chamado de pseudopolinomial.
- Usar o método guloso para resolver este problema dá o limite inferior do problema; não é o ideal, mas é uma solução. Idealmente, deve haver outras soluções.
- Para usar programação dinâmica, suponha, por exemplo, a tabela abaixo:

ITEM	PESO	VALOR
1	6	30
2	3	14
3	4	16
4	2	9

- Para resolver este problema COM REPETIÇÕES, faz-se um vetor de índices de 0 a W, sendo W o peso máximo da mochila. O índice (i), portanto, representará o peso máximo naquela etapa do algoritmo. O maior valor (v) possível será guardado dentro da posição vetor[peso.] Para cada índice (peso), deve-se verificar quais itens podem ser comportados neles ou não. Então, deve-se pegar o maior possível. Para isto, deve-se levar em conta se se pode usar valores repetidos ou não. A fórmula é:

$$k(w) = \max_{i: w_i \leq w} \left(k(w - w_i) + v_i \right)$$

- Assim, para a mochila descrita acima e um peso máximo W = 5, teríamos o máximo de valor (podendo pegar itens repetidos) igual a 30, pois:

Itens possíveis de pegar com o peso atual do índice	∅	∅	4	4 ou 2	2 ou 3 ou 4	2 ou 3 ou 4	Qualquer um dos 4
Valor máximo dentre os itens escolhidos	∅	∅	9	14	18	23	30
Índice (peso máximo atual)	0	1	2	3	4	5	6

- Para resolver SEM MULTIPLICAÇÕES, aí o bagulho complica. É necessário fazer uma matriz, de 0 a W, e de 0 a i, sendo i a quantidade de itens e W o peso máximo da mochila. Então, preenche-se tanto a primeira linha quanto a primeira coluna com 0s. Assim, para o mesmo problema anterior, teríamos uma matriz como abaixo (1). Preenchida com 0s a matriz, é hora de preencher o primeiro item. O item 1 tem peso 6. Logo, como não há W=6 na matriz, o item 1 nunca entrará, por isso a coluna do 1 será totalmente preenchida com 0 na matriz abaixo (2). Agora, repete-se o mesmo para o item 2, de peso 3. Assim, a partir de w=3, coloca-se o valor=14 do item 2, conforme a imagem abaixo (3):

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	0			
3	0	0			
4	0	0			
5	0	0			

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	0	0		
3	0	0	14		
4	0	0	14		
5	0	0	14		

- Para o item 3, a coisa muda um pouco de figura. Naturalmente, seu peso é 4, logo, ele só pode aparecer a partir de W=4. No entanto, em W=3, podemos pegar o valor 14 do lado. A matriz fica conforme abaixo (1.) O item 4 tem peso 2, portanto, a partir de W=2, podemos colocá-lo (matriz 2 abaixo.)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	0	0	0	
3	0	0	14	14	
4	0	0	14	16	
5	0	0	14	16	

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	
4	0	0	14	16	
5	0	0	14	16	

- Para o item 4 no peso 3, devemos ver se ou pegaremos o valor do lado (14), ou se pegaremos o valor da posição (peso – peso_atual, índice – 1) + valor do item. Ora, o peso atual é 2, e o índice é igual a 4. Naturalmente, pegar 14 é melhor que pegar o valor da posição (4-2,4-1)=(2,3)=0+9. Assim, a tabela fica conforme abaixo (1). O mesmo deve ser feito para w=4. Neste caso, mais uma vez será melhor pegar o valor do lado, 16, conforme abaixo (2).

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	
5	0	0	14	16	

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	

- No entanto, para o último item, vamos calcular (peso – peso_atual, índice – 1), ou seja, (5-2, 3)=(3,3)=14 + o valor do item atual, ou seja, 14 + 9 = 23. Naturalmente, este valor, 23, é maior do que 16, então é ele que será o escolhido. Daí concluímos que para o peso = 5, o maior valor que se pode levar sem repetição é 23, conforme abaixo:

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23

- Daí pode-se perceber que o algoritmo geral para escolher qual valor colocar em uma das casas da matriz segue como abaixo:
  - Se o elemento não entrou para o W atual, coloca-se:
    - M(w, i) = M(w, i-1) [o valor da posição ao lado esquerdo do elemento]
  - Se o elemento entrou:
    - M(w,i) = M(w – w_i, i – 1) + v_1 [o valor do item atual acrescido da posição (peso_total – peso_atual, índice de item atual – 1) da matriz]