Análise de Algoritmos
Técnicas de Projeto de Algoritmos

Nelson Cruz Sampaio Neto
nelsonneto@ufpa.br

Universidade Federal do Pará
Instituto de Ciências Exatas e Naturais
Faculdade de Computação

12 de novembro de 2019

Conteúdo

- Tentativa e erro (ou força bruta).
- Divisão e conquista.
- Estratégia gulosa.
- Programação dinâmica.
- Algoritmos aproximados.

Força bruta

- É a mais simples das estratégias de projeto.
- Pode ser definida como:

  Uma solução prática para resolver um problema, geralmente baseada diretamente no enunciado do problema em questão e nas definições dos conceitos envolvidos.

- A força é de um computador e não do intelecto de alguém, ou seja, segue o princípio do "somente faça".
- Em função da simplicidade de implementação, a estratégia de força bruta é uma das mais fáceis de aplicar.

Força bruta

- Apesar de ser raramente uma fonte de algoritmos refinados, a força bruta é uma importante estratégia de projeto.
- Para alguns problemas importantes (p.e. ordenação, busca, multiplicação de matrizes, casamento de cadeias, etc.), a técnica de força bruta fornece:
  - Algoritmos simples e de fácil implementação.
  - Algoritmos de valor prático.
  - Algoritmos com limitações quanto ao tamanho da entrada.

Força bruta

- Uma primeira aplicação da estratégia força bruta geralmente resulta em um algoritmo que pode ser melhorado com uma quantidade modesta de esforço.
- O esforço para projetar um algoritmo mais eficiente pode não ser justificável se somente entradas controladas do problema necessitarem ser processadas.
- Em outras palavras, mesmo pouco eficiente (ou até mesmo ineficiente), essa técnica pode ser útil para resolver problemas com instâncias pequenas.

Exemplo: Ordenação por seleção

- É conhecido como o método mais direto para resolver o problema de ordenação.
- Começamos varrendo o arranjo inteiro para encontrar o menor elemento e permutá-lo com o primeiro elemento.
- Então, varremos a lista, começando pelo segundo elemento, para encontrar o menor dentre os $n - 1$ últimos elementos e permutá-lo com o segundo elemento... e assim por diante.

Exemplo: Ordenação por seleção

- Generalizando, na i-ésima passagem pelo arranjo, o algoritmo procura pelo menor item entre os $n - i$ últimos elementos e o permuta com A_i. Após $n - 1$ passos, o vetor está ordenado.

```
SELECAO (A, n)
1. para i = 1 até n - 1 faça
2.    menor = i
3.    para j = i + 1 até n faça
4.        se A[j] < A[menor] então menor = j
5.    aux = A[menor]
6.    A[menor] = A[i]
7.    A[i] = aux
```

- A eficiência $\Theta(n^2)$ é dada pelo número de comparações realizadas na linha 4.

Exemplo: Ordenação por seleção

- O método é ilustrado abaixo. As chaves em negrito sofreram uma troca entre si.

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
i = 1	**A**	*R*	*D*	*E*	*N*	**O**
i = 2	*A*	**D**	**R**	*E*	*N*	*O*
i = 3	*A*	*D*	**E**	**R**	*N*	*O*
i = 4	*A*	*D*	*E*	**N**	**R**	*O*
i = 5	*A*	*D*	*E*	*N*	**O**	**R**

Exemplo: Busca sequencial ou linear

- Compara elementos sucessivos de uma dada lista com uma dada chave de busca até:
    - Encontrar um elemento similar (busca bem sucedida) ou
    - A lista ser exaurida sem encontrar um elemento similar (busca mal sucedida).
- A busca sequencial fornece uma ilustração excelente da força bruta, com sua principal qualidade (simplicidade) e fraqueza (lentidão).

Exemplo: Busca sequencial ou linear

- Pseudocódigo do algoritmo de busca linear:

```
LINEAR (A, n, x)
1. i  =  1
2. enquanto (i <= n  e  x != A[i]) faça
3.    i = i + 1
4. se (i <= n)
5.    então escreve ("encontrado")
6.    senão escreve ("ausente")
```

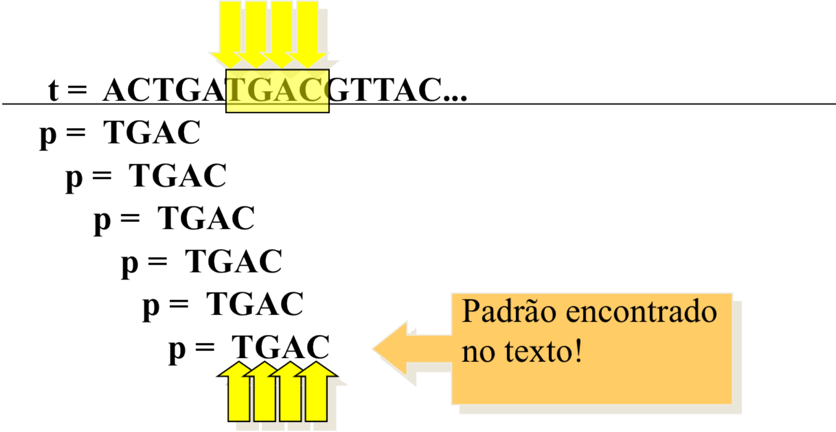
- A eficiência $O(n)$ é dada pelo número de incrementações realizadas na linha 3.

Exemplo: Casamento de cadeias

- Dada uma cadeia T de n caracteres chamada de texto, e uma cadeia P de m caracteres ($m \leq n$) chamada de padrão, a ideia é encontrar partes do texto que "coincidam" com o padrão.

```
STRING_MATCH (T, n, P, m)
1. para (i = 1) até (n - m + 1) faça
2.     j = 1
3.     k = i
4.     enquanto (P[j] = T[k] e j <= m) faça
5.         j = j + 1
6.         k = k + 1
7.     se (j > m)
8.         retorna (i)  % padrão encontrado
9. retorna (-1)
```

- O algoritmo é ilustrado abaixo.



Exemplo: Casamento de cadeias

- O pior caso acontece quando o algoritmo tem que comparar todos os m caracteres antes de deslocar-se e isso pode ocorrer para cada uma das $n - m$ tentativas.

```
t =  x x x x x x x x x x x x x x x x x x x x
--------------------------------------------
p =  x x x x x x x x y
  p =  x x x x x x x x y
    p =  x x x x x x x x y
      p =  x x x x x x x x y
        ...
```

- Logo, a eficiência $O(nm)$ é dada pelo número de operações realizadas nas linhas 5 e 6.

Força bruta

Como visto, o método força bruta fornece algoritmos simples e aplicáveis, mas que podem ser facilmente melhorados:

- A ordenação por seleção foi "substituída" por algoritmos da ordem $O(n \log n)$, como QuickSort (caso médio e melhor caso) e HeapSort.

- A complexidade da busca linear é superior assintoticamente a pesquisa binária, que requer $O(\log n)$ comparações.

- Com relação ao casamento de cadeias, existem algoritmos bem mais eficientes que a força bruta, por exemplo, o Shift-And com custo $O(n)$ e o BMH com caso médio em $O(n/m)$.

Prós e contras

- **Prós**:
  - Ampla aplicabilidade.
  - Fornece algoritmos simples para problemas importantes:
    - ordenação e busca;
    - casamento de cadeias;
    - multiplicação de matrizes;
    - soma/produto de n número;
    - encontrar o maior (ou menor) elemento em uma lista;
    - ...
- **Contras**:
  - Pouco criativa quando comparada com outras técnicas.
  - Raramente fornece algoritmos competitivos, sendo alguns até inaceitavelmente vagarosos dependendo do volume de dados que ele precisa analisar.

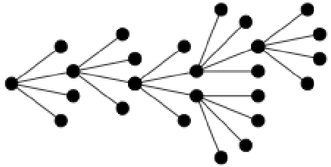
- **Decisão**: Problemas computacionais que almejam encontrar um elemento com propriedades especiais em um domínio que cresce com o tamanho da instância.

- **Otimização**: Problemas ainda mais complexos que buscam encontrar uma solução máxima (*max*) ou mínima (*min*) em relação a algum critério.

- A dificuldade surge quando o algoritmo tem que realizar essa pesquisa em um grande (possivelmente exponencial) conjunto de possibilidades.

Backtracking e branch-and-bound

- A técnica de projeto *backtracking* é uma maneira de construir um algoritmo força bruta para um problema de decisão.
- A ideia básica é decompor o processo em um número finito de subtarefas parciais que devem ser exploradas exaustivamente em busca de soluções válidas.
- O processo de tentativa gradualmente constrói e percorre uma árvore de subtarefas de forma recursiva.



- *Backtracking* incrementalmente constrói candidatas de soluções e abandona uma candidata parcialmente construída tão logo quanto for possível determinar que ela não pode gerar uma solução válida.

- Quando aplicável, *backtracking* é frequentemente mais rápido que algoritmos de enumeração total (força bruta tradicional), já que ele pode eliminar um grande número de soluções inválidas com um único teste.

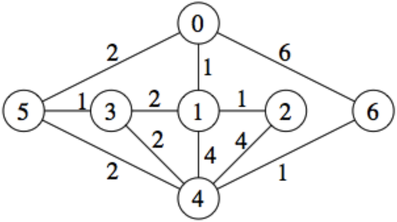
- Faz uso da busca em profundidade.

Exemplo: Labirinto

- Dado um labirinto, encontre um caminho da entrada à saída.
- Em cada interseção, você tem que decidir se:
    - Segue direto;
    - Vai para a esquerda;
    - Vai para a direita.
- Não há informação suficiente para escolher corretamente.
- Cada escolha leva a outro conjunto de escolhas.
- Uma ou mais sequência de escolhas pode ser a solução.

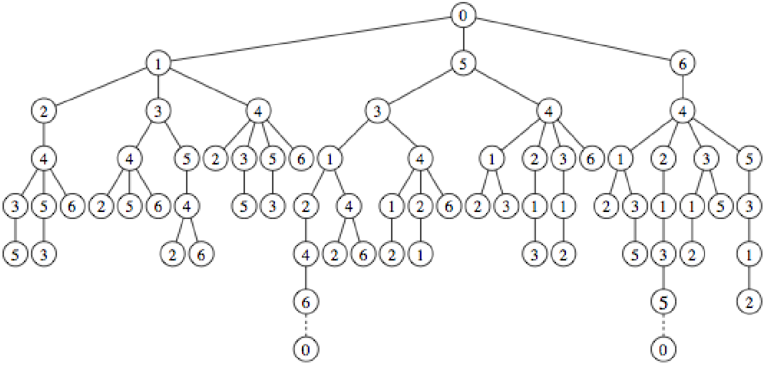
Exemplo: Ciclo Hamiltoniano

- Encontrar um trajeto que passe por todos os vértices de um grafo exatamente uma vez antes de retornar a origem.
- O algoritmo para caminhamento em um grafo faz uma busca em profundidade em tempo $O(|V| + |A|)$.
- Obter um algoritmo tentativa e erro modificando a busca em profundidade para tentar todos os caminhos possíveis.



	1	2	3	4	5	6
0	1				2	6
1		1	2	4		
2				4		
3				2	1	
4					2	1
5						

- A árvore de caminhamento é:



- Existem duas respostas: [0 5 3 1 2 4 6 0] e [0 6 4 2 1 3 5 0].

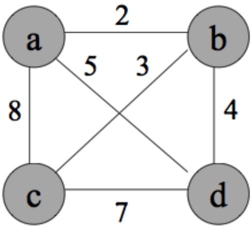
- Para um grafo completo, que contém arestas ligando todos os pares de vértices, existem $O(n!)$ ciclos simples.

- O algoritmo de *backtracking* é útil para problemas de decisão, mas não foi planejado para problemas de otimização.
- Mesmo assim, podemos estender o algoritmo de *backtracking* para trabalhar com problemas de otimização.
- Ao fazer isso, obteremos o padrão de algoritmos chamado de *branch-and-bound*.
- O *branch-and-bound* não termina ao achar a primeira solução. Ele continua até a **melhor** solução ser encontrada.
- O algoritmo tem um mecanismo de pontuação, que encerra a tentativa tão logo se saiba que a mesma não levará a uma solução melhor (problemas de minimização no caso).

Exemplo: Caixeiro viajante

- Dadas n cidades com distâncias conhecidas entre cada par, encontrar o trajeto mais curto que passe por todas as cidades exatamente uma vez antes de retornar a cidade de origem.

- Alternativamente: encontrar o ciclo Hamiltoniano mais curto em um grafo conectado e ponderado.

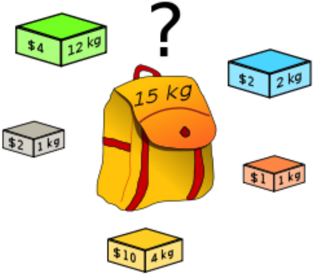


Trajeto	Custo
• a→b→c→d→a	2+3+7+5 = 17
• a→b→d→c→a	2+4+7+8 = 21
• a→c→b→d→a	8+3+4+5 = 20
• a→c→d→b→a	8+7+4+2 = 21 *
• a→d→b→c→a	5+4+3+8 = 20
• a→d→c→b→a	5+7+3+2 = 17

- Solução ineficiente: $O(n!)$

Exemplo: Problema da mochila

- Preencher uma mochila com objetos de diferentes pesos e valores. O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando seu peso máximo.



Solução: três caixas amarelas e três caixas cinzas; se apenas as caixas mostradas estiverem disponíveis, então todos, menos a caixa verde.

- Solução: Obter todos os subconjuntos ($= 2^n$) do conjunto de n itens dados, calculando o peso de cada um e separando os praticáveis. No fim, encontrar um subconjunto com o valor mais elevado entre eles.

1. Listar todas as soluções potenciais para o problema de uma maneira sistemática.
    - Todas as soluções são eventualmente listadas.
    - Nenhuma solução é repetida.
2. Avaliar as soluções, uma a uma, eliminando as não práticas e mantendo a melhor encontrada até o momento.
3. Quando a busca terminar, anunciar o vencedor.

Considerações

- Algoritmos de força bruta são executados em uma quantidade de tempo realista somente para instâncias pequenas.
- Em alguns casos, existem alternativas muito melhores, porém, em outros, a força bruta (ou variação) é a única opção para encontrar a solução exata.
- Viu-se que tanto para o problema do caixeiro viajante quanto da mochila, a força bruta leva a algoritmos ineficientes.
- Por outro lado, a computação ainda não conhece algoritmos eficientes para resolvê-los.
- Esse tipo de problema é conhecido como *NP*-Difícil e será melhor definido mais a frente.

Divisão e conquista

- O uso desse paradigma geralmente leva a soluções eficientes e elegantes, em especial quando é utilizado recursivamente.

- **Ideia:**

    1. **Dividir** o problema em um ou mais subproblemas.

    2. **Conquistar** os subproblemas, resolvendo-os recursivamente. Se os tamanhos dos subproblemas forem pequenos o bastante, basta resolvê-los de forma direta.

    3. **Combinar** as soluções encontradas dos subproblemas, a fim de forma uma solução para o problema original.

Exemplo: Máximo e mínimo

- Considere o algoritmo abaixo para obter o maior e o menor elemento de um vetor de inteiros $v[1..n]$, $n \geq 2$.

```
   MAXMIN (A, low, high)
      if (high - low + 1 = 2) then
         if (A[low] < A[high]) then
            max = A[high]; min = A[low]
         else
            max = A[low]; min = A[high]
         end if
      else
         mid = (low + high)/2
         (max_l, min_l) = MAXMIN(A, low, mid)
         (max_r, min_r) = MAXMIN(A, mid + 1, high)
         Set max to the larger of max_l and max_r
         Set min to the smaller of min_l and min_r
      end if
      return((max, min))
```

Exemplo: Máximo e mínimo

- Seja $T(n)$ uma função de complexidade tal que $T(n)$ é o número de comparações entre os n elementos de v.

  Logo,

  $$\begin{aligned} T(n) &= \Theta(1), && \text{para } n \leq 2 \\ T(n) &= T(n/2) + T(n/2) + \Theta(1), && \text{para } n > 2 \\ T(n) &= 2T(n/2) + \Theta(1), && \text{para } n > 2 \end{aligned}$$

- De acordo com o Teorema Mestre, o algoritmo é eficiente, com complexidade no tempo $T(n) = \Theta(n)$.

Exemplo: Máximo e mínimo

- O algoritmo MAXMIN recursivo tem o mesmo comportamento assintótico de um simples algoritmo iterativo.
- No entanto, a tendência é que o MAXMIN recursivo apresente um pior desempenho na prática.
- A cada chamada do procedimento, o compilador salva em uma estrutura de dados os valores de *low*, *high*, *min* e *max*, além do endereço de retorno da chamada para o procedimento.
- **Devemos evitar uso de recursividade quando existe uma solução óbvia por iteração**.

- O algoritmo MAXMIN divide o problema em subproblemas de mesmo tamanho.
- Este é um aspecto importante no projeto de algoritmos:

  Procurar sempre manter o **balanceamento** na subdivisão de um problema em partes menores.
- No problema de ordenação, o efeito provocado pelo princípio do balanceamento é bem nítido.
- Como exemplo, vamos analisar o comportamento do algoritmo de ordenação QuickSort.

Exemplo: QuickSort

- O QuickSort é o algoritmo de ordenação mais rápido para a maioria das aplicações práticas existentes.
- A chave do QuickSort é uma rotina que escolhe o elemento pivô e o coloca na sua posição correta.
- Após particionar o vetor em dois (elementos à esquerda e à direita do pivô), os segmentos são ordenados recursivamente, primeiro o da esquerda e depois o da direita.
- Seu tempo de execução depende do particionamento do vetor: balanceado (melhor caso) ou não balanceado (pior caso).

Particionamento não balanceado

- O comportamento do QuickSort no pior caso ocorre quando a sua rotina interna produz um segmento com $n-1$ elementos e outro com 0 (zero) elementos em cada nível recursivo.
- Nesse caso, a fórmula de recorrência para o algoritmo assume a seguinte forma:

  $$\begin{aligned} T(0) &= \Theta(1) \\ T(1) &= \Theta(1) \\ T(n) &= T(n-1) + T(0) + \Theta(n) \\ T(n) &= T(n-1) + \Theta(n), \text{ para } n > 1 \end{aligned}$$

- Resolvendo a formulação acima, obtém-se $O(n^2)$.
- Por exemplo, quando o vetor de entrada já está ordenado e o pivô é o último elemento.

- O comportamento no melhor caso ocorre quando a sua rotina interna produz dois segmentos, cada um de tamanho não maior que a metade de n em cada nível recursivo.

- Nesse caso, a fórmula de recorrência para o algoritmo assume a seguinte forma:

  $T(1) = \Theta(1)$

  $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + \Theta(n)$

  $T(n) = 2T(\frac{n}{2}) + \Theta(n)$, para $n > 1$

- Resolvendo a formulação acima, obtém-se $O(n \log n)$.

- Por exemplo, quando o vetor de entrada já está ordenado e o pivô é o elemento do meio.

Considerações

- A divisão e conquista é uma técnica recursiva e, por isso, descendente, em que o balanceamento é útil.
- Decompõe sucessivamente um problema em subproblemas independentes triviais, resolvendo-os e combinando suas soluções em uma solução para o problema original.
- Por exemplo, no QuickSort, o balanceamento dos tamanhos dos subproblemas levou a um resultado muito superior.
- Saiu-se de uma complexidade $O(n^2)$ para uma complexidade $O(n \log n)$ considerando valores grandes de n.

Estratégia gulosa

- Os algoritmos gulosos são tipicamente usados para resolver problemas de otimização.
- **Ideia:** Quando temos uma escolha a fazer, fazemos aquela que pareça ser a melhor no momento.
- Ou seja, fazemos uma escolha ótima local, na esperança de obter uma solução ótima global.
- Os algoritmo gulosos nem sempre levam a uma solução ótima, mas as vezes sim.
- Exemplos: árvore geradora mínima, caminho mínimo a partir de uma fonte, compressão/codificação de dados.

Estratégia gulosa

- A estratégia gulosa sugere construir uma solução através de uma sequência de passos, cada um expandindo uma solução parcialmente construída até o momento, até uma solução completa para o problema ser obtida.

- Em cada passo - e este é o ponto central desta técnica - a escolha deve ser feita:

  - **Possível**: Deve satisfazer as restrições do problema;

  - **Localmente ótima**: Deve ser a melhor escolha local dentre todas as escolhas disponíveis naquela passo;

  - **Irreversível**: Uma vez feita, ela não pode ser alterada nos passos subsequentes do algoritmo.

Estratégia gulosa

- **Problema geral**: Dado um conjunto C, determine um subconjunto $S \subseteq C$ tal que:
  - S satisfaz uma dada propriedade P, e
  - S é mínimo (ou máximo) em relação a algum critério α.
- O algoritmo guloso para resolver o problema geral consiste em um processo iterativo em que S é construído adicionando-se ao mesmo elementos de C um a um.

Algoritmo guloso genérico

- Quando o algoritmo guloso genérico funciona corretamente, a primeira solução encontrada é sempre ótima.

```
Conjunto guloso (C)
// C : conjunto de candidatos
S = 0   // S é o conjunto solução
enquanto (C não for vazio)
    x = seleciona (C)
    C = C - x
    se ((S + x) for viável) então S = S + x
se (solução (S))
    então retorna (S)
    senão retorna ("Não existe solução")
```

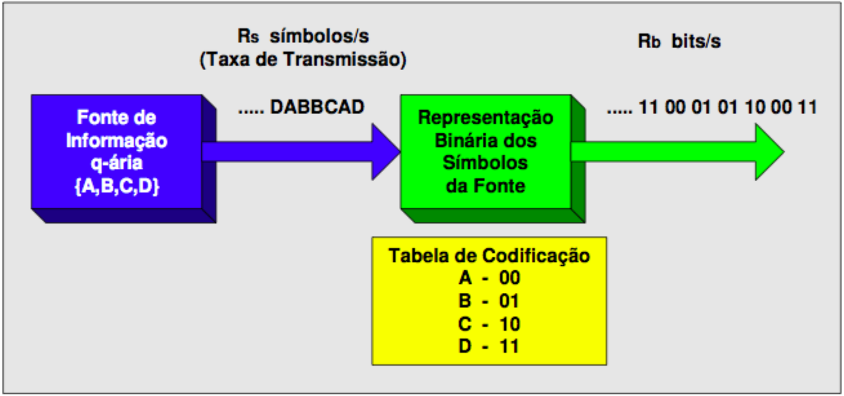
Algoritmo guloso genérico

- No início, o conjunto S de candidatos escolhidos está vazio.
- A cada passo, o melhor candidato restante ainda não tentado é considerado. O critério de escolha é ditado por uma função de seleção relacionada com um objetivo (max/min).
- Se o conjunto aumentado de candidatos se torna inviável, o candidato é rejeitado. Senão, o candidato é adicionado ao conjunto S de candidatos escolhidos.
- Lembrando que S pode ser, ou não, uma solução ótima.

Compressão de dados

- A compressão de dados consiste em representar o texto (i.e. sequência de símbolos) original usando menos espaço.
- Para isso, basta substituir os símbolos do texto por outros que possam ser representados usando um número menor de *bits*.
- O arquivo comprimido ocupa menos espaço (armazenamento); leva menos tempo para ser lido do disco, ou transmitido por um canal de comunicação; e para ser pesquisado.

Compressão de dados



- Exemplo: Tem-se um arquivo com 100.000 caracteres e deseja-se armazená-lo compactado.

Tabela: 1

	A	**B**	**C**	**D**	**E**	**F**
Frequência (x1000)	45	13	12	16	9	5
Código de comprimento fixo	000	001	010	011	100	101
Código de comprimento variável	0	101	100	111	1101	1100

Compressão de dados

- A Tabela 1 traz um arquivo de dados com 100.000 caracteres que contém no seu alfabeto os caracteres [A..F].
- Se cada caractere for representado com um código de 3 bits, será necessário 300.000 *bits* para codificar o arquivo.
- Em geral, os caracteres do alfabeto não são equiprováveis, ou seja, possuem diferentes probabilidades de ocorrência.
- Logo, é razoável pensar em códigos de comprimento variável, atribuindo códigos mais curtos a caracteres mais frequentes.
- Por exemplo, usando o código de comprimento variável da Tabela 1, é possível codificar o arquivo com 224.000 *bits*.

Compressão de dados

Caractere	Prob.(%)	Cód.I	Cód.II	Cód.III	Cód.IV
A	50	00	0	0	0
B	25	01	1	10	01
C	15	10	00	110	011
D	10	11	11	111	0111

- O Código II parece ser o mais eficiente. Porém, ao contrário dos demais, ele não é univocamente descodificável.
- No Código IV, o bit 0 funciona como separador. Assim, cada palavra de código é descodificada com atraso de 1 bit.
- Já os Códigos I e III são **códigos de prefixo**, ou seja, são univocamente descodificáveis e possuem descodificação instantânea.

Caractere	Prob.(%)	Cód.I	Cód.II	Cód.III	Cód.IV
A	50	00	0	0	0
B	25	01	1	10	01
C	15	10	00	110	011
D	10	11	11	111	0111
$\overline{L}$	-	2	1,25	1,75	1,875

- O Código III apresentou um comprimento médio $\overline{L}$ menor que o do Código I, o que o torna mais eficiente.

  $$\overline{L} = \sum_{c=1}^{n} l_c \; p_c \; bits/\text{símbolo, onde}$$

  l_c é o tamanho da palavra binária que representa o caractere c e p_c é a probabilidade de ocorrência do caractere c.

<u>Exemplo</u> – Considere uma fonte discreta que emite 4 possíveis símbolos a uma taxa de 200 símbolos/s

SÍMBOLO DA FONTE	PROBABILIDADE DE OCORRÊNCIA	REPRESENTAÇÃO BINÁRIA USUAL	REPRESENTAÇÃO BINÁRIA ALTERNATIVA
A	50%	00	0
B	25%	01	10
C	15%	10	110
D	10%	11	111

$\bar{L} = 2$ bits/símbo lo

$R_b = 400$ bits/s

$\bar{L} = 1{,}75$ bits/símbolo

$R_b = 350$ bits/s

CÓDIGO MAIS EFICIENTE

Código de Huffman

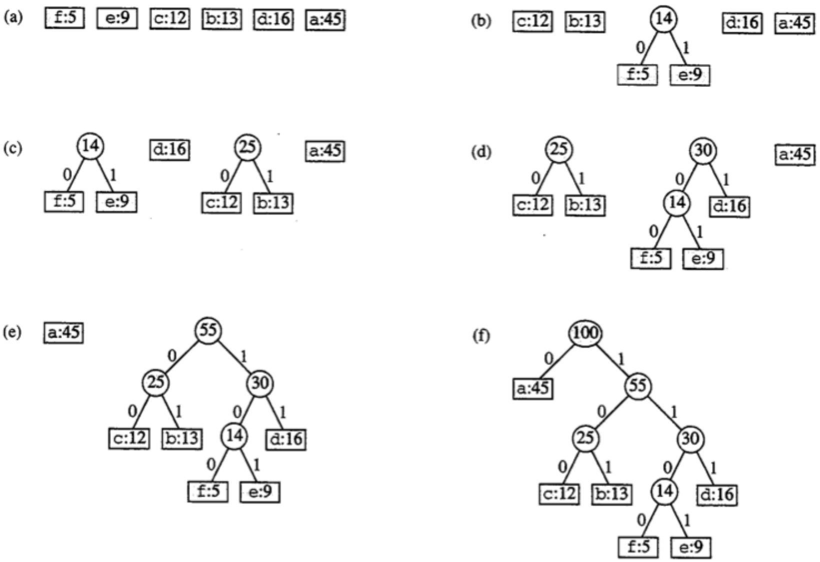
- **Código de Huffman**: Algoritmo eficiente e amplamente utilizado para compressão de dados.
- Sua ideia principal consiste em atribuir códigos mais curtos a símbolos com frequências mais altas.
- Associa códigos binários únicos e de tamanho variável a cada símbolo diferente do alfabeto.
- É gerada uma **árvore binária de prefixo ótima** ao final, ou seja, de comprimento médio mínimo: Árvore de Huffman.

Código de Huffman

O algoritmo de Huffman é **guloso** e consiste dos seguinte passos em um alfabeto de c caracteres:

1. Manter uma floresta de árvores com peso igual à soma das frequências de suas folhas.
2. Selecionar as árvores T_i e T_k de menores pesos e formar uma nova árvore com T_i e T_k.
3. Repetir o passo anterior, até obter a árvore binária final.

Código de Huffman



Código de Huffman

- Calculando quantos são os bits necessários para armazenar o arquivo com o código prefixo ótimo obtido pelo algoritmo de Huffman:

  $(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1.000$
  $= 224.000$ *bits*,

  o que representa uma economia de aproximadamente 25%, quando comparado com a codificação de comprimento fixo apresentada na Tabela 1.

Código de Huffman

- No algoritmo abaixo, assume-se que C é um conjunto de n caracteres e que cada caractere $c \in C$ é um objeto com um atributo $c.freq$ que dá a sua frequência.

```
HUFFMAN (C)
1. n = |C|
2. Q = C
3. para i = 1 até n - 1
4.     Aloca um novo nó z
5.     z.esquerda = x = Extract-min(Q)
6.     z.direita  = y = Extract-min(Q)
7.     z.freq = x.freq + y.freq
8.     Insert(Q,z)
9. retorna Extract-min(Q)  // raiz da árvore
```

- A complexidade no tempo é $O(n \log n)$ considerando a fila de prioridade Q implementada com *heap* binário.

- A eficiência da codificação de Huffman pode ser melhorada se considerarmos extensões de fonte de ordem superior.

- Por exemplo, considere, a princípio, a codificação abaixo.

Caractere	**Código**	**Probabilidade**	**Tamanho**
M1	0	0,70	1
M2	10	0,15	2
M3	11	0,15	2

- $\overline{L} = 1,3$ *bits*/símbolo. Considerando a representação usual com 2 *bits*, economia de 35%.

- Agora, a extensão de 2ª ordem do alfabeto original:

Caractere	Código	Probabilidade	Tamanho
M1M1	1	0,49	1
M1M2	010	0,105	3
M2M1	001	0,105	3
M1M3	000	0,105	3
M3M1	0111	0,105	4
M2M2	011011	0,0225	6
M2M3	011010	0,0225	6
M3M2	011001	0,0225	6
M3M3	011000	0,0225	6

- $\overline{L} = 2,395$ *bits*/símbolo. Considerando a representação usual com 4 *bits*, economia de aproximadamente 40%.

Código de Huffman

- O código da fonte original proporcionou uma economia de 35%, já o código do alfabeto estendido em 2ª ordem trouxe uma economia de 40%.
- Uma forma melhor de aliar os algoritmos de compressão às necessidades dos sistemas de recuperação de informação é considerar **palavras** como símbolos, e não caracteres.
- O método de Huffman baseado em palavras permite acesso randômico a palavras dentro do texto comprimido, que é um aspecto crítico em sistemas de recuperação de informação.
- Nesse caso, a tabela de símbolos do codificador é exatamente o vocabulário do texto. Isso permite uma integração natural entre o método de compressão e o arquivo invertido.

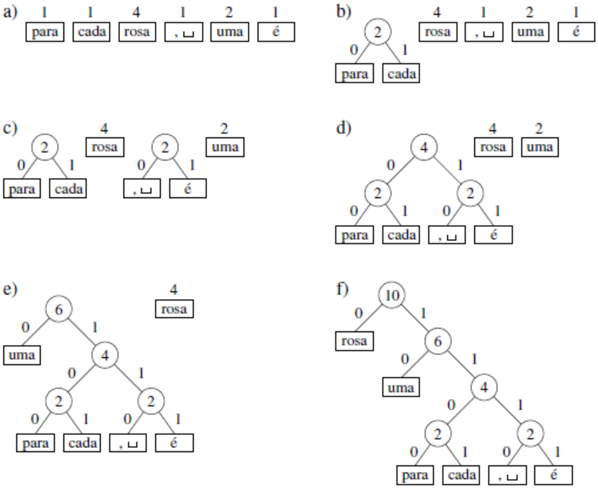
- Codificação de Huffman baseada em palavra:

  O método considera cada palavra diferente do texto em linguagem natural como um símbolo, conta suas frequências e gera um código de Huffman para as palavras. A seguir, comprime o texto substituindo cada palavra pelo seu código.

- Caso uma palavra seja seguida de um espaço, então, somente a palavra é codificada. Caso contrário, a palavra e o separador são codificados separadamente.

- Na decodificação, supõe-se que um espaço simples segue cada palavra, a não ser que o próximo símbolo seja um separador.

Exemplo: "`para cada rosa rosa, uma rosa é uma rosa`"



Considerações

- O método de Huffman codifica um texto de forma a obter-se uma compactação ótima usando códigos de prefixo.

- De posse da árvore de prefixo correspondente, o processo de codificação ou decodificação pode ser realizado em tempo linear no tamanho da sequência binária codificada.

- Possibilidade de realizar casamento de cadeia diretamente no texto comprimido, que é uma pesquisa bem mais rápida dado que menos *bytes* têm que ser lidos.

- Permite acesso direto a qualquer parte do texto comprimido, iniciando a descompressão a partir da parte acessada. Isto é, não precisa descomprimir o texto desde o início.

Considerações

- A maior dificuldade em usar Huffman é a necessidade de se conhecer *a priori* ou estimar as probabilidades dos símbolos que compõem a sequência que se pretende codificar.

- Outra desvantagem é que tanto o *encoder* quando o *decoder* precisam conhecer a árvore de codificação.

Exercício 1

- Uma fonte de informação emite 1.000 símbolos/segundo. As probabilidades de ocorrência de cada símbolo do alfabeto são mostradas na tabela abaixo.

Símbolo da fonte	**Probabilidade de ocorrência**
A	0,4
B	0,2
C	0,2
D	0,1
E	0,1

a. Construa uma tabela de codificação usando Huffman.

b. Calcule o comprimento médio do código gerado.

c. Calcule a taxa média de transmissão em *bits*/segundo após a codificação da fonte.

- Suponha que a tabela a seguir apresenta a frequência de cada letra de um alfabeto em uma *string*.

  Quantos *bits* seriam necessários para representar essa *string* usando um código de Huffman?

Letra	A	B	C	D	E	F
Frequência	20	10	8	5	4	2

(A) 392.
(B) 147.
(C) 113.
(D) 108.
(E) 49.

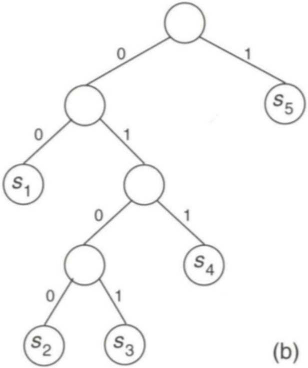
- Suponha o texto S:

$$S = [s_4\ s_3\ s_3\ s_1\ s_3\ s_1\ s_4\ s_5\ s_1\ s_3\ s_3\ s_3\ s_3\ s_3\ s_2\ s_3\ s_5\ s_2\ s_2\ s_2\ s_4]$$

e os códigos dados pela árvore de prefixo T abaixo.

símbolo	*código*
s_1	00
s_2	0100
s_3	0101
s_4	011
s_5	1

(a)



(b)

Exercício 3

a. Quantos dígitos binários são necessários para codificar o texto S usando a árvore de prefixo T?

b. O comprimento da sequência binária produzida no item (a) é mínimo considerando códigos de prefixo? Ou seja, é possível codificar o texto S com um número menor de dígitos usando uma árvore binária de prefixo? Explique.

Considerações

- Para resolver um problema de otimização, o algoritmo guloso escolhe, em cada iteração, o objeto mais "promissor" para fazer parte da solução.
- Um algoritmo guloso toma decisões com base nas informações disponíveis na iteração corrente, sem olhar as consequências que essas decisões (viáveis) terão no futuro.
- Um algoritmo guloso jamais se arrepende ou volta atrás, ou seja, as escolhas que faz em cada iteração são definitivas.
- Os algoritmos gulosos são muito rápidos e eficientes. Contudo, os problemas que admitem soluções gulosas são raros.

Programação dinâmica

- Como visto anteriormente, os problemas de otimização nos pedem que achemos a melhor solução possível, respeitando algumas restrições.
- O *branch-and-bound* busca por todas as soluções possíveis e seleciona a melhor.
  - Sempre retorna a resposta correta, mas é inviável para problemas com instâncias grandes.
- Algoritmos gulosos fazem sempre a melhor escolha em cada ponto de decisão.
  - Sem provas de corretude simples, normalmente falham em obter a melhor solução.
- Já a programação dinâmica é uma técnica de programação ascendente que armazena resultados parciais para acelerar algoritmos recursivos.

Programação dinâmica

- A programação dinâmica (PD) é um nome fantasia para "recursão com apoio de uma tabela".
- Em vez de resolver subproblemas de forma recursiva, vamos resolvê-los sequencialmente e armazenar suas soluções em uma tabela.
- O truque é resolvê-los de maneira que quando a solução de um subproblema seja novamente requerida, esta já esteja disponível na tabela.

- O que um problema de otimização deve ter para que a PD seja aplicável são duas características principais:
  - Subestrutura ótima (ou otimalidade); e
  - Superposição de subproblemas.
- Um problema apresenta uma **subestrutura ótima** quando sua solução ótima contém nela própria soluções ótimas para subproblemas do mesmo tipo.
- A **superposição de subproblemas** surge quando o algoritmo recursivo reexamina o mesmo subproblema muitas vezes.
- Nesses casos, faz sentido calcular cada solução pela primeira vez e armazená-la em uma tabela para uso posterior, ao invés de recalculá-la recursivamente sempre que for necessário.

Sequência de Fibonacci

- A sequência de Fibonacci F é dada por:

$$
\begin{aligned}
&F_i = F_{i-1} + F_{i-2} \\
&F_0 = 0 \\
&F_1 = 1
\end{aligned}
$$

- A sequência de Fibonacci foi inicialmente proposta e estudada por Leonardo Fibonacci no contexto de reprodução animal.

- Esse padrão numérico tem aplicações na computação, análise de mercados financeiros, teoria dos jogos, entre outras.

Sequência de Fibonacci

- Algoritmo recursivo para calcular *fib*(n):

```
fib(n)
1. se (n = 0 ou n = 1)
2.    então retorna (n)
3.    senão retorna (fib(n - 2) + fib(n - 1))
```

- A relação de recorrência assume a fórmula:

$T(0) = 0$
$T(1) = 1$
$T(n) = T(n-2) + T(n-1)$, para $n \geq 2$

- Limite inferior para $T(n)$:

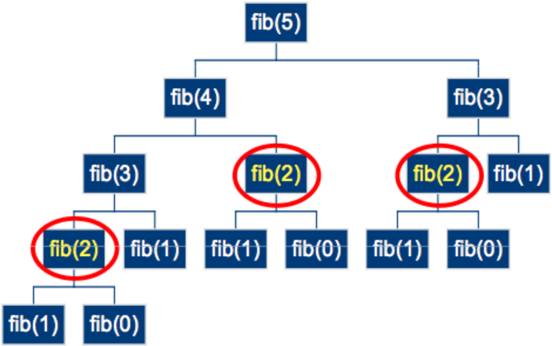
$$T(n) \geq T(n-2) + T(n-2) = 2T(n-2)$$
$$T(n) \geq 4T(n-4)$$
$$T(n) \geq 8T(n-6)$$
$$\ldots$$
$$T(n) \geq 2^k T(n-2k)$$

- A expansão para quando $n - 2k = 1$, ou seja, $k = \frac{n-1}{2}$.

  Logo, $T(n) \geq 2^{\frac{n-1}{2}}$

- Conclui-se que esse algoritmo recursivo é ineficiente, levando um tempo exponencial para calcular $fib(n)$.

Sequência de Fibonacci



- Por exemplo, para calcular $fib(5)$, chamamos 3 (três) vezes o subproblema do mesmo tipo $fib(2)$.
- Contudo, é possível desenvolver um algoritmo eficiente (em tempo linear) armazenando os valores das instâncias menores e não os recalculando.

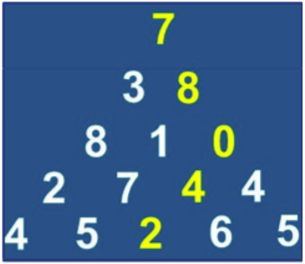
Sequência de Fibonacci

- Programação dinâmica: Armazena os valores das entradas menores e não os recalcula.

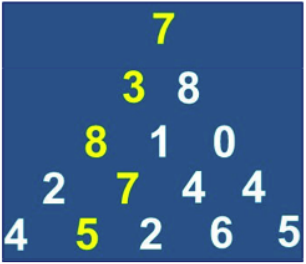
```
Fibonacci(n)
se (n = 0 ou n = 1)
então retorna (n)
senão
  F1 = 0
  F2 = 1
  para i = 1 até n - 1 faça
     F = F1 + F2
     F1 = F2
     F2 = F
  retorna (F)
```

- O algoritmo mantem sempre em memória os dois últimos números da sequência e é executado em $O(n)$.

- Pirâmide de números é um problema clássico das Olimpíadas Internacionais de Informática de 1994.
- Objetivo: Calcular a rota, que começa no topo da pirâmide e acaba na base, com **maior soma**. Em cada passo, podemos ir diagonalmente para baixo e para a esquerda ou para baixo e para a direita.



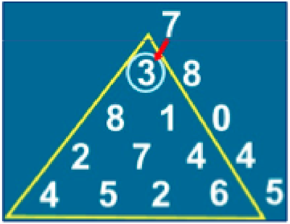
Soma = 21

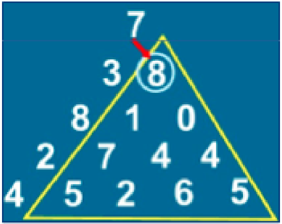


Soma = 30

Pirâmide de números

- Quando estamos no topo da pirâmide, temos duas decisões possíveis (esquerda ou direita):





- Mas o que nos interessa saber sobre essas subpirâmides?
- Apenas interessa o valor da sua "melhor" rota interna, que é uma instância menor do mesmo problema.
- Para o exemplo, a solução é 7 mais o máximo entre o valor da melhor rota de cada uma das subpirâmides.

Pirâmide de números

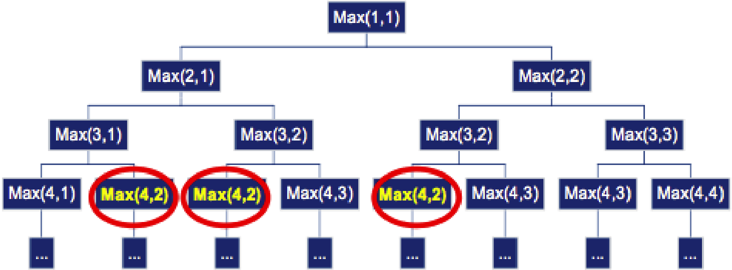
- Então, o problema pode ser resolvido recursivamente.

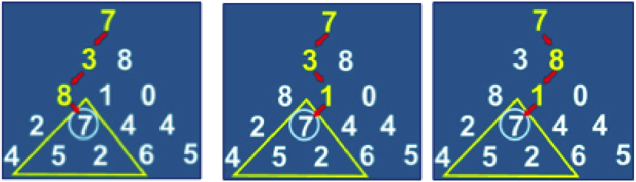
```
Max(i,j)
% P[i,j] : j-ésimo número da i-ésima linha
% n : altura da pirâmide
1.  se (i = n)
2.      então soma = P[i,j]
3.  senão
4.      esq = Max(i+1,j)
5.      dir = Max(i+1,j+1)
6.      se (esq > dir)
7.          então soma = P[i,j] + esq
8.          senão soma = P[i,j] + dir
9.  retorna (soma)
```

- Para resolver o problema basta executar $Max(1,1)$.

Pirâmide de números

- O algoritmo recursivo tem crescimento exponencial: $O(2^n)$. Note que o mesmo subproblema é calculado várias vezes.

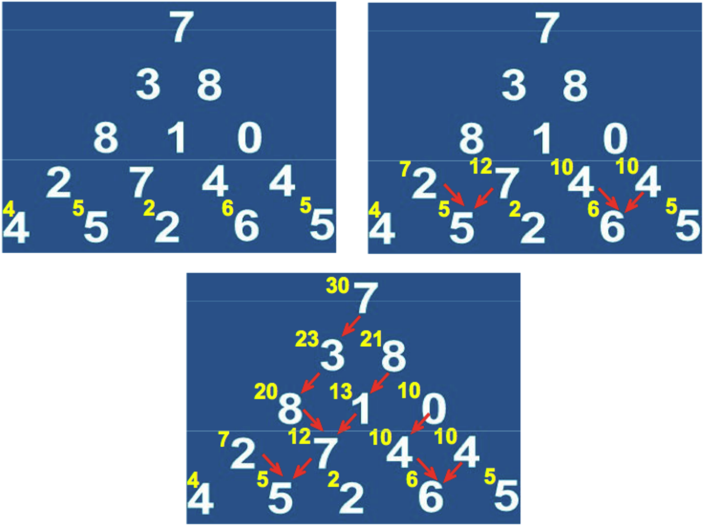




Pirâmide de números

- O problema das pirâmides apresenta as características para ser resolvido com PD: **subestrutura ótima** e **superposição de subproblemas**.
- A solução ótima contém nela própria os melhores percursos de subpirâmides, ou seja, soluções ótimas de subproblemas.
- Para uma dada instância do problema, muitos subproblemas que aparecem são coincidentes.
- A ideia é guardar o valor calculado para cada subproblema em uma tabela para, quando for o caso, reaproveitá-lo.

- Começar a partir do fim!



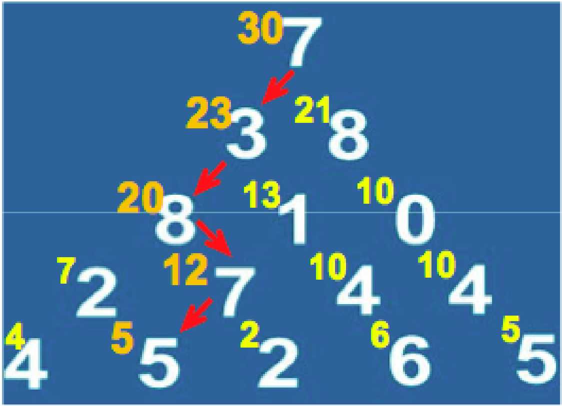
Pirâmide de números

- Tendo em conta a maneira como preenchemos a tabela, é possível aproveitar a matriz P.

```
Calcular(n)
% n : altura da pirâmide
1. Para i = n - 1 até 1
2.    Para j = 1 até i
3.       esq = P[i+1,j]
4.       dir = P[i+1,j+1]
5.       se (esq > dir)
6.           então P[i,j] = P[i,j] + esq
7.           senão P[i,j] = P[i,j] + dir
```

- Assim, a solução fica em $P[1,1]$, sendo $n > 1$.
- E o tempo necessário para resolver o problema cresce de forma quadrática: $O(n^2)$.

- Para saber a constituição da melhor solução, basta usar a tabela já calculada.



Considerações

- Quando o algoritmo recursivo tem complexidade exponencial, a técnica de PD pode levar a um algoritmo mais eficiente.

- Para resolver os problemas da pirâmide de números e da sequência de Fibonacci usamos PD.

- A PD calcula a solução para todos os subproblemas, partindo dos subproblemas menores para os maiores, armazenando os resultados em uma tabela.

- A vantagem é que uma vez que um subproblema é resolvido, sua resposta é guardada em uma tabela e ele nunca mais é recalculado.

- Problemas que podem ser resolvidos em tempo polinomial são considerados tratáveis, enquanto problemas que só podem ser resolvidos por algoritmos exponenciais são intratáveis.
- Problemas considerados intratáveis são comuns na natureza e nas diversas áreas do conhecimento.
- Um exemplo é o Problema do Caixeiro Viajante (PCV), cuja complexidade no tempo dos melhores algoritmos conhecidos para resolvê-lo é $O(n!)$.

Algoritmos aproximados

- Diante de um problema considerado intratável é comum não exigir que o algoritmo sempre obtenha a solução ótima.
- Normalmente, procuramos por algoritmos **eficientes** que não garantem a solução ótima, mas tentam encontrar uma solução que seja a mais próxima possível do ótimo.
- Para isso, existem dois tipos de algoritmos: heurísticas e algoritmos aproximados.

- Uma **heurística** é um algoritmo que pode produzir um bom resultado, até mesmo a solução ótima, mas pode também não produzir solução alguma ou uma solução distante do ótimo.
- Na heurística não existe comprometimento com a qualidade dos resultados produzidos.
- Um **algoritmo aproximado** gera soluções aproximadas dentro de um limite para a razão entre a solução ótima e a produzida pelo algoritmo aproximado.
- O comportamento dos algoritmos aproximados é monitorado do ponto de vista da qualidade dos resultados.

Algoritmos aproximados

- Seja I uma instância de um problema P e seja $S^*(I)$ o valor da solução ótima para I.
- Um algoritmo aproximado A gera uma solução possível para I cujo valor $S(I)$ é pior do que o valor ótimo $S^*(I)$.
- Dependendo do problema, o objetivo será por minimizar ou maximizar $S(I)$.
- Por exemplo, em um problema de minimização (p.e. PCV), $0 < S^*(I) \leq S(I)$ e a ideia do algoritmo aproximado é obter uma $S(I)$ o mais próximo possível de $S^*(I)$.

- O comportamento do algoritmo aproximado A é descrito pela razão de aproximação:

$$R_A(I) = \frac{S(I)}{S^*(I)},$$

que representa um problema de minimização (no caso de um problema de maximização, a razão é invertida). Em ambos os casos, $R(I) \geq 1$. Com $R(I) = 1$, tem-se uma solução ótima.

Considerações

- Para muitos problemas, foram desenvolvidos algoritmos de aproximação de tempo polinomial com pequenas razões de aproximação constantes.
- Enquanto, para outros problemas, os melhores algoritmos de aproximação de tempo polinomial conhecidos têm razões de aproximação que crescem em função da entrada n.
- Em alguns casos, é possível alcançar razões de aproximação cada vez menores, usando mais tempo de computação.
- Isto é, existe um compromisso entre tempo de computação e qualidade de aproximação.