# Algoritmos Gulosos

Escolha gulosa
Seleção de atividades
Código de Huffman

# Algoritmos Gulosos

- Problemas de otimização
  - Requer algoritmos que retornam a melhor solução
  - Algoritmos de busca exaustiva encontram o resultado ótimo, mas muitas vezes é impraticável
  - Programação dinâmica permite o projeto de algoritmos customizados que buscam todas possibilidades enquanto armazenam resultados para evitar cálculos repetitivos
  - Algoritmos gulosos: nem sempre garantem a melhor a solução
    - Para alguns problemas, algoritmos gulosos podem garantir a solução ótima

# Algoritmos Gulosos

- Assim como em programação dinâmica, para que algoritmos gulosos funcionem de forma ótima, é preciso que o problema possua
  - Subestrutura ótima: as soluções ótimas do problema incluem soluções ótimas de subcasos
  - Propriedade gulosa: garante que a cada subproblema uma escolha gulosa leva a uma solução ótima deste subproblema
- Critério guloso
  - O algoritmo faz uma escolha que parece ser a melhor
    - Decisão localmente ótima
  - Não revê as decisões tomadas, ou seja, a solução recursiva não realiza backtracking

# Seleção de Atividades

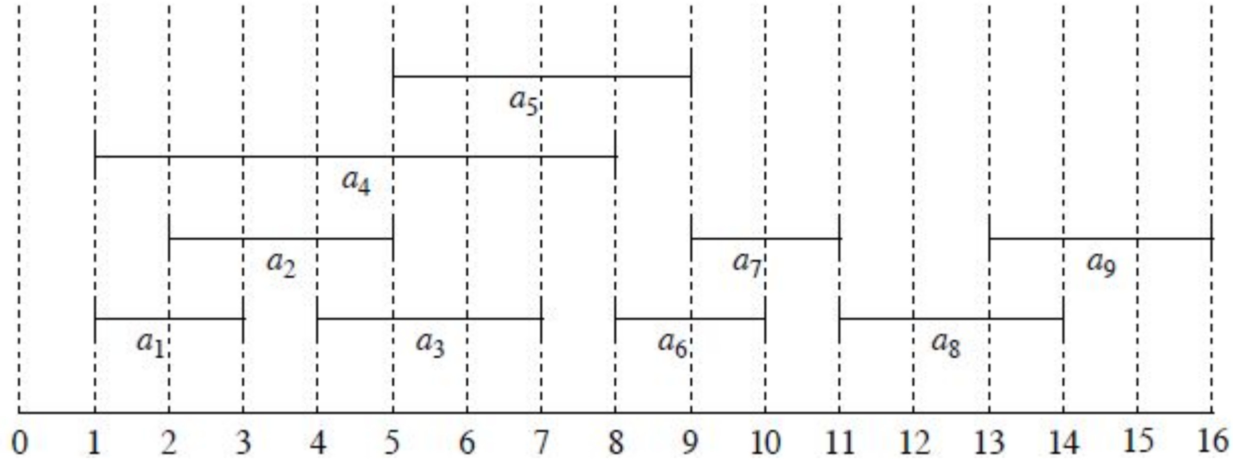
- Sejam um conjunto de atividades $S = \{a_1, a_2, \dots, a_n\}$
- Cada atividade $a_i$ tem início no instante $s_i$ e término em $f_i$
  - Para cada $a_i$: $0 \leq s_i < f_i < \infty$ .
  - Intervalo de um $a_i$: $[s_i, f_i)$
- Uma atividade $a_i$ é compatível com outra atividade $a_j$ se não há sobreposição entre $[s_i, f_i)$ e $[s_j, f_j)$
  - Se $f_i \leq s_j$ ou $f_j \leq s_i$, então $a_i$ e $a_j$ não se sobrepõem

# Seleção de Atividades

- Problema: Determinar um subconjunto de atividades que são mutuamente compatíveis de tamanho máximo
  - Assumimos que as atividades são ordenadas em ordem crescente de tempo de término
    - $f_1 \leq f_2 \leq f_3 \leq \ \dots \leq f_{n-1} \leq f_n$

# Exemplo

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 1 | 5 | 8 | 9 | 11 | 13 |
| | 3 | 5 | 7 | 8 | 9 | 10 | 11 | 14 | 16 |

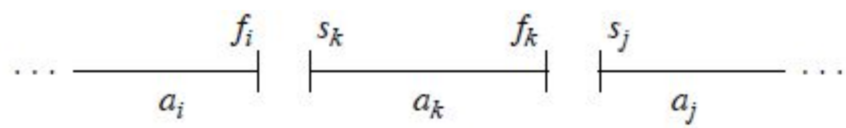



- $\{a_1, a_5, a_9\}$ é um subconjunto de atividades mutuamente compatíveis
- $\{a_1, a_3, a_6, a_8\}$ é o maior, mas não é o único
- $\{a_1, a_3, a_7, a_9\}$ é outra possível solução

# Subestrutura Ótima

- Seja $\boldsymbol{S_{ij}}$ o conjunto de atividades que começam após o término de $a_i$ e terminam antes do início de $a_j$

$$S_{ij} = \{a_k \in S: f_i \le s_k < f_k \le s_j\}$$




- Conjunto está **ordenado em ordem crescente de tempos de término**
  - Logo, $i < j$

# Subestrutura Ótima

$$S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$$

Supondo também que

- $A_{ij}$: conjunto máximo de atividades compatíveis em $S_{ij}$
- $a_k \in A_{ij}$
  - Temos 2 subproblemas (disjuntos): encontrar conjuntos máximos de atividades compatíveis em
  - $S_{ik}$ (atividades que começam após $a_i$ terminar e que terminam antes de $a_k$ começar)
  - $S_{kj}$ (atividades que começam após $a_k$ terminar e que terminam antes de $a_j$ começar)
- Sejam:
  - $A_{ik} = A_{ij} \cap S_{ik}$
  - $A_{kj} = A_{ij} \cap S_{kj}$
- Então,

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

$$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$$

# Subestrutura Ótima

- Mostrar que $A_{ij}$ (solução ótima de $S_{ij}$) deve incluir soluções ótimas dos subproblemas $S_{ik}$ e $S_{kj}$
- Argumento "recortar e colar":
  - Supor que houvesse uma solução $A'_{kj}$ de $S_{kj}$ tal que $|A'_{kj}| > |A_{kj}|$
  - Então, seria possível construir um conjunto de tamanho
    $|A'_{ij}| = |A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$
  - Contradição com a suposição de que $A_{ij}$ é uma solução ótima de $S_{ij}$
  - Aplica-se argumento simétrico para $S_{ik}$

# Solução recursiva

A forma da subestrutura ótima demonstrada sugere uma solução por programação dinâmica da seguinte maneira:

- $c[i,j]$: tamanho da solução ótima $\left|A_{ij}\right|$
- Se $i \geq j \Rightarrow S_{ij} = \emptyset \Rightarrow c[i,j] = 0$
- Se $S_{ij} \neq \emptyset$, então existe algum $a_k \in S_{ij}$
  - Como $a_i \neq a_k \neq a_j$, então $i < k < j$
- $c[i,j] = \begin{cases} 0, & \text{se } S_{ij} = \emptyset \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i,k] + c[k,j] + 1\}, & \text{se } S_{ij} \neq \emptyset \end{cases}$

# Escolha Gulosa

- Para cada $c[i, j]$ existem diversos subproblemas para $i < k < j$
- Se fosse possível escolher um deles e adicionar à solução ótima sem resolver todos os subproblemas, o custo computacional seria reduzido
- Qual critério utilizar para a escolha gulosa? Essa escolha garantiria a solução ótima global?
- Intuitivamente, é melhor escolher uma atividade que deixe o tempo restante para o número máximo de atividades seguintes
  - Escolher atividade com o menor tempo de término ($f_k$)

# Escolha Gulosa

- Escolher atividade com o menor tempo de término ($f_k$)
  - Subproblema: atividades que começam a partir do término da atividade escolhida ($f_k$)
  - Cada subproblema é resolvido de forma gulosa
- Para a escolha de uma atividade com término em $f_k$, temos o subproblema $S_k$:

$$S_k = \{a_i \in S : f_k \leq s_i\}$$

# Propriedade Gulosa

Teorema: considerando um subproblema não-vazio $S_k$, seja $a_m$ uma atividade com o término mais cedo em $S_k$. Então, $a_m$ faz parte de algum subconjunto de tamanho máximo de atividades mutuamente compatíveis de $S_k$.

# Propriedade Gulosa

Teorema: considerando um subproblema não-vazio $S_k$, seja $a_m$ uma atividade com o término mais cedo em $S_k$. Então, $a_m$ faz parte de algum subconjunto de tamanho máximo de atividades mutuamente compatíveis de $S_k$.

- Prova:
  - Sejam:
    - $A_k$: subconjunto de atividades mutuamente compatíveis em $S_k$ de tamanho máximo
    - $a_j$: atividade em $A_k$ com menor tempo de término

# Propriedade Gulosa

Teorema: considerando um subproblema não-vazio $S_k$, seja $a_m$ uma atividade com o término mais cedo em $S_k$. Então, $a_m$ faz parte de algum subconjunto de tamanho máximo de atividades mutuamente compatíveis de $S_k$.

- Prova:
  - Sejam:
    - $A_k$: subconjunto de atividades mutuamente compatíveis em $S_k$ de tamanho máximo
    - $a_j$: atividade em $A_k$ com menor tempo de término
  - Se $a_j = a_m$
    - Então, $a_m$ faz parte de uma solução ótima
  - Se $a_j \neq a_m$
    - Então, seja
      - $A'_k = A_k - \{a_j\} \cup \{a_m\}$
      - Atividades em $A'_k$ são disjuntas, pois
        - Atividades em $A_k$ são disjuntas
        - $a_j$ é a primeira atividade em $A_k$ a terminar
        - $f_m \leq f_j$ ($a_m$ é a primeira atividade em $S_k$ a terminar)
      - Portanto, $A'_k$ é um subconjunto (máximo) de atividades mutuamente compatíveis de $S_k$ e inclui $a_m$.

# Solução Gulosa

- O problema possui a propriedade gulosa: o critério guloso basta para obter a solução ótima

| | Antes do teorema | Depois do teorema |
|---|---|---|
| # subproblemas na solução ótima | 2 | 1 |
| # de escolhas para considerar | *j*-*i*-1 | 1 |

- Solução top-down
  - Escolhe $a_m$ uma atividade com o término mais cedo em $S_k$, então resolve $S_m$

# Solução Gulosa

- Solução top-down
  - Considerar a adição de atividades fictícias $a_0$ e $a_{n+1}$
    - $a_0 = [-\infty, 0)$
    - $a_{n+1} = [\infty - 1, \infty)$
  - Chamada inicial `Recursive-Activity-Selector(s, f, 0, n)`

```
Recursive-Activity-Selector(s, f, i, n){
  //entrada: arranjos s[0..n+1] e f[0..n+1] com os tempos de início e
  //término das atividades
  //retorno: subconjunto de atividades mutualmente compatíveis
  //de tamanho máximo
  m = i + 1
  while (m ≤ n and s[m] < f[i]) do
    m = m + 1 //encontra a primeira atividade em Sk a terminar
  if(m ≤ n)
    return {a_m} ∪ Recursive-Activity-Selector(s,f,m,n)
  else return ∅
}
```

# Solução Gulosa

- Solução iterativa
  - Manter uma variável $k$ que indexa a última atividade adicionada a $A$
    - $f_k = \max\{f_i : a_i \in A\}$, ou seja, $a_k$ possui o maior tempo de término entre as atividades selecionadas em $A$ até o momento
  - Procura o próximo $a_m$ que pode ser inserido em $A$
    - $f_k \leq s_m$

```
Greedy-Activity-Selector(s, f){
  //entrada: arranjos s[1..n] e f[1..n] com os tempos de início e
  //término das atividades
  //retorno: subconjunto de atividades mutualmente compatíveis
  //de tamanho máximo
  n = s.length
  A = {a₁}
  k = 1
  for m = 2 to n
    if(f[k] ≤ s[m])
      A = A ∪ {aₘ}
      k = m
  return A
}
```

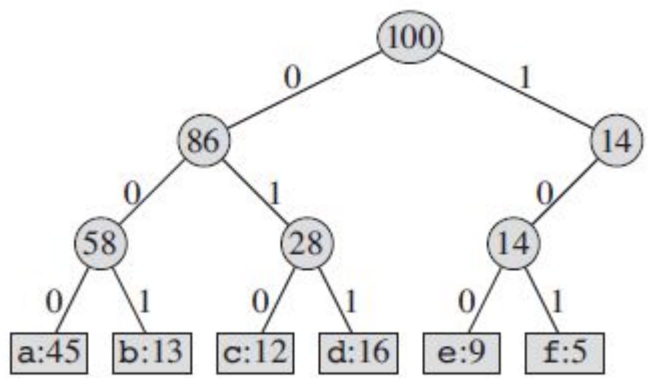
# Código de Huffman

- Método de compressão de dados sem perda
- Algoritmo desenvolvido por David Huffman durante o seu doutorado no MIT e publicado em 1952.
- Ex: arquivo texto contendo 100.000 caracteres com o alfabeto $\Sigma = \{a, b, c, d, e, f\}$ com as frequências na tabela abaixo
  - Se cada caractere ocupar 1 byte, então o arquivo teria tamanho de 800.000 bits

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| | 45 | 13 | 12 | 16 | 9 | 5 |
| Código de tam. Fixo | 000 | 001 | 010 | 011 | 100 | 101 |

# Exemplo

- Utilizando a codificação a=000, b=001, ..., f=101, cada caractere poderia ser representado por 3 bits.
- Logo, 300.000 bits para codificar todo o arquivo.



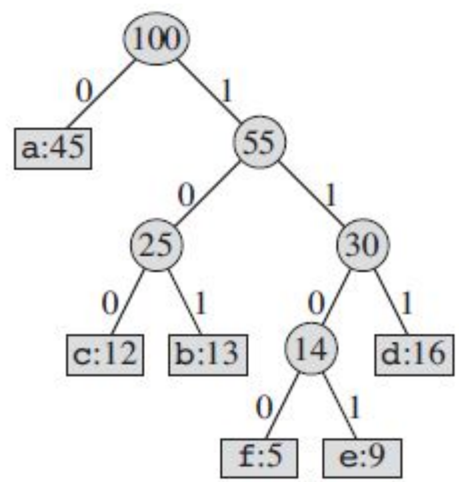
# Exemplo

- Se utilizarmos um código de tamanho variável, podemos atribuir codificações mais curtas para caracteres mais frequentes e as mais longas para os menos frequentes.
- Utilizando o código de tamanho variável da tabela abaixo
  (45x1 + 13x3 + 12x3 + 16x3 + 9x4 + 5x4) x 1000 = 224.000 bits
- Ex: aacf = 001001100 = 0.0.100.1100
- Problema
  - Como separar os caracteres?

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| | 45 | 13 | 12 | 16 | 9 | 5 |
| Código de tam. Fixo | 000 | 001 | 010 | 011 | 100 | 101 |
| Código de tam var. | 0 | 101 | 100 | 111 | 1101 | 1100 |

# Código de Huffman

- O código de Huffman é um código livre de prefixo
  - O código de cada caractere não é prefixo do código de nenhum outro caractere
  - Ex: 001001100 = 0.0.100.1100 = aacf



# Código de Huffman

- Uma codificação ótima é representada por uma árvore cheia
  - Cada vértice interno tem sempre 2 filhos
- Tamanho do arquivo
  - Caracteres $c$ de um alfabeto $C$ são representados por folhas na árvore. A frequência de $c$ no arquivo é denotada por $c.freq$ e a profundidade de c na árvore é denotada por $d_T(c)$.
  - O tamanho $B(T)$ define o custo da árvore

$$B(T) = \sum_{c \in C} c.freq * d_T(c)$$

# Algoritmo de Huffman

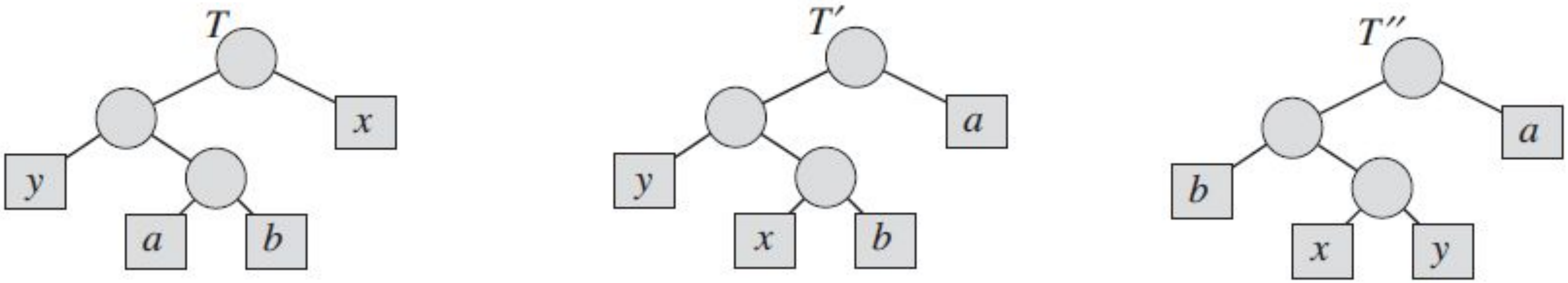
- Ideia do algoritmo
  - Começar com $|C|$ folhas
  - Realizar $|C| - 1$ intercalações entre 2 vértices da árvore
  - Intercalações são realizadas na ordem crescente de frequências. Um nó interno possui como frequência as somas das frequências das folhas de sua subárvore

# Algoritmo de Huffman

```
Huffman(C){
  //entrada: Conjunto de caracteres com as frequências
  //retorno: raiz da árvore
  //Q é uma fila de prioridade (min-heap)
  n = |C|
  Q = C
  for i = 1 to n-1
    z = new node
    z.left  = x = Extract_Min(Q)
    z.right = y = Extract_Min(Q)
    z.freq  = x.freq + y.freq
    Insert(Q,z)
  return Extract_Min(Q)
}
```

# Corretude

- Lema 1 (escolha gulosa): Seja $C$ um alfabeto onde cada caractere $c \in C$ tem frequência $f[c]$. Sejam $x$ e $y$ dois caracteres em $C$ com as menores frequências. Então, existe um código ótimo livre de prefixo para $C$ no qual os códigos para $x$ e $y$ têm os mesmos comprimentos e diferem apenas no último bit.
- Prova:
  - Seja uma árvore ótima $T$
  - Sejam $a$ e $b$ duas folhas irmãs **mais profundas** de $T$ e $x$ e $y$ as folhas de $T$ de **menor frequência**.
  - Ideia: obter uma outra árvore $T$'', a partir de $T$, em que $x$ e $y$ são irmãs.




$$B(T) - B(T') = \sum_{c \in C} f[c] * d_T(c) - \sum_{c \in C} f[c] * d_{T'}(c)$$
$$= f[x] * d_T(x) + f[a] * d_T(a) - f[x] * d_{T'}(x) - f[a] * d_{T'}(a)$$
$$= f[x] * d_T(x) + f[a] * d_T(a) - f[x] * d_T(a) - f[a] * d_T(x)$$
$$= (f[a] - f[x])(d_T(a) - d_T(x)) \geq 0$$

- Portanto, $T$’ não tem custo superior a $T$.
- Analogamente, temos que

$$B(T) \geq B(T') \geq B(T'')$$

# Corretude

- Lema 2 (subestrutura ótima): Seja $C$ um alfabeto com frequência $f[c]$ definida para cada caractere $c \in C$. Sejam $x$ e $y$ dois caracteres de $C$ com as menores frequências. Seja C' o alfabeto obtido pela remoção de $x$ e $y$ e pela inclusão de um novo caractere $z$, tal que $C' = C \cup \{z\} - \{x, y\}$. As frequências dos novos caracteres em $C' \cap$ C são as mesmas que em $C$ e $f[z]$ é definida como sendo $f[z] = f[x] + f[y]$. Seja $T'$ uma árvore binária representando um código ótimo livre de prefixo para $C'$. Então, a árvore binária $T$ obtida de $T'$ substituindo-se o vértice (folha) $z$ por um vértice interno tendo $x$ e $y$ como filhos, representa um código ótimo livre de prefixo para $C$.

Prova

- Comparando os custos de $T$ e $T$'
- Para caractere $c \in C - \{x, y\}$, temos que $d_T(c) = d_{T'}(c)$.
- Logo, $f[c] * d_T(c) = f[c] * d_{T'}(c)$
- Como $d_T(x) = d_T(y) = d_{T'}(z)$ +1, então temos

$$\begin{aligned} f[x] * d_T(x) + f[y] * d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= f[z] * d_{T'}(z) + (f[x] + f[y]) \end{aligned}$$

- Conclui-se que
  - $B(T) = B(T') + f[x] + f[y]$
  - $B(T') = B(T) - f[x] - f[y]$
- Supondo que $T$ não representa um código ótimo livre de prefixo para $C$. Então existiria um $T$'' tal que $B(T'') < B(T)$.
- Seja $T$''' a árvore gerada a partir $T$'' com o pai comum de $x$ e $y$ trocado por uma folha $z$ de frequência $f[z] = f[x] + f[y]$. Então,

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \ = B(T') \end{aligned}$$

- Isso contradiz a suposição de que $T$' representa um código ótimo para $C$'. Então, $T$ deve ser um código ótimo livre de prefixo para $C$.

# Exercícios

1) Considere o Problema da Mochila. Seja uma mochila de capacidade W e um conjunto de objetos S={1,2,...,n}, sendo que cada objeto i tem peso $w_i$ e valor $v_i$. O objetivo do problema Mochila 0-1 é determinar um subconjunto S'⊆S que maximize o valor de $\sum_{i \in S'} v_i$, dado que $\sum_{i \in S'} w_i \leq W$. Uma versão deste problema, o problema da Mochila Fracional, permite que frações de objetivos possam ser inseridos na mochila.

a) Considere a seguinte estratégia de solução do problema da Mochila 0-1: colocar elementos de S que entrem na mochila do maior para o menor. Essa solução garante a solução ótima? Explique.

b) Forneça uma solução por programação dinâmica para o problema da Mochila 0-1 com tempo O(nW).

c) A solução por programação dinâmica é um algoritmo de tempo polinomial?

d) Forneça um critério guloso que resolva o problema da Mochila Fracional. Esse critério garante a solução ótima? Explique.

# Exercícios

2) Considere o problema de retornar $n$ centavos de troco com o número mínimo de moedas.

a) Seja T o valor a ser retornado e o seguinte $D = \{25, 10, 5, 1\}$ o conjunto de diferentes valores de moedas disponíveis. Para esse conjunto de possíveis moedas, é possível projetar um algoritmo guloso que encontre a quantidade mínima de moedas para retornar o troco $T$? Justifique. Caso possível, forneça um algoritmo guloso que encontre a solução ótima.

b) Forneça um outro conjunto $D$ para o qual o algoritmo guloso não encontra a solução ótima. Mostre um caso para o qual este algoritmo falha.

# Exercícios

3) Você precisa dirigir um carro de uma cidade A até uma cidade B. Um tanque cheio do carro possui autonomia para viajar $m$ quilômetros e você sabe as distâncias, a partir de A, dos postos de combustível existentes no caminho. Sejam $d_1 < d_2 < \ldots < d_n$ as distâncias dos $n$ postos do caminho. Você deve encontrar onde você deve parar para abastecer o carro de forma a fazer o menor número de paradas para chegar até B.

a) Forneça um algoritmo guloso para resolver o problema.

b) Mostre que o algoritmo encontra a solução ótima.

# Referências

- CLRS, Introduction to Algorithms, 3rd ed.
  - Cap. 16, 16.1, 16.3