- Compressão de dados
- Consiste em representar o texto (i.e. seq. 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

	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

↑ Frequência
↓ bits

- Código de prefixo: são univocamente descodificáveis e possuem codificação instantânea. Nunca um código vai ser prefixo do outro (não é dubil e não tem atraso)

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.

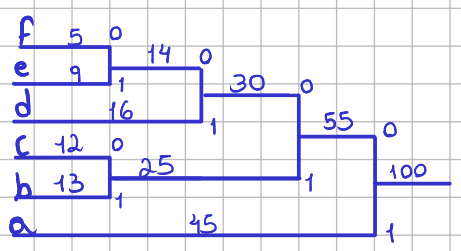
$\overline{L} = \sum_{c=1}^{n} l_c \; p_c$ *bits*/símbolo, onde

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

- Código de Huffman
  - Algoritmo eficiente e amplamente usado para compressão de dados
  - Gera semprre código de prefixo
  - É gerada uma **árvore binária de prefixo ótima** ao final, ou seja, de comprimento médio mínimo: Árvore de Huffman
  - É um algoritmo guloso e consiste em:

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.

- Exemplo:
f:5 | e: 9 | c: 12 | b: 13 | d: 16 | a: 45



a: 1
b: 011
c: 010
d: 001
e: 0001
f: 0000

Comprimento médio:
1.45 + 3.13 + 3.12 + 3.16 + 4.9 + 4.5
45 + 39 + 36 + 48 + 36 + 20
93 + 59 + 72 = 224 bits/símbolo

- 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
```

supondo ter um heap, temos: logn

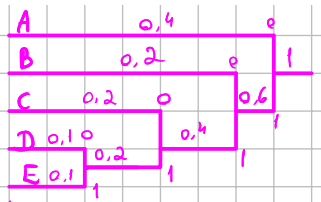
- A complexidade no tempo é $O(nlog(n))$ considerando a fila de prioridades Q implementada com heap binário.

- Com Huffman, podemos fazer pesquisas em um arquivo comprimido diretamente
- O tempo de codificação e decodificação ficam lineares
- Precisa do conhecimento, a priori, das probabilidades dos símbolos
- Tanto o encoder quanto o decoder precisam conhecer a árvore de codificação (ocupa espaço)

- 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.

a)


A = 0
B = 10
C = 110
D = 1110
E = 1111

b) 1.0,4 + 2.0,2 + 3.0,2 + 4.0,1 + 4.0,1
1,6 + 0,6 = 2,2 bits/símbolo

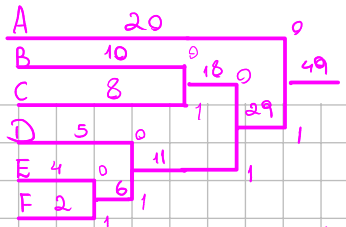
c) 2,2 x 1000 = 2200 bits/s

- 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.



A: 0
B: 100
C: 101
D: 110
E: 1110
F: 1111

1.20 + 3.10 + 3.8 + 3.5 + 4.4 + 4.2
50 + 24 + 15 + 24
98 + 15 = 113

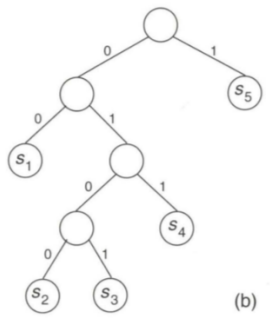
- Suponha o texto S:

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

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)

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

3.2 + 4.4 + 9.4 + 3.3 + 2.1 = 69 bits

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.

Não, usando o alg. de Huffman, observamos que o comprimento pode ser reduzido a:

3.3 + 3.4 + 9.1 + 3.3 + 3.2

27 + 12 + 6

27 + 18 = 45 bits

Símb. Freq.

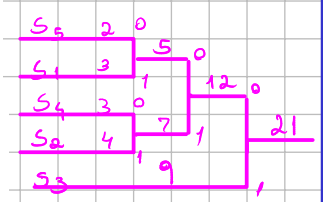
s_4 3

s_3 9

s_1 3

s_5 2

s_2 4



s_1: 001

s_2: 011

s_3: 1

s_4: 010

s_5: 000

Programação dinâmica

- ✔ é uma técnica de programação ascendente que armazena resultados parciais para acelerar algoritmos recursivos
- ✔ recursão com apoio de uma tabela
- ✔ resolve sequencialmente e armazena as soluções
- ✔ O truque é: quando a solução de um subproblema for requerida, ela já vai esté disponível na tabela
- ✔ O problema de otimização tem que ter isso para poder aplicar PD:
  - Subestrutura ótima (ou otimalidade): e → subproblema tem sol. ótima
    → é a estrutura usada para guardar essas soluções ótimas
  - Superposição de subproblemas → O alg. recursivo reexamina o mesmo subproblema várias vezes

Fibonacci

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

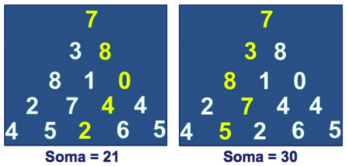
```
Fibonacci(n)
1.  se (n = 0 ou n = 1)
2.  então retorna (n)
3.  senão
4.     F1 = 0
5.     F2 = 1
6.     para i = 1 até n - 1 faça
7.        F = F1 + F2
8.        F1 = F2
9.        F2 = F
10.    retorna (F)
```

O Fibonacci recursivo tem complexidade $O(2^n)$ mas com a programação dinâmica, temos:

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

Pirâmide de números

- 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.



- Então, o problema pode ser resolvido recursivamente.

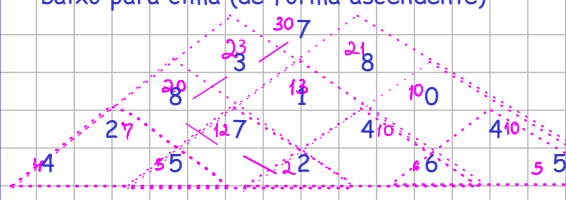
```
Max(i,j)
% P[i,j] : j-ésimo número da i-ésima linha
% n : altura da pirâmide
1.  se (i = n)
2.     então soma = P[i,j]
3.  senão
4.     esq = Max(i+1,j)
5.     dir = Max(i+1,j+1)
6.     se (esq > dir)
7.        então soma = P[i,j] + esq
8.        senão soma = P[i,j] + dir
9.  retorna (soma)
```

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

Mas dessa forma, temos o mesm o problem sendo executado várias vezes, dando em uma complexidade $O(2^n)$

- Para melhorar isso, podemos usar PD:
  A ideia é guardar o valor calculado para cada subproblema em uma tabela para, quando for o caso, reaproveitá-lo.

- Além disso, começaremos a partir do fim! De baixo para cima (de forma ascendente)



- 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: $O(n^2)$.

Assim, a complexidade cai para $O(n^2)$

Algoritmos aproximados

✔ Problema do caixeiro viajante -> Complexidade O(n!)

✔ Como são problemas intratáveis, o algoritmo nem sempre dá a solução ótima

✔ A diferença é que, aqui, nos algoritmos aproximados, ele entrega uma solução dentro de um limite especificado (tem compromisso com a solução ótima)

✔ Dependendo do problema, o objetivo será minimizar ou maximizar o valor da solução ótima para um problema

Heurística

✔ Sem compromisso com a solução ótima

✔ Pode produzir um bom resultado, até mesmo a solução ótima, mas também pode não produzir solução alguma ou uma solução muito distante do ótimo.