Projeto e Análise de Algoritmos

Prof. Dr. Ednaldo B. Pizzolato

www.dc.ufscar.br/~ednaldo

ednaldo@dc.ufscar.br

RELAÇÃO ESPAÇO X TEMPO E PROGRAMAÇÃO DINÂMICA

Agenda

- Parte A – Relação Espaço x Tempo
  - Introdução
  - Algoritmo de ordenação
  - String matching
    - Algoritmo de Horspool
    - Algoritmo de Boyer-Moore (intro)
  - Hashing (intro)
  - B-Trees (intro)
- Parte B – Programação Dinâmica

INTRODUÇÃO

Introdução

- O principal conceito envolvido com esta técnica é a avaliação do que é mais crucial para o desenvolvimento do algoritmo: espaço ou tempo.
- Considere, por exemplo, uma situação em que calcular o resultado de uma função para diversos pontos é importante, mas ao mesmo tempo consome muito tempo. Se o tempo é importante, basta alocar mais espaço e armazenar os valores pré-calculados em uma tabela.

Introdução

- Isso, na verdade, já foi muito empregado no passado com a elaboração de livros com valores pré-calculados de determinadas funções matemáticas.
- O conceito geral da técnica envolve pré-processamento/otimização de uma entrada de dados e armazenamento de informações extras (como resultado do processamento) de forma a obter uma aceleração na resolução do problema. Anany Levitin chama isso de ***melhoramento de entrada de dados***.

Introdução

- Dois tipos de algoritmos são estudados envolvendo este conceito:
  - Ordenação por contagem
  - Algoritmos de busca de palavras em textos

Introdução

- Dois tipos de algoritmos são estudados envolvendo este conceito:
  - Ordenação por contagem
  - Algoritmo de Horspool (string matching)
  - Algoritmo de Boyer-Moore (string matching)

Introdução

- Outro tipo de abordagem é o simples uso de espaço extra para facilitar o acesso rápido ou mais flexível aos dados. Anany Levitin chama isso de ***pré-estruturação***.
- Esta estruturação (mais voltada para o acesso aos dados) pode ser notada em soluções envolvendo hashings e indexação com árvores B.

Introdução

- Há ainda uma terceira forma de se utilizar a técnica de espaço por tempo associada a ***programação dinâmica***.
- Como abordado no capítulo 8 do livro “Introduction to the Design and Analysis of Algorithms – 3rd edition”, a programação dinâmica está associada a soluções de subproblemas que se inter-relacionam (overlap). A ideia é armazenar os resultados dos subproblemas em uma tabela e, a partir dela, obter a solução do problema geral.

ORDENAÇÃO POR CONTAGEM

Ordenação por contagem

- Uma forma bem simples de se ordenar um vetor é contar quantos elementos são menores que um determinado valor sendo avaliado. Se, por exemplo, detectar-se que existem 10 elementos menores que ele, então ele será o 11º elemento do vetor. Basta, depois, copiar o elemento em um outro vetor, na posição correta (11 se começar do 1 ou 10 se começar do 0). O algoritmo é chamado Comparison-CountingSort.

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

A	Cont
10	0
22	0
31	0
4	0
16	0
9	0
27	0
19	0
15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	0
j →	22	0
	31	0
	4	0
	16	0
	9	0
	27	0
	19	0
	15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	0
j →	22	1
	31	0
	4	0
	16	0
	9	0
	27	0
	19	0
	15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	0
	22	1
j →	31	0
	4	0
	16	0
	9	0
	27	0
	19	0
	15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	0
	22	1
j →	31	1
	4	0
	16	0
	9	0
	27	0
	19	0
	15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	0
	22	1
	31	1
j →	4	0
	16	0
	9	0
	27	0
	19	0
	15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	1
	22	1
	31	1
j →	4	0
	16	0
	9	0
	27	0
	19	0
	15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	1
	22	1
	31	1
	4	0
j →	16	0
	9	0
	27	0
	19	0
	15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	1
	22	1
	31	1
	4	0
j →	16	1
	9	0
	27	0
	19	0
	15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

i →

j →

A	Cont
10	1
22	1
31	1
4	0
16	1
9	0
27	0
19	0
15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

i →

j →

A	Cont
10	2
22	1
31	1
4	0
16	1
9	0
27	0
19	0
15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

i →

j →

A	Cont
10	2
22	1
31	1
4	0
16	1
9	0
27	0
19	0
15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

i →

j →

A	Cont
10	2
22	1
31	1
4	0
16	1
9	0
27	1
19	0
15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	2
	22	1
	31	1
	4	0
	16	1
	9	0
	27	1
j →	19	0
	15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	2
	22	1
	31	1
	4	0
	16	1
	9	0
	27	1
j →	19	1
	15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	2
	22	1
	31	1
	4	0
	16	1
	9	0
	27	1
	19	1
j →	15	0

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	2
	22	1
	31	1
	4	0
	16	1
	9	0
	27	1
	19	1
j →	15	1

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        cont[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                cont[j] ← cont[j] + 1
            senão
                cont[i] ← cont[i] + 1
    para i ← 0 até n-1 faça
        S[cont[i]] ← A[i]
    retorna S
```

	A	Cont
i →	10	2
	22	1
	31	1
	4	0
	16	1
	9	0
	27	1
	19	1
j →	15	1

O que deu para descobrir até o momento?

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        contador[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                contador[j] ← contador[j] + 1
            senão
                contador[i] ← contador[i] + 1
    para i ← 0 até n-1 faça
        S[contador[i]] ← A[i]
    retorna S
```

Eficiência?

Ordenação por contagem

```
ComparisonCountingSort(A[0..n-1])
    para i ← 0 até n-1 faça
        contador[i] ← 0
    para i ← 0 até n-2 faça
        para j ← i+1 até n-1 faça
            se A[i] < A[j] então
                contador[j] ← contador[j] + 1
            senão
                contador[i] ← contador[i] + 1
    para i ← 0 até n-1 faça
        S[contador[i]] ← A[i]
    retorna S
```

Eficiência?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$$= \sum_{i=0}^{n-2} (n-1+i) = \frac{n(n-1)}{2}$$

Ordenação por contagem

- Em resumo, o algoritmo faz o mesmo número de comparações que o algoritmo de seleção (SelectionSort) e, além disso, usa espaço extra.
- Por outro lado, o algoritmo faz as atribuições de forma cirúrgica.
- Uma variação deste algoritmo é o Distribution Counting (ver animação no ambiente virtual).

OTIMIZAÇÃO DE ENTRADA DE DADOS

Otimização de entrada de dados

Para ilustrar a utilidade da otimização (ou melhoramento) da entrada de dados, será utilizado o problema de *string matching*, ou seja, encontrar uma sequência (padrão) de m caracteres em um texto de n caracteres.

Considerando o algoritmo de força bruta que compara os caracteres da sequência buscada com os do texto um a um, teríamos, no pior caso, um algoritmo pertencente à classe O(n.m). No caso médio, com textos em linguagem natural, espera-se que o algoritmo pertença à classe O(n+m).

Otimização de entrada de dados

Só para ilustrar, vamos considerar o exemplo de busca por um padrão em uma sequência genética.

Uma **sequência de DNA** ou **sequência genética** é uma série de letras representando a estrutura primária de uma molécula ou cadeia de DNA, real ou hipotética, com a capacidade de carregar informações.

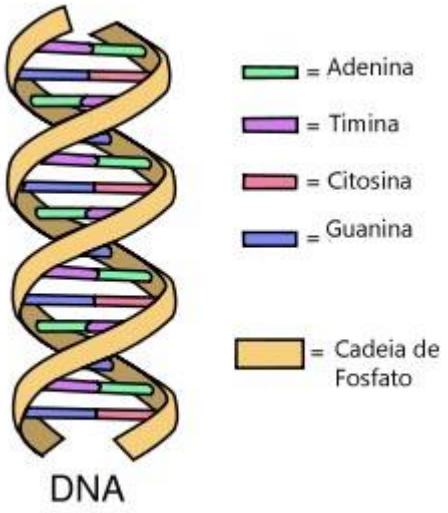
As letras possíveis são *A*, *C*, *G* e *T*, representando os quatro nucleotídeos (subunidades) de uma cadeia de DNA – as bases adenina, citosina, guanina e timina.

Otimização de entrada de dados

C G T A A A C G G G T A T T G

Deseja-se buscar a sequência

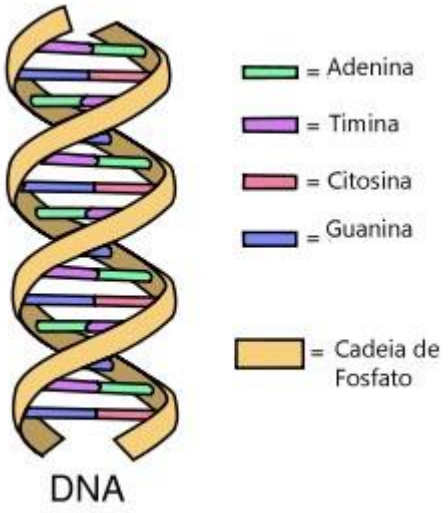
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

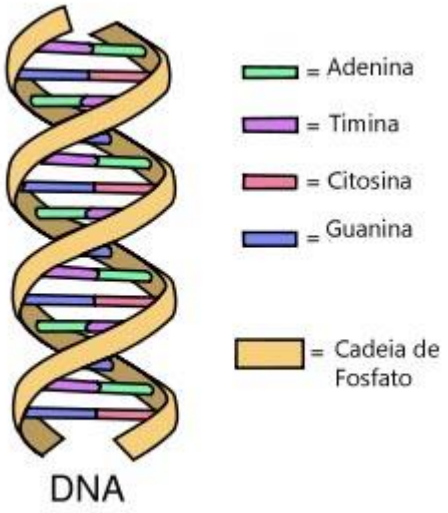
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

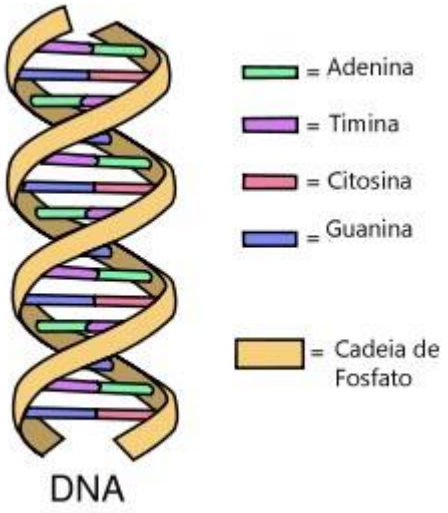
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

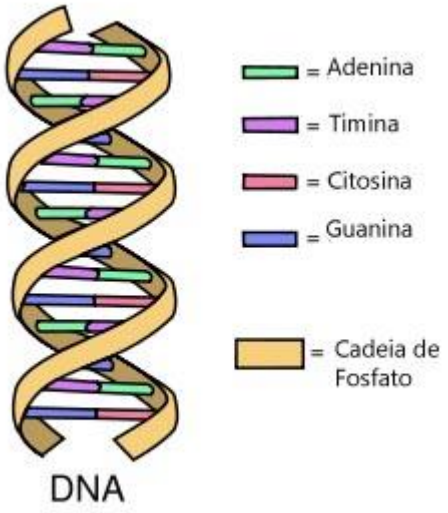
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

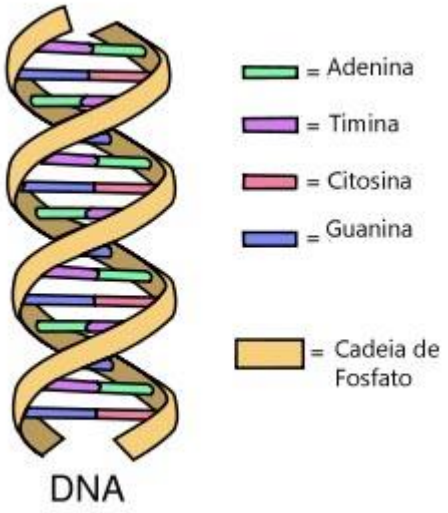
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

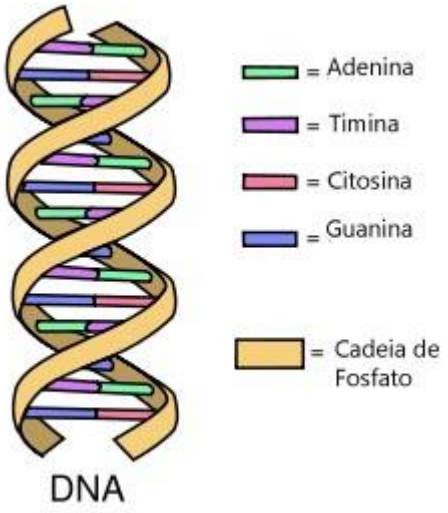
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

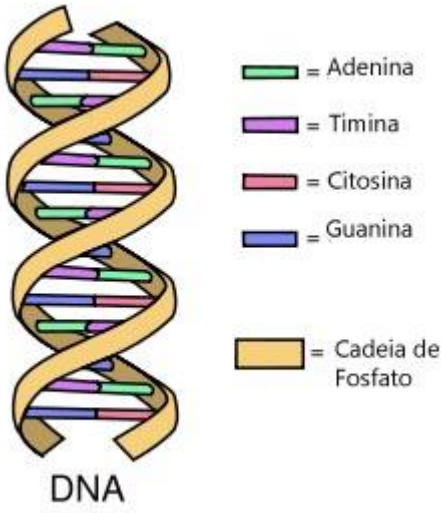
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

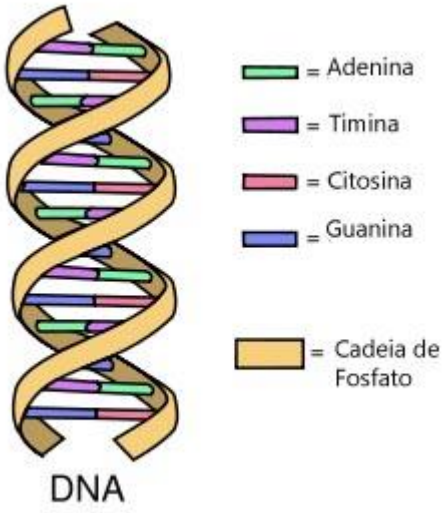
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

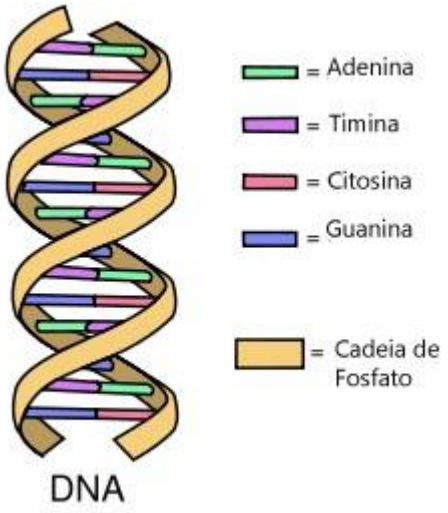
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

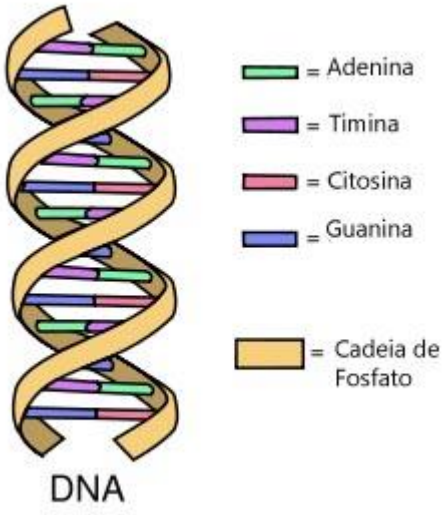
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

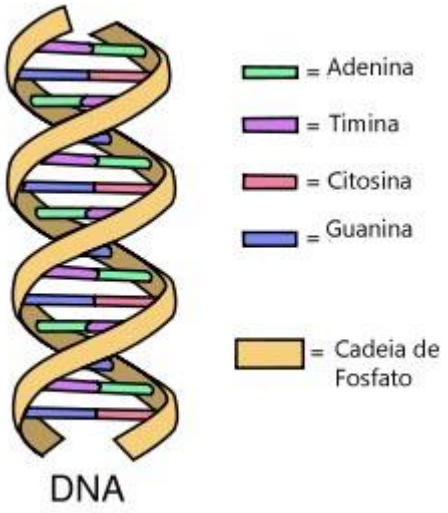
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

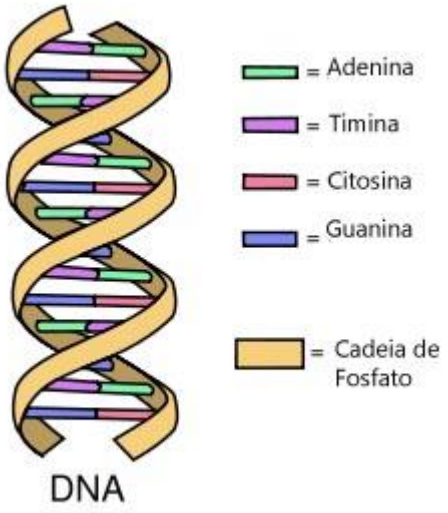
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

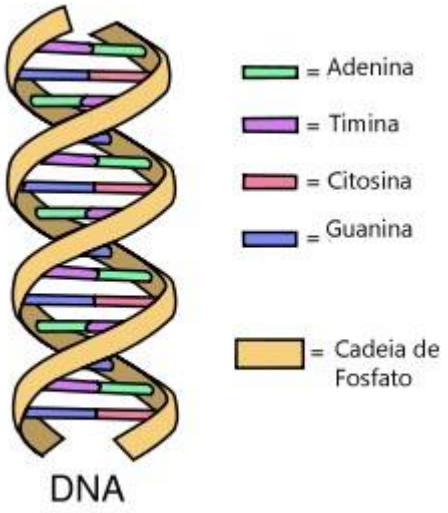
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

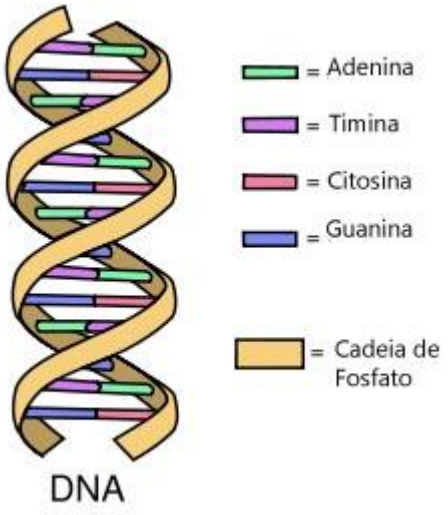
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

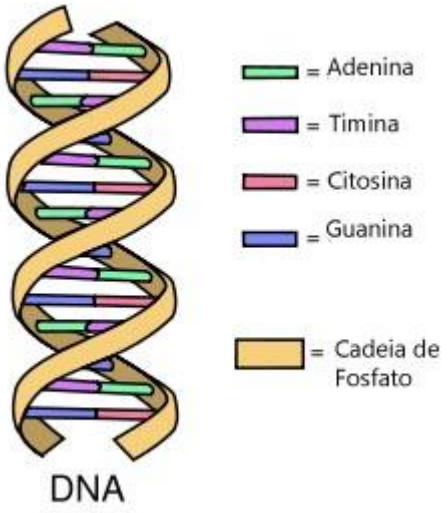
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

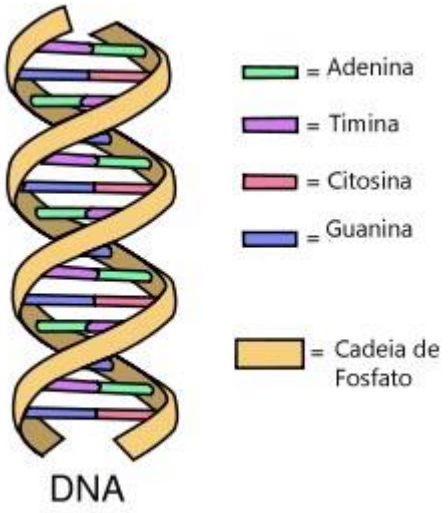
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

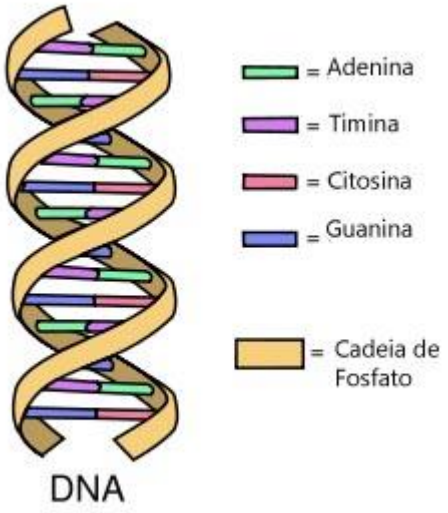
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

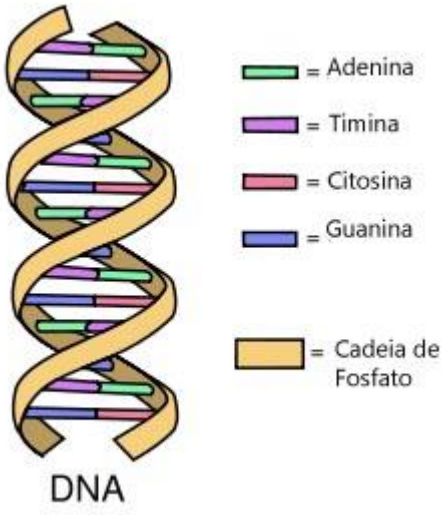
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

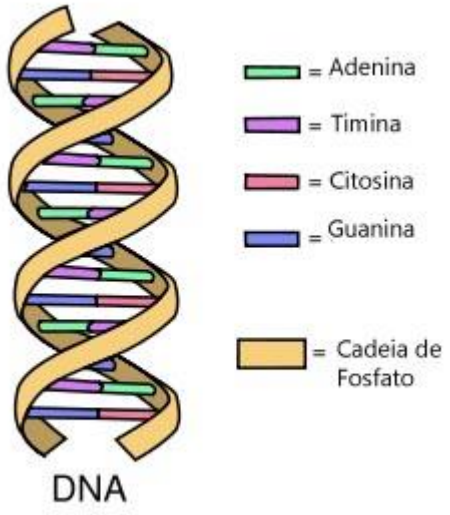
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

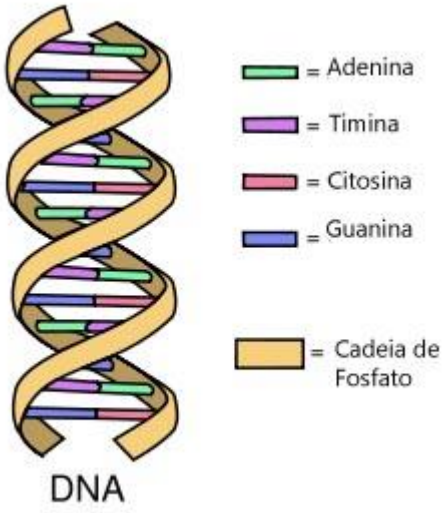
G T A T



Otimização de entrada de dados

C G T A A A C G G G T A T T G

G T A T



Otimização de entrada de dados

Diversos algoritmos mais eficientes que os de força bruta foram descobertos para resolver tal problema. A maioria utiliza a técnica de *melhoramento da entrada de dados*:

1. Pre-processar o padrão para se obter uma informação rele-vante;
2. Armazenar a informação em uma tabela;
3. Utilizar tal informação durante a busca pelo padrão em um texto.

Esta é a ideia de alguns algoritmos como os conhecidos: Horspool, Boyer-Moore e Knuth-Morris-Pratt (KMP)

Otimização de entrada de dados

- Algoritmo de Horspool

Considere a busca do padrão BARBER em um texto:

s_0, ... c ... s_{n-1}

BARBER

Começando com o último R do padrão e movendo da direita para a esquerda, são comparados pares de caracteres do padrão e do texto.

Se todos os caracteres do padrão combinarem, uma cadeia foi encontrada.

Otimização de entrada de dados

- Se ocorrer uma combinação mal sucedida, deve-se deslocar o padrão o **máximo possível** sem correr o risco de perder uma cadeia coincidente.
- A questão é identificar o **máximo possível** de tal forma que o risco de perder uma cadeia não exista.
- O algoritmo de Horspool determina o tamanho do deslocamento olhando para o caractere **c** do texto que foi alinhado com o último caractere padrão.

Otimização de entrada de dados

- Caso 1: Se não há **c** no padrão, pode-se deslocar o padrão pelo seu comprimento total.
- Caso 2: Caso existam ocorrências do caractere c no padrão, mas não é o último, o deslocamento deve alinhar a ocorrência mais a direita do padrão com o **c** do texto.
- Caso 3: Se acontecer de **c** ser o último caractere no padrão e não existirem outros **c**´s dentre os m-1 outros caracteres, o deslocamento deve ser similar ao caso 1.
- Caso 4: Pode acontecer de **c** ser o último caractere no padrão e também existirem outros **c**´s dentre os m-1 outros caracteres. Neste caso, o deslocamento deve ser similar ao caso 2.

Otimização de entrada de dados

- Estes exemplos demonstram que são possíveis deslocamentos maiores que 1 posição (como os feitos pelo algoritmo de força bruta).
- Entretanto, se o algoritmo tiver que verificar todos os caracteres do padrão a cada deslocamento, ele perderá sua eficiência.
- É aí que entra a ideia de melhoramento da entrada de dados para evitar comparações repetitivas desnecessárias.

Algoritmo de Horspool

- Pode-se pre-calcular tamanhos de deslocamentos e armazena-los em uma tabela.
- A tabela será indexada por todos os caracteres possíveis que possam ser encontrados em um texto, incluindo pontuação, espaço e outros símbolos especiais.
- A tabela será preenchida com tamanho de deslocamentos.

Algoritmo de Horspool

- Para cada caractere c podemos calcular o valor do deslocamento pela fórmula:

  Se **c** não estiver dentre os m-1 primeiros caracteres do padrão, então

  t(c) = comprimento m do padrão

  Caso contrário

  t(c) = distância do c mais a direita dentre os m-1 caracteres do padrão e seu último caractere

  Por exemplo, para o padrão BARBER, todas as entradas da tabela serão iguais a 6, exceto as entradas correspondentes a A, B, E e R que serão 4, 2, 1 e 3 respectivamente.

Algoritmo de Horspool

Algoritmo ShiftTable(P[0..m-1])

for i ← 0 até TAM-1 faça

tabela[i] ← m

for j ← 0 até m-2 faça

tabela[P[j]] ← m – 1 – j

Retorna tabela

Onde TAM indica o número máximo de caracteres do alfabeto em questão.

Letra	Desloca
A	4
B	2
C	6
D	6
E	1
F	6
G	6
H	6
I	6
...	6
R	3
...	6

BARBER

Algoritmo de Horspool

- Passo 1: Para um dado padrão de comprimento m e o alfabeto usado tanto no padrão como no texto, construir a tabela de deslocamentos;
- Passo 2: Alinhar o padrão como começo do texto;
- Passo 3: Repetir até encontrar uma *substring* coincidente ou o padrão atingir o último caractere do texto.

Algoritmo de Horspool

- Passo 3 (continuação):
  - Começando pelo último caractere do padrão, comparar os caracteres correspondentes no padrão e no texto até que todos os m caracteres coincidam ou até que ocorra uma combinação mal sucedida;
  - No caso de combinação mal sucedida, ler a entrada t(c) da coluna c da tabela de deslocamentos, onde c é o caractere do texto sendo alinhado com o último caractere do padrão;
  - Deslocar o padrão t(c) caracteres a direita.

Algoritmo de Horspool

```
HorspoolMatching(P[0..m-1], T[0..n-1])
    ShiftTable(P[0..m-1])   // cria a tabela de deslocamentos
    i ← m -1        // POSIÇÃO DO ÚLTIMO CARACTERE DO PADRÃO
    enquanto i ≤ n -1 faça  // ENQUANTO NÃO SE ATINGIR FIM DO TEXTO
        k ← 0                           // NÚMERO DE CARACTERES IDÊNTICOS
        enquanto k ≤ m -1 E P[m-1-k] = T[i-k] faça
            k ← k + 1
        se k = m então  retorna i – m + 1  // ACHOU !!!
        senão i ← i + Tabela[T[i]]     // AQUI OCORRE O DESLOCAMENTO
    retorna -1
```

Algoritmo de Horspool - exemplo

- Buscar a palavra BARBER em um texto:

letra	A	B	C	D	E	...	R	...	Z	-	...
Shift	4	2	6	6	1	6	3	6	6	6	6

J	I	M		S	A	W		M	E		I	N		A		B	A	R	B	E	R	S	H	O	P
B	A	R	B	E	R																				

i ← m -1 // NOSSO CASO I ← 5

k ← 0 // NÚMERO DE CARACTERES IDÊNTICOS

enquanto k ≤ m -1 E P[m-1-k] = T[i-k] faça

    k ← k + 1

se k = m então retorna i – m + 1

senão i ← i + Tabela[T[i]]

Algoritmo de Horspool - exemplo

- Buscar a palavra BARBER em um texto:

letra	A	B	C	D	E	...	R	...	Z	-	...
Shift	4	2	6	6	1	6	3	6	6	6	6

J	I	M		S	A	W		M	E		I	N		A		B	A	R	B	E	R	S	H	O	P
B	A	R	B	E	R																				

i ← m -1 // NOSSO CASO I ← 5

k ← 0 // NÚMERO DE CARACTERES IDÊNTICOS

enquanto k ≤ m -1 E P[m-1-k] = T[i-k] faça

    k ← k + 1

se k = m então retorna i – m + 1

senão i ← i + Tabela[T[i]]

Algoritmo de Horspool - exemplo

- Buscar a palavra BARBER em um texto:

letra	A	B	C	D	E	...	R	...	Z	-	...
Shift	4	2	6	6	1	6	3	6	6	6	6

J	I	M		S	A	W		M	E		I	N		A		B	A	R	B	E	R	S	H	O	P
				B	A	R	B	E	R																

i ⟵ m -1 // NOSSO CASO I ← 5

k ⟵ 0 // NÚMERO DE CARACTERES IDÊNTICOS

enquanto k ≤ m -1 E P[m-1-k] = T[i-k] faça

    k ⟵ k + 1

se k = m então retorna i – m + 1

senão i ⟵ i + Tabela[T[i]]

Algoritmo de Horspool - exemplo

- Buscar a palavra BARBER em um texto:

letra	A	B	C	D	E	...	R	...	Z	-	...
Shift	4	2	6	6	1	6	3	6	6	6	6

J	I	M		S	A	W		M	E		I	N		A		B	A	R	B	E	R	S	H	O	P
				B	A	R	B	E	R																

i ⟵ m -1 // NOSSO CASO I ← 5

k ⟵ 0 // NÚMERO DE CARACTERES IDÊNTICOS

enquanto k ≤ m -1 E P[m-1-k] = T[i-k] faça

k ⟵ k + 1

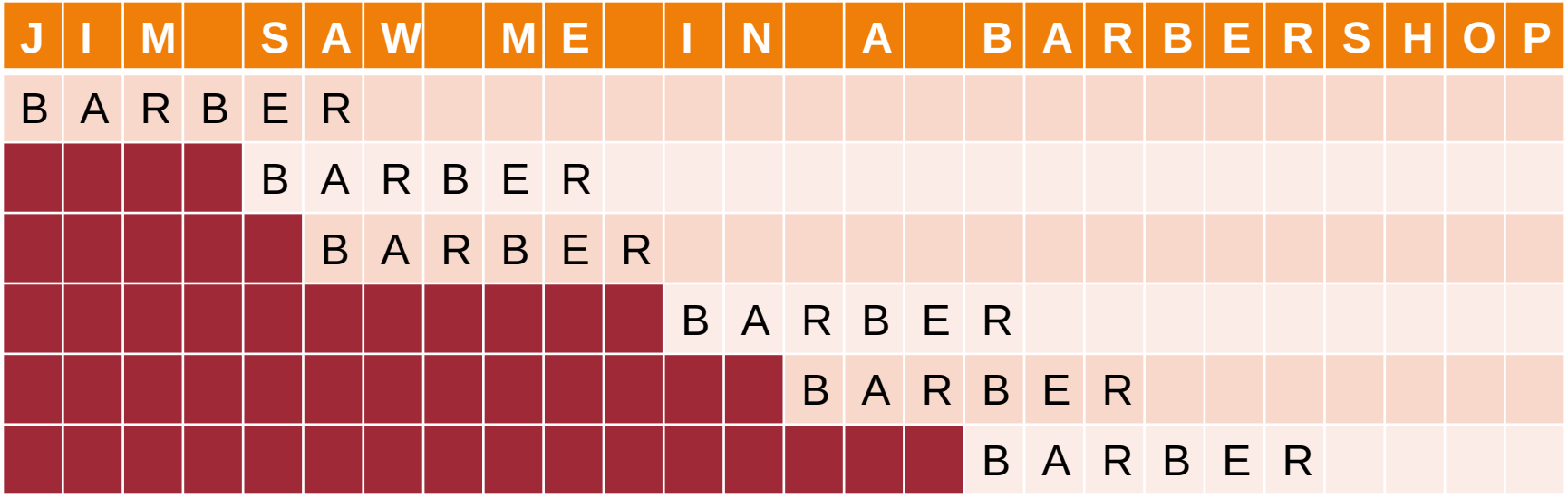
se k = m então retorna i – m + 1

senão i ⟵ i + Tabela[T[i]]

Algoritmo de Horspool - exemplo

- Buscar a palavra BARBER em um texto:

letra	A	B	C	D	E	...	R	...	Z	-	...
Shift	4	2	6	6	1	6	3	6	6	6	6



Algoritmo de Horspool

- A eficiência no pior caso é O(n.m)
- Para textos aleatórios, a eficiência é Θ(n)
- Na média, o algoritmo é mais rápido que o de força bruta.

Otimização de entrada de dados

- Algoritmo Boyer-Moore

Se a comparação do caractere mais a direita do padrão com o caractere correspondente **c** no texto falhar, o algoritmo faz a mesma coisa que o algoritmo Horspool, ou seja, desloca o padrão a direita pelo número de caracteres pre-calculados (armazenados na tabela).

Se houver um número k (k > 0 e k < m) de comparações bem sucedidas, aí é que aparece a diferença entre os algoritmos.

Otimização de entrada de dados

- Nesta situação, o algoritmo B-M determina o tamanho do deslocamento considerando duas quantidades:
  - Deslocamento pelo símbolo incorreto (bad-symbol shift);
  - Deslocamento pelo bom sufixo (good-suffix shift)

PRE-ESTRUTURAÇÃO

Hashing

- A técnica Hashing baseia-se na ideia de distribuir chaves entre um arranjo unidimensional H[0... M-1] chamado de tabela hash.
- A distribuição é feita calculando-se para cada uma das chaves o valor de alguma função pré-definida h, chamada de função hash.
- Esta função atribui um inteiro entre 0 e m-1, chamado de endereço hash para uma chave.

Hashing

- Por exemplo, se as chaves forem inteiros não negativos, uma função hash pode ser da forma:

$$h(K) = K \bmod m$$

Se as chaves forem letras do alfabeto, podemos inicialmente atribuir a uma letra sua posição no alfabeto indicado por ord(K) e aplicar a mesma função hash.

Hashing

- Alguém pode perguntar como teríamos que proceder se as chaves fossem cadeias de caracteres (strings).
- Para strings de tamanho s, poder-se-ia utilizar, por exemplo, a fórmula a seguir:

$$\left(\sum_{i=0}^{s-1} ord(c_i) \right) \bmod m$$

Hashing

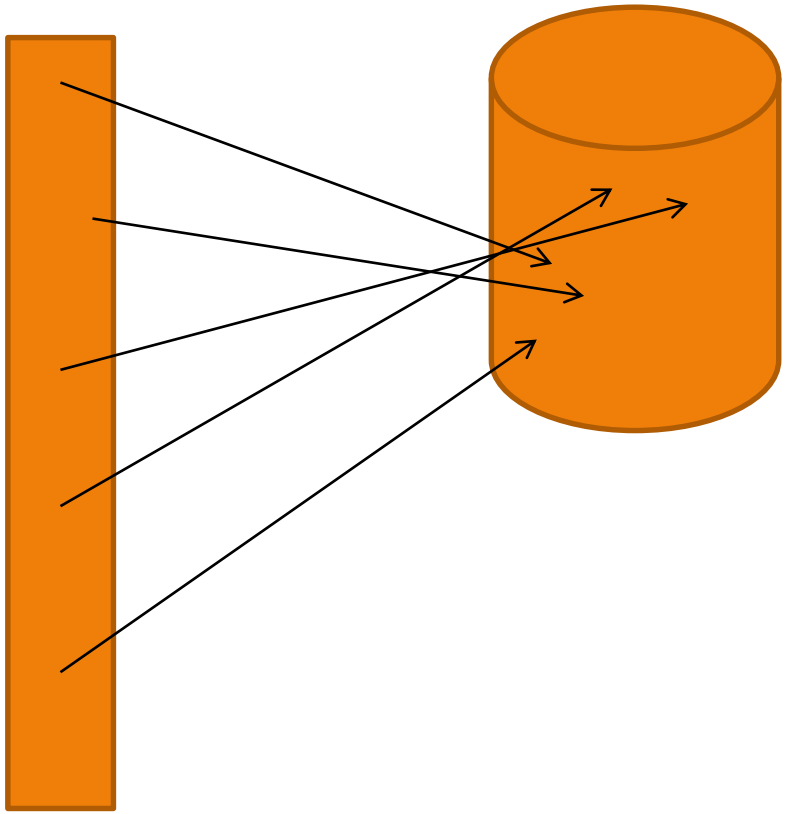
- Uma função Hash deve satisfazer dois requisitos:
  - Deve distribuir as chaves entre as células da tabela de uma maneira tão justa quanto possível;
  - Uma função hash deve ser fácil de calcular

Se escolhermos uma tabela hash de tamanho m (m menor que o número de chaves – n), existirão colisões.

Hashing

- Como colisões podem ocorrer, é necessário que existam mecanismos que resolvam tal problema.
- Em hashing temos o open hashing e o closed hashing.

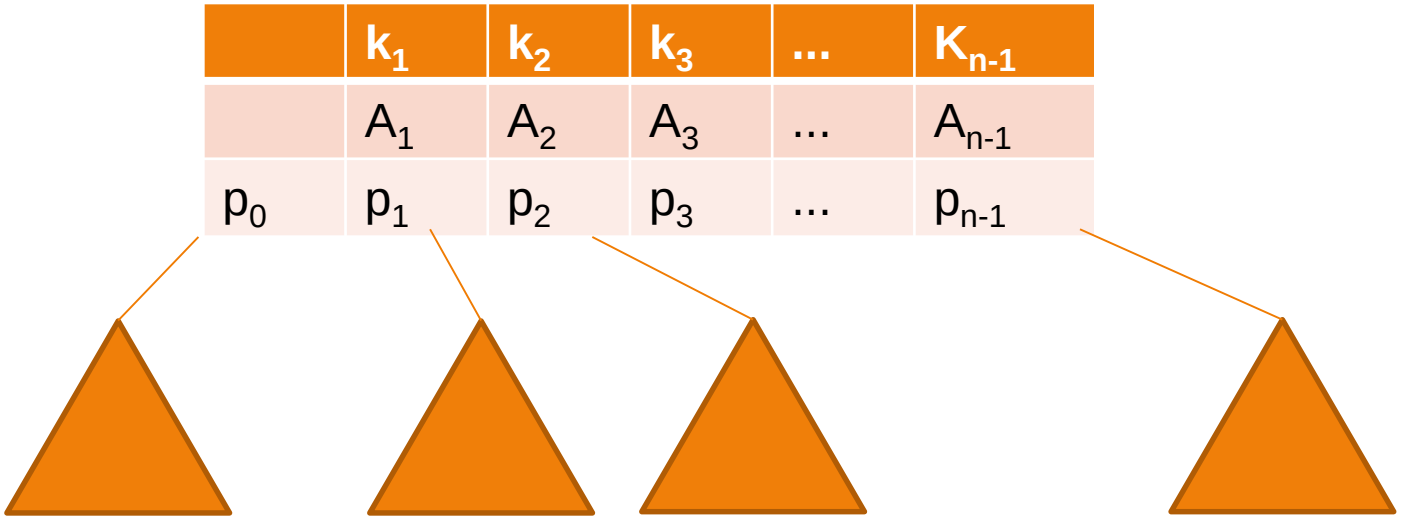
Árvores B



- A ideia de utilizar espaço extra para facilitar rápido acesso a uma determinada informação é particularmente importante quando se trata de uma quantidade muito grande de dados.
- Os índices fornecem informações sobre a localização dos registros segundo uma chave de acesso.
- Claro que podem existir várias chaves de acesso e, portanto, vários índices.

Árvores B

- Uma das formas de se organizar um índice é através de árvores B.



Árvores B

- Uma árvore B de ordem m ≥ 2 deve satisfazer as seguintes propriedades estruturais:
  - A raiz ou é uma folha ou tem entre 2 e m filhos;
  - Cada nó, exceto a raiz e as folhas tem entre teto(m/2) e m filhos (e, portanto, entre teto(m/2)-1 e m-1 chaves);
  - A árvore é perfeitamente balanceada, ou seja, suas folhas estão todas no mesmo nível.

RESUMO

Resumo

- O uso de espaço extra para “ganhar tempo” ou de tempo extra para “economizar espaço” é um conceito amplamente conhecido e estudado por cientistas teóricos e práticos.
- O mais comum é o uso de espaço extra para “ganhar tempo” de execução/processamento.
- Dentro das variedades da técnica, a de otimização de entrada é a que objetiva pré-processar os dados de entrada de forma a acelerar o algoritmo posteriormente.

Resumo

- Os algoritmos de Horspool e de Boyer-Moore são exemplos de melhoramento da entrada de dados com o objetivo de obter melhor desempenho (tempo) do algoritmo.
- Pré-estruturação é outra variação da técnica que explora a troca de espaço por tempo que explora o conceito de uso de memória extra para melhoria do desempenho (tempo). Hashing e árvores B+ são 2 importantes exemplos.

Resumo

- Por fim duas considerações importantes:
  - Nem sempre é necessário "sacrificar" memória para se obter melhor desempenho. Vários algoritmos podem ser desenvolvidos com uso eficiente de memória e ao mesmo tempo conseguem apresentar desempenhos elevados;
  - A técnica desconsidera o uso de técnicas de compressão de dados. Seria possível combinar técnicas – em algumas situações – para se obter algoritmos eficientes tanto do ponto de vista de espaço como de tempo.

THE END