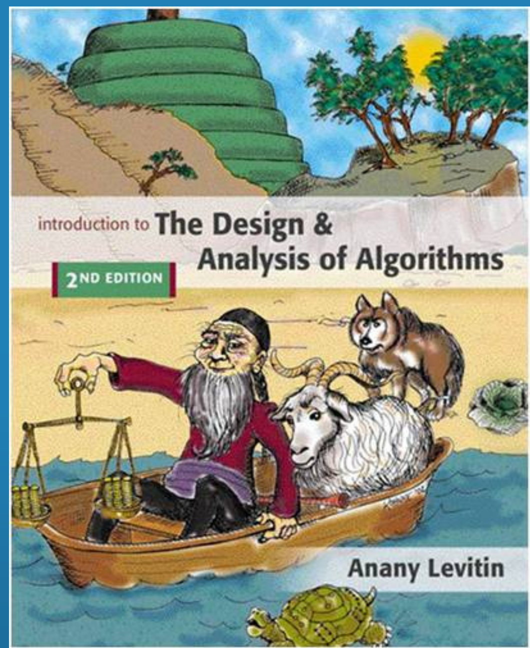
Chapter 7

Space and Time Tradeoffs



Números Primos

- Um **número primo** é um inteiro $p > 1$ que somente é divisível por 1 e por ele mesmo:
  - Se p é um número primo, então $p = a \times b$ para inteiros $a \leq b$ implica que $a = 1$ e $b = p$;
  - Os 10 primeiros primos são: 2, 3, 5, 7, 11, 13, 17, 19, 23 e 29
- Qualquer número não-primo é chamado de **composto**
- **Sugestões???**

Encontrando Primos

- Força bruta

- Ignorar pares

- Até $\sqrt{n}$

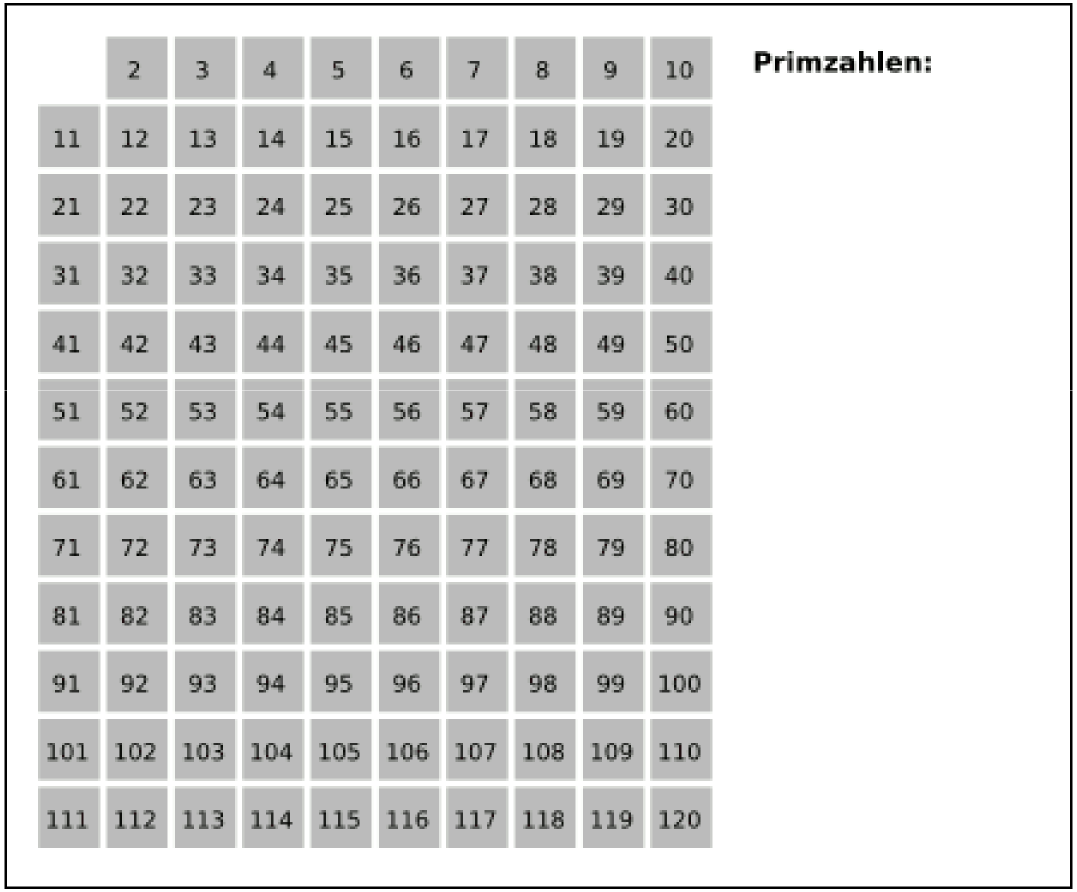
Encontrando Primos

```
int is_prime (unsigned int x) {
     unsigned int i;

     if (x > 2 && x % 2 == 0)
          return 0;

     i = 3;
     while (i <= sqrt(x)+1) {
          if (x % i == 0)
               return 0;
          i += 2;
     }
     return 1;
}
```

Geração de uma lista de Primos: Crivo de Eratóstenes



http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Crivo de Eratóstenes

- A complexidade do algoritmo é
  - $O((n \log n)(\log \log n))$

- e possui um requisito de memória de $O(n)$

)

Space-for-time tradeoffs

Two varieties of space-for-time algorithms:

- ***input enhancement*** **— preprocess the input (or its part) to store some info to be used later in solving the problem**
  - **counting sorts**
  - **string searching algorithms**

- ***prestructuring*** **— preprocess the input to make accessing its elements easier**
  - **hashing**
  - **indexing schemes (e.g., B-trees)**

- **Dynamic programming**

Sorting by counting

- **Idea:** **Comparison Counting Sorting**

62 31 84 96 19 47

Sorting by counting

- **Idea**
  - **Comparison Counting Sorting**
  - **62 31 84 96 19 47**

- **Algorithm**
  - **Contar, para cada elemento i, os que sao menores que i**
  - **Dado que há k elementos menores que i, ele vai para a posição i**

ComparisonCountingSort

```
ALGORITHM ComparisonCountingSort (A[0..n-1])
// A[0..n-1]: input
// S[0..n-1]: output

for i =0 to n-1 do Count[i]=0
for i=0 to n-2 do
    for j=i+1 to n-1 do
        if A[i] < A[j] Count[j] +=1
        else Count[i] +=1
for i =0 to n-1 do S[Count[i]] = A[i]
return S
```

time efficiency? n(n-1)/2

Counting Sort special case: Distribution sorting

- time efficiency
- CountingSort = n(n-1)/2
- InsertionSort = n(n-1)/2
  - (decrease & conquer)
- CountingSort vs InsertionSort?

Counting Sort special case: Distribution sorting

- **time efficiency CountingSort = n(n-1)/2**
  - Same of the InsertionSort? n(n-1)/2
  - CountingSort x InsertionSort?

- **Quando pode valer a pena?**

13 11 12 13 12 12

98 100 102 120 112 100 118 115 116 115 98 99

ComparisonCountingSort

```
ALGORITHM DistributionCountingSort (A[0..n-1], l, u)
// A[0..n-1]: input
// S[0..n-1]: output

for j=0 to u-l do D[j]=0
for i=0 to n-1 do
    j = A[i] -l;  D[j]  +=1
for j=1 to u-l do D[j] = D[j-1] + D[j]
for i =n-1 to n do
    j = A[i] -l;  S[D[j]-1] = A[i];  D[j] -=1

return S
```

time efficiency?

Review: String searching by brute force

pattern**: a string of *****m***** characters to search for**

text**: a (long) string of *****n***** characters to search in**

Brute force algorithm

Step 1 **Align pattern at beginning of text**

Step 2 **Moving from left to right, compare each character of pattern to the corresponding character in text until either all characters are found to match (successful search) or a mismatch is detected**

Step 3 **While a mismatch is detected and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2**

Brute-force string matching

```
for (i=0; T[i] != '\0'; i++) {
  for ( j=0;
        T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j];
        j++) ;
  if (P[j] == '\0') found a match
}
```

String searching by preprocessing

Several string searching algorithms are based on the input enhancement idea of **preprocessing the pattern**

- **Knuth-Morris-Pratt (KMP) (1977)** algorithm preprocesses pattern left to right to get useful information for later searching
- **Boyer –Moore (1977)** algorithm preprocesses pattern right to left and store information into two tables
- **Horspool's (1980)** algorithm simplifies the Boyer-Moore algorithm by using just one table

Horspool's Algorithm

A simplified version of Boyer-Moore algorithm:

- **preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs**

- **always makes a shift based on the text's character *c* aligned with the last character in the pattern according to the shift table's entry for *c***

How far to shift?

Look at first (rightmost) character in text that was compared:

- The character is not in the pattern

  `.....BAS........................` (`c` not in pattern)

  `BAOBAB`

- The character is in the pattern (but not the rightmost)

  `.......O........................` (`O` occurs once in pattern)

  `BAOBAB`

  `......A.........................` (`A` occurs twice in pattern)

  `BAOACB`

- The rightmost characters do match

  `....PAB.........................`

  `CAOAB` *no others*

  `.....AB.........................`

  `BAOBAB` *there are others*

Shift table

- Shift sizes can be precomputed by the formula

  distance from c's rightmost occurrence in pattern among its first m-1 characters to its right end

  $t(c)$ =

  pattern's length m, otherwise

  by scanning pattern before search begins and stored in a table called *shift table*

- Shift table is indexed by text and pattern alphabet
  Eg, for `BAOBAB` :

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6

Example of Horspool's alg. application

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

```
BARD LOVED BANANAS
BAOBAB (L=6)
      BAOBAB (B=2)
        BAOBAB (N=6)
              BAOBAB
```
(unsuccessful search)

PINGADO em

ALMOCEI PINGA COM LINGUADO PINGADO

- PINGADO outras
- 6543217 7

- 1234567890123456789012345678901234
- ALMOCEI PINGA COM LINGUADO PINGADO
- PINGADO (I=5)
- -----PINGADO (^G=3)
- ---PINGADO (^C=7)
- -------PINGADO (^G=3)
- ---PINGADO (^D=1)
- -PINGADO (^=ADO, U)
- PINGADO (^U=7)
- -------PINGADO (^D=1)
- -PINGADO (=PINGADO)

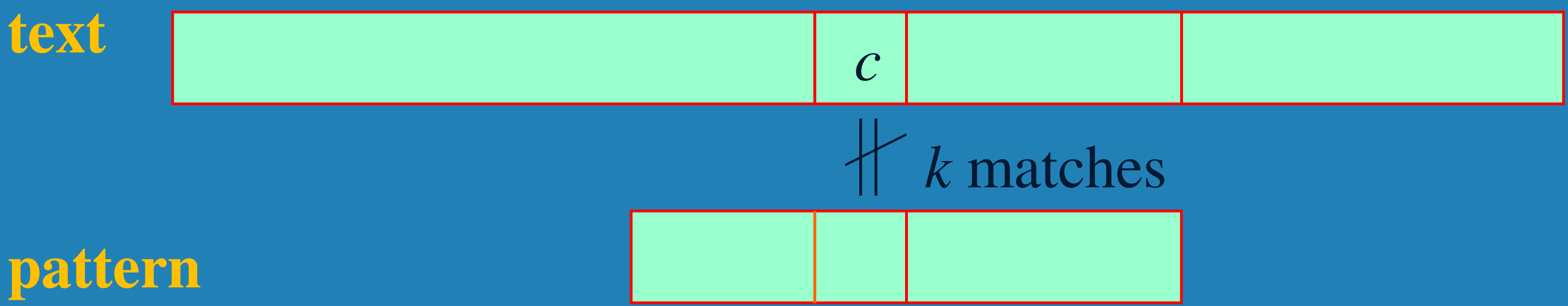
Boyer-Moore algorithm

Based on same two ideas:

- **comparing pattern characters to text from right to left**
- **precomputing shift sizes in two tables**
  - ***bad-symbol table*** **indicates how much to shift based on text's character causing a mismatch**
  - ***good-suffix table*** **indicates how much to shift based on matched part (suffix) of the pattern**

Bad-symbol shift in Boyer-Moore algorithm

- **If the rightmost character of the pattern doesn't match, BM algorithm acts as Horspool's**
- **If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern's characters match or a mismatch on text's character *c* is encountered after $k > 0$ matches**



bad-symbol shift $d_1 = \max\{t_1(c) - k, 1\}$

Good-suffix shift in Boyer-Moore algorithm

- Good-suffix shift d_2 is applied after $0 < k < m$ last characters were matched

- $d_2(k)$ = the distance between matched suffix of size k and its rightmost occurrence in the pattern that is not preceded by the same character as the suffix

  Example: `CABABA` $d_2(1) = 4$

- If there is no such occurrence, match the longest part of the k-character suffix with corresponding prefix;
  if there are no such suffix-prefix matches, $d_2(k) = m$

  Example: `WOWWOW` $d_2(2) = 5$, $d_2(3) = 3$, $d_2(4) = 3$, $d_2(5) = 3$

Boyer-Moore Algorithm

After matching successfully $0 < k < m$ characters, the algorithm shifts the pattern right by

$$d = \max\{d_1, d_2\}$$

where $d_1 = \max\{t_1(c) - k, 1\}$ is bad-symbol shift

$d_2(k)$ is good-suffix shift

Example: Find pattern `AT_THAT` in

`WHICH_FINALLY_HALTS. _ _ AT_THAT`

Boyer-Moore Algorithm (cont.)

Step 1 Fill in the bad-symbol shift table

Step 2 Fill in the good-suffix shift table

Step 3 Align the pattern against the beginning of the text

Step 4 Repeat until a matching substring is found or text ends:

Compare the corresponding characters right to left.

If no characters match, retrieve entry $t_1(c)$ from the bad-symbol table for the text's character c causing the mismatch and shift the pattern to the right by $t_1(c)$.
If $0 < k < m$ characters are matched, retrieve entry $t_1(c)$ from the bad-symbol table for the text's character c causing the mismatch and entry $d_2(k)$ from the good-suffix table and shift the pattern to the right by

$$d = \max\{d_1, d_2\}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

Example of Boyer-Moore alg. application

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

B E S S _ K N E W _ A B O U T _ B A O B A B S

B A O B A B

$d_1 = t_1(K) = 6$ B A O B A B

$d_1 = t_1(\_)-2 = 4$

$d_2(2) = 5$

max{d1,d2}

B A O B A B

$d_1 = t_1(\_)-1 = 5$

$d_2(1) = 2$

B A O B A B (success)

k	pattern	d_2
1	BAO**B**A**B**	2
2	**B**AOB**AB**	5
3	**B**AO**BAB**	5
4	**B**A**OBAB**	5
5	**BAOBAB**	5

KMP

- Knuth-Morris-Pratt (KMP)
- Cormen, Leiserson, Rivest & Stein, pg 1002-1013

REF: string search comparision

- The Exact String Matching Problem: a Comprehensive Experimental Evaluation
- Simone Faro, Thierry Lecroq
- (Submitted on 12 Dec 2010)
- http://arxiv.org/abs/1012.2547

- This paper addresses the online exact string matching problem which consists in finding all occurrences of a given pattern p in a text t. It is an extensively studied problem in computer science, mainly due to its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, data compression, information retrieval, computational biology and chemistry. Since 1970 more than 80 string matching algorithms have been proposed, and more than 50% of them in the last ten years. In this note we present a comprehensive list of all string matching algorithms and present experimental results in order to compare them from a practical point of view. From our experimental evaluation it turns out that the performance of the algorithms are quite different for different alphabet sizes and pattern length.

Recent work

- Domenico Cantone, Simone Faro, Emanuele Giaquinta: Adapting Boyer-Moore-like Algorithms for Searching Huffman Encoded Texts. Int. J. Found. Comput. Sci. 23(2): 343-356 (2012)

- In this paper we propose an efficient approach to the compressed string matching problem on Huffman encoded texts, based on the BOYER-MOORE strategy. Once a candidate valid shift has been located, a subsequent verification phase checks whether the shift is codeword aligned by taking advantage of the skeleton tree data structure. Our approach leads to algorithms that exhibit a sublinear behavior on the average, as shown by extensive experimentation.

)

Hashing

- A very efficient method for implementing a *dictionary*, i.e., a set with the operations:
  - find
  - insert
  - delete

- Based on representation-change and space-for-time tradeoff ideas

- Important applications:
  - symbol tables
  - databases (*extendible hashing*)

Hash tables and hash functions

The idea of *hashing* is to map keys of a given file of size n into a table of size m, called the *hash table*, by using a predefined function, called the *hash function*,

$$h: K \rightarrow \text{ location (cell) in the hash table}$$

Example: student records, key = SSN. Hash function:
$h(K) = K \bmod m$ where m is some integer (typically, prime)
If $m = 1000$, where is record with SSN= 314159265 stored?

Generally, a hash function should:

- be easy to compute
- distribute keys about evenly throughout the hash table

Collisions

If $h(K_1) = h(K_2)$, there is a *collision*

- Good hash functions result in fewer collisions but some collisions should be expected (*birthday paradox*)

- Two principal hashing schemes handle collisions differently:
  - *Open hashing*
    – each cell is a header of linked list of all keys hashed to it
  - *Closed hashing*
    - one key per cell
    - in case of collision, finds another cell by
      - *linear probing:* use next free bucket
      - *double hashing:* use second hash function to compute increment

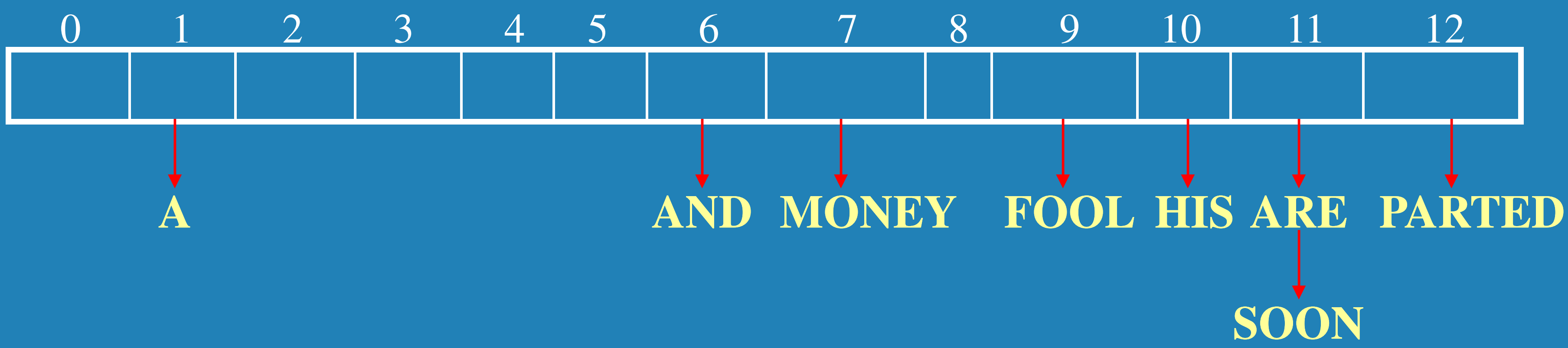
Open hashing (Separate chaining)

Keys are stored in linked lists <u>outside</u> a hash table whose elements serve as the lists' headers.

Example: A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

***h*(*K*) = sum of *K* 's letters' positions in the alphabet MOD 13**

Key	**A**	**FOOL**	**AND**	**HIS**	**MONEY**	**ARE**	**SOON**	**PARTED**
***h*(*K*)**	**1**	**9**	**6**	**10**	**7**	**11**	**11**	**12**



Example: Search for KID, h(KID) = 11

Open hashing (cont.)

- **If hash function distributes keys uniformly, average length of linked list will be $\alpha = n/m$. This ratio is called *load factor*.**
  - **file of size n into a table of size m**

- **Average number of probes in successful, S, and unsuccessful searches, U:**

$$S \approx 1+\alpha/2, \qquad U = \alpha$$

- **Load α is typically kept small (ideally, about 1)**

- **Open hashing still works if $n > m$**

- ***Efficiency from method´s approach + extra space***

Closed hashing (Open addressing)

Keys are stored inside a hash table.

Key	**A**	**FOOL**	**AND**	**HIS**	**MONEY**	**ARE**	**SOON**	**PARTED**
$h(K)$	**1**	**9**	**6**	**10**	**7**	**11**	**11**	**12**

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								**FOOL**			
	A					**AND**			**FOOL**			
	A					**AND**			**FOOL**	**HIS**		
	A					**AND**	**MONEY**		**FOOL**	**HIS**		
	A					**AND**	**MONEY**		**FOOL**	**HIS**	**ARE**	
	A					**AND**	**MONEY**		**FOOL**	**HIS**	**ARE**	**SOON**
PARTED	**A**					**AND**	**MONEY**		**FOOL**	**HIS**	**ARE**	**SOON**

Closed hashing (cont.)

- Does not work if $n > m$
  - file of size n into a table of size m
- Avoids pointers
- Deletions are *not* straightforward
- Number of probes to find/insert/delete a key depends on load factor $\alpha = n/m$ (hash table density) and collision resolution strategy. For linear probing:

$$S = (½) (1+ 1/(1- \alpha)) \quad \text{and} \quad U = (½) (1+ 1/(1- \alpha)^2)$$

α	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

- As the table gets filled (α approaches 1), number of probes in linear probing increases dramatically -> clustering!!!

Double hashing

- Alleviating collision…

- **Double hashing**: use another hash function s(K) to determine a fixed increment for the probing sequence after a collision at location l=h(K)

- (l+s(k)) mod m, (l+2s(K)) mod m, …

  - m prime
  - Recommendations
    - Small tables: s(k) m-2-k mod (m-2) s(k) = 8- (k mod 8)
    - Large tables: s(k) = k mod 97 + 1

Recommendation

Faça uma implementação de uma

- Open hash table
- Close hash table

Que permita

- Inserção
- Eliminação de chaves
- Chaves: strings de até 12 chars

- Discuta o fator de carga de suas implementações

indexing schemes (e.g., B-trees)

- Index
- Idea: from 2-3 tree to B-Tree
- Example
- Height
- B* : redistribuição
- B+ : chaves nas folhas

Recommendation

Faça uma implementação de uma variação de Árvore B que permita

- Até 3 chaves por bloco
- Eliminação e eliminação de chaves
- Redistribuição de chaves
- Cópia das chaves nas folhas