

Informatics

The R language and system with programming examples Part 3

Claudio Sartori

Department of Computer Science and Engineering

claudio.sartori@unibo.it

<https://www.unibo.it/sitoweb/claudio.sartori/>

Determinant of a square matrix

- Definition well known from algebra
 - e.g. Wikipedia <https://en.wikipedia.org/wiki/Determinant>
- Laplace expansion theorem: given matrix A
 - cofactor
 - $A_{i,j} = (-1)^{i+j} * M_{ij}$
 - minor
 - submatrix obtained by elimination of row i and column j from A
 - for $n=1$ $\det(a) = a$
 - for $n>1$, for any k between 1 and n the expansion can be done along column or row k

$$\det(A) = \sum_{i=1}^n a_{ik} (-1)^{(i+k)} \det(M_{ik}) = \sum_{j=1}^n a_{kj} (-1)^{(k+j)} \det(M_{kj})$$

Determinant with Laplace formula: recursive solution for $n \geq 2$

- if $n = 2$ return $(a_{11} * a_{22}) - (a_{21} * a_{12})$
- choose expansion line, e.g. column 1
- initialize sign to $1^{(\text{expansion row/column} + 1)}$
- initialize d to 0
- repeat varying i from 1 to n
 - if a_{i1} different from 0
 - $d \leftarrow d + \text{sign} * a_{i1} * \text{determinant}(M_{i1})$
 - $\text{sign} \leftarrow -\text{sign}$
- return d

RREF and the determinant ($n \times n$ only)

- the determinant of the matrix in reduced form is either 1, if the rank is n , or 0
- the determinant of the matrix in reduced form is equal to the determinant of the original matrix multiplied by a **scale factor**
 - each row swap generates a change of sign in the scale
 - each scalar multiplication causes a multiplication of the scale with the inverse of the scalar

$$\det(m) = \det(\text{Rref}(m)) * \text{scale}(\text{Rref}(m))$$

Useful R functions

```
list.files(path = "<path>")
    pattern = "*.txt")
```

- produces a vector with the names of the files in the *path* directory that satisfy the *pattern*

```
rep.int(<integer number>,<number of repetitions>)
```

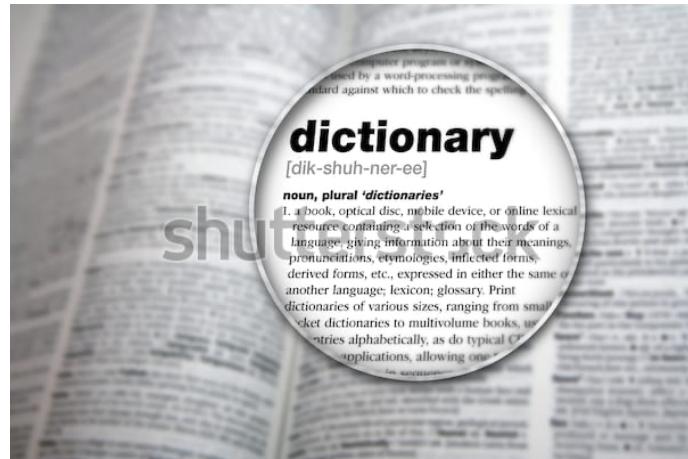
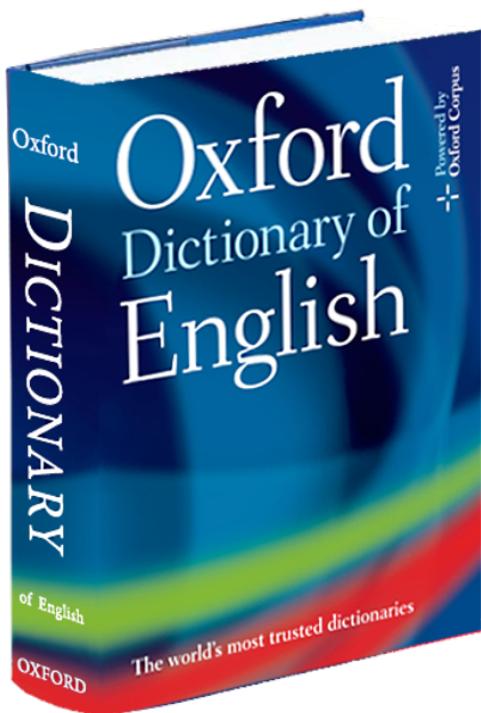
- produces a vector with a repeated integer, first argument is the integer, second is the repetition
- suitable for vector initialization

```
all(<logical vector>)
```

- produces TRUE if and only if **all** the elements of the logical vector are TRUE
- suitable for comparing two vectors

See the R algorithm and implementation in the R examples

Looking for a word in a dictionary



www.shutterstock.com · 392723320

Looking for a word in a dictionary

Algorithm – first attempt

1. You are looking for the word *thing*
2. Open the dictionary at some position
3. If the word is in the open pages stop and *return the position*
4. If the word is *smaller* than the smaller word in the open pages
5. then open the dictionary at some position preceding the current one
6. else open the dictionary at some position following the current one
7. go back to step 3

Looking for a word in a dictionary

Algorithm – first attempt

1. You are looking for the word *thing*
 2. Open the dictionary at some position
 3. If the word is in the open pages stop and *return the position*
 4. If *thing* is *smaller* than the smaller word in the open pages
 5. then open the dictionary at some position preceding the current one
 6. else open the dictionary at some position following the current one
 7. go back to step 3
- What's wrong with this algorithm?*

Looking for a word in a dictionary

Algorithm – second attempt

- You are looking for the word *thing*, the pages to search are *from the start to the end of the dictionary*
- Repeat while there are pages to search
 - Open the dictionary at some position inside the pages to search
 - If *thing* is in the open pages
 - stop and *return the position of the open pages*
 - If the word is *smaller* than the smaller word in the open pages
 - open the dictionary at some position *preceding* the current one: the pages to search are from the current start to the preceding position
 - *else*
 - open the dictionary at some position *following* the current one: the pages to search are from the following position to the current end
- *stop and return not found*

Looking for a word in a dictionary

Algorithm – second attempt

- You are looking for the word *thing*, the pages to search are *from the start to the end of the dictionary*
- Repeat while there are pages to search
 - Open the dictionary at some position inside the pages to search
 - If *thing* is in the open pages
 - stop and *return the position of the open pages*
 - If the word is *smaller* than the smaller word in the open pages
 - open the dictionary at some position *preceding* the current one
 - else
 - open the dictionary at some position *following* the current one
- stop and *return not found*

exit point: found

exit point: not found

How to test if *"there are pages to search"*

- Define the search area with two variables
 - **start** of the search area
 - **end** of the search area
 - after each **failed search** inside the open pages **update either** the start or the end of the search area

Looking for a word in a dictionary

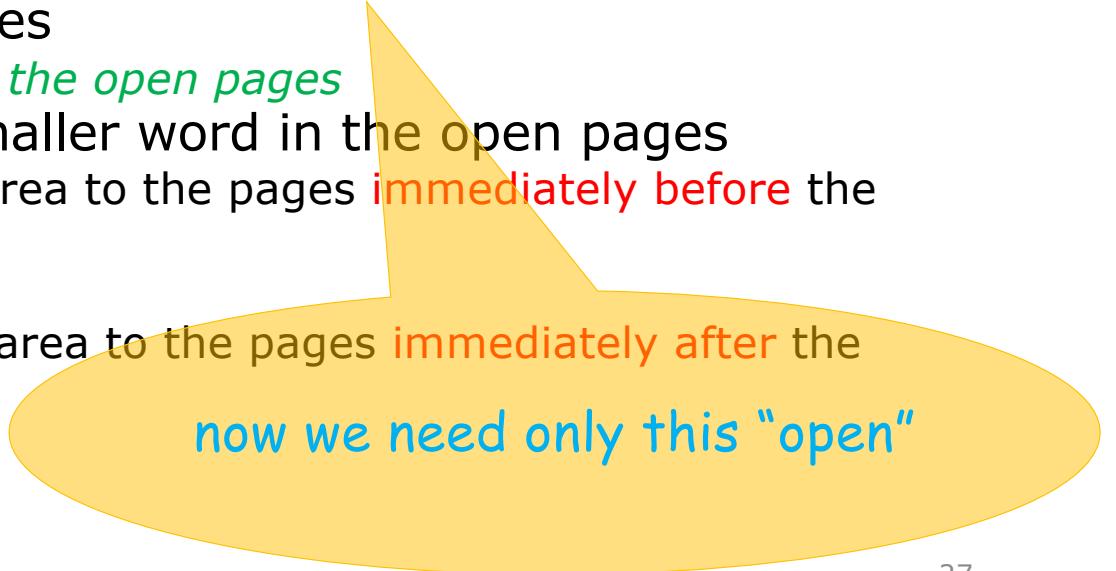
Algorithm – third attempt

- You are looking for the word *thing*, *the pages to search are from the start to the end of the dictionary*
- Repeat while *the set of pages to search is not empty*
 - Open the dictionary at some position inside the pages to search
 - If the word is in the open pages
 - stop and *return the position of the open pages*
 - If *thing* is *smaller* than the smaller word in the open pages
 - update the *end* of the search area to the pages *immediately before* the open pages
 - else
 - update the *start* of the search area to the pages *immediately after* the open pages
- stop and *return not found*

Looking for a word in a dictionary

Algorithm – third attempt

- You are looking for the word *thing*, *the pages to search are from the start to the end of the dictionary*
- Repeat while *the set of pages to search is not empty*
 - Open the dictionary at some position inside the pages to search
 - If the word is in the open pages
 - stop and *return the position of the open pages*
 - If *thing* is *smaller* than the smaller word in the open pages
 - update the *end* of the search area to the pages *immediately before* the open pages
 - else
 - update the *start* of the search area to the pages *immediately after* the open pages
- stop and *return not found*



now we need only this "open"

left part	center	right part
<i>start</i>	$(start+end)/2$	<i>end</i>

how can we specify now the test
the set of pages to search is not empty?

Looking for a word in a dictionary

Algorithm – fourth attempt

- You are looking for the word *thing*,
- set *start* to the *first page* of the dictionary
- set *end* to the *last page* of the dictionary
- Repeat while *start* \leq *end*
 - Open the dictionary at some position inside the pages to search
 - If the word is in the open pages
 - stop and *return the position of the open pages*
 - If *thing* is *smaller* than the smaller word in the open pages
 - update the *end* of the search area to the pages *immediately before* the open pages
 - else
 - update the *beginning* of the search area to the pages *immediately after* the open pages
- stop and *return not found*

Looking for a word in a dictionary

Algorithm – fourth attempt

- You are looking for the word *thing*,
- set *start* to the first page of the dictionary
- set *end* to the last page of the dictionary
- Repeat while *start* \leq *end*
 - Open the dictionary at some position between start and end
 - If the word is in the open pages
 - stop and return the position of the open pages
 - If *thing* is smaller than the smaller word in the open pages
 - update the end of the search area to the pages immediately before the open pages
 - else
 - update the beginning of the search area to the pages immediately after the open pages
- stop and return not found

how can we specify now
at some position between start and end?

We could use any point between start and end,
but the **best strategy** is using the **central point!**
In other words, the algorithm works with any
choice, but using the center it **works better**

Challenge: can you prove it?

Looking for a word in a dictionary

Algorithm – final

- You are looking for the word *thing*,
- set *start* to the first page of the dictionary
- set *end* to the last page of the dictionary
- Repeat while *start* \leq *end*
 - Open the dictionary in the position **center** = $(\text{start}+\text{end})/2$
 - If the word is in the open pages
 - stop and *return the position of the open pages*
 - If *thing* is *smaller* than the smaller word in the open pages
 - update the end of the search area to the pages immediately before the open pages
 - else
 - update the beginning of the search area to the pages immediately after the open pages
- stop and return not found

Searching a target in a sorted (e.g. alphabetized) list: binary search

- Sequential search
 - it can be done on a list, no requirement on sorting
- If the list is sorted the search can be done in a more efficient way

parameters:

`searchList`: vector containing the sorted values to be searched

`target`: value to search

use:

`start, end`: integers indicating the start and the ending indexes of the scope

`center`: index of the center of the current scope

result:

`found`: boolean

`position`: index where value is equal to target, if found, or where the target could be inserted keeping the list sorted, if not found

- set start, end to the first and last element of searchList
- repeat if $\text{start} \leq \text{end}$ (i.e. the scope is not empty)
 - compute center as $(\text{start}+\text{end})/2$ using **integer arithmetic**
 - if $\text{target} == \text{searchList}[\text{center}]$
 - found: exit
 - else
 - if $\text{target} < \text{searchList}[\text{center}]$
 - continue search with scope $\text{start}:\text{center}-1$
 - else
 - continue search with scope $\text{center}+1:\text{end}$
- if $\text{searchList}[\text{center}] < \text{target}$
 - the output position is $\text{center}+1$
 - else the output position is center
- output: not found

initialization

- set start, end to the first and last element of searchList
- repeat if start<=end (i.e. the scope is not empty)
 - compute center as (start+end)/2 using integer arithmetic
 - if target==searchList[center]
 - found: exit
 - else
 - if target < searchList[center]
 - continue search with scope start:center-1
 - else
 - continue search with scope center+1:end
- if searchList[center] < target
 - the output position is center+1
- else the output position is center
- output: not found

main loop

finalization

Why should we use *integer arithmetic*?
What happens if *start+end* is odd?

left part	center	right part
<i>start</i>	$(start+end)/2$	<i>end</i>

Binary search (1)

Target: Eddy

	searchList	
1	Ann	start
2	Eddy	
3	Fred	
4	Mary	center
5	Oliver	
6	Peter	
7	Terry	
8	Victoria	end

algo:

- set start, end to the first and last element of searchList
- repeat if start<=end
i.e. the scope is not empty
 - compute center as (start+end) / 2 using integer arithmetic
 - if target==searchList[center]
 - found: exit
 - else
 - if target < searchList[center]
 - continue search with scope start:center-1
 - else
 - continue search with scope center+1:end
 - if searchList[center] < target
 - the output position is center+1
 - else the output position is center
 - output: not found

Binary search (2)

Target: Eddy

	searchList	
1	Ann	start
2	Eddy	center
3	Fred	End
4	Mary	
5	Oliver	
6	Peter	
7	Terry	
8	Victoria	

algo:

- set start, end to the first and last element of searchList
- repeat if start<=end
i.e. the scope is not empty
 - compute center as (start+end) / 2 using integer arithmetics
 - if target==searchList[center]
 - found: exit
 - else
 - if target < searchList[center]
 - continue search with scope start:center-1
 - else
 - continue search with scope center+1:end
 - if searchList[center] < target
 - the output position is center+1
 - else the output position is center
 - output: not found

Binary search (3)

Target: Ada

	searchList	
1	Ann	Start
2	Eddy	
3	Fred	
4	Mary	Center
5	Oliver	
6	Peter	
7	Terry	
8	Victoria	End

algo:

- set start, end to the first and last element of searchList
- repeat if start<=end
i.e. the scope is not empty
 - compute center as (start+end) / 2 using integer arithmetics
 - if target==searchList[center]
 - found: exit
 - else
 - if target < searchList[center]
 - continue search with scope start:center-1
 - else
 - continue search with scope center+1:end
 - if searchList[center] < target
 - the output position is center+1
 - else the output position is center
 - output: not found

Binary search (4)

Target: Ada

	searchList	end
1	Ann	Start
2	Eddy	Center
3	Fred	End
4	Mary	
5	Oliver	
6	Peter	
7	Terry	
8	Victoria	

algo:

- set start, end to the first and last element of searchList
- repeat if start<=end
i.e. the scope is not empty
 - compute center as (start+end) / 2 using integer arithmetics
 - if target==searchList[center]
 - found: exit
 - else
 - if target < searchList[center]
 - continue search with scope start:center-1
 - else
 - continue search with scope center+1:end
- if searchList[center] < target
 - output position is center+1: exit
 - else output position is center: exit
- target not found: exit

Binary search

Target: Ada

	searchList	Start End Center
1	Ann	Start End Center
2	Eddy	
3	Fred	
4	Mary	
5	Oliver	
6	Peter	
7	Terry	
8	Victoria	

algo:

- set start, end to the first and last element of searchList
- repeat if start<=end
i.e. the scope is not empty
 - compute center as (start+end) / 2 using integer arithmetics
 - if target==searchList[center]
 - found: exit
 - else
 - if target < searchList[center]
 - continue search with scope start:center-1
 - else
 - continue search with scope center+1:end
- if searchList[center] < target
 - the output position is center+1
 - else the output position is center
- output: not found

Binary search

Target: Ada

	searchList	End
1	Ann	Start
2	Eddy	
3	Fred	
4	Mary	
5	Oliver	
6	Peter	
7	Terry	
8	Victoria	

algo:

- set start, end to the first and last element of searchList
- repeat if start<=end
i.e. the scope is not empty
 - compute center as (start+end) / 2 using integer arithmetics
 - if target==searchList[center]
 - found: exit
 - else
 - if target < searchList[center]
 - continue search with scope start:center-1
 - else
 - continue search with scope center+1:end
 - if searchList[center] < target
 - the output position is center+1
 - else the output position is center
 - output: not found

- If the list has n elements and we repeat the execution several times with different targets,
 - we keep track of the number of loop iterations in each experiment
- what is the maximum number of loop iterations we should expect?

Search area size along the iterations

Search area size – n=1023	Iteration	$\log_2(n)$ -iteration
1023	1	9
511	2	8
255	3	7
127	4	6
63	5	5
31	6	4
15	7	3
7	8	2
3	9	1
1	10	0

In general, the number of iteration is

$\text{ceiling}(\log_2(n+1))$

Binary search is a
divide-and-conquer
algorithm

Recap:

What is the purpose of this algorithm?

- input t, v
- set f to 0
- repeat varying i from 1 to length of v
 - if the position i of v is equal to t
 - set f to i and stop repeating
 - if f is zero
 - display "no""
 - else
 - display the value of f

Recap:

If the length of v is **n**, what is the
maximum number of repetitions?

- input t, v
- set f to 0
- repeat varying i from 1 to length of v
 - if the position i of v is equal to t
 - set f to i and stop repeating
 - if f is zero
 - display "no"
 - else
 - display the value of f

Recap:

If the length of v is **n**, what is the
minimum number of repetitions?

- input t, v
- set f to 0
- repeat varying i from 1 to length of v
 - if the position i of v is equal to t
 - set f to i and stop repeating
 - if f is zero
 - display "no"
 - else
 - display the value of f

Recap:

If the length of v is **n**, what is the average number of repetitions?

- input t, v
- set f to 0
- repeat varying i from 1 to length of v
 - if the position i of v is equal to t
 - set f to i and stop repeating
 - if f is zero
 - display "no"
 - else
 - display the value of f



**Continue to module 8:
Computational complexity**