Universidade Nova de Lisboa
OMNIS CIVITAS CONTRA SE DIVISA NON STABIT
Faculdade de Ciências e Tecnologia

Optimised Search in Vectors; Structures

Pedro Barahona
DI/FCT/UNL

Métodos Computacionais
1st Semester 2016/2017

Optimised Search in Vectors

- Insert Sort and Bubble Sort with a complexity **O(n^2)**, are useful to sort "small" vectors, but larger vectors require better algorithms.
- A useful strategy often used to solve complex problems is to divide them into smaller and simpler problems, and combine the solutions of the simpler problems to obtain the overall solution.
- This strategy, known as ***divide-and-conquer*** principle, is followed by several advanced sorting algorithms, namely Merge Sort and Quick Sort.
- This principle allows not only a simple (***recursive***) specification, but usually leads to a better complexity. In case of these algorithms, it leads to asimptotical complexity of **O(n • ln(n)).**

Optimised Search in Vectors

- This divide-and-conquer principle is implemented differently in these algorithms.

Merge Sort:

- Divide the vector in two sub-vectors.
- Sort both the sub-vectors.
- ***Merge*** their solutions, taking advantage of having them already sorted.

QuickSort:

- Get a pivot.
- Divide the vector into two sub-vectors, composed of all the values smaller and larger than the pivot.
- Sort these two sub-vectors.
- ***Append*** their solutions

Merge Sort

- As any recursive algorithm, the recursive function that implements it checks whether the recursion should stop, i.e. the problem is sufficiently simple to be solved directly.
- Here, we stop when the vector has length 1, in which case it is already sorted.
- Otherwise the function calls itself to obtain the sorted versions of the Left and Right sub-vectors, and merges them.

```
function S = merge_sort(V);
   n = length(V);
   if n > 1
      mid = floor((n+1)/2);
      L = merge_sort(V(1:mid));
      R = merge_sort(V(mid+1:end);
      S = merge(L, R)
   else
      S = V
endfunction
```

Merge Sort

- Merging two sorted lists is straightforward, and can also be implemented recursively.
- The recursion stops when one of the sub-lists is empty, in which case the merged vector is the non-empty sub-vector.
- Otherwise, the smaller of the two initial values is the initial value of the solution, and the rest is obtained by merging the remaining vector with the other sub-vector.

```
function S = merge(L,R);
   if length(L) == 0
      S = R;
   elseif length(R) == 0
      S = L;
   elseif L(1) < R(1)
      S = [L(1),merge(L(2:end),R)];
   else % R(1) < L(1)
      S = [R(1),merge(L,R(2:end))];
   endif;
endfunction
```

Merge Sort – Complexity

- The asymptotical complexity of Merge Sort can be obtained as follows (assuming a vector with a size $n = 2^k$; the analysis of other sizes require some rounding that does not affect the asymptotical complexity).
- At each bipartition, it is necessary to merge two vectors. The number of operations is the sum of the sizes of the two vectors.
  - At level 1, there are 2^1 vectors of size $n/2^1 \rightarrow$ n operations
  - At level 2, there are 2^2 vectors of size $n/2^2 \rightarrow$ n operations
  - …
  - At level k, there are 2^k vectors of size $n/2^k \rightarrow$ n operations
- Summing the cost of the above operations, we have n + n + … + n, with k terms,

$$T(n) \approx k*n$$

- At level k, the sub-vectors have size 1. Hence $1 = n/2^k$ and therefore

$$T(n) \approx n* \log_2(n)$$

- So the asymptotical complexity of Merge Sort is **O(n In(n))**

Quick Sort

- Quick Sort also adopts the divide-and-conquer principle, but in a different way. The main steps of the function are the following:

1. An element of the vector, **p**, is selected for pivot. Typically, this is the element that occurs in the **mid** position of the vector (but this is not necessarily so).
2. Then the vector is swept with two indices starting at both ends of the vector range:
   – Index **i**, starts at **1**, and increases during the sweep
   – Index **j**, starts at **n**, and decreases during the sweep
3. The sweep ends when both indices **i** and **j** take the same value. At this point,
   – **V(i) = p**;
   – all values in positions less than **i** are less than **p**; and
   – all values in positions greater than **i** are greater than **p**.
4. Then, all that is needed is to sort the lower and upper sub-vectors, and append them together
   – The sorting of the sub-vectors can of course be done through recursive calls to quick sort.

Quick Sort

- The basic structure of the `quick_sort` function are shown below, where the recursive calls are made to sort the two sub-vectors.

```
function S = quick_sort(V);
   n = length(V);
   p = V(floor((1+n)/2));
   i = 1
   j = n;
      % sweep and obtain an index k such that all
      % values before (after) k are respectively
      % less (greater) than V(k) = p.
   L = quick_sort(V(1:i-1));
   G = quick_sort(V(i+1:n));
   S = [L, v(k), G];
endfunction
```

- Of course, it is necessary that after the block in comments, that performs the sweep, all elements of V with indices less/greater than I are less/greater than p.
- We can now discuss the implementation of this sweeping block.

Quick Sort

- The sweeping, starts with **i = 1** and **j = n**, and proceeds while **i < j** as follows
  1. While **V(i) < p**, index **i** increases; while **V(j) > p**, index **j** decreases.
  2. If **i < j**, then **V(i) ≥ p** and **V(j) ≤ p**, and then values **V(i)** and **V(j)** are swapped, and the indices updated by 1.

```
...
while i < j
   while V(i) < p    i = i+1; endwhile
   while V(j) > p    j = j-1; endwhile
   if i < j
      aux = V(i); V(j) = V(i); V(i) = aux;
      i = i+1; j = j-1;
   endif;
endwhile;
...
```

- The outer while loop stops when **i = j** . Moreover, as intended, all values lower/greater than **p** either were initially, or were moved for positions less/greater than **i**.
- Note that in this process **p** is moved to position **i**. Why?

Quick Sort – Complexity

- The asymptotical complexity of Quick Sort **depends on the pivot** that is chosen. In average, we may assume that the pivot is such that divides a vector of size n into 2 sub-vectors of size n/2.
- As with the merge sort, we assume that the initial vector has size $n = 2^k$ (for other sizes the required rounding that does not change the asymptotical complexity).
- Now at each level the whole vector is swept, before the recursive calls to the 2 sub-vectors are made. Hence, the number of accesses is
  - At level 1, sweeping of 1 vector of size n $\rightarrow$ n accesses
  - At level 2, sweeping of 2 vector of size n/2 $\rightarrow$ n accesses
  - At level k, sweeping of 2^k vector of size $n/2^k$ $\rightarrow$ n accesses
- Again the cost of the above operations, is n + n + … + n, with k terms,

$$T(n) \approx k*n$$

- At level k, the sub-vectors have size 1. Hence 1 = n/2k and therefore

$$T(n) \approx n* \log_2(n)$$

- So the asymptotical complexity of Quick Sort is **O(n In(n))**

Structures

- Arrays (vectors, matrices, or multi-arrays) are very convenient structures to organize numerical information, since each "cell" should contain a number.
- In many cases, information is not only numeric, e.g. it includes text (we do not consider other types of information, such as visual or sound or video).
- Moreover, the data is organized in a mixed way, combining text and numerical information in "records".
- Take for example the information about the employees of a certain company. For each employee we may consider:
  - **id** – integer, representing a unique identification number in the company
  - **date** – text, in format YYYY-MM-DD, representing the date of employment
  - **name** – text, with the name of the employee
  - **salary** – real number, representing the monthly salary of the employee
- Although complex information is better maintained in a database, in simple applications, this heterogeneous information may be organized in a record, or in the MATLAB notation (borrowed from C) in a **structure**.

Structures

- A structure is similar to a vector with two main differences.
  - Different positions may contains different types of data; and
  - Positions are identified by field names, following the name after a “.”.
- Example: An employee, **emp**, may be represented by the structure

```
>> emp.name = "Rui Silva";
>> emp.id = 35;
>> emp.date = "2011-10-23";
>> emp.salary = 1654.30;
>> emp
emp =
  scalar structure containing the fields:
    id =  35
    name = Rui Silva
    date = 2011-10-23
    salary =  1654.3
```

Structure Arrays

- As shown a structure is initialised by simply assigning values of its fields.
  - Different positions may contains different types of data; and
  - Positions are identified by field names, following the name after a “.”.
- Conversely, the data of the fields may be accessed “individually”

```
>> x = emp.name
x = Rui Silva
>>
```

- Most applications require to maintain several records in a “table”, i.e. a set of records of the same type, organised as a “vector”, i.e. each with its index.
- In MATLAB, this table data structure is available as a **structure arr**ay. A table of employees may contain several records, each with the information of an employee.
- MATLAB is very “permissive” regarding these structure arrays. It is good practice that all records have the same fields, and that these are defined before the structure array is filled with information.

Structure Arrays

- In MATLAB a structure array, can be obtained by simply assigning the values of the fields of the different structures that compose the structured array.
- For example a table with 2 employees may be obtained as follows

```
>> table(1) = emp; % Rui Silva
>> table(2).name = "Ana Matos";
>> table(2).id = 54;
>> table(2).date = "2013-09-02";
>> table(2).salary = 895.60;
>> table
table =  1x2 struct array containing the fields:
    id
    name
    date
    salary
>>
```

- Notice that the values are usually not displayed in the terminal, only the composition of the structure array is displayed.

Structure Arrays

- Structure arrays share many properties of “usual” arrays, and in particular
  - Their size is available through function **length**;
    - In fact a structured array has 2 dimensions, but is similar to a row vector, whose elements are structures
  - Their ranges always start in index 1;
  - Structure sub-arrays may be obtained by a projection operation, and are composed of the structures whose indices are selected
    - Projection on the fields is not directly available, and must be programmed if needed.
  - As any vector, elements may be deleted by assigning them the empty (`[]`) value.
    - The remaining elements are shifted “downwards” so that no “holes” are created in the structured array

Structure Arrays

Examples:

```
>> E
E =  1x10 struct array containing the fields:
    id     name     date     salary
>> F = E;
>> F(5) = [];
F =  1x9 struct array containing the fields:
    id     name     date     salary
>> F(5) == E(6)
ans = 1
>> n = ndims(E), r = rows(E), c = columns(F)
n = 2
r = 1
c = 9
>> G = E(5:7)
G =  1x3 struct array containing the fields:
    id     name     date     salary
>> length(G)
ans = 3
>> F(5) == G(2)
ans = 1
```