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

Search in Vectors

Pedro Barahona
DI/FCT/UNL
Métodos Computacionais
1st Semester 2016/2017

Search

- A key goal in Informatics is to find the information that is needed. And to do so, one needs some type of **search**.
- In this lecture we will focus on finding information in a (numerical) vector. Most of the techniques discussed may later be adapted to other data structures.
- In a vector V, of length n, there are two types of searches that are basic:
  - Given an index i, find the value v = V(i), if any;
  - Given a value v, find the index i such that v = V(i), if any;
- Of course, in a vector these two types of search have completely different complexity properties.
  - In the first case, all that is needed is to guarantee that index i is valid, i.e. $i \leq n$.
    - In a vector this requires a **single** access to the vector.
  - In the second case, the different values of the vector must be considered.
    - In the worst case, all n values must be considered, requiring **n** accesses.

Sequential earch

- A general procedure to search for a value in a vector, sequentially, checks the values one by one.
  - If it finds the value it returns the value of the index where x occurs.
  - Otherwise it stops after checking the last element of V.
- This sequential search can be specified by the following MATLAB code

```
function find (x, V);
% returns the index p of vector V where x is to
% be found; if V does not contain x, then the
% function returns p = 0
   p = 0;
   i = 1;
   while i < n && p == 0
      if V(i) == x
         p = i;
      else
         i = i + 1;
      endif
   endwhile
endfunction
```

Search

- Given the speed of current processors, in many cases, it is acceptable to pay this cost. But if one is interested in doing several searches in a very large vector, it is convenient to adopt a better policy.
- Clearly, a better policy is possible if the information is adequately maintained (stored) so as to ease the searching task.
- In the case of a vector, searching is much easier if the vector is sorted:
  a) Even if the search is sequential, as before, there is now the possibility to give up earlier, if one passes the value of interest;
     - In average, this reduces the number of accesses to **n/2**.
  b) A better search policy may be quite effective
     - A divide and conquer policy may bound the number of accesses to $\mathbf{log_2(n)}$.

Bipartite Search

- If the vector **S** is sorted, than the search for a value **x** can be delimited between two indices, **lo** and **up**, assuming that if x occurs in **S** then it should be between these two indices. Of course the initial indices should be 1 and n (the length of **S**).
- As before, the result of the function, p, is either the index where x is or 0 if x does not occur in S. The search proceeds until either:
  - The value is found (i.e. **p > 0**); or
  - The interval is empty (i.e. **lo > up**) .

```
function p = find_s_ite(x,V);
   lo = 1;
   up = length(V);
   p = 0;
   while lo <= up && p == 0
      ...
   endwhile
endfunction
```

Bipartite Search

- In each search step, the value in the mid point of the interval, **V(mid),** is compared with **x**. Then
  - If **x == V(mid)** the value is found, and **p** becomes the index that is returned. Moreover, since **p** is positive the loop is never entered anymore.
  - If **x > V(mid**), then x, if it occurs in V, must be between indices mid+1 and up.
  - If **x < V(mid)**, then x, if it occurs in V, must be between indices lo and mid-1.

```
while lo < up && p == 0
   mid = floor((lo+up)/2);
   if x == V(mid)
      p = mid;
   elseif x > V(mid)
      lo = mid + 1;     % up remains the same
   else % x < V(mid)
      up = mid - 1;     % lo remains the same
   endif
endwhile
```

Bipartite Search

- An alternative way of specifying the search is through a **recursive** function, i.e. a a function that calls **itself** during its execution.
- Of course, to guarantee that the function finishes execution, there must be a **termination** condition that is tested before the recursive call.
- In this case, this condition is as before: either the value was found or the interval of interest is empty.
- Hence the recursive function is specified with parameters **lo** and **up** , representing the limits of the interval of interest, as shown below:

```
function p = find_s(x, V);
% returns the index p of a sorted vector S where x
% is to be found; if V does not contain x, then the
% function returns p = 0
   n = length(V);
   p = find_between(x, S, 1, n);
endfunction
```

Bipartite Search

- As with any recursive function, the stopping condition of the recursion must be tested before calling itself any further.
- In fact this stopping condition is divided in two parts. From the outset, only the length of the interval can be checked, as shown.

```
function p = find_between(x, S, lo, op);
   if lo > up
      mid = floor((lo+up)/2);
      ...
   else
      p = 0
   endif
endfunction
```

- The second part of the condition, whether the value occurs in the vector, is dealt with inside the if block.

Bipartite Search

```
function p = find_between(x, S, lo, op);
   ...
   if lo < up
      mid = floor((lo+up)/2);
      if x == V(mid)
         p = mid
      elseif x > V(mid)
         p = find_between(x, S, i, m-1);
      else % x < V(mid)
         p = find_between(x, S, i+1, m);
      endif
   endif
endfunction
```

- It can be checked that, as intended, a recursive call is only made if
  - the interval is not empty (i.e. **lo < up**) ;
  - The value has not been found (**not x== V(mid)**)
- In particular, a recursive call of the function is always made with a narrower interval, which guarantees **termination**.

Bipartite Search

- This type of binary search, that is followed by both versions of the function, requires **$\log_2(n)$** accesses to the vector.
  - Access 0 is made to an interval of size n; 0 → n
  - Access 1 is made to an interval of size n/2; 1 → $n/2^1$
  - Access 2 is made to an interval of size n/4; 2 → $n/2^2$
  - …
  - Access k is made to an interval of size 1; k → $n/2^k$
- Since in every call to the function the interval is halved, than the number of calls k, until the interval has size 1 is such that $n/2^k = 1$, i.e.

$$\mathbf{k = \log_2(n)}$$

- In this analysis we do not take into account the rounding that is used to obtain integer indices, but for large values of **n** this does not make much of a difference.
- Moreover the base of the logarithm that is chosen is not a big issue, since the ratio between logarithms of diffeernt bases is a constant. In particular,

$$\mathbf{\log_2(n) = \ln(n) / \ln(2)}$$

Bipartite Search

- More formally, we typically specify the **asymptotic** complexity of an algorithm by the **O** notation.
- Given a problem of **size** n, an algorithm is said to be of temporal complexity **O(f(n))** if we have

$$\lim_{n \to \infty} t(n) \leq k*f(n)$$

where

  - **k** is a constant.
  - **t(n)** is number of "simple" operations, executed in constant time, it requires;
  - the size **n** has a more formal definition but can be regarded as the number of scalar values that are needed to define the problem. In case of a vector, is the number of its elements.
- Hence finding an elements by sequential search in an array of size n has complexity **O(n).**
- Finding it on a sorted array of the same size is **O(ln(n)).**

Sorting

- Of course, sorting a vector takes time! If the number of required searches is small, it might not pay off to spend a lot of time in the sorting, to save a small time in the search. But for large values of n, the **speed up** in the search can be very large.
- For n = 10^{10}, the size of the population of a middle sized country as Portugal, rather than $5*10^9$ accesses we only need about $\ln(10^{10}) \approx 33$ accesses, a speed up of about 10^8 in each search!
- If each access takes 1 μsec, than a bipartite search is done in 33μsec, whereas sequential search would require $5*10^9$ μsec, i.e. 5000 sec, which is more than 1 hour!!!
- Of course, the data structure must be sorted, and this takes time, but often it can be done at idle times (i.e. at night) so that the accesses can be done very efficiently during normal office hours (i.e. daytime).

Vector Sort

- So sorting is a possibly one of the most used and studied operations in Information Systems. Given its relevance, a number of algorithms have been proposed for sorting, and in particular for sorting vectors. Among them we can list:
  - **Insert Sort**
  - **Bubble sort**
  - **Quick Sort**
  - **Bucket Sort**
  - **Heap Sort**
  - **Quick Sort**
- We will next study some of them. The simplest ones have complexity $O(n^2)$, whereas the best have complexity $O(n \bullet \ln(n))$. For small values of n both are acceptable, but for larger ones, the best algorithms are needed.

For $n = 10^3$ and 1 op = 1 ns, we have

- $n^2 \approx 10^6$ ns ≈ **1 msec**
- $n \bullet \ln(n) \approx 7000$ ns ≈ **7 μsec**

For $n = 10^{10}$ we have

- $n^2 \approx 10^{20}$ ns ≈ **132 years**
- $n \bullet \ln(n) \approx 2.3 \bullet 10^{11}$ ns (**4 min**)

Insert Sort

- Insert sort is an algorithm that progressively sorts the beginning of the vector.
- At each step, it assumes that a **prefix of size k** of the vector, i.e. the first k elements of the vector, are already sorted.
- Then it proceeds by inserting the k+1th element in this prefix, to obtain a new prefix of size k+1.
- Of course this operation must be executed n-1 times
  - Starting with a prefix of size 1 and inserting the 2nd element in it
  - Continuing with a prefix of size 2 and inserting the 3rd element in it
  - …
  - Ending with a prefix of size n-1 and inserting the nth element in it.
- Of course, at the end of this process, the whole vector is sorted.

Insert Sort

- Insert sort can be illustrated with a simple example

4	2	5	1	3

  - Insert the 2nd into the prefix of size 1

4	2	5	1	3
2	4	5	1	3

  - Insert the 3rd into the prefix of size 2

2	4	5	1	3
5	2	4	1	3
2	5	4	1	3
2	4	5	1	3

  - Insert the 4th into the prefix of size 3

2	4	5	1	3
1	2	4	5	3

  - Insert the 5th into the prefix of size 4

1	2	4	5	3
3	1	2	4	5
1	3	2	4	5
1	2	3	4	5

Insert Sort

- The iterative version of the algorithm can be specified following the previous explanation.
- The algorithm initialises the sorted vector to be equal to the original vector.
- Then it executes a **for loop**, to insert the k^{th} element into the prefix of size k-1.
  - starting with k = 2; and
  - ending with k = n, the size of the vector

```
function S = insert_sort_ite(V)
   S = V;
   n = length(V);
   for k = 2:n
      % insert kth element in prefix of size k-1
   endfor;
endfunction;
```

Insert Sort

- The loop block inserts x, the k^{th} element of S, into the prefix of size k-1, by
  - Starting in position **i = 1**, comparing the values of **S(i)** with **x**, and advancing **i** while they are less than x;
  - If no value is larger than the k^{th} element, than the prefix of size k+1 is sorted;
  - Otherwise i becomes the position for x, after pushing forward all elements previously in positions i to k-1 (starting from the end ! - why?).

```
for k = 2:n
   x = S(k);
   i = 1;
   while i < k && S(i) < x   i = i+1;   endwhile;
   if i < k
      for j = k-1:-1:i
         S(j+1) = S(j);
      endfor;
      S(i) = x;
   endif;
endfor
```

Insert Sort - Complexity

- The complexity of Insert Sort can be assessed, looking at the structure of the algorithm.
- In each loop, the element in the k^{th} position, with value **x**, is inserted in its position in a prefix of size k-1, where k varies from 2 to n.
- This insertion requires x to be compared c times until it finds a larger element in position c. Then the k-(c+1) elements must be shifted one position to allow x to be inserted in position c.
- All things considered, in each loop a total of c+(k-(c+1)) + 1 = k operations are executed.
  - Note these operations comparisons and assignments are not the same, but their timing difference is bound by a constant).
- For all the loops the number of operations is then **T(n) = 2+3+…+n**, and hence

$$T(n) = (n-2+1)*(2+n)/2 \approx n^2 / 2$$

- Hence the time complexity of Insert Sort is thus **$O(n^2)$**.

Bubble Sort

- Bubble sort is another very simple algorithm for sorting that is based in a simple idea: if neighbouring elements of the vector (**a bubble**) are in the wrong order they should be swapped.
- Of course this swap operation has to be repeated several times.
  - Sweeping the vector with a bubble from start to end, it is easy to see that in the end, the largest element of the vector is in the last position.
  - Sweeping it again, the 2nd largest element is in the 2nd last position.
  - Sweeping n times all elements of the vector are in their right order.
- In fact:
  - Only n-1 sweeps are needed: if all but the smallest element are sorted in the last positions, the smallest element is correctly placed in the first position;
  - Since the largest elements of the vector are being placed in their right order, i.e. in the end of the vector, the successive sweeps may be executed in successively smaller prefixes of the vector.

Insert Sort

- Bubble sort can be illustrated with the same simple example

4	2	5	1	3

  - 1st sweep, placing the largest element

4	2	5	1	3
2	4	5	1	3
2	4	5	1	3
2	4	1	5	3
2	4	1	3	5

  - 2nd sweep, placing the 2nd largest element

2	4	1	3	5
2	4	1	3	5
2	1	4	3	5
2	1	3	4	5

  - 3rd sweep, placing the 3rd largest element

2	1	3	4	5
1	2	3	4	5
1	2	3	4	5

  - 4th sweep, placing the 4th largest element

1	2	3	4	5
1	2	3	4	5

  - A 5th sweep is not needed

1	2	3	4	5

Bubble Sort

- The iterative version of the algorithm can be specified following the previous explanation.
- The algorithm performs n-1 sweeps of the bubble, each sweep ending in successively smaller positions of the bubble, starting n position n and ending in position 2.

```
function S = bubble_sort_ite(V)
   S = V;
   n = length(V);
   for k = n:-1:2
      % sweeps a bubble from positions 1 to k
   endfor;
endfunction;
```

Bubble Sort

- In each sweep, the bubble progressively advances, its first element starting in position 1 and ending in position k-1.
- In each position of the bubble, its elements are compared and if needed they are swapped.

```
function S = bubble_sort_ite(V)
   ...
      % sweeping the vector with a bubble
      for i = 1:k-1
         if S(i) > S(i+1)
            x = S(i);
            S(i) = S(i+1);
            S(i+1) = x;
         endif;
      endfor;
   ...
endfunction;
```

Bubble Sort - Complexity

- The complexity of Bubble Sort can be easily assessed, looking at the structure of the algorithm with 2 nested loops.
- The body of the second loop, regarding the advance of the bubble in each sweep, is executed a variable number of times: n-1 in the 1st sweep, n-2 in the 2nd, …, 1 in the n-1th, i.e.

$$T(n) = (n-1) + (n-2) + \ldots + 1 = n-1 * (1+ n-1)/2 \approx n^2/2$$

```
...
for k = n:-1:2
   for i = 1:k-1
      ...
   endfor;
endfor
...
```

- Hence the time complexity of bubble sort is **O(n^2)**.