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

Search in Vectors; Sorting

Pedro Barahona
DI/FCT/UNL
Métodos Computacionais
1st Semestre 2018/2019

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)**;
  - 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 ≤ 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 Search

- 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 p = 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;                  % value not found yet
   n = length(V)
   for i = 1:n
      if V(i) == x
         p = i;
         return;
      end
   end
end
```

Sequential Search

- The same procedure can be implemented without interrupting the loop, replacing the FOR loop by a WHILE loop, and maintaining a Boolean variable to indicate whether the element was already found.

```
function p = 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;                          % value not found yet
   found = false;
   i = 1;
   while i <= n && !found
      if V(i) == x
         p = i;
         found = true;
      else
         i = i + 1;
      end
   end
end
```

Search

- In general, the algorithms are classified according to their complexity on the size n of the input data.
- We say that an algorithm has worst case time complexity of **O(f(n))** if the number of basic operations it requires is asymptotically bound by **f(n)**, i.e.

$$\lim_{n\to\infty} \frac{f(n)}{n} = k$$

where k is some constant.

- Hence, the above algorithms for searching an element in a vector of size n have **linear complexity**, i.e. complexity

O(n)

Search

- Given the speed of current processors, in many cases, it is acceptable to pay this cost, i.e. to spend at most **n** accesses to find an element.
- But if one is interested in doing several searches in a very large vector, it is convenient to adopt a better policy.
- However, a better policy is only 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**:
  - Even if the search is sequential, as before, there is now the possibility to give up the search earlier, if one “passes” the value of interest;
    - In average, this reduces the number of accesses to **n/2**.
  - A better search policy may be quite effective
    - In fact, a divide and conquer policy may bound the number of accesses to **log(n)**.

Improved Sequential Search

- If the vector **S** is sorted, than the search for a value **x** can be interrupted before the end of the vector. Assuming S is sorted in increasing order, the previous algorithm (with the FOR loop) can be adapted to

```
function p = find(x, S);
% returns the index p of a vector S, sorted in
% increasing order, where x is to be found;
% if S does not contain x, it returns p = 0
   p = 0;                  % value not found yet
   n = length(V)
   for i = 1:n
      if V(i) == x          % x found in position p
         p = i; return;
      elseif S(i) > x       % x is not in S
         return             % with p = 0
      end
   end
end
```

- Although the number of accesses is decreased in average to n/2 the complexity of the algorithm is still **O(n)**.

Bipartite Search

- However, we may still improve this complexity when the vector **S** is sorted.
- To do so, we will define a function, that searches an element in the vector between indices **lo** and **up**, where **1 ≤ lo ≤ up ≤ n**.
- The algorithm to implement the search can be informally defined as follows
  - Look at the index **m**, in the middle of the range of interest, i.e. **m = (lo+up)/2**
    - If **S(m) = x**, the element was found, so return position m
    - If **S(m) > x**, **x** must be searched before the mid point, i.e. in range **lo..m-1**
    - If **S(m) < x**, **x** must be searched after the mid point, i.e. in range **m+1..up**
    - Eventually, the range is null (i.e. the lower bound is larger than the upper bound, in which case the element is not present in the vector, and the procedure returns 0.

Bipartite Search

- The above algorithm is easily implemented with a recursive function.

```
function p = find_between(x, S, lo, up);
% returns the index p of a vector S, sorted in
% increasing order, if x is to be found between
% indices lo and up. Otherwise, it returns p = 0
   p = 0;
   if lo <= up
      m = round((lo+up))/2;
      if S(m) == x
         p = m; return;
      elseif x < S(m)
         p = find_between(x,S,lo, m-1)
      else
         p = find_between(x,S,m+1, up)
      end
   end
end
```

- Note that the test for the termination of recursion (**lo <= up**) is done before the recursive calls.

Bipartite Search

- The above function can be used directly or, if one wants to maintain the interface with a find/2 function, this function may be rewritten as

```
function p = find(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);
end
```

- In any case, we should analyse the complexity of the algorithms implemented in function **find_between/4**, and this is done next.

Bipartite Search

- To analyse the complexity of this binary search, made to a vector S of size n, we note the size of the ranges for consecutive calls.
- In particular, we note that in each call, the size of the range is reduced to half, as we check the element in the middle of the input range. Hence
  - Call 1 is made to a range of size n — **$1 \rightarrow n$**
  - Call 2 is made to a range of size n/2 — **$2 \rightarrow n/2$**
  - Call 3 is made to a range of size n/4 — **$3 \rightarrow n/2^2$**
  - …
- In the limit, i.e. the element has not been found, the k^{th} call is made to a vector of size 1, i.e.
  - Call k is made to a range of size **$n/2^{k-1}$** — **$k \rightarrow n/2^{k-1} = 1$**
- Hence we have

$$\mathbf{k-1 = \log_2(n)}$$

and hence the algorithm has a logarithmic complexity, i.e. its complexity is

$$\mathbf{O(\log(n))}$$

Bipartite Search

- 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 different bases is a constant. In particular,

$$\log_2(n) = \ln(n) / \ln(2)$$

- What is significant is the decrease in complexity of the algorithms for searching, namely for very large vectors, as shown in the next table

n	$\log_{10}(n)$	$\log_2(n)$	ln(n)
10	1.000	3.322	2.303
100	2.000	6.644	4.605
1 000	3.000	9.966	6.908
10 000	4.000	13.288	9.210
100000	5.000	16.610	11.513
1 000 000	6.000	19.932	13.816
10 000 000	7.000	23.253	16.118
100 000 000	8.000	26.575	18.421
1 000 000 000	9.000	29.897	20.723
10 000 000 000	10.000	33.219	23.026

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 $\log_2(10^{10}) \approx 23$ accesses, a speed up of about 200 000 in each search!
- If each access takes 1 msec, than a bipartite search is done in 23 msec, whereas sequential search would require $5*10^9$ msec, 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

- Sorting is 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**
  - **Merge Sort**
  - **Quick Sort**
  - Bucket Sort
  - Heap 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**)

- **Note:** A good animation of sorting algorithms is available in youtube at https://www.youtube.com/watch?v=kPRA0W1kECg

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 $k+1^{th}$ element in this prefix, to obtain a new prefix of size k+1.
- This operation must thus 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-1, 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
   end;
end;
```

Insert Sort

- The loop block inserts x, the k^{th} element of S, into the prefix of size k-1, by
  - Finding p, the position where x should be inserted
  - Pushing forward the elements of the vector from index p to index k-1
    - Care must be taken not to write over the existing elements the pushing should be done from k-1 to k, then from k-2 to k-2, …, until p to p+1;
    - Of course, if p = k (i.e. the element remains in the same position) no pushing is done.

```
for k = 2:n
   x = S(k);
   ... % p is the position to insert x;
   for j = k-1:-1:p
      S(j+1) = S(j); % push elements forward
   end;
   S(p) = x;
end
```

Insert Sort

- The position p where to insert x is found by sweeping the vector starting from position 1, until the position is found.
- The position p to insert x = S(k) is thus either
  - The first position p (<k) where S(p) > x; or
  - position k, when all elements in positions 1..k-1 are less than x
    - In this last case, element S(k) = x remains in the same position k

```
i = 0;
found = false;
while !found
    i = i + 1;
    if i = k || S(i) > x
        p = i;
        found = true;
    end
end
```

Insert Sort - Complexity

- The complexity of Insert Sort can be assessed, looking at the structure of the algorithm.
- There are **n-1** loops, where in each loop, the element in the **k^{th}** position ($2 \leq k \leq n$), with value **x**, is inserted in a prefix of size **k-1**.
- In the worst case (when the prefix) is already sorted, this requires **k-1** comparisons.
- Summing for all the loop instances, the worst-case number of comparisons is

$$(2-1) + (3-1) + (4-1) + \ldots + (n-1) = 1 + 2 + \ldots + n-1 = (1+n-1)\,(n-1) / 2 \approx n^2/2$$

- In the worst case, there is one insertion in each of n-1 loop instances.
- Hence in the worst case, the algorithm requires **$n^2/2$** comparisons and **n-1** insertions.
- Assuming these operations take roughly the same time, the time complexity of insert sort is thus

$O(n^2)$.

Insert Sort – Recursive version

- A recursive version of the algorithm is possibly more understandable, as shown below
  - If the vector has zero or one element, it is already sorted;
  - Otherwise
    - sort the first n-1 elements to obtain a sorted prefix of n-1 elements, and
    - insert the last element in the sorted prefix

```
function S = insert_sort_rec(V)
% S is the sorted version of vector V
   n = length(V);
   if n <= 1
      S = V;
   else
      P = insert_sort_rec(V(1:n-1));
      S = insert_one(V(n), P);
   end;
end;
```

- **Note:** The time complexity of the recursive version is the same, but it requires several function calls, and auxiliary vectors, which makes execution slower.

Insert Sort – Recursive version

- Inserting **x** in a sorted vector **V** can also be defined recursively
  - If the element is less than the least element of the vector insert it at the head
  - Otherwise,
    - Keep the first element of the vector, V(1); and
    - Insert x in the rest of vector V; and
    - Append the resulting vector to V(1)

```
function S = insert_one(x,V)
% insert x in the right position of sorted vector V
   n = length(V);
   if n = 0
      S = [x]
   else
      if x < V(1)    % x is less than the least of V
         S = [x, V]
      else
         U = insert_one(x,V(2:n))
         S = [V(1), U]
      end;
   end;
end;
```

Bubble Sort

- Bubble sort is a very simple and popular algorithm for sorting that is based on 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.

Bubble 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(V)
% S is the sorted version of V
   S = V;
   n = length(V);
   for k = n:-1:2
      ... % sweeps a bubble from positions 1 to k
      ... % where k decreases from n down to 2
   end;
end;
```

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 (V)
   ...
      % sweeping the vector with a bubble
   for k = n:-1:2
      for i = 1:k-1 % advances the bubble from 1:2 to k-1:k
         if S(i) > S(i+1)
            x = S(i);          % swap the bubble
            S(i) = S(i+1);     % swap the bubble
            S(i+1) = x;        % swap the bubble
         end;
      end;
   end
end;
```

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 inner 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 sweep, …, and 1 in the n-1th sweep with a total of

$$(n-1) + (n-2) + \ldots + 1 = n-1 * (1+ n-1) (n-1) /2 \approx n^2/2 \text{ sweeps}$$

```
...
 for k = n:-1:2
   for i = 1:k-1
      if S(i) > S(i+1)
         ...
      end;
   end;
end
...
```

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