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

Lists: Search and Sorting

Pedro Barahona
DI/FCT/UNL
Métodos Computacionais
1st Semestre 2020/21

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 list (a vector).
  - But the same algorithms may be used with any data type with a "<" operator.
- In a list **V**, of length **n**, there are two types of search 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, 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 list this requires a **single** access to the vector.
  - In the second case, the different values of the list 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 list (vector), sequentially, checks the values one by one, **while it is worth doing it**, i.e. if finds the value while
  - it has not found the value yet; and
  - there are still values in the list.
- This sequential search can be specified by the following Python code

```
def find_seq_while(x, V):
    """returns the first index p of list V where x is to
    be found; if V does not contain x, then the
    function returns p = -1"""
    p = -1           # position returned if x is not found
    found = False    # x not found yet
    i = 0
    while i < len(V) and not found:
        if V[i] == x:
            p = i
            found = True
        else:
            i = i + 1
    return p
```

Sequential Search

- Alternatively, the search may be declared for the whole list, but it is interrupted as soon as the the value is found.
- This alternative sequential search can have a leaner specification in Python.

```
def find_seq_for(x, V):
    """returns the first index p of list V where x is to
    be found; if V does not contain x, then the
    function returns p = -1"""
    n = len(V)
    for i in range(n):
        if V[i] == x:
            p = i;
            return p
    return -1
```

Search

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

$$\lim_{n\to\infty} \#(n) \leq k\, f(n)$$

  where k is some constant.
- This complexity of an algorithm is usually expressed in Big O notation as

`O(f(n))`

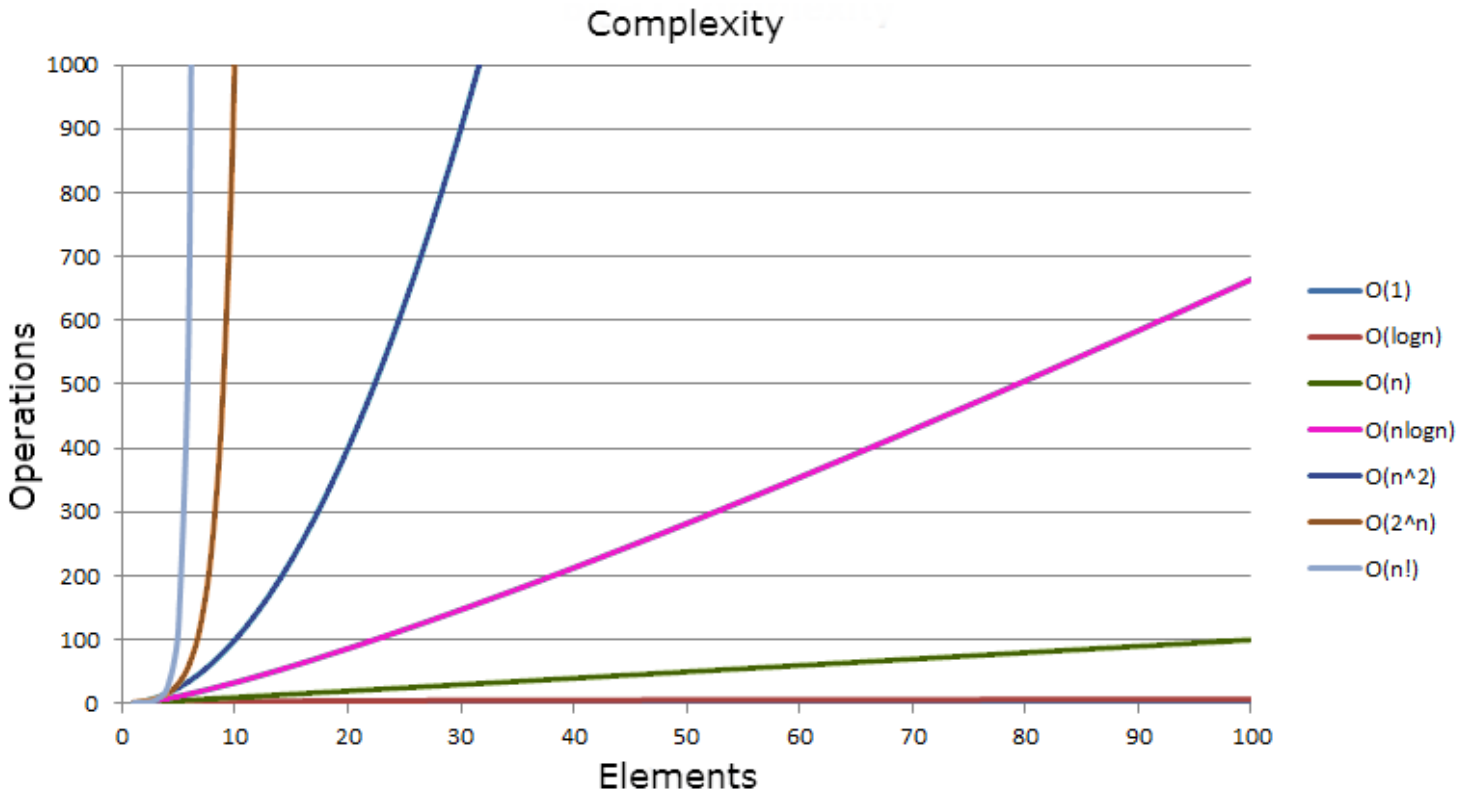
- In the above algorithms, for a list of length n, there is a maximum of n accesses to the elements of the list (to compare them with x) so both algorithms have **linear complexity**, i.e. complexity

`O(n)`

- **Note**: This complexity is **worst-case**. In practise the number of accesses is usually lower than n.

Search

- Algorithms may have different complexities and it is, of course, important to develop and use those with least complexity.
- The importance of the complexity issue can be highlighted in the following figure (more later)



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 list, 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 list, searching is much easier if the list is **sorted**:
  - Even if the search is sequential, as before, there is now the possibility to give up the search earlier, if one “overtakes” the value of interest;
    - In average, this reduces the number of accesses to **n/2**, although the worst-case complexity is kept as **O(n)**
  - A better search policy may be quite effective. As we will see, a a divide and conquer policy may bound the number of accesses to **log(n)**.
    - In which case the complexity becomes O(**log(n)**)

Improved Sequential Search

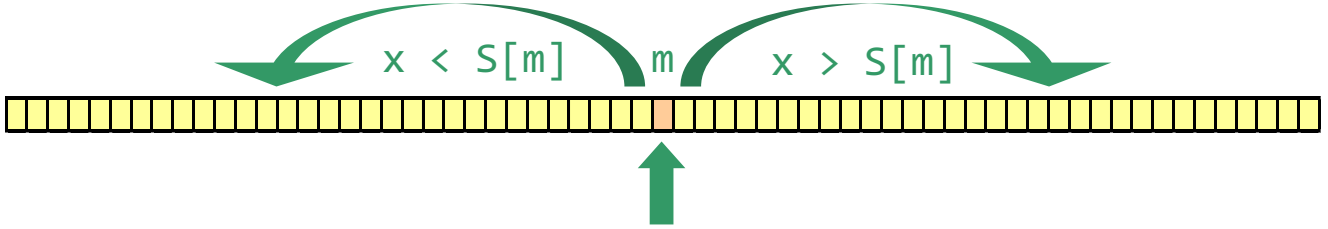
- If list **V** is sorted, the search for a value **x** can be interrupted earlier.
- Assuming **V** is sorted in increasing order, the previous algorithm can be adapted, interrupting the search **also** when an element greater than x is found (all the other elements will also be greater than x)

```
def find_seq_sorted(x, S):
    """returns the first index p of a sorted list S (in
    increasing order) where x is to be found; if V does
    not contain x, then the function returns p = -1"""
    n = len(S)
    for i in range(n):
        if S[i] == x:
            p = i;
            return p
        elif S[i] > x:   # abandon search
            return - 1
    return -1
```

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

Bipartite Search

- When a list **S** is sorted, we may indeed improve very significantly this complexity.
- To do so, we will define a function, that searches an element in the list between indices **lo** and **up**, where **0 ≤ lo ≤ up ≤ n-1**.
- 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 this mid point, i.e. in range **lo..m-1**
  - If **S[m] < x**, **x** must be searched after this mid point, i.e. in range **m+1..up**
- If the element is not present in the vector, the range eventually becomes null (i.e. the lower bound is larger than the upper bound) in which case the procedure returns -1.

Bipartite Search

- The above algorithm is easily implemented with a recursive function

```
def find_between_bounds(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, returns p = -1 """
    if lo <= up:
        m = round((lo+up)/2)
        if S[m] == x:
            return m     # x was found
        elif x < S[m]:   # look for x before m
            return find_between_bounds(x, S, lo, m-1)
        else:            # look for x after m
            return find_between_bounds(x, S, m+1, up)
    else:
        return -1
```

- **Note 1:** The test for the termination of recursion (lo <= up) must be done before the recursive calls.
- **Note 2**: Initially, this function must be called with lo = 0 and up = len(S)-1

Bipartite Search

- As mentioned, the above function can be used directly (if the bounds lo and up are provided) or it may be called from a “front-end” function that requires no bounds to be provided.

```
def find_in_sorted_list(x, S):
    """returns the index p of a vector S, sorted in
    increasing order, if x is to be found between
    indices lo and up. Otherwise, returns p = -1 """
    return find_between_bounds(x, S, 0, len(S)-1)
```

- In any case, the complexity of this bipartite seach is the complexity of function **find_between_bounds**, which is addressed next.

Bipartite Search

- To analyse the complexity of this binary search, made in a sorted list 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 → n**
  - Call 2 is made to a range of size n/2 — **2 → n / 2**
  - Call 3 is made to a range of size n/4 — **3 → $n / 2^2$**
  - …
- If the element has not been found yet, a 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 → $n / 2^{k-1} = 1$**
- At most, we have thus **k** calls (assumed to be elementary operations), and

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

- Hence, the algorithm has a worst-case logarithmic complexity, i.e. its complexity is

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

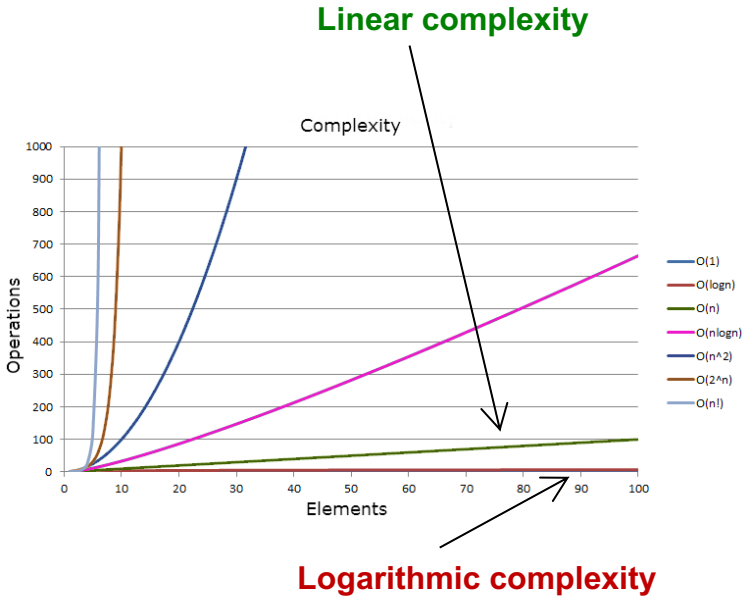
Bipartite Search

- 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 may 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, to find the data of any citizens, given their id number, rather than $5*10^9$ accesses, only $\log_2(10^{10}) \approx 23$ accesses are needed, a speed up of about 200 000 in each search!
- If each access takes 1 μsec, than
  - whereas a bipartite search is done in 23 μsec;
  - a 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).

Sorting

- 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 list.
- At each step, it assumes that a **prefix of size k** of the list, i.e. the first k elements of the list, 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 2^{nd} element in it
  - Continuing with a prefix of size 2 and inserting the 3^{rd} element in it
  - …
  - Ending with a prefix of size n-1 and inserting the n^{th} element in it.
- Of course, at the end of this process, the whole list 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 list to be equal to the original one.
- Then it executes a FOR loop, to insert the k^{th} element into the prefix of size k-1.
  - starting with k = 1; (comparing S[1] with S[0]), and
  - ending with k = n-1 (the last element of the original list)

```
def insert_sort_ite(V):
    """sorts vector V by progressively orderly inserting the
    k element into the prefix of S, from 0 to k-1"""
    S = V.copy()          # create a copy of V
    for k in range(1,len(V)):
       ...
    return S
```

Insert Sort

- The loop block inserts x, element in position k of S, into the prefix 0 …k-1, by
  - Finding p, the position where x should be inserted (later);
  - 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 down to p;
    - Of course, if p = k (i.e. the element remains in the same position) no pushing is actually done.

```
def insert_sort_ite(V):
    ...
    for k in range(1,len(S)):
       x = S[k]
       ... # p
       for j in range(k-1,p-1,-1):
          S[j+1] = S[j];      # push elements forward
       S[p] = x;
    return S
```

Insert Sort

- The position p where to insert x is found by sweeping the vector starting from position 0, 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 0..k-1 are less than x
    - In this last case, element S[k] = x remains in the same position k

```
def insert_sort_ite(V):
        ...
        i = 0
        found = False
        while not found:
            if i == k or S[i] >= x:
                p = i
                found = True
            i = i + 1
```

Insert Sort

- The full algorithm is shown below:

```
def insert_sort_ite(V):
    """sorts vector V by progressively orderly inserting the
    k element into the prefix of S, from 0 to k-1"""
    S = V.copy()          # create a copy of V
    for k in range(1,len(V)):
       x = S[k]
       i = 0
       found = False
       while not found:
          if i == k or S[i] >= x:
             p = i
             found = True
          i = i + 1
       for j in range(k-1,p-1,-1):
          S[j+1] = S[j];      # push elements forward
       S[p] = x;
    return S
```

Insert Sort - Complexity

- The complexity of Insert Sort can be assessed, looking at the structure of the algorithm (in the worst case).
- There are **n-1** loops, and in each loop, the element in the position **k** ($1 \leq k \leq n-1$), with value **x**, is inserted in a prefix of size **k**.
- This requires comparing the element with **m** elements until finding its position and pushing forward the remaining **k-m** elements. The number of accesses to the vector is thus **k** (for comparisons or push forward moves).
- Summing for all the loop instances, the number of accesses is about

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

- Hence, the algorithm requires $\mathbf{n^2/2}$ comparisons and **n-1** insertions.
- Assuming all accesses for comparisons and push forwards take roughly the same time, the time complexity of insert sort is thus

$$\mathbf{O(n^2)}.$$

- **Note**: This is both the typical and worst case complexity.

Insert Sort – Recursive version

- A recursive version of the algorithm, possibly more understandable, is shown below
  - If the vector has zero or one element,
    - it is already sorted and is returned.
  - Otherwise,
    - sort the first n-1 elements to obtain a sorted list (of length n-1); and
    - insert the last element in this sorted list, returning the extended list.

```
def insert_sort_rec(V):
    """returns the sorted version of vector V"""
    n = len(V)
    if n <= 1:
        return V
    else:
        P = insert_sort_rec(V[0:n-1])
        return insert_one(V[n-1], P)
```

- **Note:** The complexity of the recursive version is the same, but it requires several function calls, and auxiliary vectors, which makes execution somewhat slower.
- Inserting **x** in a sorted list **V** can also be defined recursively.

Insert Sort – Recursive version

- If the list is empty,
  - return the list with the element.
- Else, if the element is less than V[0],
  - insert it at the “head” of the list, and return the result
- otherwise,
  - insert x in the “tail” of list **V**; and
  - extend V[0] with the resulting list, and return the result

```
def insert_one(x,V):
    """ inserts x in the right position of sorted list V"""
    if len(V) == 0:
        return [x]
    elif x < V[0]:   # x is less than the first of V
        S = [x]
        S.extend(V)
        return S
    else:
        U = insert_one(x,V[1:len(V)])
        S = [V[0]]
        S.extend(U)
        return S
```

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 list (a bubble) are in the wrong order they should be swapped.
- Of course this swap operation has to be repeated several times.
  - Sweeping the list 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.
  - Further sweeping eventually places all elements of the list in the right position.
- In fact, for a list of length **n**:
  - Only **n-1** sweeps are needed: if all but the smallest element are sorted in the last positions, the smallest element must be in the first position;
  - Since the largest elements of the list are being placed in their right order, i.e. in the end of the list, the successive sweeps may be executed in successively smaller lists.

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 follows the previous explanation.
- The algorithm performs n-1 sweeps of the bubble, each sweep ending in successively smaller positions of the bubble, ranging from position n-1 down to 1.

```
def bubble_sort(V):
    """ returns the (bubble) sorted version of vector V"""
    S = V.copy              # create a copy of V
    n = len(V)
    for k in range(n-1,0,-1):   # sets the last position for the bubble
        for i in range(0,k):    # advances bubble
            if S[i] > S[i+1]:
                x = V[i]        # swaps the bubble
                S[i] = S[i+1]
                S[i+1] = x
    return S
```

Bubble Sort - Complexity

- The complexity of Bubble Sort can be assessed, looking at the structure of the algorithm with 2 nested loops.

```
...
for k in range(n-1,0,-1 ):
        for i in range(0,k):
            if V[i] > V[i+1]:
...
```

- The body of the inner loop, regarding the advance of the bubble in each sweep, is executed a variable number of times:
  - n-1 bubble advances in the 1st sweep,
  - n-2 bubble advances in the 2nd sweep,
  - …,
  - and 1 bubble advance in the n-1th sweep

  with a total number of sweeps (assumed to be elementary operations)

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

- The algorithm has thus quadratic complexity **$O(n^2)$**.