Optimised Sorting; Graphics

Pedro Barahona
DI/FCT/UNL
Métodos Computacionais
1st Semestre 2020/2021

Optimised Sorting in Lists

- **Insert Sort** and **Bubble Sort** are useful to sort "small" lists, due to its complexity **O(n^2)**.
- But larger lists 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.
- Hence, several different methods have been proposed to improve this quadratic complexity, and an animation that shows several such methods is available in URL

  https://www.youtube.com/watch?v=kPRA0W1kECg

- 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.

Optimised Sorting in Vectors

- As we will see next, these algorithms have an asymptotical complexity of

  O(n • ln(n))

- The difference between this complexity and the quadratic complexity **O(n²)** of the Bubble and Insert sort algorithms can be assessed in vectors of variable size **n**.

- The number difference in the number of elementary operations is

n	n^2	n • ln(n)
10	1.000E+02	2.303E+01
100	1.000E+04	4.605E+02
1 000	1.000E+06	6.908E+03
10 000	1.000E+08	9.210E+04
100000	1.000E+10	1.151E+06
1 000 000	1.000E+12	1.382E+07
10 000 000	1.000E+14	1.612E+08

- If an elementary operations takes 1 nsec, the time to sort the vector is

n	n^2	n • ln(n)
10	100 nsec	23 nsec
100	10 μsec	460 nsec
1 000	1 msec	6.9 μsec
10 000	100msec	92 μsec
100000	10 sec	1.2 msec
1 000 000	17 min	13.8 msec
10 000 000	28 hor	0.16 sec

Optimised Sorting in Vectors

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

Merge Sort:

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

QuickSort:

- Get a pivot.
- Divide the list into two sub-lists, composed of all the values smaller and larger than the pivot.
- Sort these two sub-lists.
- **Append** their solutions (virtually, since the vector is always the same)

Merge Sort

- As any recursive algorithm, the recursive function checks whether the recursion should stop, i.e. the problem is sufficiently simple to be solved directly.
- Here, we stop when the list 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-lists, and merges them.

```
def merge_sort(V):
    """ sorts list V with the merge_sort algorithm"""
    n = len(V);
    if n > 1:
        mid = math.floor((n/2)   # get mid index
        L = merge_sort(V[1:mid])    # left subvector
        R = merge_sort(V[mid:])   # right subvector
        return merge(L,R)
    else:
        return V
```

Merge Sort

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

```
def merge(L,R):
    """ merges two sorted lists L and R"""
    if len(L) == 0:
        return R
    elif len(R) == 0:
        return L
    elif L[0] <= R[0]:
        S = [L[0]]
        S.extend(merge(L[1:],R))
        return S
    else:                 # R[0] < L[0]
        S = [R[0]]
        S.extend(merge(L, R[1:]))
        return S
```

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).

- The complexity of sorting a list with $n = 2^k$ elements is the complexity of sorting two lists of 2^{k-1} elements plus merging two lists of 2^{k-1} elements each. This merge requires one operation per element, hence requires 2^k operations.

- Hence, and abusing notation, we have

$$C(2^k) = 2 \cdot C(2^{k-1}) + 2^k$$

- Now, we can use this recursive definition to obtain

$$\begin{aligned} C(2^k) &= 2 \cdot C(2^{k-1}) + 2^k \\ &= 2\,[\,2 \cdot C(2^{k-2}) + 2^{k-1}] + 2^k \\ &= 2^2 \cdot C(2^{k-2}) + 2 \cdot 2^k \end{aligned}$$

- More generally we have

$$C(2^k) = 2^m \cdot C(2^{k-m}) + m \cdot 2^k$$

Merge Sort – Complexity

- Now, the complexity of merge_sorting a list with size 1 is 1 (the function just returns the list).

- Combining the previous result

$$C(2^k) = 2^m * C(2^{k-m}) + m * 2^k$$

with the fact that for m = k we have

$$C(2^{k-k}) = C(1) = 1$$

we finally obtain

$$\begin{aligned} C(2^k) &= 2^k \bullet C(2^{k-k}) + k \bullet 2^k \\ &= 2^k \bullet 1 + k \bullet 2^k \\ &= 2^k (k+1) \approx k \bullet 2^k \end{aligned}$$

- Hence the asymptotical complexity of $O(2^k \bullet k)$. Finally, given that the size of the initial list is $n = 2^k$ (or $k = \log(n)$), we can express the complexity in terms of the size of the input list and so, *the complexity of merge sort for a list of size n* is

O(n log(n)).

Quick Sort

- Although Merge Sort offers good asymptotical complexity, the fact that it requires the creation of several sub-lists to be merged may be regarded as a significant disadvantage, specially in case of very large lists.
- An alternative would be to work always in elements of the list, such that only accesses to the existing list would be required.
- This can of course be done with Merge Sort, but then the merge of two sub-lists within a list is not very obvious (left as an exercise).
- This is not so with Quick Sort that does not require such merging. Basically, it analyses a list **V** of size **n** and swaps, if necessary, its elements until
  - An element, the **pivot**, occupies some mid position **k** in the vector ($\mathbf{V_k = p}$).
  - All elements **V(i)**, $\mathbf{1 \leq i < k}$, are less (or equal) than the pivot ($\mathbf{V(i) \leq p}$).
  - All elements **V(j)** ($\mathbf{k < i \leq n}$), are greater (or equal) than the pivot ($\mathbf{V(j) \geq p}$).
- Then all that is required is to sort (e.g. through a recursive call of Quick Sort) the sub-lists **left** and **right** of position **k**.

Quick Sort

- In more detail, Quick Sort 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 list, **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 list is swept with two indices starting at both ends of the vector range:
   - Index **i**, starts at 0, and increases during the sweep
   - Index **j**, starts at n-1, and decreases during the sweep
3. During this sweep, elements are swapped when they are not in the ***right side*** of the pivot.
4. The sweep ends when both indices i and j take the same value, **k**. At this point,
   - **V(k) = p**;
   - all values in positions less than **i** are less or equal than **p**; and
   - all values in positions greater than **i** are greater or equal than **p**.
5. Then, all that is needed is to sort the lower and upper sub-lists, which can of course be done recursively.
6. Some examples illustrate the algorithm.

Quick Sort

lo = 1; hi = 9, mid = 5; pivot = V(mid) = 6

i = 1	1	7	3	8	6	4	2	9	5	**j = 9**
i -> 2	1	7	3	8	6	4	2	9	5	**9 <- j**
	1	5	3	8	6	4	2	9	7	**swap**
i = 2	1	5	3	8	6	4	2	9	7	**j = 9**
i -> 4	1	5	3	8	6	4	2	9	7	**7 <- j**
	1	5	3	2	6	4	8	9	7	**swap**
i = 4	1	5	3	2	6	4	8	9	7	**j = 7**
i -> 5	1	5	3	2	6	4	8	9	7	**6 <- j**
	1	5	3	2	4	6	8	9	7	**swap**
i = 5	1	5	3	2	4	6	8	9	7	**j = 6**
i -> 6	1	5	3	2	4	6	8	9	7	**6 <- j**
k = 6	1	5	3	2	4	6	8	9	7	**stop**

Quick Sort

- Another example, where the pivot is quite skewed.

lo = 1; hi = 9, mid = 5; pivot = V(mid) = 8

i = 1	1	7	3	9	8	4	2	6	5	**j = 9**
i -> 4	1	7	3	9	8	4	2	6	5	**9 <- j**
	1	7	3	5	8	4	2	6	9	**swap**
i = 4	1	7	3	5	8	4	2	6	9	**j = 9**
i -> 5	1	7	3	5	8	4	2	6	9	**8 <- j**
	1	7	3	5	6	4	2	8	9	**swap**
i = 5	1	7	3	5	6	4	2	8	9	**j = 8**
i -> 8	1	7	3	5	6	4	2	8	9	**8 <- j**
k = 8	1	7	3	5	6	4	2	8	9	**stop**

- The remaining vectors to sort are quite different in size, but the algorithm is safe.

Quick Sort

- The basic structure of the **quick_sort** function is shown below. Note that the algorithm always deal with the same list, but with different parts of it, namely between the indices **lo** and **hi** (initially, 0 and len(V)-1, respectively).
- The sweeping illustrated before is implemented in function partition, that returns
  - the index k where the pivot lies, p = V[k] and the list V updated so that
  - elements in indices less/greater than k are less/greater or equal to pivot p.
- Then a recursive call is made to sort the left and right “parts” of V,
- … and the result is returned.

```
def quick_sort(V):
 """ sorts list V with the quick_sort algorithm"""
    qs(V, 0, len(V)-1)

def qs(V, lo, hi):
    """ quick sorts list V, between indices lo and hi"""
    if lo < hi:
        (V,k) = partition(V,lo,hi)
        V = qs(V, lo, k-1)
        V = qs(V, k+1, hi)
    return V
```

Quick Sort

- The sweeping starts with **i = lo** and **j = hi**, and the pivot is arbitrarily selected as the element in the midpoint of the range. Then, a sweeping proceeds while **i < j** as follows:
  - Indices **i/j** increase/decrease until an element is found no smaller/larger than the pivot
  - They are then swapped, unless **V[i]** and **V[j]** both take the value of the pivot
  - In the end, the partitioned list is returned together with the index of the pivot

```
def partition(V,lo,hi):
    i = lo
    j = hi
    mid = round((lo+hi)/2)
    pivot = V[mid]
    while i < j:


        while V[i] < pivot:
            i = i + 1
        while V[j] > pivot:
            j = j - 1
        if V[i] > V[j]:
            V = swap(V,i,j)
    return (V,i)
```

Quick Sort

- In fact there might be the case that V[i] = V[j] = pivot but i < j , i.e.
  - when the list has repeated elements, and one was chosen for pivot.

.

lo = 1; hi = 9, mid = 5; pivot = V(mid) = 6

i = 1	1	2	3	5	6	4	9	6	7	**j = 9**
i -> 5	1	2	3	5	6	4	9	6	7	**8 <- j**
	1	2	3	5	6	4	9	6	7	**swap??**
i => 6	1	2	3	5	6	4	9	6	7	**j = 8**
i -> 7	1	2	3	5	6	4	9	6	7	**8 <- j**
	1	2	3	5	6	4	6	9	7	**swap**
i = 7	1	2	3	5	6	4	6	9	7	**j = 8**
i = 7	1	2	3	5	6	4	6	9	7	**7 <- j**
k = 7	1	2	3	5	6	4	6	9	7	**stop**

- Hence, when V[i] and V[j] are both equal to the pivot and i < j than i must be increased to continue the sweep .

Quick Sort

- Hence, when V[i] = V[j] = pivot but i < j
  - i.e. the list has repeated elements, and one was chosen for pivot.

In this case, index i is incremented, as explained in the previous animated example, so that the sweep proceeds until i = j

```
def partition(V,lo,hi):
    i = lo
    j = hi
    mid = round((lo+hi)/2)
    pivot = V[mid]
    while i < j:
        if V[i] == pivot and V[j] == pivot:
            i = i + 1
        while V[i] < pivot:
            i = i + 1
        while V[j] > pivot:
            j = j - 1
        if V[i] > V[j]:
            V = swap(V,i,j)
    k = i;
    return (V,k)
```

Quick Sort

- Finally, the swapping of two elements of the vector with indices i and j is implemented in the obvious way.

```
def swap(V,i,j):
   aux  = V[i]
   V[i] = V[j]
   V[j] = aux
   return V
```

Quick Sort – Complexity

- The asymptotical complexity of Quick Sort can be obtained similarly to what was done with Merge Sort, but is not so "clear", since it depends on the returned position **k** of the pivot.
- If k is the mid point between lo and hi, then each range of size $\mathbf{n = 2^k}$ is divided into two equal subranges of size **n/2 -1**.
- Hence, the analysis is similar to what was done with Merge Sort, taking into account that function partition visits all **n** elements of the range once, and swaps elements a fraction of n, i.e. $\mathbf{a \bullet 2^k}$ times (where **a** is less than **1)**, hence

$$C(2^k) = 2 \bullet C(2^{k-1}) + (1+a) \bullet 2^k$$

$$C(2^k) \approx 2 \bullet C(2^{k-1}) + 2^{k+1}$$

- Doing a similar analysis as before, we note that

$$C(2^{k-1}) \approx 2 \bullet C(2^{k-2}) + 2^k \qquad \text{hence}$$

$$C(2^k) \approx 2 \bullet [2 \bullet C(2^{k-2}) + 2^k\,] + 2 \bullet 2^k$$

$$C(2^k) \approx 2^2 \bullet C(2^{k-2}) + 2^k + 2^k \text{ and, more generally}$$

$$C(2^k) \approx 2^m \bullet C(2^{k-m}) + m \bullet 2 \bullet 2^k$$

Quick Sort – Complexity

$$C(2^k) \approx 2^m \cdot C(2^{k-m}) + m \cdot 2 \cdot 2^k$$

- Taking into account that $C(2^0) = 1$, we make $m = k$ to obtain

$$C(2^k) \approx 2^k \cdot C(2^{k-k}) + k \cdot 2 \cdot 2^k$$

$$C(2^k) \approx 2^k (1 + k \cdot 2)$$

$$C(2^k) \approx 2^k (k \cdot 2)$$

- Again, since $n = 2^k$, this means the complexity of the search is

O(n log(n))

Quick Sort – Complexity

- In fact, although Quick Sort tends to be very efficient, its efficiency depends on a number of factors, overall, the choice of the pivot.

- In the limit, if the pivot is the smallest or the largest element of the vector, in each call of a vector with a range of size **n**, rather than having 2 subranges of size n/2 there is one empty range and another of size n-1.

- Hence, and simplifying, the complexity becomes

$$C \approx n + (n-1) + (n-2) + \ldots 1$$
$$\approx n\,(n+1) / 2$$
$$\approx \mathbf{O(n^2)}$$

i.e. quadratic, as in the case of Bubble Sort

- In fact, the number of accesses, **a**, to elements of the vector **V**, and the number of swaps, **s**, can be "counted" in a modified version of the algorithm, that rather than returning vector V, it returns the triple **(V,a,s).**

This is left as exercise.

Graphics in Python

- Several types of graphics (line graphs, pie graphs, histograms, …) and images can be drawn with Python, namely through the library **matplotlib**.

- Documentation on this library is available in

  https://matplotlib.org/

- Here we will only address line graphs, drawn with the following steps
  1. Clear all previous graph draws (**clf()**)
  2. Fill a vector x with the x-coordinate values.
  3. Fill one or more vectors with the y-coordinate values.
  4. Use function **plot(x, y, fmt)** to draw each of the lines of the graph.
  5. Define the title of the graph, axis and legend of the graph (all optional)
  6. Show and save the graph in a file (optional)

Graphics in Python

- There are many possibilities available to format the graphs.
- For the style of the lines a number of options can be used in the 3rd parameter of the **`plt.plot(…)`** function, that takes the X and Y coordinates of the line to be drawn:
  - **Colours**: ‘b’; blue, ‘g’: green, ‘r’: red, ‘y’: yellow, ‘k’: black
  - **Markers**: ‘.’: point, ‘o’: circle, ‘+’: plus, ‘x’-times, ‘*’: star
  - **Styles**: ‘-’ : solid, ‘--’: dotted, ‘:’ : dashed, ‘-.’ : dash-dot
- The graphs can be completed with further commands to provide:
  - a **title**, `plt.title(`*`title`*`)`
  - a **legend**, `plt.legend(`*`legend`*`)`
  - **labels** for the x and y- axes: `plt.xlabel(`*`xlabel`*`)` and `plt.ylabel(`*`ylabel`*`)`
  - saving into a **file** `plt.savefig(`*`filename`*`)`
- The graphs are shown in the console with command `plt.show()`, and can also be stored in a file, with command `plt.savefig(filename)` (usually with a png or pdf extension) for further use.
- see help(plt.plot) for more information on formats

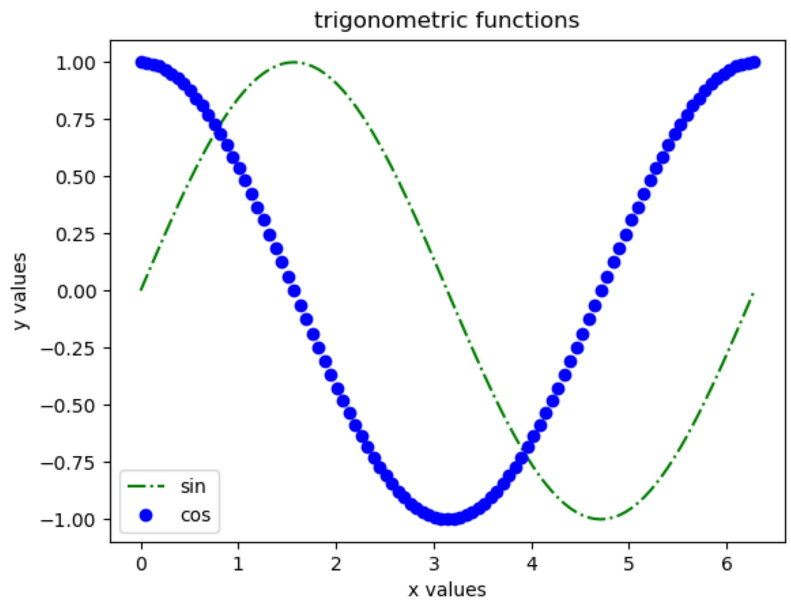
Graphics in Python

- The following example illustrates these steps to draw a graphic of the sine and cosine functions, with **np points**, in the range **`x_min .. x_max`**.

```
def plot_sine_cosine(np, x_min, x_max):
    """ plots sine and cosine functions, with np points,
    in the range x_min .. x_max"""
    delta = (x_max-x_min)/n                       # interval size
    X = [x_min + i * delta for i in range(np+1)]  # x-coordinates
    S = [m.sin(x) for x in X]                     # sine values
    C = [m.cos(x) for x in X]                     # cosine values
    plt.clf()                                     # clear graph
    plt.plot(X,S,'g-.’)                           # sine line format
    plt.plot(X,C,'bo’)                            # cosine line format
    plt.title('trigonometric functions')          # title
    plt.legend(['sin', 'cos'])                    # legend
    plt.xlabel('x values')                        # x-axis label
    plt.ylabel('y values')                        # y-axis label
    plt.savefig('trigo.png')                      # save the graph
    plt.show()                                    # draw the graph
```

Graphics in Python

- The graphic is shown in the console and also saved in file `‘trigo.png’`.




```
plt.clf()                               # clear graph
plt.plot(X,S,'g-.’)                     # sine line format
plt.plot(X,C,'bo’)                      # cosine line format
plt.title('trigonometric functions')    # title
plt.legend(['sin', 'cos'])              # legend
plt.xlabel('x values')                  # x-axis label
plt.ylabel('y values')                  # y-axis label
plt.savefig('trigo.png')                # save the graph
plt.show()                              # draw the graph
```

Bar Plots in Python

- Several types of histograms (bar plots) may be produced with Python, with a similarly way. The simplest plots, with a single category, may be drawn with, at least, the following steps:
  1. **Clear** the graph;
  2. **Fill** a vector **Y** with the values of each bar;
  3. **Fill** a vector **X**with the legend of each bar;
  4. **Draw** the bar chart `plt.bar(X, Y)`
- Additionally, one may specify
  - the **colour** of the bar — 3rd parameter of `plt.plot(…)`
  - a **title**, — `plt.title(`*`title`*`)`
  - **labels** for the x and y- axes: — `plt.xlabel(`*`xlabel`*`)` and `plt.ylabel(`*`ylabel`*`)`
  - a **legend** — `plt.legend(`*`legend`*`)`
  - saving into a **file** — `plt.savefig(`*`filename`*`)`

Bar Plots in Python

- The following example illustrates the specification of a simple bar plot.

```
def plot_single_bar_chart():
    """ plots a bar chart from vectors V and X"""
    X = ['0-9','10-13','14-16','17-18','19-20']
    V = [10, 20, 15, 35, 5]
    plt.clf()                                        # clear graph
#colors:
   # one of {'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w’}; or
   # one of the Tableau Colors from the 'T10' categorical palette:
   # {'tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple’,
   #'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan’
    plt.bar(X, V, color = 'tab:red')                # plot
    plt.title('Students Average Grades')            # title
    plt.legend(['grades %'])                        # legend
    plt.xlabel('grade ranges')                      # x-axis label
    plt.ylabel('% of total')                        # y-axis label
    plt.savefig('simple_plot_chart.png')            # save the graph
    plt.show()                                      # show graph
```

Bar Plots in Python

- To draw a multiple histogram, the procedure is similar, but care must be taken to specify the different X and Y coordinates, as well as the xticks (which are now convenient for labelling the categories).

```
def plot_double_bar_chart():
    """ plots a double bar chart"""
    V1 = [65, 75, 90, 80, 70]                    # bars 1 Heights
    X1 = [0.85 + v for v in range(5)]            # bars 1 x-position
    V2 = [90, 80, 85, 80, 95]                    # bars 2 Heights
    X2 = [1.15 + v for v in range(5)]            # bars 2 x-position
    plt.clf()                                    # clear graph
    plt.bar(X1, V1, width = 0.3, color = 'g')
    plt.bar(X2, V2, width = 0.3, color = 'tab:brown')
    plt.xticks( [1,2,3,4,5] , ['Calculus','Finance','Computing',\
               'Statistics','Optimisation'])  # bar names
    plt.title('Statitics on Course Grades ‘)
    plt.legend(['Theory','Labs'])
    plt.ylabel('% of Positive Grades’)
    plt.savefig('double_grades_chart.png’)
    plt.show()
```

Images in Python

- Images may also be drawn in Python. To draw a (rectangular) image the following steps must be made:

1. Define a dictionary of n colours, by means of an **nx3** matrix.
   - For each row of the matrix, define the [R,G,B] components, each in the range 0..1,
   - Create an object, for example, with name **my_cm**, with function `ListedColorMap` from library matplotlib.colors
2. Define a matrix **M**, corresponding to the (rectangular) grid of the image;
   - Fill the elements of matrix **M** with an integer **c** in the range 0..n-1
   - `Remove the axis information` `plt.axis('off')`
   - Draw the image `plt.imshow(M, cmap = my_cm)`
   - Saving into a **file** `plt.savefig(`*`filename`*`)`

- An example clarifies this procedure.

Images in Python

```
def plot_image():
    """ plots an image"""
    from matplotlib.colors import ListedColormap
    plt.clf()
    cores = [[0,1,0],[1,0,0],[1,1,0]] # [green, red, yellow]
    imagem = [[0,0,1,1,1],[0,2,2,1,1],\
              [0,2,2,1,1],[0,0,1,1,1]]
    my_cmap = ListedColormap(cores)
    plt.axis('off')
    plt.imshow(imagem,cmap = my_cmap)
    plt.savefig('flag.png')
```