Optimised Sorting in Vectors

Pedro Barahona
DI/FCT/UNL

Métodos Computacionais
1st Semestre 2018/2019

Optimised Sorting in Vectors

- **Insert Sort** and **Bubble Sort** are useful to sort “small” vectors, due to its complexity **O(n^2)**.
- 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.
- 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
- A 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(n^2)** 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 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 (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 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);
% S is the sorted version of vector V
% obtained by the merge sort method
   n = length(V);
   if n > 1
      mid = floor((1+n)/2);           % get mid index
      L = merge_sort(V(1:mid));       % left subvector
      R = merge_sort(V(mid+1:end)); % right subvector
      S = merge(L,R)
   else
      S = V;
   end
end
```

Merge Sort

- Merging two sorted lists is straightforward, and is implemented recursively below.
- The recursion stops when one of the sub-vectors is empty, in which case the merged vector is obtained by appending the two sub-vectors (since one is empty).
- 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);
% S is a sorted vector, obtained by
% merging the two sorted vectors L and R
   if length(L) == 0 || length(R) == 0
      S = [L,R];
   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))];
   end;
end
```

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 vector with $n = 2^k$ elements is the complexity of sorting two vectors of 2^{k-1} elements plus merging two vectors of 2^{k-1} elements each. This merge requires one operation per element, hence requires 2^k operations.
- Hence, and abusing notation, we have

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

- Now, we can use this recursive definition to obtain

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

- More generally we have

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

Merge Sort – Complexity

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

- 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

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

- Hence the asymptotical complexity of $O(2^k \bullet k)$. Finally, given that the size of the initial vector is $n = 2^k$ (or $k = \log(n)$), we can express the complexity in terms of the size of the input vector and so, *the complexity of merge sort for a vector 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-vectors to be merged may be regarded as a significant disadvantage, specially in case of very large vectors.
- An alternative would be to work always in elements of the vector, such that only accesses to an existing vector would be required.
- This can of course be done with Merge Sort, but then the merge of two subvector within a vector 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 vector **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)**, $\mathbf{1 \le i < k}$, are less (or equal) than the pivot ($\mathbf{V(i) \le p}$).
  - All elements **V(j)** ($\mathbf{k < i \le n}$), are greater (or equal) than the pivot ($\mathbf{V(j) \ge p}$).
- Then all that is required is to sort (e.g. through a recursive call of Quick Sort) the sub-vectors **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 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, **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**.
4. Then, all that is needed is to sort the lower and upper sub-vectors, which can of course be done recursively.
5. Some examples illustrate the algorithm.

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

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 are shown below. Note that the algorithm always deal with the same vector, but with different parts of it, namely between the indices lo and hi (initially they should be 1 and length(V)).
- The sweeping illustrated before is implemented in function partition, that returns
  - the index k where the pivot lies, and the vector V updated so that
  - elements in indices less/greater than k are less/greater or equal to pivot V(k).
- Then a recursive call is made to sort the left and right "parts" of V

```
function V = quick_sort(V, lo, hi)
   if lo < hi
      % sweep and change V to obtain an index k
      % such that all values before (after) k are
      % less (greater) or equal than V(k).
      [V, k] = partition(V,lo,hi);
      V = quick_sort(V, lo, k-1);
      V = quick_sort(V, k+1, hi);
   end
end
```

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 of interest.
- The 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

```
function [V, k] = 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; end
      while V(j) > pivot     j = j - 1; end
      if V(i) > V(j)         V = swap(V,i,j) end
   end
```

Quick Sort

- Eventually, i becomes equal to j (and V(i) = pivot) so the returned k is equated to i
- In fact there might be the case that V(i) = V(j) = pivot but i < j
    - If the vector has repeated elements, and one was chosen for pivot.

```
function [V, k] = 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; end
      while V(j) > pivot      j = j - 1; end
      if V(i) > V(j)          V = swap(V,i,j) end
   end
   k = i;
end
```

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

Quick Sort

- A more problematic example, with repetitions, namely when the chosen pivot appears more than once in the vector.

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**

- Note that 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

- Eventually, i becomes equal to j (and V(i) = pivot) so the returned k is equated to i
- In fact there might be the case that V(i) = V(j) = pivot but i < j
  - If the vector 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

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

Quick Sort

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

```
function V = swap(V,i,j)
   aux  = V(i);
   V(i) = V(j);
   V(j) = aux;
end
```

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 **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. **a • 2^k** times (where **a** is less than **1)**, hence

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

- This is similar to what was done before and leads to

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

thus leading to the same level of complexity of

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

$$
\begin{aligned}
C &\approx n + (n-1) + (n-2) + \dots 1 \\
&\approx n\,(n+1) / 2 \\
&\approx \mathbf{O(n^2)}
\end{aligned}
$$

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, with signature

```
function [V, a, s] = quick_sort(V, lo, hi)
```

which is left as exercise.