Discussing Sorting

Quicksort & Three More

Francesco Andreussi

Bauhaus-Universität Weimar

10 May 2019

Bauhaus-Universität Weimar

Fakultät Medien

Quicksort (1)

How does it sort the sequence *2-8-7-1-3-5-6-4*?

Quicksort (1)

How does it sort the sequence *2-8-7-1-3-5-6-4*?

first call QUICKSORT(A,1,A.len-1)

```
QUICKSORT(A,p,r):
  if p<r:
    q = PARTITION(A,p,r)
    QUICKSORT(A,p,q-1)
    QUICKSORT(A,q+1,r)
PARTITION(A,p,r):
  x = A[r]
  i = p-1
  for j = p to r-1:
    if A[j] <= x:
      i = i+1
      swap A[i] with A[j]
    swap A[i+1] with A[r]
  return i+1
```

x is the pivot now

i will point to the last item $\leq x$

pivot between the two subarrays

return position of current pivot

Quicksort Challenge!

Sort the array `A = <13,19,9,5,12,8,7,4,21,2,6,11>`.
You can do it by hand or write a program that does it for you:)

More efficient algorithms

Is it possible to sort an array in **linear** time?

More efficient algorithms

Is it possible to sort an array in **linear** time?
YES! With some restrictions, avoiding (as much as possible) comparing operations!

More efficient algorithms

Is it possible to sort an array in **linear** time?
YES! With some restrictions, avoiding (as much as possible) comparing operations!

- Counting Sort
- Radix Sort
- Bucket Sort

Counting Sort (1)

This algorithm works in time $\Theta(n)$ **if**...

- ...the n items of the input array are integers between 0 and k for some integer k,
- ...$k = O(n)$.

Moreover, Counting Sort is **stable** and that will be useful later.

Counting Sort (2)

Pseudocode

A is the input array, B the output array

```
CountingSort(A,k):
  B = emptyArray[A.len]
  C = emptyArray[k]
  for j = 0 to A.len-1:
    C[A[j]]++                      C[i] stores n. of occurrences of i
  for i = 1 to k:
    C[i] += C[i-1]                 C[i] stores n. of elems ≤ i
  for j = A.len-1 downto 0:
    B[C[A[j]]-1] = A[j]
    C[A[j]]--                      finds the correct place in B for A[j]
  return B
```

Let's try to order `A = <6,0,2,0,1,3,4,6,1,3,2>`!

Radix Sort

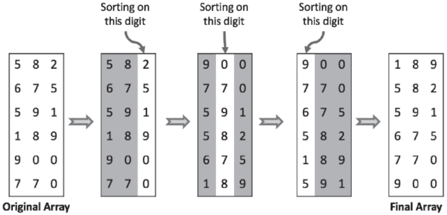

Fig. 1: Radix Sort Example

The algorithm runs sorting the input numbers/strings from the end (the least significant element) and uses as a subroutine another **stable** sorting algorithm for performing the intermediate steps.
For n items made of d elements, which can have k different values, the algorithm has a complexity of $\Theta(d(n+k))$ if the subroutine takes $\Theta(n+k)$ time to be executed. If $n \approx k$ Counting Sort is the perfect choice and if d is constant Radix Sort works in linear time.

Radix Sort Challenge!

Order the following series of items (write a program for this purpose in your spare time...if you want):

```
COW, DOG, SEA, RUG,
ROW, MOB, BOX, TAB,
BAR, EAR, TAR, DIG,
BIG, TEA, NOW, FOX.
```

Bucket Sort (1)

Bucket Sort is a *family* of algorithms, i.e. it can be implemented in a lot of different ways using the same core idea: subdivide the elements of the input in **buckets**, sort the obtained subsets and put everything together again.
If the input is uniformly distributed it is possible to assume that, for a good number of buckets and a good *partial-order preserving* mapping (hashing) function, this is true:

$$\sum_{i \in buckets} size(i)^2 = \Theta(n)$$

Thus, ordering with Insertion Sort all the buckets will take **linear** time!

Bucket Sort (2)

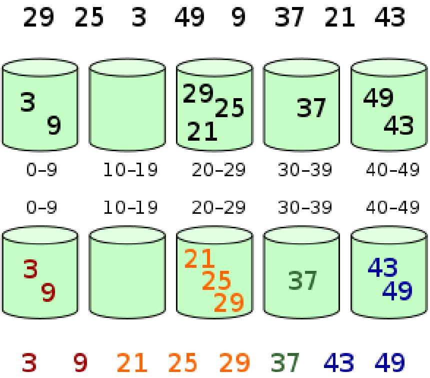

Fig. 2: Bucket Sort Example

Counting Sort and **Quicksort** are special cases of Radix Sort.

Thanks for the Attention!