Parma University, Math. Department

Algorithms and Data Structures
Linear time sorting

Marco Pellegrini

Summary of lecture 8.

Lower bounds on sorting
Counting Sort
Radix sor
Bucket sort

Comparison-based sorting.

- Insertion-sort, merge-sort, heap-sort and quick-sort are examples of sorting methods based on comparing elements (asking a=b, a>b or a<b).
- The fastest comparison based algorithms take time O(n log n).
- We will see that every comparison based sorting algorithm must use $\Omega(n \log n)$ time.
- We will see then three methods that sort in linear time, not using only comparisons, provided that the input has special properties.

Permutations and sorting

A *permutation* p is a one-to-one function $p:[1,..,n] \rightarrow [1,..,n]$.

The inverse p^{-1} of a permutation p is a permutation.

Sorting an array A[1,..n] is equivalent to finding a permutation p such that for every $i \in [1,..,n-1]$

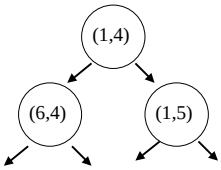
$$A[p(i)] < A[p(i+1)].$$

Afterwads, the sorted array B is just obtained by assigning B[i]=A[p(i)].

Since we can go from sorted B back to A by applying p^{-1} and we obatin a legal input, every permutation must be a possible outcome of a sorting algorithm.

Decision trees

Algorithms based on comaprisons may be seen abstractly as *decision trees.* The nodes are comparisons (i,j), comparing A[i] with A[j]. There are two outgoing edges correspondign to outcome $A[i] \leq A[j]$, and $A[i] > A[j]$



In a decision tree we *only allow comparisons* among the content of the input array A, and the height of a node *counts only the number of comparisons*. We can do anything on the indices.

Insertion-sort, merge-sort, quick-sort, heap-sort fit in the model.

The longest path from root to a leaf is the worst case count on the nubmer of comaprisons needed (height of the tree).

A tree of height h has at most 2^h leaves. Each leaf correspont to at most 1 permutation, and for input size n, there are n! permutaions. So $n! \leq 2^h$ must hold. Taking logs:

$$h \geq \log(n!) \geq \log(n/e)^n = n \log n - n \log e = \Omega(n \log n)$$

Stirling

Counting Sort

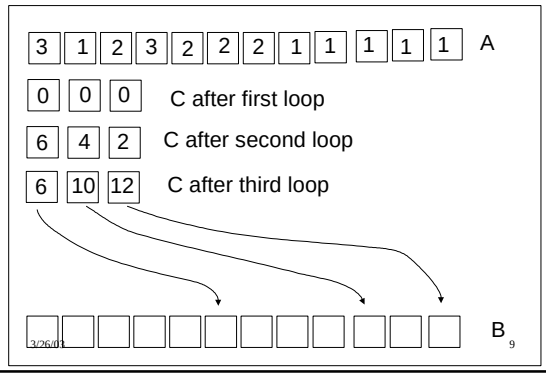
- We assume that the elements to sort are integers in the range [1..k] and that k is not large, that is for a constant c, k<cn.
- Under these assumption Counting Sort runs in time O(n).
- Here the new ingredient is using the elements to be sorted as indices.
- We have the input array A[1..n], the output array B[1..n] and an auxiliary array C[1..k].
- Note that, if k < n than some elements are repeated in A.

Counting Sort

(the code)

```
Counting-sort(A,k)
      FOR i := 1 TO k DO  C[i] := 0 END FOR
      FOR i := 1 TO lenght(A) DO
          C[A[i]] ++
      END FOR % C[j] counts the number of j's in A.
      FOR i := 2 TO k DO
        C[i] := C[i-1] + C[i]
      END FOR % C[j] counts the elements ≤ j.
      FOR i := length[A] DOWNTO 1 DO
         B[C[A[j]]] := A[j]
         C[A[j]]--
      END FOR
      RETURN B
```


Example k=3, n= 12



Time analysis and Stability

- Exercise: Find the asymptotic time of count-sort and justify your answer.
- Stability. Counting sort is a *stable* algorithm.
- By stability we mean that for any two elements that are equal in A maintain the relative order of their indices.
- That is if A[i] = A[k] and i < k, than A[i] is copied to position j and A[k] is copied to position m in the output array, then j < m.
- Being careful in the implementation we can make sure that insertion-sort, merge sort are or become stable.

Radix-sort

- Radix-sort is useful when we sort simultaneously on different keys and we want to produce the *lexicographic* order of the element.
- Example: lexicographic order of words (of same length)
- Example: sort by year,month,day.
- Example: sorting long-integers by their decimal representation.
- Radix-sort uses an other sorting algorithm that must be stable.

Radix-sort

(the code)

- Input: an array of A decimal numbers of d digits Digit d is the most significant, digit 1 the least significant.
- ```
  Radix-Sort(A,d)
    FOR i := 1 TO d DO
       Sort A on the i-th digit.
    END FOR
  ```
- "Sort" must be ***stable***, and possibly ***in place***.
- The proof of correctness is by induction on d.
  ```

Observations and exercise

Exercise: Use radix-sort to sort lexicographically: COW, DEG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

In Radix-sort we start from the least significant digit (or key) working up towards the most significant one.

Would radix sort work if we would go the other way around from the most significant to the least significant digit?

Exercise:

Sort n integers with values in the range $[1..n^2]$ in time O(n).

Bucket-sort

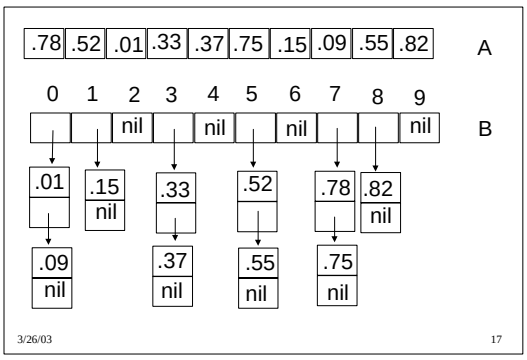
- Bucket sort runs in expected linear time when we assume that the n elements of the input are (real) numbers drawn uniformly in the interval [0..1].
- The idea is to split the interval [0..1] into n buckets and put each element in the right bucket. Sort each bucket separately. Collect the elements by scanning the buckets.
- Input A[1..n] where $0 \leq A[i] < 1$. We use an auxiliary array B[0,..,n-1] of pointers to lists. On such lists we can do standard list operations.

Bucket-sort

(the code)

```
Bucket-sort(A)
    n := length(A)
    FOR i := 1 TO n DO
      insert A[i] in list B[ ⌊nA[i]⌋ ]
    ENDFOR
    FOR i := 0 TO n-1 DO
       Sort  list B[i] by insertion-sort
    ENDFOR
    FOR i := n-1 DOWNTO 1 DO
      Append list B[i] to list B[i-1]
    ENDFOR
    Output B[0]
```


Example of buckets



Exercise:

Simulate bucket-sort on:

.79, .13, .16, .64, .39, .20, .89, .53, .71, . 42

Binomial distribution

- The crucial point is: how many elements do we expect to land in the same bucket?
- To analyze this we use some fundamental properties of the binomial distribution
- We are given a coin that gives *head* with probability p, and *tail* with probability q=1-p.
- We flip the coin *m* times and we count the number of times it is head. This number X follows the binomial distribution.
- In particular $E[X]=mp$ and $E[X^2]=mpq + p^2m^2$.

Analysis of Bucket-sort

- Let us fix a particular bucket b_k.
- Since the numbers in A are drawn from a uniform distribution, each element has probability $p = 1/n$ of landing in b_k and $q=1-p=1-1/n$ of missing it.
- Since this is true for all elements, the number of elements in b_k follows the binomial distribution with m=n.
- Substituting we have $E[|b_k|^2] = 1 -1/n +1 < 2$.
- So the expected time to sort one bucket is O(1).
- Considering all other pieces of the code, we get O(n) expected time for the whole algorithm

Homework

HW2.2
Change Counting-Sort so that it sorts in place:
use A and C (but not B) plus possibly O(1) extra memory.

Conclusions

We have seen two sorting methods (**counting-sort** and **bucket-sort**) that run in worst case linear time or expected linear time when we have special conditions on the input numbers.

We have seen **radix-sort** that allow to sort lexicographically multy-key entries.