APPLIED ROOTS

Source: MIT 6.006 Fall 2009

Sorting Lower Bounds, Counting Sort, Radix Sort

Lecture Overview

- Lower bounds vs Upper Bounds
- Sorting lower bounds
- Linear-Time Sorting: Counting Sort
- Stable Sorting
- Radix Sort

Readings

CLRS 8.1-8.4

Upper Bounds vs Lower Bounds

An upper bound shows that a specific computational problem can be solved in time $O(T(n))$.

Algorithms are upper bounds. E.g. Insertion Sort shows an upper bound of $O(n^2)$ on sorting, while Binary Insertion Sort shows a better upper bound of $O(n\log n)$.

A lower bound shows that there is no hope of solving a specific computational problem in time better than $\Omega(T(n))$.

(Trivial) example: time needed to find the smallest number in an array of length n is $\Omega(n)$, since in particular we need to read the elements of the array.

Lower Bound for Comparison Sorting

The algorithms we have seen so far are all comparison-based algorithms: recipes for sorting n elements by comparisons.

All these algorithms can be described by a binary tree, explaining the further comparisons that the algorithm will perform depending on the outcomes of the comparisons it has already performed.

Figure 1, shows a decision tree for sorting 3 numbers.

The input to the algorithm is an array containing three numbers: ha_1,a_2,a_3i. Every node of the tree is labeled with a pair of indices $i : j$ denoting that at the corresponding step of the algorithm a comparison between elements a_i and a_j will occur. If the outcome of the comparison is $a_i \leq a_j$ the algorithm follows the left branch, otherwise it follows the right branch. Every leaf contains a permutation of the indices, i.e. an ordering of the input elements.

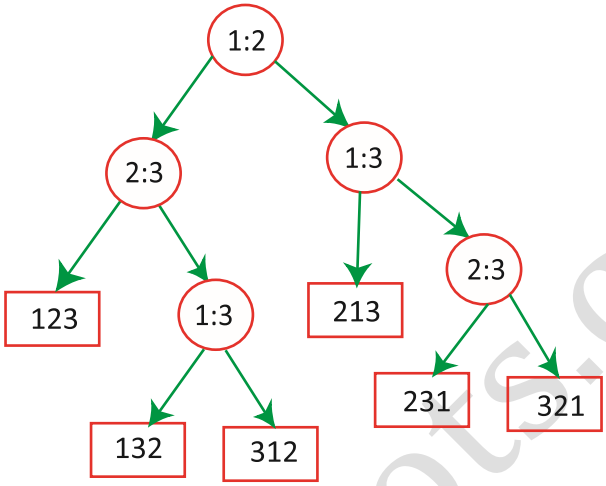

Figure 1: Decision Tree

e.g. using the decision tree of Figure 1 to sort the input array $< a_1,a_2,a_3 >:=< 9,4,6 >$

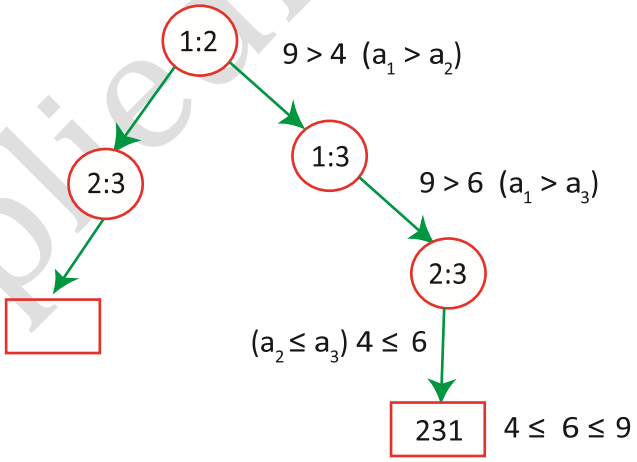

Figure 2: Decision Tree Execution

Decision Tree Model for Sorting n elements

Can model the execution of any comparison-based sorting algorithm via a decision tree:

- one tree for each n
- running time of algo for a given instance: length of the path taken
- worst-case running time: height of the tree

Theorem

Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Proof: How many leaves does the decision tree have? Tree must be $\geq n!$ leaves since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves. Thus,

$$\begin{aligned} n! &\leq 2^h \\ \implies h &\geq \log(n!) \quad \left(\geq \log\left(\left(\frac{n}{e}\right)^n\right)\right) \\ &\geq n \log n - n \log e \\ &= \Omega(n \log n) \end{aligned}$$

by Stirling's approximation

Counting Sort

If we stick to comparison-based sorting methods cannot do better than $\Omega(n \lg n)$.

In certain cases, we can exploit the small range of data to break the lower bound. E.g. suppose input array $A[1...n]$ satisfies: $A[j] \in \{1, 2, \cdots, k\}$, for all j. In this case, can order in time $O(n + k)$, as follows:

- create auxiliary array $C[1...k]$; set $C[i] = 0$ for all i;
  /* element $C[j]$ represents the number of elements in A having value j; in other words C is the array of **frequencies** of the numbers $1,...,k$ in the input array A.*/
- for all $i = 1,...,n$: increase $C[A[i]]$ by 1.
- **Now what?**
  - initialize an empty output array $B[1...n]$ of length n;
  - for all $j = 1,...,k$: fill in the next $C[j]$ unoccupied positions of array B with the value j.

Stable Sorting

Preserves input order among equal elements. Stability becomes an important property when each element of the input array is associated with a record carrying more information than

just the element being sorted. An application where stability becomes important is the radix sort method given below.

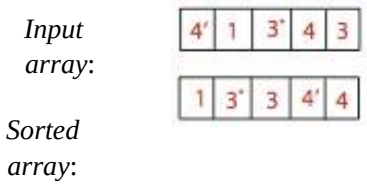

Figure 3: Stability

e.g.1 **Merge Sort** is a stable sorting method. And **Counting Sort** can be modified into a stable sorting method (we give the modification below).

e.g.2 **Heap Sort** and **Selection Sort** [1] are NOT STABLE! See Figure 4.

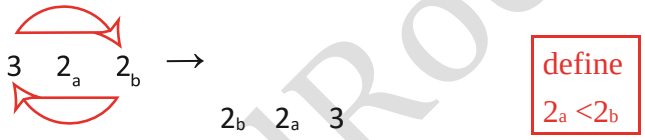

Figure 4: Selection Sort Instability

A Stable Counting-Sort Algorithm

In Figure 5 we show the modification of the counting sort algorithm to make it a stable sorting method. The trick is to replace the frequency array by a cumulative frequency array. That is, $C[j]$ represents the number of elements in A with value $\leq j$ (as opposed to $= j$ in the previous implementation). The cumulative frequency array is rather useful because it gives us the range in the output array in which we should store the elements of A having a particular value. In particular, the elements of A with value j should be stored in positions $C[j-1]+1$ through $C[j]$ of the output array B. The pseudocode of the new implementation is given below. It is easy to check that it is a stable sorting method.

[1] Find maximum element and put it at end of array (swap with element at end of array). Continue.

```
θ(k)  { for i ← 1 to k
        do C [i] = 0
θ(n)  { for j ← 1 to n do C [A[j]] =
        C [A[j]] + 1
θ(k)  { for i ← 2 to k do C [i] = C
        [i] + C [i-1]
θ(n)  { for j ← n downto 1 do
        B[C [A[j]]] = A[j]
        C [A[j]] = C [A[j]] - 1
      ---------------------------
θ(n+k)
```

Figure 5: Counting Sort

Example Execution

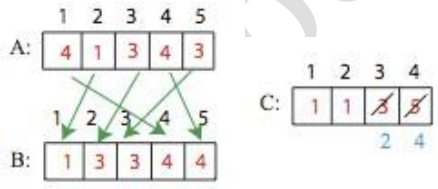

Figure 6: Counting Sort Execution

Radix Sort

Sort multi-digit numbers digit-by-digit.

Least Significant Digit (LSD) strategy: sort numbers based on least significant digit first, then sort based on second to least significant digit, etc. *Most Significant Digit (MSD) strategy*: go the other way around.

e.g.: A sample execution of radix sort using the LSD strategy is given in Figure 7. If the sorting method used for sorting by digit is stable, the correct ordering is output in the end of the execution.

Source: MIT 6.006 Fall 2009

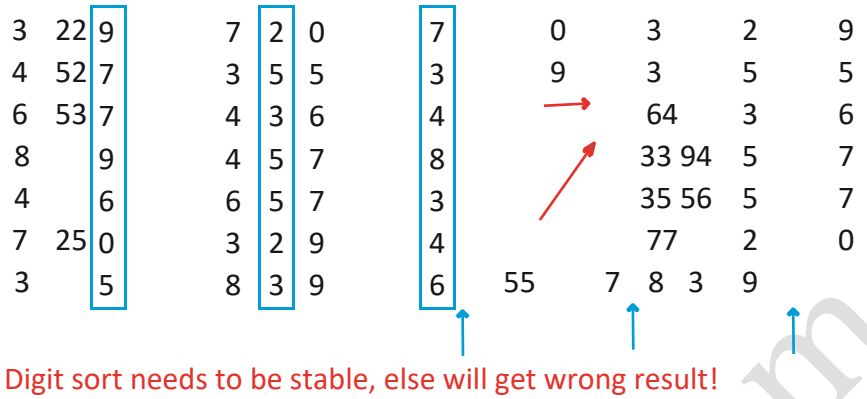

Figure 7: Example of Radix Sort with the LSD strategy.

The LSD strategy works because more significant digits have stronger impact on the ordering than less significant digits (since they are processed at a later round). In particular, if an element $A[i]$ has smaller MSD than an element $A[j]$, then $A[i]$ is placed before $A[j]$ (at the last round of the algorithm, corresponding to MSD). If two elements $A[i]$ and $A[j]$ have equal MSD's but differ in the second-to-most-significant-digit, then the last round does not affect the ordering of the two elements (it is IMPORTANT for this to be true that we use a stable sorting algorithm for sorting each digit). So the ordering is determined by the second to last round of the algorithm, corresponding to the second-to-MSD; etc.

e.g.2: **Does the MSD strategy work?** Below we show a sample execution of radix sort with the MSD strategy. Radix sort fails to sort correctly with this strategy.

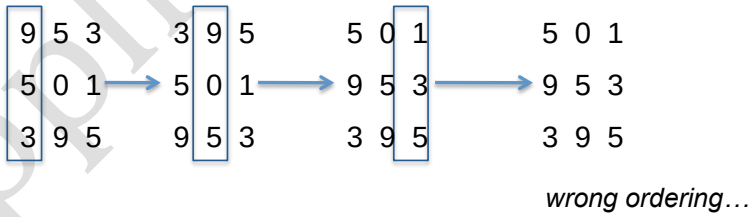

Figure 8: Example of Radix Sort with the MSD strategy. The wrong ordering is output.

Analysis

Assume that n numbers are given in base b, each number having d digits. Assume also that counting sort is implemented to be a stable sorting method.

→ Radix sort using counting sort for sorting per digit takes time: $d \cdot (n+b)$.

Tradeoffs

Suppose we have n numbers of b bits each.

One pass of counting sort $\Theta(n + 2^b)$

b passes of counting sort $= \Theta(nb)$ $\left(\frac{b}{r}(n + 2^r)\right)$ $\Theta(b(n + 2))$ $r \approx \log n$ $\longrightarrow \Theta\left(\frac{bn}{\log n}\right)$

b passes Θ —minimized when r