CS231 Algorithms
Prof. Lyn Turbak
Wellesley College

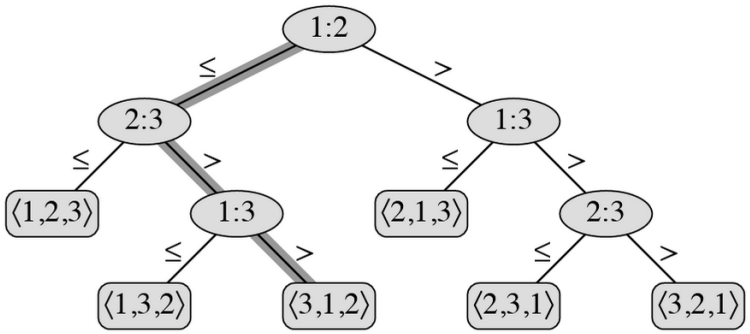
Handout # 8
March 8, 2022
Revised April 19, 2022

Comparison-based Sorts and Linear Sorting (Notes with Solutions)

Reading: *CLRS* 8.1–8.3

The Best Worst-Case Running Time for Comparison-Based Sorts

We can view **comparison-based** sorts as **binary decision trees** that process sets of arrays over n distinct elements. For example:



Given n distinct elements, how many different arrays can we make with these elements (with no duplicates)? (For example, consider {A, B, C}.)

$n! = \prod_{i=1}^{n} = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$

So a decision tree must have this number of leaves (shown as permutations in the above picture).

What is the minimum height of a binary tree with L leaves?

$\lceil(\log_2(L))\rceil$. E.g. a binary tree with 1 leaf has height 0, with 2 leaves has height 1, with 3 to 4 leaves has height 2, with 5 to 8 leaves has height 3, and so on. The above example binary decision tree with 6 leaves has height 3.

What is the minimum height of a decision tree for arrays of length n?

A decision tree for n elements has $n!$ leaves, so has height $\lceil(\log_2(n!))\rceil = \lceil(\lg(n!))\rceil$

Note that $\lg(n!) \approx n \cdot \lg(n)$ via Stirling's equation or calculus (see *CLRS* Exercise 8.1-2).

Key takeaway from this analysis: The worst-case running time of a comparison-based sort is $\Omega(n \cdot \lg(n))$.

Linear Sorting Algorithms

All **comparison-based** sorting algorithms have worst-case time $\Omega(n \cdot \lg(n))$, and we know one sorting algorithm (MergeSort) whose worst-case time is $\Theta(n \cdot \lg(n))$. (We will see another — HeapSort — next week.)

Yet there are linear sorting algorithms – algorithms with worst-case time $\Theta(n)$. How can this be? They're not comparison-based! They take advantage of some feature of the input. E.g.:

- The elements are integers in a restricted range, or easily convertible to integers in a restricted range (e.g., the letters of the alphabet). Today we will focus on this situation.
- The elements have a special probability distribution. This is the key idea in Bucket Sort (*CLRS* 8.4), which will will **not** study.

Integer Bucket Sort

Problem: sort n integers taken from the range $1..k$.

function IntegerBucketSort(A,B,k) ▷ A is input array of length n containing keys in range $[1..k]$.
▷ B is output array of length n.
  $Counts \leftarrow newArray(k, 0)$ ▷ Create new array of length k in which every slot is filled with 0
  for all num **in** A **do** ▷ Find count of each number in A.
    $Counts[num] \leftarrow Counts[num] + 1$
  $p \leftarrow 1$ ▷ p is a index in the output array B
  for all $num \in [1..k]$ **do**
    for all $\_ \in [1..Counts[num]]$ **do** ▷ $\_$ is an unused variable in this **for all** loop,
▷ which just repeats the loop body $Counts[num]$ times.
      $B[p] \leftarrow num$
      $p \leftarrow p + 1$

Exercise 1: For $k = 6$, $A =$ `4 2 1 3 2 4 2 6 1 2 4`, show *Counts* and B.

Counts is

index	1	2	3	4	5	6
value	2	4	1	3	0	1

and B is

index	1	2	3	4	5	6	7	8	9	10	11
value	1	1	2	2	2	2	3	4	4	4	6

Can `A` be the same as `B` in this algorithm?

Yes. After *Counts* is created, no information from A needs to be retained for populating B

Keys and Satellite Data

In practice, elements to be sorted are compound values/objects with multiple components/fields. Example from CS111 sorting lecture:

```
[ ('Ed', 'Jones', 18), # first name, last name, age
  ('Ana' 'Doe', 25),
  ('Ed', 'Doe', 18),
  ('Bob', 'Doe', 25),
  ('Ana', 'Jones', 18) ]
```

In general, want to sort these objects by some field (known as the **key**) and the rest of the object (sometimes called **satellite data**) moves with it. E.g., in the above example, we could sort the Python tuples by first name, last name, or age.

Counting Sort (a.k.a. Distribution Counting)

Integer bucket sort has a problem when there is satellite data. E.g., suppose $(k,\ d)$ pairs a key k with a satellite data integer d:

```
(4,1) (2,1) (1,1) (3,1) (2,2) (4,2) (2,3) (6,2) (1,1) (2,4) (4,3)
```

(Here, it happens that the satellite data for a particular key is sorted left to right, but that's just "coincidence" to illustrate the stability property of CountingSort; see the next page.)

function CountingSort(A,B,k) ▷ A is input array of length n containing keys in range $[1..k]$. ▷ B is output array of length n.
  $Counts \leftarrow newArray(k, 0)$ ▷ Create new array of length k in which every slot is filled with 0
  for all *elt* **in** A **do** ▷ Find count of each element key in A.
    $Counts[key(elt)] \leftarrow Counts[key(elt)] + 1$
  for all $key \in [2..k]$ **do** ▷ Find partial sums of key counts in A.
    $Counts[key] \leftarrow Counts[key] + Counts[key - 1]$
  for all p in $[len(A)..1]$ **do**
    $B[Counts[key(A[p])] \leftarrow A[p]$
    $Counts[key(A[p])] \leftarrow Counts[key(A[p])] - 1$

Exercise 2: For $k = 6$ and A= `(4,1) (2,1) (1,1) (3,1) (2,2) (4,2) (2,3) (6,1) (1,2) (2,4) (4,3)` modify the *Counts* array from Exercise 1 to calculate the partial sums and show the final contents of the array B.

Counts is initially

index	1	2	3	4	5	6
value	2	4	1	3	0	1

but becomes

index	1	2	3	4	5	6
value	2	6	7	10	10	11

after the partial sums step, and its slots are cleverly decremented as B is populated in order to fill the the contiguous segment of slots "set aside" for each key in B from back to front. E.g., in the partial sums version of *Counts*, slots 1–2 are allocated for pairs with key = 1; slots 3–6 are allocated for pairs with key=2; slot 7 is allocated for the pair with key=3; slots 8–10 are allocated for pairs with key=4, and slot 11 is allocated for the pair with key=11. The final version of B is:

index	1	2	3	4	5	6	7	8	9	10	11
value	(1,1)	(1,2)	(2,1)	(2,2)	(2,3)	(2,4)	(3,1)	(4,1)	(4,2)	(4,3)	(6,1)

Stability

A sorting algorithm is said to be **stable** if elements with the same key have the same relative order in the input and output arrays.

Is CountingSort stable?

Yes. As illustrated in the above example, for each key that appears in at least one pair, the slots set aside for that key in the output array B are filled from back to front by processing the pairs in A from back to front. This guarantees that the relative order of pairs with the same key in A is preserved in B.

Would CountingSort be stable if the **for all** loop counted up rather than down?

No. If the **for all** loop populating B counted up rather than down, the slots set aside for a given key in the output array B would still be filled back to front, but the relative order of the pairs would be reversed.

Stability is an incredibly important sorting property, and is usually documented in APIs.

Other Properties of Counting Sort

Can A be the same as B in CountingSort ?

No. The original A is referenced throughout the part of CountingSort that populates B, so the two arrays must be distinct for the algorithm to work correctly.

How much time does CountingSort take?

- The creation of the *Counts* array takes time $\Theta(k)$;
- The initial population of of the *Counts* array takes time $\Theta(n)$, where $n = len(A)$.
- The partial sums step on *Counts* takes time $\Theta(k)$;
- The step that populates B takes time $\Theta(n)$;
- The sum of all the above steps is $\Theta(k) + \Theta(n) = \Theta(n + k)$. It's necessary to keep n and k distinct in this analysis to handle the situation where k depends on n. E,g., in some situations, it might be that $k = n^2$, in which case $\Theta(n + k) = \Theta(n + n^2) = \Theta(n^2)$, which most certainly is not linear!

How much space does Counting-Sort need?

It depends on what space is counted. Typically, space taken up by the input and output (in this case, arrays of size n) is not charged to the algorithm, but space used **internal** to the algorithm **is** charged. The main space used in the algorithm is the *Counts* array, which takes $\Theta(k)$ space.

Radix Sort

Problem: Suppose you have a machine that can perform a stable sort on the ith digit of a d-digit number. How can you use the machine to sort a "pile" of n d-digit numbers?

Example:

443		431		211		121
124		321		321		122
232		211		121		123
431		441		122		124
132		121		123		132
123	sort by last digit	232	sort by middle digit	124	sort by first digit	211
321	$\Longrightarrow$	132	$\Longrightarrow$	431	$\Longrightarrow$	232
211		122		232		321
121		442		132		431
441		443		441		441
122		123		442		442
442		124		443		443

To avoid lots of intermediate piles, process digits right-to-left, not left-to-right!

This approach to sorting is known as RadixSort.

function RadixSort(A,d) ▷ Permute the elts of A to be in ascending sorted order by d "digits"
  for all $j \in [1..d]$ **do**
    StableSort(A,j) ▷ Can use any stable sort that uses "digit" j as a key

Any stable sort will do. Counting-Sort is a good candidate since it's $\Theta(n+k)$.

What is the running time for sorting n d-digit numbers, where digits are in the range $[1..k]$?

$\Theta(d \cdot (n+k))$

Note:

- If d is constant and $k = O(n)$, then running time of Radix-Sort is $\Theta(n)$.
- In many applications, $d = \log_k(n)$, so the sort ends up being $\Theta(n \cdot \log_k(n))$ for a given k.