Lecture 13 – Count sort/ radix sort

Review

- Quicksort – through a linear partitioning algorithm

Today's topics: efficiency of quicksort, non-comparison based algorithm

Topic 1: efficiency of quicksort

The running time of quicksort depends on the partitioning of the subarrays:

- If the subarrays are balanced, then quicksort can run as fast as mergesort.
- If they are unbalanced, then quicksort can run as slowly as insertion sort.

Worst case

- Occurs when the subarrays are completely unbalanced.
- Have 0 elements in one subarray and $n-1$ elements in the other subarray.
- Get the recurrence

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \,. \end{aligned}$$

- Same running time as insertion sort.
- In fact, the worst-case running time occurs when quicksort takes a sorted array as input, but insertion sort runs in $O(n)$ time in this case.

Best case

- Occurs when the subarrays are completely balanced every time.
- Each subarray has $\leq n/2$ elements.
- Get the recurrence

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \,. \end{aligned}$$

Input size: # of items n

Q:/ Why is quick sort a good sorting algorithm since its worst case is the same as insertion sort?
A:/ It's because it's average case has very good properties.

Balanced partitioning

- Quicksort's average running time is much closer to the best case than to the worst case.
- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence

$$\begin{aligned} T(n) &\le T(9n/10) + T(n/10) + \Theta(n) \\ &= O(n \lg n) . \end{aligned}$$

- Intuition: look at the recursion tree.
  - It's like the one for $T(n) = T(n/3) + T(2n/3) + O(n)$ in Section 4.4.
  - Except that here the constants are different; we get $\log_{10} n$ full levels and $\log_{10/9} n$ levels that are nonempty.
  - As long as it's a constant, the base of the log doesn't matter in asymptotic notation.
  - Any split of constant proportionality will yield a recursion tree of depth $\Theta(\lg n)$.

The average case is T(n) =1.38nlogn and it can also be improved by 20-25%.

Q:/ How can we avoid the worst case scenario?
A:/ avoid sorted array → randomly pick the pivot!!

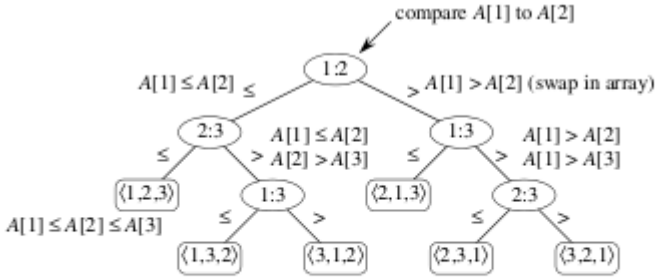
Topic 2: non-comparison based sorting algorithms
In chapter 8, we will first study how fast can we sort.

Q:/ how fast can we sort?
A:/ It depends

Quicksort – nlgn if we randomize the pivot
heapsort – nlgn always
merge sort – nlgn
insertion – n^2

We can say we need at least Ω(n) to examine all the elements in the list.
Q:/ Can we do better than nlgn?
A:/ For comparison based sorting,
the only operation that may be used to gain order information about a sequence is comparison of a pair of elements (<, >, ==, !=, >=, <=), no other operations like add or multiply.
All sorts seen so far are comparison based.

Decision tree model
It is an abstraction of any comparison sort. We only counts the number of comparisons, nothing else (such as data examination etc)
E.g. sort <a1, a2, a3>
left side is <= and right side is >. each non-leaf node has a pair of i, j where it means we compare ai with aj.



*[Each internal node is labeled by indices of array elements **from their original positions**. Each leaf is labeled by the permutation of orders that the algorithm determines.]*

example as 9, 4, and 6.

In general

How many leaves on the decision tree? There are $\geq n!$ leaves, because every permutation appears at least once.

For any comparison sort,

- 1 tree for each n.
- View the tree as if the algorithm splits in two at each node, based on the information it has determined up to that point.
- The tree models all possible execution traces.

What is the length of the longest path from root to leaf?

- Depends on the algorithm
- Insertion sort: $\Theta(n^2)$
- Merge sort: $\Theta(n \lg n)$

Now let's prove the lower bound of comparison based sorting algorithms

Lemma

Any binary tree of height h has $\leq 2^h$ leaves.

In other words:

- l = # of leaves,
- h = height,
- Then $l \leq 2^h$.

Theorem: any decision tree that sorts n elements has a height of Ω(nlgn)
Reason we are interested in the height is because that is the worst case scenario.

Proof:
# of leaves >=n! Because every permutation is possible and a correct algorithm must be able to cover all possible permutations.
Height of the tree h → # of leaves <=2^h
Thus n!<=2^h → h>=lg(n!) since lg is an increasing function.
Using stirling's approximation, then lg(n!)>=lg$(n/e)^n$--> nlg(n/e)=Ω(nlgn)
Thus proved.

This means merge sort, heap sort, and randomized quicksort are asymptotically optimal among comparison algorithms.

Topic 3: linear time sorting
Assume that elements are integers with a certain range. We will introduce 2 algorithms.

Counting sort

Depends on a *key assumption*: numbers to be sorted are integers in $\{0, 1, \ldots, k\}$.

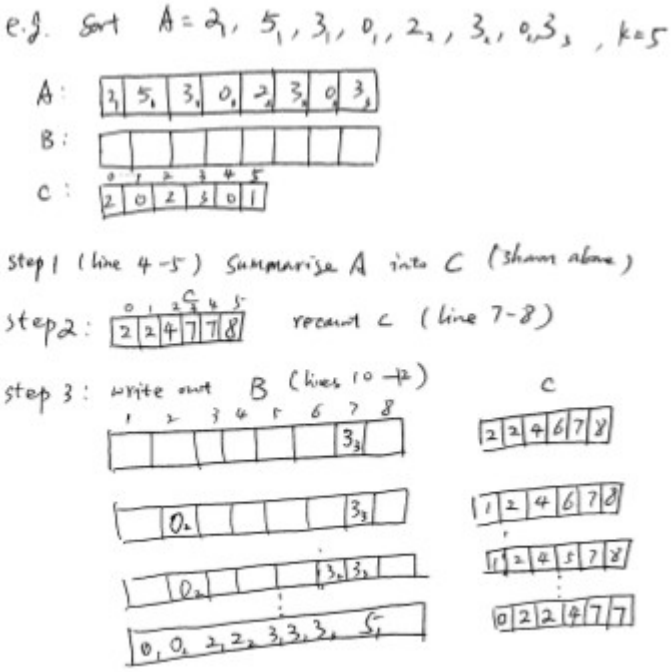
Input: $A[1..n]$, where $A[j] \in \{0, 1, \ldots, k\}$ for $j = 1, 2, \ldots, n$. Array A and values n and k are given as parameters.

Output: $B[1..n]$, sorted. B is assumed to be already allocated and is given as a parameter.

Auxiliary storage: $C[0..k]$

```
COUNTING-SORT(A, B, n, k)
  let C[0..k] be a new array
  for i = 0 to k
      C[i] = 0
  for j = 1 to n
      C[A[j]] = C[A[j]] + 1
  for i = 1 to k
      C[i] = C[i] + C[i − 1]
  for j = n downto 1
      B[C[A[j]]] = A[j]
      C[A[j]] = C[A[j]] − 1
```

What this algorithm does is to first summarize the input array and then distribute the elements in the array into the right places in the output array.



Q:/ What's the worst case scenario?
A:/ there is really no worst case since there is no if. First and third for loops take k and the other two for loops take n. Thus the running time is O(k+n).

if k <=n, then it is a good algorithm. Or else, if k is like n^2 or n^3, then it isn't such a good algorithm.
e.g. if you have 8 bit integers, then k=256. Thus for a regular n, it is a good algorithm. But if you have 32 bit integers, then you need n>=2^{32} which is about 4 billion. Thus our C array is like 16G bytes (4*4.2billion) to be initialized.

One important property of counting sort is stability (Stable sort preserves the relative order of equal elements.) - some of the sorting algorithms we learned are stable, some are not.

To handle large range of integers, radix sort uses count sort as a subroutine.

Radix sort is one of the oldest sorting algorithms (around 1900). There are punch cards with grids of 80 columns and 12 rows.

This is how IBM made its money. Punch card readers for census tabulation in early 1900's. It usually took about 10 years to analyze the data from the census. :)

Card sorters, worked on one column at a time. It's the algorithm for using the machine that extends the technique to multi-column sorting. The human operator was part of the algorithm!

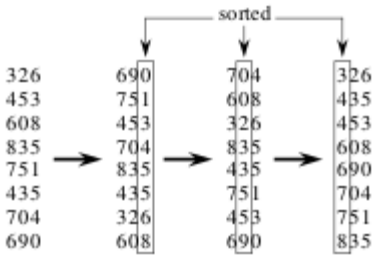
To summerize the problem, we have n numbers of d digits each.

Key idea: sort the least significant digit first.

To sort d digits:

RADIX-SORT(A, d)
for $i = 1$ **to** d
use a stable sort to sort array A on digit i

e.g.



Correctness can be inducted very easily.

- Assume digits $1, 2, \ldots, i-1$ are sorted.
- Show that a stable sort on digit i leaves digits $1, \ldots, i$ sorted:
  - If 2 digits in position i are different, ordering by position i is correct, and positions $1, \ldots, i-1$ are irrelevant.
  - If 2 digits in position i are equal, numbers are already in the right order (by inductive hypothesis). The stable sort on digit i leaves them in the right order.

This argument shows why it's so important to use a stable sort for intermediate sort.

Efficiency:
for count sort, it's efficiency is O(k+n). The running time is O(d(k+n))

But we only use the range in count sort, not the digits like in radix sort
Q:/ How to handle this?
A:/ If we have n integers each is b bits long, the range is $0 - 2^b-1$. We split each integer into b/r digits each digit is r bits long. Thus d=b/r and $k=2^r-1$. Thus for radix sort, it's efficiency O(b/r *(n+k))=O(b/r $(n+2^r)$). The min of this function has a min of b/lgn * n when r=lgn.

- If we choose $r < \lg n$, then $b/r > b/\lg n$, and $n + 2^r$ term doesn't improve.
- If we choose $r > \lg n$, then $n + 2^r$ term gets big. Example: $r = 2\lg n \Rightarrow 2^r = 2^{2\lg n} = (2^{\lg n})^2 = n^2$.

So, to sort 2^{16} 32-bit numbers, use $r = \lg 2^{16} = 16$ bits. $\lceil b/r \rceil = 2$ passes.

Compare radix sort to merge sort and quicksort:

- 1 million (2^{20}) 32-bit integers.
- Radix sort: $\lceil 32/20 \rceil = 2$ passes.
- Merge sort/quicksort: $\lg n = 20$ passes.
- Remember, though, that each radix sort "pass" is really 2 passes—one to take census, and one to move data.

How does radix sort violate the ground rules for a comparison sort?

- Using counting sort allows us to gain information about keys by means other than directly comparing 2 keys.
- Used keys as array indices.