CS 344: Design & Analysis of Algorithms

Lecture 6

Sep 19, 2019

- Median finding
- Counting sort
- Radix sort

Averages

In one study, 94% of college professors considered themselves to be above-average teachers.

("Not can, but will college teaching be improved?", K.P. Cross, 1977)

Averages

Suppose we rate teachers from 1 (worst) to 5 (best).

And suppose we have 100 teachers:

- 6 teachers are rated 1
- 94 teachers are rated 5

Average: $(6 \cdot 1 + 94 \cdot 5)/100 = 4.76$

So 94% of teachers are above average.

Averages

Averages are also very sensitive to outliers.

Suppose the test scores on midterm 1 are:

$$35, 36, 37, 38, 39, 40$$

Average: 37.5

But on midterm 2, one person did much better:

$$35, 36, 37, 38, 39, 100$$

Average: 47.5

Medians

The *median* of a set of numbers is a value such that half the numbers are greater than it and half are smaller than it.

For (sorted) finite sets with an odd number of elements, we can take the middle number.

- The median of $\{1, 3, 5, 10, 27\}$ is 5

Medians

The *median* of a set of numbers is a value such that half the numbers are greater than it and half are smaller than it.

For (sorted) finite sets with an even number of elements, we can take the smaller of the two middle numbers.

- The median of $\{2, 7, 8, 14, 23, 32\}$ is 8

(Note: some sources define it to be the average of the two middle numbers instead)

Medians

For a sorted set, finding the middle element(s) takes constant time:

```
median = A[n/2]
```

For unsorted finite sets, we could sort it first and then take the middle elements:

$$O(n \log n) + O(1) = O(n \log n)$$

But can we do better?

Medians

It's useful to consider a more general problem:

Given a list of numbers S (with n elements) and a an integer $k \in [1..n]$, find the kth smallest element of S.

Then when $k = \lfloor n/2 \rfloor$, we get the median.

Medians

Suppose this is our S array ($n = 11$):

2	36	5	21	8	13	11	20	5	4	1

Medians

Let's pick some v and split the list into three new lists:

- Values less than v (call this S_L)
- Values equal to v (call this S_v)
- Values greater than v (call this S_R)

Medians

The original array, S:

2	36	5	21	8	13	11	20	5	4	1

If we pick $v = 5$, we split S into these new lists:

S_L :

2	4	1

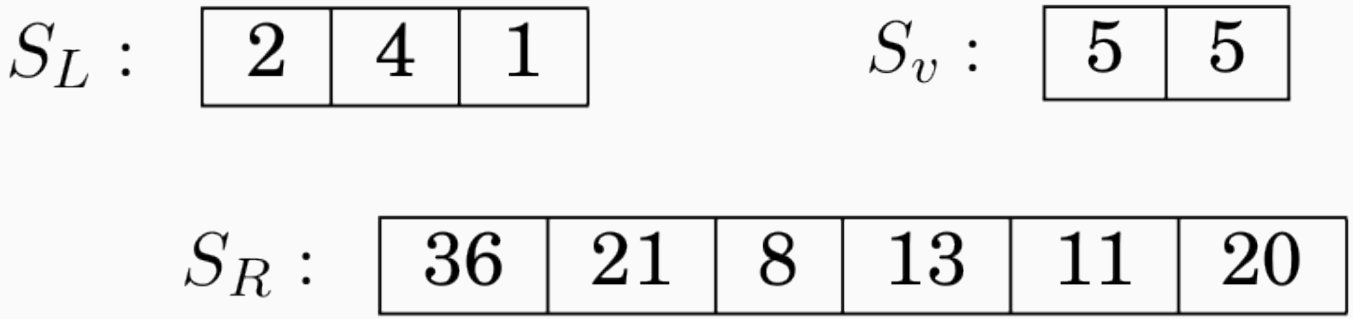
S_v :

5	5

S_R :

36	21	8	13	11	20

Medians



Then suppose we wanted the 8th smallest value. Where must it be?

Medians

So we can recursively search one of these three new arrays based on k and the three sizes:

$$\textbf{selection}(S,k) = \begin{cases} \textbf{selection}(S_L,k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ \textbf{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

Medians

So we can recursively search one of these three new arrays based on k and the three sizes:

```
def selection(S, k):
    # partition S into SL, SV, SR
    if k < len(SL):
        return selection(SL, k)
    elif k < len(SL) + len(SV):
        return v
    else:
        return selection(SR, k - len(SL) - len(SV))
```

Medians

What's the running time?

- Partitioning S: $O(n)$
- Recursively search, but how big are S_L and S_R?

Medians

If we can pick a v that splits the list evenly:

$$T(n) = T(n/2) + O(n)$$

which is linear.

But that requires that we pick v to be the median, which is back where we started!

Medians

Idea: pick v randomly.

Medians

In the worst case, v is the max/min value, so we get $T(n) = T(n-1) + O(n)$, which is $O(n^2)$.

In the best case, v is the median, and we get $O(n)$.

What is the average time?

Medians

Call v *good* if it's between the 25th and 75th percentile of S.

- Then S_L and S_R have sizes at most $\frac{3}{4}n$.
- And there are many such values!
- Half the values in S would work!

Medians

How many values of v would we have to pick before we get a good one?

Since the probability is $1/2$, the expected number is 2:

- We must pick at least one
- If it's bad, we need to pick again
- So $E = 1 + \frac{1}{2}E$
- Which gives $E = 2$

Medians

Hence after two splits on average, S shrinks to $3/4$ its size.

Then the *expected* running time is:

$$T(n) = T(3n/4) + O(n)$$

Medians

Why?

$$
\begin{aligned}
T(n) &= T(3n/4) + \text{Time to shrink } S \text{ to size } \leq 3n/4 \\
E[T(n)] &= E[T(3n/4) + \text{Time to shrink } S \text{ to size } \leq 3n/4] \\
E[T(n)] &= E[T(3n/4)] + E[\text{Time to shrink } S \text{ to size } \leq 3n/4] \\
E[T(n)] &= E[T(3n/4)] + E[2 \cdot O(n)] \\
E[T(n)] &= E[T(3n/4)] + O(n)
\end{aligned}
$$

Medians

The *expected* running time is:

$$T(n) = T(3n/4) + O(n)$$

Which is $O(n)$

Counting sort

We said there's a $n \log n$ lower bound for sorting based on comparisons.

Suppose we could guarantee all input values are in the range $[0, k]$.

Could we use that to do better than $n \log n$?

Counting sort

The idea behind counting sort:

- For each element x of the input,
- figure out how many elements are less than x.
- That tells us where x belongs in the sorted array.

For example, if there are 17 elements smaller than x, then x should be placed at $A[18]$.

Counting sort

We need three arrays:

- $A[1..n]$: input array
- $B[1..n]$: sorted output array
- $C[1..k]$: temporary storage

Counting sort

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

Counting sort

Suppose we're given the following input:

2	5	3	0	2	3	0	3

Counting sort

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

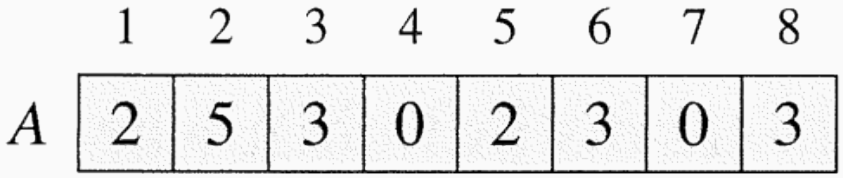
After line 5:

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

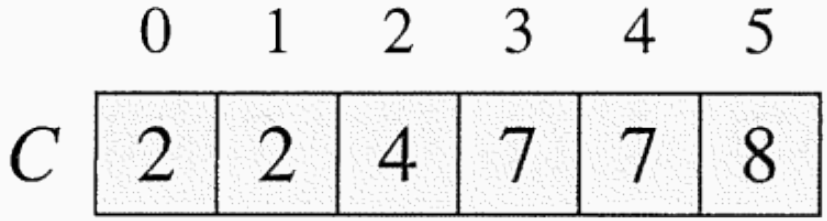
	0	1	2	3	4	5
C	2	0	2	3	0	1

Counting sort

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

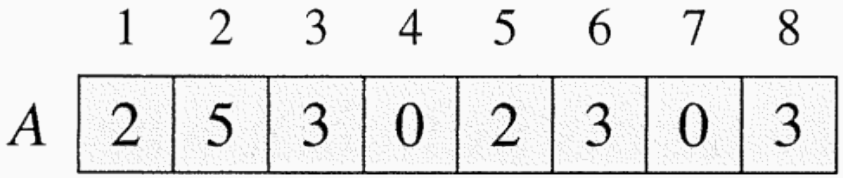


After line 8:

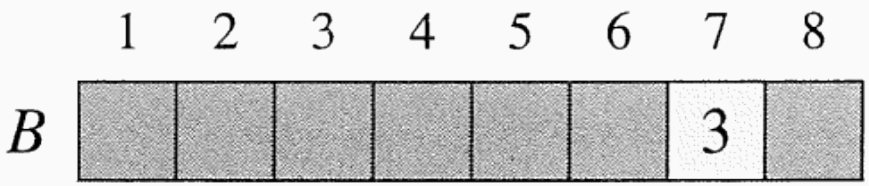


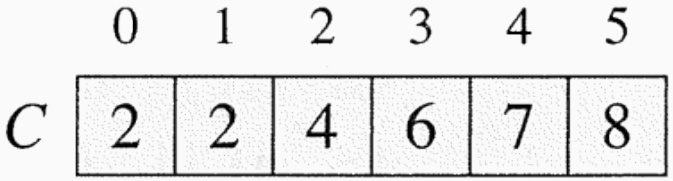
Counting sort

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```



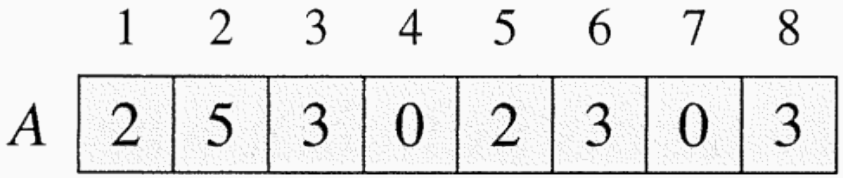
After one iteration of lines 10–12:



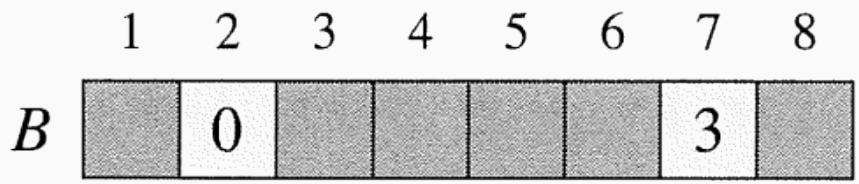


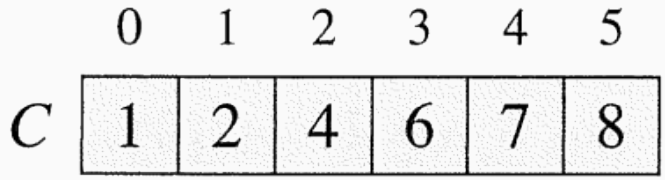
Counting sort

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```



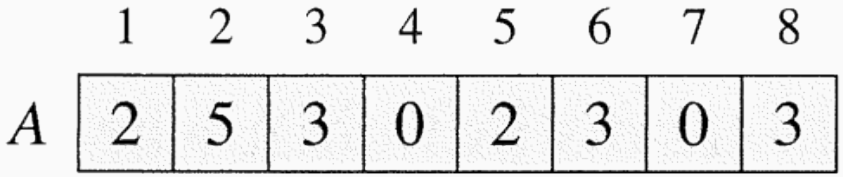
After two iterations of lines 10–12:



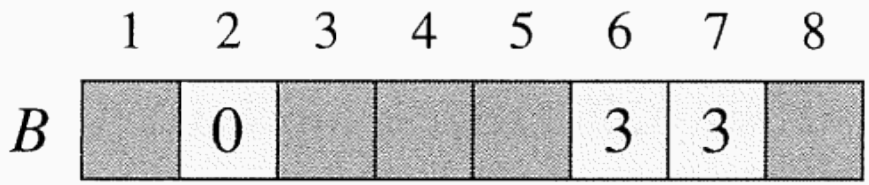


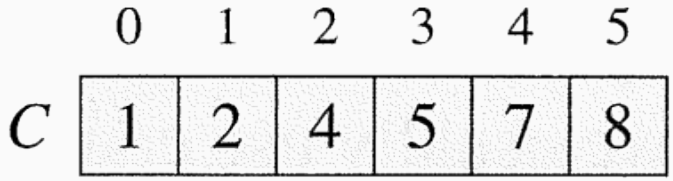
Counting sort

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```



After three iterations of lines 10–12:





Counting sort

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

The final sorted array B:

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

Counting sort

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

How much time does this take?

- for loop on lines 2–3: $\Theta(k)$
- for loop on lines 4–5: $\Theta(n)$
- for loop on lines 7–8: $\Theta(k)$
- for loop on lines 10–12: $\Theta(n)$

Total: $\Theta(k + n)$

(Usually $k = O(n)$, in which case this is $\Theta(n)$)

Stable sorts

Counting sort is a *stable* sort:

- If two elements have the same value, their order is preserved.
- If $i < j$ and $A[i] = A[j]$, then $\pi(i) < \pi(j)$

This is often important when sorting complex objects on a single key, or doing multiple rounds of sorting:

- For example, people by last name, then by first name

Stable sorts

Stable:

- Merge sort
- Counting sort
- Insertion sort

Unstable (as we've defined them):

- Selection sort
- Quicksort

Radix sort

Suppose we have n numbers to sort, each d-digits long.

Idea of radix sort:

- Sort by one column of digits at a time

Radix sort

Should we start from the most significant digit (MSD), on the left?

Or from the least significant digit (LSD), on the right?

Either can work, but it requires some care.

Radix sort

Suppose we have digits $1, 2, \ldots, 9, 10$ in some order:

$$
\begin{array}{cc}
2 & \\
7 & \\
4 & \\
8 & \\
1 & \\
1 & 0 \\
6 & \\
9 & \\
3 & \\
5 &
\end{array}
$$

Radix sort

If we do lexicographic sorting on the MSD:

1	
1	0
2	
3	
4	
5	
6	
7	
8	
9	

Radix sort

If we do lexicographic sorting on the LSD:

1
2
3
4
5
6
7
8
9
1 0

Radix sort

Let's sort from the LSD!

- Sort by rightmost column
- Then sort by 2nd rightmost column
- ...

But would it be correct?

Radix sort

Say we have 121, 122, and 782:

$$\begin{matrix} 7 & 8 & 2 \\ 1 & 2 & 2 \\ 1 & 2 & 1 \end{matrix}$$

Radix sort

Sort on the one's column:

$$\begin{matrix} 7 & 8 & 2 \\ 1 & 2 & 2 \\ 1 & 2 & 1 \end{matrix} \quad \rightarrow \quad \begin{matrix} 1 & 2 & 1 \\ 7 & 8 & 2 \\ 1 & 2 & 2 \end{matrix}$$

Radix sort

Sort on the ten's column:

$$\begin{matrix} 1 & 2 & 1 \\ 7 & 8 & 2 \\ 1 & 2 & 2 \end{matrix} \quad \rightarrow \quad \begin{matrix} 1 & 2 & 2 \\ 1 & 2 & 1 \\ 7 & 8 & 2 \end{matrix}$$

Radix sort

Sort on the hundred's column:

$$\begin{matrix} 1 & 2 & 2 \\ 1 & 2 & 1 \\ 7 & 8 & 2 \end{matrix} \quad \rightarrow \quad \begin{matrix} 1 & 2 & 2 \\ 1 & 2 & 1 \\ 7 & 8 & 2 \end{matrix}$$

But the final result isn't in the right order!

Radix sort

Something went wrong during the ten's column:

$$\begin{matrix} 1 & 2 & 1 \\ 7 & 8 & 2 \\ 1 & 2 & 2 \end{matrix} \quad \rightarrow \quad \begin{matrix} 1 & 2 & 2 \\ 1 & 2 & 1 \\ 7 & 8 & 2 \end{matrix}$$

Radix sort

Let's sort from the LSD!

- Sort by rightmost column
- Then sort by 2nd rightmost column
- ...

But... use a stable sort for the columns!

Radix sort

Sort on the one's column:

$$\begin{matrix} 7 & 8 & 2 \\ 1 & 2 & 2 \\ 1 & 2 & 1 \end{matrix} \quad \rightarrow \quad \begin{matrix} 1 & 2 & 1 \\ 7 & 8 & 2 \\ 1 & 2 & 2 \end{matrix}$$

Radix sort

Sort on the ten's column:

$$\begin{matrix} 1 & 2 & 1 \\ 7 & 8 & 2 \\ 1 & 2 & 2 \end{matrix} \quad \rightarrow \quad \begin{matrix} 1 & 2 & 1 \\ 1 & 2 & 2 \\ 7 & 8 & 2 \end{matrix}$$

Radix sort

Sort on the hundred's column:

$$\begin{matrix} 1 & 2 & 1 \\ 1 & 2 & 2 \\ 7 & 8 & 2 \end{matrix} \quad \rightarrow \quad \begin{matrix} 1 & 2 & 1 \\ 1 & 2 & 2 \\ 7 & 8 & 2 \end{matrix}$$

Radix sort

Claim: Radix sorting from LSD does give the correct order.

Proof: by induction on the columns

- For a single column, the order is correct
- Suppose the $k-1$ columns to the right have been sorted, and we sort column k
  - If two values in column k are different, they will be placed in the correct relative order
    - E.g., the hundred's column for 146 vs. 246
  - If two values in column k are the same, their order will be preserved
    - (and the earlier $k-1$ columns are in the right order, by the induction hypothesis)
    - E.g., the hundred's column for 146 vs. 136

Radix sort

A slightly larger example:

Radix sort

How long does it take to radix sort n d-digit numbers?

- The range of values in each column is bounded, so we can use counting sort
- One pass over n d-digit numbers then takes $\Theta(n + k)$
- We must do d passes
- Total time: $\Theta(d(n + k))$

If d is a constant, and $k = O(n)$, then this is again linear.

Summary

- Some problems have poor worst case performance, but good average case performance:
  - Finding the median via selection (quickselect)
- Sorts may be stable or unstable
- We have a $n \log n$ lower bound for comparison-based sorting, but we can get down to linear time in some situations!
  - Counting sort: $\Theta(k + n)$
  - Radix sort: $\Theta(d(n + k))$