

CSC 611: Analysis of Algorithms

Lecture 7

Medians and Order Statistics

Medians and Order Statistics

Def.: The i -th **order statistic** of a set of n elements is the i -th smallest element.

- The minimum of a set of elements:
 - The first order statistic $i = 1$
- The maximum of a set of elements:
 - The n -th order statistic $i = n$
- The median is the “halfway point” of the set
 - $i = (n+1)/2$, is unique when n is odd
 - $i = \lfloor (n+1)/2 \rfloor = n/2$ (lower median) and $\lceil (n+1)/2 \rceil = n/2 + 1$ (upper median), when n is even

Finding Minimum or Maximum

Alg.: MINIMUM(A, n)

min \leftarrow A[1]

for i \leftarrow 2 to n

do if min > A[i]

then min \leftarrow A[i]

return min

- How many comparisons are needed?
 - $n - 1$: each element, except the minimum, must be compared to a smaller element at least once
 - The same number of comparisons are needed to find the maximum
 - The algorithm is **optimal** with respect to the number of comparisons performed

CSC611/Lecture07

Simultaneous Min, Max

- Find min and max independently
 - Use $n - 1$ comparisons for each \Rightarrow total of **$2n - 2$**
- However, we can do better: at most **$3n/2$** comparisons
 - Process elements in pairs
 - Maintain the minimum and maximum of elements seen so far
 - Don't compare each element to the minimum and maximum separately
 - Compare the elements of a pair to each other
 - Compare the larger element to the maximum so far, and compare the smaller element to the minimum so far
 - This leads to only 3 comparisons for every 2 elements

CSC611/Lecture07

Analysis of Simultaneous Min, Max

- Setting up initial values:
 - n is odd: set both **min** and **max** to the first element
 - n is even: compare the first two elements, assign the smallest one to **min** and the largest one to **max**
- Total number of comparisons:
 - n is odd: we do $3(n-1)/2$ comparisons
 - n is even: we do 1 initial comparison + $3(n-2)/2$ more comparisons = $3n/2 - 2$ comparisons

CSC611/Lecture07

Example: Simultaneous Min, Max

- $n = 5$ (odd), array $A = \{2, 7, 1, 3, 4\}$
 1. Set **min** = **max** = 2
 2. Compare elements in pairs:
 - $1 < 7 \Rightarrow$ compare 1 with **min** and 7 with **max**
 \Rightarrow **min** = 1, **max** = 7 } 3 comparisons
 - $3 < 4 \Rightarrow$ compare 3 with **min** and 4 with **max**
 \Rightarrow **min** = 1, **max** = 7 } 3 comparisons

We performed: $3(n-1)/2 = 6$ comparisons

CSC611/Lecture07

Example: Simultaneous Min, Max

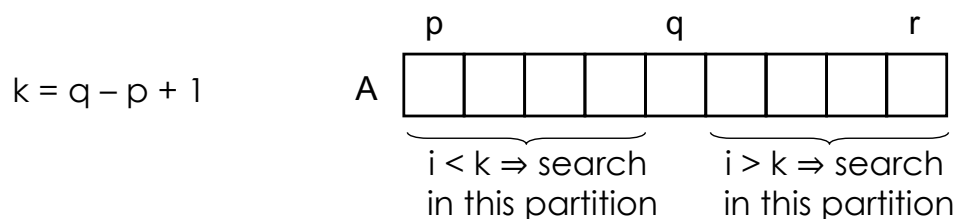
- $n = 6$ (even), array $A = \{2, 5, 3, 7, 1, 4\}$
 1. Compare 2 with 5: $2 < 5$ } 1 comparison
 2. Set **min** = 2, **max** = 5
 3. Compare elements in pairs:
 - $3 < 7 \Rightarrow$ compare 3 with **min** and 7 with **max**
 \Rightarrow **min** = 2, **max** = 7 } 3 comparisons
 - $1 < 4 \Rightarrow$ compare 1 with **min** and 4 with **max**
 \Rightarrow **min** = 1, **max** = 7 } 3 comparisons

We performed: $3n/2 - 2 = 7$ comparisons

CSC611/Lecture07

General Selection Problem

- Select the i -th order statistic (i -th smallest element) from a set of n distinct numbers



- Idea:
 - Partition the input array similarly with the approach used for Quicksort (use RANDOMIZED-PARTITION)
 - Recurse on one side of the partition to look for the i -th element depending on where i is with respect to the pivot
- We will show that selection of the i -th smallest element of the array A can be done in $\Theta(n)$ time

CSC611/Lecture07

Randomized Select

Alg.: RANDOMIZED-SELECT(A, p, r, i)

if $p = r$

then return $A[p]$

$$q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r)$$
$$k \leftarrow q - p + 1$$
if $i = k$

then return $A[q]$ ▷ pivot value is the answer

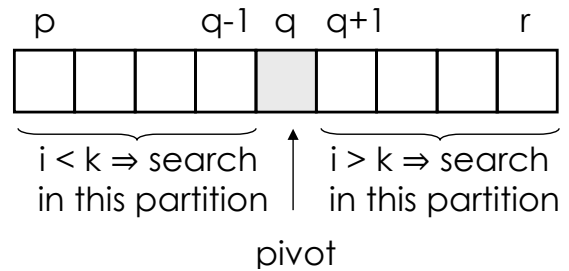
```
elseif i < k
```

```

then return RANDOMIZED-SELECT( $A, p, q-1, i$ )

```

```
else return RANDOMIZED-SELECT(A, q + 1, r, i-k)
```

RANDOMIZED-PARTITION(A, p, r)1 $i = \text{RANDOM}(p, r)$

```

2  exchange  $A[r]$  with  $A[i]$ 

```

```

3  return PARTITION( $A, p, r$ )

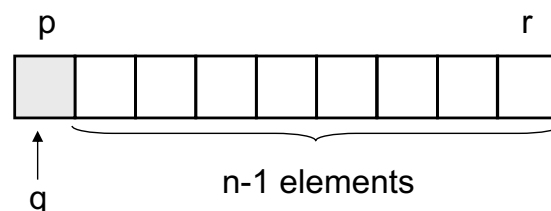
```

CSC611/Lecture07

Analysis of Running Time

- **Worst case** running time: $\Theta(n^2)$
 - If we always partition around the largest/smallest remaining element
 - Partition takes $\Theta(n)$ time
 - $T(n) = \Theta(1)$ (compute k) + $\Theta(n)$ (partition) + $T(n-1)$

$$= 1 + n + T(n-1) = \Theta(n^2)$$

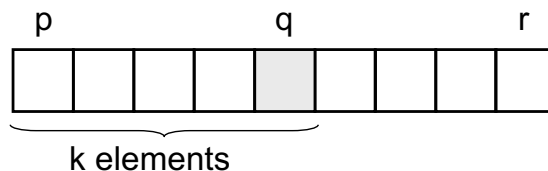


CSC611/Lecture07

Analysis of Running Time

- **Expected** running time (on **average**)

- Let $T(n)$ be a random variable denoting the running time of RANDOMIZED-SELECT



- RANDOMIZED-PARTITION is equally likely to return any element of A as the pivot \Rightarrow
- For each k such that $1 \leq k \leq n$, the subarray $A[p \dots q]$ has k elements (all \leq pivot) with probability $1/n$

CSC611/Lecture07

Analysis of Running Time

- When we call RANDOMIZED-SELECT we could have three situations:
 - The algorithm terminates with the answer ($i = k$), or
 - The algorithm recurses on the subarray $A[p..q-1]$, or
 - The algorithm recurses on the subarray $A[q+1..r]$
- The decision depends on where the i -th smallest element falls relative to $A[q]$
- To obtain an upper bound for the running time $T(n)$:
 - assume the i -th smallest element is always in the larger subarray

CSC611/Lecture07

Analysis of Running Time (cont.)

$$E[T(n)] = \underbrace{\text{Probability that } T(n) \text{ takes a value}} \times \underbrace{\text{The value of the random variable } T(n)}_{\text{Summed over all possible values}}$$

$$E[T(n)] = \frac{1}{n} [T(\max(0, n-1))] + \frac{1}{n} [T(\max(1, n-2))] + \dots + \frac{1}{n} [T(\max(n-1, 0))] + O(n)$$

↑
since select recurses only on the larger partition
↑
PARTITION

$$= \frac{1}{n} \left[T(n-1) + T(n-2) + T(n-3) \dots + T(n/2) \dots + T(n-3) + T(n-2) + T(n-1) \right] + O(n)$$

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} [T(k)] + O(n) \quad \mathbf{T(n) = O(n) \text{ (prove by substitution)}}$$

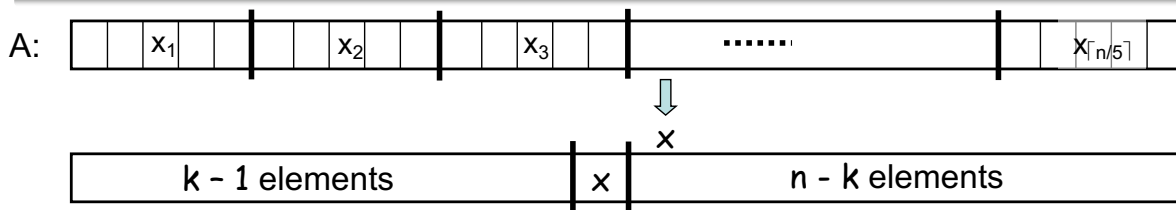
CSC611/Lecture07

A Better Selection Algorithm

- Can perform Selection in $O(n)$ Worst Case
- Idea: guarantee a good split on partitioning
 - Running time is influenced by how “balanced” are the resulting partitions
- Use a modified version of PARTITION
 - Takes as input the element around which to partition

CSC611/Lecture07

Selection in $O(n)$ Worst Case



1. Divide the n elements into groups of 5 $\Rightarrow \lceil n/5 \rceil$ groups
2. Find the median of each of the $\lceil n/5 \rceil$ groups
 - Use insertion sort, then pick the median
3. Use SELECT recursively to find the median x of the $\lceil n/5 \rceil$ medians
4. Partition the input array around x , using the modified version of PARTITION
 - There are $k-1$ elements on the low side of the partition and $n-k$ on the high side
5. If $i = k$ then return x . Otherwise, use SELECT recursively:
 - Find the i -th smallest element on the low side if $i < k$
 - Find the $(i-k)$ -th smallest element on the high side if $i > k$

CSC611/Lecture07

Example

- Find the 11th smallest element in the array:
 $A = \{12, 34, 0, 3, 22, 4, 17, 32, 3, 28, 43, 82, 25, 27, 34, 2, 19, 12, 5, 18, 20, 33, 16, 33, 21, 30, 3, 47\}$

1. Divide the array into groups of 5 elements

12	4	43	2	20	30
34	17	82	19	33	3
0	32	25	12	16	47
3	3	27	5	33	
22	28	34	18	21	

CSC611/Lecture07

Example (cont.)

2. Sort the groups and find their medians

0	4	25	2	20	3
3	3	27	5	16	30
12	17	34	12	21	47
34	32	43	19	33	
22	28	82	18	33	

3. Find the median of the medians

12, 12, 17, 21, 34, 30

CSC611/Lecture07

Example (cont.)

4. Partition the array around the median of medians (17)

First partition:

{12, 0, 3, 4, 3, 2, 12, 5, 16, 3}

Pivot:

17 (position of the pivot is $q = 11$)

Second partition:

{34, 22, 32, 28, 43, 82, 25, 27, 34, 19, 18,
20, 33, 33, 21, 30, 47}

To find the 6-th smallest element we would have to recurse our search in the first partition.

CSC611/Lecture07

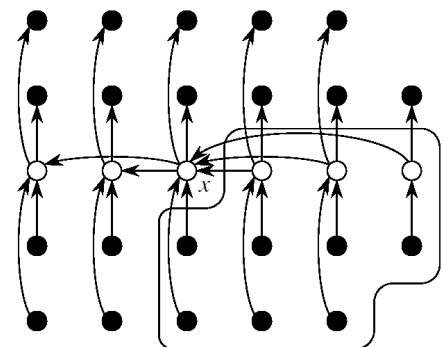
Analysis of Running Time

- Step 1: making groups of 5 elements takes $O(n)$
- Step 2: sorting $n/5$ groups in $O(1)$ time each takes $O(n)$
- Step 3: calling SELECT on $\lceil n/5 \rceil$ medians takes time $T(\lceil n/5 \rceil)$
- Step 4: partitioning the n -element array around x takes $O(n)$
- Step 5: recursion on one partition takes
depends on the size of the partition!!

CSC611/Lecture07

Analysis of Running Time

- First determine an upper bound for the sizes of the partitions
 - See how bad the split can be
- Consider the following representation
 - Each column represents one group of 5 (elements in columns are sorted)
 - Columns are sorted by their medians

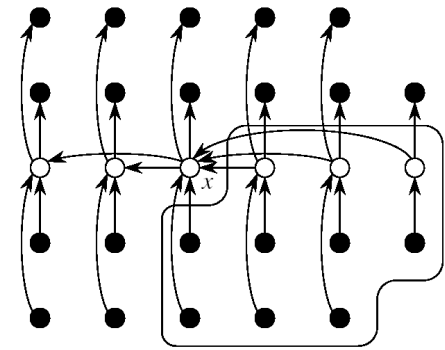


CSC611/Lecture07

Analysis of Running Time

- At least half of the medians found in step 2 are $\geq x$: $\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil$
- All but two of these groups contribute 3 elements $> x$

$$\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \text{ groups with 3 elements } > x$$



- At least $3 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$ elements greater than x
- SELECT is called on at most $n - \left(\frac{3n}{10} - 6 \right) = \frac{7n}{10} + 6$ elements

CSC611/Lecture07

Recurrence for the Running Time

- Step 1: making groups of 5 elements takes $O(n)$
- Step 2: sorting $n/5$ groups in $O(1)$ time each takes $O(n)$
- Step 3: calling SELECT on $\lceil n/5 \rceil$ medians takes time $T(\lceil n/5 \rceil)$
- Step 4: partitioning the n -element array around x takes $O(n)$
- Step 5: recursion on one partition takes time $\leq T(7n/10 + 6)$
- $T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$
- We can show that $T(n) = O(n)$

CSC611/Lecture07

Substitution

- $T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$

Show that $T(n) \leq cn$ for some constant $c > 0$ and all $n \geq n_0$

$$\begin{aligned} T(n) &\leq c \lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \\ &\leq cn \quad \text{if: } -cn/10 + 7c + an \leq 0 \end{aligned}$$

- $c \geq 10a(n/(n-70))$
 - choose $n_0 > 70$ and obtain the value of c

CSC611/Lecture07

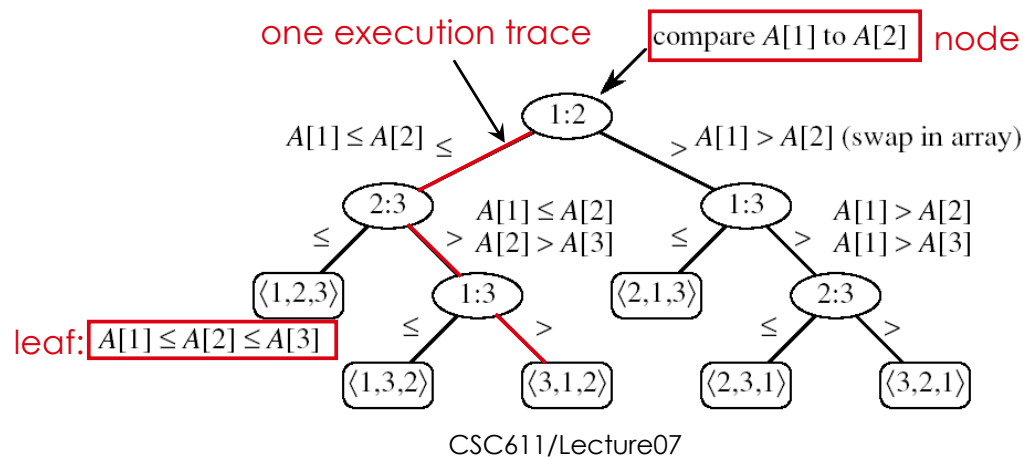
How Fast Can We Sort?

- Insertion sort, Bubble Sort, Selection Sort $\Theta(n^2)$
- Merge sort $\Theta(n \lg n)$
- Quicksort $\Theta(n \lg n)$
- What is common to all these algorithms?
 - These algorithms sort by making comparisons between the input elements
- To sort n elements, comparison sorts must make $\Omega(n \lg n)$ comparisons in the worst case

CSC611/Lecture07

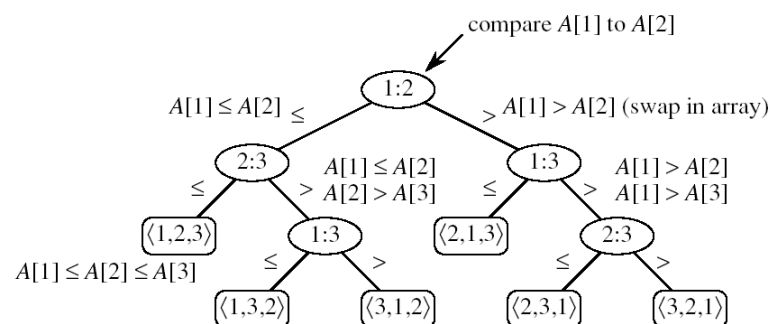
Decision Tree Model

- Represents the comparisons made by a sorting algorithm on an input of a given size: models all possible execution traces
- Control, data movement, other operations are ignored
- Count only the comparisons
- Decision tree for insertion sort on three elements:



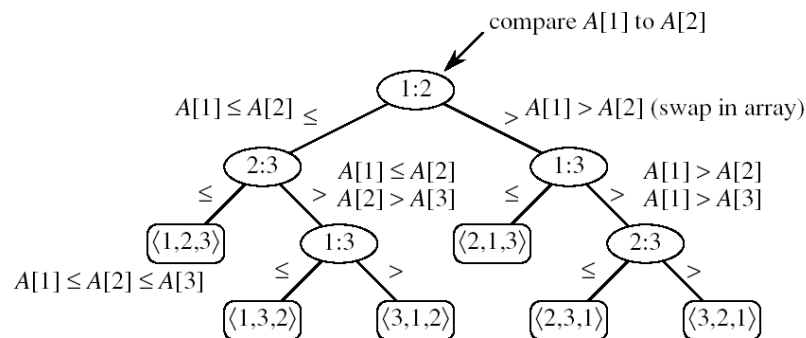
Decision Tree Model

- All permutations on n elements must appear as one of the leaves in the decision tree $n!$ permutations
- Worst-case number of comparisons
 - the length of the longest path from the root to a leaf
 - the height of the decision tree



Decision Tree Model

- Goal: finding a lower bound on the running time on any comparison sort algorithm
 - find a lower bound on the heights of all decision trees for all algorithms



CSC611/Lecture07

Lemma

- Any binary tree of height h has at most **2^h leaves**

Proof: induction on h

Basis: $h = 0 \Rightarrow$ tree has one node, which is a leaf

$$2^h = 1$$

Inductive step: assume true for $h-1$

- Extend the height of the tree with one more level
- Each leaf becomes parent to two new leaves

No. of leaves for tree of height $h =$

$$= 2 \times (\text{no. of leaves for tree of height } h-1)$$

$$\leq 2 \times 2^{h-1}$$

$$= 2^h$$

CSC611/Lecture07

Lower Bound for Comparison Sorts

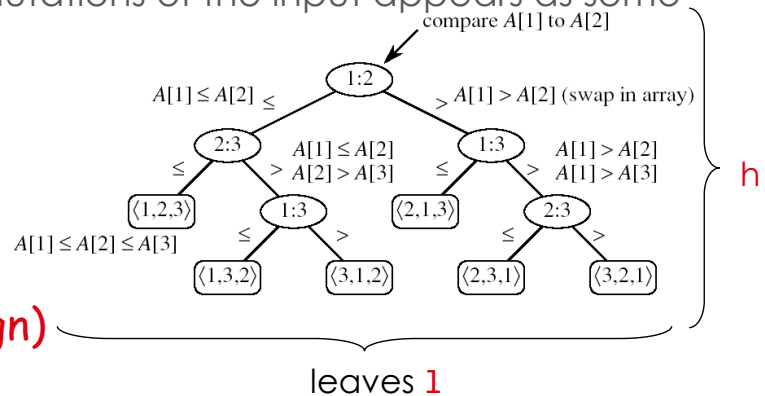
Theorem: Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof: How many leaves does the tree have?

- ℓ reachable leaves
- Each of the $n!$ permutations of the input appears as some leaf $\Rightarrow n! \leq \ell$
- At most 2^h leaves

$$\Rightarrow n! \leq \ell \leq 2^h$$

$$\Rightarrow h \geq \lg(n!) = \Omega(n \lg n)$$

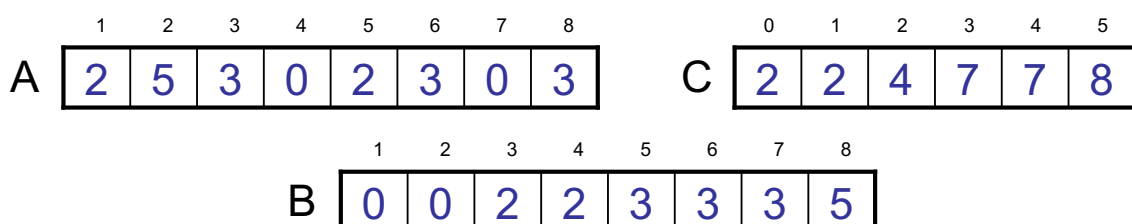


We can beat the $\Omega(n \lg n)$ running time if we use other operations than comparisons!

CSC611/Lecture07

Counting Sort

- Assumption:
 - The elements to be sorted are integers in the range 0 to k
- Idea:
 - Determine for each input element x , the number of elements smaller than x
 - Place element x into its correct position in the output array

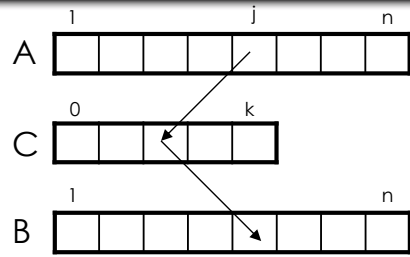


CSC611/Lecture07

COUNTING-SORT

Alg.: COUNTING-SORT(A, B, n, k)

1. **for** $i \leftarrow 0$ **to** k
2. **do** $C[i] \leftarrow 0$
3. **for** $j \leftarrow 1$ **to** n
4. **do** $C[A[j]] \leftarrow C[A[j]] + 1$
5. $\triangleright C[i]$ contains the number of elements equal to i
6. **for** $i \leftarrow 1$ **to** k
7. **do** $C[i] \leftarrow C[i] + C[i-1]$
8. $\triangleright C[i]$ contains the number of elements $\leq i$
9. **for** $j \leftarrow n$ **downto** 1
10. **do** $B[C[A[j]]] \leftarrow A[j]$
11. $C[A[j]] \leftarrow C[A[j]] - 1$



CSC611/Lecture07

Example

	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
---	---	---	---	---	---	---

	0	1	2	3	4	5
C	2	2	4	7	7	8

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

C	2	2	4	6	7	8
---	---	---	---	---	---	---

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

C	1	2	4	6	7	8
---	---	---	---	---	---	---

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

C	1	2	4	5	7	8
---	---	---	---	---	---	---

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

C	1	2	3	5	7	8
---	---	---	---	---	---	---

CSC611/Lecture07

Example (cont.)

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

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

C	0	2	3	5	7	8
---	---	---	---	---	---	---

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

C	0	2	3	4	7	7
---	---	---	---	---	---	---

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

C	0	2	3	4	7	8
---	---	---	---	---	---	---

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

CSC611/Lecture07

Analysis of Counting Sort

Alg.: COUNTING-SORT(A, B, n, k)

1. for $i \leftarrow 0$ to k } $\Theta(k)$
2. do $C[i] \leftarrow 0$ }
3. for $j \leftarrow 1$ to n } $\Theta(n)$
4. do $C[A[j]] \leftarrow C[A[j]] + 1$ }
5. ▷ $C[i]$ contains the number of elements equal to i
6. for $i \leftarrow 1$ to k } $\Theta(k)$
7. do $C[i] \leftarrow C[i] + C[i-1]$ }
8. ▷ $C[i]$ contains the number of elements $\leq i$
9. for $j \leftarrow n$ downto 1 } $\Theta(n)$
10. do $B[C[A[j]]] \leftarrow A[j]$ }
11. $C[A[j]] \leftarrow C[A[j]] - 1$ }

CSC611/Lecture07

Overall time: $\Theta(n + k)$

Analysis of Counting Sort

- Overall time: $\Theta(n + k)$
- In practice we use COUNTING sort when $k = O(n)$
 \Rightarrow running time is $\Theta(n)$
- Counting sort is **stable**
 - Numbers with the same value appear in the same order in the output array
 - Important when additional data is carried around with the sorted keys

CSC611/Lecture07

Radix Sort

- Considers keys as numbers in a base-k number
 - A d-digit number will occupy a field of d columns
- Sorting looks at one column at a time
 - For a d digit number, sort the least significant digit first
 - Continue sorting on the next least significant digit, until all digits have been sorted
 - Requires only d passes through the list

326
453
608
835
751
435
704
690

CSC611/Lecture07

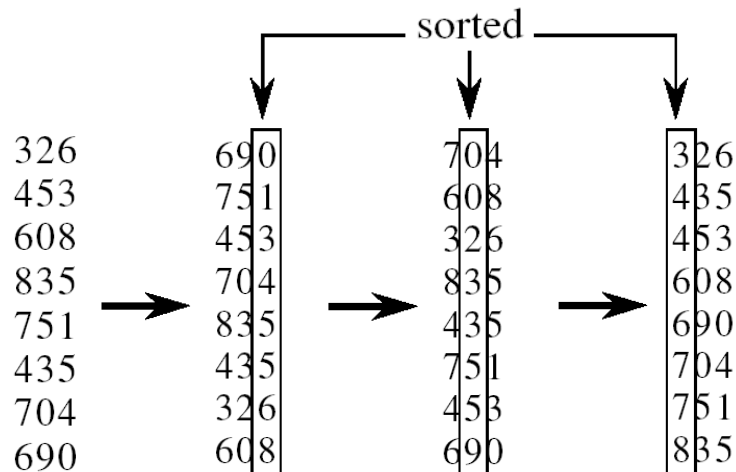
RADIX-SORT

Alg.: RADIX-SORT(A, d)

for $i \leftarrow 1$ **to** d

do use a stable sort to sort array A on digit i

- 1 is the lowest order digit, d is the highest-order digit



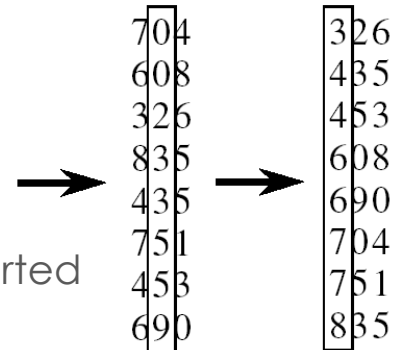
CSC611/Lecture07

Analysis of Radix Sort

- Given n numbers of d digits each, where each digit may take up to k possible values, RADIX-SORT correctly sorts the numbers in $\Theta(d(n+k))$
 - One pass of sorting per digit takes $\Theta(n+k)$ assuming that we use counting sort
 - There are d passes (for each digit)

Correctness of Radix sort

- We use induction on the number d of passes through the digits
- **Basis:** If $d = 1$, there's only one digit, trivial
- **Inductive step:** assume digits $1, 2, \dots, d-1$ are sorted
 - Now sort on the d -th digit
 - If $a_d < b_d$, sort will put a before b : correct
 $a < b$ regardless of the low-order digits
 - If $a_d > b_d$, sort will put a after b : correct
 $a > b$ regardless of the low-order digits
 - If $a_d = b_d$, sort will leave a and b in the same order and a and b are already sorted on the low-order $d-1$ digits



CSC611/Lecture07

Bucket Sort

- Assumption:
 - the input is generated by a random process that distributes elements uniformly over $[0, 1)$
- Idea:
 - Divide $[0, 1)$ into n equal-sized buckets
 - Distribute the n input values into the buckets
 - Sort each bucket
 - Go through the buckets in order, listing elements in each one
- **Input:** $A[1 \dots n]$, where $0 \leq A[i] < 1$ for all i
- **Output:** elements in A sorted
- **Auxiliary array:** $B[0 \dots n - 1]$ of linked lists, each list initially empty

CSC611/Lecture07

BUCKET-SORT

Alg.: BUCKET-SORT(A, n)

for $i \leftarrow 1$ **to** n

do insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$

for $i \leftarrow 0$ **to** $n - 1$

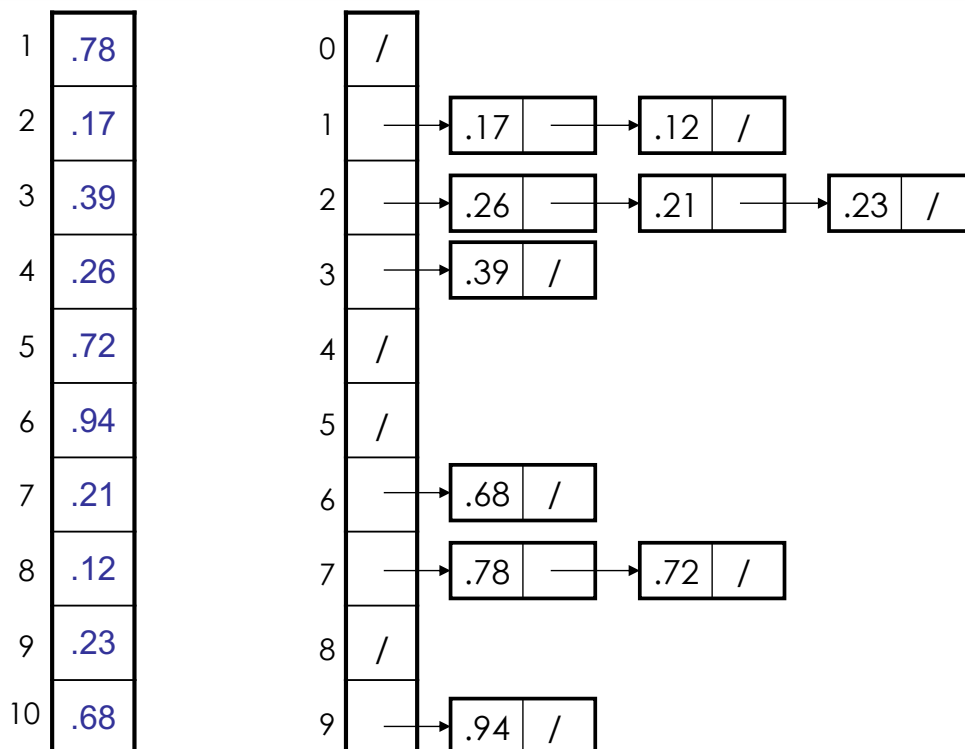
do sort list $B[i]$ with insertion sort

concatenate lists $B[0], B[1], \dots, B[n-1]$
together in order

return the concatenated lists

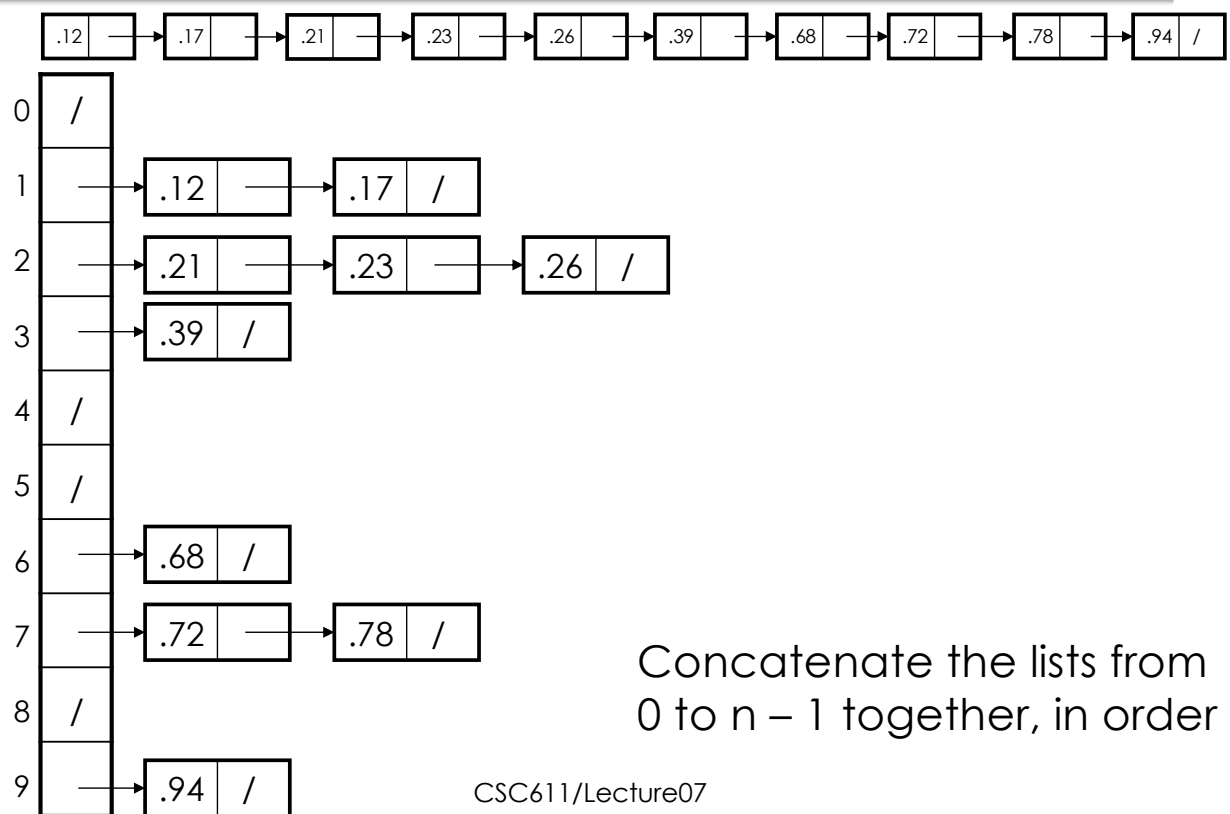
CSC611/Lecture07

Example - Bucket Sort



CSC611/Lecture07

Example - Bucket Sort



Correctness of Bucket Sort

- Consider two elements $A[i]$, $A[j]$
- Assume without loss of generality that $A[i] \leq A[j]$
- Then $\lfloor \log A[i] \rfloor \leq \lfloor \log A[j] \rfloor$
 - $A[i]$ belongs to the same group as $A[j]$ or to a group with a lower index than that of $A[j]$
- If $A[i]$, $A[j]$ belong to the same bucket:
 - insertion sort puts them in the proper order
- If $A[i]$, $A[j]$ are put in different buckets:
 - concatenation of the lists puts them in the proper order

Analysis of Bucket Sort

Alg.: BUCKET-SORT(A, n)

for $i \leftarrow 1$ to n	}	$O(n)$
do insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$		
for $i \leftarrow 0$ to $n - 1$	}	$\Theta(n)$
do sort list $B[i]$ with insertion sort		
concatenate lists $B[0], B[1], \dots, B[n-1]$	}	$O(n)$
together in order		
return the concatenated lists		

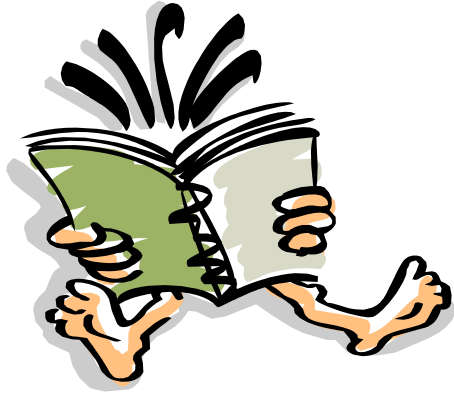
$\Theta(n)$

CSC611/Lecture07

Conclusion

- Any comparison sort will take at least $n \lg n$ to sort an array of n numbers
- We can achieve a better running time for sorting if we can make certain assumptions on the input data:
 - **Counting sort:** each of the n input elements is an integer in the range 0 to k
 - **Radix sort:** the elements in the input are integers represented with d digits
 - **Bucket sort:** the numbers in the input are uniformly distributed over the interval $[0, 1)$

Readings



- Chapter 6, 7, 8