Divide and Conquer
CISC5835, Algorithms for Big Data
CIS, Fordham Univ.

Instructor: X. Zhang

Acknowledgement

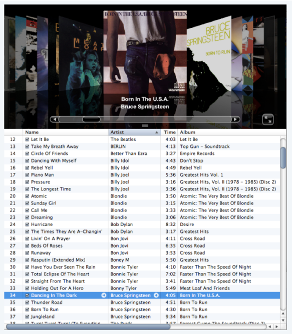
- The set of slides have use materials from the following resources
  - Slides for textbook by Dr. Y. Chen from Shanghai Jiaotong Univ.
  - Slides from Dr. M. Nicolescu from UNR
  - Slides sets by Dr. K. Wayne from Princeton
    - which in turn have borrowed materials from other resources
  - Other online resources

Outline

- Sorting problems and algorithms
- Divide-and-conquer paradigm
- Merge sort algorithm
- Master Theorem
  - recursion tree
- Median and Selection Problem
  - randomized algorithms
- Quicksort algorithm
- Lower bound of comparison based sorting

Sorting Problem

- Problem: Given a list of n elements from a totally-ordered universe, rearrange them in ascending order



Sorting applications

- Straightforward applications:
  - organize an MP3 library
  - Display Google PageRank results
  - List RSS news items in reverse chronological order
- Some problems become easier once elements are sorted
  - identify statistical outlier
  - binary search
  - remove duplicates
- Less-obvious applications
  - convex hull
  - closest pair of points
  - interval scheduling/partitioning
  - minimum spanning tree algorithms
  - …

Classification of Sorting Algorithms

- Use what operations?
  - Comparison based sorting: bubble sort, Selection sort, Insertion sort, Mergesort, Quicksort, Heapsort, …
  - Non-comparison based sort: counting sort, radix sort, bucket sort
- Memory (space) requirement:
  - in place: require O(1), O(log n) memory
  - out of place: more memory required, e.g., O(n)
- Stability:
  - stable sorting: elements with equal key values keep their original order
  - unstable sorting: otherwise

Stable vs Unstable sorting

- A STABLE sort preserves relative order of records with equal keys

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

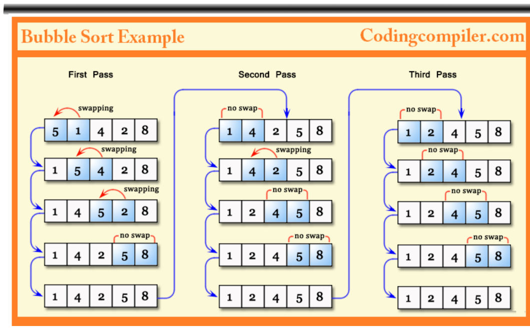
Sort file on second key:

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Furia	3	A	766-093-9873	22 Brown
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Gazsi	4	B	665-303-0266	113 Walker
Aaron	4	A	664-480-0023	097 Little

Records with key value 3 are not in order on first key!!

therefore, not stable!

Bubble Sort High-level Idea



Questions:
1. How many passes are needed?
2. No need to scan/process whole array on second pass, third pass…

Algorithm Analysis: bubble sort

Algorithm/Function.: bubblesort (a[1…n])
input: an array of numbers a[1…n]
output: a sorted version of this array

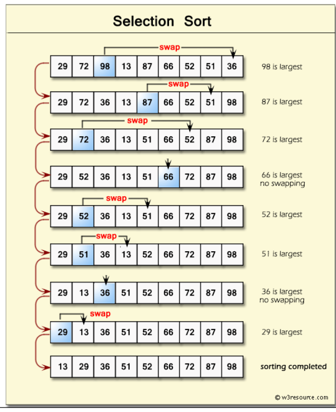
worst case: always swap
assume the <u>if … swap …</u>
takes c time/steps

```
for e=n-1 to 2:
    swapCnt=0;
    for j=1 to e:  //no need to scan whole list every time
            if a[j] > a[j+1]: swap (a[j], a[j+1]); swapCnt++;

        if (!swapCnt==0) //if no swap, then already sorted
            break;
return
```

- Memory requirement: memory used (note that input/output does not count)
- Is it stable?

Selection Sort: Idea



Running time analysis:

Insertion Sort

Insertion sort (Card game)	comparisons	data movements
8 5 7 1 9 3	1	<=2
5 8 7 1 9 3	2	<=3
5 7 8 1 9 3	3 (n - 3)*	<=4
1 5 7 8 9 3	1 (n - 2)*	<=5
1 5 7 8 9 3	5 (n - 1)*	<=6
1 3 5 7 8 9	0	<=7

Sorted list. Total comparisons = n (n - 1)/2 (worst case)* ~ O (n²)

Current element.

Inserted element.

O(n^2) Sorting Algorithms

- Bubble sort: O(n^2)
  - stable, in-place
- Selection sort: O(n^2)
  - Idea: find the smallest, exchange it with first element; find second smallest, exchange it with second, …
  - stable, in-place
- Insertion Sort: O(n^2)
  - idea: expand “sorted” sublist, by insert next element into sorted sublist
  - stable (if inserted after elements of same key), in-place
- asymptotically same performance
- selection sort is better: less data movement (at most n)

From quadric sorting algorithms to nlogn sorting algorithms
—- using divide and conquer

Divide-and-conquer paradigm

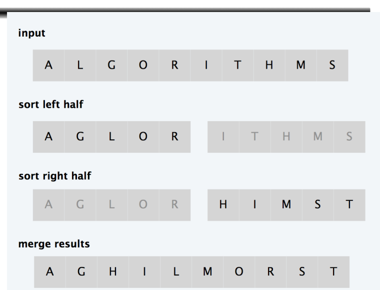
Divide-and-conquer.
- Divide up problem into several subproblems.
- Solve each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Most common usage.
- Divide problem of size n into two subproblems of size $n/2$ in linear time.
- Solve two subproblems recursively.
- Combine two solutions into overall solution in linear time.

Consequence.
- Brute force: $\Theta(n^2)$.
- Divide-and-conquer: $\Theta(n \log n)$.

MergeSort: think recursively!

1. recursively sort left half
2. recursively sort right half
3. merge two sorted halves to make sorted whole



"Recursively" means "following the same algorithm, passing smaller input"

Pseudocode for mergesort

mergesort (a[left … right])

if (left>=right) return; //base case, nothing to do

m = (left+right)/2;

mergeSort (a[left … m])

mergeSort (a[m+1 … right])

merge (a, left, m, right)

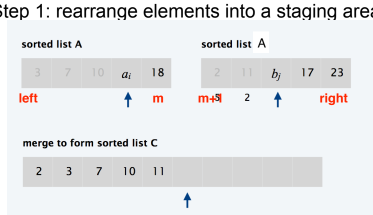
// given a[left…m], a[m+1 … right] are sorted:

// 1) first merge them one sorted array c[left…right]

// 2) copy c back to a

merge (A, left, m, right)

- Goal: Given A[left…m] and A[m+1…right] are each sorted, make A[left…right] sorted
  - Step 1: rearrange elements into a staging area (C)



  - Step 2: copy elements from C back to A
- T(n)=c*n   //let n=right-left+1
  - Note that each element is copied twice, at most n comparisons

Running time of MergeSort

- T(n): running time for MergeSort when sorting an array of size n
  - Input size n: the size of the array
- Base case: the array is one element long
  - $T(1) = C_1$
- Recursive case: when array is more than one element long
  - T(n)=T(n/2)+T(n/2)+O(n)
  - O(n): running time of merging two sorted subarrays
- What is T(n)?

Master Theorem

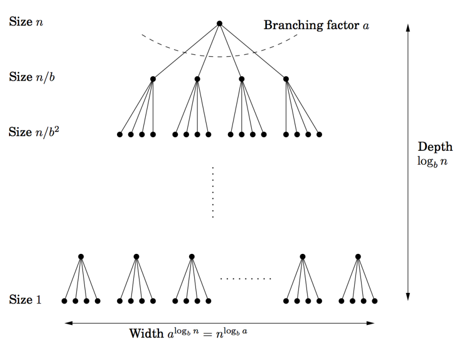
- If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants a>0, b>1, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d), & \text{if } d > log_b a \\ O(n^d logn), & \text{if } d = log_b a \\ O(n^{log_b a}), & \text{if } d < log_b a \end{cases}$$

- for analyzing divide-and-conquer algorithms
  - solve a problem of size n by solving a subproblems of size n/b, and using $O(n^d)$ to construct solution to original problem
- binary search: a=1, b=2, d=0 (case 2), T(n)=log n
- mergesort: a=2,b=2, d=1, (case 2), T(n)=O(nlogn)

Proof of Master Theorem

Figure 2.3 Each problem of size n is divided into a subproblems of size n/b.



Proof

- Assume n is a power of b, i.e., $n=b^k$
- size of subproblems decreases by a factor of b at each level of recursion, so it takes $k=\log_b n$ levels to reach base case
- branching factor of the recursion tree is a, so the i-th level has a^i subproblems of size n/b^i
- total work done at i-th level is:

$$a^i \times O(\frac{n}{b^i})^d = O(n^d) \times (\frac{a}{b^d})^i$$

- Total work done:

$$\sum_{i=0}^{log_b n} O(n^d) \times (\frac{a}{b^d})^i$$

Proof (2)

- Total work done: $\sum_{i=0}^{log_b n} O(n^d) \times (\frac{a}{b^d})^i$
- It's the sum of a geometric series with ratio $\frac{a}{b^d}$
  - if ratio is less than 1, the series is decreasing, and the sum is dominated by first term: $O(n^d)$
  - if ratio is greater than 1 (increasing series), the sum is dominated by last term in series,

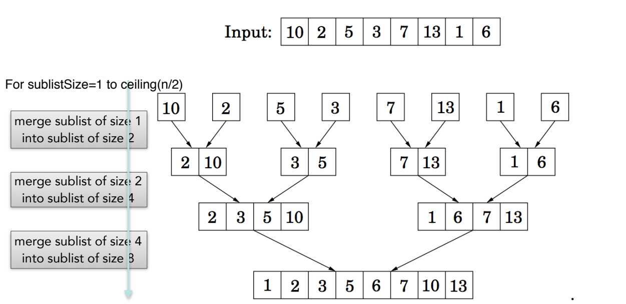
$$n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}.$$

  - if ratio is 1, the sum is $O(n^d \log_b n)$.
  - Note: see hw2 question for details

Iterative MergeSort

- Recursive MergeSort
  - pros: conceptually simple and elegant (language's support for recursive function calls helps to maintain status)
  - cons: overhead for function calls (allocate/ deallocate call stack frame, pass parameters and return value), hard to parallelize
- Iterative MergeSort
  - cons: coding is more complicated
  - pros: efficiency, and possible to take advantage of parallelism

Iterative MergeSort (bottom up)



Question: what if there are 9 elements? 10 elements?
pros: O(n) memory requirement; cons: harder to code (keep track starting/ending index of sublists)

MergeSort: high-level idea

```
function iterative-mergesort(a[1...n])
Input:  elements a_1, a_2, ..., a_n to be sorted

Q = [ ] (empty queue)        //Q stores the sublists to be merged
for i = 1 to n:
    inject(Q, [a_i])         //create sublists of size 1, add to Q
while |Q| > 1:
    inject(Q, merge(eject(Q), eject(Q)))
return eject(Q)
```

eject(Q): remove the front element from Q
inject(a): insert a to the end of Q
Pros: could be parallelized!
e.g., a pool of threads, each thread obtains two lists from Q to merge…
Cons: memory usage O(nlogn)

Outline

- Sorting problems and algorithms
- Divide-and-conquer paradigm
- Merge sort algorithm
- Master Theorem
  - recursion tree
- Median and Selection Problem
  - randomized algorithms
- Quicksort algorithm
- Lower bound of comparison based sorting

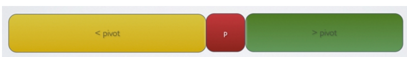
Find Median & Selection Problem

- median of a list of numbers: bigger than half of the numbers, and smaller than half of the numbers
  - A better summary than average value (which can be skewed by outliers)
- A straightforward way to find median
  - sort it first O(nlogn), and return elements in middle index
  - if list has even # of elements, then return average of the two in the middle
  - Can we do better?
    - we did more than needed by sorting…

Selection Problem

- More generally, Selection Problem: find K-th smallest element from a list S
- Idea:
  1. randomly choose an element p in S
  2. partition S into three parts:

S_L (those less than p) S_p (those equal to p) S_R (those greater than p)



|<— $|S_L|$ —>| S_p |<— $|S_R|$ —>|

  3. Recursively select from S_L or S_R as below

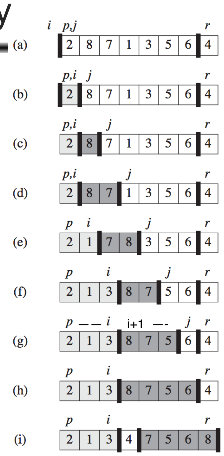
$$\text{selection}(S,k) = \begin{cases} \text{selection}(S_L,k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

Partition Array

PARTITION(A, p, r)

```
x = A[r]
i = p − 1              //i: wall
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

A[p…r]: i represents the wall
subarray p… i: less than x
subarray i+1…j-1: greater than x
subarray j…r: not yet processed



Selection Problem: Running Time

- T(n)=T(?)+O(n) //// linear time to partition

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

- How big is subproblem?
  - Depending on choice of p, and value of k
- Worst case: p is largest value, k is 1; or p is smallest value, k is k…,
  - $T(n)=T(n-1)+O(n)$ => $T(n)=n^2$
- As k is unknown, best case is to cut size by half
  - $T(n)=T(n/2)+O(n)$
  - By Master Theorem, T(n)=?

Selection Algorithm

- Observation: with deterministic way to choose pivot value, there will be some input that deterministically yield worst performance
  - if choose last element as pivot, input where elements in sorted order will yield worst performance
  - if choose first element as pivot, same
  - If choose 3rd element as pivot, what if the largest element is always in 3rd position
  - …

Randomized Selection Algorithm

- How to achieve good "average" performance in face of all inputs? (Answer: Randomization!)
  - Choose pivot element uniformly randomly (i.e., choose each element with equal prob)
  - Given any input, we might still choose "bad" pivot, but we are equally likely to choose "good" pivot
    - with prob. 1/2 the pivot chosen lies within 25% - 75% percentile of data, shrinking prob. size to 3/4 (which is good enough!)
    - in average, it takes 2 partition to shrink to 3/4
  - By Master Theorem, T(n)=O(n)

$$T(n) \leq T(3n/4) + O(n)$$

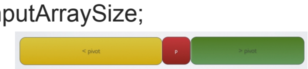
  - By Master Theorem: T(n)=O(n)

Randomized Selection Problem

1. randomly choose an element v in S:

   int index=random() % inputArraySize;

   v = S[index];

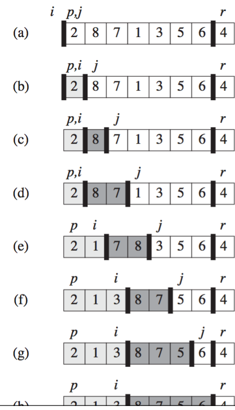


2. partition S into three parts: S_L (those less than v), S_v (those equal to v), S_R (those greater than v)
3. Recursively select from S_L or S_R as below

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

Partitioning array

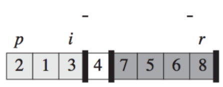
```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



p… i: less than x
i…j: greater than x
j…r: not yet processed

quicksort

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



Observation: after partition, pivot is in the correct sorted position
We now just need to sort left and right partition!

That's quicksort!

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p – 1)
        quicksort(A, p + 1, hi)
```


quicksort: running time

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p – 1)
        quicksort(A, p + 1, hi)
```

T(n)=O(n)+T(?) + T(?)

The size of subproblems depend on choice of pivot

- If pivot is smallest (or largest) element:
  $T(n)=T(n-1)+T(1)+O(n) \Rightarrow T(n)=O(n^2)$
- If pivot is median element:
  $T(n)=2T(n/2)+O(n) \Rightarrow T(n)=O(n \log n)$
- Similar to selection problem, choose pivot randomly

Outline

- Sorting problems and algorithms
- Divide-and-conquer paradigm
- Merge sort algorithm
- Master Theorem
  - recursion tree
- Median and Selection Problem
  - randomized algorithms
- Quicksort algorithm
- Lower bound of comparison based sorting

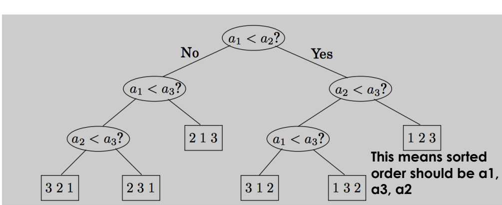
sorting: can we do better?

- MergeSort and quicksort both have average performance of O(n log n)
- quicksort performs better than merge sort
  - perform less copying of data
- Can we do better than this?
  - no, if sorting based on comparison operation (i.e., merge sort and quick sort are asymptotically optimal sorting algorithm)
  - if (a[i]>a[i+1]) in bubblesort
  - if (a[i]>max) in selection sort …

Lower bound on sorting

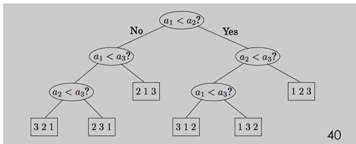
- Comparison based sorting algorithm as a decision tree:
  - leaves: sorting outcome (true order of array elements)
  - nodes: comparison operations
  - binary tree: two outcomes for comparison

Ex: Consider sorting an array of three elements a_1, a_2, a_3
  - number of possible true orders? (P(3,3)= 3!)
  - if input is 4,1,3, which path is taken? how about 2, 1, 5?



lower bound on sorting

- When sorting an array of n elements: $a_1, a_2, \ldots, a_n$
  - Total possible true orders is: n!
  - Decision tree is a binary tree with n! leaf nodes
  - Height of tree: worst case performance
- Recall: a tree of height n has at most 2^n leaf nodes
- So a tree with n! leaf nodes has height of at least $\log_2(n!)=O(n\log_2 n)$
- Any sorting algorithm must make in the worst case $\Omega(n\log_2 n)$ comparison.



Comment

- Such lower bound analysis applies to the problem itself, not to particular algorithms
  - reveal fundamental lower bound on running time, i.e., you cannot do better than this
- Can you apply same idea to show a lower bound on searching an sorted array for a given element, assuming that you can only use comparison operation?

Outline

- Sorting problems and algorithms
- Divide-and-conquer paradigm
- Merge sort algorithm
- Master Theorem
  - recursion tree
- Median and Selection Problem
  - randomized algorithms
- Quicksort algorithm
- Lower bound of comparison based sorting

Non-Comparison based sorting

- Example: sort 100 kids by their ages (taken value between 1-9)
  - comparison based sorting: at least nlogn
- Counting sort:
  - count how many kids are of each age
  - and then output each kid based upon how many kids are of younger than him/her
  - Running time: a linear time sorting algorithm that sort in O(n+k) time when elements are in range from 1 to k.

Counting Sort

COUNTING-SORT(A, B, k)

let $C[0..k]$ be a new array
for $i = 0$ **to** k
    $C[i] = 0$
for $j = 1$ **to** $A.length$
    $C[A[j]] = C[A[j]] + 1$
// $C[i]$ now contains the number of elements equal to i.
for $i = 1$ **to** k
    $C[i] = C[i] + C[i-1]$
// $C[i]$ now contains the number of elements less than or equal to i.
for $j = A.length$ **downto** 1
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$

Counting Sorting

```
For simplicity, consider the data in the range 0 to 9.
Input data: 1, 4, 1, 2, 7, 5, 2
  1) Take a count array to store the count of each unique object.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  2  0   1  1  0  1  0  0

  2) Modify the count array such that each element at each index
  stores the sum of previous counts.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  4  4  5  6  6  7  7  7

The modified count array indicates the position of each object in
the output sequence.

  3) Output each object from the input sequence followed by
  decreasing its count by 1.
  Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.
  Put data 1 at index 2 in output. Decrease count by 1 to place
  next data 1 at an index 1 smaller than this index.
```

Recall: stable vs unstable sorting

- A STABLE sort preserves relative order of records with equal keys

Sort file on first key:

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

Sort file on second key:

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Furia	3	A	766-093-9873	22 Brown
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Gazsi	4	B	665-303-0266	113 Walker
Aaron	4	A	664-480-0023	097 Little 6

Records with key value 3 are not in order on first key!!

therefore, not stable!

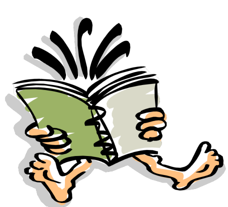
Radix Sorting

- What if range of value, k, is large?
- Radix Sort: sort digit by digit
  - starting from least significant digit to most significant digit, uses counting sort (or other stable sorting algorithm) to sort by each digit

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

What if unstable soring is used?

Readings



- Chapter 2