

CSCE 3110

Data Structures & Algorithms

- Sorting
- Reading: Chap.7, Weiss

Content

- Eight sorting algorithms
 - Comparison-based
 - Insertion sort
 - Selection sort
 - Heapsort
 - Merge sort
 - Quick sort
 - Lower Bounds for Sorting
 - Integer sorting
 - Bucket sort
 - Counting sort
 - Radix sort

Sorting

- Given a set (container) of n elements
 - e.g., array, set of words, etc.
- Suppose there is an order relation that can be set across the elements
- Goal: Arrange the elements in a certain order, e.g., ascending/descending orders

Start -> 1 23 2 56 9 8 10 100

End -> 1 2 8 9 10 23 56 100

Importance of Sorting

Why don't CS profs ever stop talking about sorting?

- Computers spend a lot of time sorting, historically 25% on mainframes.
- Sorting is the best studied problem in computer science, with many different algorithms known.
- Most of the interesting ideas we will encounter in the course can be taught in the context of sorting, such as divide-and-conquer, randomized algorithms, and lower bounds.

Stable Sorting

- A property of sorting
- If a sort guarantees the relative order of equal items stays the same, then it is a stable sort
- $[7_1, 6, 7_2, 5, 1, 2, 7_3, -5]$
 - subscripts added for clarity
- $[-5, 1, 2, 5, 6, 7_1, 7_2, 7_3]$
 - result of stable sort

In Place Sorting

- Sorting of a data structure does not require any external data structure for storing the intermediate steps
- The amount of extra space required to sort the data is constant with the input size.

Insertion Sort

- Insertion sort: orders a list of values by repetitively inserting a particular value into a sorted subset of the list
- More specifically:
 1. consider the first item to be a sorted sub list of length 1
 2. insert the second item into the sorted sub list, shifting the first item if needed
 3. insert the third item into the sorted sub list, shifting the other items as needed
 4. repeat until all values have been inserted into their proper positions

```
template <class Item>
void insertion_sort(Item data[ ], size_t n) {
    size_t i, j;
    Item temp;

    if(n < 2) return; // nothing to sort!!

    for(i = 1; i < n; ++i)
    {
        // take next item at front of unsorted part of array
        // and insert it in appropriate location in sorted part of array
        temp = data[i];
        for(j = i; data[j-1] > temp and j > 0; --j)
            data[j] = data[j-1]; // shift element forward

        data[j] = temp;
    }
}
```


Insertion Sort: Example

3 is sorted.
Shift nothing. Insert 9.



3 and 9 are sorted.
Shift 9 to the right. Insert 6.



3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 1.



1, 3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 2.



Insertion Sort Time Analysis

- In O -notation, what is:
 - Worst case running time for n items? $O(n^2)$
 - Best case running time for n items? $O(n)$
- Steps of algorithm:

for $i = 1$ to $n-1$

take next key from unsorted part of array

insert in appropriate location in sorted part of array:

for $j = i$ down to 0 ,

shift sorted elements to the right if $key > key[j]$

increase size of sorted array by 1

Outer loop:
 $O(n)$

Inner loop:
 $O(n)$

Stable and In-Place

Selection Sort

Basic idea:

- Find the smallest element in the array
- Exchange it with the element in the first position
- Find the second smallest element and exchange it with the element in the second position
- Continue until the array is sorted

Selection Sort

SELECTION-SORT(A):

$n \leftarrow \text{length}[A]$

 for $j \leftarrow 1$ to $n - 1$

 do $\text{smallest} \leftarrow j$

 for $i \leftarrow j + 1$ to n

 do if $A[i] < A[\text{smallest}]$

 then $\text{smallest} \leftarrow i$

 exchange $A[j] \leftrightarrow A[\text{smallest}]$

Selection Sort: Example

8	4	6	9	2	3	1
---	---	---	---	---	---	---

1	4	6	9	2	3	8
---	---	---	---	---	---	---

1	2	6	9	4	3	8
---	---	---	---	---	---	---

1	2	3	9	4	6	8
---	---	---	---	---	---	---

1	2	3	4	9	6	8
---	---	---	---	---	---	---

1	2	3	4	6	9	8
---	---	---	---	---	---	---

1	2	3	4	6	8	9
---	---	---	---	---	---	---

1	2	3	4	6	8	9
---	---	---	---	---	---	---

Selection Sort Time Analysis

It's clearly quadratic:

- The first pass, we search through exactly $n - 1$ elements (no difference between average-case and worst-case), then swap (constant time).
- Second time, $n - 2$ elements, then $n - 3$, etc.

We get (yet again!) the arithmetic sum — $\Theta(n^2)$

In-Place but not stable, [4a, 3, 4b, 1]

Heapsort

Sorting Strategy:

1. Build Max Heap from unordered array;
2. Find maximum element $A[1]$;
3. Swap elements $A[n]$ and $A[1]$: now max element is at the end of the array!
4. Discard node n from heap (by decrementing heap-size variable)
5. New root may violate max heap property, but its children are max heaps. Run `max_heapify` to fix this.
6. Go to Step 2 unless heap is empty.

Heapsort Time Analysis

- After n iterations the Heap is empty
- Every iteration involves a swap and a max_heapify operation;
- Hence it takes $O(n \log n)$ time overall
- In-place, but not stable sort

Divide and Conquer

Very important technique in algorithm design

1. Divide problem into smaller parts
2. Independently solve the simpler parts
 - Think recursion
 - Or potential parallelism
3. Combine solution of parts to produce overall solution

Two great sorting methods are fundamentally Divide-and-conquer

- Merge Sort
- Quick Sort

Merge Sort

- So simple — really, soooooo simple:
- Split the array into two halves.
 - Sort (using the same merge sort) the first half
 - Then, sort the second half
 - Then, merge them (since they are ordered sequence, it should be easy to merge them in linear time into a single ordered sequence... right?)

Merge Sort

- Merging two sorted sequences into a single sorted sequence (in linear time):

Example:

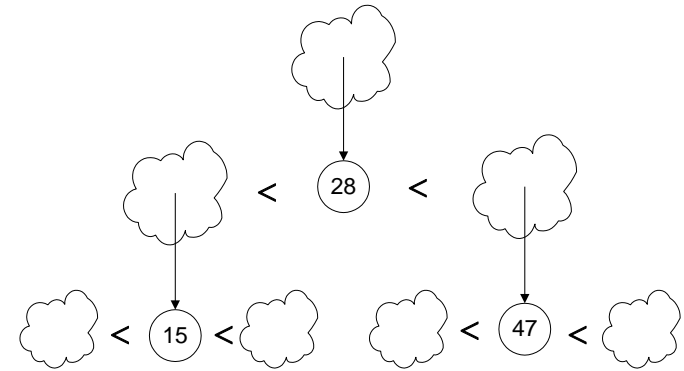
- Sequence A: 11, 23, 40, 57, 78, 93
- Sequence B: 5, 9, 35, 36, 39, 63

Merge Sort Time Analysis

- We already saw that the run time is $\Theta(n \log n)$
- Stable sort
- What about space? Can it be executed in-place?
 - Really not! The merge operation does require extra space.

Quick Sort

1. Pick a “pivot”.
2. Divide list into two lists:
 - One less-than-or-equal-to pivot value
 - One greater than pivot value
3. Sort each sub-problem recursively
4. Answer is the concatenation of the two solutions



Quick Sort: Example

Pick pivot:

7	2	8	3	5	9	6
---	---	---	---	---	---	---

Partition
with cursors

7	2	8	3	5	9	6
---	---	---	---	---	---	---



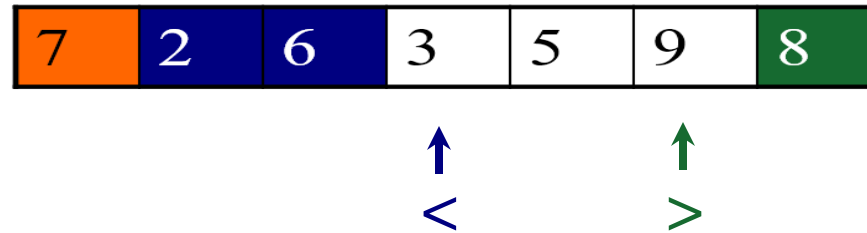
2 goes to
less-than

7	2	8	3	5	9	6
---	---	---	---	---	---	---



Quick Sort: Example

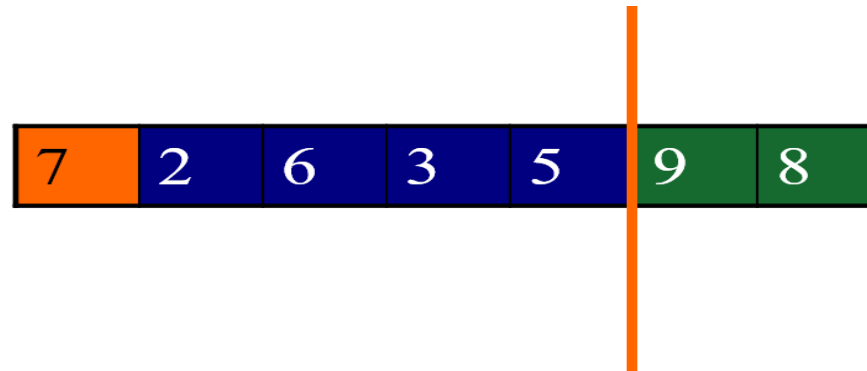
6, 8 swap
less/greater-than



3,5 less-than
9 greater-than

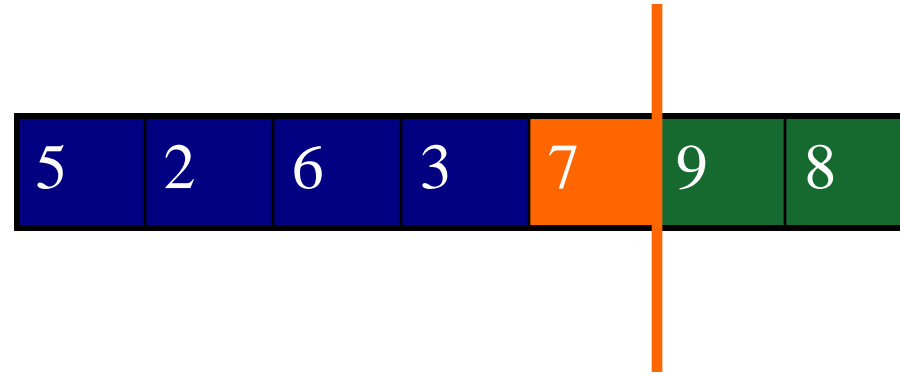


Partition done.

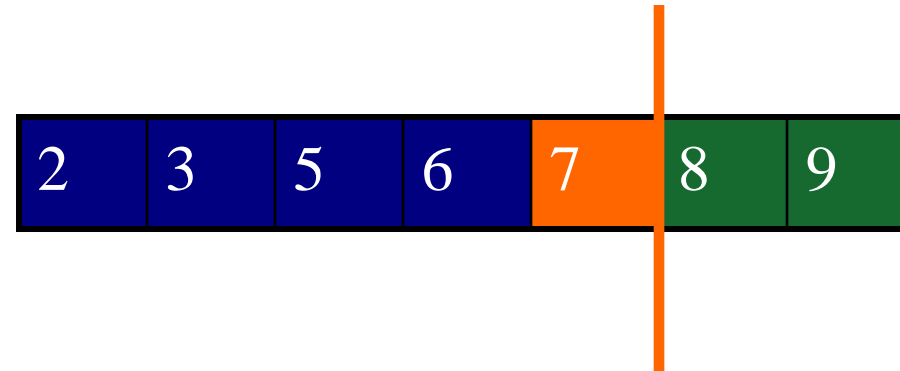


Quick Sort: Example

Put pivot
into final
position.



Recursively
sort each side.



Quick Sort

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

First element as pivot

```
algorithm partition(A, lo, hi):  
    pivot = A[lo]  
    i = lo  
    j = hi + 1  
    loop forever  
        do  
            i = i + 1  
            while A[i] < pivot  
                do  
                    j = j - 1  
            while A[j] > pivot  
                if i >= j then  
                    break  
            else  
                swap A[i] with A[j]  
    swap A[j] with A[lo]  
    return j
```

Quick Sort

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

Last element as pivot

```
algorithm partition(A, lo, hi):  
    pivot = A[hi]  
    i = lo - 1  
    j = hi  
    loop forever  
        do  
            i = i + 1  
            while A[i] < pivot  
                do  
                    j = j - 1  
                    while A[j] > pivot  
                        if i >= j then  
                            break  
                        else  
                            swap A[i] with A[j]  
    swap A[i] with A[hi]  
    return i
```

Quick Sort:

Lomuto partition

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

Example: 7 2 8 3 5 9 6

Last element as pivot

```
algorithm partition(A, lo, hi):  
    pivot = A[hi]  
    i = lo  
    for j = lo to hi - 1 do  
        if A[j] < pivot then  
            swap A[i] with A[j]  
            i = i + 1  
    swap A[i] with A[hi]  
    return i
```

Quick Sort:

Hoare partition

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

Example: 7 2 8 3 5 9 6

Middle element as pivot

```
algorithm partition(A, lo, hi):  
    pivot = A[(lo + hi) / 2]  
    i = lo - 1  
    j = hi + 1  
    loop forever  
        do  
            i = i + 1  
        while A[i] < pivot  
        do  
            j = j - 1  
        while A[j] > pivot  
    if i >= j then  
        return j  
    swap A[i] with A[j]
```

Quick Sort Time Analysis

- Picking pivot: constant time
- Partitioning: linear time
- Recursion: time for sorting left partition (say of size i) + time for right (size $N - i - 1$) + time to combine solutions

$$T(N) = T(i) + T(N - i - 1) + cN$$

where i is the number of elements smaller than the pivot

Quick Sort Time Analysis

- Quick Sort is fast in practice but has $\theta(N^2)$ worst-case complexity.

- Pivot is always smallest element, so $i = 0$:

$$T(N) = T(i) + T(N - i - 1) + cN$$

$$T(N) = T(N - 1) + cN$$

$$= T(N - 2) + c(N - 1) + cN$$

$$= T(N - k) + c \sum_{i=0}^{k-1} (N - i)$$

$$= O(N^2)$$

Quick Sort Best Case

Pivot is always middle element:

$$T(N) = T(i) + T(N - i - 1) + cN$$

$$T(N) = 2T\left(\frac{N-1}{2}\right) + cN$$

$$< 2T\left(\frac{N}{2}\right) + cN$$

$$< 4T\left(\frac{N}{4}\right) + c\left(2\frac{N}{2} + N\right)$$

$$< 8T\left(\frac{N}{8}\right) + cN(1 + 1 + 1)$$

$$< kT\left(\frac{N}{k}\right) + cN \log k = O(N \log N)$$

Dealing with Slow Quick Sort

- Randomly choose pivot
 - Good theoretically and practically, but call to random number generator can be expensive
- Pick pivot cleverly
 - “Median-of-3” rule takes Median(first, middle, last element elements) as pivot. Also works well.
 - e.g., Swap Median with either first or last element, then partition as usual.

Quick Sort Average Case

- Suppose pivot is picked at random from values in the list
- All the following cases are equally likely:
 - Pivot is smallest value in list
 - Pivot is 2nd smallest value in list
 - Pivot is 3rd smallest value in list
 - ...
 - Pivot is largest value in list
- Same is true if pivot is e.g., always first element, but the input itself is perfectly random

Quick Sort Avg Case, cont.

- Expected running time = sum of (time when partition size i) \times (probability partition is size i)
- In either random case, all size partitions are equally likely – probability is just $1/N$

$$T(N) = T(i) + T(N - i - 1) + cN$$
$$E(T(N)) = \sum_{i=0}^{N-1} \frac{1}{N} [T(i) + T(N - i - 1) + cN]$$

Solving this recursive equation (see Weiss pg. 320) yields:

$$E(T(N)) = O(N \log N)$$

Why is Quick Sort Faster than Merge Sort?

- Quick sort typically performs more comparisons than Merge sort, because partitions are not always perfectly balanced
 - Merge sort: $n \log n$ comparisons
 - Quick sort: $1.38n \log n$ comparisons on average
- Quick sort performs many fewer copies, because on average half of the elements are on the correct side of the partition – while Merge sort copies every element when merging
 - Merge sort: $2n \log n$ copies (using “temp array”)
 - Quick sort: $n/2 \log n$ copies on average

Sorting HUGE Data Sets

- US Telephone Directory:
 - 300,000,000 records
 - 64-bytes per record
 - Name: 32 characters
 - Address: 54 characters
 - Telephone number: 10 characters
 - About 2 gigabytes of data
 - Sort this on a machine with 128 MB RAM...



Merge Sort Good for Something!

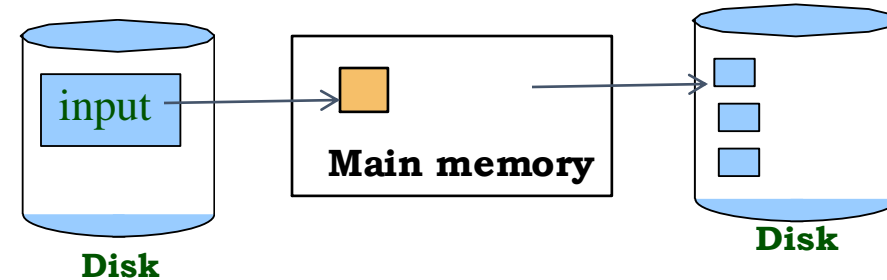
- Basis for most external sorting routines
- Can sort any number of records using a tiny amount of main memory
 - in extreme case, only need to keep 2 records in memory at any one time!



2-Way External MergeSort: Requires 3 Buffers

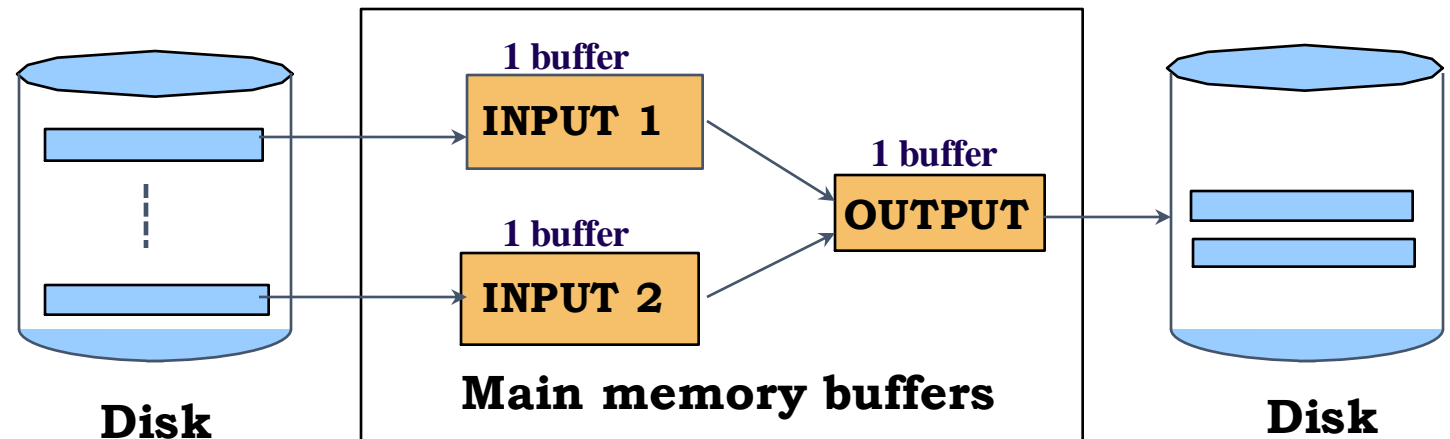
❖ Phase 1: PREPARE.

- Read a page, sort it, write it.
- only one buffer page is used





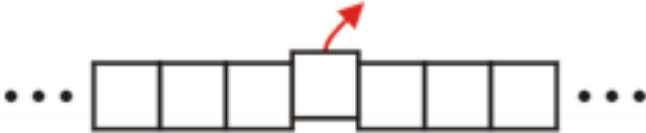
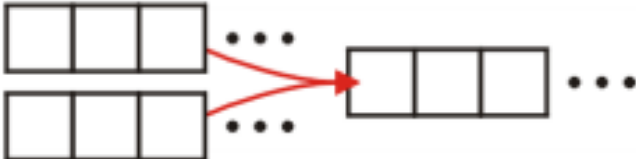
❖ Phase 2, 3, ..., etc.: MERGE:

- Three buffer pages used.



Comparison-based Sorting

Some of the typical operations used by sorting algorithms are:

- Insertion A diagram showing an array of six cells. The first three cells are grouped by an ellipsis on the left, and the last three by an ellipsis on the right. A single cell is positioned above the third cell, with a red arrow pointing down into it.
- Swapping A diagram showing an array of six cells. The first three cells are grouped by an ellipsis on the left, and the last three by an ellipsis on the right. A red curved arrow starts above the second cell and ends above the fifth cell, indicating a swap.
- Selection A diagram showing an array of six cells. The first three cells are grouped by an ellipsis on the left, and the last three by an ellipsis on the right. A red arrow starts from the fourth cell and points up and to the right, indicating a selection or swap.
- Merging A diagram showing two rows of three cells each. The first row is grouped by an ellipsis on the right, and the second row by an ellipsis on the right. Two red arrows start from the first and second cells of the first row and point to the first and second cells of a single row of six cells on the right, which is also followed by an ellipsis.

Lower bound for comparison-based sorting algorithms

- **Input:** A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.
- **Output:** A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.
- Comparison-based sort: all decisions based on comparing keys (“if statement”)
- A correct sorting algorithm must generate a different sequence of true/false answers for each permutation of $1 \dots n$.
- If algorithm asks $\leq d$ true/false questions, it generates $\leq 2^d$ different sequences of true/false answers.

Lower bound for comparison-based sorting algorithms

- So, $n! \leq 2^d$
- $n! \leq n^n$ and $n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$
- $\log n! = \Theta(n \log n)$
- $d = \Omega(n \log n)$
- Every comparison-based sorting algorithm takes $\Omega(n \log n)$ worst-case time.

Integer sorting

- We've already discussed that (under some more or less standard assumptions), no sort algorithm can have a run time better than $n \log n$
- However, there are algorithms that run in linear time (huh???)
- The key is that one of the conditions for that lower-bound is that it is applicable to sorting arbitrary (possibly random) data.

Bucket Sort

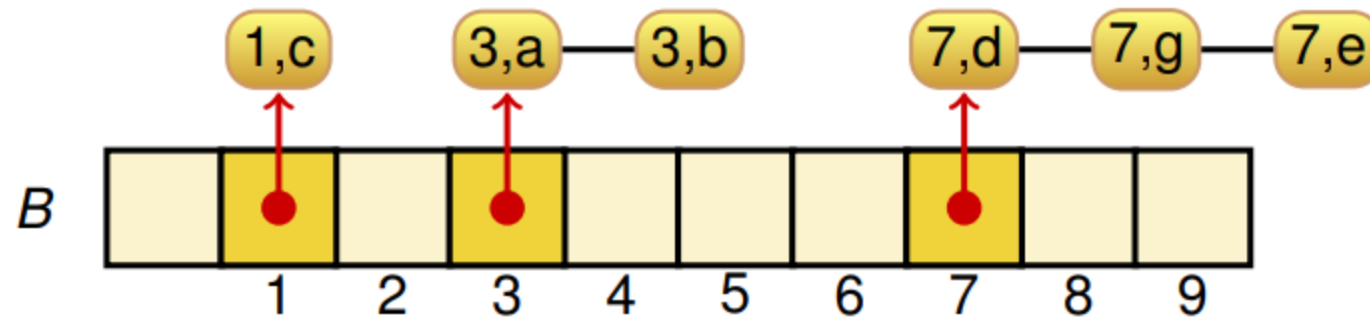
- If all keys are $0 \dots K - 1$
- Have an array of K buckets (linked lists)
- Put keys into correct bucket of array
 - linear time!
- Bucket Sort is a stable sorting algorithm:
 - Items in input with the same key end up in the same order as when they began
- Impractical for large K .

Bucket Sort: Example

- key range [0, 9]



- Phase 1: filling the buckets



- Phase 2: emptying the buckets into the list



Bucket Sort Time Analysis

Performance:

- phase 1 takes $O(n)$ time
- phase 2 takes $O(n + K)$ time

Thus bucket-sort is $O(n + K)$.

- very efficient if keys come from a small interval $[0, K - 1]$ (or in the extended version from a small set D , e.g., the names of the 50 U.S. states)

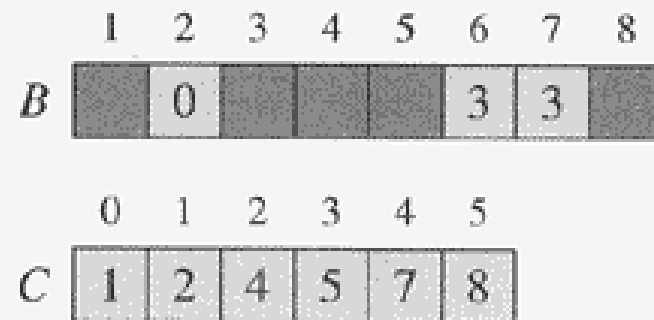
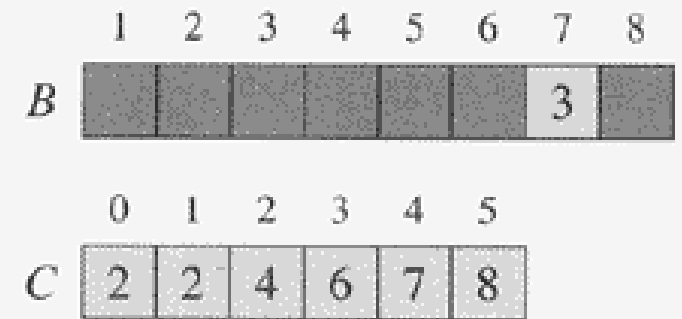
Counting Sort

- Depends on assumption about the numbers being sorted
 - Assume numbers are in the range $1 \dots k$
- The algorithm:
 - Input: $A[1..n]$, where $A[j] \in \{1, 2, 3, \dots, k\}$
 - Output: $B[1..n]$, sorted (not sorted in place)
 - Also: Array $C[1..k]$ for auxiliary storage

Counting Sort

```
1  CountingSort(A, B, k)
2      for i = 1 to k
3          C[i] = 0;
4      for j = 1 to n
5          C[A[j]] += 1;
6      for i = 2 to k
7          C[i] = C[i] + C[i-1];
8      for j = n downto 1
9          B[C[A[j]]] = A[j];
10         C[A[j]] -= 1;
```

Counting Sort Example



Counting Sort Running Time

```
1  CountingSort(A, B, k)
2      for i=1 to k
3          C[i] = 0;
4      for j=1 to n
5          C[A[j]] += 1;
6      for i=2 to k
7          C[i] = C[i] + C[i-1];
8      for j=n downto 1
9          B[C[A[j]]] = A[j];
10         C[A[j]] -= 1;
```

Takes time $O(k)$

Takes time $O(n)$

Counting Sort Running Time

- Total time: $O(n + k)$
 - Works well if $k = O(n)$
 - This sorting is stable.
-
- Why don't we always use counting sort?
 - Could we use counting sort to sort 32-bit integers? Why or why not?

Radix Sort

- Radix = “The base of a number system” (Webster’s dictionary)
 - Alternate terminology: radix is number of bits needed to represent 0 to base-1; can say “base 8” or “radix 3”
- Idea: Bucket Sort on each digit, bottom up.

The Magic of Radix Sort

- Input list:
126, 328, 636, 341, 416, 131, 328
- Bucket Sort on lower digit:
341, 131, 126, 636, 416, 328, 328
- Bucket Sort result on next-higher digit:
416, 126, 328, 328, 131, 636, 341
- Bucket Sort that result on highest digit:
126, 131, 328, 328, 341, 416, 636

Inductive Proof that Radix Sort Works

- Keys: d -digit numbers
 - (that wasn't hard!)
- Claim: after i th Bucket Sort, least significant i digits are sorted.
- Base case: $i = 0$. 0 digits are sorted.
- Inductive step: Assume for i , prove for $i + 1$.
- Consider two numbers: X, Y . Say X_i is i th digit of X :
 - $X_{i+1} < Y_{i+1}$ then $i + 1$ th Bucket Sort will put them in order
 - $X_{i+1} > Y_{i+1}$, same thing
 - $X_{i+1} = Y_{i+1}$, order depends on last i digits. Induction hypothesis says already sorted for these digits because Bucket Sort is stable

Running time of Radix sort

- n items, d digit keys
- How many passes?
- How much work per pass?
-
- Total time?

Running time of Radix sort

- n items, d digit keys
- How many passes?
 - Radix Sort requires one pass per digit. So, for d digit keys, there are d passes.

- How much work per pass?

Each pass involves a stable sort (like counting sort) over n items. If there are k possible values per digit, the time complexity per pass is $O(n+k)$

- Total time? $O(d(n+k))$

In many cases, we assume that the number of possible values per digit k (like the base of the number system) is constant or relatively small compared to n . This simplifies the time complexity to:

$$O(dn)$$

Summary

	Best	Average	Worst
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Bucket sort	$O(n + k)$	$O(n + k)$	$O(n + k)$
Counting sort	$O(n + k)$	$O(n + k)$	$O(n + k)$
Radix sort	$O(dn)$	$O(dn)$	$O(dn)$