BIRZEIT UNIVERSITY

COMP242

Data Structure

Lectures Note: Sorting

Prepared by: **Dr. Mamoun Nawahdah**

2016/2017

Sorting

In Place vs. not in Place Sorting:

In place sorting algorithms are those, in which we sort the data array, without using any additional memory.
What about **selection**, **bubble**, **insertion** sort algorithms?

- Well, our implementation of these algorithms is **IN PLACE**.
- The thing is, if we use a **constant** amount of extra memory (like one temporary variable/s), the sorting is **In-Place**.

But in case extra memory (**merging** sort algorithm), which is **proportional** to the input data size, is used, then it is **NOT IN PLACE sorting**.

- But because memory these days is so cheap, that we usually don't bother about using extra memory, **if** it makes the program run faster.

Stable vs. Unstable Sort:

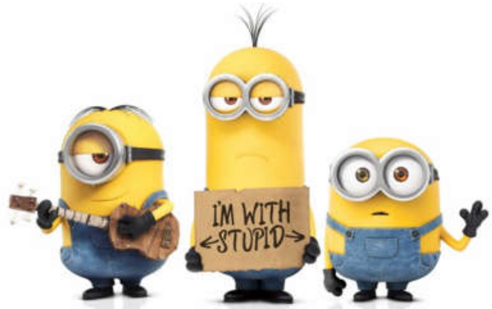
3	5	2	1	5’	10	**Unsorted Array**
1	2	3	5	5’	10	**Stable sort**
1	2	3	5’	5	10	**Unstable Sort**

Example: Insertion Sort Code:

```
public void sort(int[] data) {
    for (int i =0; i < data.length; i++) {
        int current = data[i];
        int j = i-1;
        while (j >=0 && data[j] > current) {
            data[j+1] = data[j];
            j--;
        }
        data[j+1] = current;
    }
}
```

```
public void sort(int[] data) {
    for (int i =0; i < data.length; i++) {
        int current = data[i];
        int j = i-1;
        while (j >=0 && data[j] >= current) {
            data[j+1] = data[j];
            j--;
        }
        data[j+1] = current;
    }
}
```

Example:



Unsorted Array

Name	Age
Bob	25
Kevin	24
Stuart	21
Kevin	28

1) Sorted By Age

Name	Age
Stuart	21
Kevin	24
Bob	25
Kevin	28

2) Sorted By Name (Stable)

Name	Age
Bob	25
Kevin	24
Kevin	28
Stuart	21

3) Sorted By Name (Unstable)

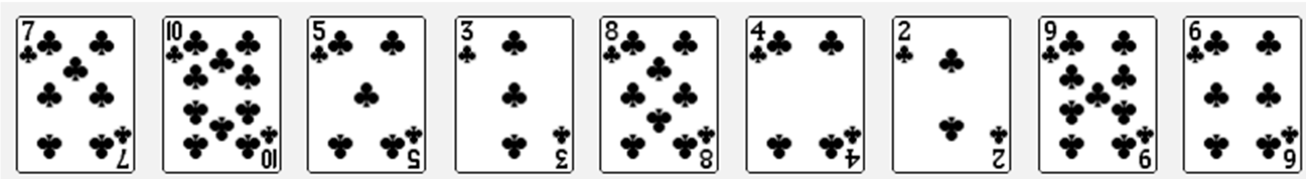
Name	Age
Bob	25
Kevin	28
Kevin	24
Stuart	21

http://www.sorting-algorithms.com/

1- Selection Sort:

- In iteration ***i***, find index ***min*** of smallest remaining entry.
- Swap ***a[i]*** and ***a[min]***.

Demo:



Java implementation:

```
public class Selection
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch(a, i, min);
      }
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

Mathematical analysis:

- Selection sort uses $(N-1) + (N-2) + \dots + 1 + 0 \approx N^2/2$ compares and N exchanges.

Trace of selection sort:

- Running time insensitive to input: **Quadratic time**, **even if input is sorted**.
- Data movement is minimal: **Linear number of exchanges**.

i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum

entries in red are a[min]

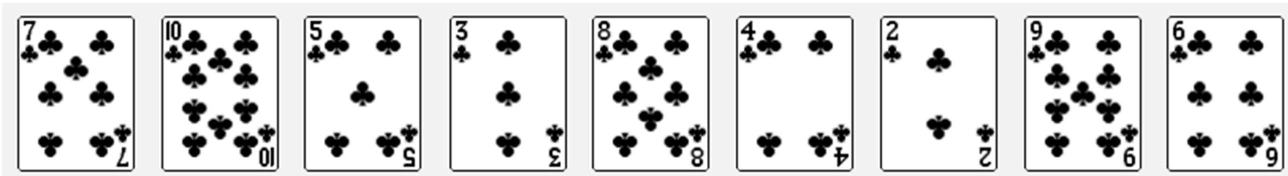
entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

2- Insertion Sort:

- In iteration ***i***, swap ***a[i]*** with each larger entry to its left.

Demo:



Java implementation:

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0; j--)
            if (less(a[j], a[j-1]))
               exch(a, j, j-1);
            else break;
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

Mathematical analysis:

- To sort a randomly-ordered array with distinct keys, insertion sort uses ≈ ***¼N***^{2} compares and ≈ ***¼N***^{2} exchanges on average.
- Expect each entry to move halfway back.

Trace of insertion sort:

- **Best case**: If the array is in ascending order, insertion sort makes ***N-1*** compares and ***0*** exchanges.
- **Worst case**: If the array is in descending order (and no duplicates), insertion sort makes ≈ ***½N***^{2} compares and ≈ ***½N***^{2} exchanges.
- For **partially-sorted** arrays, insertion sort runs in linear time.

i	j	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in gray do not move

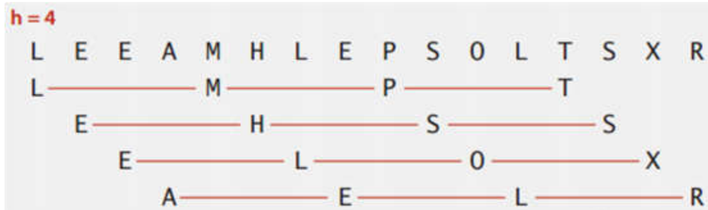
entry in red is a[j]

entries in black moved one position right for insertion

Trace of insertion sort (array contents just after each insertion)

3- Shell Sort:

Idea: Move entries more than one position at a time by **h-sorting** the array.
an **h-sorted** array is ***h*** interleaved sorted subsequences:



Shell sort: [**Shell 1959**] **h-sort** array for decreasing sequence of values of ***h***.

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

How to ***h-sort*** an array? Insertion sort, with stride length ***h***.

Shell sort example: increments **7, 3, 1**

Shell sort: which increment sequence to use?

- **Powers of two**: 1, 2, 4, 8, 16, 32, ... **No**
- **Powers of two minus one**: 1, 3, 7, 15, 31, 63, … **Maybe**
- **3x+1**: 1, 4, 13, 40, 121, 364, … **OK. Easy to compute**

Java implementation

```java
public class Shell
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;

      int h = 1;
      while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, ...

      while (h >= 1)
      {  // h-sort the array.
         for (int i = h; i < N; i++)
         {
            for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
               exch(a, j, j-h);
         }

         h = h/3;
      }
   }

   private static boolean less(Comparable v, Comparable w)
   { /* as before */ }
   private static void exch(Comparable[] a, int i, int j)
   { /* as before */ }
}
```

3x+1 increment sequence

insertion sort

move to next increment

Analysis

- The **worst-case** number of compares used by shell sort with the ***3x+1*** increments is ***$O(N^{3/2})$***.

4- Merge Sort

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

Mergesort overview

Java implementation:

Merging:

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   assert isSorted(a, lo, mid);    // precondition: a[lo..mid]   sorted
   assert isSorted(a, mid+1, hi);  // precondition: a[mid+1..hi] sorted

   for (int k = lo; k <= hi; k++)                                   copy
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)                                   merge
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }

   assert isSorted(a, lo, hi);     // postcondition: a[lo..hi] sorted
}
```

Java implementation:

Merge Sort:

```
public class Merge
{
   private static void merge(...)
   {  /* as before */  }

   private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
   {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid+1, hi);
      merge(a, aux, lo, mid, hi);
   }

   public static void sort(Comparable[] a)
   {
      aux = new Comparable[a.length];
      sort(a, aux, 0, a.length - 1);
   }
}
```

Merge Sort: trace

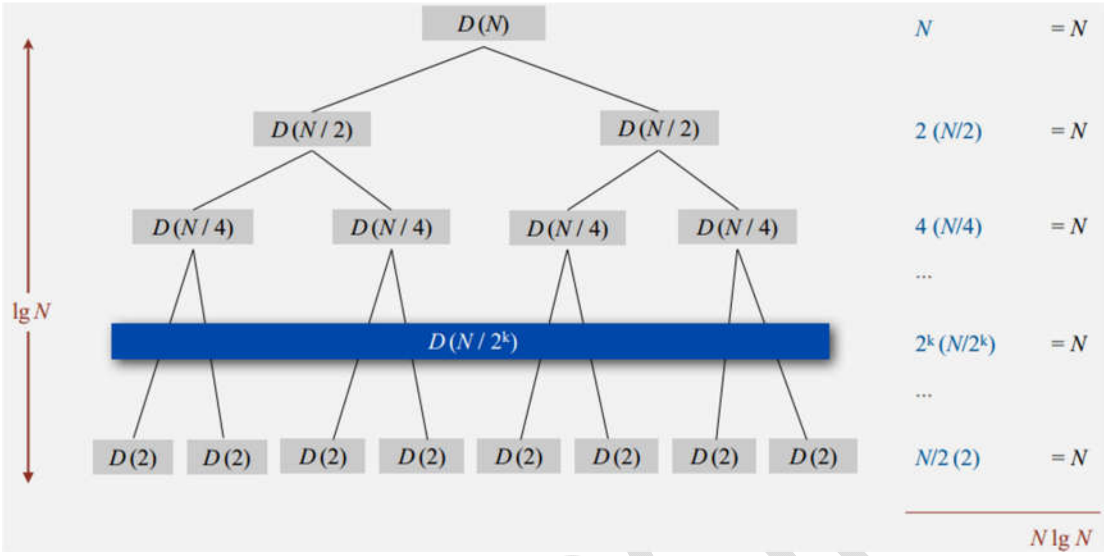
	lo		hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux,	0,	0,	1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux,	2,	2,	3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux,	0,	1,	3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux,	4,	4,	5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux,	6,	6,	7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux,	4,	5,	7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, aux,	0,	3,	7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, aux,	8,	8,	9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, aux,	10,	10,	11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, aux,	8,	9,	11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux,	12,	12,	13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux,	14,	14,	15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, aux,	12,	13,	15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux,	8,	11,	15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, aux,	0,	7,	15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Merge Sort: Empirical Analysis

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Good algorithms are better than supercomputers.

Divide-and-conquer recurrence: number of compares



Merge Sort analysis: memory (array accesses)

- Merge sort uses extra space proportional to ***N***.
- The array ***aux[]*** needs to be of size ***N*** for the last merge.

Practical Improvements:

I. Use **insertion** sort for small subarrays:
   - Merge sort has too much overhead for tiny subarrays.
   - ***Cutoff*** to insertion sort for ≈ **7** items.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

II. Stop if already sorted:
   - If biggest item in first half ≤ smallest item in second half?
   - Helps for partially-ordered arrays.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   if (!less(a[mid+1], a[mid])) return;
   merge(a, aux, lo, mid, hi);
}
```

III. Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)             aux[k] = a[j++];
      else if (j > hi)              aux[k] = a[i++];
      else if (less(a[j], a[i]))    aux[k] = a[j++];
      else                          aux[k] = a[i++];
   }
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (aux, a, lo, mid);
   sort (aux, a, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

merge from a[] to aux[]

Note: sort(a) initializes aux[] and sets aux[i] = a[i] for each i.

switch roles of aux[] and a[]

Complexity of sorting

- Compares? Merge sort is optimal with respect to number compares.
- Space? Merge sort is not optimal with respect to space usage.

5- Bottom-up Merge Sort

Basic plan:

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16, ...

	a[i] 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
sz = 1	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, aux, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, aux, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																
merge(a, aux, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, aux, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, aux, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4																
merge(a, aux, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 8																
merge(a, aux, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Java implementation:

```
public class MergeBU
{
   private static void merge(...)
   {  /* as before */  }

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      Comparable[] aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)
         for (int lo = 0; lo < N-sz; lo += sz+sz)
            merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

6- Radix Sort

What is Radix? The **radix** (or **base**) is the number of unique digits, including **zero**, used to represent numbers in a positional numeral system.

For example, for the decimal system: radix is **10**, Binary system: radix is **2**.

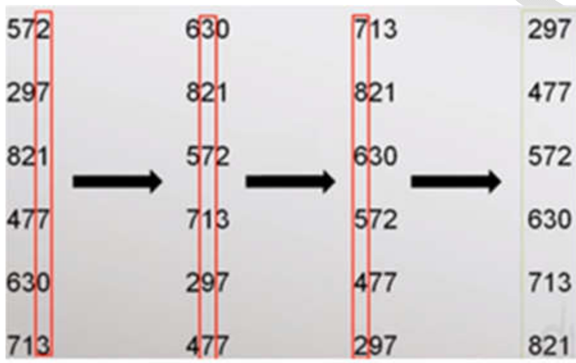
Example Radix Sort:

Step 1: take the least significant digits (***LSD***) of the values to be sorted.

Step 2: sort the list of elements based on that digit.

Step 3: take the 2nd ***LSD*** and repeat step 2.

Then the 3rd ***LSD*** and so on.



Radix Sort Algorithm using linked lists:

- Consider the following array:

A

9	179	139	38	10	5	36

- Create an array of **10** linked lists as follow:
  - **0** to **9** refer to actual numbers.
  - With input numbers, we will start with **mod (%) 10** then **divide (/)** the resulted number by **1**.

0	→
1	→
2	→
3	→
4	→
5	→
6	→
7	→
8	→
9	→

Code:

- **m=10** ➔ mod operation
- **n=1** ➔ find the specific digit at that column

e.g. **A[0]** = 9

9 % m = 9 → 9 / n = 9

- In this case add **A[0]** to the **10th** linked list
- Repeat for remaining array elements.

0	→ 10
1	→
2	→
3	→
4	→
5	→ 5
6	→ 36
7	→
8	→ 38
9	→ 9 → 179 → 139

- If we reach the end of array: make a new array by removing data from the head of each linked list in order:

10	5	36	38	9	179	139

Is this sorted? NO

- **Next step:** consider the **2**nd significant digit from the previous resulted array:

Code:

- **m =** m * 10 = **100**
- **n** = n * 10 = **10**

  e.g. **A[0]** = 10

  10 **%** m = 10 → 10 **/** n = 1

0	→ 5 → 9
1	→ 10
2	→
3	→ 36 → 38 → 139
4	→
5	→
6	→
7	→ 179
8	→
9	→

Result:

5	9	10	36	38	139	179

Is this sorted? **Yes,** in this case but we are not done yet

- **Next step:** consider the **3**rd significant digit from the previous array:

Code:

- **m =** m * 10 = **1000**
- **n =** n * 10 = **100**

  e.g. **A[0]** = 5

  5 **%** m = 5 → 5 **/** n = 0

0	→ 5 → 9 → 10 → 36 → 38
1	→ 139 → 179
2	→
3	→
4	→
5	→
6	→
7	→
8	→
9	→

Result:

5	9	10	36	38	139	179

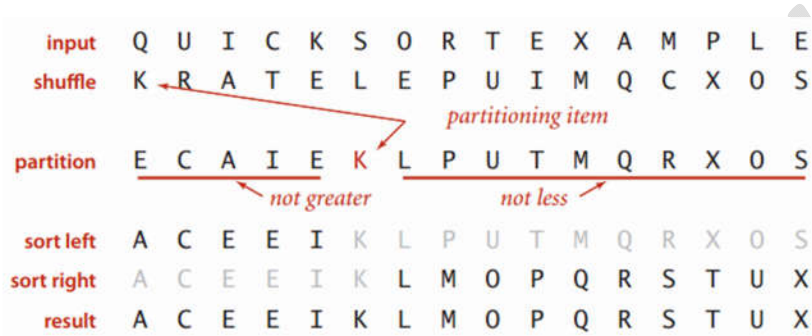
Is this sorted? What is the time complexity?

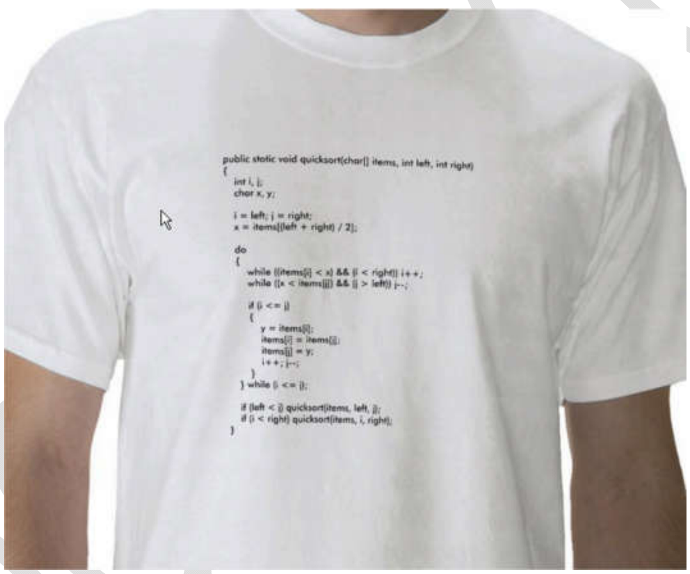
HW: implement Radix sort using Doubly Linked List

7- Quick Sort

Basic plan:

- Shuffle the array. (*shuffle needed for performance guarantee*)
- Partition so that, for some ***j***
  - entry ***A[j]*** is in place
  - no larger entry to the left of ***j***
  - no smaller entry to the right of ***j***
- Sort each piece recursively.





Quicksort t-shirt

Quicksort partitioning demo

Repeat until ***i*** and ***j*** pointers cross.

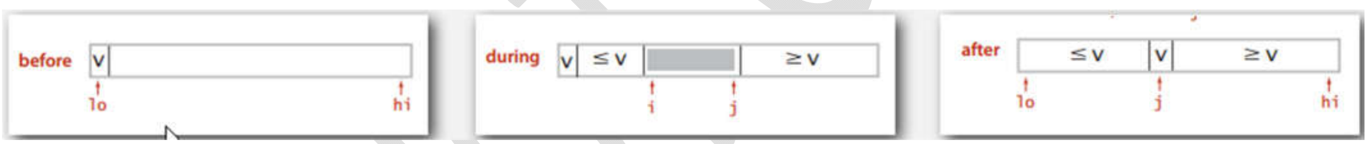
- Scan ***i*** from left to right so long as (***A[i] < A[lo]***).
- Scan ***j*** from right to left so long as (***A[j] > A[lo]***).
- Exchange ***A[i]*** with ***A[j]*** .

When pointers (***i*** and ***j***)cross.

- Exchange ***A[lo]*** with ***A[j]*** .

Quicksort: Java code for partitioning

```
private static int partition(Comparable[] a, int lo, int hi)
{
   int i = lo, j = hi+1;
   while (true)
   {
      while (less(a[++i], a[lo]))            find item on left to swap
         if (i == hi) break;

      while (less(a[lo], a[--j]))            find item on right to swap
         if (j == lo) break;

      if (i >= j) break;                     check if pointers cross
      exch(a, i, j);                         swap
   }

   exch(a, lo, j);                           swap with partitioning item
   return j;                                 return index of item now known to be in place
}
```




```
public class Quick
{
   private static int partition(Comparable[] a, int lo, int hi)
   {  /* see previous slide */  }

   public static void sort(Comparable[] a)
   {
      StdRandom.shuffle(a);
      sort(a, 0, a.length - 1);
   }

   private static void sort(Comparable[] a, int lo, int hi)
   {
      if (hi <= lo) return;
      int j = partition(a, lo, hi);
      sort(a, lo, j-1);
      sort(a, j+1, hi);
   }
}
```

Quicksort trace

	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
no partition for subarrays of size 1	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort trace (array contents after each partition)

Quicksort: Empirical Analysis

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Quicksort: Compare analysis

Best case: Number of compares is ≈ ***N log N***

	lo	j	hi	a[] 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values				H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle				H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
	0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
	0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
	0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
	0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
	2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
	4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
	4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
	6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
	8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
	8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
	8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
	10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
	12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
				A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Worst case: Number of compares is ≈ ***½N²***

a[ ]

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Average-case analysis: Complicated ➔ ***2N log N***

Quicksort: summary of performance characteristics

Worst case: Number of compares is quadratic.

- ***N + (N - 1) + (N - 2) + … + 1*** ≈ ***½ N²***
- but this **rarely** to happen.

Average case: Number of compares is ≈ ***1.39 N log N***

- 39% more compares than Merge sort
- But faster than Merge sort in practice because of less data movement.

Random shuffle

- Probabilistic guarantee against worst case.

Quicksort is an **in-place** sorting algorithm.

Quicksort is **not stable**.

Quicksort: practical improvements

I. Insertion sort small subarrays:

- Even quicksort has too much overhead for tiny subarrays.
- **Cutoff** to insertion sort for ≈ 10 items.
- Note: could delay insertion sort until one pass at end.

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

II. Median of sample:

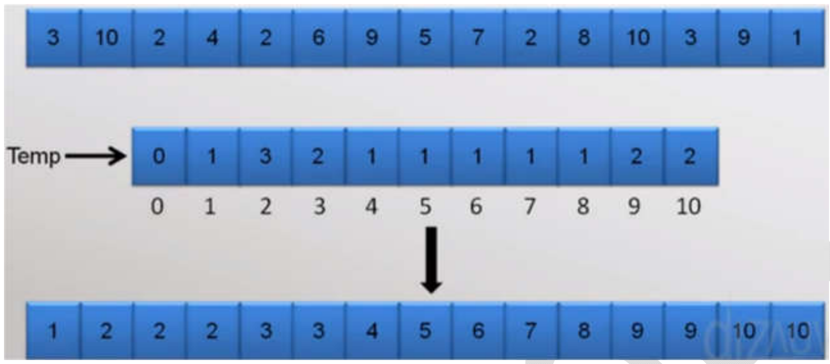
- Best choice of pivot item = median.
- Estimate true median by taking median of sample.

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo) return;

   int m = medianOf3(a, lo, lo + (hi - lo)/2, hi);
   swap(a, lo, m);

   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

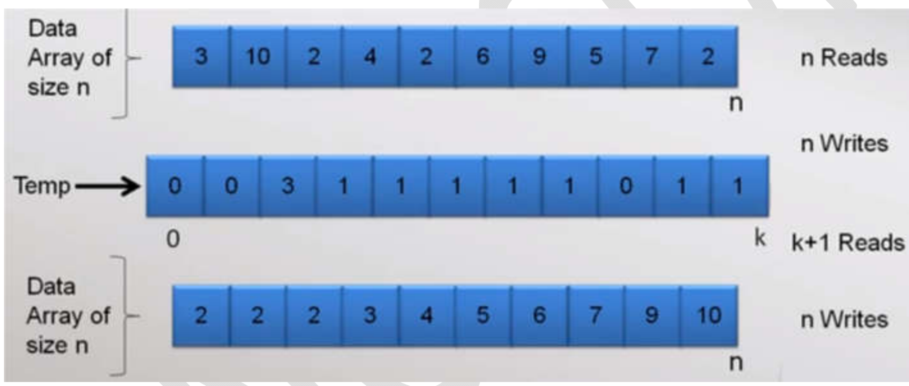
8- Counting Sort

If we know some information about data to be sorted (e.g. students' marks [Range 55 to 99]), we can achieve linear time sorting

Example: assume data range from 1 to 10

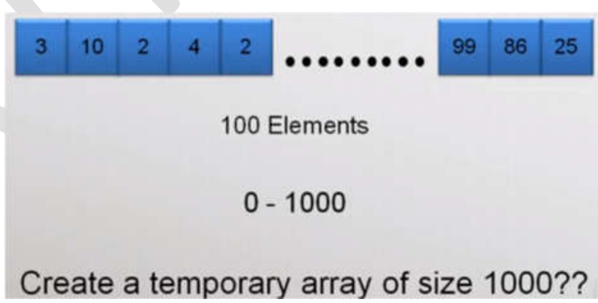


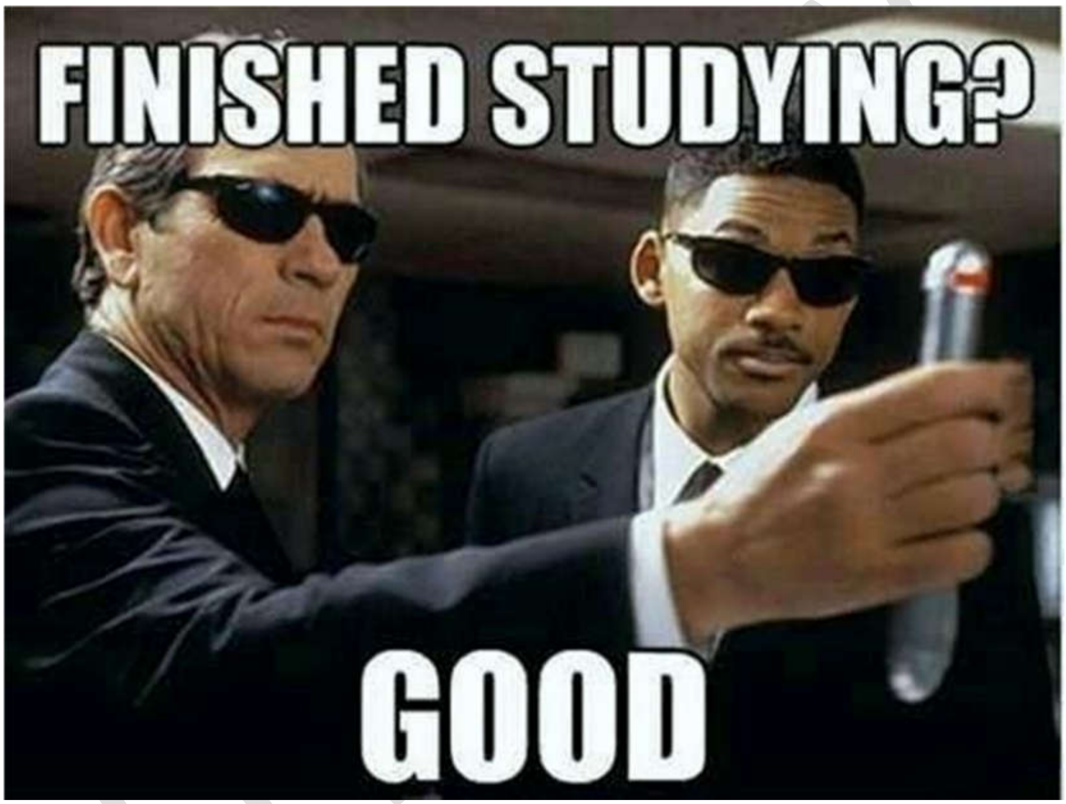
Time analysis:



Note: K is typically small comparing to n

Bad Situation: what if K is larger than n ??




FINISHED STUDYING?
GOOD