Elementary Sorts (Week 2):

- Sorting Introduction
    - Let's say you have some piece of information that contains multiple records, you want to sort the records into order according to their ID (numbers). This is the sort problem.
    - Sort problem
        - Re-arrange an array of N items into ascending order according to a defined key which is part of the item (ex. ID number).
        - Our goal is to be able to sort any type of data.
            - This includes real numbers, strings, files, and so on.
    - How can we make it so that we can implement one sort program that can be used by multiple clients to implement different types of data?
        - The answer brings us to callbacks.
    - Callback
        - A reference to executable code.
            - The client passes an array of objects into the sort() function.
            - In Java, there's an implicit mechanism that says that any such array of objects is going to have the compareTo() method.
            - The sort() function calls back the object's compareTo() method whenever it needs to compare two objects.
        - Implementing callbacks.
            - Java: interfaces.
            - C: function pointers.
            - C++: class-type functors.
            - C#: delegates.
            - Python, Perl, ML, Javascript: first-class functions.
            - It's all about the idea of passing functions as arguments to other functions.
            - For Java, because of the desire to check types at compile time, they use a specific method called an interface.
                - Allows us to use the sorts that we developed for any type of data in a typesafe manner.
    - Sort Roadmap using callbacks

client

```
import java.io.File;
public class FileSorter
{
   public static void main(String[] args)
   {
      File directory = new File(args[0]);
      File[] files = directory.listFiles();
      Insertion.sort(files);
      for (int i = 0; i < files.length; i++)
         StdOut.println(files[i].getName());
   }
}
```

object implementation

```
public class File
implements Comparable<File>
{
   ...
   public int compareTo(File b)
   {
      ...
      return -1;
      ...
      return +1;
      ...
      return 0;
   }
}
```

Comparable interface (built in to Java)

```
public interface Comparable<Item>
{
   public int compareTo(Item that);
}
```

sort implementation

```
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 (a[j].compareTo(a[j-1]) < 0)
            exch(a, j, j-1);
         else break;
}
```

key point: no dependence on File data type

-
  - The comparable interface is provided by Java, then we implement the interface into our object class, then our sort class takes in an array of "comparable" objects (Comparable[] a) and "calls back" the compareTo() method from the respective object's (the actual type of the objects inside the array) class so that our sort works on all objects that implement the Comparable interface. The client can now call sort on any object with the Comparable interface implemented.
- Rules for implementing the compareTo() method
  - **Total order**

    A total order is a binary relation $\leq$ that satisfies:
    - Antisymmetry: if $v \leq w$ and $w \leq v$, then $v = w$.
    - Transitivity: if $v \leq w$ and $w \leq x$, then $v \leq x$.
    - Totality: either $v \leq w$ or $w \leq v$ or both.
  - Ex.
    - Standard order for natural and real numbers.
    - Chronological order for dates or times.
    - Alphabetical order for strings.
    - ...

- A total order must exist in the compareTo() method, there must be an order like the alphabet. An example of a violation of total order is rock, paper, scissors.

- Comparable API
  - Implement `compareTo()` so that `v.compareTo(w)`
    - Is a total order.
    - Returns a negative integer, zero, or positive integer if v is less than, equal to, or greater than w, respectively.
    - Throws an exception if incompatible types (or either is `null`).

  - Built-in comparable types. `Integer`, `Double`, `String`, `Date`, `File`, ...
  - User-defined comparable types. Implement the `Comparable` interface.
- Implementing the Comparable interface
  - An example would be a Date data type.

```
public class Date implements Comparable<Date>
{
   private final int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }

   public int compareTo(Date that)
   {
      if (this.year  < that.year ) return -1;
      if (this.year  > that.year ) return +1;
      if (this.month < that.month) return -1;
      if (this.month > that.month) return +1;
      if (this.day   < that.day  ) return -1;
      if (this.day   > that.day  ) return +1;
      return 0;
   }
}
```

only compare dates to other dates

- Two useful sorting abstractions
  - Helper functions
    - You can use helper functions to refer to data through compares and exchanges.
    - less()
      - Checks if the first item is less than the second item.
      - Calls back to the object's compareTo() method.

```
private static boolean less(Comparable v, Comparable w)
{  return v.compareTo(w) < 0;  }
```

- exch()
  - Takes in an array and swaps the value of two indices.

```
private static void exch(Comparable[] a, int i, int j)
{
   Comparable swap = a[i];
   a[i] = a[j];
   a[j] = swap;
}
```

- Testing
  - The goal is to test if an array is sorted.

```
private static boolean isSorted(Comparable[] a)
{
   for (int i = 1; i < a.length; i++)
      if (less(a[i], a[i-1])) return false;
   return true;
}
```

  - If the sorting algorithm passes the test, did it correctly sort the array?
    - Yes if you used only the less and exch methods to refer to data because then you know that the data in the array at the end of the sort is the same data in the array before it went into the sort.

- Selection Sort
  - Explanation
    - In the i'th iteration, find the index (min) of the smallest remaining entry, then swap a[i] and a[min], then increment i.
    - Pretty much, just keep finding the smallest entry of the unsorted part of the array then add it to the sorted part of the array.
      - Example
        - 3, 4, 2, 1, 7
        - 1, 4, 2, 3, 7
        - 1, 2, 4, 3, 7
        - 1, 2, 3, 4, 7
        - 1, 2, 3, 4, 7
  - Algorithm
    - The pointer (i) scans from left to right.
    - Invariants
      - Entries on the left of and on our pointer (i) are fixed and in ascending order.

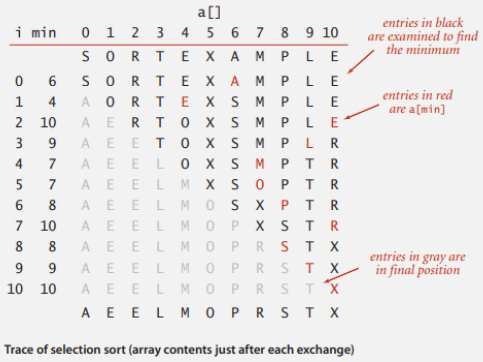
- No entry to the right of our pointer (i) is smaller than any entry to the left of our pointer (i).

- To maintain the algorithm's invariants
  - Move the pointer to the right.
    - i++; - occurs in the outer for-loop
  - Identify the index of the minimum entry on the right of our pointer including the entry at the pointer.
    -
```
int min = i;
for (int j = i+1; j < N; j++)
   if (less(a[j], a[min]))
      min = j;
```
  - Swap/exchange the entry at our pointer and the minimum entry.
    -
```
exch(a, i, min);
```

- 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

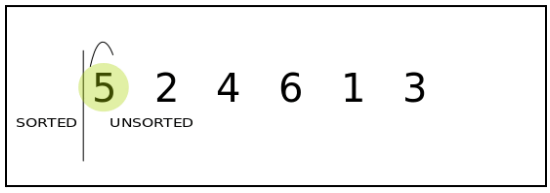
Proposition. Selection sort uses $(N-1)+(N-2)+\ldots+1+0 \sim N^2/2$ compares and N exchanges.



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

Running time insensitive to input. Quadratic time, even if input is sorted.

Data movement is minimal. Linear number of exchanges.

- Insertion Sort
  - Explanation
    - In the i'th iteration, swap the entry at i with the entry to its left if it is greater. Continue swapping the entry until there are no entries greater than a[i] to its left.
    - Think of the swapping as a mini reverse bubble sort. Instead of taking a value and swapping it to the right if it is larger, you take the entry at i and keep swapping it to the left if it is smaller until it is in place (in place meaning all entries to the left of where the selected entry ends up, are smaller than our selected entry).
      - Example
    - 

  - Algorithm
    - The pointer (i) scans from left to right.
    - Invariants
      - Entries on and to the left of our pointer (i) are in ascending order.
      - Entries to the right of our pointer have not been seen yet.
      - 

    - To maintain the algorithm's invariants

- Move the pointer to the right.
    - i++; - occurs in the outer for-loop
- From right to left, exchange the entry at i with each larger entry to its left.
    -
```
for (int j = i; j > 0; j--)
   if (less(a[j], a[j-1]))
        exch(a, j, j-1);
   else break;
```
- Example
    - 

        - A[i] = 3
    - 

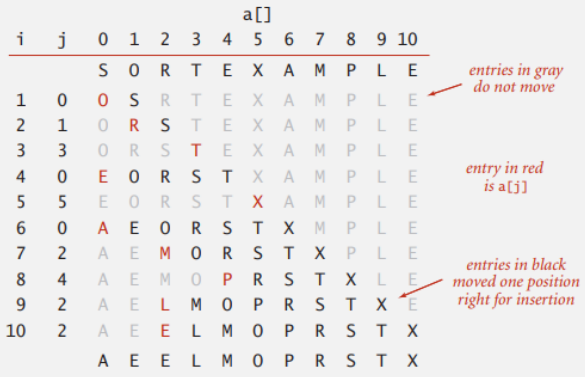
        - Keep swapping 3 with each larger value to its left. We end up with 3 after 2 and before 4 which is the correct spot. This way, all the entries up to and including the pointer (i) are in ascending order and any entries to the right, have not been seen yet.

- 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

Proposition. To sort a randomly-ordered array with distinct keys, insertion sort uses $\sim \frac{1}{4} N^2$ compares and $\sim \frac{1}{4} N^2$ exchanges on average.

Pf. Expect each entry to move halfway back.



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

-

Best case. If the array is in ascending order, insertion sort makes $N - 1$ compares and 0 exchanges.

A E E L M O P R S T X

-

Worst case. If the array is in descending order (and no duplicates), insertion sort makes $\sim \frac{1}{2} N^2$ compares and $\sim \frac{1}{2} N^2$ exchanges.

X T S R P O M L E E A

-

Insertion sort: partially-sorted arrays

Def. An inversion is a pair of keys that are out of order.

A E E L M O T R X P S

T-R T-P T-S R-P X-P X-S
(6 inversions)

Def. An array is partially sorted if the number of inversions is $\leq c N$.

- Ex 1. A subarray of size 10 appended to a sorted subarray of size N.
- Ex 2. An array of size N with only 10 entries out of place.

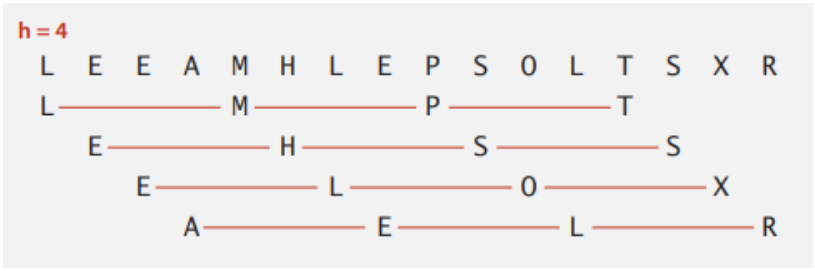
Proposition. For partially-sorted arrays, insertion sort runs in linear time.

Pf. Number of exchanges equals the number of inversions.

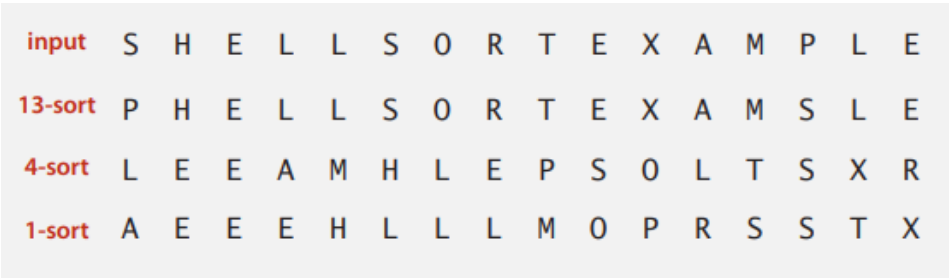
number of compares = exchanges + (N – 1)

-

- Shell Sort
  - The idea of shell sort stems from the fact that insertion sort is inefficient because elements really move only one position at a time even when you know a certain element needs to move a far way down.
    - This leads us to shell sort, we'll move entries several positions at a time in a method called "h-sorting".
  - H-sorting
    - An h-sorted array is h interleaved sorted subsequences.

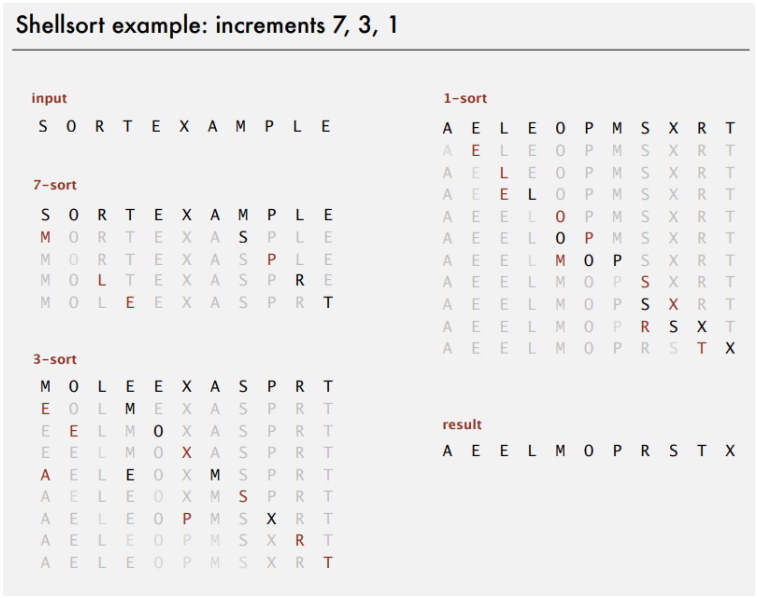


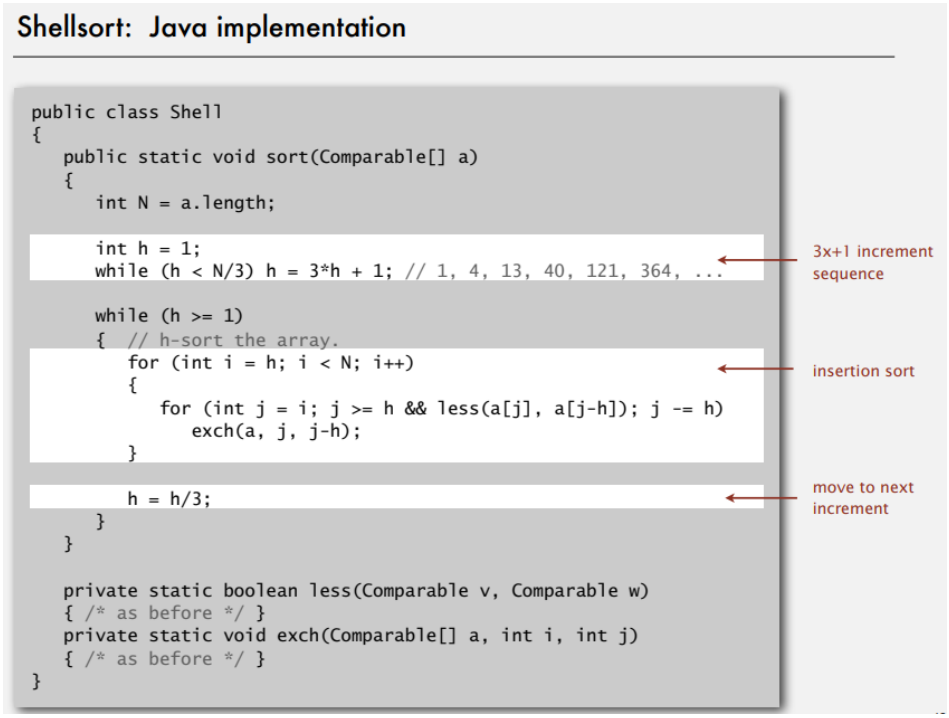
      - So since h = 4 here, you take every 4th index from the current index and sort this subsequence.
  - Concept of shell sort
    - H-sort the array for decreasing sequence of values of h.



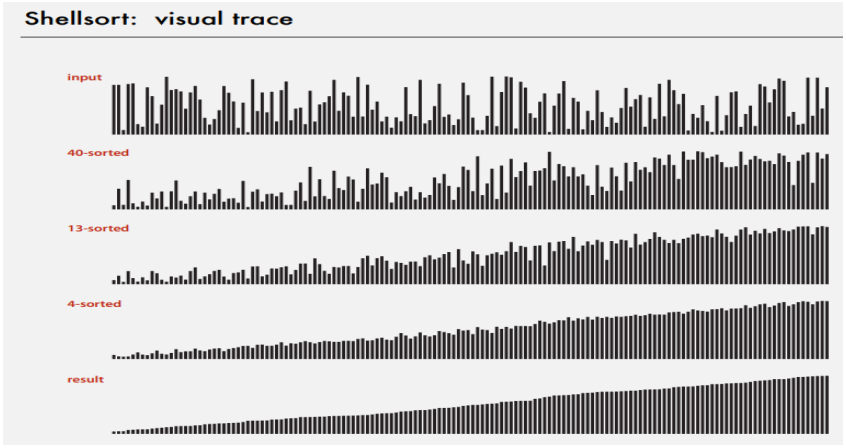
      - In 13-sort, the 0th index and the 13th index get ordered.
      - In 4-sort, subsequences of every 4th index get ordered.
      - In 1-sort, everything gets ordered.
    - By having high-value sorts, you can order the subsequences so that elements that would've been normally compared against lots of other elements to get moved far up or down, can instead compare to other elements in the subsequence and get moved further up or down the array in less compares (since the h is high, the subsequence has fewer elements the indices are more spread out, this means even a single compare can move an element many many indices up or down).
    - How do we h-sort an array?
      - Implement insertion sort, but change the compare statement with the index h to the left instead of the index directly on the left ([j-h] instead of [j-1]) and decrement the for-loop by h instead of 1. Also, include a check that j is greater than h so as to not move too far left and go out of bounds of the array.
      - Why insertion sort?

- For big h-values, any sorting method will do, however when you get to small h-values, this means that the array is already partially sorted due to the h-sorts with high h-values, since the array is partially sorted, insertion sort is the best method since it works better the more sorted an array is.

- Example
  - 

- What increment sequence should we use for h-values?
  - 3x + 1 (1, 4, 13, 40, 121, 364…)
  - Empirical studies (1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905…)
- Java implementation
  - 

- First, we use a while loop to set h to be the maximum h-value without being too big for the array.
    - The while loop will run and increase h until h is no longer smaller than 1/3rd of the array, this makes it so that there are at least two values to compare to each other.
- Then we have a while loop that runs as long as h is larger than or equal to 1. We also decrement h by dividing by 3 each loop (which reverses the incrementation we did about). This decrementation and the while loop make sure the array gets h-sorted by each value and eventually, insertion sorted (h-sort of h=1).
    - Inside the while loop, we have a modified insertion sort that checks every h'th index before the current index.

- Visualization
    - 

- Benefits of shell sort
    - Fast unless array size is huge.
    - Doesn't take too much code (can be used in embedded systems)
    - Hardware sort prototype (???)