Today
- Sorting
- Bubble Sort
- Insertion Sort
- Shell Sort
Comparison Based Sorting

- **Input** – 2, 3, 1, 15, 11, 23, 1
- **Output** – 1, 1, 2, 3, 11, 15, 23

- **Class** ‘**Animals**’
  - Sort Objects – Rabbit, Cat, Rat ??
  - Class must specify how to compare Objects
  - ‘<‘ and ‘>’ operators
In place sorting
- Sorting of a data structure does not require any external data structure for storing the intermediate steps

External sorting
- Sorting of records not present in memory

Stable sorting
- If the same element is present multiple times, then they retain the original positions
STL Sorting

- `sort` function template

- `void sort (iterator begin, iterator end)`
- `void sort (iterator begin, iterator end, Comparator cmp)`
Bubble Sort

- Simple and uncomplicated
- Compare neighboring elements
- Swap if out of order
- Two nested loops
- $O(N^2)$
for (i=0; i<n-1; i++) {
    for (j=0; j<n-1-i; j++)
        /* compare the two neighbors */
        if (a[j+1] < a[j]) {
            /* swap a[j] and a[j+1] */
            tmp = a[j];
            a[j] = a[j+1];
            a[j+1] = tmp;
        }
}

http://www.ee.unb.ca/brp/lib/java/bubblesort/
Insertion Sort

- $O(N^2)$ sort
- $N-1$ passes
  - After pass $p$ all elements from 0 to $p$ are sorted
  - Following step inserts the next element in correct position within the sorted part
Insertion Sort

```cpp
/**
 * Simple insertion sort.
 */

template <typename Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size(); p++ )
    {
        Comparable tmp = a[ p ];
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
Insertion Sort

<table>
<thead>
<tr>
<th>Original</th>
<th>34</th>
<th>8</th>
<th>64</th>
<th>51</th>
<th>32</th>
<th>21</th>
<th>Positions Moved</th>
</tr>
</thead>
<tbody>
<tr>
<td>After $p = 1$</td>
<td>8</td>
<td>34</td>
<td>64</td>
<td>51</td>
<td>32</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>After $p = 2$</td>
<td>8</td>
<td>34</td>
<td>64</td>
<td>51</td>
<td>32</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>After $p = 3$</td>
<td>8</td>
<td>34</td>
<td>51</td>
<td>64</td>
<td>32</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>After $p = 4$</td>
<td>8</td>
<td>32</td>
<td>34</td>
<td>51</td>
<td>64</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>After $p = 5$</td>
<td>8</td>
<td>21</td>
<td>32</td>
<td>34</td>
<td>51</td>
<td>64</td>
<td>4</td>
</tr>
</tbody>
</table>
Insertion Sort - Analysis

- Pass $p$ involves at most $p$ comparisons
- Total comparisons:

$$\sum_{i=2}^{N} i = O(N^2)$$
Insertion Sort - Analysis

- **Worst Case?**
  - Reverse sorted list
  - Max possible number of comparisons
  - $O(N^2)$

- **Best Case?**
  - Sorted input
  - 1 comparison in each pass
  - $O(N)$
Inversions

- An ordered pair \((i, j)\) such that \(i < j\) but \(a[i] > a[j]\)
- \(34, 8, 64, 51, 32, 21\)
- \((34, 8), (34, 32), (34, 21), (64, 51) \ldots\)

Once an array has no inversions it is sorted

So sorting bounds depend on ‘average’ number of inversions performed
Theorem 1

- Average number of inversions in an array of $N$ distinct elements is $N(N-1)/4$

- List $L$, Reverse list $L_1$
- All possible number of pairs is $\binom{N}{2}$

- $N(N-1)/2$,
- Average = $N(N-1)/4$
Theorem 2

- Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ average time.

- Average number of inversions = $O(N^2)$

- Number of swaps required = $O(N^2)$
Tighter Bound

- $O(N \log N)$
  - Optimal bound for comparison based sorting algorithms
  - Achieved by Quick Sort, Merge Sort, and Heap Sort
Shell Sort

Also referred to as *Diminishing Increment Sort*

- Incrementing sequence – $h_1, h_2, \ldots, h_k$
- $h_1 = 1$
- After a phase using $h_k$, for each $i$, $a[i] \leq a[i+h_k]$
- In other words – all elements spaced $h_k$ apart are sorted
- Start with $h = \lceil N/2 \rceil$; keep reducing by half in each iteration
### Shell Sort

<table>
<thead>
<tr>
<th></th>
<th>81</th>
<th>94</th>
<th>11</th>
<th>96</th>
<th>12</th>
<th>35</th>
<th>17</th>
<th>95</th>
<th>28</th>
<th>58</th>
<th>41</th>
<th>75</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>After 1-sort</strong></td>
<td>35</td>
<td>17</td>
<td>11</td>
<td>28</td>
<td>12</td>
<td>41</td>
<td>75</td>
<td>15</td>
<td>96</td>
<td>58</td>
<td>81</td>
<td>94</td>
<td>95</td>
</tr>
<tr>
<td><strong>After 3-sort</strong></td>
<td>28</td>
<td>12</td>
<td>11</td>
<td>35</td>
<td>15</td>
<td>41</td>
<td>58</td>
<td>17</td>
<td>94</td>
<td>75</td>
<td>81</td>
<td>96</td>
<td>95</td>
</tr>
<tr>
<td><strong>After 5-sort</strong></td>
<td>11</td>
<td>12</td>
<td>15</td>
<td>17</td>
<td>28</td>
<td>35</td>
<td>41</td>
<td>58</td>
<td>75</td>
<td>81</td>
<td>94</td>
<td>95</td>
<td>96</td>
</tr>
</tbody>
</table>
Shell Sort

```cpp
1 /**
2  * Shellsort, using Shell's (poor) increments.
3  */
4 template <typename Comparable>
5 void shellsort( vector<Comparable> & a )
6 {
7     for( int gap = a.size() / 2; gap > 0; gap /= 2 )
8         for( int i = gap; i < a.size(); i++ )
9             {
10                Comparable tmp = a[ i ];
11                int j = i;
12
13                     for( ; j >= gap && tmp < a[ j - gap ]; j -= gap )
14                         a[ j ] = a[ j - gap ];
15                     a[ j ] = tmp;
16             }
17     }
```
Shell Sort - Analysis

- Each pass \((h_k)\) is \(O(h_k(N/h_k)^2)\)
- \(h = 1, 2, 4, \ldots N/2\)
- Total sums to \(O(N^2)\sum 1/h_k\)
- \(\sum 1/h_k = 2\)
- So .. \(O(N^2)\)
int n, k = 20;
for(n = 0; n < k; n--)
    printf("X");

Change, remove, add ONLY one character anywhere to make this code print X 20 times (give or take a few)