Sorting

- Sorting is ordering a set of elements in increasing or decreasing order.
- We will assume that
  - Elements are comparable
  - They are kept in an array
  - Each cell of the array keep one element
  - For simplicity the elements are integers. But the same methods are valid for any type of element that can be ordered.
  - We will express the number of element to be sorted as N.
There are various sorting algorithms
- Easy algorithms: $O(N^2)$ running time
  - Insertion sort, etc.
- Very easy to implement ones: $o(N^2)$
  - Efficient in practice
- More complicated ones
  - Running time of $O(N\log N)$
  - Such as Quick Sort, Merge Sort, etc.

A general purpose sorting algorithm requires $\Omega(N\log N)$ comparisons.

The data to be sorted can fit in memory;
- We will first see the algorithms for this case.

The data can also be residing in disk and algorithm can be run over disk
- This is called external sorting.
Insertion Sort

- A simple algorithm
- Requires N-1 passes over the array to be sorted (of size N).
- For passes p=1 to N
  - Ensures that the elements in positions 0 through p are in sorted order.

Example

Array to be sorted.

N = 6

34  8  64  51  32  21
Pass 1

Compare

34  8  64  51  32  21  8

Current Item

move

8  34  64  51  32  21

insert

Pass 2

compare

8  34  64  51  32  21  64

Current Item
Pass 3

<table>
<thead>
<tr>
<th>8</th>
<th>34</th>
<th>64</th>
<th>51</th>
<th>32</th>
<th>21</th>
<th>51</th>
</tr>
</thead>
</table>

Current Item

Pass 4

<table>
<thead>
<tr>
<th>8</th>
<th>34</th>
<th>51</th>
<th>64</th>
<th>32</th>
<th>21</th>
<th>32</th>
</tr>
</thead>
</table>

Current Item
**Pass 5**

```
8 32 34 51 64
```

```
8 32 34 51 64
```

```
8 32 34 51 64
```

```
8 32 34 51 64
```

```
8 32 34 51 64
```

```
8 21 32 34 51 64
```

RESULT!!

---

**Pseudo-Code**

```cpp
void insertionSort(vector<int> &a)
{
    int j;
    for ( int p = 1; p < a.size(); ++p )
    {
        int tmp = a[p];
        for (j=p; j > 0 && tmp < a[j-1]; j--) /* compare */
        a[j] = a[j-1]; /* move */
        a[j] = tmp; /* insert */
    }
}
```
Analysis of Insertion Sort

- The test the line shown in the previous slide is done at most:
  - \( p+1 \) times for each value of \( p \).

\[
\sum_{i=2}^{N} i = 2 + 3 + 4 + \ldots + N = \Theta(N^2)
\]

Lower bound for simple sorting algorithms

- Simple sorting algorithms are the ones that make swaps of adjacent items.
  - Insertion sort
  - Bubble sort
  - Selection sort

- Inversion definition:
  - An inversion in an array of numbers is any ordered pair \((i,j)\) having the property that \( i < j \) but \( a[i] > a[j] \)
Inversion

Example:
- Array items: 34 8 64 51 32 21
- Inversions:
  - (34,8), (34,32), (34,21), (64,51), (64,32), (64,21),
  - (51,32), (51,21), and (32,21).
  - We have total of 9 inversions.
- Each inversion requires a swap in insertion sort to order the list.
- A sorted array has no inversions.
- Running time = O(I + N), where I is number of inversions.

Inversion

- Compute the average number of inversions in an array.
  - Assume no duplicates in the array (or list).
  - Assume there are N elements in range [1,N].
- Then input to the sorting algorithms is a permutation of these N distinct elements.
Theorem

**Theorem**: The average number of inversions in an array of $N$ distinct elements is $N(N-1)/4$.

**Proof**:
- For any list of items, $L$, consider the list in reverse order $L_r$.
  - $L = 34 \ 8 \ 64 \ 51 \ 32 \ 21$
  - $L_r = 21 \ 32 \ 51 \ 64 \ 8 \ 34$
- Consider any pair $(x,y)$ in list $L$, with $x < y$.
- The pair $(x,y)$ is certainly an inversion in one of the lists $L$ and $L_r$.

Proof continued

- The total number of these pairs (which are inversions) in a list $L$ and its reverse $L_r$ is $N(N-1)/2$.
- Therefore, an average list $L$ has half of this amount, which is $N(N-1)/4$. 
Shell Sort

- Invented by Donald Shell.
- Also referred to as diminishing increment sort.
- Shell sort uses a sequence $h_1, h_2, \ldots, h_t$, called the increment sequence.
  - $h_1$ must be 1.
  - Any sequence will do.

Shell Sort

- It is executed in phase.
  - One phase for each $h_k$
  - After a phase where increment were $h_k$
    - For every $i$, $a[i] \leq a[i+h_k]$.
    - This means all elements spaced $h_k$ apart are sorted.
    - The input is then said to be $h_k$ sorted.
- An $h_k$ sorted input, which is then $h_{k-1}$ sorted, is still $h_k$ sorted.
Shell Sort

<table>
<thead>
<tr>
<th>Original List</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>After 1-sort</td>
<td>37</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>After 3-sort</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>After 5-sort</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>

Shellsort Algorithm

```cpp
void shellsort(vector<int> &a) {
    int j, i;
    int gap;
    for (gap = a.size() / 2; gap > 0; gap /= 2) {
        for (i=gap; i < a.size(); i++) {
            int tmp = a[i];
            for (j=i; j>=gap && tmp < a[j-gap]; j -= gap) {
                a[j] = a[j-gap];
                a[j] = tmp;
            }
        }
    }
}
```
Choosing Increment Sequence

\[
\begin{align*}
h_l &= \left\lfloor \frac{N}{2} \right\rfloor \\
h_k &= \left\lfloor \frac{h_k + 1}{2} \right\rfloor \\
h_1 &= 1
\end{align*}
\]

Suggested by Donald Shell
N: the number of items to sort.

Worst Case Analysis of Shell Sort

- **Theorem:**
  - The worst case running time of Shell sort using Shell’s increments is \( \Theta(N^2) \).

- .
  - We will show a lower bound for the running time.
  - We will also show an upper bound for the running time.
Lower bound

We will show that there exists an input that causes the algorithm to run in $\Omega(N^2)$ time.

- Assume $N$ is a power of 2.
- Assume these $N$ elements are stored in an array indexed from 1 to $N$.
- Assume that
  - odd index values contain the $N/2$ largest elements and
  - even index values contain the $N/2$ smallest element.

1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15, 8, 16

is such a sequence.

Shell's increments are:
- 1, 2, 3, ..., $N/2$
- All increments except the last one are even.
- When we come to the last pass,
  - all largest items are in even positions and
  - all smallest items are in odd positions.

Snapshot before last pass
- 1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15, 8, 16
**Lower bound**

- The $i^{th}$ smallest number is at position $2i-1$ before the last pass.
- Restoring the $i^{th}$ element to its correct position requires:
  - $2i-1 - i = i - 1$ moves towards the beginning of the array (each move make the item go one cell left).
- Therefore to place $N/2$ smallest elements to their correct positions require amount of work in the order:
  \[ \sum_{i=1}^{N/2} i - 1 = \Omega(N^2) \]

**Upper Bound**

- A pass with increment $h_k$ consists of $h_k$ insertion sorts of about $N/h_k$ elements
  \[ h_k = 3, \quad N = 16 \]
  
  ![Diagram](Diagram.png)

  - Insertion sort of $16/3 \approx 5$ items, items are $1, 10, 4, 6, 15$
  - Insertion sort of $16/3 \approx 5$ items, items are $9, 3, 12, 14, 8$
  - Insertion sort of $16/3 \approx 5$ items

  \[ h_k \cdot (N/h_k)^2 \]
Upper Bound

- Summing over all passed

\[
\sum_{i=1}^{t} \frac{N^2}{h_i} = O(N^2 \sum_{i=1}^{t} 1/h_i) = O(N^2)
\]

since \[
\sum_{i=1}^{t} 1/h_i < 2
\]