Improving on Insertion Sort

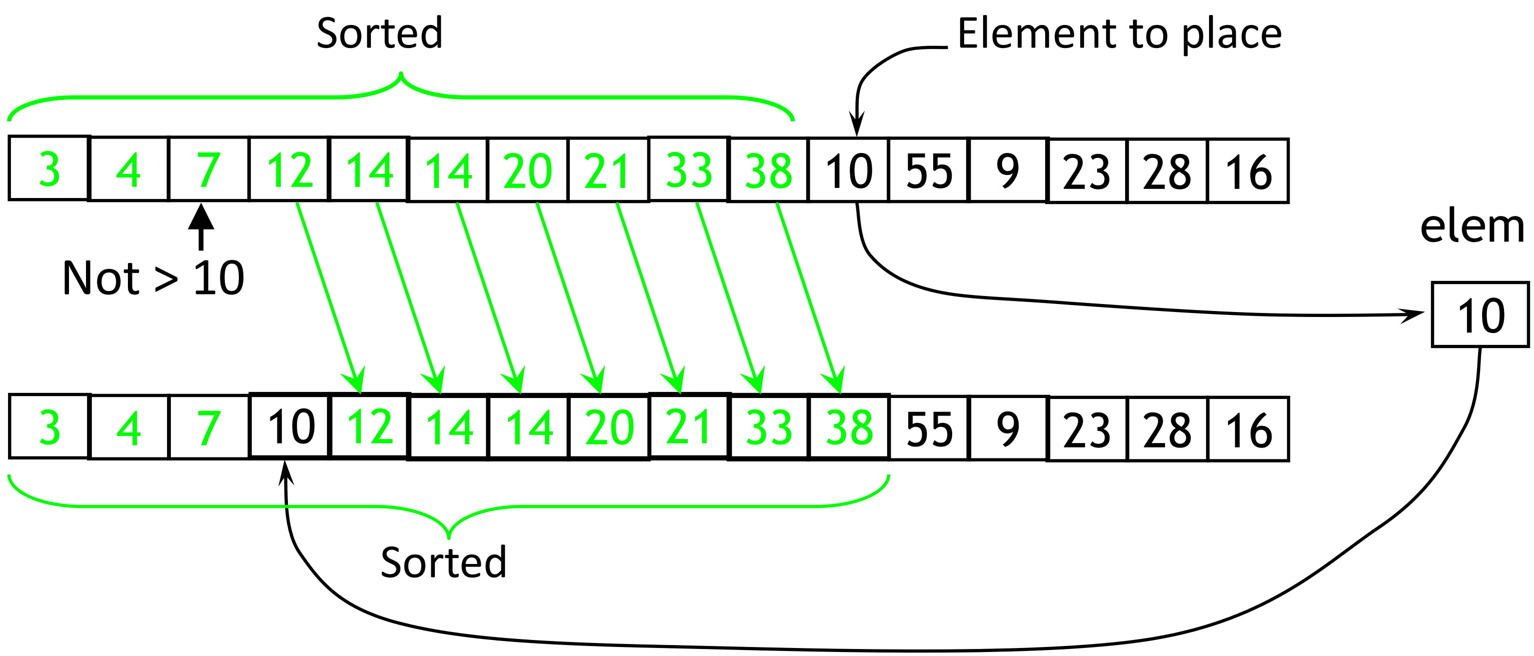
Shell Sort

Insertion Sort

Fast (O(N)) when sequence nearly sorted; otherwise s…l…o…w

```
for (idxToInsert = 1; idxToInsert < v.size ();
     ++idxToInsert)
{
  k = idxToInsert;
  elem = v[k];
  while (k >= 1 && elem < v[k - 1])
  {
    v[k] = v[k - 1];
    k = k - 1;
  }
  v[k] = elem;
}
```

One Iteration



How to Improve?

- Each time we insert an element other elements get nudged *one step* closer to where they ought to be

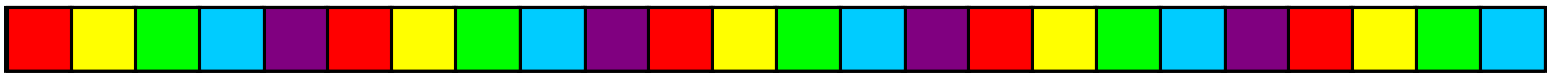
- What if we move elements a *much longer distance* each time?

- Move each element long distances initially, and decrease that distance to 1 eventually

  - This leads to *Shell sort*

Sorting Subsequences

- Vector to be sorted



- Insertion sort red elements
- Insertion sort yellow elements …
- … and finally purple elements

- Resultant array is sorted?

Elements Compared

0, 5, 10, 15, 20
1, 6, 11, 16, 21
2, 7, 12, 17, 22
3, 8, 13, 18, 23
4, 9, 14, 19

h-Sorting

- We sorted 5 sequences of elements spaced 5 apart – a (single) *h-sort* with h=5
  - Insertion sort is a 1-sort
- What if we follow the 5-sort with a 1-sort?
  - Expect each insertion would involve moving fewer elements
  - Resulting vector would be sorted

Values of ‘h’

- For large vectors we don’t want to start with a 5-sort
- Start with h = f (v.size ())
- Reduce h to 1
- Values of h form *increment* or *decrement sequence*

Values of ‘h’

- Hibbard suggests < 1, 3, 7, …, $2^k - 1$ >
  - To find initial ‘h’:

    ```
    for (h = 1; h <= N / 4; h = h * 2 + 1)
        /* empty */;
    ```

  - Repeat while h > 0
    - Do h-sort
    - h = h / 2
  - Worst case $O(N^{1.5})$

Increment Sequences

- Performance sensitive to increment/decrement sequence
- Optimal sequence not known
  - Shell proposed decrement seq. $< 1, 2, 4, 8, \ldots >$
    - Good?
  - One from Donald Knuth: $< 1, 4, 13, \ldots >$
    - Decrement by dividing by 3
  - Many others…
- So what does the code for h-sorting look like?

Analysis

- You cut the size of the array, N, by some fixed amount ($N = N / k$)
- Consequently, you have about log N stages
- Each stage takes O(N) time
- Hence, the algorithm takes O(N log N) time
- Right?

Analysis

- Wrong!

  - This analysis assumes that each stage actually moves elements closer to where they ought to be, by a fairly large amount

- What if all the red cells, for instance, contain the largest numbers in the array?

- In fact, if we just cut the size in half each time, sometimes we get $O(N^2)$ behavior!

Analysis

- What is the *real* complexity?
- Depends on sequence
- Sometimes unknown
- Some complexities determined empirically