Elementary Sorts

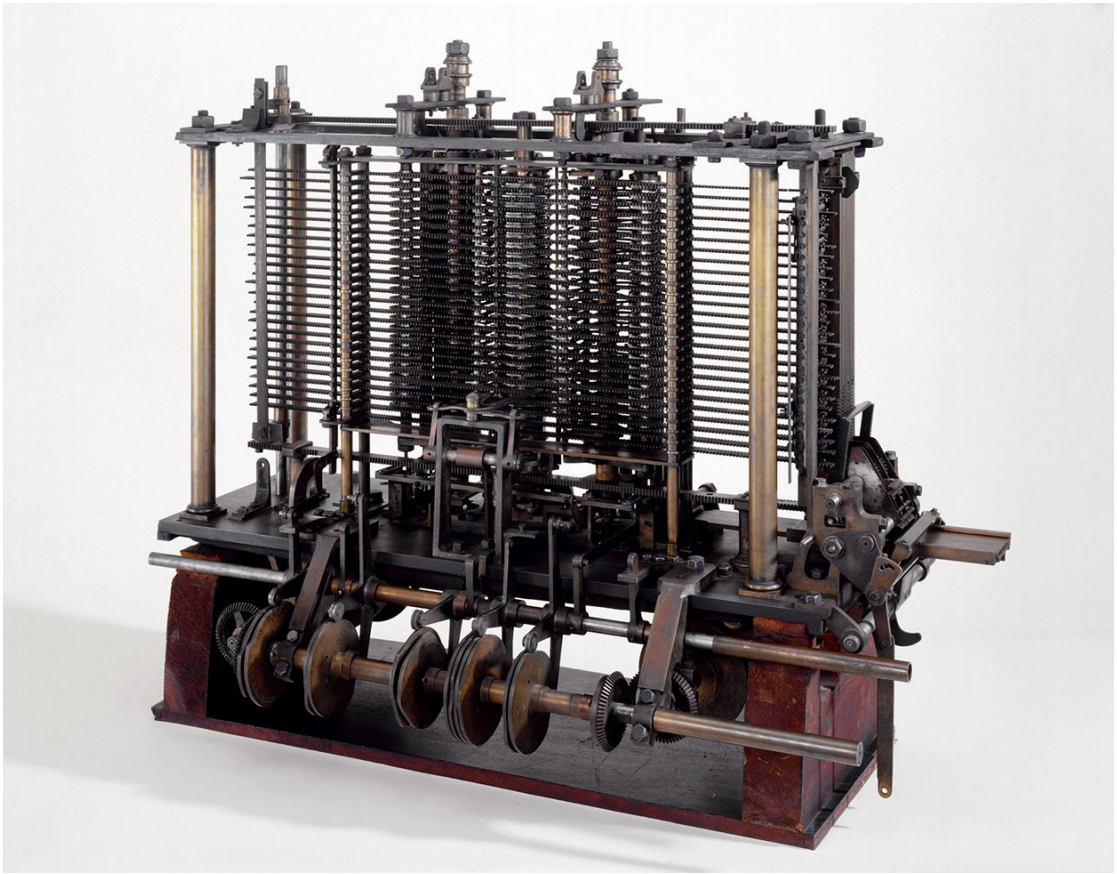
Nicholas Moore

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 2.2), Prof. Mhaskar's course slides, and `https://www.cs.princeton.edu/~rs/AlgsDS07/04Sorting.pdf`

Sorting Algorithms

An Early Computer



Charles Babbage's Analytical Engine - 1830s

Introduction

In the early days of computing, conventional wisdom stated that 30% of all computing cycles were spent sorting.

- If that fraction is lower today, it is because algorithms have become more efficient, not that sorting has become less important.
- Any time data is arranged by date, magnitude, or any other means of comparison, a sort has most likely been performed.

Over the next few lectures, we will be discussing **elementary sorts**, the simplest sorting algorithms.

Sorting Algorithms

Notable sorting algorithms:

- Selection Sort
- Insertion Sort
- Shellsort
- Heapsort
- Timsort (Python)
- Mergesort
- Quicksort
- Priority Queue
- Bubblesort
- Library sort

`https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms`

A Definition of Sorts

Sorting is the process of rearranging a sequence of objects so as to put them in some logical order.

- We will be concerned with sorting arrays primarily, since we are completely addicted to that sweet sweet constant lookup time.
- In general it doesn't matter if you're sorting into ascending or descending order, the only thing that changes is the comparator.
- For ascending order, the following property holds after sorting.

$$n < m \implies array(n) \leq array(m)$$

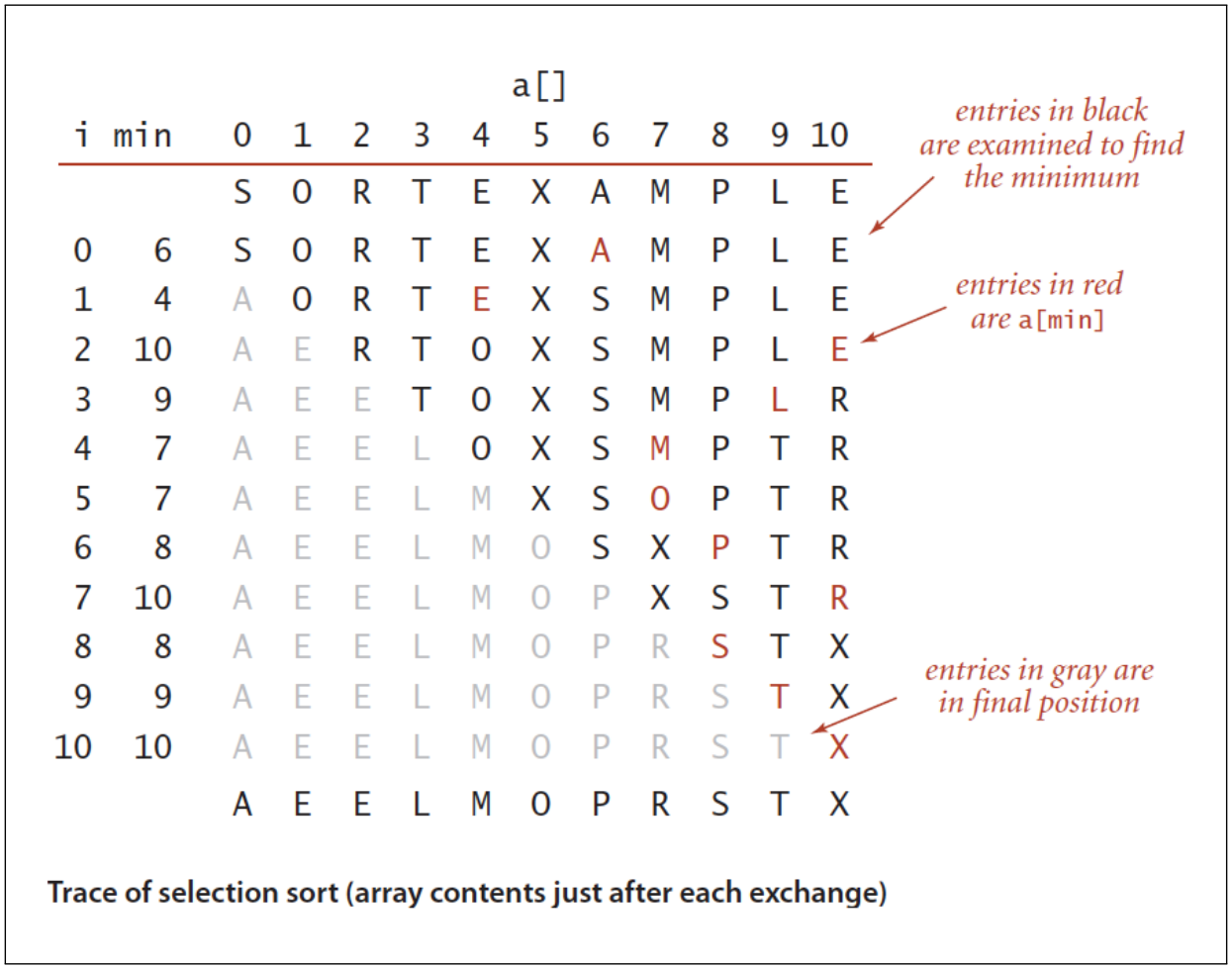
Selection Sort

Selection Sort is one of the simplest possible sorting algorithms, and is in the same category as Bubblesort.

- We proceed by building up a *sorted section* of the array, starting at the front, and moving to the back.
- First, we find the minimum element of the array.
- Then, we swap that minimum element with the first element of the array.
- Then, we examine a sub-array of $N - 1$ elements, excluding the one we just swapped.

We repeat this process of swapping the minimum to the end of the sorted section until we have processed the whole array.

Selection Sort Example



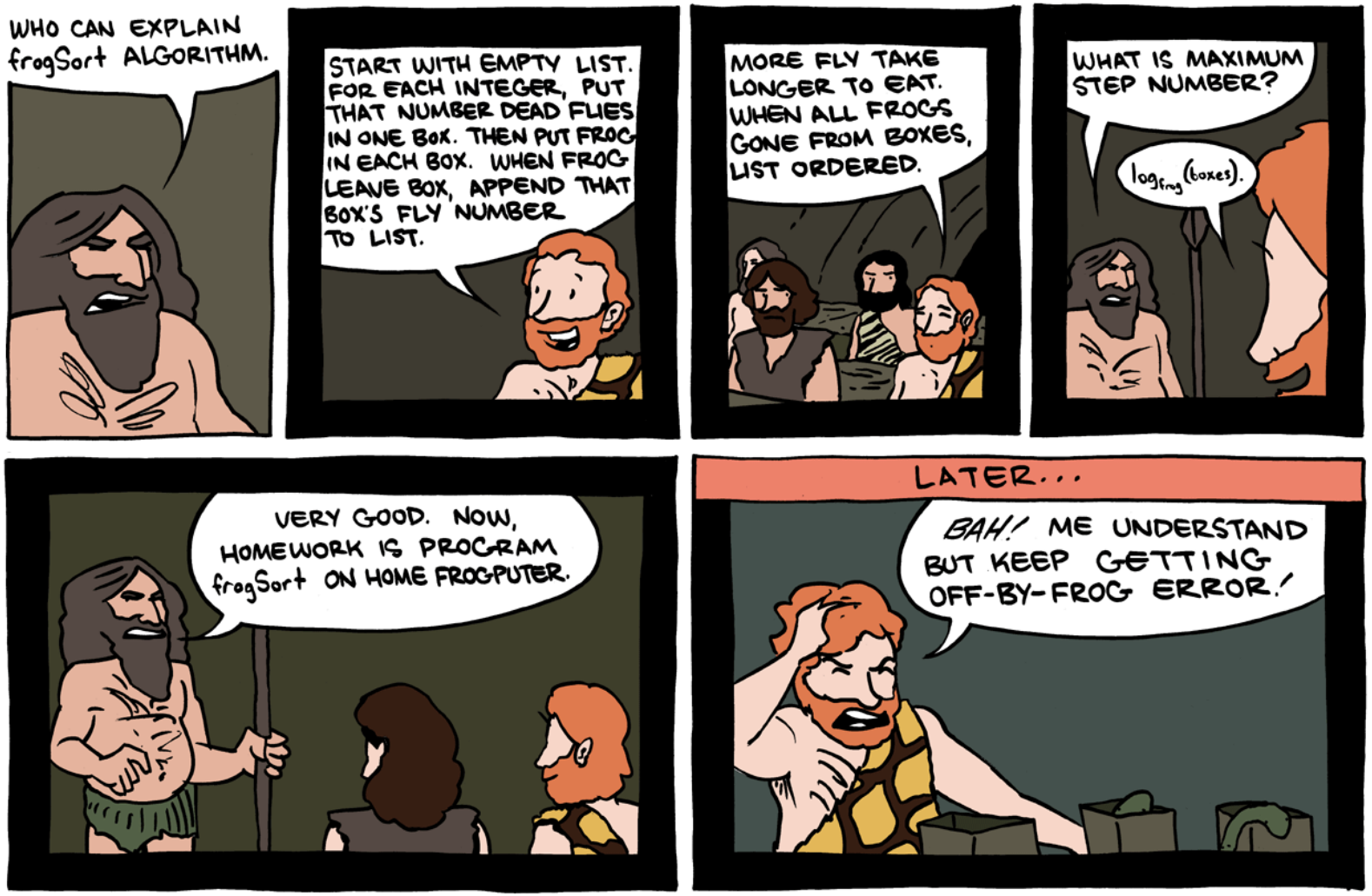
Trace of selection sort (array contents just after each exchange)

Selection Sort Analysis

Selection sort performs $(N-1)+(N-2)+\ldots+0$ comparisons and at most N exchanges.

- Complexity is $\Theta(N^2)$.
- The running time insensitive to input.
  - Even if only one swap needs to be performed in adjacent numbers, we would still run through the entire Selection Sort algorithm.
  - *What's the complexity for checking sortedness!?*
- Tiny advantage: Minimal number of write operations possible.
  - Possibly useful if all you've got for RAM is clay tablets.

Insertion Sort



Insertion Sort!

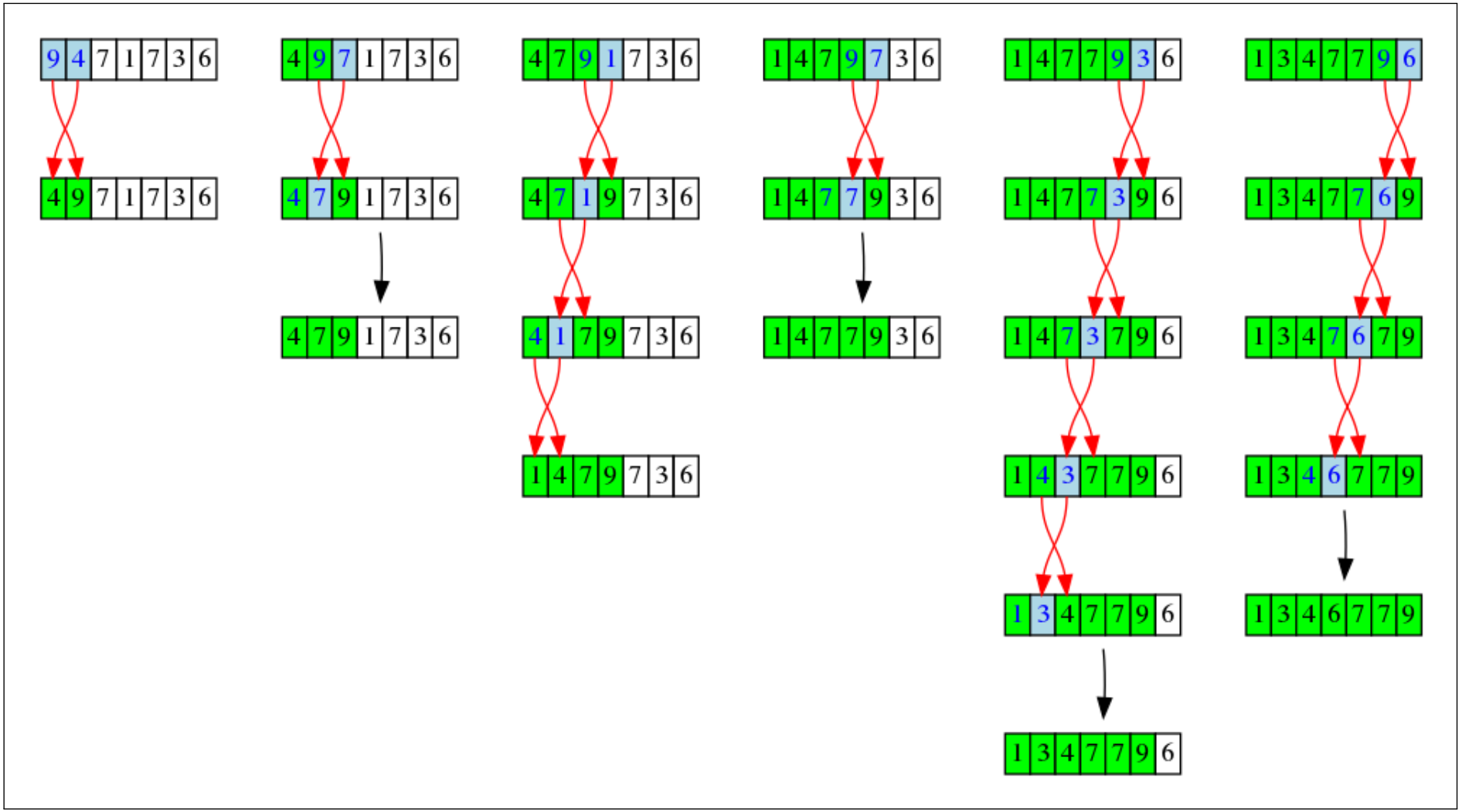
Selection Sort works, but it contains many redundant examinations!

- Insertion Sort minimizes the number of examinations that do not lead to a swapping operation.
- Insertion sort only makes one pass of the array, sorting as it goes.
- The overall algorithm is to examine each element in the array, passing the element back through the sorted section of the array until the correct spot is found for it.

Insertion Sort Pass Back Procedure

- For each element of the array:
  - The unsorted element is compared to the largest element in the sorted array, which it will always be adjacent to.
  - If the unsorted element is smaller than the element it has been compared to, those two elements are swapped, and the unsorted element is then compared to the next largest element in the sorted section.
  - This proceeds until the unsorted element is greater than or equal to the element it has been compared to. This means it has been sorted.
  - At this point, the size of the section of the array we consider sorted increases by one, and we examine the next unsorted element.

Visualizing Insertion Sort



Analysis of Insertion Sort

- In Selection Sort, the number of operations to sort the array is **invariant** with respect to the size of the sorted section.
- In Insertion Sort, the operations it takes to sort each element depends on the **degree of disorder** in the array. This makes it an *adaptive* algorithm.
  - Degree of disorder is a measure of how far off the array is from a sorted configuration.
  - A good measure of this is the number of adjacent swaps it would take to order the array.

It was the Best of Cases...

Best Case

- Zero disorder in array (i.e., already sorted)
- $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$
- Requires only linear traversal of array.
- Complexity: $\Omega(N)$

Worst Case

- Fully disordered array (i.e., reverse sorted)
- $[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]$
- Each element must be swapped back to the front of the sorted section.
- Complexity: $O(n^2)$

Shellsort!

Although insertion sort is an improvement over selection sort, we can do better!

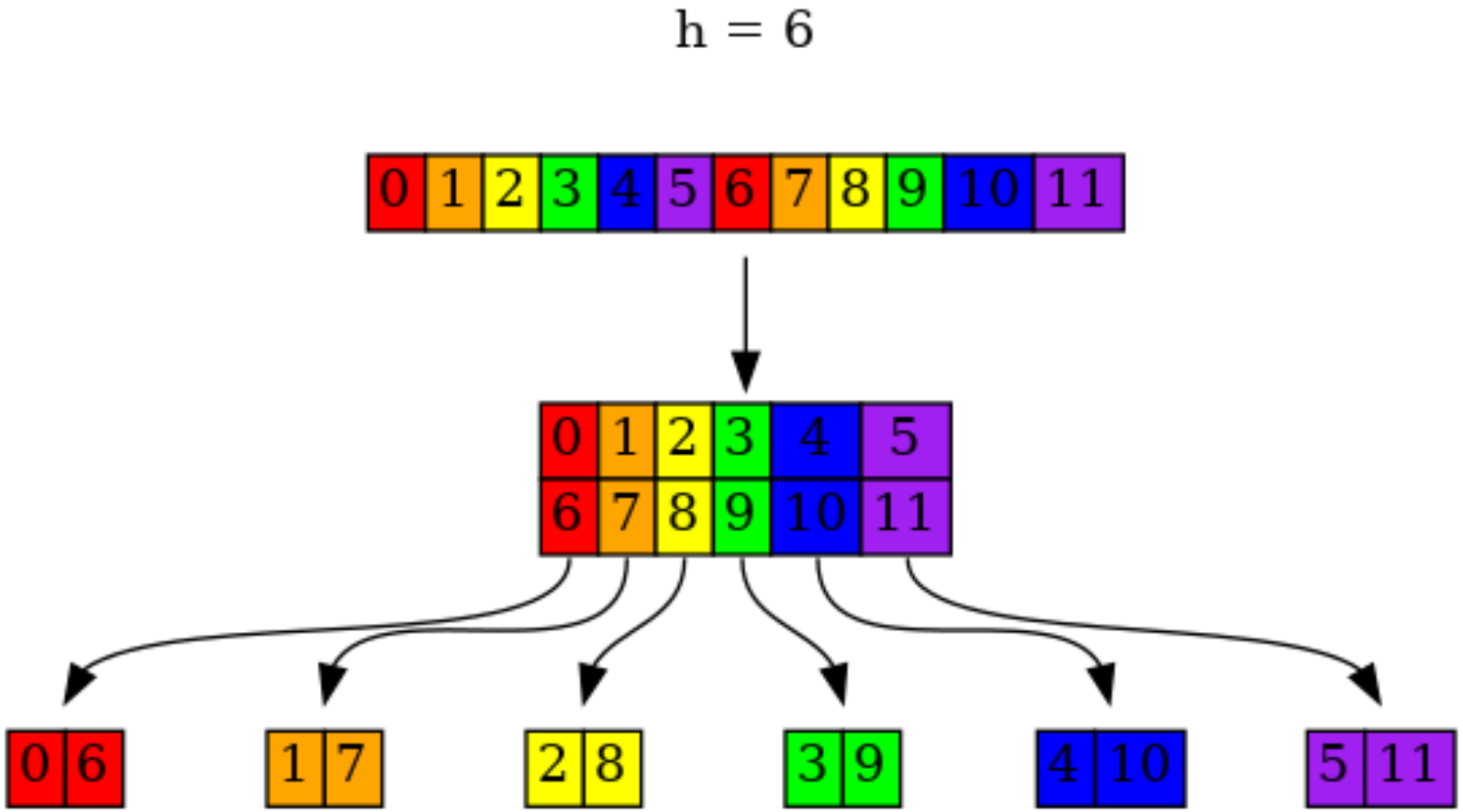
- In insertion sort, each swap only fixes one unit of disorder.
- It is possible to increase our efficiency by performing swaps between array elements that are not adjacent.
- In **shellsort**, we extract sub-arrays, sort them, and put them back.
- By selecting elements that are positionally distant, we can reduce large amounts of disorder quickly.
- Shellsort is used in embedded systems applications due to it's small program size and memory efficiency.

Interleaving and H-gaps

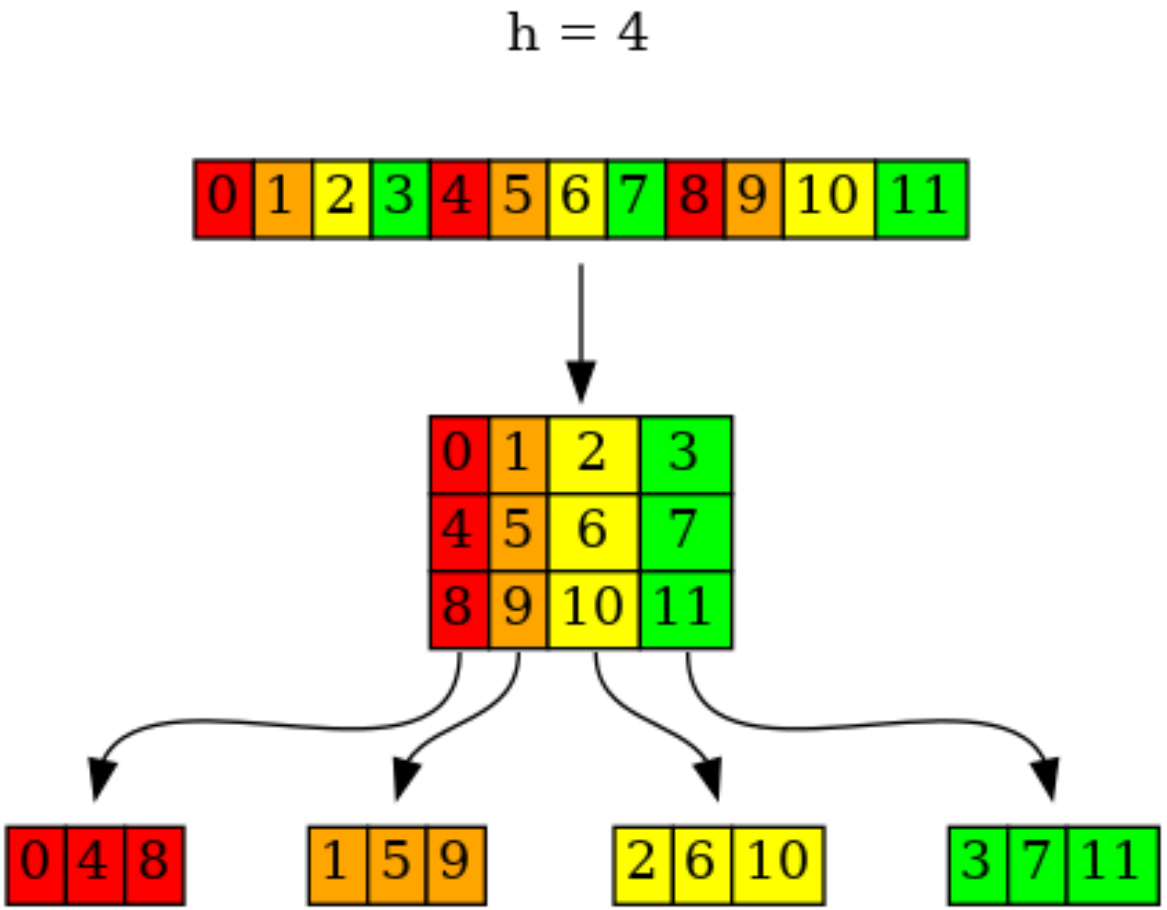
The key to shellsort is knowing how to break down the array into subarrays.

- Sub arrays are formed by taking every h^{th} element, until the end of the array is reached.
- The sequence of h-values used varies by implementation, with different sequences being proposed by different researchers (most recently in 2001).
  - One such sequence is $(1, 4, 10, 23, 57, 132, 301, 701)$
  - This order was experimentally derived. No equation has yet been discovered which can predict the next number in the sequence.
  - The worst case performance of this sequence is currently unknown.

Visualizing H-Gaps 1



Visualizing H-Gaps 2



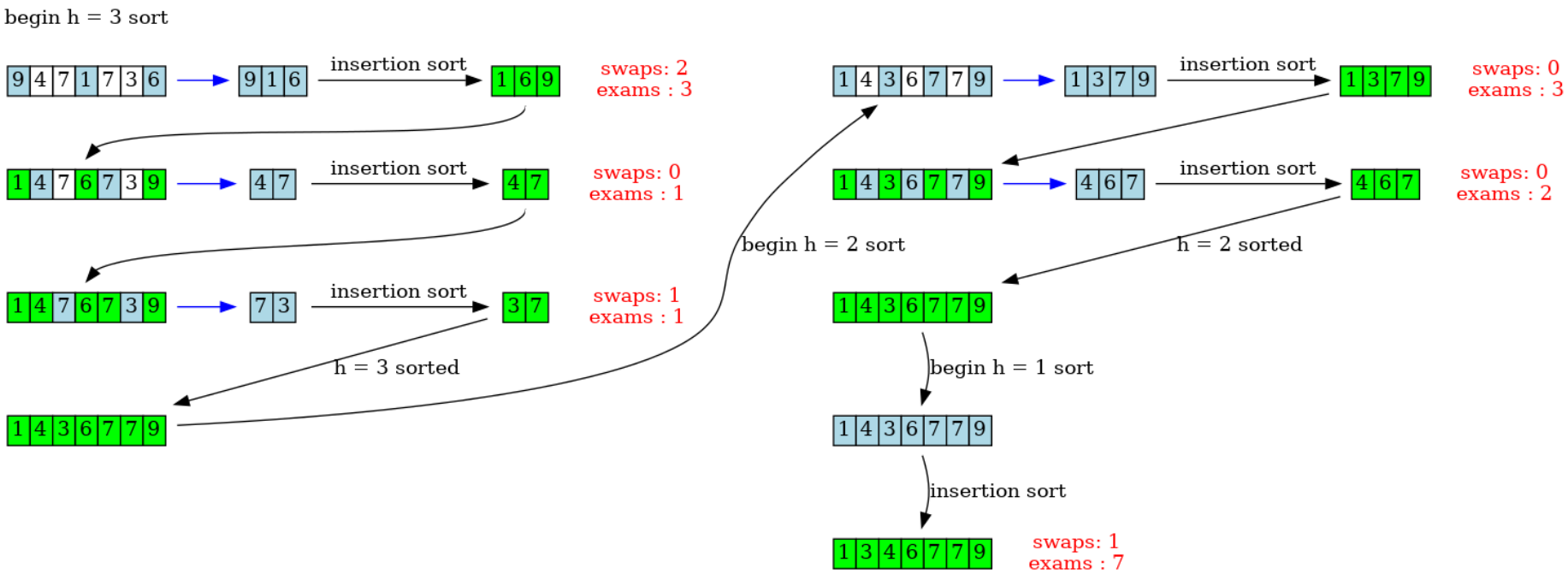
Shellsort! (cont.)

Once we have our sub-arrays established, we perform insertion sort on each one of them, and then put the elements back into the original array.

- Once all the sub arrays have been sorted and replaced for one h-value, we say that the array is **h-sorted**.
- The success of shellsort hinges on the fact that smaller h-sortings do not disorder larger ones.
  - E.g., if an array is 40-sorted, and we 14-sort it, the finished array is still 40-sorted.
- As long as the last h-value is 1 (which is just insertion sort), the array is guaranteed to be sorted!

Visualizing Shellsort

Shellsort, using an h-gap sequence 3, 2, 1.



Analysing Shellsort

- While the visualization slide implies the sub-arrays are instantiated separatedly from original array, this does not need to be the case. Our Java version of Shellsort does not allocate any new memory beyond the swap space insertion sort uses.
- The worst-case analysis for shellsort is highly dependent on the h-gap sequence.
  - Early versions (1960s) are guaranteed to be no worse than $O(n^{3/2})$
  - The best version we have results for are guaranteed to be no worse than $O(n^{4/3})$

H-Gap and That's a Wrap!

Many different h-gap sequences have been studied, but no provably best sequence has been found.

- Most researchers have approached this problem by proposing a mathematical function to generate h-gap sequences.
- The sequence used by our Java implementation was proposed by Knuth in 1973, and is easy to compute and remember.
- Time complexity for 3x+1 h-gap sequence:
  - Best: $O(N \log N)$
  - Worst: $O(N^{1.5})$
  - Average: open research problem even for uniform input distributions.

People Should Sort Themselves Out

No one has yet come to appreciate Professor Ramesh's "existentialSort."