# COMP90038 Algorithms and Complexity

## Decrease-and-Conquer-by-a-Constant

Michael Kirley

Lecture 9

Semester 1, 2016

# Decrease-and-Conquer by a Constant

In this approach, the size of the problem is reduced by some constant in each iteration of the algorithm.

A simple example is the following approach to sorting: To sort an array of length $n$, just

1. sort the first $n - 1$ items, then
2. locate the cell that should hold the last item, shift all elements to its right to the right, and place the last element.

# Insertion Sort

Sorting an array:

**function** InsertionSort($A[0..n-1]$)
  **for** $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    **while** $j \geq 0$ **and** $v < A[j]$ **do**
      $A[j+1] \leftarrow A[j]$
      $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

The idea behind insertion sort is recursive, but the code given here, using iteration, is preferable to the recursive version.

# Complexity of Insertion Sort

The for loop is traversed $n-1$ times. In the $i$th round, the test $v < A[j]$ is performed $i$ times, in the worst case.

Hence the worst-case running time is

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

What does input look like in the worst case?

# The Trick of Posting a Sentinel

If we are sorting elements from a domain that is bounded from below, that is, there is a minimal element *min*, and the array $A$ was known to have a free cell to the left of $A[0]$, then we could simplify the test. Namely, we would place *min* (a sentinel) in that cell ($A[-1]$) and change the test from

$$j \geq 0 \textbf{ and } v < A[j]$$

to just

$$v < A[j]$$

That will speed up insertion sort by a constant factor.

For this reason, extreme array cells (such as $A[0]$ in C, and/or $A[n+1]$) are sometimes left free deliberately, so that they can be used to hold sentinels; only $A[1]$ to $A[n]$ hold proper data.

# Properties of Insertion Sort

Easy to understand and implement.

Average-case and worst-case complexity both quadratic.

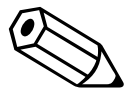
However, linear for almost-sorted input.

Some cleverer sorting algorithms perform almost-sorting and then let insertion sort take over.

Very good for small arrays (say, a couple of hundred elements).

In-place?

Stable?



# Shellsort

This is a version of insertion sort which aims at moving out-of-order elements over bigger distances (not just to the neighbouring cell as in insertion sort).

98 14 55 31 44 83 25 77 47 57 49 52 72 29 64 26 33 89 38 32 94 17

The idea is to think of the list of elements as the interleaving of $k$ separate lists (the example uses $k = 4$).

We first sort these lists separately, using 4 passes of insertion sort:

33 14 25 26 44 17 38 31 47 29 49 32 72 57 55 52 94 83 64 77 98 89

Notice how the result is almost-sorted.

# Shellsort's Passes and Gap Sequences

A final round of insertion sort finishes the job:

14 17 25 26 29 31 32 33 38 44 47 49 52 55 57 64 72 77 83 89 94 98

For larger files, start with a larger $k$ and repeat with smaller $k$s.

It is common to start from somewhere in the sequence 1, 4, 13, 40, 121, 364, 1093, ... and work backwards.

For example, for an array of size 20,000, start by 364-sorting, then 121-sort, then 40-sort, and so on.

Sequences with smaller gaps (a factor of about 2.3) appear to work better, but nobody really understands why.

# Properties of Shellsort

Fewer comparisons than insertion sort. Known to be $O(n\sqrt{n})$ for good gap sequences.

Conjectured to be $O(n^{1.25})$ but the algorithm is very hard to analyse.

Very good on medium-sized arrays (up to size 10,000 or so).

In-place?

Stable?

# Other Decrease-and-Conquer Methods

Insertion sort is a simple instance of the "decrease-and-conquer by a constant" approach.

Another is the approach to topological sorting that repeatedly removes a source.

In the next lecture we look at examples of "decrease by some factor", leading to methods with logarithmic time behaviour or better!