THE UNIVERSITY OF MELBOURNE

COMP90038
Algorithms and Complexity

Lecture 9: Decrease-and-Conquer-by-a-Constant
(with thanks to Harald Søndergaard)

Toby Murray

toby.murray@unimelb.edu.au

DMD 8.17 (Level 8, Doug McDonell Bldg)

http://people.eng.unimelb.edu.au/tobym

@tobycmurray

Decrease-and-Conquer-by-a-Constant

THE UNIVERSITY OF MELBOURNE

- 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.

Sorting n items

A:

23	9	52	12	41	83	46
0	1	2	3	4	5	6

Sorting n items

THE UNIVERSITY OF MELBOURNE

A:

A:	23	9	52	12	41	83	46
	0	1	2	3	4	5	6

Sort first n-1 items

Sorting n items

A:

23	9	52	12	41	83	46
0	1	2	3	4	5	6

Sort first n-1 items

Sorting n items

A:

0	1	2	3	4	5	6
9	12	23	41	52	83	46

Sorting n items

A:							
	9	12	23	41	52	83	46
	0	1	2	3	4	5	6

Sorting n items

A:

9	12	23	41	52	83	46
0	1	2	3	4	5	6

Sorting n items

A:

9	12	23	41	52	83	46
0	1	2	3	4	5	6

Sorting n items

A:	9	12	23	41	52	83	46
	0	1	2	3	4	5	6

Sorting n items

A:

9	12	23	41	52	83	46
0	1	2	3	4	5	6

Sorting n items

A:

0	1	2	3	4	5	6
9	12	23	41	46	52	83

Sorting n items

A:

9	12	23	41	52	83	46
0	1	2	3	4	5	6

Sorting n items

A:

9	12	23	41	52	83	46
0	1	2	3	4	5	6

v: 46

Sorting n items

A:

9	12	23	41	52	83	46
0	1	2	3	4	5	6

A[j]

v: 46

Sorting n items

A:

9	12	23	41	52	83	**83**
0	1	2	3	4	5	6

↑
A[j]

v: 46

Sorting n items

A:

9	12	23	41	52	83	83
0	1	2	3	4	5	6

↑

A[j]

v: 46

Sorting n items

A:

9	12	23	41	52	**52**	83
0	1	2	3	4	5	6

A[j]

v: 46

Sorting n items

A:

9	12	23	41	52	52	83
0	1	2	3	4	5	6

↑
A[j]

v: 46

Sorting n items

A:

9	12	23	41	**46**	52	83
0	1	2	3	4	5	6

A[j]

v: 46

Sorting n items

A:

9	12	23	41	46	52	83
0	1	2	3	4	5	6

A[j]

v: 46

Insertion Sort

- Sorting an array A[0]..A[n – 1]:
- To sort A[0] .. A[i] first sort A[0] .. A[i-1], then insert A[i] in its proper place

$$
\begin{array}{l}
\textbf{function } \textsc{InsertionSort}(A[\cdot], n) \\
\quad \textbf{for } i \leftarrow 1 \textbf{ to } n-1 \textbf{ do} \\
\quad\quad v \leftarrow A[i] \\
\quad\quad j \leftarrow i-1 \\
\quad\quad \textbf{while } j \geq 0 \textbf{ and } v < A[j] \textbf{ do} \\
\quad\quad\quad A[j+1] \leftarrow A[j] \\
\quad\quad\quad j \leftarrow j-1 \\
\quad\quad A[j+1] \leftarrow v
\end{array}
$$

Complexity of Insertion Sort

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

- Hence the worst-case running time is

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

- What does input look like in the worst case?

Complexity of Insertion Sort

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

- Hence the worst-case running time is

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i$$

- What does input look like in the worst case?

Complexity of Insertion Sort

- The for loop is traversed n – 1 times. In the ith 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

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

  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.

Posting a Sentinel

A:

9	23	52	12	41	83	46
0	1	2	3	4	5	6

↑

A[j]

Test required: $j \geq 0$ and $v < A[j]$

Posting a Sentinel

A:	-1	9	23	52	12	41	83	46
	0	1	2	3	4	5	6	7

A[j]

Test required: $v < A[j]$

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?

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? yes
- Stable?

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? yes
- Stable? ?

Insertion Sort Stability

0	1	2	3
key: 4 val: ab	key: 3 val: bc	key: 4 val: de	key: 3 val: fg

Insertion Sort Stability

key: 4 val: ab	key: 3 val: bc	key: 4 val: de	key: 3 val: fg
0	1	2	3

Insertion Sort Stability

0	1	2	3
key: 3 val: bc	key: 4 val: ab	key: 4 val: de	key: 3 val: fg

Insertion Sort Stability

key: 3 val: bc	key: 4 val: ab	key: 4 val: de	key: 3 val: fg
0	1	2	3

Insertion Sort Stability

0	1	2	3
key: 3 val: bc	key: 3 val: fg	key: 4 val: ab	key: 4 val: de

Insertion Sort Stability

key: 3 val: bc	key: 3 val: fg	key: 4 val: ab	key: 4 val: de
0	1	2	3

Stable

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?

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? yes
- Stable?

Properties of Insertion Sort

THE UNIVERSITY OF MELBOURNE

- 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? yes
- Stable? yes

Shellsort: Motivation

A:

9	8	7	6	5	4	3
0	1	2	3	4	5	6

Shellsort: Motivation

A:

9	8	7	6	5	4	3
0	1	2	3	4	5	6

A:

8	**9**	7	6	5	4	3
0	1	2	3	4	5	6

Shellsort: Motivation

A:

9	8	7	6	5	4	3
0	1	2	3	4	5	6

A:

8	**9**	7	6	5	4	3
0	1	2	3	4	5	6

A:

7	**8**	**9**	6	5	4	3
0	1	2	3	4	5	6

Shellsort: Motivation

A:

9	8	7	6	5	4	3
0	1	2	3	4	5	6

A:

8	**9**	7	6	5	4	3
0	1	2	3	4	5	6

A:

7	**8**	**9**	6	5	4	3
0	1	2	3	4	5	6

It would be better if we could move the 9, 8, etc. to the right faster

Shellsort: Motivation

A:

9	8	7	6	5	4	3
0	1	2	3	4	5	6

Shellsort: Motivation

THE UNIVERSITY OF MELBOURNE

Sort the yellow entries

A:

9	8	7	6	5	4	3
0	1	2	3	4	5	6

Shellsort: Motivation

THE UNIVERSITY OF MELBOURNE

Sort the yellow entries

A:

9	8	7	6	5	4	3
0	1	2	3	4	5	6

A:

6	8	7	9	5	4	3
0	1	2	3	4	5	6

Shellsort: Motivation

Sort the yellow entries

A:

9	8	7	6	5	4	3
0	1	2	3	4	5	6

A:

6	8	7	9	5	4	3
0	1	2	3	4	5	6

A:

3	8	7	6	5	4	9
0	1	2	3	4	5	6

Shellsort: Motivation

THE UNIVERSITY OF MELBOURNE

A:	3	8	7	6	5	4	9
	0	1	2	3	4	5	6

Shellsort: Motivation

Sort the blue entries

A:

3	8	7	6	5	4	9
0	1	2	3	4	5	6

Shellsort: Motivation

Sort the blue entries

A:	3	8	7	6	5	4	9
	0	1	2	3	4	5	6

A:	3	5	7	6	8	4	9
	0	1	2	3	4	5	6

Shellsort: Motivation

Sort the blue entries

A:

0	1	2	3	4	5	6
3	8	7	6	5	4	9

A:

0	1	2	3	4	5	6
3	5	7	6	8	4	9

Sort the pink entries

Shellsort: Motivation

THE UNIVERSITY OF MELBOURNE

Sort the blue entries

A:	3	8	7	6	5	4	9
	0	1	2	3	4	5	6

A:	3	5	7	6	8	4	9
	0	1	2	3	4	5	6

Sort the pink entries

A:	3	5	4	6	8	7	9
	0	1	2	3	4	5	6

Shellsort: Motivation

Sort the blue entries

A:

3	8	7	6	5	4	9
0	1	2	3	4	5	6

A:

3	5	7	6	8	4	9
0	1	2	3	4	5	6

Sort the pink entries

A:

3	5	4	6	8	7	9
0	1	2	3	4	5	6

Notice how it is now **almost sorted**

Shellsort: Motivation

Sort the blue entries

A:

3	8	7	6	5	4	9
0	1	2	3	4	5	6

A:

3	5	7	6	8	4	9
0	1	2	3	4	5	6

Sort the pink entries

A:

3	5	4	6	8	7	9
0	1	2	3	4	5	6

Notice how it is now **almost sorted**

Now do a final round of insertion sort over the entire array

Shellsort: Motivation

THE UNIVERSITY OF MELBOURNE

Sort the blue entries

A:	3	8	7	6	5	4	9
	0	1	2	3	4	5	6

A:	3	5	7	6	8	4	9
	0	1	2	3	4	5	6

Sort the pink entries

A:	3	5	4	6	8	7	9
	0	1	2	3	4	5	6

Notice how it is now **almost sorted**

Now do a final round of insertion sort over the entire array

A:	3	4	5	6	7	8	9
	0	1	2	3	4	5	6

Shellsort

THE UNIVERSITY OF MELBOURNE

- We just did a shellsort for k=3
- In general:
  - Think of the array as an interleaving of k lists
  - Sort each list separately using insertion sort
  - Then sort the resulting entire array using a final pass of insertion sort

Shellsort Passes and Gap Sequences

- For large files, start with larger *k* and then 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.
  - what is the sequence?
- 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 worst-case $O(n\sqrt{n})$ 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?

Properties of Shellsort

- Fewer comparisons than insertion sort. Known to be worst-case $O(n\sqrt{n})$ 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? yes
- Stable?

Properties of Shellsort

THE UNIVERSITY OF MELBOURNE

- Fewer comparisons than insertion sort. Known to be worst-case $O(n\sqrt{n})$ 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? yes
- Stable? ?

Shellsort: Stability

A:

0	1	2	3	4	5	6
1	7	4	6	4	8	9

Shellsort: Stability

A:

1	7	4	6	4	8	9
0	1	2	3	4	5	6

after sorting
the blues

Shellsort: Stability

A:

1	7	4	6	4	8	9
0	1	2	3	4	5	6

A:

1	4	4	6	7	8	9
0	1	2	3	4	5	6

after sorting
the blues

Shellsort: Stability

A:

1	7	4	6	4	8	9
0	1	2	3	4	5	6

A:

1	4	4	6	7	8	9
0	1	2	3	4	5	6

after sorting the blues

relative order of the two 4s has changed!

Properties of Shellsort

- Fewer comparisons than insertion sort. Known to be worst-case $O(n\sqrt{n})$ 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?

Properties of Shellsort

- Fewer comparisons than insertion sort. Known to be worst-case $O(n\sqrt{n})$ 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? yes
- Stable?

Properties of Shellsort

- Fewer comparisons than insertion sort. Known to be worst-case $O(n\sqrt{n})$ 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? yes
- Stable? no

Other Instances of Decrease-and-Conquer by a Constant

- 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!