CS2010: Data Structures and Algorithms II

Topic 01: Sorting Algorithms

Ivana.Dusparic@scss.tcd.ie

Lecture Outline

- Introduce algorithm design techniques
- Review sorting algorithms
  - Insertion sort
- Learn some new ones
  - Bubble sort
  - Selection sort
  - Shellsort
  - Mergesort
  - Quicksort
- Analyse and classify by
  - Order of growth
  - Best, average, worst running time
  - Design approach
  - Stable vs unstable

- Textbook and lecture notes: Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne

Algorithm Design Approaches

Algorithm design

- Brute-force/exhaustive search
- Decrease and conquer
- Divide and conquer
- Transform and conquer
- Greedy
- Dynamic programming

Introduction to the Design and Analysis of Algorithms, Anany Levitin, 3rd edition, Pearson, 2012

Brute-force/exhaustive search

- Systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement
- Often simplest to implement but not very efficient
- Impractical for all but smallest instances of a problem
- Examples:
  - Selection sort
  - Bubble sort
  - In graphs – depth-first search (DFS), breadth-first search (BFS)

Decrease and conquer

- Establish relationship between a problem and a smaller instance of that problem
- Exploit that relationship top down or bottom up to solve the bigger problem
- Naturally implemented using recursion
- Examples
  - Insertion sort
  - In graphs – topological sorting

Divide and conquer

- Divide a problem into several subproblems of the same type, ideally of the same size
- Solve subproblems, typically recursively
- If needed, combine solutions
- Examples:
  - Mergesort
  - Quicksort
  - Binary tree traversal – preorder, inorder, postorder
    - Visit root, its left subtree, and its right subtree

Transform and conquer

- Modify a problem to be more amenable to solution, then solve
  - Transform to a simpler/more convenient instance of the same problem – *instance simplification*
  - Transform to a different representation of the same instance – *representation change*
  - Transform to an instance of a different problem for which an algorithm is available – *problem reduction*
- Examples:
  - Balanced search trees – AVL trees, 2-3 trees
  - Gaussian elimination – solving a system of linear equations

Dynamic programming

- Similar to divide and conquer, solves problems by combining the solutions to subproblems
  - In divide and conquer subproblems are disjoint
  - In dynamic programming, subproblems overlap, ie share subsubproblems
    - Solutions to those are stored, index and reused

- Examples:
  - A more efficient solution to Knapsack problem
  - Warshall's and Floyd's shortest path algorithms

Greedy

- Always make the choice that looks best at the moment
  - Does not always yield the most optimal solution, but often does
- Examples
  - Graphs:
    - Dijkstra – find the shortest path from the source to the vertex nearest to it, then second nearest etc
    - Prim
    - Kruskal
  - Strings
    - Huffman coding tree

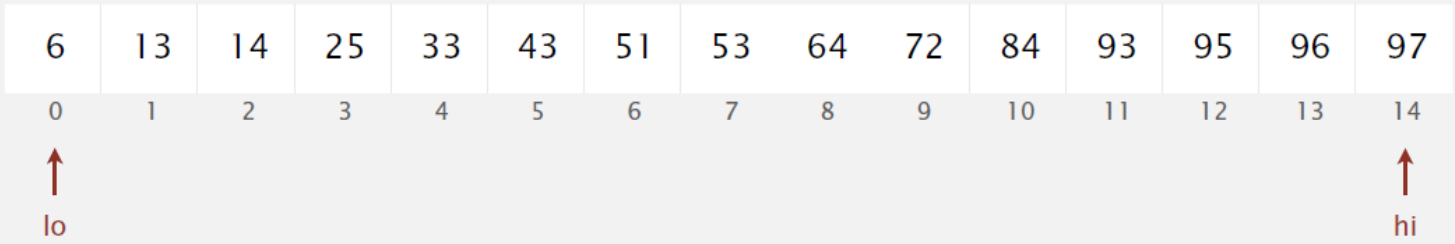
Example: Binary Search

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



Binary Search is an example of what kind of algorithm:

› A – divide and conquer

› B – decrease and conquer

› C – brute-force

› D – greedy

https://responseware.turningtechnologies.eu/responseware/polling or use Turning Point App

Session id:

Fun fact: ex-bug in JDK implementation of binary search

› https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

– Blog post by Joshua Bloch, Software Engineer who implemented binary search in java.util.Arrays

– Undiscovered for 9 years

```
1:     public static int binarySearch(int[] a, int key) {
2:         int low = 0;
3:         int high = a.length - 1;
4:
5:         while (low <= high) {
6:             int mid = (low + high) / 2;
7:             int midVal = a[mid];
8:
9:             if (midVal < key)
10:                 low = mid + 1
11:             else if (midVal > key)
12:                 high = mid - 1;
13:             else
14:                 return mid; // key found
15:         }
16:         return -(low + 1);  // key not found.
17:     }
```

Sorting Algorithms

Sorting

- Sort data in order
  - Numbers in ascending/descending order
  - Strings alphabetically
  - Dates chronologically
  - etc
- Total order
  - Ascending $x_0 \leq x_1 \leq x_2 \leq x_3 \leq .... \leq x_{n-1}$
  - Descending $x_0 \geq x_1 \geq x_2 \geq x_3 \geq .... \geq x_{n-1}$

Total order

$$x_0 \leq x_1 \leq x_2 \leq x_3 \leq .... \leq x_{n-1}$$

› Is a binary relation ≤ that satisfies

- Antisymmetry: if both $v \leq w$ and $w \leq v$, then $v = w$.
- Transitivity: if both $v \leq w$ and $w \leq x$, then $v \leq x$.
- Totality: either $v \leq w$ or $w \leq v$ or both.

Useful sorting abstractions

Less. Is item v less than w ?

```
private static boolean less(Comparable v, Comparable w)
{  return v.compareTo(w) < 0;  }
```

Exchange. Swap item in array a[] at index i with the one at index j.

```
private static void exch(Comparable[] a, int i, int j)
{
   Comparable swap = a[i];
   a[i] = a[j];
   a[j] = swap;
}
```

- public interface Comparable <T> -This interface imposes a total ordering on the objects of each class that implements it.
- public int compareTo (Item x)
- Implemented by String, Integer, Double, Short, Calendar, Year, etc

Performance Analysis

- Cost models
  - Running time
  - Memory cost
- Methods to measure/express
  - Tilde notation, T(n) – counting number of executions of certain operations as a function of input size n
- Order of growth classification
  - Big Theta Θ(n) – asymptotic order of growth
  - Big Oh O(n) - upper bound
  - Big Omega Ω (n) – lower bound

Performance Analysis

- Time complexity
  - Worst Case Analysis – usually done
    - Upper bound on running time of an algorithm
    - Must know the case that causes the maximum number of operations to be performed, eg in linear search, if the element is not in the array
  - Average – not easy to do in practice
    - Take all possible inputs and calculate computing time for all of the inputs, and average
    - Must know/predict distribution of cases
  - Best – is it any use if worst case bad?
    - Lower bound on running time of an algorithm
    - Must know the case that causes the minimum number of operations to be performed

Refresher

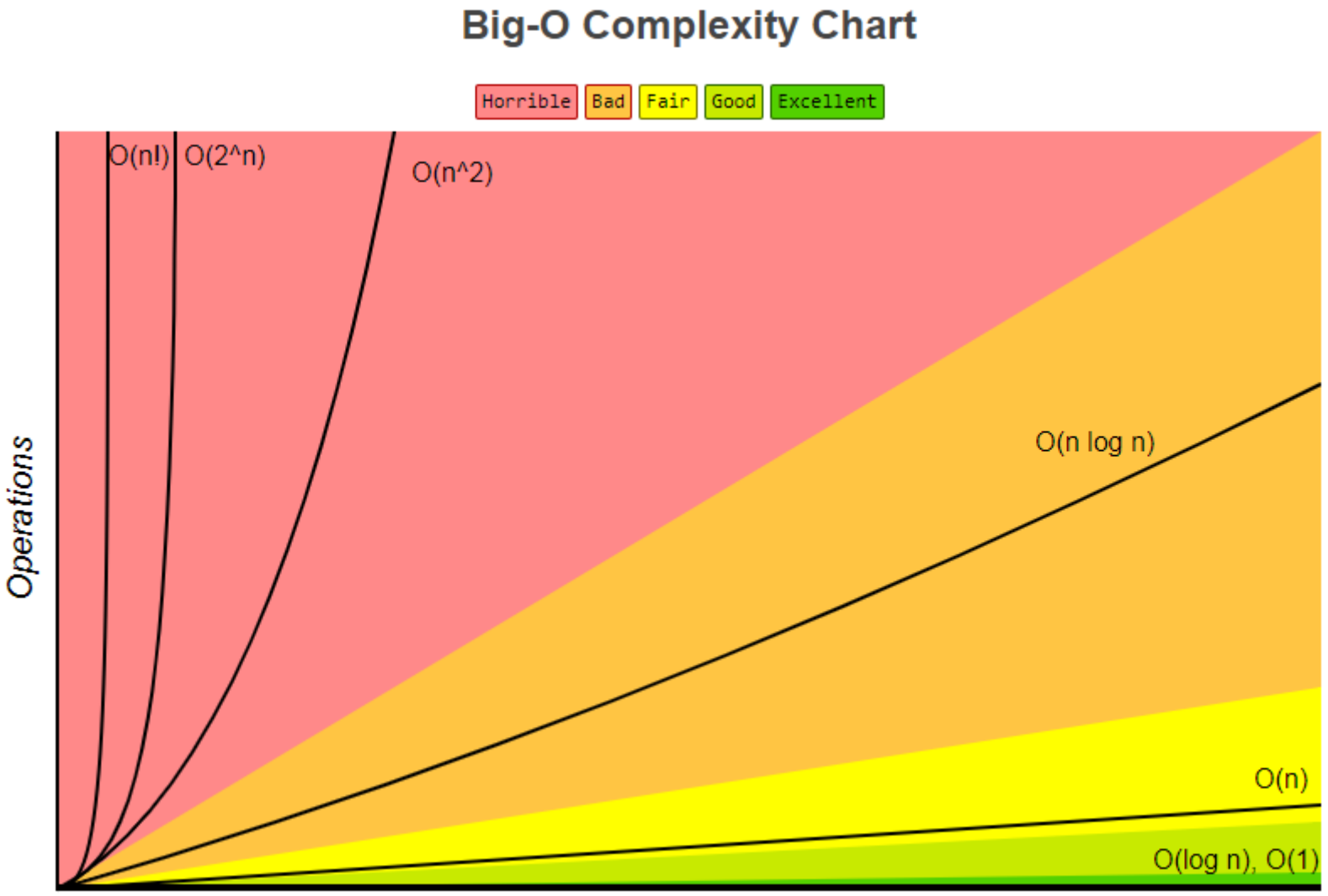
- Refer to semester 1 slides for more details on performance analysis

Common order-of-growth classifications

			uble loop	check all pairs	4

N^3	**cubic**	`for (int i = 0; i < N; i++)`
		`  for (int j = 0; j < N; j++)`
		`  for (int k = 0; k < N; k++)`
		`  { ... }`
			triple loop	check all triples	8

2^N	**exponential**	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$



http://bigocheatsheet.com/

WΛSSIM CHΞGHΛM
@manekinekko

Follow

We should reconsider big O notations...

Θ(1) ⟶ Θ(😎)
Θ(log(n)) ⟶ Θ(😁)
Θ((log(n))^c) ⟶ Θ(😇)
Θ(n) ⟶ Θ(🙂)
Θ(n log(n)) ⟶ Θ(😕)
Θ(n^2) ⟶ Θ(😖)
Θ(n^c) ⟶ Θ(😭)
Θ(c^n) ⟶ Θ(😡)
Θ(n!) ⟶ Θ(😲)

9:02 AM - 24 Oct 2018

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

http://bigocheatsheet.com/

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Visualisation of sorting algorithm performance

- https://www.youtube.com/watch?v=ZZuD6iUe3Pc
- Tested for different types of input

Why do we need so many then?

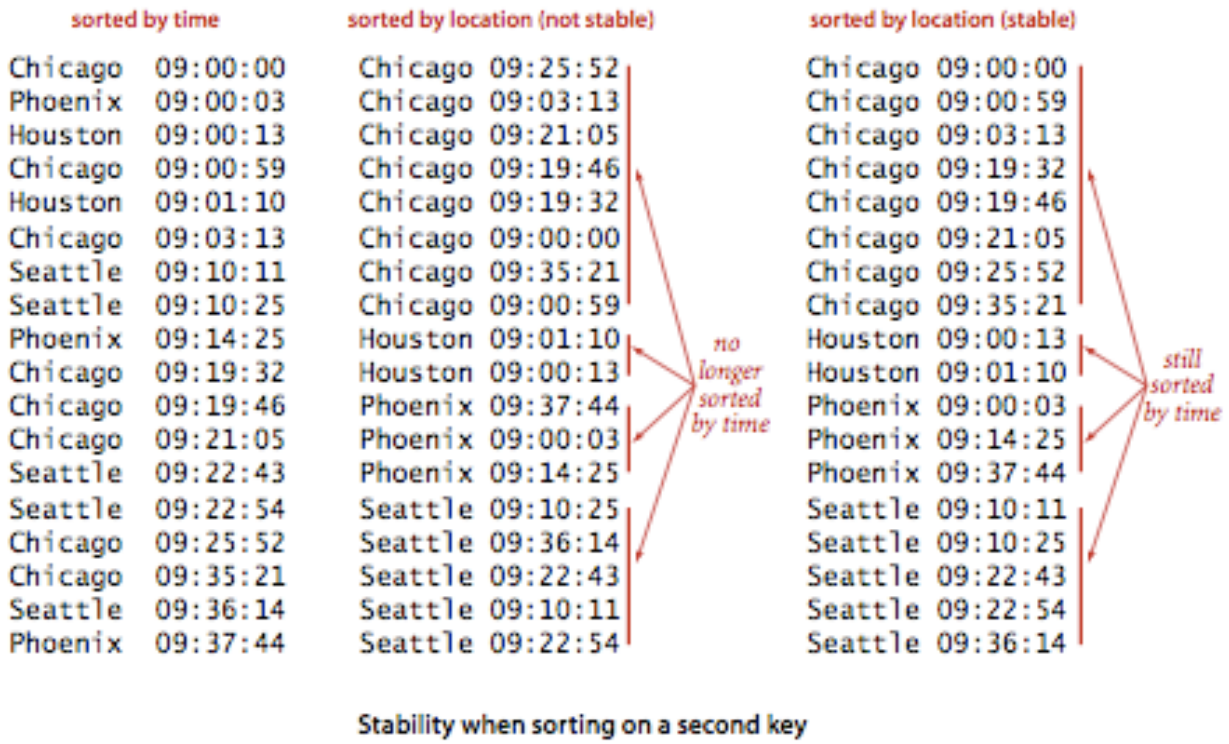
- No Free Lunch Theorem
- Different applications/different behaviour based on input
- Examples
  - Merge sort – useful for linked lists
  - Heapsort – sorting arrays, predictable, very little extra RAM
  - Quicksort – excellent average-case behaviour
  - Insertion sort – good if your list is already almost sorted
  - Bubble sort – if small enough data set, it is the simplest to implement

- Also, a handy way to learn different algorithm design strategies on the same example!

Stability of Sorting Algorithms

- Stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted
- Do we care?
  - NO: When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key
  - NO: If all keys are different.
  - YES: if duplicate keys and want to maintain original order by eg secondary key.
  - When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. Stability is also not an issue if all keys are different.

Stability of Sorting Algorithms



Stability when sorting on a second key

› Stable sorting algorithms: Insertion sort, bubble sort, merge sort

Memory requirements/In-place algorithms

- Transforms input without additional auxiliary data structure, eg array
- A small amount of extra storage space is allowed for auxiliary variables
- The input is usually overwritten by the output as the algorithm executes
- In-place algorithm updates input sequence only through replacement or swapping of elements

- Affects space complexity of an algorithm
- Selection, insertion, shell, quick

Checkpoint – is the material clear enough?

Which of these O(n) has the highest complexity (worst worst performance)?

- A – O(log n)
- B – O (n log (n))
- C – O (n^2)
- D – O (2^n)

https://responseware.turningtechnologies.eu/responseware/polling or use Turning Point App

Session id:

Finally: let’s do actual sorting algorithms

Review: Insertion Sort

Insertion sort

- Algorithm
  1. Start from 1st element of the array
  2. Shift element back until you find a smaller element – maintain the array from 0 to (current position) sorted.
  3. Continue to next element
  4. Repeat (2) and (3) until the end of the array

Insertion sort

i=1

62 83 18 53 07 17 95 86 42 69 25 28

i=2

62 83 18 53 07 17 95 86 42 69 25 28

Insertion sort

i=1

62 83 18 53 07 17 95 86 42 69 25 28

i=2

18 62 83 53 07 17 95 86 42 69 25 28

i=3

18 62 83 53 07 17 95 86 42 69 25 28

Insertion sort

i=1

62 83 18 53 07 17 95 86 42 69 25 28

i=2

18 62 83 53 07 17 95 86 42 69 25 28

i=3

18 53 62 83 07 17 95 86 42 69 25 28

i=4

18 53 62 83 07 17 95 86 42 69 25 28

. . .

Insertion sort implementation – ints in Java

```
public static int[] insertionSort(int[] input){

        int temp;
        for (int i = 1; i < input.length; i++) {
            for(int j = i ; j > 0 ; j--){
                if(input[j] < input[j-1]){
                    temp = input[j];
                    input[j] = input[j-1];
                    input[j-1] = temp;
                }
            }
        }
        return input;
    }
```

Insertion sort implementation – ints in Java

```
void insertionSort(int numbers[])
{
  int i, j, index;
  int size = numbers.size;

  for (i = 1; i < size; i++)
  {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j − 1] > index))
        {
              numbers[j] = numbers[j − 1];
              j = j − 1;
        }
        numbers[j] = index;
  }
}
```

Insertion sort properties

- Performance
  - Best case?
  - Worst case?
- O?
- Stable?
- In-place?

Insertion sort properties

- Performance
  - Best case – array is sorted
  - Worst case – array is sorted in reverse order
- O?
  - Comparisons and swaps (lecture 3.2 in semester 1)
- But best case performance is $\Omega(n)$
- Stable? YES
- In-place? YES
- Example use: often used to speed up other algorithms like quicksort: you quicksort until the partitions are around 8 items in size and then insertion sort the whole array. This tends to be faster than just allowing quicksort to complete down to one-item partitions.

Insertion sort exercise

(c) Consider the code for Insertion sort given below.

```
public void iterInsertSort(Comparable[] list) {
  int N = list.length;
  for(int i=1; i<N; i++) {
     for(int j=i; j>0; j--) {
        if(list[j].compareTo(list[j-1]) < 0) {
           Comparable temp = list[j];
           list[j] = list[j-1];
           list[j-1] = temp;
        }
     }
  }
}
```

Assume you are given an array containing the following integers 5, 4, 2, 5, 1.

Provide the trace of the array content at each time step of Insertion sort algorithm

Insertion sort exercise

i	j	array

i	j	array

i	j	array

Bubble Sort

Bubble sort

- Make multiple passes through a list
- In each pass, compare adjacent items and exchange those that are out of order
- Each pass through the list places the next largest value in its proper place

Bubble sort – int array in Java

```
public static void bubbleSort(int[] numArray) {

    int n = numArray.length;
    int temp = 0;

    for (int i = 0; i < n; i++) {
        for (int j = 1; j < (n - i); j++) {

            if (numArray[j - 1] > numArray[j]) {
                temp = numArray[j - 1];
                numArray[j - 1] = numArray[j];
                numArray[j] = temp;
            }

        }
    }
}
```

Bubble Sort

62.0 , 83.0 , 18.0 , 53.0 , 7.0 , 17.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,
swapping 83.0 and 18.0

62.0 , 18.0 , 83.0 , 53.0 , 7.0 , 17.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,
swapping 83.0 and 53.0

62.0 , 18.0 , 53.0 , 83.0 , 7.0 , 17.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,
swapping 83.0 and 7.0

62.0 , 18.0 , 53.0 , 7.0 , 83.0 , 17.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,
swapping 83.0 and 17.0

...

Bubble sort properties

- Performance
  - Best case?
  - Worst case?
- O?
- Stable?
- In-place?

Bubble sort properties

- Performance
  - Best case – array is sorted
  - Worst case – array is sorted in reverse order
- O?
  - Two nested loops O(n^2)
- But best case performance is Ω (n)
- Stable? YES
- In-place? YES
- Question: So how/why is it worse than insertion sort?

Bubble sort properties

- Simple to implement so easy for small lists
- If there are no swaps during the pass, means the array is sorted -> can stop
  - Keep track by adding swapNeed=true statement
  - Useful in nearly ordered lists where there are very few passes

- Bubble sort uses (from exam answers)
- What kind of problems is bubble sort useful for?
  - “it is useful for getting points on the exam”
  - “it is useful if you want to sort a list really slowly”
  - “it is useful for when a teacher in school asks you to line up by height”

Selection Sort

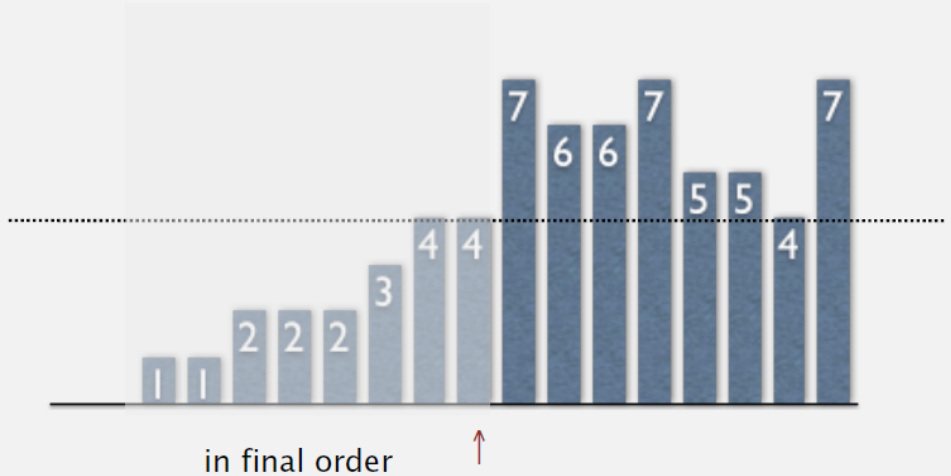
Selection sort

- In each iteration find the smallest remaining entry
- Swap current entry and the one you find

Algorithm. ↑ scans from left to right.

Invariants.

- Entries the left of ↑ (including ↑) fixed and in ascending order.
- No entry to right of ↑ is smaller than any entry to the left of ↑.



Selection sort

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```

- Identify index of minimum entry on right.

```
int min = i;
for (int j = i+1; j < N; j++)
   if (less(a[j], a[min]))
      min = j;
```

- Exchange into position.

```
exch(a, i, min);
```

Selection sort – int array in Java

```
void sort(int arr[])
 {
     int n = arr.length;

     // One by one move boundary of unsorted subarray
     for (int i = 0; i < n-1; i++)
     {
         // Find the minimum element in unsorted array
         int min_idx = i;
         for (int j = i+1; j < n; j++)
             if (arr[j] < arr[min_idx])
                 min_idx = j;

         // Swap the found minimum element with the first
         // element
         int temp = arr[min_idx];
         arr[min_idx] = arr[i];
         arr[i] = temp;
     }
 }
```

Selection Sort

62.0 , 83.0 , 18.0 , 53.0 , 7.0 , 17.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,
swapping 7.0 and 62.0

7.0 , 83.0 , 18.0 , 53.0 , 62.0 , 17.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,
swapping 17.0 and 83.0

7.0 , 17.0 , 18.0 , 53.0 , 62.0 , 83.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,
swapping 18.0 and 18.0

Selection sort properties

- Performance
  - Best case?
  - Worst case?
- O?
- Stable?
- In-place?

Selection sort properties

- Performance
- Worse case?
  - Two nested loops O(n^2)
- Best case performance also Ω (n^2)
  - Insensitive to input – finding the smallest one in one pass, implies nothing about where smallest one will be at the next, so still need a full pass
- Stable? NO
  - Eg 4 2 3 4 1 – find smallest which is 1, swap with 1st 4 = 1 2 3 4 4
- In-place? YES
- Example use
  - Minimal number of swaps/writes (n writes), if want to avoid writing to memory
  - Fast on small input sizes, 20-30

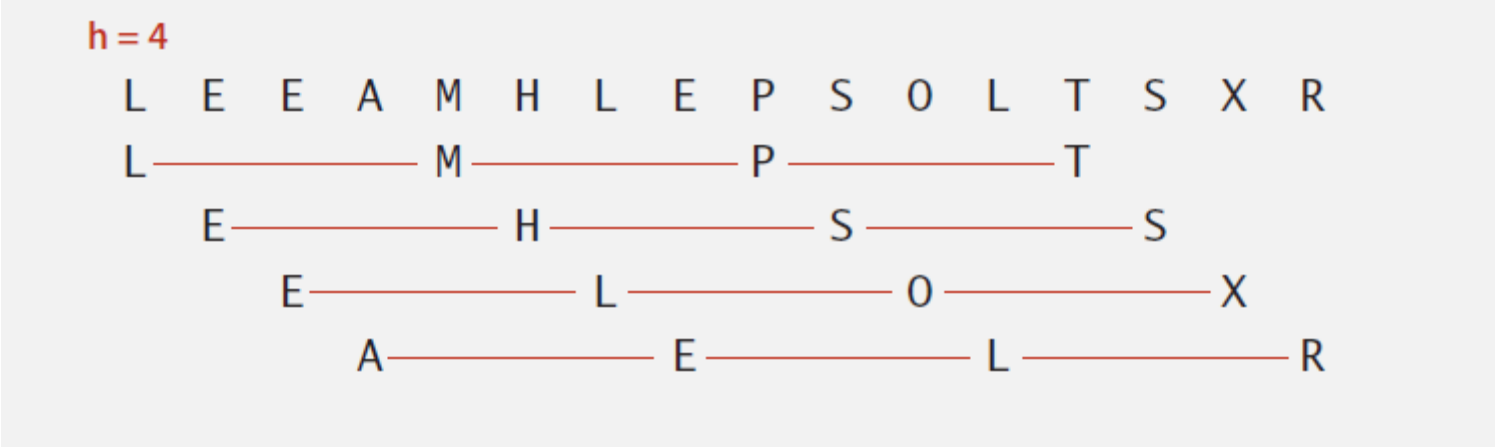
Shellsort

Shellsort

- Based on Insertion sort
  - Insertion slow for larger lists - it considers only adjacent items, so items move through array only 1 slot at a time
- Shellsort allows exchanges of entries that are far apart to produce partially sorted arrays, which are then sorted by insertion sort

Shellsort

› H-sorted array: take every h-th entry (starting anywhere) to get a sorted sequence



› h independent sorted sequences, interleaved together

Shellsort

› Use increment sequence of h, ending at h=1, to produce a sorted array

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

Shellsort

- How to select increment sequence?
- No provably best sequence has been found
- ½ (3^k-1)
  - Easy to compute and use
  - Performs nearly as well as more sophisticated ones

```
h=1;
while (h < n/3)
      h = 3h +1;
h-h/3;
//1, 4, 13, 40, 121, etc
```

Shellsort – java ints

```
public static void sort(int[] a)    {

   int N = a.length;

   int h = 1;
   while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, ...

   while (h >= 1)
   {  // h-sort the array.
      for (int i = h; i < N; i++)
      {
         for (int j = i; j >= h && (a[j] < a[j-h]); j -= h)
            exch(a, j, j-h);
      }

      h = h/3;
   }
}
```

3x+1 increment sequence

insertion sort

move to next increment

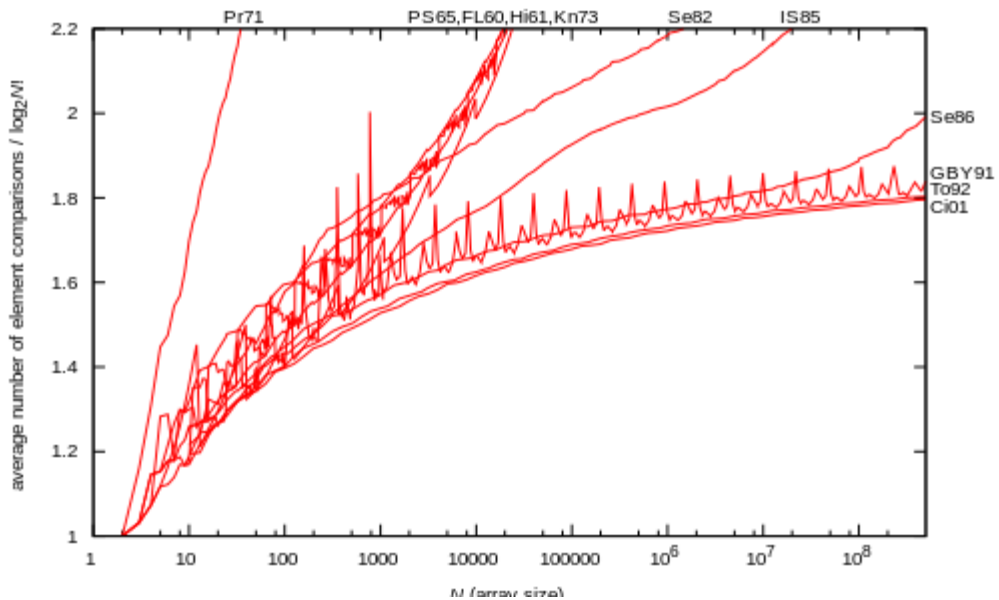
Swap method

```
private static void exch(int[] a, int i, int j){

   int swap = a[i];
   a[i] = a[j];
   a[j] = swap;
}
```

Shellsort properties

- https://en.wikipedia.org/wiki/Shellsort - per increment formula
- No precise model
- N^3/2, N^4/3, NlogN^2



- Stable? no
- In-place? yes

Shellsort

› **What is the best case input for Shellsort?**

a) A reverse sorted array because because all the sublists are sorted in linear time

b) A sorted array because each sublist is sorted in linear time

c) A random array

d) It doesn't matter, all inputs of a given size will cost the same

Sorting so far summary

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges
insertion	✔	✔	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small n or partially ordered
shell	✔		$n \log_3 n$	?	$c\, n^{3/2}$	tight code; subquadratic

Sorting algorithms review

- Consider the array a ={4.0, 6.0, 2.0, 3.0, 1.0, 7.0}
- Which algorithm produces the following trace while sorting the array?

1.0 , 6.0 , 2.0 , 3.0 , 4.0 , 7.0 ,
1.0 , 2.0 , 6.0 , 3.0 , 4.0 , 7.0 ,
1.0 , 2.0 , 3.0 , 6.0 , 4.0 , 7.0 ,
1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,
1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,
1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

Insertion sort (unoptimized)

4.0 , 6.0 , 2.0 , 3.0 , 1.0 , 7.0 ,

4.0 , 2.0 , 6.0 , 3.0 , 1.0 , 7.0 ,

2.0 , 4.0 , 6.0 , 3.0 , 1.0 , 7.0 ,

2.0 , 4.0 , 3.0 , 6.0 , 1.0 , 7.0 ,

2.0 , 3.0 , 4.0 , 6.0 , 1.0 , 7.0 ,

2.0 , 3.0 , 4.0 , 6.0 , 1.0 , 7.0 ,

2.0 , 3.0 , 4.0 , 1.0 , 6.0 , 7.0 ,

2.0 , 3.0 , 1.0 , 4.0 , 6.0 , 7.0 ,

2.0 , 1.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0

Selection sort

4.0 , 6.0 , 2.0 , 3.0 , 1.0 , 7.0 ,

1.0 , 6.0 , 2.0 , 3.0 , 4.0 , 7.0 ,

1.0 , 2.0 , 6.0 , 3.0 , 4.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 6.0 , 4.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

Shell sort (same as insertion sort optimized)

4.0 , 6.0 , 2.0 , 3.0 , 1.0 , 7.0 ,

1.0 , 6.0 , 2.0 , 3.0 , 4.0 , 7.0 ,

1.0 , 2.0 , 6.0 , 3.0 , 4.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 6.0 , 4.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

Bubble sort

4.0 , 6.0 , 2.0 , 3.0 , 1.0 , 7.0 ,

4.0 , 2.0 , 6.0 , 3.0 , 1.0 , 7.0 ,

4.0 , 2.0 , 3.0 , 6.0 , 1.0 , 7.0 ,

4.0 , 2.0 , 3.0 , 1.0 , 6.0 , 7.0 ,

4.0 , 2.0 , 3.0 , 1.0 , 6.0 , 7.0 ,

2.0 , 4.0 , 3.0 , 1.0 , 6.0 , 7.0 ,

2.0 , 3.0 , 4.0 , 1.0 , 6.0 , 7.0 ,

2.0 , 3.0 , 1.0 , 4.0 , 6.0 , 7.0 ,

2.0 , 3.0 , 1.0 , 4.0 , 6.0 , 7.0 ,

2.0 , 3.0 , 1.0 , 4.0 , 6.0 , 7.0 ,

2.0 , 3.0 , 1.0 , 4.0 , 6.0 , 7.0 ,

2.0 , 1.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

2.0 , 1.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

2.0 , 1.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

2.0 , 1.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

1.0 , 2.0 , 3.0 , 4.0 , 6.0 , 7.0 ,

,