Sorting

- Sorting
- The Sorting Problem
- Comparison of Sorting Algorithms upper limit O(n log n)
- Implementing Sorting
- Implementing **`isSorted()`**
- Sorts on Linux
- Describing Sorting Algorithms

❖ Sorting

Sorting involves arranging a collection of items in order

- based on some property of the items (e.g. key)
- using an ordering relation on that property

Why is sorting useful?

- speeds up subsequent searching
- arranges data in a human-useful way
  (e.g. list of students in a tute class, ordered by family-name or id)
- arranges data in a computationally-useful way
  (e.g. duplicate detection/removal, many DBMS operations)

❖ ... Sorting

Sorting occurs in many data contexts, e.g.

- arrays, linked-lists (internal, in-memory)
- files (external, on-disk)

Different contexts generally require different approaches

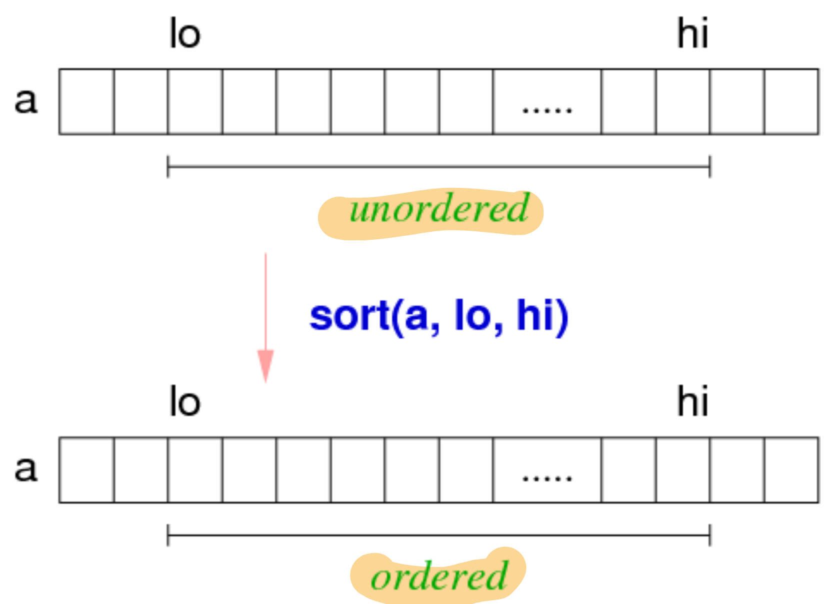
- and sorting has been well-studied over the last 50 years

Our view of the sorting problem:

- arrange an array of `Item`s in ascending order
- could sort whole array, or could sort a slice of the array

❖ The Sorting Problem

Arrange items in array slice `a[lo..hi]` into sorted order:



For `Item a[N]`, frequently `(lo == 0)`, `(hi == N-1)`

❖ ... The Sorting Problem

More formally ...

Precondition: 先决条件

- **lo**,**hi** are valid indexes, i.e. **0** ≤ **lo** < **hi** ≤ **N-1**
- **a[lo..hi]** contains defined values of type **Item**

Postcondition: 后置条件.

- **a[lo..hi]** contains same set (bag) of values
- foreach **i** in **lo..hi-1**, **a[i]** ≤ **a[i+1]**

❖ ... The Sorting Problem

We sort arrays of **`Item`**s, which could be

- simple values, e.g. **`int`**, **`char`**, **`float`**
- structured values, e.g. **`struct`**

Each **`Item`** contains a **`key`**, which could be

- a simple value, or a collection of values

The order of **`key`** values determines the order of the sort.

Duplicate **`key`** values are not precluded. 不排除

In our discussions, we often use the **`key`** value as if it is the whole **`Item`**

❖ ... The Sorting Problem

Properties of sorting algorithms: stable, adaptive 适合的.

Stable sort:

重复的.

- let x = `a[i]`, y = `a[j]`, `key(x) == key(y)`
- "precedes" = occurs earlier in the array (smaller index)
- if x precedes y in **a**, then x precedes y in sorted **a**

x始终在y前

Adaptive:

- behaviour/performance of algorithm affected by data values
- i.e. best/average/worst case performance differs

unsort
8
x → 2
90
y → 2
7

sort
2 ← x
2 ← y
7
8
90

4, 7, 8, 12 sorted. (will quickly)

8, 1, 7, 25 random. (sort slowly)

❖ ... The Sorting Problem

In analysing sorting algorithms:

- N = number of items = `hi-lo+1`
- C = number of comparisons between items
- S = number of times items are swapped

Aim to minimise C and S

Cases to consider for initial order of items:

- random order: `Items` in `a[lo..hi]` have no ordering
- sorted order: `a[lo]` ≤ `a[lo+1]` ≤ ... ≤ `a[hi]`
- revserse order: `a[lo]` ≥ `a[lo+1]` ≥ ... ≥ `a[hi]`

❖ Comparison of Sorting Algorithms

A variety of sorting algorithms exist

- most are in-memory algorithms, some also work with files
- two major classes: $O(n^2)$, $O(n \log n)$
- $O(n^2)$ are acceptable if n is small (hundreds)

Ways to compare algorithms:

- implement and monitor performance
- graphic visualisations
- or even folk dancing

❖ Implementing Sorting

Concrete framework:

```
// we deal with generic Items
typedef SomeType Item;

// abstractions to hide details of Items
#define key(A) (A)
#define less(A,B) (key(A) < key(B))
#define swap(A,B) {Item t; t = A; A = B; B = t;}

// Sorts a slice of an array of Items, a[lo..hi]
void sort(Item a[], int lo, int hi);

// Check for sortedness (to validate functions)
int isSorted(Item a[], int lo, int hi);
```

❖ Implementing `isSorted()`

Implementation of the **`isSorted()`** check.

```
bool isSorted(Item a[], int lo, int hi)
{
   for (int i = lo; i < hi; i++) {
      if (!less(a[i],a[i+1])) return false;
   }
   return true;
}
```

Checks pairs (**`a[lo]`**,**`a[lo+1]`**), ... (**`a[hi-1]`**,**`a[hi]`**)

Check whole array **`Item a[N]`** via **`isSorted(a, 0, N-1)`**

a[0] — a[N-1]

❖ Sorts on Linux

The `sort` command

- sorts a file of text, understands fields in line
- can sort alphabetically, numerically, reverse, random

The `qsort()` function

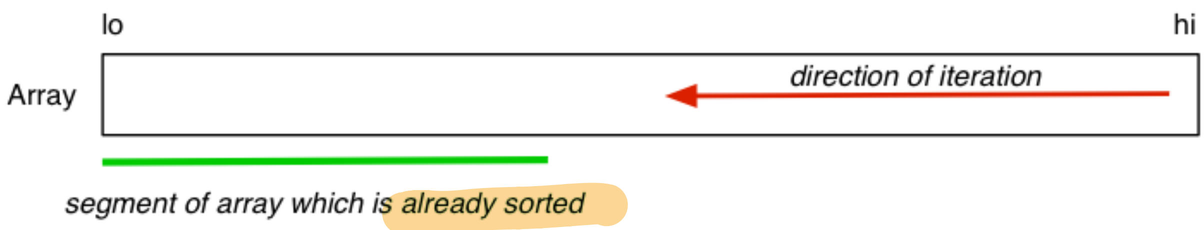
size of every element

- `qsort(void *a, int n, int size, int (*cmp)())`
- sorts any kind of array (`n` objects, each of `size` bytes)
- requires the user to supply a comparison function (e.g. `strcmp()`)
- sorts list of items using the order given by `cmp()`

Note: the comparison function is passed as a parameter; discussed elsewhere.

❖ Describing Sorting Algorithms

To describe sorting, we use diagrams like:



In these algorithms ...

- some part(s) of the array is already sorted
- each iteration makes more of the array sorted

See also animations by David R. Martin, Boston College, based on Sedgewick's idea

O(n^2) Sorts

❖ O(n²) Sorting Algorithms

One class of sorting methods has complexity $O(n^2)$

- selection sort ... simple, non-adaptive sort
- bubble sort ... simple, adaptive sort track changes of array
- insertion sort ... simple, adaptive sort
- shellsort ... improved version of insertion sort

There are sorting methods with better complexity $O(n \log n)$

But for small arrays, the above methods are adequate

❖ Selection Sort

Simple, non-adaptive method:

- find the smallest element, put it into first array slot
- find second smallest element, put it into second array slot
- repeat until all elements are in correct position

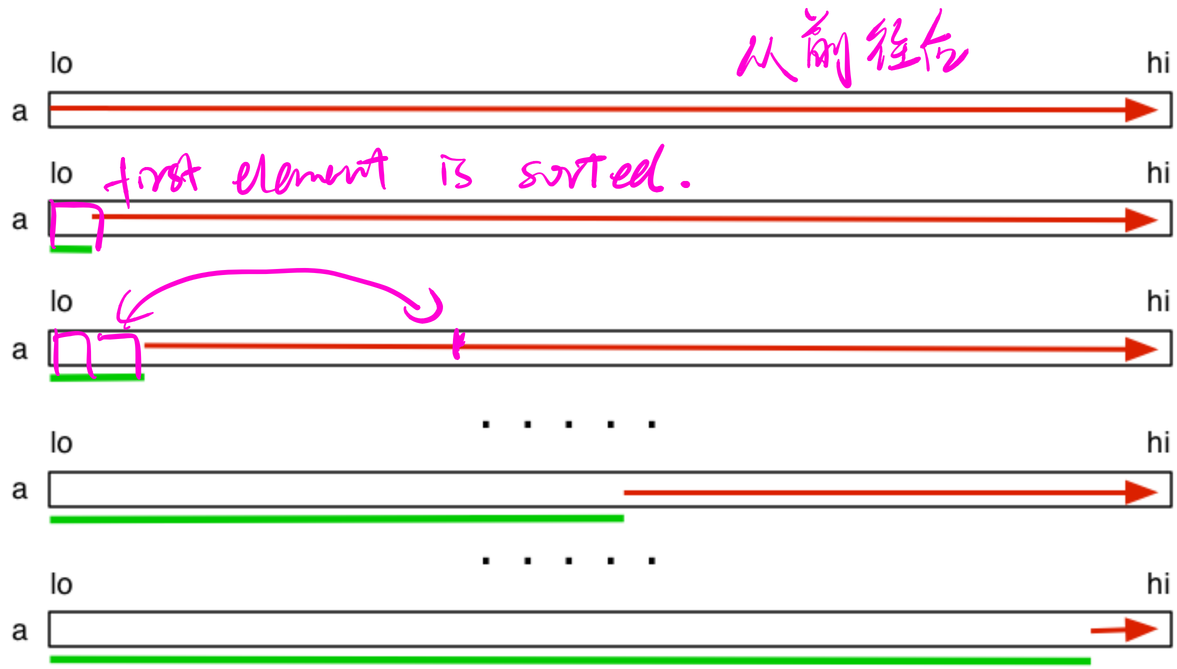
"Put in x^{th} array slot" is accomplished by:

- swapping value in x^{th} position with x^{th} smallest value

Each iteration improves "sortedness" by one element

❖ ... Selection Sort

State of array after each iteration:



❖ ... Selection Sort

C function for Selection sort:

```
                                       array
void selectionSort(int a[], int lo, int hi)
{
   int i, j, min;
   for (i = lo; i < hi-1; i++) {
      min = i;                 i之前的已sort       i=first.
      for (j = i+1; j <= hi; j++) {
         if (less(a[j],a[min])) min = j;
      }
      swap(a[i], a[min]);
   }
}
```

❖ ... Selection Sort

Cost analysis (where n = **`hi-lo+1`**):

- on first pass, *n-1* comparisons, 1 swap
- on second pass, *n-2* comparisons, 1 swap
- ... on last pass, *1* comparison, 1 swap
- compare C = $(n-1)+(n-2)+...+1 = n*(n-1)/2 = (n^2-n)/2 \Rightarrow O(n^2)$
- S = *n-1* swap.

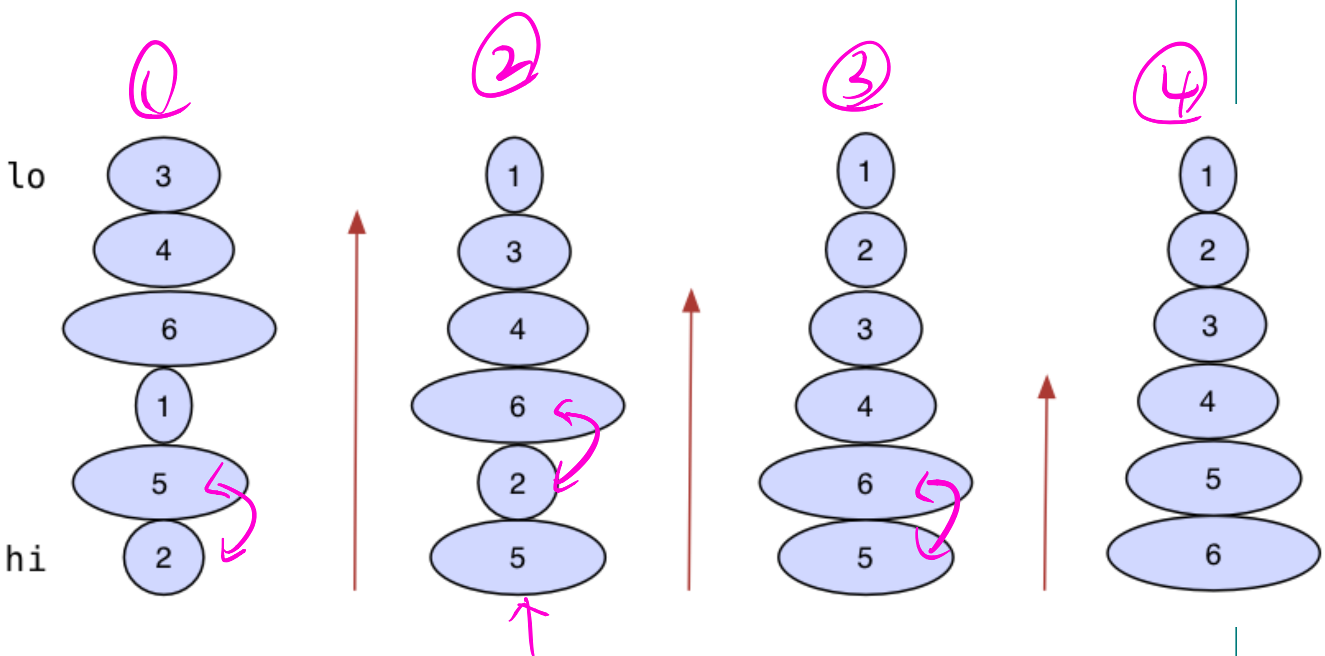
Cost is same, regardless of sortedness of original array.

❖ Bubble Sort

Simple adaptive method:

- make multiple passes from *N* to *i* (*i=0..N-1*)
- on each pass, swap any out-of-order adjacent pairs 相邻的.
- elements move until they meet a smaller element
- eventually smallest element moves to i^{th} position
- repeat until all elements have moved to appropriate position
- stop if there are no swaps during one pass (already sorted)

... Bubble Sort



1

3	3	3	1
4	4	1	3
6	1	4	4
1	6	6	6
2	2	2	2
5	5	5	5

$O(n^2)$ ✓

$(n-1)+(n-2)+(n-3)\cdots 1$

2

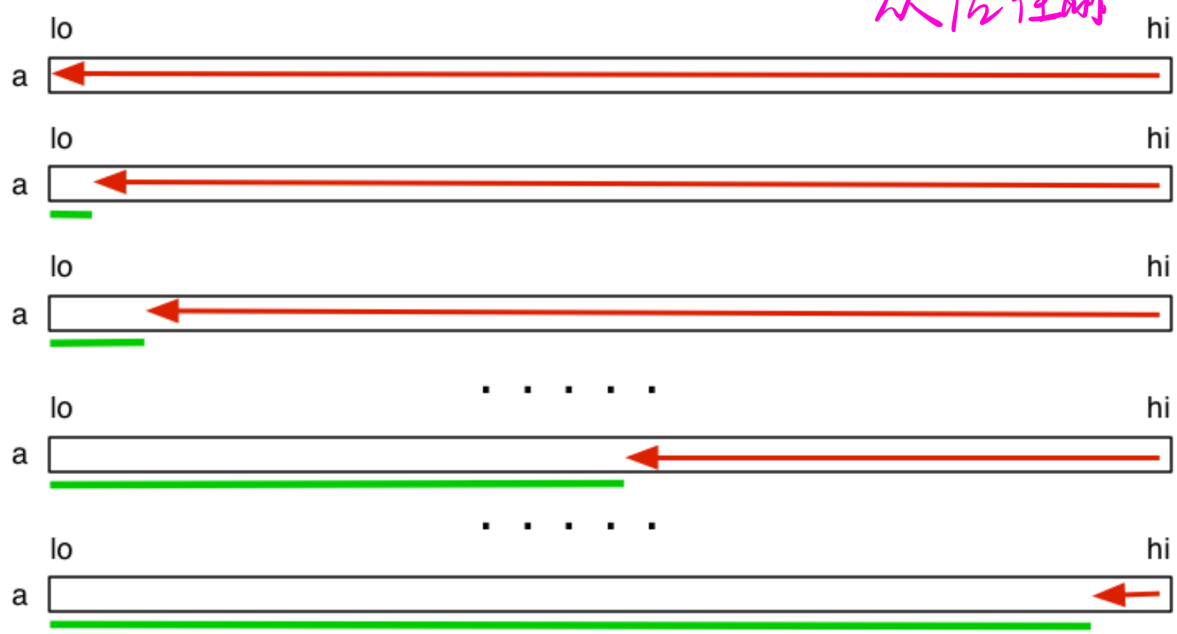
1	1	1
3	3	2
4	2	3
2	4	4
6	6	6
5	5	5

$$\frac{(1+(n-1)) \times n}{2} = \frac{n^2}{2}$$

❖ ... Bubble Sort

State of array after each iteration:

从后往前



❖ ... Bubble Sort

Bubble sort example (from Sedgewick):

```
S O R T E X A M P L E
A S O R T E X E M P L
A E S O R T E X L M P
A E E S O R T L X M P
A E E L S O R T M X P
A E E L M S O R T P X
A E E L M O S P R T X
A E E L M O P S R T X
A E E L M O P R S T X
... no swaps ⇒ done ...
A E E L M O P R S T X
```

... Bubble Sort

C function for Bubble Sort:

```
void bubbleSort(int a[], int lo, int hi)
{
   int i, j, nswaps;
   for (i = lo; i < hi; i++) {
      nswaps = 0;
      for (j = hi; j > i; j--) {
         if (less(a[j], a[j-1])) {
            swap(a[j], a[j-1]);
            nswaps++;
         }
      }
      if (nswaps == 0) break;
   }
}
```

已经前移.

already sorted.

[i]= 45 j=5.

45 23 67 5

5 45 23 67 [i]=45 j=67

5 23 45 67

❖ ... Bubble Sort

Cost analysis (where *n* = **`hi-lo+1`**):

- cost for i^{th} iteration:
  - *n-i* comparisons, ?? swaps
  - *S* depends on "sortedness", best=0, worst=*n-i*
- how many iterations? depends on data orderedness
  - best case: 1 iteration, worst case: *n-1* iterations
- $Cost_{best}$ = *n* (data already sorted)
- $Cost_{worst}$ = *n-1 + ... + 1* (reverse sorted)
- Complexity is thus $O(n^2)$

❖ Insertion Sort

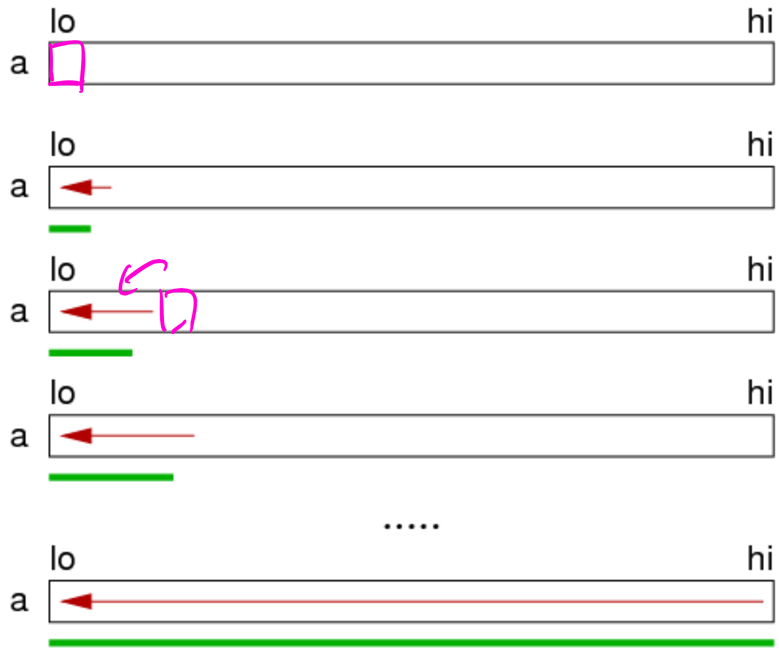
Simple adaptive method:

- take first element and treat as sorted array (length 1)
- take next element and insert into sorted part of array so that order is preserved
- above increases length of sorted part by one
- repeat until whole array is sorted

❖ ... Insertion Sort

65, 2, 78, 55

insert in an empty array



❖ ... Insertion Sort

Insertion sort example (from Sedgewick):

```
S O R T E X A M P L E
S O R T E X A M P L E
O S R T E X A M P L E
O R S T E X A M P L E
O R S T E X A M P L E
E O R S T X A M P L E
E O R S T X A M P L E
A E O R S T X M P L E
A E M O R S T X P L E
A E M O P R S T X L E
A E L M O P R S T X E
A E E L M O P R S T X
```

❖ ... Insertion Sort

C function for insertion sort:

```
void insertionSort(int a[], int lo, int hi)
{
   int i, j, val;
   for (i = lo+1; i <= hi; i++) {
      val = a[i];
      for (j = i; j > lo; j--) {
         if (!less(val,a[j-1])) break;
         a[j] = a[j-1]; 往后移一个
                        整体
      }
      a[j] = val;
   }
}
```

lo　　　　　　　　　hi
45 , 75 , 2 , 6 , 50
[0] [1] [2] [3] [4]

$i = 1$

$val = a[1] = 75$

$j = i = 1$ 　if 　$75 < a[j-1] = a[0] = 45$ ✗

$a[j] = a[j-1] = a[1] = 45$

$a[j] = 75$

❖ ... Insertion Sort

Cost analysis (where n = **`hi-lo+1`**):

- cost for inserting element into sorted list of length i
  - C=??, depends on "sortedness", best=1, worst=i
  - S=??, don't swap, just shift, but do $C-1$ shifts
- always have n iterations
- $\text{Cost}_{best} = 1 + 1 + ... + 1$ (already sorted)
- $\text{Cost}_{worst} = 1 + 2 + ... + n = n*(n+1)/2$ (reverse sorted)
- Complexity is thus $O(n^2)$

❖ ShellSort: Improving Insertion Sort

Insertion sort:

- based on exchanges that only involve adjacent items
- already improved above by using moves rather than swaps
- "long distance" moves may be more efficient

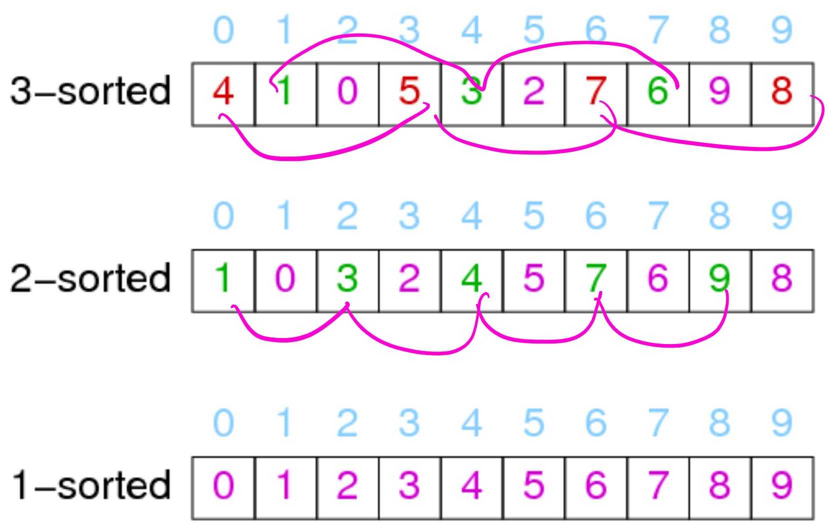
Shellsort: basic idea

- array is *h*-sorted if taking every *h*'th element yields a sorted array
- an *h*-sorted array is made up of *n/h* interleaved sorted arrays
- Shellsort: *h*-sort array for progressively smaller *h*, ending with 1-sorted

❖ ... ShellSort: Improving Insertion Sort

Example *h*-sorted arrays:

Sorted every 3 elements



❖ ... ShellSort: Improving Insertion Sort

```
void shellSort(int a[], int lo, int hi)
{
   int hvals[8] = {701, 301, 132, 57, 23, 10, 4, 1};
   int g, h, start, i, j, val;
   for (g = 0; g < 8; g++) {
      h = hvals[g];
      start = lo + h;
      for (i = start+1; i <= hi; i++) {
         val = a[i];
         for (j = i; j >= start; j -= h) {
            if (!less(val,a[j-h]) break;
            a[j] = a[j-h];
         }
         a[j] = val;
      }
   }
}
```

❖ ... ShellSort: Improving Insertion Sort

Effective sequences of *h* values have been determined empirically.

E.g. $h_{i+i} = 3h_i+1$ ... 1093, 364, 121, 40, 13, 4, 1

Efficiency of Shellsort:

- depends on the sequence of *h* values
- suprisingly, Shellsort has not yet been fully analysed
- above sequence has been shown to be $O(n^{3/2})$
- others have found sequences which are $O(n^{4/3})$

❖ Summary of Elementary Sorts

Comparison of sorting algorithms (animated comparison)

	#compares			#swaps			#moves		
	min	avg	max	min	avg	max	min	avg	max
Selection sort	n^2	n^2	n^2	n	n	n	.	.	.
Bubble sort	n	n^2	n^2	0	n^2	n^2	.	.	.
Insertion sort	n	n^2	n^2	.	.	.	n	n^2	n^2
Shell sort	n	$n^{4/3}$	$n^{4/3}$	.	.	.	1	$n^{4/3}$	$n^{4/3}$

Which is best?

- depends on cost of compare vs swap vs move for `Items`
- depends on likelihood of average vs worst case

❖ Sorting Linked Lists

Selection sort on linked lists

- L = original list, S = sorted list (initially empty)
- find largest value V in L; unlink it 大→小
- link V node at front of S

Bubble sort on linked lists

- traverse list: if current > next, swap node values
- repeat until no swaps required in one traversal

Insert

~~Selection~~ sort on linked lists

- L = original list, S = sorted list (initially empty)
- scan list L from start to finish
- insert each item into S in order