O(n^2) Sorts

- O(n^2) Sorting Algorithms
- Selection Sort
- Bubble Sort
- Insertion Sort
- ShellSort: Improving Insertion Sort
- Summary of Elementary Sorts
- Sorting Linked Lists

❖ O(n^2) Sorting Algorithms

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

- selection sort ... simple, non-adaptive sort
- bubble sort ... simple, adaptive sort
- 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

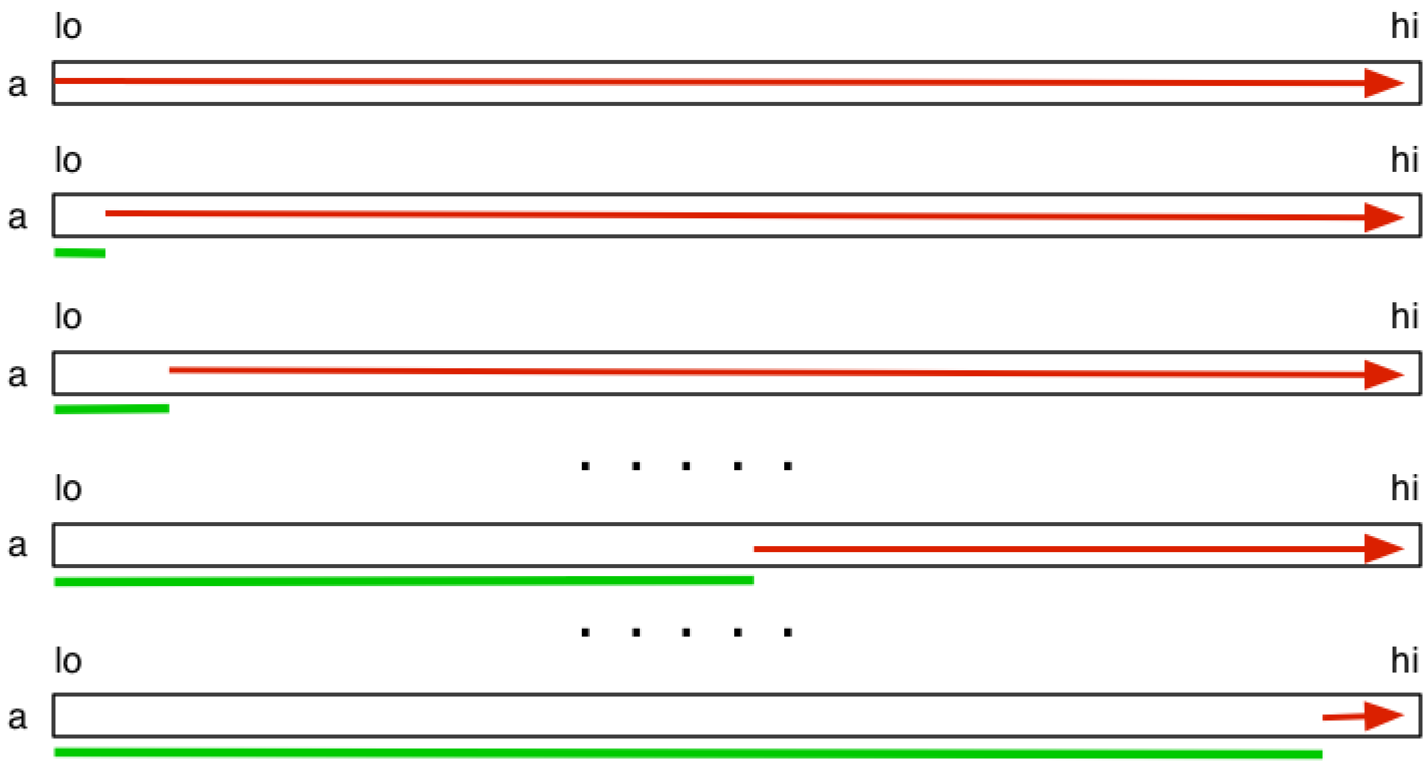
"Put in x^{th} array slot" is accomplished by:

- 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:

```
void selectionSort(int a[], int lo, int hi)
{
   int i, j, min;
   for (i = lo; i < hi-1; i++) {
      min = i;
      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
- C = $(n-1)+(n-2)+...+1 = n*(n-1)/2 = (n^2-n)/2 \Rightarrow O(n^2)$
- S = *n-1*

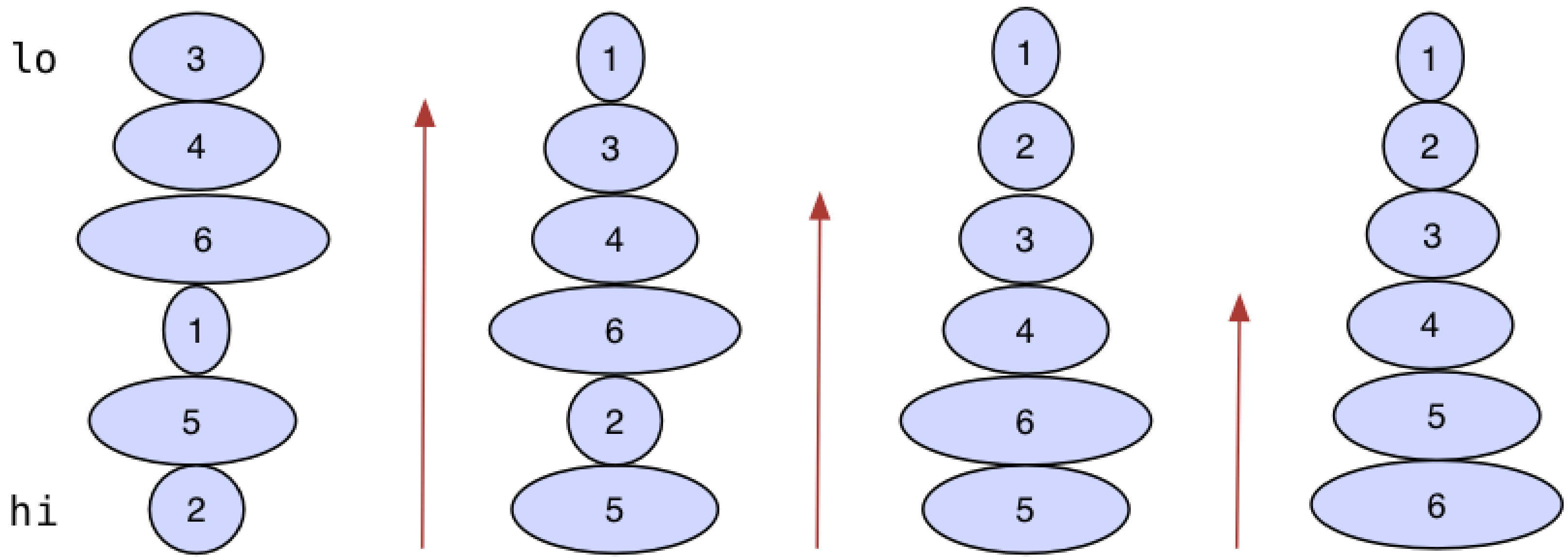
Cost is same, regardless of sortedness of original array.

❖ Bubble Sort

Simple adaptive method:

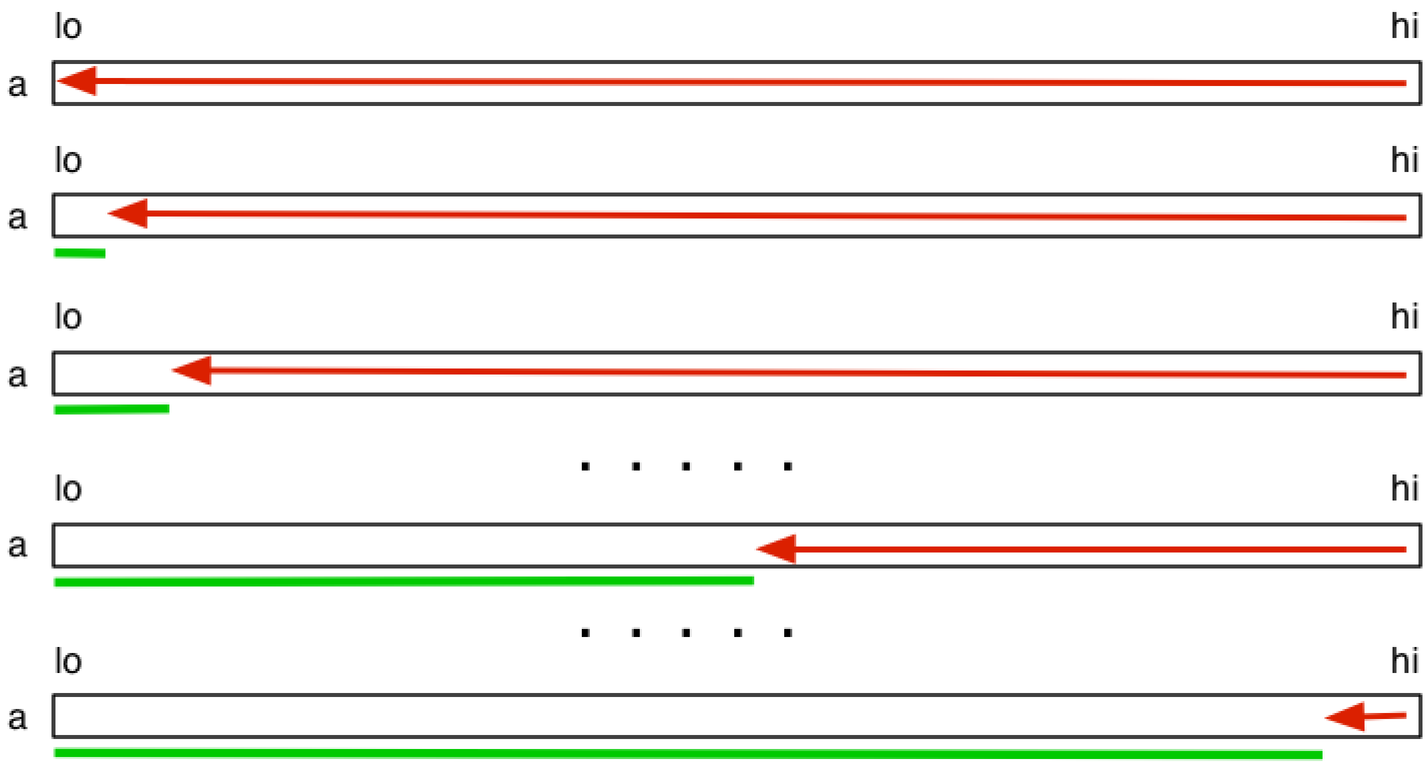
- make multiple passes from *N* to *i* (*i=0..N-1*)
- on each pass, swap any out-of-order adjacent pairs
- elements move until they meet a smaller element
- eventually smallest element moves to i^{th} position
- repeat until all elements have moved to appropriate position
- stop if there are no swaps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

❖ ... Bubble Sort



❖ ... Bubble Sort

State of array after each iteration:



❖ ... 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;
   }
}
```

❖ ... Bubble Sort

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

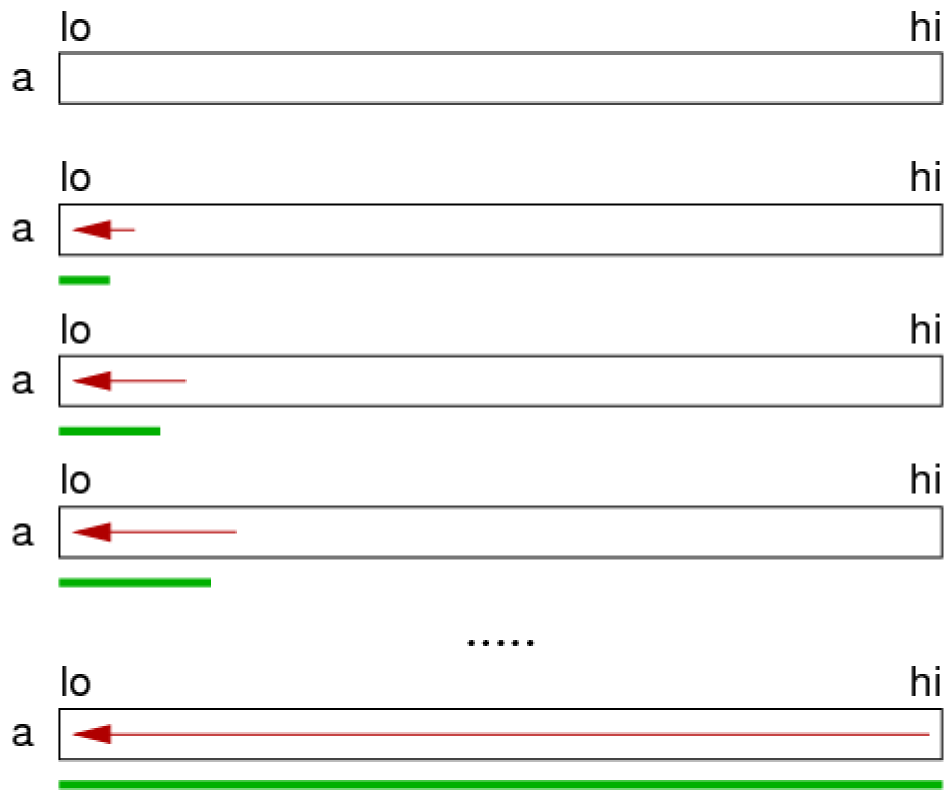
- cost for i^{th} iteration:
  - n-i comparisons, ?? swaps
  - S depends on "sortedness", best=0, worst=n-i
- how many iterations? depends on data orderedness
  - best case: 1 iteration, worst case: n-1 iterations
- $Cost_{best} = n$ (data already sorted)
- $Cost_{worst} = n\text{-}1 + ... + 1$ (reverse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



❖ ... 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];
      }
      a[j] = val;
   }
}
```

❖ ... 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
- $\text{Cost}_{\text{best}} = 1 + 1 + ... + 1$ (already sorted)
- $\text{Cost}_{\text{worst}} = 1 + 2 + ... + n = n*(n+1)/2$ (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:

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

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

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

❖ ... 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 `Item`s
- 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
- 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

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