Elementary Sorting Algorithms

COMP1927 17x1

Sedgewick Chapter 6

The Problem of Sorting

- **Why sort?**
  - makes searching easier
  - useful for reading reports/lists/tables

- **A variety of algorithms to solve this problem…**
  - Which is better?
  - How can we compare them?
  - How can we classify them?

Sorting on Linux

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

- Use the **sort** command to sort data1:

```
5059413 Daisy 3762  15
3461045 Yan   3648  42
3474881 Sinan 8543 16
5061020 Yu     3970  3
```

- To learn more about sort, try sorting data 2 and data 3

Sorting

- For the time being …
  - sorting an array of **ints** (e.g. **int a[N]**)
  - order determined by natural order on **ints**
  - implemented as **sort(int a[], int lo, int hi)**

- For the rest of the course and real world apps…
  - Sorting an array of **Items** (e.g., **Item a[N]**
  - each **Item** has a **key** value
  - using an ordering relation on that **key** – order on keys determines order on the **Items**
  - implemented as **sort (Item a[], int lo, int hi)**

Comparing Sorting Algorithms

- Core operations for sorting: compare, swap and move

- In analyzing sorting algorithms:
  - Worst case time complexity
  - N = number of items = **hi –lo + 1**
  - C = number of comparisons between items
  - S = number of times items are swapped
  - M = number of times items are moved

    Aim to minimise C, S and M (but often M = 0)
- Cases to consider for initial ordering of items. What is the worst case for the given algorithm?
  - random order? sorted order? reverse sorted order?
  - sometimes specific non-random, non-ordered permutations?

Properties of Sorting Algorithms

- Efficiency: $O(n^2)$, $O(n \log n)$, $O(n)$
- Stability:
  - Stable sorting methods preserve the relative order of items with duplicate keys i.e. two ojects with equal keys appear in the same orde in the **sorted** output as they appear in the input **unsorted** array
  - Non-stable sorting methods may change the relative order of items with duplicate keys

Adams	2001
Black	2002
Jackson	2002
Brown	2004
Jones	2004
White	2003
Wilson	2003
Smith	2001

Adams	2001
Smith	2001
Black	2002
Jackson	2002
White	2003
Wilson	2003
Brown	2004
Jones	2004

Properties of Sorting Algorithms

- Adaptability:

  Non-Adaptive sort (aka oblivious sort) :

  - uses the same sequence of operations, independent of input data

  Adaptive sort :

  - behaviour changes with “orderedness" of input
  - different performance for ascending/descending/random input
    - ★ can take advantage of existing order already present in the sequence

```
#define compexch(A, B)
                if (less(B, A)) exch(A, B)
void sort(Item a[], int l, int r)
{
        int i, j;
        for (i = l+1; i <= r; i++)
                for (j = i; j > l; j--)
                        compexch(a[j-1], a[j]);
}
```

```
void insertion(Item a[], int l, int r)
  { int i;
    for (i = r; i > l;  i--) compexch(a[i-1], a[i]);
    for (i = l+2; i <= r; i++)
      { int j = i; Item v = a[i];
        while (less(v, a[j-1]))
          { a[j] = a[j-1]; j--; }
        a[j] = v;
      }
  }
```

Comparing Sorting Algorithms

- In-place algorithm implementation
  - sorts the data within the original structure
  - uses only a small constant amount of extra storage space
    - eg swapping elements within an array
    - moving pointers within a linked list
    - All sorting algorithms CAN be implemented in-place, but some algorithms are naturally in-place and others are not

Describing Sorting Algorithms

- To describe simple sorting, we use diagrams like:

In these algorithms

- a segment of the array is already sorted
- each iteration makes more of the array sorted

Sorting

- Three simple sorting algorithms:
  - Bubble sort
    - Bubble sort with Early Exit
  - Selection sort
  - Insertion sort
- One more complex sorting algorithm:
  - Shell sort

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 the array after each iteration:

Selection Sort on an Array

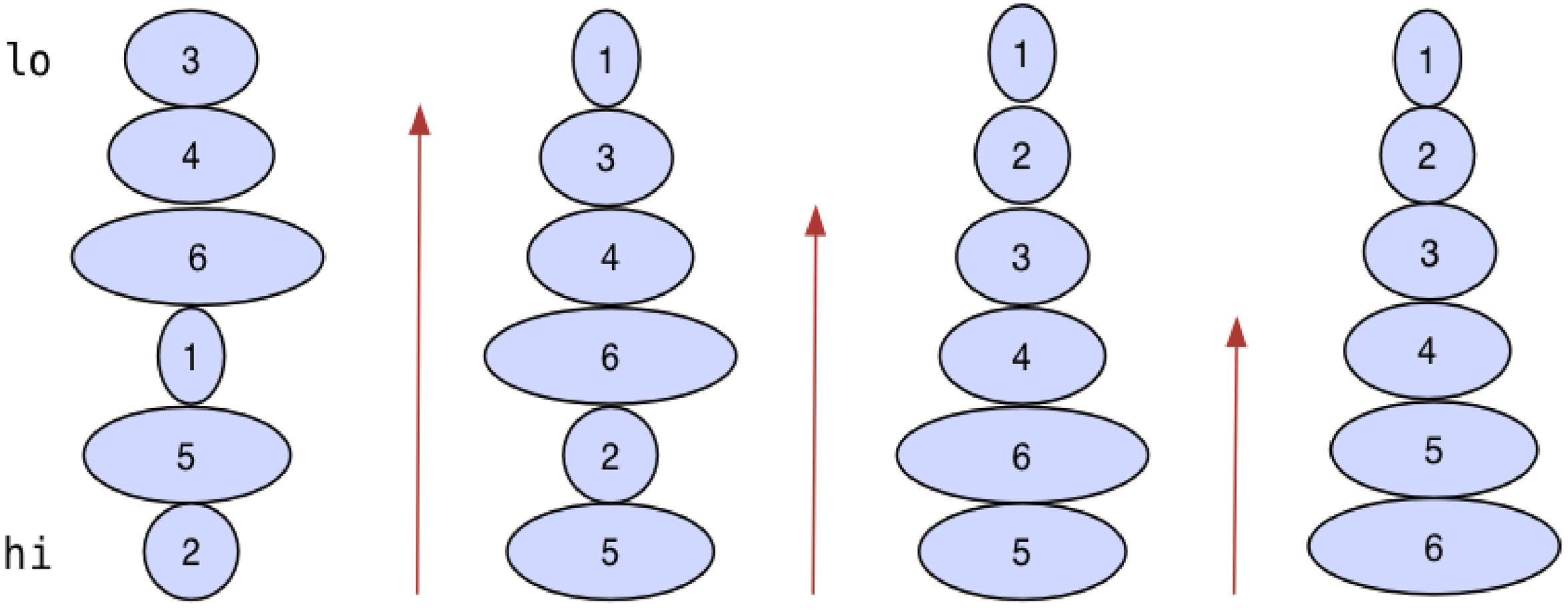
```
//Does not use a second array. Sorts within the original
  array
          void selectionSort(int a[], int lo, int hi) {
    int i, j, min;
    for (i = lo; i < hi; i++) {
        min = i; // current minimum is first unsorted
    element
        // find index of minimum element
        for (j = i + 1; j <= hi; j++) {
            if (less(a[j],a[min])) {
                min = j; }
        }
        // swap a[i] with a[min]
        swap(a, i, min);
    }
}
```

Selection sort – Cost Analysis

- How many steps does it take to sort a collection of $n=(hi-lo+1)$ elements?
  - on first pass, *n-1* comparisons, *1* swap
  - on second pass, *n-2* comparisons, 1 swap
  - …. on lass pass, *1* comparisons, 1 swap
  - **C** = $(n-1)+(n-2)+\ldots+1 = n*(n-1)/2 = (n^2-n)/2 \Rightarrow O(n^2)$
  - **S =** *n-1*
  - Selection is in $O(N^2)$
- Implementation is not stable
- Implementation is in-place,
- Non-adaptive sort, as cost is same regardless of "orderedness" of original array

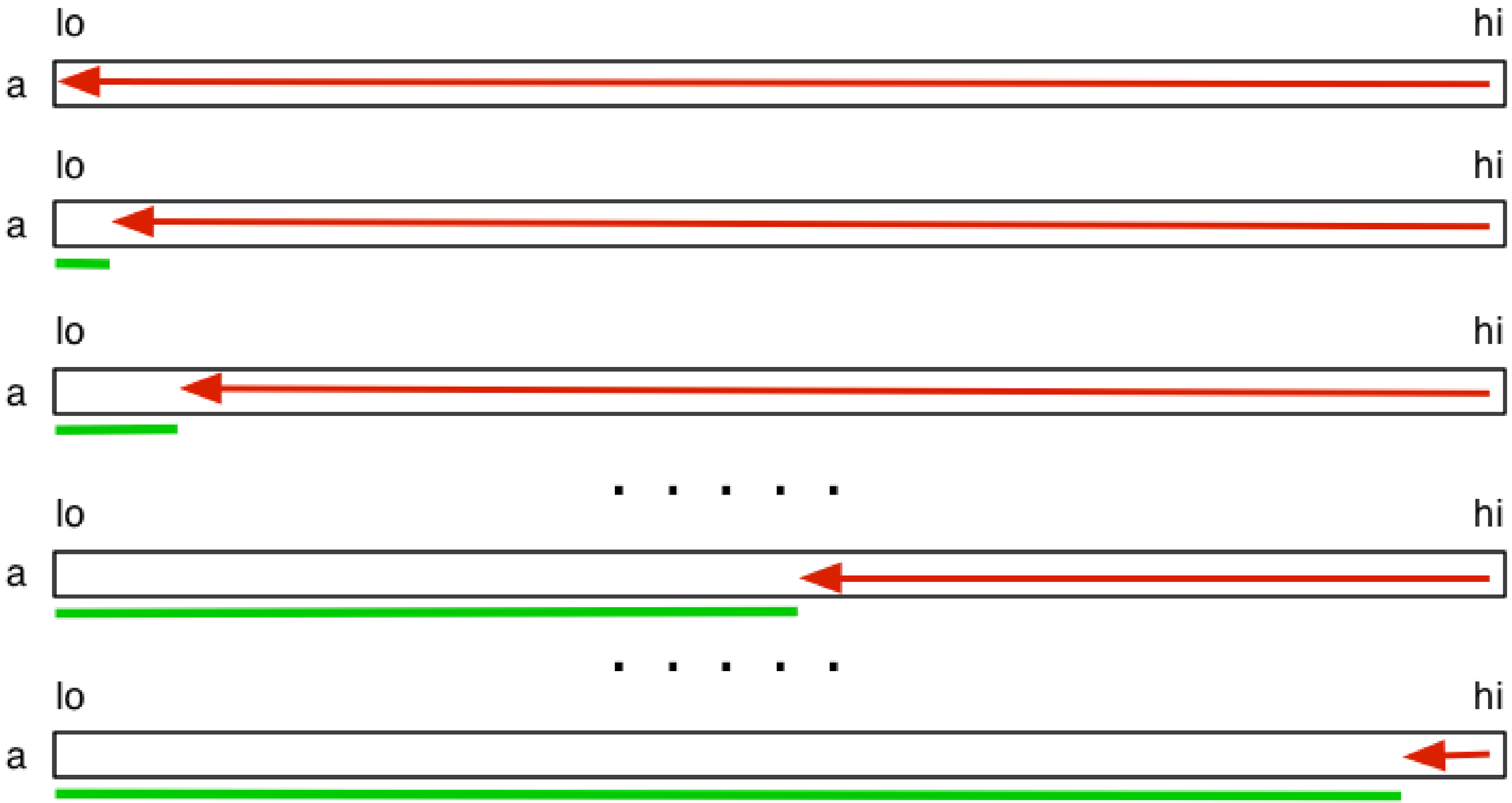
Bubblesort

- A simple adaptive sort
- `Bubbles‘ rise to the top until they hit a bigger bubble, which then starts to rise



Bubblesort (cont)

State of the array after each pass (iteration):



Bubble sort

```
// 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,j-1);
                nswaps++;
            }
        }
        if (nswaps == 0) break;
    }
}
```

Bubble sort Cost Analysis

Cost Analysis for n (*n= hi-lo +1*) elements:

- Cost for i th iteration:
  - *n-i* comparsions, ?? swaps
  - S depends on "sortedness", best=0, worst=*n-i* (where we always swap)
- How many iterations? Depends on data orderedness
  - best-case: *1* iteration, worst-case: *n-1* iterations
- Efficiency = O(N^2)
  - $Cost_{best} = n$ (data already sorted, 1 iteration with n-1 comparisons, 0 swaps)
  - $Cost_{worst} = (n-1) + (n-2) + (n-3) + \ldots + 1 =$
    $\sum_{i=1}^{n-1}(n-i) = (n(n-1))/2$ (reverse sorted

Bubble Sort: Analysis

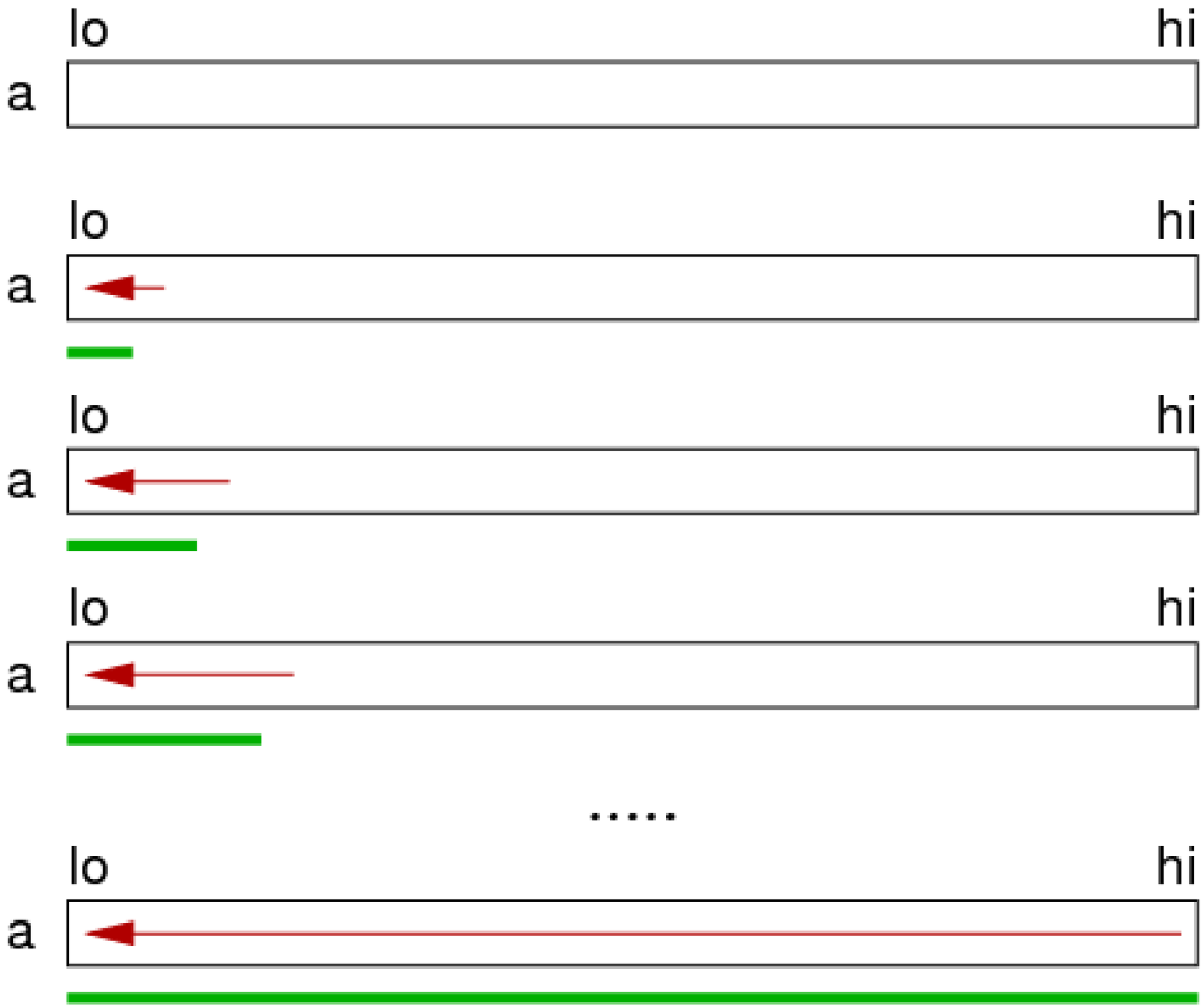
- Adaptive sort
  - We are able to improve sort by stopping when the elements are sorted
  - If we complete a whole pass with any swaps, we know it must be in order
  - Will not help cases that are in reverse order
- Stable, in-place sort

Insertion Sort

A simple adaptive sort method:

1. Take first element and insert it into first position (trivially sorted, because it has only one element)
2. Take next element, and insert it such that order is preserved
3. Continue, until all elements are in the correct positions

Insertion Sort



Simple Insertion Sort

```
void insertionSort(int a[], int lo, int hi)
{
    int i, j, min, val;
    min = lo;
    for (i = lo+1; i <= hi; i++)
        if (less(a[i],a[min])) min = i;
    swap(a, lo, min);
    for (i = lo+2; i <= hi; i++) {
        val = a[i];
        while (less(val, a[j-1]) {
                    move(a, j, j-1); j-- }
        a[j] = val;
    }
}
```

Insertion Sort : Complexity Analysis

Complexity analysis ...

- cost for inserting element into sorted list of length *i*
  - *C*=??, depends on "sortedness", best=1, worst=*i*
  - S=0, don't swap
  - *M*=??, depends where val fits, best=1, worst=*i*
- always have *N* iterations
- $Cost_{best} = 1 + 1 + ... + 1$ (already sorted)
- $Cost_{worst} = 1 + 2 + ... + N = N*(N+1)/2$ (reverse sorted)
- Complexity is thus $O(n^2)$

Shell Sort

- Short comings of insertion sort/bubble sort
  - Exchanges only involve adjacent elements
  - Long distance exchanges can be more efficient
- Shell sort basic idea:
  - Sequence is h-sorted
    - taking every h-th element gives a sorted sequence
    - h-sort the sequence with smaller values of h until h=1
- What sequence of h values should we use?
  - Knuth proposed 1 4 13 40 121 364…
    - It is easy to compute and results in an efficient sort
  - What is the best sequence ? No-one knows

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

Shell Sort (with h-values 1,4,13,40...)

```
void shellSort(int items[], int n) {
  int i, j, h;
  //the starting size of h is found.
  for (h = 1; h <= (n - 1)/9; h = (3 * h) + 1);
  for (; h > 0; h /= 3) {
    //when h = 1 this is insertion sort ☺
    for (i = h; i < n; i++) {
       int key = items[i];
       for(j=i; j>=h && key<items[j - h]; j -=h){
          items[j] = items[j - h];
       }
       items[j] = key;
    }
  }
}
```

Shell Sort: Work Complexity

- Exact time complexity properties depend on the h-sequence
  - So far no-one has been able to analyse it precisely
  - For the h-values we have used Knuth suggests around $O(n^{3/2})$
- It is adaptive as it does less work when items are in order – based on insertion sort.
- It is not stable,
- In-place

Linked List Implementations

- Bubble Sort :
  - Traverse list: if current element bigger than next, swap places, repeat.
- Selection Sort:
  - Straight forward: delete selected element from list and insert as first element into the sorted list, easy to make stable
- Insertion Sort:
  - Delete first element from list and insert it into new list. Make sure that insertion preserves the order of the new list
- Shell Sort:
  - Can be done …but better suited to arrays