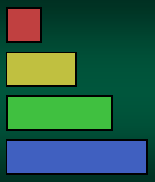
Elementary Sorts

Section 2.1

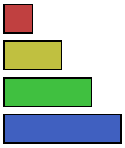


Sorting

Bringing Order of "Chaos"

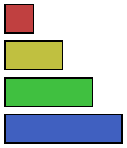
Sorting

- It is useful (and efficient) to *sort* a list of data – to put it in specific order
- There are multiple sorting algorithms which get complex as they become more efficient



Sorting

- Examples:
  - sorting scores by highest to lowest
  - sorting filenames in alphabetical order
  - sorting students by their student-id

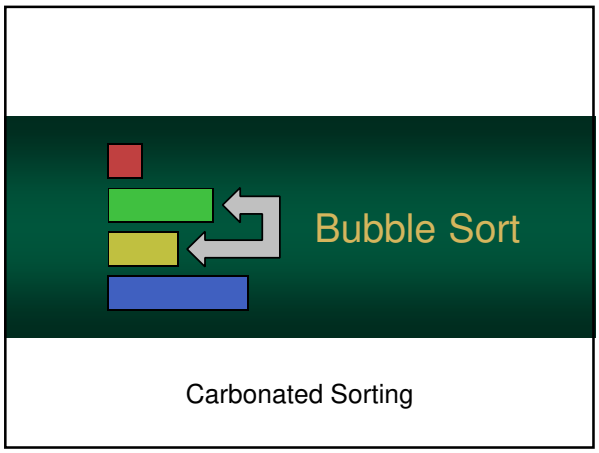


Sorting Algorithm Attributes

- Time Complexity
  - Big-O classification
  - naturally, the smallest classification is better
- Auxiliary space
  - how much memory is needed to run the algorithm
  - some algorithms require extra memory as large as the array itself

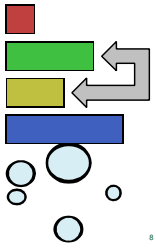
Sorting Algorithm Attributes

- Stable
  - what happens when two array elements, a and b, have the same sort value?
  - a "stable" sort will not change their positions – so, if a is before b, they will remain in that order
- Online
  - elements can be added at the same time that the data is being sorted
  - data can be *streamed* into the array at runtime



Bubble Sort

- The *bubble sort* is one of the least efficient algorithms …but it is easy to understand
- Basic approach
  - "lighter" elements "bubble up" to the top of the array
  - "heavier" items sink to the bottom



How It Works

- Consists of two For Loops
- Outer loop runs from the first to the last
- Inner loop …
  - runs from the bottom of the array *up* to the top (well, the position of the first loop)
  - it checks every two neighbor elements, if the they are out of order, it swaps them
  - so, the smallest element moves up the array

The Bubble Sort (Java-ish)

```
for(i = 0; i < count-1; i++)
{
   for(j = count-1; j > i; j--)
   {
      if (array[j-1] < array[j])
      {
         //swap array[j-1] and array[j]
      }
   }
}
```


Bubble Sort Example

Outer Loop
Inner Loop

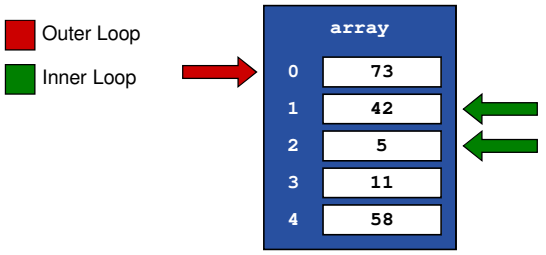
	array
0	73
1	42
2	11
3	58
4	5

Bubble Sort Example

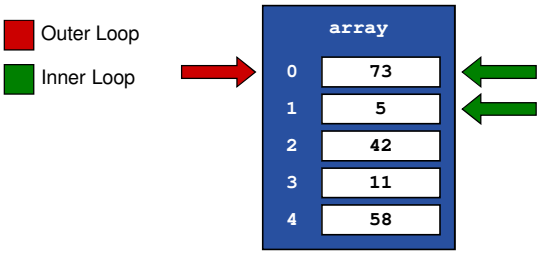
Outer Loop
Inner Loop

	array
0	73
1	42
2	11
3	5
4	58

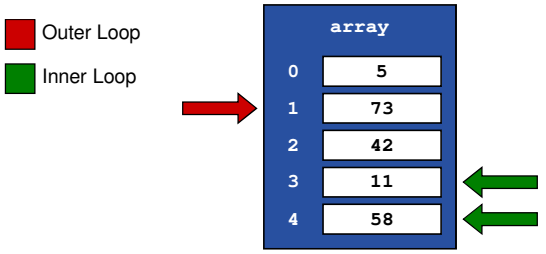
Bubble Sort Example



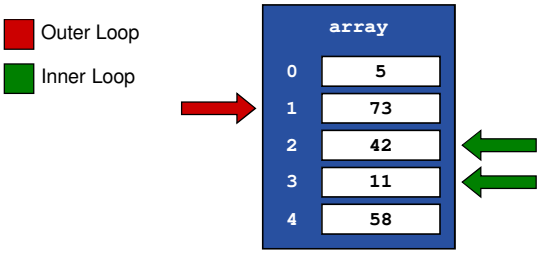
Bubble Sort Example



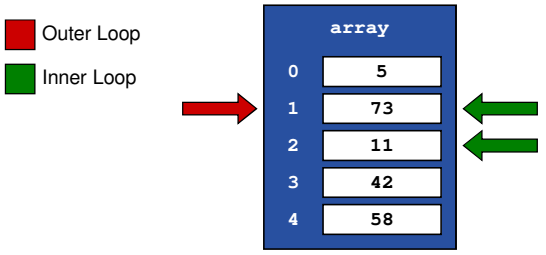
Bubble Sort Example



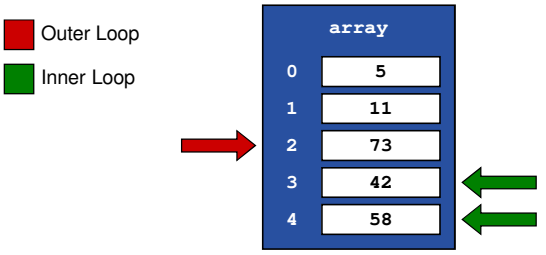
Bubble Sort Example



Bubble Sort Example



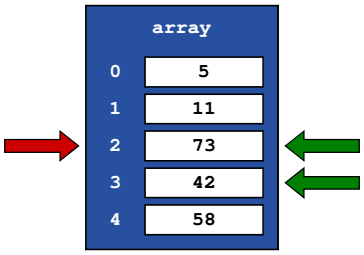
Bubble Sort Example



Bubble Sort Example

Outer Loop

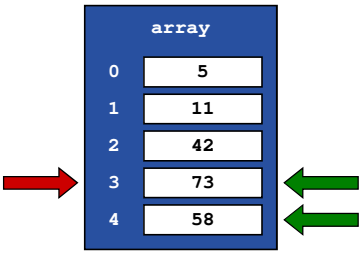
Inner Loop



Bubble Sort Example

Outer Loop

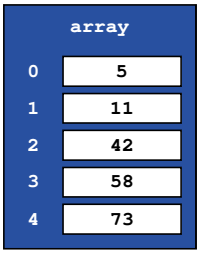
Inner Loop



Bubble Sort Example

Outer Loop

Inner Loop



Efficiency of the Bubble Sort

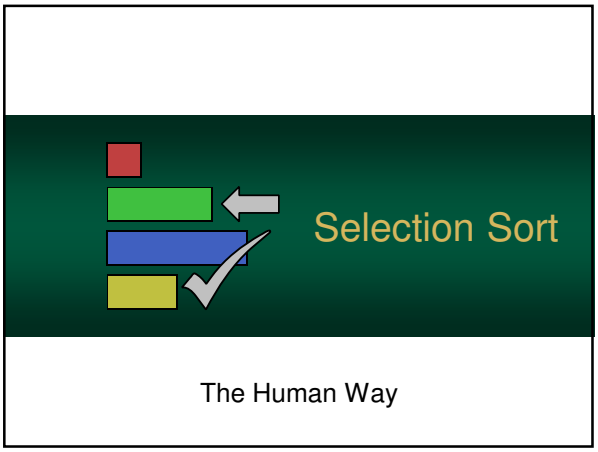
- The Bubble Sort is **extremely** inefficient and only good for tiny arrays
- Since Bubble Sort uses two embedded loops
  - the outer loop looks at all *n* items
  - the inner loop looks at basically *n* items
  - the resulting algorithm gets exponentially less efficient as *n* increases

Efficiency of the Bubble Sort

- The Bubble Sort, is considered $O(n^2)$
  - two embedded loops that are based on *n*
  - and… all that swapping doesn't help either!

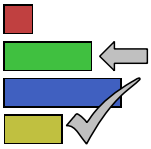
Bubble Sort Summary

Bubble Sort	
Time Average	$O(n^2)$
Time Best	$O(n^2)$
Time Worst	$O(n^2)$
Auxiliary space	O(1)
Stable	Yes – Equal element order preserved
Online?	No – Entire array in use



Selection Sort

- The *Selection Sort* is a similar to the Bubble Sort
- However…
  - rather than "bubble up" smaller items, it scans the entire array
  - it finds the smallest element
  - only *then* does it swap the values



Selection Sort

- Like the Bubble Sort, it consists of two For Loops – one outer and one inner
- Outer loop runs from the first to the last
- Inner loop …
  - starts at the position of the outer loop
  - scans down and finds the *smallest* value
- Then, after the scan, do a single swap

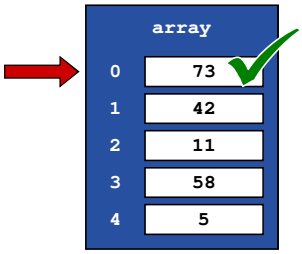
The Selection Sort

```
for(i = 0; i < count-1; i++)
{
   m = i;
   for(j = i; j < count; j++)
   {
      if (array[j] < array[m])
      {
         m = j;
      }
   }
   //swap array[i] and array[m]
}
```


Selection Sort Example

Outer Loop

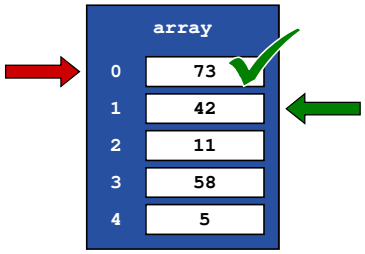
Inner Loop



Selection Sort Example

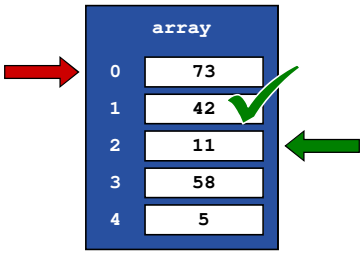
Outer Loop

Inner Loop



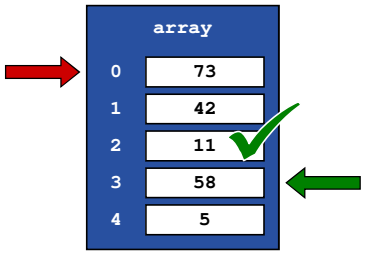
Selection Sort Example

Outer Loop
Inner Loop



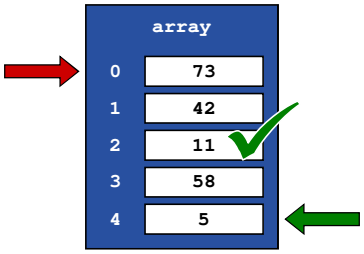
Selection Sort Example

Outer Loop
Inner Loop



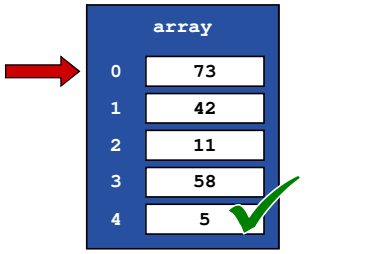
Selection Sort Example

Outer Loop
Inner Loop



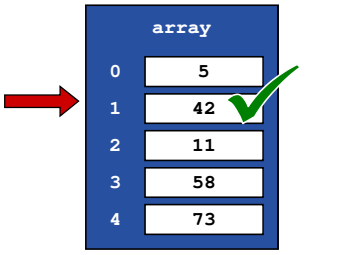
Selection Sort Example

Outer Loop
Inner Loop



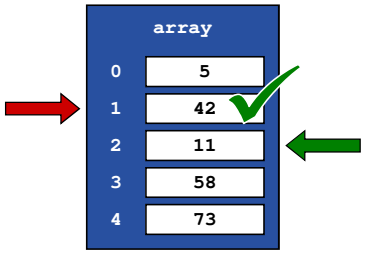
Selection Sort Example

Outer Loop
Inner Loop



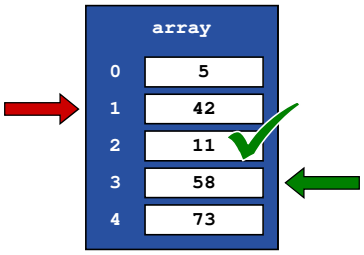
Selection Sort Example

Outer Loop
Inner Loop



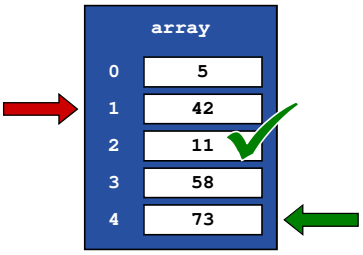
Selection Sort Example

Outer Loop
Inner Loop



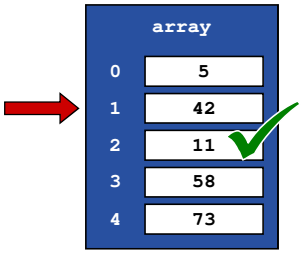
Selection Sort Example

Outer Loop
Inner Loop



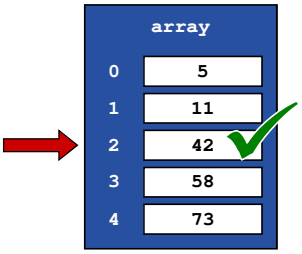
Selection Sort Example

Outer Loop
Inner Loop



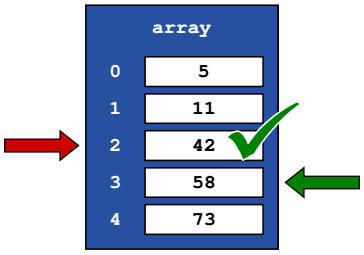
Selection Sort Example

Outer Loop
Inner Loop



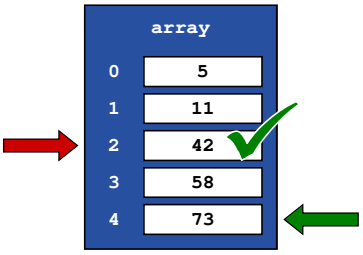
Selection Sort Example

Outer Loop
Inner Loop



Selection Sort Example

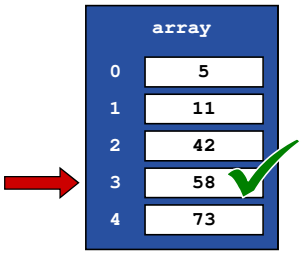
Outer Loop
Inner Loop



Selection Sort Example

Outer Loop

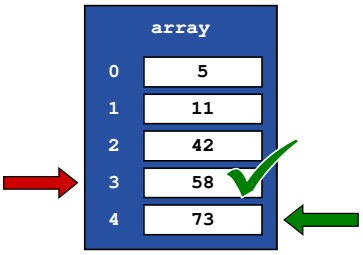
Inner Loop



Selection Sort Example

Outer Loop

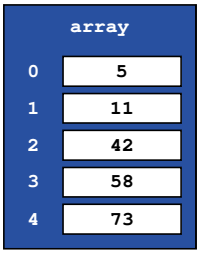
Inner Loop



Selection Sort Example

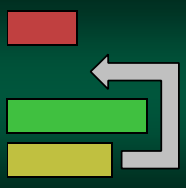
Outer Loop

Inner Loop



Selection Sort Summary

Selection Sort	
Time Average	$O(n^2)$
Time Best	$O(n^2)$
Time Worst	$O(n^2)$
Auxiliary space	O(1)
Stable	Yes – Equal element order preserved
Online?	No – Entire array in use



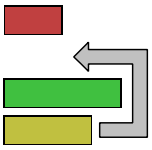
Insertion Sort

Building an sorted array... bit by bit

(er... byte by byte?)

Insertion Sort

- The *Insertion Sort* is a $O(n^2)$ sorting algorithm with several advantages over bubble-sort and selection-sort
- While it is still $O(n^2)$ is far more efficient than the other two



Deck of Cards

- Often, it is compared to sorting a deck of cards
- This is how you would manually sort a row of cards
  - if you start sorting on the left side, you will find a card, move it, and shift the rest of the cards right
  - you build a sorted list a bit at a time – on the left side of your row

How it Works

- The algorithm consists of two loops – one embedded within the other
- The outer loop starts and the top of the array and moves down
- The algorithm builds a sorted array above the outer loop.

How it Works

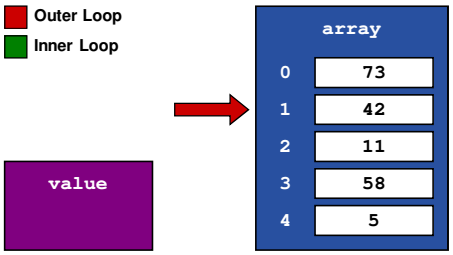
- Current array value is saved into a temporary variable
- Inner loop then searches all the values that come before it in the array
- If the value, being looked at, is larger than the saved value, it's moved down

The Insertion Sort

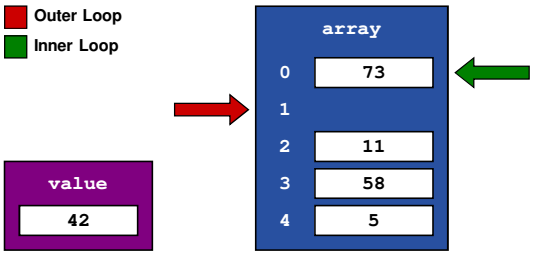
```
for (i = 1; i < count; i++)
{
   value = array[i];

   j = i - 1;
   while (j >= 0 && array[j] > value)
   {
      array[j + 1] = array[j];
      j--;
   }
   array[j + 1] = value;
}
```

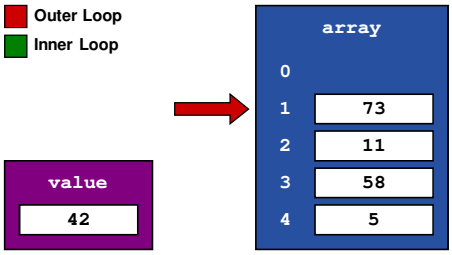

Insertion Sort Example



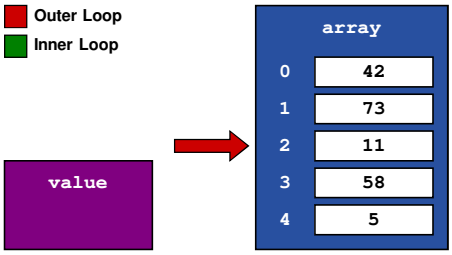
Insertion Sort Example



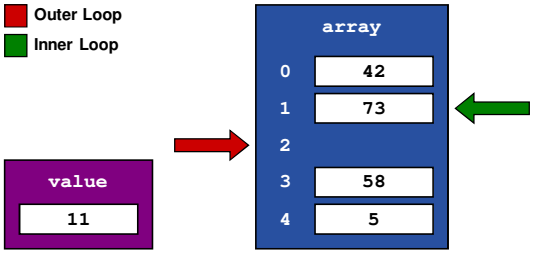
Insertion Sort Example



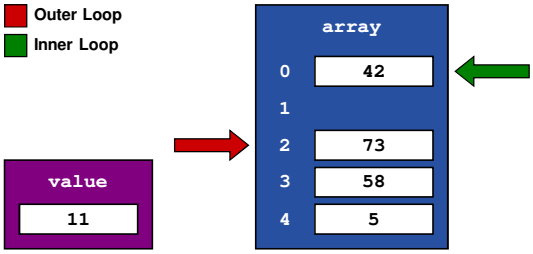
Insertion Sort Example



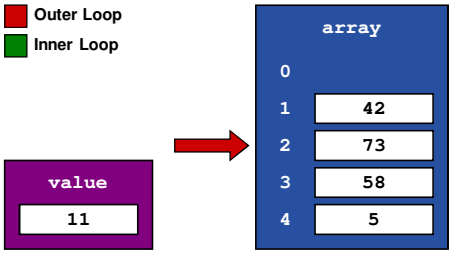
Insertion Sort Example



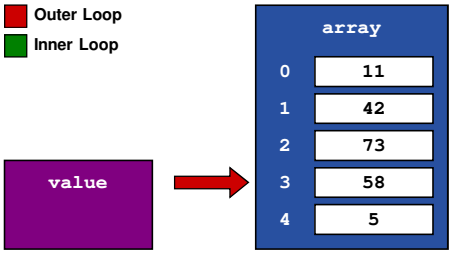
Insertion Sort Example



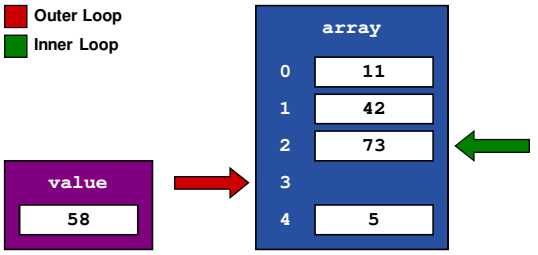
Insertion Sort Example



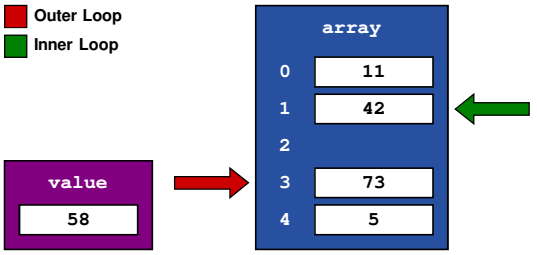
Insertion Sort Example



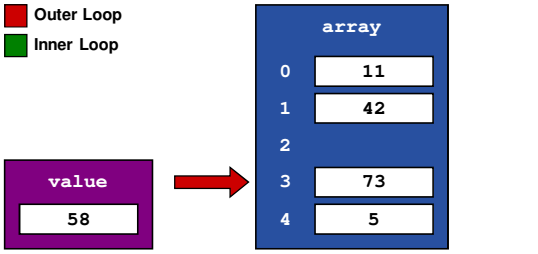
Insertion Sort Example



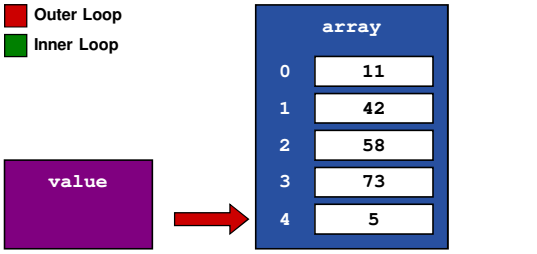
Insertion Sort Example



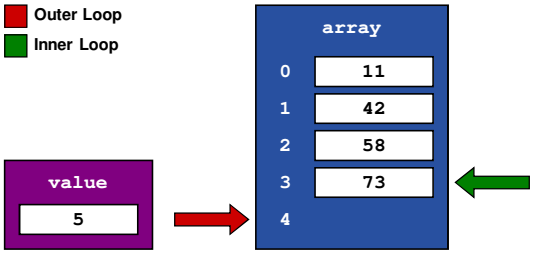
Insertion Sort Example



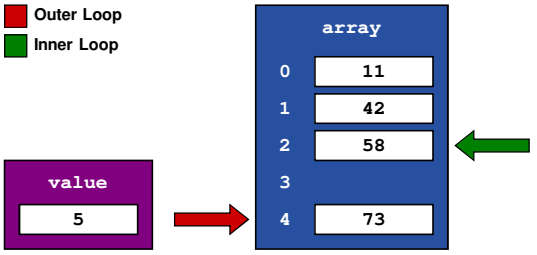
Insertion Sort Example



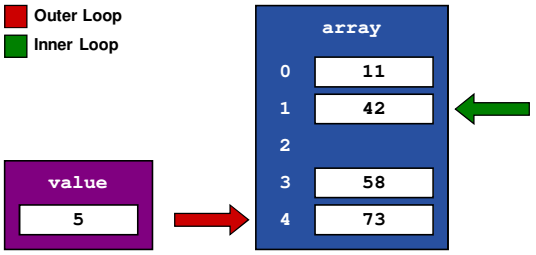
Insertion Sort Example



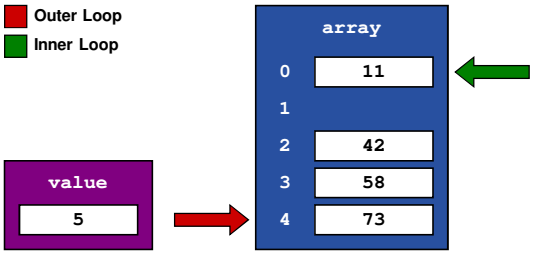
Insertion Sort Example



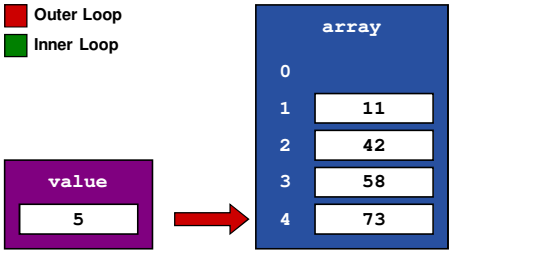
Insertion Sort Example



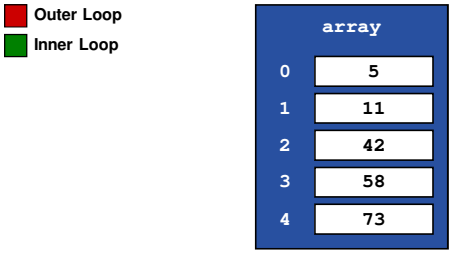
Insertion Sort Example



Insertion Sort Example



Insertion Sort Example



Advantages

- Because *Insertion Sort* creates a sorted array above the outer loop
  - inner loop, on average, only needs to move 1 / 2 positions up
  - data can be sent during the sorting process
  - this means the algorithm is considered *"online"* – i.e. it can sort *streaming* data

Advantages

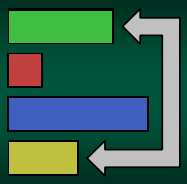
- Insertion sort does not "swap" values
  - most of the overhead of bubble and selection-sort is swapping
  - insertion sort moves data as it sorts, so, there is little unnecessary overhead
- Insertion sort is O(n) on sorted lists
  - inner loop stops when the current array value cannot be moved up
  - the more sorted the list, the more the inner loop approaches O(1)

Advantages

- Little to no auxiliary storage overhead
  - like Bubble-Sort and Selection-Sort, Insertion-Sort requires little storage overhead
  - so, in regards to *n*, storage complexity is O(1)

Insertion Sort Summary

Insertion Sort	
Time Average	$O(n^2)$
Time Best	O(n)
Time Worst	$O(n^2)$
Auxiliary space	O(1)
Stable	Yes – Equal element order preserved
Online?	Yes – Can sort streamed data

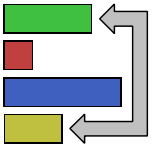


Shell Sort

Insertion Sort with an identity crisis

Shell Sort

- *Shell-Sort* is a special version of the Insertion-Sort created by Donald Shell in 1959
- Yes, it is named after the guy, *not a shell metaphor*
- But, ironically, that metaphor works



Shell Sort

- It was the first algorithm to break the $O(n^2)$ barrier
- For a few years, this was the fastest sort algorithm available – until O(n log n) was invented

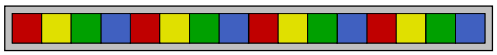
What is Going On?

- With insertion sort, each time we insert an element, the rest are moved one step closer to where they belong
- Can we move elements a larger distance than just one?
- Yes, Shell Sort works like Insertion Sort, but works on elements at large distances
- This distance is called the *gap*

What's Going On?

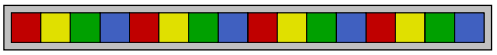
- Gap changes with each outer loop iteration
  - the distance between comparisons decreases as the sorting algorithm runs
  - in the last iteration, the gap is 1
  - so adjacent elements are compared – so it is a regular Insertion Sort
- Shell Sort is also known as a "diminishing increment sort"

Sorting "Shells"



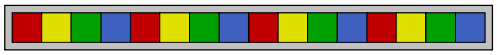
- Shell Sort orders elements that are spaced a relative distance from each other
- So, the red cells above are sorted *relative* to each other, as are the yellow, green, and blue cells

Sorting "Shells"



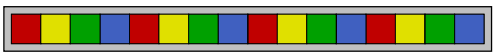
- The decreasing gaps are a sequence
- The notation $h_1, h_2, h_3, \ldots, h_t$ represents a sequence of increasing integer values which will be used (from right to left)
- Any sequence works if it $h_n > h_{n-1}$ and $h_1 = 1$

Gap Mathematics



- *h_k-sorted array* - all elements with gap h_k are sorted relative to each other
- So, after each phase of h_k...
  - for each i, we have array[i] ≤ array[i + h_k]
  - all elements spaced h_k apart are sorted.

Gap Mathematics



- Shell-Sort only works because an array that is h_k-sorted…
- …remains h_k-sorted when h_{k-1}-sorted.

So, What are Gap Values?

- For $h_1, h_2, h_3, \ldots, h_t$ we need to determine what the actual values will be
- Some sequences will be far more efficient than others
- Shell's original design...
  - cuts the gap in half for each iteration
  - starts at N / 2 (where N is the size of the array)
- There are several competing sequences

So, What are the Gap Values

- The algorithm is most efficient when...
  - the gap sequence are relatively prime
  - i.e. the sequence does not share any divisors
- However....
  - using a prime sequence is often not practical in a program – too much to store!
  - so, real, practical solutions attempt to approximate a relatively prime sequence

So, What are Gap Values?

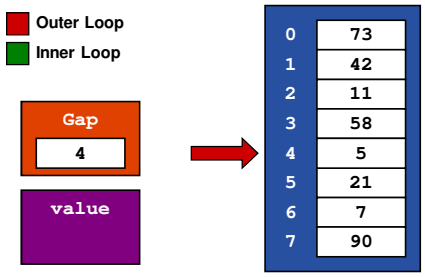
Creator	Sequence
Shell	1, ..., (n / 8), (n / 4), (n / 2)
Hibbard	1, 3, 7, ..., $2^k - 1$
Knuth	1, 4, 13, ..., $(3^k - 1) / 2$
Sedgewick	1, 5, 19, 41, 109, ..., $(4^k - 3 * 2^k + 1)$

The Shell Sort: Original

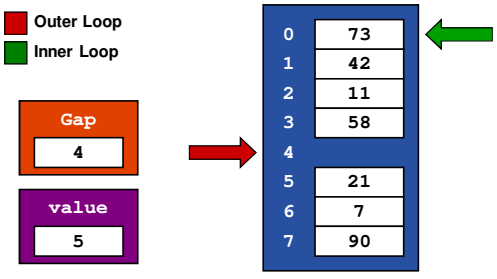
```
for (gap = count / 2; gap > 0; gap /= 2)
{
  for(i = gap; i < count; i++)
  {
     value = array[i];
     j = i;
     while (j >= gap && array[j - gap] > value)
     {
        array[j] = array[j - gap];
        j -= gap;
     }

     a[j] = value;
  }
}
```

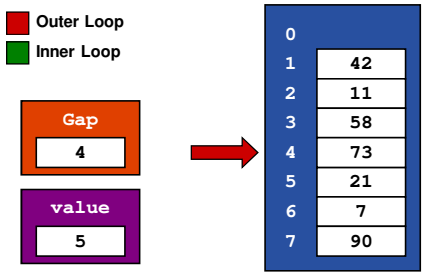

Gap = 4, First Outer Loop Pass



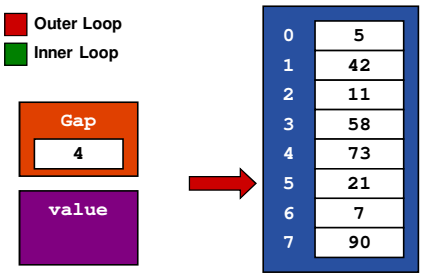
Gap = 4, Inner Loop



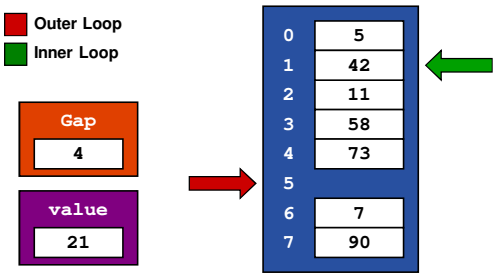
Gap = 4, Inner Loop Complete



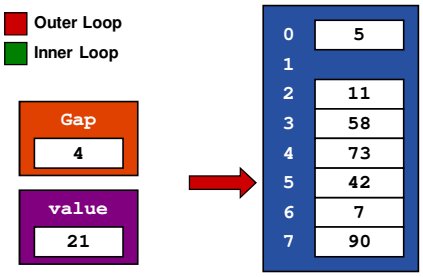
Gap = 4, Outer Loop #2



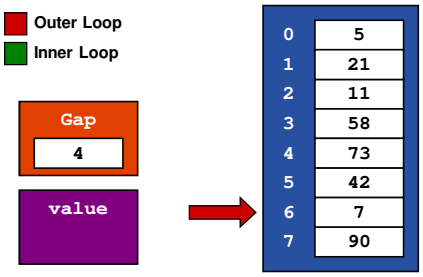
Gap = 4, Inner Loop



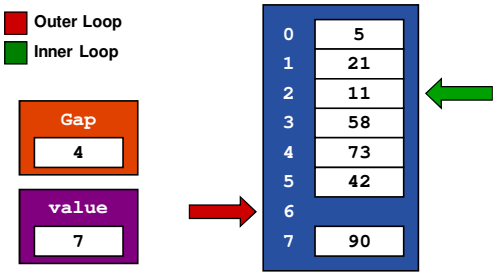
Gap = 4, Inner Loop Complete



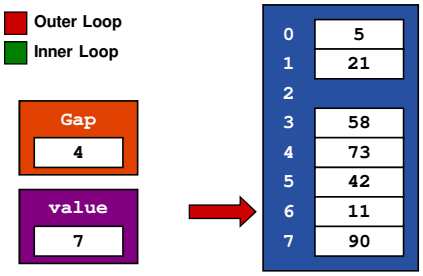
Gap = 4, Outer Loop #3



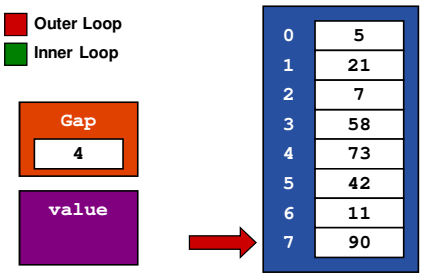
Gap = 4, Inner Loop



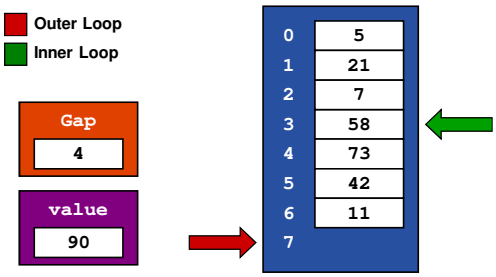
Gap = 4, Inner Loop Complete



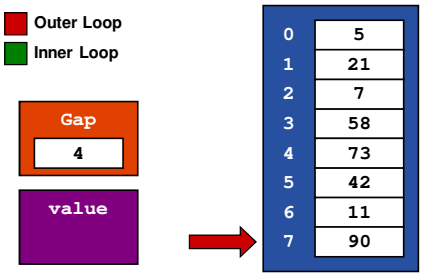
Gap = 4, Outer Loop #4



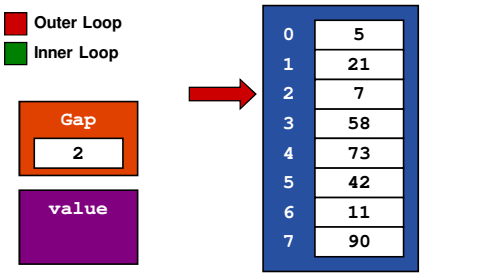
Gap = 4, Inner Loop



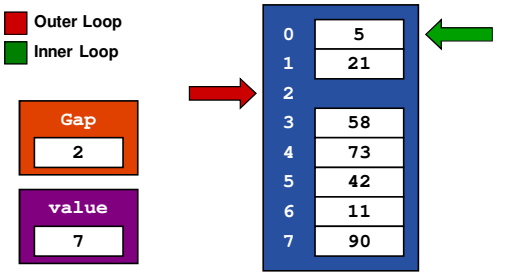
Gap = 4, Inner Loop Complete



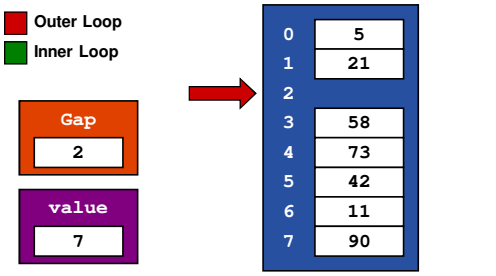
Gap = 2, Outer Loop First Pass



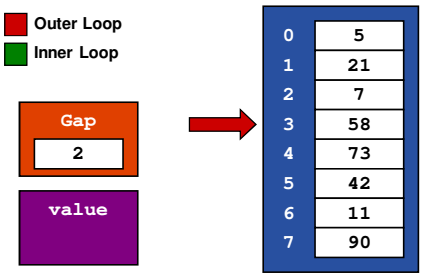
Gap = 2, Inner Loop



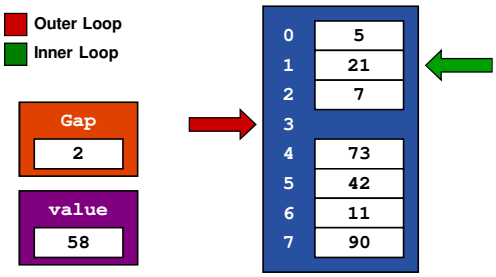
Gap = 2, Inner Loop Complete



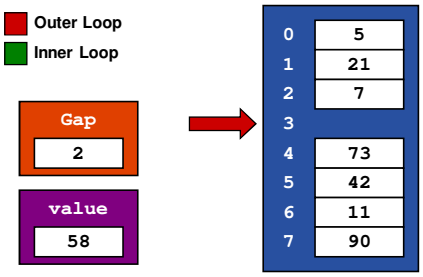
Gap = 2, Outer Loop #2



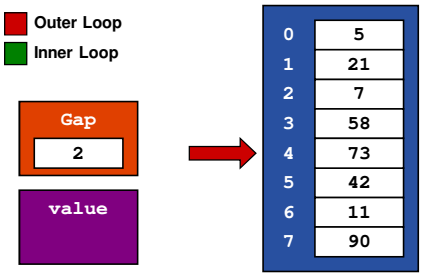
Gap = 2, Inner Loop



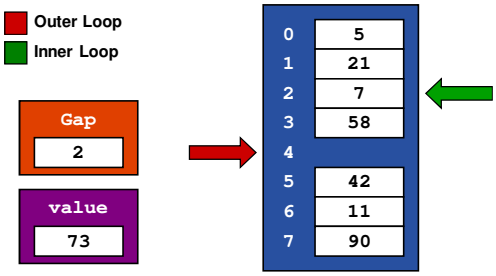
Gap = 2, Inner Loop Complete



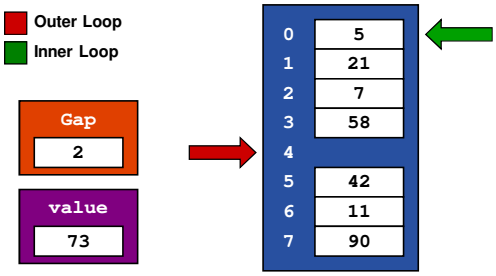
Gap = 2, Outer Loop #3



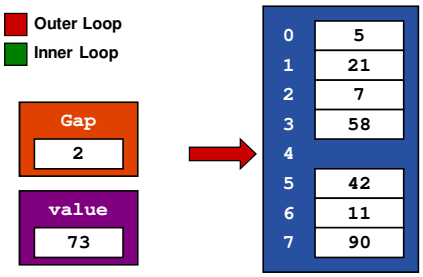
Gap = 2, Inner Loop



Gap = 2, Inner Loop



Gap = 2, Inner Loop Complete



... and so on....

- The example continues to sort for each h_k
- The outer loop continues to the bottom of the array
- Finally, gap will go to one and the sort acts just like an Insertion-Sort

Time Complexity

- Time complexity of Shell Sort is up for debate
- Although the algorithm is fairly simple, proving its time complexity is not
- What is known...
  - it is approximately $O(n^r)$ where $1 < r < 2$
  - this is ultimately faster than $O(n^2)$ but worse than O(n log n)

Time Complexity

- Empirical analysis of the algorithm has given some widely accepted values for average, best, and worst times
- Worst case performance (using Hibbard's sequence) is $O(n^{3/2})$
- Average performance is thought to be about $O(n^{5/4})$

Shell Sort Summary

Shell Sort	
Time Average	$\approx O(n^{5/4})$
Time Best	$\approx O(n \log n)$ – For a near sorted list
Time Worst	$\approx O(n^{3/2})$
Auxiliary space	O(1)
Stable	No – Equal element order not preserved
Online?	No – Entire array in use