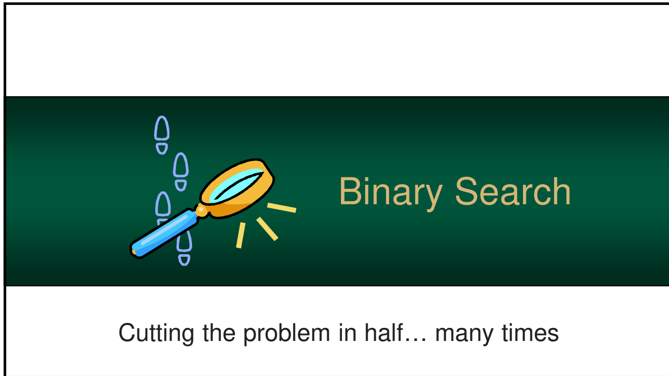
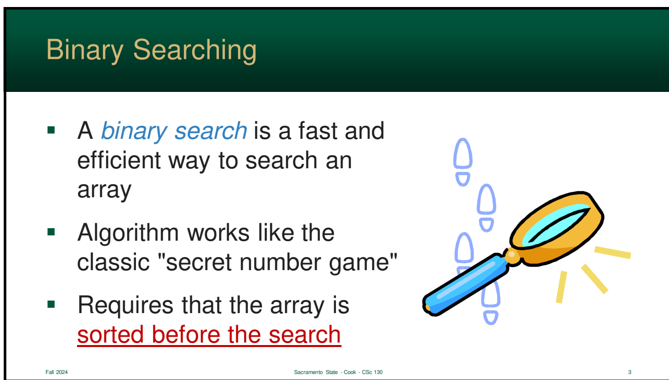
1



2

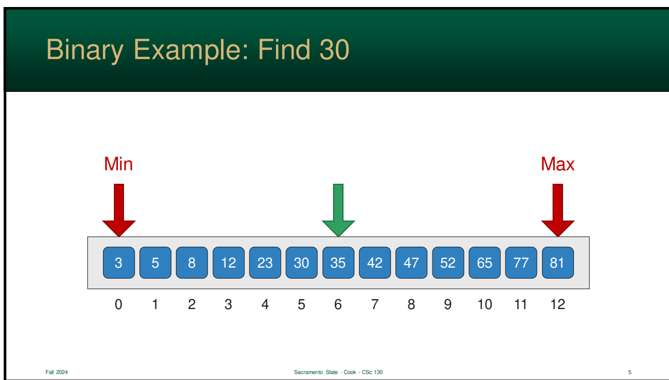


3

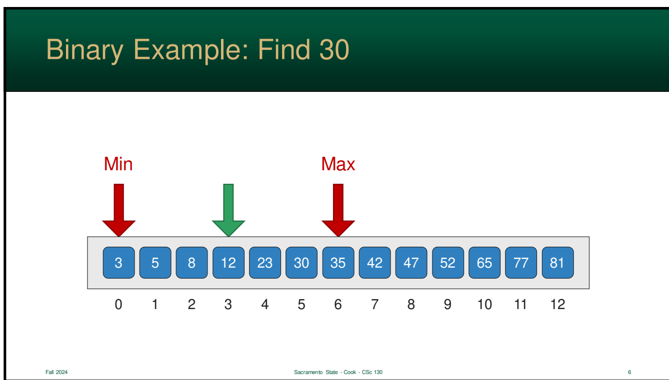
How it Works

- Starts knowing the max & min values
  - in the case of arrays, this is the min and max index
  - in the number game, it is the min and max value
- Algorithm continues
  - it looks at the midpoint between the first and last
  - if the value > target, the max is set to the midpoint
  - if the value < target, the min is set to the midpoint
  - *this eliminates half of the numbers each iteration*

4



5



6

Binary Example: Find 30

Min Max

3	5	8	12	23	30	35	42	47	52	65	77	81
0	1	2	3	4	5	6	7	8	9	10	11	12

Binary Example: Find 30

Min Max

3	5	8	12	23	30	35	42	47	52	65	77	81
0	1	2	3	4	5	6	7	8	9	10	11	12

Binary Example: Find 30

Min Max

3	5	8	12	23	30	35	42	47	52	65	77	81
0	1	2	3	4	5	6	7	8	9	10	11	12

Binary Example: Find 30

Min Max

3	5	8	12	23	30	35	42	47	52	65	77	81
0	1	2	3	4	5	6	7	8	9	10	11	12

Benefits

- The binary search is incredibly efficient and **absolutely necessary** for large arrays
- Any item can be found only $\log_2(n)$ searches! It is O(log n)
- However, since array must be sorted, sorting algorithms are equally vital

Maximum # of Searches

Array Size	Sequential	Binary
10	10	4
100	100	7
1,000	1,000	10
10,000	10,000	14
100,000	100,000	17
1,000,000	1,000,000	20
10,000,000	10,000,000	24
100,000,000	100,000,000	27
1,000,000,000	1,000,000,000	30

Sorting

Bringing Order of "Chaos"

13

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

14

Sorting

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

15

Sorting Algorithm Attributes

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

16

Sorting Algorithm Attributes

3. Stable
   - what happens when two array elements, a and b, have the same sort value?
   - if a is initially stored before b, a "stable" sort will not change their relative positions
4. Online
   - elements can be added at the same time that the data is being sorted
   - data can be *streamed* into the array at runtime

17

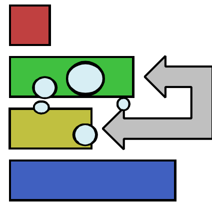
Bubble Sort

Carbonated Sorting

18

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



Fall 2024 Sacramento State - Cook - CSc 130 19

19

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

Fall 2024 Sacramento State - Cook - CSc 130 20

20

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]
      }
   }
}
```

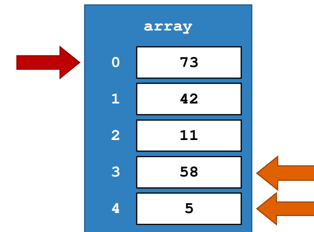
Fall 2024 Sacramento State - Cook - CSc 130 21

21

Bubble Sort Example

Outer Loop

Inner Loop



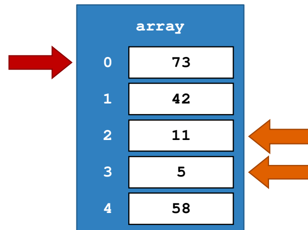
Fall 2024 Sacramento State - Cook - CSc 130 22

22

Bubble Sort Example

Outer Loop

Inner Loop



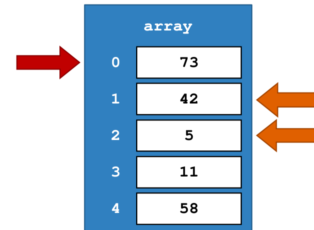
Fall 2024 Sacramento State - Cook - CSc 130 23

23

Bubble Sort Example

Outer Loop

Inner Loop



Fall 2024 Sacramento State - Cook - CSc 130 24

24

Bubble Sort Example

Outer Loop
Inner Loop

	array
0	73
1	5
2	42
3	11
4	58

25

Bubble Sort Example

Outer Loop
Inner Loop

	array
0	5
1	73
2	42
3	11
4	58

26

Bubble Sort Example

Outer Loop
Inner Loop

	array
0	5
1	73
2	42
3	11
4	58

27

Bubble Sort Example

Outer Loop
Inner Loop

	array
0	5
1	73
2	11
3	42
4	58

28

Bubble Sort Example

Outer Loop
Inner Loop

	array
0	5
1	11
2	73
3	42
4	58

29

Bubble Sort Example

Outer Loop
Inner Loop

	array
0	5
1	11
2	73
3	42
4	58

30

Bubble Sort Example

- Outer Loop
- Inner Loop

array	
0	5
1	11
2	42
3	73
4	58

31

Bubble Sort Example

- Outer Loop
- Inner Loop

array	
0	5
1	11
2	42
3	58
4	73

32

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

33

Efficiency of the Bubble Sort

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

34

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

35

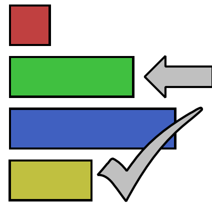
Selection Sort

The Human Way

36

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



37

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

38

The Selection Sort

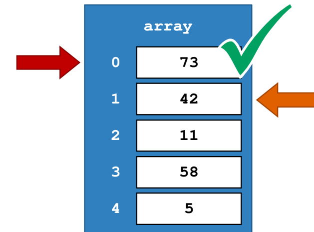
```
for(i = 0; i < count-1; i++)
{
    best = i;
    for(j = i; j < count; j++)
    {
        if (array[j] < array[best])
        {
            best = j;
        }
    }
    //swap array[i] and array[best]
}
```


39

Selection Sort Example

Outer Loop

Inner Loop

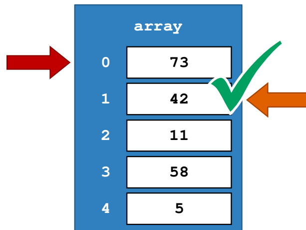


40

Selection Sort Example: New Best

Outer Loop

Inner Loop

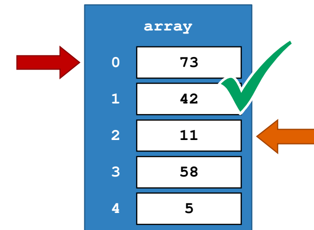


41

Selection Sort Example

Outer Loop

Inner Loop



42

Selection Sort Example: New Best

Outer Loop
Inner Loop

array	
0	73
1	42
2	11
3	58
4	5

Fall 2024 Sacramento State - Cook - CSc 130 43

43

Selection Sort Example

Outer Loop
Inner Loop

array	
0	73
1	42
2	11
3	58
4	5

Fall 2024 Sacramento State - Cook - CSc 130 44

44

Selection Sort Example

Outer Loop
Inner Loop

array	
0	73
1	42
2	11
3	58
4	5

Fall 2024 Sacramento State - Cook - CSc 130 45

45

Selection Sort Example: New Best

Outer Loop
Inner Loop

array	
0	73
1	42
2	11
3	58
4	5

Fall 2024 Sacramento State - Cook - CSc 130 46

46

Selection Sort Example: Swapped

Outer Loop
Inner Loop

array	
0	5
1	42
2	11
3	58
4	73

Fall 2024 Sacramento State - Cook - CSc 130 47

47

Selection Sort Example: Search Again

Outer Loop
Inner Loop

array	
0	5
1	42
2	11
3	58
4	73

Fall 2024 Sacramento State - Cook - CSc 130 48

48

Selection Sort Example: New Best

Outer Loop
Inner Loop

array	
0	5
1	42
2	11
3	58
4	73

49

Selection Sort Example

Outer Loop
Inner Loop

array	
0	5
1	42
2	11
3	58
4	73

50

Selection Sort Example

Outer Loop
Inner Loop

array	
0	5
1	42
2	11
3	58
4	73

51

Selection Sort Example: Swapped

Outer Loop
Inner Loop

array	
0	5
1	11
2	42
3	58
4	73

52

Selection Sort Example: Search Again

Outer Loop
Inner Loop

array	
0	5
1	11
2	42
3	58
4	73

53

Selection Sort Example

Outer Loop
Inner Loop

array	
0	5
1	11
2	42
3	58
4	73

54

Selection Sort Example: No Swap

Outer Loop

Inner Loop

array	
0	5
1	11
2	42
3	58
4	73

55

Selection Sort Example: Search Again

Outer Loop

Inner Loop

array	
0	5
1	11
2	42
3	58
4	73

56

Selection Sort Example: No Swap

Outer Loop

Inner Loop

array	
0	5
1	11
2	42
3	58
4	73

57

Selection Sort Example: Done

Outer Loop

Inner Loop

array	
0	5
1	11
2	42
3	58
4	73

58

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

59

Insertion Sort

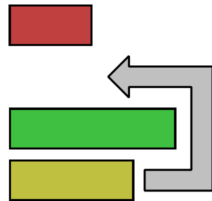
Building a sorted array... bit by bit

(er... byte by byte?)

60

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

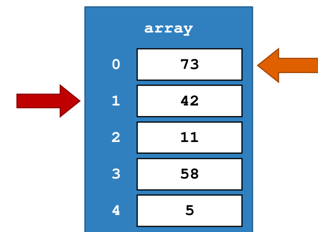
```
for (i = 1; i < count; i++)
{
   value = array[i];

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

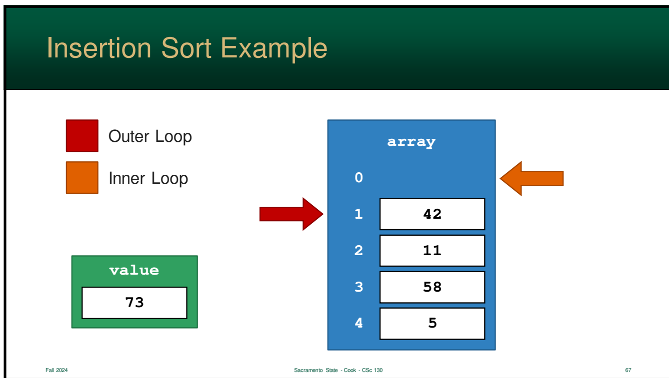

Insertion Sort Example

Outer Loop

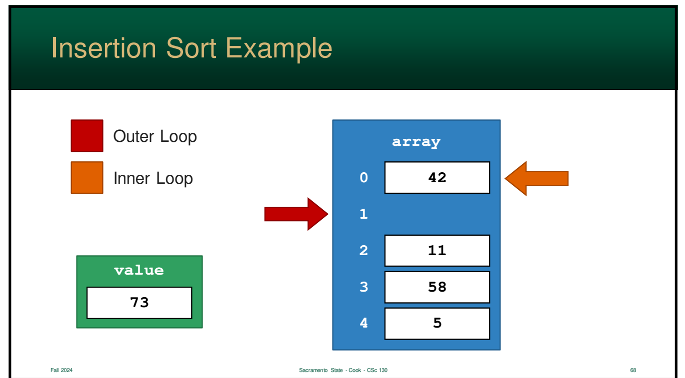
Inner Loop

value

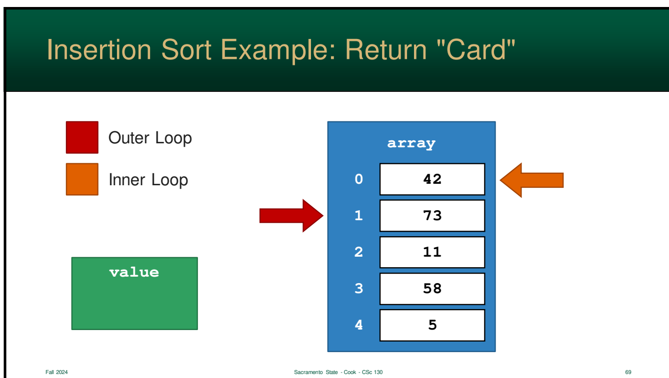




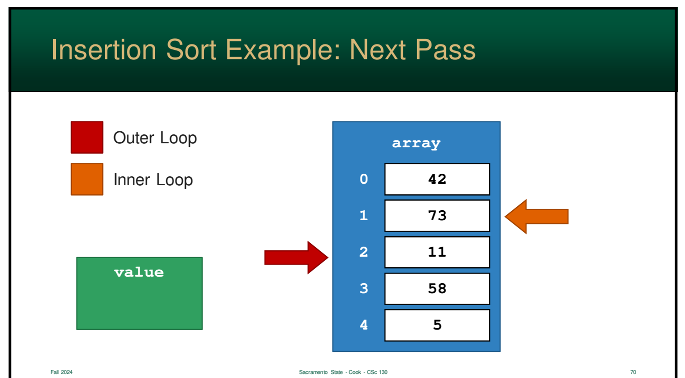
67



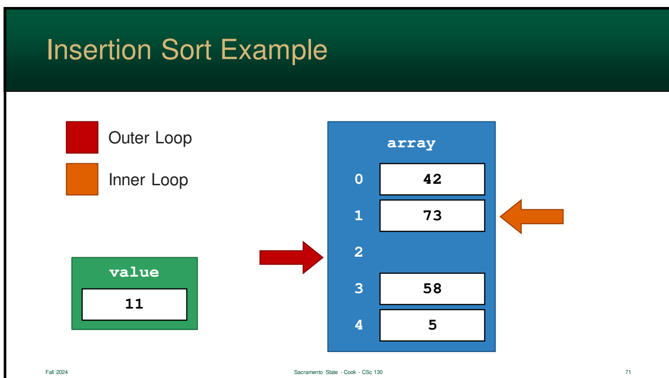
68



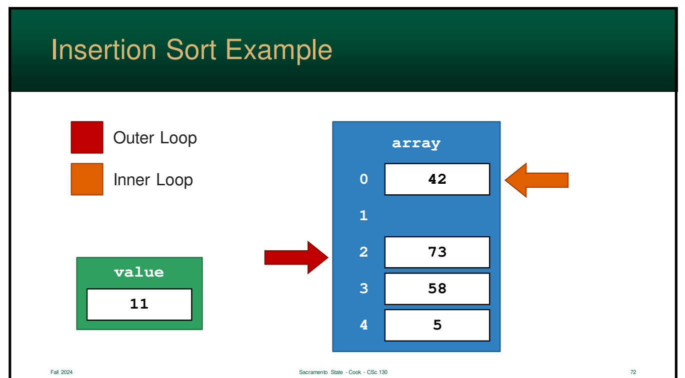
69



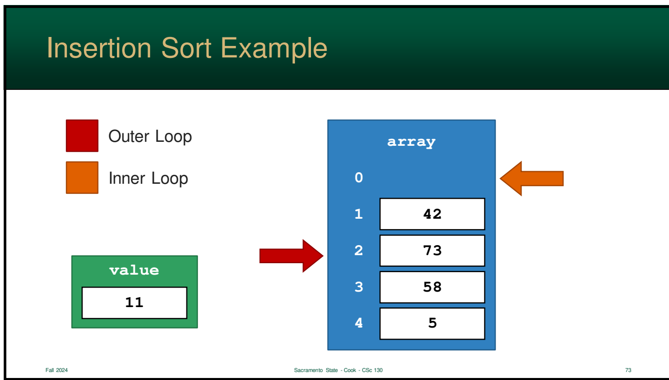
70



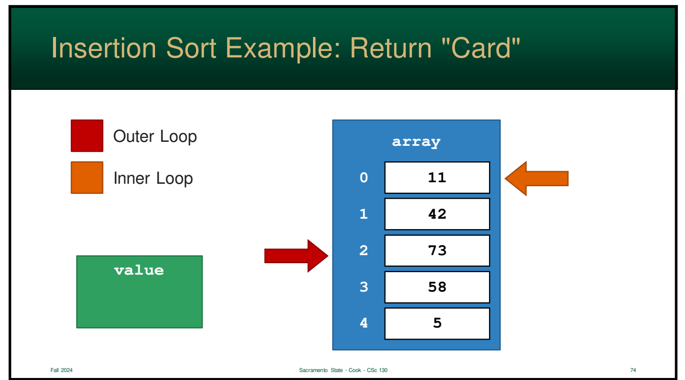
71



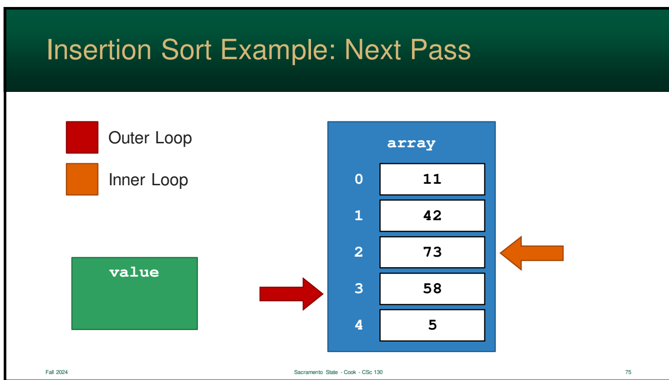
72



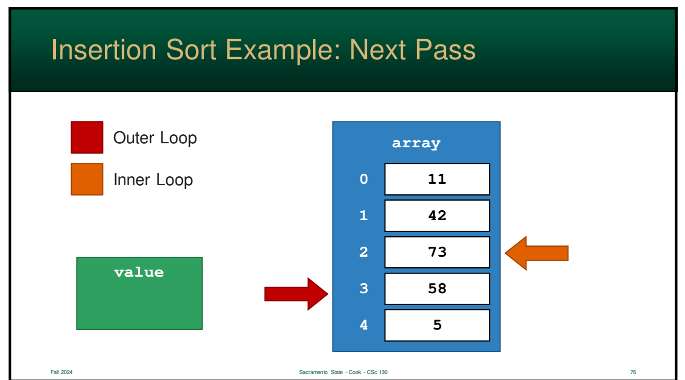
73



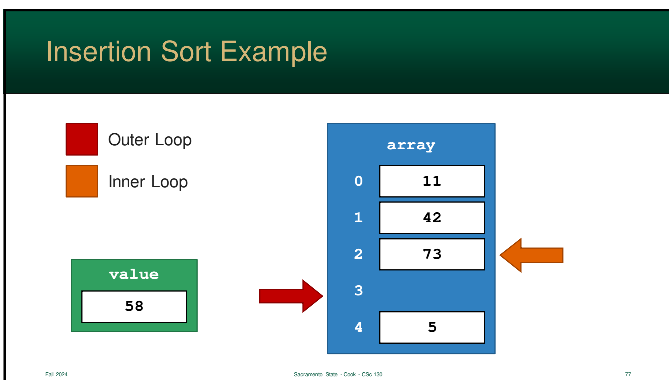
74



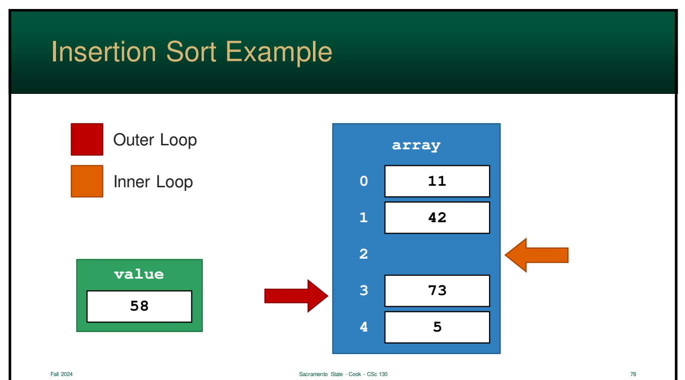
75



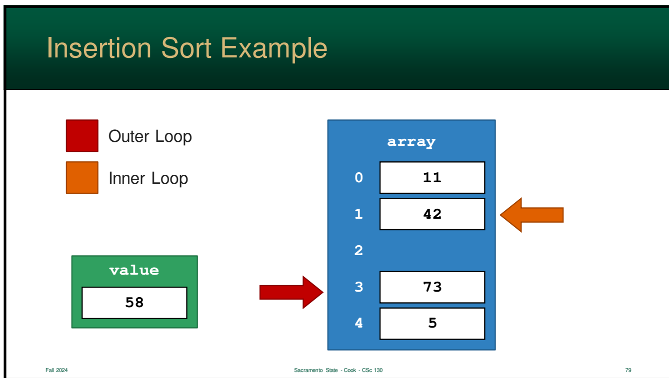
76



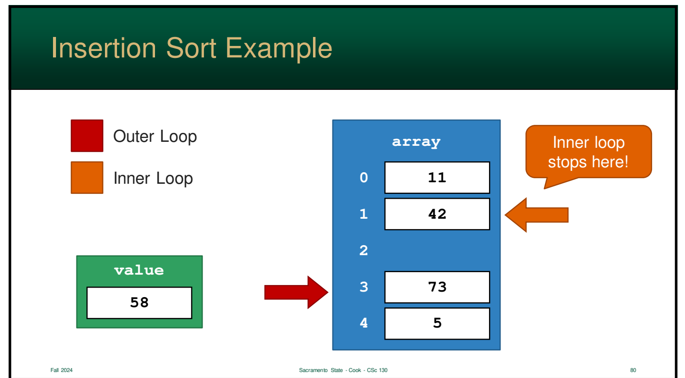
77



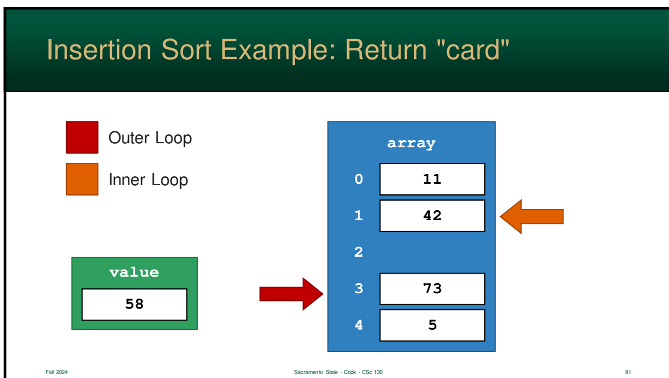
78



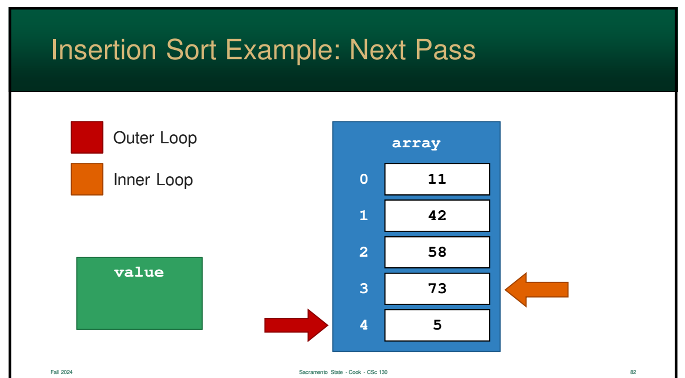
79



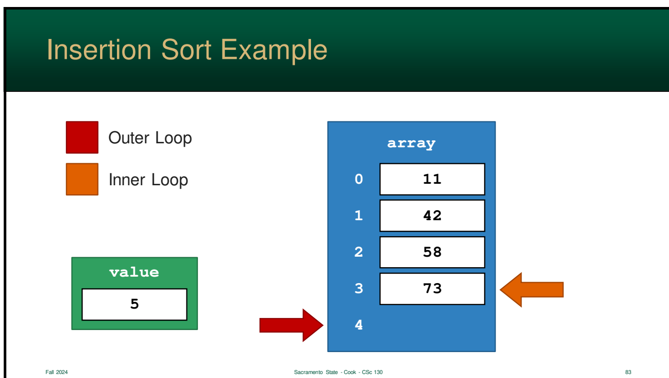
80



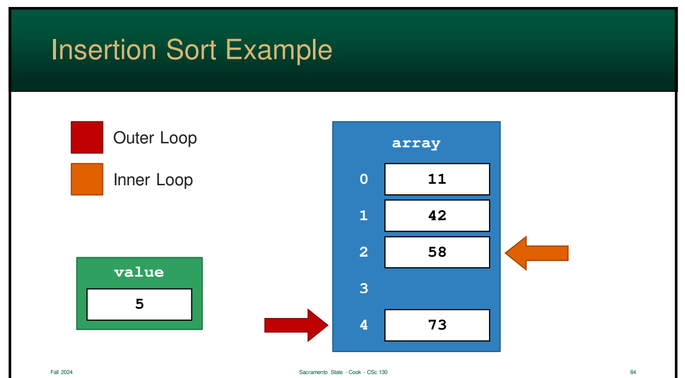
81



82



83



84

Insertion Sort Example

Outer Loop
Inner Loop

array	
0	11
1	42
2	
3	58
4	73

value: 5

Fall 2024 Sacramento State - Cook - CSc 130 85

85

Insertion Sort Example

Outer Loop
Inner Loop

array	
0	11
1	
2	42
3	58
4	73

value: 5

Fall 2024 Sacramento State - Cook - CSc 130 86

86

Insertion Sort Example

Outer Loop
Inner Loop

array	
0	
1	11
2	42
3	58
4	73

value: 5

Fall 2024 Sacramento State - Cook - CSc 130 87

87

Insertion Sort Example: Return "Card"

Outer Loop
Inner Loop

array	
0	5
1	11
2	42
3	58
4	73

value

Fall 2024 Sacramento State - Cook - CSc 130 88

88

Insertion Sort Example: Done

Outer Loop
Inner Loop

array	
0	5
1	11
2	42
3	58
4	73

value

Fall 2024 Sacramento State - Cook - CSc 130 89

89

Advantages

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

Fall 2024 Sacramento State - Cook - CSc 130 90

90

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
- 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)

91

Advantages

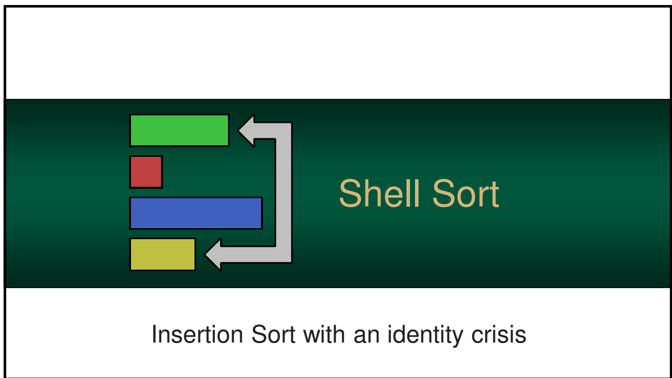
- Insertion sort is O(n) on sorted lists
  - inner loop stops when the current array value cannot be moved up
  - the more sorted the list, the more the inner loop approaches O(1)

92

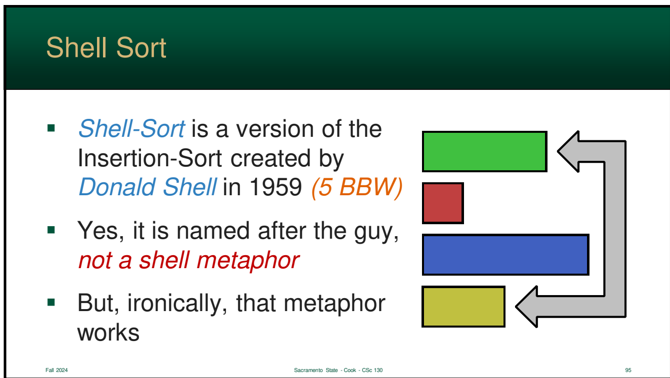
Insertion Sort Summary

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

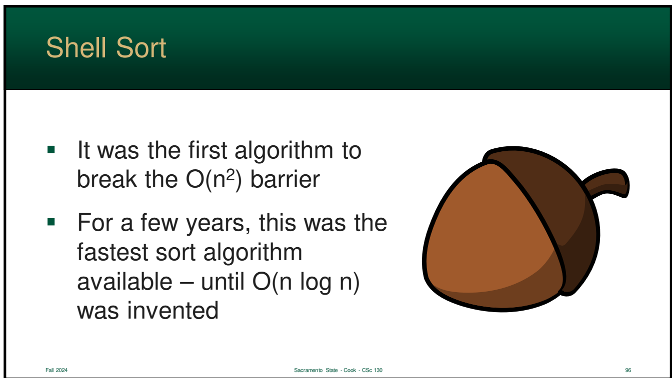
93



94



95



96

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*

97

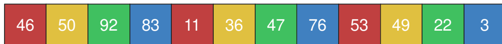
What's Going On?

- 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, at that point, adjacent elements are compared – so it is a regular Insertion Sort
- Shell Sort is also known as a "diminishing increment sort"

98

Sorting "Shells"

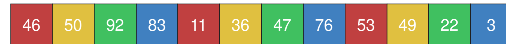
- Shell Sort orders elements that are spaced a relative distance from each other
- So, the red cells above are sorted relative to each other, as are the yellow, green, and blue elements



99

Sorting "Shells"

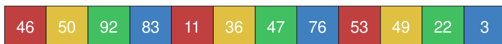
- The decreasing gaps are a sequence
- The notation $h_1, h_2, h_3, \ldots, h_t$ represents a sequence of increasing integer values which will be used (from right to left)
- Any sequence works if it $h_n > h_{n-1}$ and $h_1 = 1$



100

Each Shell is Sorted

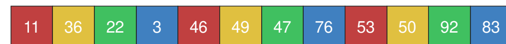
- *h_k-sorted array* - all elements with gap h_k are sorted relative to each other
- For each i, we have array[i] ≤ array[i + h_k]
- All elements spaced h_k apart will be sorted



101

Each Shell is Sorted

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



102

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 as better than others
- Shell's original design...
  - starts at N / 2 (where N is the size of the array)
  - cuts the gap in half for each iteration
- There are several competing sequences

103

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

104

So, What are Gap Values?

Creator	Sequence
Shell	1, ..., (n / 8), (n / 4), (n / 2)
Hibbard	1, 3, 7, 15, 31, ..., 2^k - 1
Knuth	1, 4, 13, 40, 121, ..., (3^k - 1) / 2
Sedgewick	1, 5, 19, 41, 109, ..., (4^k - 3 * 2^k + 1)

105

The Shell Sort: Original

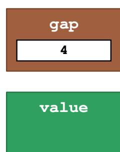
```
for (gap = count / 2; gap > 0; gap /= 2)
{
   for(i = gap; i < count; i++)
   {
      value = array[i];
      j = i;
      while (j >= gap && array[j - gap] >  value)
      {
          array[j] = array[j - gap];
          j -= gap;
      }

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


106

Gap = 4, First Outer Loop Pass

Outer Loop

Inner Loop



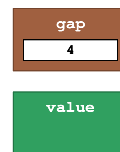
array	
0	73
1	42
2	11
3	58
4	5
5	21
6	7
7	90

107

Gap = 4, So, 4 overlapped arrays

Outer Loop

Inner Loop



array	
0	73
1	42
2	11
3	58
4	5
5	21
6	7
7	90

108

Gap = 4, Outer Loop starts at 4 (the gap)

Outer Loop
Inner Loop

gap: 4

value:

array	
0	73
1	42
2	11
3	58
4	5
5	21
6	7
7	90

109

Inner Loop starts at "gap" indexes up

Outer Loop
Inner Loop

gap: 4

value:

array	
0	73
1	42
2	11
3	58
4	5
5	21
6	7
7	90

110

Outer Loop (Index 4): Remove 5

Outer Loop
Inner Loop

gap: 4

value: 5

array	
0	73
1	42
2	11
3	58
4	
5	21
6	7
7	90

111

Move value down (Insertion Sort-like)

Outer Loop
Inner Loop

gap: 4

value: 5

array	
0	
1	42
2	11
3	58
4	73
5	21
6	7
7	90

112

Inner Loop Done – Return Value

Outer Loop
Inner Loop

gap: 4

value:

array	
0	5
1	42
2	11
3	58
4	73
5	21
6	7
7	90

113

Outer Loop (Index 5): remove 21

Outer Loop
Inner Loop

gap: 4

value:

array	
0	5
1	42
2	11
3	58
4	73
5	21
6	7
7	90

114

Move 42 Down (its greater than 21)

Outer Loop
Inner Loop

array	
0	5
1	42
2	11
3	58
4	73
5	
6	7
7	90

gap: 4

value: 21

Fall 2024 — Sacramento State - Cook - CSc 130 — 115

115

Inner Loop Complete. Return 21

Outer Loop
Inner Loop

array	
0	5
1	
2	11
3	58
4	73
5	42
6	7
7	90

gap: 4

value: 21

Fall 2024 — Sacramento State - Cook - CSc 130 — 116

116

Outer Loop (Index 6): Remove 7

Outer Loop
Inner Loop

array	
0	5
1	21
2	11
3	58
4	73
5	42
6	7
7	90

gap: 4

value:

Fall 2024 — Sacramento State - Cook - CSc 130 — 117

117

Move 11 Down (its greater than 7)

Outer Loop
Inner Loop

array	
0	5
1	21
2	11
3	58
4	73
5	42
6	
7	90

gap: 4

value: 7

Fall 2024 — Sacramento State - Cook - CSc 130 — 118

118

Inner Loop Complete: Return 7

Outer Loop
Inner Loop

array	
0	5
1	21
2	
3	58
4	73
5	42
6	11
7	90

gap: 4

value: 7

Fall 2024 — Sacramento State - Cook - CSc 130 — 119

119

Outer Loop (Index 7): Remove 90

Outer Loop
Inner Loop

array	
0	5
1	21
2	7
3	58
4	73
5	42
6	11
7	90

gap: 4

value:

Fall 2024 — Sacramento State - Cook - CSc 130 — 120

120

Don't Move 58 Down. It's less than 90.

Outer Loop
Inner Loop

gap: 4
value: 90

array	
0	5
1	21
2	7
3	58
4	73
5	42
6	11
7	

Fall 2024 Sacramento State - Cook - CSc 130 121

121

Inner Loop Complete: Return 90

Outer Loop
Inner Loop

gap: 4
value: 90

array	
0	5
1	21
2	7
3	58
4	73
5	42
6	11
7	

Fall 2024 Sacramento State - Cook - CSc 130 122

122

Gap of 4 is Compete

Outer Loop
Inner Loop

gap: 4
value:

array	
0	5
1	21
2	7
3	58
4	73
5	42
6	11
7	90

Note that each overlapped array is sorted

Fall 2024 Sacramento State - Cook - CSc 130 123

123

Gap = 2. So now, 2 overlapped arrays

Outer Loop
Inner Loop

gap: 2
value:

array	
0	5
1	21
2	7
3	58
4	73
5	42
6	11
7	90

Fall 2024 Sacramento State - Cook - CSc 130 124

124

Gap = 2, Outer Loop #1: Remove 7

Outer Loop
Inner Loop

gap: 2
value:

array	
0	5
1	21
2	7
3	58
4	73
5	42
6	11
7	90

Fall 2024 Sacramento State - Cook - CSc 130 125

125

Don't Move 5 Down (smaller than 7)

Outer Loop
Inner Loop

gap: 2
value: 7

array	
0	5
1	21
2	
3	58
4	73
5	42
6	11
7	90

Fall 2024 Sacramento State - Cook - CSc 130 126

126

Inner Loop Complete: Return 7

Outer Loop
Inner Loop

gap: 2
value: 7

array	
0	5
1	21
2	
3	58
4	73
5	42
6	11
7	90

Fall 2024 Sacramento State - Cook - CSc 130 127

127

Inner Loop Complete: Return 7.

Outer Loop
Inner Loop

gap: 2
value:

array	
0	5
1	21
2	7
3	58
4	73
5	42
6	11
7	90

Fall 2024 Sacramento State - Cook - CSc 130 128

128

Outer Loop (Index 3): Remove 58

Outer Loop
Inner Loop

gap: 2
value:

array	
0	5
1	21
2	7
3	58
4	73
5	42
6	11
7	90

Fall 2024 Sacramento State - Cook - CSc 130 129

129

Don't move 21 down (smaller than 58)

Outer Loop
Inner Loop

gap: 2
value: 58

array	
0	5
1	21
2	7
3	
4	73
5	42
6	11
7	90

Fall 2024 Sacramento State - Cook - CSc 130 130

130

Outer Loop (Index 4): Remove 73

Outer Loop
Inner Loop

gap: 2
value:

array	
0	5
1	21
2	7
3	58
4	73
5	42
6	11
7	90

Fall 2024 Sacramento State - Cook - CSc 130 131

131

Don't move 7 down. Inner Loop Stops.

Outer Loop
Inner Loop

gap: 2
value: 73

array	
0	5
1	21
2	7
3	58
4	
5	42
6	11
7	90

Inner loop stops here!

Fall 2024 Sacramento State - Cook - CSc 130 132

132

Inner Loop Complete: Return 7.

Outer Loop
Inner Loop

gap: 2
value: 73

array	
0	5
1	21
2	7
3	58
4	
5	42
6	11
7	90

Fall 2024 Sacramento State - Cook - CSc 130 133

133

Outer Loop (Index 5): Remove 42

Outer Loop
Inner Loop

gap: 2
value:

array	
0	5
1	21
2	7
3	58
4	73
5	42
6	11
7	90

Fall 2024 Sacramento State - Cook - CSc 130 134

134

Move 58 down (it's greater than 42)

Outer Loop
Inner Loop

gap: 2
value: 42

array	
0	5
1	21
2	7
3	58
4	73
5	
6	11
7	90

Fall 2024 Sacramento State - Cook - CSc 130 135

135

Inner Loop Complete. Return 42

Outer Loop
Inner Loop

gap: 2
value: 42

array	
0	5
1	21
2	7
3	
4	73
5	58
6	11
7	90

Fall 2024 Sacramento State - Cook - CSc 130 136

136

... 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

Fall 2024 Sacramento State - Cook - CSc 130 137

137

Gap of 2 is Compete

Outer Loop
Inner Loop

gap: 2
value:

array	
0	5
1	21
2	7
3	42
4	11
5	58
6	73
7	90

Note that each overlapped array is sorted

Fall 2024 Sacramento State - Cook - CSc 130 138

138

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 $O(n \log n)$

Fall 2024 Sacramento State - Cook - CSc 130 139

139

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 $O(n^{5/4})$

Fall 2024 Sacramento State - Cook - CSc 130 140

140

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

Fall 2024 Sacramento State - Cook - CSc 130 141

141