Sorting

Class 28

Introduction

- two of the most fundamental concepts in computer science are, given an array of values:
  - search through the values to see if a specific value is present and, if so, where
  - sort the values into order

Introduction

- two of the most fundamental concepts in computer science are, given an array of values:
    - search through the values to see if a specific value is present and, if so, where
    - sort the values into order

- you must be able to understand and program several different algorithms for each of these tasks

Introduction

- two of the most fundamental concepts in computer science are, given an array of values:
  - search through the values to see if a specific value is present and, if so, where
  - sort the values into order
- you must be able to understand and program several different algorithms for each of these tasks
- in all of these slides, "array" is a generic term
- it means either an old-fashioned C-array or a C++ vector

Sorting

- the most-studied algorithm category in all of computer science
- literally hundreds of sorting algorithms have been invented, some very simple, some incredibly complex

Sorting

- the most-studied algorithm category in all of computer science
- literally hundreds of sorting algorithms have been invented, some very simple, some incredibly complex

- values may be sorted in ascending or descending order

Sorting

- the most-studied algorithm category in all of computer science
- literally hundreds of sorting algorithms have been invented, some very simple, some incredibly complex
- values may be sorted in ascending or descending order
- we will study two of the simplest
  - bubble sort
  - selection sort

Bubble Sort Pass 1

7	2	3	8	9	1

- bubble sort starts by comparing elements 0 and 1; if they are out of order (here, they are) they are swapped

Bubble Sort Pass 1

2	7	3	8	9	1

- bubble sort starts by comparing elements 0 and 1; if they are out of order (here, they are) they are swapped

Bubble Sort Pass 1

2	7	3	8	9	1

- bubble sort starts by comparing elements 0 and 1; if they are out of order (here, they are) they are swapped
- this is repeated with elements 1 and 2, which are also swapped

Bubble Sort Pass 1

2	3	7	8	9	1

- bubble sort starts by comparing elements 0 and 1; if they are out of order (here, they are) they are swapped
- this is repeated with elements 1 and 2, which are also swapped

Bubble Sort Pass 1

2	3	7	8	9	1

- bubble sort starts by comparing elements 0 and 1; if they are out of order (here, they are) they are swapped
- this is repeated with elements 1 and 2, which are also swapped
- elements 2 and 3 are compared; they do not need to be swapped

Bubble Sort Pass 1

2	3	7	8	9	1

- bubble sort starts by comparing elements 0 and 1; if they are out of order (here, they are) they are swapped
- this is repeated with elements 1 and 2, which are also swapped
- elements 2 and 3 are compared; they do not need to be swapped
- nor do elements 3 and 4

Bubble Sort Pass 1

2	3	7	8	9	1

- bubble sort starts by comparing elements 0 and 1; if they are out of order (here, they are) they are swapped
- this is repeated with elements 1 and 2, which are also swapped
- elements 2 and 3 are compared; they do not need to be swapped
- nor do elements 3 and 4
- finally, elements 4 and 5 are compared and swapped

Bubble Sort Pass 1

2	3	7	8	1	9

- bubble sort starts by comparing elements 0 and 1; if they are out of order (here, they are) they are swapped
- this is repeated with elements 1 and 2, which are also swapped
- elements 2 and 3 are compared; they do not need to be swapped
- nor do elements 3 and 4
- finally, elements 4 and 5 are compared and swapped

Bubble Sort Pass 1

2	3	7	8	1	9

- bubble sort starts by comparing elements 0 and 1; if they are out of order (here, they are) they are swapped
- this is repeated with elements 1 and 2, which are also swapped
- elements 2 and 3 are compared; they do not need to be swapped
- nor do elements 3 and 4
- finally, elements 4 and 5 are compared and swapped
- at the end of the first pass, the largest element, 9, has bubbled up to the end of the array

Bubble Sort Pass 2

2	3	7	8	1	9

- the second pass begins exactly like the first: compare elements 0 and 1 — no swap

Bubble Sort Pass 2

2	3	7	8	1	9

- the second pass begins exactly like the first: compare elements 0 and 1 — no swap
- compare elements 1 and 2 — no swap

Bubble Sort Pass 2

2	3	7	8	1	9

- the second pass begins exactly like the first: compare elements 0 and 1 — no swap
- compare elements 1 and 2 — no swap
- compare elements 2 and 3 — no swap

Bubble Sort Pass 2

2	3	7	8	1	9

- the second pass begins exactly like the first: compare elements 0 and 1 — no swap
- compare elements 1 and 2 — no swap
- compare elements 2 and 3 — no swap
- compare elements 3 and 4; swap

Bubble Sort Pass 2

2	3	7	1	8	9

- the second pass begins exactly like the first: compare elements 0 and 1 — no swap
- compare elements 1 and 2 — no swap
- compare elements 2 and 3 — no swap
- compare elements 3 and 4; swap

Bubble Sort Pass 2

2	3	7	1	8	9

- the second pass begins exactly like the first: compare elements 0 and 1 — no swap
- compare elements 1 and 2 — no swap
- compare elements 2 and 3 — no swap
- compare elements 3 and 4; swap
- the second pass stops at elements 3 and 4 because we know element 5 is in the correct place at the end of the first pass

Bubble Sort Pass 2

2	3	7	1	8	9

- the second pass begins exactly like the first: compare elements 0 and 1 — no swap
- compare elements 1 and 2 — no swap
- compare elements 2 and 3 — no swap
- compare elements 3 and 4; swap
- the second pass stops at elements 3 and 4 because we know element 5 is in the correct place at the end of the first pass
- after the second pass, both elements 4 and 5 have bubbled up to their proper place

Bubble Sort

- if there are 6 elements in the array, bubble sort takes 5 passes to complete
- when elements 1 through 5 have bubbled up to their correct place, then element 0 must also be correct

Bubble Sort Algorithm

```
void bubble_sort(vector<int>& array)
{
  for (size_t pass_indx = array.size() - 1; pass_indx > 0; pass_indx--)
  {
    for (size_t compare_indx = 0; compare_indx < pass_indx; compare_indx++)
    {
      if (array.at(compare_indx) > array.at(compare_indx + 1))
      {
        swap(array.at(compare_indx), array.at(compare_indx + 1));
      }
    }
  }
}
```

- Gaddis shows you the function swap
- and we have already seen the swap function back in class 20 on October 4

Bubble Sort Algorithm

```
void bubble_sort(vector<int>& array)
{
  for (size_t pass_indx = array.size() - 1; pass_indx > 0; pass_indx--)
  {
    for (size_t compare_indx = 0; compare_indx < pass_indx; compare_indx++)
    {
      if (array.at(compare_indx) > array.at(compare_indx + 1))
      {
        swap(array.at(compare_indx), array.at(compare_indx + 1));
      }
    }
  }
}
```

- Gaddis shows you the function swap
- and we have already seen the swap function back in class 20 on October 4
- but in fact swap is built in to C++11, so you can just use it without writing a function

Bubble Sort

Bubble Sort Pros

- very easy algorithm to understand
- easy algorithm to code correctly (just have to get the indices correct)
- pretty decent algorithm for an array that's already mostly sorted

Bubble Sort Cons

- a very inefficient algorithm in general
- typically performs many swaps to get each element into position

Bubble Sort

Bubble Sort Pros

- very easy algorithm to understand
- easy algorithm to code correctly (just have to get the indices correct)
- pretty decent algorithm for an array that's already mostly sorted

Bubble Sort Cons

- a very inefficient algorithm in general
- typically performs many swaps to get each element into position

- with a little more effort, we can do better

Selection Sort Pass 1

7	2	3	8	9	1

- selection sort also proceeds by passes
- in pass 1, select the smallest element in the entire array
- this is essentially identical to the “find smallest” algorithm you have coded in previous labs

Selection Sort Pass 1

1	2	3	8	9	7

- selection sort also proceeds by passes
- in pass 1, select the smallest element in the entire array
- this is essentially identical to the “find smallest” algorithm you have coded in previous labs
- the smallest element is found in position 5; swap it with the element in position 0

Selection Sort Pass 1

1	2	3	8	9	7

- selection sort also proceeds by passes
- in pass 1, select the smallest element in the entire array
- this is essentially identical to the "find smallest" algorithm you have coded in previous labs
- the smallest element is found in position 5; swap it with the element in position 0
- at the end of the first pass, the smallest element is in its correct place

Selection Sort Pass 2

1	2	3	8	9	7

- in pass 2, select the smallest element in positions 1 to 5

Selection Sort Pass 2

1	2	3	8	9	7

- in pass 2, select the smallest element in positions 1 to 5
- swap this element with the element in position 1 (here, value 2 is swapped with itself!)

Selection Sort Pass 2

1	2	3	8	9	7

- in pass 2, select the smallest element in positions 1 to 5
- swap this element with the element in position 1 (here, value 2 is swapped with itself!)
- now the first two elements are correct

Selection Sort Pass 2

1	2	3	8	9	7

- in pass 2, select the smallest element in positions 1 to 5
- swap this element with the element in position 1 (here, value 2 is swapped with itself!)
- now the first two elements are correct
- proceed this way through passes 3 through 6
- the same number of passes as bubble sort
- but each element is only swapped once into its final position

Selection Sort Algorithm

```
void selection_sort(vector<int>& array)
{
  size_t size = array.size();
  for (size_t select_indx = 0; select_indx < size - 1; select_indx++)
  {
    int smallest_value = array.at(select_indx);
    size_t smallest_indx = select_indx;
    for (size_t compare_indx = select_indx + 1; compare_indx < size;
        compare_indx++)
    {
      if (array.at(compare_indx) < smallest_value)
      {
        smallest_value = array.at(compare_indx);
        smallest_indx = compare_indx;
      }
    }
    swap(array.at(smallest_indx), array.at(select_indx));
  }
}
```

A Note on Swap

- some students will complain that it's silly to swap a value with itself

A Note on Swap

- some students will complain that it's silly to swap a value with itself
- but the built-in swap is smart enough to know that if the two array positions are the same, no swap is needed, and so it doesn't actually swap something with itself

Selection Sort

Selection Sort Pros

- easy algorithm to understand
- easy algorithm to code correctly (just have to get the indices correct)
- typically far fewer swaps than bubble sort

Selection Sort Cons

- not as efficient as more sophisticated sort algorithms that we will study later