Data Structures
Simple Sorts

Phil Tayco

Slide version 1.6

Oct. 19, 2021

Recall Maintaining Sorted Arrays

When to sort…

- We know that Search is significantly improved, when the data is in a sorted state
- To keep the data sorted, if we have to update, insert and delete, the 2 schools of thought are:
  - Keep the array sorted as you perform these functions
  - Don't keep it sorted as you perform these functions and only perform a sort when you need to (such as before a search takes place)
- When we kept the order as we performed the functions, we got the following performances

Sorted Arrays

Summary (Sorted using Binary Search):

Sorted Arrays (Binary Search)	Worst Case
Search	O(log n)
Insert	O(n)
Update	O(n)
Delete	O(n)

Simple Sorts

Sort when we need to

- If we do not keep the array sorted, we get the same performances of insert, update and delete for unsorted arrays
- This means insert performs at an ideal rate of O(1) while update and delete will still be O(n)
  - Note: Update/Delete is O(n) for unsorted and sorted arrays
- Since the data is then unsorted, search will be O(n)
- If we want to optimize on search, we have to sort the data before doing it
- The next key question is: How well does sorting an array perform?

Simple Sorts

Shift and swap

- As humans, we sort lists intuitively by moving and shifting elements naturally
- For computers, the steps in the algorithm have to be explicit
- We know from previous discussions that shifts are a costly deal so if we use them, we have to do so wisely
- If we don't shift, then the only way elements can be repositioned is if they switch places with another. We call this operation a "swap" and is a fundamental operation in sorting algorithms

Simple Sorts

```
void swap(int a, int b)
{
  int temp = numbers[a];
  numbers[a] = numbers[b];
  numbers[b] = temp;
}
```

Simple Sorts

The beginning 3

- Traditional data structures classes like this one introduce 3 basic sorting algorithms:
  - Bubble sort
  - Selection sort
  - Insertion sort
- As we investigate these algorithms, consider their performance in terms of comparisons as well as swap operations

Bubble Sort

Float them to the right

- The bubble sort is usually taught first as it is seen as the simplest to code and understand from it
- The algorithm:
  - Examine each pair of elements from left to right
  - For each pair, if the number on the left is greater than the number on the right, swap the two elements
  - Continue until you reach the end of the array
  - Repeat the same process again from left to right ignoring the last element in the array (this is because from the previous iteration, the number on the far right is in the correct position in the array)
  - Continue the pair examinations until you are down to only one pair left to examine

Bubble Sort

```
void bubbleSort()
{
   for (int i = numbers.length - 1; i > 0; i--)
       for (int j = 0; j < i; j++)
             if (numbers[j] > numbers[j+1])
                   swap(j, j+1);
}
```

Bubble Sort

Example

- 4 7 1 8 6 3 2 5 start

- 4 7 1 8 6 3 2 5 no swap
- 4 1 7 8 6 3 2 5 swap
- 4 1 7 8 6 3 2 5 no swap
- 4 1 7 6 8 3 2 5 swap
- 4 1 7 6 3 8 2 5 swap
- 4 1 7 6 3 2 8 5 swap
- 4 1 7 6 3 2 5 8 swap

Bubble Sort

Example

- 4 7 1 8 6 3 2 5 start
- 4 1 7 6 3 2 5 8 after 1st pass

- 1 4 7 6 3 2 5 8 swap
- 1 4 7 6 3 2 5 8 no swap
- 1 4 6 7 3 2 5 8 swap
- 1 4 6 3 7 2 5 8 swap
- 1 4 6 3 2 7 5 8 swap
- 1 4 6 3 2 5 7 8 swap

Bubble Sort

Example

- 4 7 1 8 6 3 2 5 start
- 4 1 7 6 3 2 5 8 after 1st pass
- 1 4 6 3 2 5 7 8 after 2nd pass

- 1 4 6 3 2 5 7 8 no swap
- 1 4 6 3 2 5 7 8 no swap
- 1 4 3 6 2 5 7 8 swap
- 1 4 3 2 6 5 7 8 swap
- 1 4 3 2 5 6 7 8 swap

Bubble Sort

Example

- 4 7 1 8 6 3 2 5 start
- 4 1 7 6 3 2 5 8 after 1st pass
- 1 4 3 2 5 6 7 8 after 2nd pass
- 1 4 3 2 5 6 7 8 after 3rd pass

- 1 4 3 2 5 6 7 8 no swap
- 1 3 4 2 5 6 7 8 swap
- 1 3 2 4 5 6 7 8 swap
- 1 3 2 4 5 6 7 8 no swap

Bubble Sort

Example

- 4 7 1 8 6 3 2 5 start
- 4 1 7 6 3 2 5 8 after 1st pass
- 1 4 3 2 5 6 7 8 after 2nd pass
- 1 3 2 4 5 6 7 8 after 3rd pass
- 1 3 2 4 5 6 7 8 after 4th pass

- 1 3 2 4 5 6 7 8 no swap
- 1 2 3 4 5 6 7 8 swap
- 1 2 3 4 5 6 7 8 no swap

Bubble Sort

Example

- 4 7 1 8 6 3 2 5 start
- 4 1 7 6 3 2 5 8 after 1st pass
- 1 4 3 2 5 6 7 8 after 2nd pass
- 1 3 2 4 5 6 7 8 after 3rd pass
- 1 2 3 4 5 6 7 8 after 4th pass
- 1 2 3 4 5 6 7 8 after 5th pass

- 1 2 3 4 5 6 7 8 no swap
- 1 2 3 4 5 6 7 8 no swap

Bubble Sort

Example

- 4 7 1 8 6 3 2 5 start
- 4 1 7 6 3 2 5 8 after 1st pass
- 1 4 3 2 5 6 7 8 after 2nd pass
- 1 3 2 4 5 6 7 8 after 3rd pass
- 1 2 3 4 5 6 7 8 after 4th pass
- 1 2 3 4 5 6 7 8 after 5th pass
- 1 2 3 4 5 6 7 8 after 6th pass

- 1 2 3 4 5 6 7 8 no swap

Bubble Sort

Example

- 4 7 1 8 6 3 2 5 start
- 4 1 7 6 3 2 5 8 after 1st pass
- 1 4 3 2 5 6 7 8 after 2nd pass
- 1 3 2 4 5 6 7 8 after 3rd pass
- 1 2 3 4 5 6 7 8 after 4th pass
- 1 2 3 4 5 6 7 8 after 5th pass
- 1 2 3 4 5 6 7 8 after 6th pass
- 1 2 3 4 5 6 7 8 last pass and done

Bubble Sort

Analysis

- The sort gets its name from the fact that the larger numbers “float up” to the right of the array with each iteration
- Using comparisons, the performance of the algorithm is a brutal $O(n^2)$ and applies to all cases from best to worst (what are the best and worst case scenarios?)
- Using swaps, the count is lower than comparisons and has a range for best to worst case, but the category is still $O(n^2)$ when we look at average and worst case scenarios
- Question: what pattern in the code is a good indicator of the $O(n^2)$ performance?

Bubble Sort

Possible Optimizations

- Each pass is scanning for any swaps needed to be performed
- If a particular pass resulted in no swaps, then the array is already sorted and further passes are not needed
- What if we added an additional check with each pass to see if any swaps occurred in the previous pass
- Does this give us any improvement to any of the scenarios? Does it make any of them worse?

Bubble Sort

```
void bubbleSort()
{
   boolean swapFlag = true;
   for (int i = numbers.length - 1; i > 0; i--)
   {
       if (swapFlag == false)
              break;
       swapFlag = false;
       for (int j = 0; j < i; j++)
              if (numbers[j] > numbers[j+1])
              {
                     swap(j, j+1);
                     swapFlag = true;
              }
   }
}
```

Bubble Sort

Improvements?

- In a situation where the numbers are randomly distributed unordered, the $O(n^2)$ is potentially made worse because of the additional swapFlag check (worst case n times!)
- Nice case is if the array is already or nearly sorted, as the number of passes through the loop can decrease to near O(n)
- Bubble sort performance ranges from O(n) to $O(n^2)$ with the comparisons and swaps leaning towards $O(n^2)$ as the data itself is more random

Selection Sort

Swap me into place

- The selection sort is a little closer to how we intuitively sort lists by finding the smallest (or largest) value in the list and performing a swap
- The algorithm (smallest value search):
  - Start at the first index
  - Go through the entire array looking for the smallest value in the array
  - After finding the smallest element, swap that with the first element
  - Repeat the sequence starting from the next element
  - Repeat the sequences until your starting element is the last element in the array

Selection Sort

```
void selectionSort()
{
   int lowestValue, lowestIndex;

   for (int i = 0; i < numbers.length - 1; i++)
   {
        lowestValue = numbers[i];
        lowestIndex = i;
        for (int j = i+1; j < numbers.length; j++)
               if (numbers[j] < lowestValue)
               {
                      lowestValue = numbers[j];
                      lowestIndex = j;
               }
        swap (i, lowestIndex);
   }
}
```

Selection Sort

Example

- 4 7 1 8 6 3 2 5 start
- 1 7 4 8 6 3 2 5 after 1st pass
- 1 2 4 8 6 3 7 5 after 2nd pass
- 1 2 3 8 6 4 7 5 after 3rd pass
- 1 2 3 4 6 8 7 5 after 4th pass
- 1 2 3 4 5 8 7 6 after 5th pass
- 1 2 3 4 5 6 7 8 after 6th pass
- 1 2 3 4 5 6 7 8 last pass and done

Selection Sort

Analysis

- The number of comparisons is exactly the same as the number in all situations and is in the $O(n^2)$ category
- Looking at swaps, though, the number is a consistent O(n)
- The code is a little more involved than the bubble sort, but shows a clear improvement in the number of swaps
- Consistency in performance can also be considered a good thing as it improves the reliability of the algorithm – you know exactly how the performance is in all situations
- Selection is preferred over Bubble when the swap operation needs to be considered. Examples include ordering cars in a parking lot or blocks between disk arrays
- Selection and Bubble are the same worst case for comparisons, though Bubble can be optimized if near sorted situations occur
- The comparison count with these two algorithms in the worst case is consistent because of the for loops. Perhaps the counts can be reduced by doing more with the inner loop

Insertion Sort

Shifting gears

- The insertion sort is even closer to how we intuitively sort lists by finding the small values, inserting them to where they should be at that moment and shifting elements along the way
- The algorithm:
  - Start at the value of the second index
  - Walk towards the first element, shifting along the way until you hit an element that is less than the current or you are at the beginning of the list
  - At that point, “insert” the current value into that spot
  - Repeat the sequence starting with the next element
  - Repeat the sequences until you hit the end of the list

Insertion Sort

```
void insertionSort()
{
   int i, j;
   for(i = 1; i < numbers.length; i++)
   {
        int temp = numbers[i];
        j = i;
        while(j > 0 && numbers[j - 1] >= temp)
        {
                numbers[j] = numbers[j-1];
                j--;
        }
        numbers[j] = temp;
   }
}
```

Insertion Sort

Example

- 4 7 1 8 6 3 2 5 start
- 4 7 1 8 6 3 2 5 7 inserted, no shift
- 1 4 7 8 6 3 2 5 1 inserted, shift 2
- 1 4 7 8 6 3 2 5 8 inserted, no shift
- 1 4 6 7 8 3 2 5 6 inserted, shift 2
- 1 3 4 6 7 8 2 5 3 inserted, shift 4
- 1 2 3 4 6 7 8 5 2 inserted, shift 5
- 1 2 3 4 5 6 7 8 5 inserted, shift 3

Insertion Sort

Analysis

- The insertion process is basically looking for the smallest element in the subset list and placing into the correct spot, shifting elements along the way
- As the subset list gets larger, all elements to the left of the new element are effectively sorted. By inserting the next element in the right spot, you are sorting the subset list each time
- The inner loop is essentially performing a shift, but the shift is “intelligent” in that as seen with the other loops, the inner loop is necessary to perform anyway. We use that fact to perform “mini-sorts” with these shifts

Insertion Sort

Analysis

- The number of comparisons still falls in the $O(n^2)$ category in the worst case. However, the algorithm itself is optimized for near sorted situations (similar to Bubble with the swapFlag) and can lean towards O(n)
- Nearly sorted array situations can be a common occurrence (such as after adding a few elements to an already sorted list)
- Although no swaps being performed, the total number of assignment operations taking place during the inner loop leads to similar counts. The reduced comparison count on average is a strong win for insertion sort

Simple Sorts

Summary:

	Comparisons	**Notes**
Bubble	$O(n^2)$ (worst case)	Optimized range is similar to Insertion
Selection	$O(n^2)$ (all cases)	Swaps are O(n)
Insertion	$O(n^2)$ (worst case)	Range is O(n) to (n^2) Excellent for partially sorted lists. Better than Bubble

Simple Sorts

How are we doing?

- Looking back at the question of when to sort for an array, if we maintain the order as data is inserted, updated and deleted, worst case performance is essentially O(n)
- Sorting the data when it is in any state ranges from O(n) to $O(n^2)$ depending on the algorithm. The more random the data is, the more it tends to $O(n^2)$
- It appears maintaining the order as data is updated is preferred. If sorting the data as a whole has to occur, it should be infrequent
- Understanding the need to sort the data is because of the need to search. If the context of your program suggests searching is not required, keeping the data unsorted (in perhaps a linked list) may be preferred
- Advanced sorts is where we are headed next to see if we get any significant improvement beyond n^2 – to understand them requires learning another style of coding…