Programming Design and Implementation

Lecture 8: Searching and Sorting

Updated: 18th May, 2020

Mark Upston
Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

“Zero” Marks

- A student who does any of the following in a submitted, assessable answer will receive heavy penalties, up to and including **zero** marks for that question:
  - Uses **`continue`**
  - Uses **`break`** in any other place than a **`switch`** statement
  - Uses **`goto`**
  - Has more than one **`return`** statement in a method
  - Has a **`return`** statement in a method anywhere but the last statement of the method
  - Uses **`System.exit()`** anywhere but the last statement of the **`main()`** method
  - Uses global variables for anything other than class fields
  - Uses a ternary operator
- Note: similar efforts in pseudo code will also receive zero marks

Introduction to Time Complexity Analysis

- In computers, time in seconds is not a useful measure of an algorithm since faster hardware can reduce the time
- Instead, we need to talk about how many steps are needed
    - Which is independent of hardware speed, so is a better 'absolute' measure of speed
    - And where 'steps' really is CPU instructions
- Unfortunately, we can **never** know exactly how many CPU instructions something takes
    - Different CPUs have different instruction sets and are faster/slower with different instructions vs other CPUs

Searching a Sorted List

- Searching a list for an item (or items) is common
  - Naïve search: go sequentially through the list
    - Also called a 'linear search'
    - As simple as it gets, but still pretty effective

```
SUBMODULE: linearSearch
IMPORT: array (ARRAY OF X), searchTgt (X)
EXPORT: matchIdx (Integer)
ASSERTION:
ALGORITHM:
    ii := 0
    matchIdx := -1
    WHILE (ii < array.length) AND (matchIdx == -1)
        IF(array[ii] EQUALS searchTgt)
            matchIdx := ii
        END IF
        ii := ii + 1
    END WHILE
END linearSearch
```

Binary Search

- Linear search is okay, but nothing to brag about
- A faster alternative: Binary Search
  - Takes advantage of (and needs) sorted data to 'jump around'
    Step 1: Set an upper and lower bound on the search location
    - The target is known to exist within these bounds
    - Start off with the full range and refine the search from there

    Step 2: Check the value halfway between these bounds and use this to update the bounds
    - If the halfway value is still too low, it becomes the new lower bound
    - If it is too high, it becomes the new upper bound

    Repeat until you hit the target value being searched for

Binary Search - Strategy

- Analogy: Guessing a number between 0..100
  - Example target: **`85`**; Initial bounds: **`0..100`**

  – Initial Guess: $(0+100)/2 = 50$
  Response: "No, higher" | New bound: **`50..100`**
  – Guess: $(50+100)/2 = 75$
  Response: "No, higher" | New bound: **`75..100`**
  – Guess: $(75+100)/2 = 88$
  Response: "No, lower" | New bound: **`75..88`**
  – Guess: $(75+88)/2 = 82$
  Response: "No, higher" | New bound: **`82..88`**
  – Guess: $(82+88)/2 = 85$
  Response: "Got it!"

Binary Search – Algorithm

- Replace 'guessing the number' with 'guessing the index in a sorted list' and you have binary search

```
SUBMODULE: binarySearch
IMPORT: sortedArray (ARRAY OF X), searchTgt (X)
EXPORT: matchIdx (Integer)
ASSERTION: matchIdx will be -1 if searchTgt not found
ALGORITHM:
    matchIdx := -1
    lowerBd := 0
    upperBd := sortedArray.length
    WHILE (NOT found) AND (lowerBd <= upperBd)
        chkIdx = (lowerBd + upperBd) / 2
        IF(sortedArray[chkIdx] < searchTgt)
            lowerBd := chkIdx + 1
        ELSE IF(sortedArray[chkIdx] > searchTgt)
            upperBd := chkIdx - 1
        ELSE
            matchIdx := chkIdx
            found := TRUE
        END IF
    END WHILE
END binarySearch
```

Binary Search – Limitations

- Requires data to be in sorted order
  - Needs sorting so that the upper and lower bound indexes are guaranteed to contain the target
  - Thus, although binary searches are very fast, it requires a complex pre-step to sort the data first.
    - Hence one-off searches are better done with a Linear Search
    - Only for repeated searching, is it worth the sorting pre-step

Sorting: Why Sort?

- For clear presentation of data
  - We often need to present data in an organised manner to a person so that they can make sense of it
  - Sorting is usually a good way to organise data
    - Just imagine using a randomly-ordered phone book!
- To facilitate efficient processing:
  - Selecting a range is simple if the data is in a sorted list
    - e.g., in a supermarket database, finding all transactions between 01/01/2010 and 31/01/2010 is easy if the data is sorted by date
  - Sorting also allows us to search for an item quickly
    - Analogy: finding a name in a phone book; you go directly to first letter, then second, etc

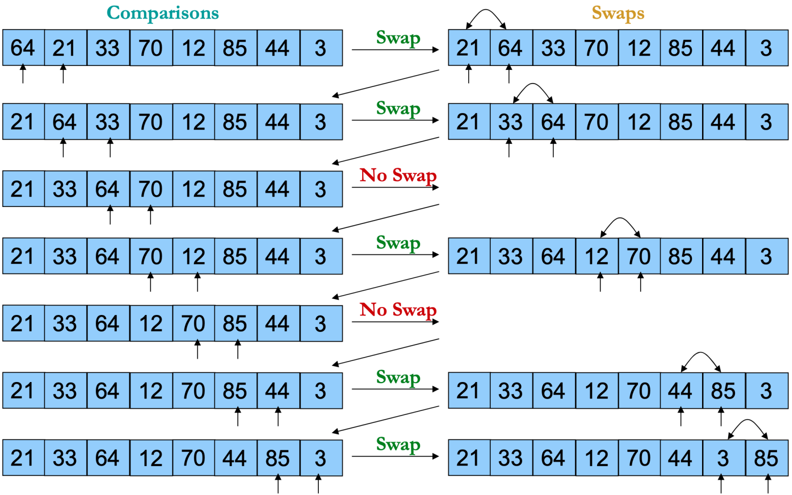
Sorting Algorithms

- We'll start by examining the sort algorithms
  - These all have two loops (nested). Each iterating $\sim$N times
    - Approximately running N^2 times
  - The inner loop defines a single pass through the data, performing checks and swaps to improve sorted order
  - The outer loop forces multiple inner-loop passes until the array is completely sorted
    - Exactly how many passes depends on the algorithm and on the initial state of the array (i.e., how unsorted it is)

Bubble Sort – Strategy

- Incrementally sorts the array by comparing adjacent pairs and swapping them if they are not in order
  - One pass through the array will only improve ordering
  - Need multiple passes P (up to N) to fully sort the array
  - Each pass will 'bubble up' the largest value to the end
- Some 'optimisations' can be done:
  - Can stop sorting when no swaps are needed in a pass
    - Detects sorted: if all adjacent pairs are in order, array is sorted
  - Don't need to check the last P values in the array
    - The previous P passes have put the largest P values at the end

Bubble Sort – Example (one pass)



Sorting Visualisations

- This is an old one, but a good one:
  - Visualization of 24 Sorting Algorithms In 2 Minutes
- We'll also use a simulation to help understand the sorts:
  - VisuAlgo
- This plays through animated examples of many algorithms:
  - Data Structure Visualizations

Bubble Sort Algorithm – Basic Version

```
SUBMODULE: bubbleSort
IMPORT: array (ARRAY OF X)
EXPORT: array (ARRAY OF X)
ASSERTION: array will be sorted using simple Bubble Sort
ALGORITHM:
    FOR pass := 0 TO (array.length - 1) - 1 INC BY 1
        FOR ii := 0 TO (array.length - 1 - pass) - 1 INC BY 1
            IF (array[ii] > array[ii+1])
                temp := array[ii]
                array[ii] := array[ii+1]
                array[ii+1] := temp
            END IF
        END FOR
    END FOR
END bubbleSort
```

Bubble Sort Algorithm – Optimised

```
SUBMODULE: bubbleSort
IMPORT: array (ARRAY OF X)
EXPORT: array (ARRAY OF X)
ASSERTION: array will be sorted using optimised Bubble Sort
ALGORITHM:
    pass := 0
    DO
        sorted := TRUE
        FOR ii := 0 TO (array.length - 1 - pass) - 1 INC BY 1
            IF (array[ii] > array[ii+1])
                temp := array[ii]
                array[ii] := array[ii+1]
                array[ii+1] := temp
                sorted := FALSE
            END IF
        END FOR
        pass := pass + 1
    WHILE (NOT sorted)
END bubbleSort
```

Bubble Sort – Discussion

- Problem: Lots of swaps as well as comparisons
- Good example of why we don't focus on best case scenario
  - Best case looks great: only one pass (N times)
    - BUT: 'almost-sorted' has a very specific meaning here!
    - In particular, it requires that small elements start not very far from their final sorted position. Unlikely except if already-sorted
- Average/worst cases are $\sim N^2$ iterations are even worse than they look due to the sheer amount of swaps involved
  - The other algorithms manage $\sim N^2$ iterations but with far fewer swaps

Insertion Sort – Strategy

- Inspired from the idea of adding items to an array in sorted order
  - Every time a new item is added, insert it in sorted position
  - Great for reading in items from a file, into sorted order.
- Can also be applied to sorting an existing array
  - Maintain a marker (index) and insertion-sort the element at the marker into the items to the left of the marker
    - i.e., take the next element and insert it in sorted order into the sub-array that precedes the element
    - Start the marker at index 1 and move it up by one after each inserted element. Then elements before marker will be sorted
    - Searches for the insert position backwards so that we can take advantage of semi- sorted arrays

Insertion Sort Algorithm

```
SUBMODULE: insertionSort
IMPORT: array (ARRAY OF X)
EXPORT: array (ARRAY OF X)
ASSERTION: array will be sorted using Insertion Sort
ALGORITHM:
    FOR nn := 1 TO array.length - 1 INC BY 1
        ii := nn
        WHILE (ii > 0) AND (array[ii-1] > array[ii])
            temp := array[ii]
            array[ii] := array[ii-1]
            array[ii-1] := temp
            ii := ii - 1
        END WHILE
    END FOR
END insertionSort
```

Insertion Sort Algorithm – Alternative

```
SUBMODULE: insertionSort
IMPORT: array (ARRAY OF X)
EXPORT: array (ARRAY OF X)
ASSERTION: array will be sorted using Insertion Sort
ALGORITHM:
    FOR nn := 1 TO array.length - 1 INC BY 1
        ii := nn
        temp := array[ii]
        WHILE (ii > 0) AND (array[ii-1] > temp)
            array[ii] := array[ii-1]
            ii := ii - 1
        END WHILE
        array[ii] := temp
    END FOR
END insertionSort
```

- Only requires half the work inside the loop!

Insertion Sort – Discussion

- Although it looks superficially like Bubble Sort in its number of iterations, insertion sort is generally superior
  - Better at being efficient with semi-sorted data
    - Elements are placed in their sorted position directly
    - ... whereas Bubble Sort is swapping things around all the time
- But Insertion Sort still does a lot of swaps per pass
  - In fact, the same number of swaps as compares
  - ... because once the insertion point is found, the pass is complete

Example of Insertion Sort

0	1	2	3	4	5
5	2	4	6	1	3

$\longrightarrow$

0	1	2	3	4	5
2	5	4	6	1	3

0	1	2	3	4	5
2	5	4	6	1	3

$\longrightarrow$

0	1	2	3	4	5
2	4	5	6	1	3

0	1	2	3	4	5
2	4	5	6	1	3

$\longrightarrow$

0	1	2	3	4	5
2	4	5	6	1	3

0	1	2	3	4	5
2	4	5	6	1	3

$\longrightarrow$

0	1	2	3	4	5
2	4	5	1	6	3

Example of Insertion Sort (2)

0	1	2	3	4	5		0	1	2	3	4	5
2	4	5	1	6	3	⟶	2	4	1	5	6	3

0	1	2	3	4	5		0	1	2	3	4	5
2	4	1	5	6	3	⟶	2	1	4	5	6	3

0	1	2	3	4	5		0	1	2	3	4	5
2	1	4	5	6	3	⟶	1	2	4	5	6	3

0	1	2	3	4	5		0	1	2	3	4	5
1	2	4	5	6	3	⟶	1	2	4	5	3	6

Example of Insertion Sort (3)

0	1	2	3	4	5
1	2	4	5	3	6

→

0	1	2	3	4	5
1	2	4	3	5	6

0	1	2	3	4	5
1	2	4	3	5	6

→

0	1	2	3	4	5
1	2	3	4	5	6

Selection Sort – Strategy

- Select smallest item and swap it with the first item
  - Requires one full pass through the array to find smallest
- Repeat with the second-smallest item, swapping it with the second item
- Repeat until all items have been selected and placed in their sorted position
  - i.e., needs N-1 passes (N^{th} pass isn't needed since its job will be done automatically by the previous passes)

Selection Sort Algorithm

```
SUBMODULE: selectionSort
IMPORT: array (ARRAY OF X)
EXPORT: array (ARRAY OF X)
ASSERTION: array will be sorted using Selection Sort
ALGORITHM:
    FOR nn := 0 TO array.length - 1 INC BY 1
        minIdx := nn
        FOR jj := nn+1 TO array.length - 1 INC BY 1
            IF (array[jj] < array[minIdx])
                minIdx := jj
            END IF
        END FOR
        temp := array[minIdx]
        array[minIdx] := array[nn]
        array[nn] := temp
    END FOR
END selectionSort
```

Selection Sort – Discussion

- Only one swap per pass
  - Each selected item is placed directly in the position it will ultimately occupy and never swapped again
- But always do N-P comparisons per pass P
  - With N passes, selection sort is thus $\sim N^2$ iterations in all cases (best/worst/average)
- Summary: many passes with minimal work per pass
  - Consistent speed: bit faster on avg than other $\sim N^2$ sorts
  - But does not take advantage of semi-sorted data like Insertion Sort or (to a lesser extent) Bubble Sort

Bubble Sort

- ☑ Simple to implement
- ☑ Can finish early (i.e., faster) if data is almost-sorted
  - But 'almost-sorted' has a very specific meaning here!
  - In particular, it requires that smaller elements start not very far from their final sorted position
- ✗ Lots of work per pass – constantly swapping
- ✗ Very slow on reverse-ordered data
  - Ends up swapping every element on each pass

Insertion sort

- ☑ Very fast with almost-sorted data (minimal swaps/compares)
  - Plus, performance degrades 'gently' from best case
    - e.g., single out-of-place elements don't destroy efficiency
  - Hence effective with most arrays that are partially ordered
- ☑ Stable sort (discussed in later slides)
- ✗ Conceptually trickier than other $\sim N^2$ sorts
- ✗ Lots of swaps – need to shuffle larger elements up
- ✗ Very slow on reverse-ordered data

Selection Sort

- ☑ Simple to implement
- ☑ Minimal work per pass – only one swap
  - Thus generally faster in the average case than the others
- ☑ Best case = average case = worst case
  - i.e., always has to perform the full N-1 passes, and each pass always involves the same amount of work no matter what the unsorted array looks like
  - But this consistency could be considered a positive too, depending on the situation
- ☑ In-place (discussed next)
- ✗ Unstable sort (we'll discuss what this means next)

Other Factors in Sorting

- Speed is not the only consideration in algorithms
- Extra memory use (memory overhead) is another
  - In sorting, extra memory is often needed for temporary storage to help organise the data
    - In-place vs not in-place sorting
- And different problem domains have their own particular issues
  - For sorting, one of them is whether two identical values will stay in the same relative order after sorting
    - Stable vs unstable sorting

In-Place Sorting

- All the $\sim N^2$ sorting algorithms we've looked at didn't need much temporary storage
  - Only enough to handle the swap, i.e., one temp element
- This makes them '**in-place**' sorting algorithms
  - i.e., they sort the array in the same place as the array, without needing to copy chunks to another temp array
- Some sorting algorithms are not in-place
  - For very large data sets that fill up almost all RAM, the extra space of non-in- place sorting can be a problem!

Stable vs Unstable Sorting

- If you look closely at the $\sim N^2$ sorting algorithms, you'll see that the compares were done carefully
  - i.e., we chose very deliberately whether to do $>$ or $>=$
  - Goal: choose the compare that will ensure that duplicate values will remain in the same order with referece to each other
- Why bother? So what if identical items get swapped?
  - Ah, but we rarely sort just a list of single values
  - Instead, we usually sort rows of data, by choosing one of the columns to use as the sorting key (i.e., the comparer)
    - The other columns get 'dragged along' with the sort column
    - These other columns won't be identical, hence we'd like to preserve the relative order of rows with identical sort keys

Stable vs Unstable Sorting

- Thus a 'stable sort' is one which guarantees that identically-valued sort keys will keep their ordering
  - And conversely, an unstable sort is one that does not
- Bubble sort and Insertion sort are both stable
  - But only because we carefully selected > or >=
    - (they all ended up as >, but that's not a blanket rule!)
- Selection sort is unstable
  - Consider selection sort on the list of values 8, 8, 3
  - The first 8 will swap with the 3, then no more swaps
    - This puts the first 8 after the second 8 – unstable
  - No matter how you code it, Selection Sort cannot guarantee stability for all possible arrays

Next Week

- The next Lecture will address the following:
  - Number Systems