Murdoch
UNIVERSITY

Data Structures and Abstractions

Searching & Sorting & STL Sort

Lecture 10

Testing

- When testing searching and merging algorithms, it is important to check boundary and unusual conditions:
- In other words test for containers with **[1]**:
  - 0 elements;
  - 1 element;
  - 2 elements;
  - 3 elements;
  - a large number of odd elements;
  - a large number of even elements;
  - The Cyclomatic Complexity of your algorithm can be used to guide your test cases. **[2]**

Linear Search

- Linear searches involve starting at the beginning, then checking each element in the container until we find the right one.
- In other words, a brute force approach.
- This is sure but slow: its average complexity is O(n). [1]
- This code is usually put inside a Find() or Search() routine.
- It is the only search available to linked lists and unsorted arrays, and is therefore the search used for the STL vector and list.

Linear Search Algorithm

```
Boolean Find (DataClass target, Address targetPosition) [code in
textbook]

    Boolean found
    Set found to false
    Start at the beginning of the container
    WHILE not at the end of container AND found is false
        IF the current element is the target
            targetPosition = Address (index of the current
                                     element}
            found = true
        ENDIF
        set current to next element
    ENDWHILE
    Return found

END FIND
```

Binary Search

- A faster search than linear search exists for **sorted**, direct access containers such as sets and maps.
- This is *binary* search, where the search space is halved after each guess.
- The "Guess a number between 1 and 100" game played by children is a binary search.
- It is a divide and conquer strategy.
- The order of complexity is O(log(n) + 1) for the number of iterations.
- See diagrams explaining this in the textbook – section on binary search, Chapter on Searching and Sorting Algorithms.
  - Go through the worked examples in this chapter found in the section on Asymptotic Notation: Big-O Notation. **[1]**

Murdoch UNIVERSITY

Iterative Binary Search Algorithm

```
Find (DataClass target, integer targetIndex) [code in textbook, data must be sorted]

    integer bottomIndex, middleIndex, topIndex  [1]
    boolean targetFound
    targetFound = false
    bottomIndex = 0
    topIndex = arraySize-1
    WHILE topIndex >= bottomIndex AND targetFound = false
        middleIndex = (topIndex + bottomIndex)/2
        IF target = value at middleIndex
          targetIndex = middleIndex
          targetFound = true
        ELSEIF target < value at middleIndex
          topIndex = middleIndex-1 // no point searching above
        ELSEIF target > value at middleIndex
          bottomIndex = middleIndex+1 // no point searching below
        ENDIF
    ENDWHILE

END Find
```

Iteration versus Recursion

- Anything that can be done with recursion can be done with iteration.
- Anything that can be done with iteration can be done with recursion.
- Advantages of Iteration:
  - Often easier to understand
  - Uses less memory
- Advantages of Recursion: **[1]**
  - Sometimes much easier to understand
  - Often simpler to code
  - Reduces code complexity

Recursive Binary Search Algorithm [1]

```
Find (DataClass target, integer targetIndex) : boolean
          Set targetIndex to -1
          return Find (target, 0, arraySize-1, targetIndex)
End Find

Find (DataClass target, integer bottomIndex, integer topIndex,
           integer targetIndex) : boolean // the overloaded version
          Boolean found
          Set found to false
          Integer middleIndex
          middleIndex = (topIndex + bottomIndex)/2
          IF target = array[middleIndex]
                    targetIndex = middleIndex
                    found = true        // line A
          ELSEIF topIndex <= bottomIndex
                    found = false
          ELSEIF target < array[middleIndex]
                    Find (target, bottomIndex, middleIndex-1, targetIndex)//Line B
          ELSEIF target > array[middleIndex]
                    Find (target, middleIndex+1, topIndex, targetIndex)
          ENDIF
          Return found [2]       // Line C
End Find
```

Merging Sorted Containers

- When we looked at Sets in a previous lecture, we looked at algorithms for subset, difference, union and intersection.
- They all operate in O(n) time.
- They were all very similar.
- This is because they were all variations of the standard *merge* algorithm for sorted containers.
- The merge algorithm is also important for *merge sort* which is the best (only) sort to use for very large amounts of data stored on disk.
- Note that the STL `<algorithm>` class contains a merge algorithm that works on **sorted** containers.

```
Merge(container1, container2, newContainer) [1]

    datum1 = first element in container1
    datum2 = first element in container2
    WHILE there are elements in both container1 and container2
        IF datum1 < datum2
            Put datum1 in newContainer
            datum1 = next element in container1
        ELSEIF datum2 < datum1
            Put datum2 in newContainer
            datum2 = next element in container2
        ELSE
            Put datum1 in newContainer
            Put datum2 in newContainer  // duplicates are being kept
            datum1 = next element in container1
            datum2 = next element in container2
        ENDIF
    ENDWHILE

    WHILE there are elements in container1
        Put datum1 in newContainer
        datum1 = next element in container1
    ENDWHILE

    WHILE there are elements in container2
        Put datum2 in newContainer
        datum2 = next element in container2
    ENDWHILE

End Merge
```

Categorisation of Sorting Algorithms

- Categorising sorting algorithms allows decisions to be made on the best sort to use in a particular situation.
- Algorithms are categorised based on:
  - what is actually moved (direct or indirect);
  - where the data is stored during the process (internal or external);
  - whether prior order is maintained (stable vs unstable);
  - how the sort progresses;
  - how many *comparisons* are made on average and in the worst case;
  - how many *moves* are made on average and in the worst case.

Direct vs Indirect

- Direct sorting involves moving the elements themselves.
  For example when sorting an array

50	20	10	60	10

It becomes

10	10	20	50	60

- Indirect sorting involves moving objects that designate the elements (also called address table sorting). This is particularly common where the actual data is stored on disk or in a database. For example, if sorting an array:

50	20	10	60	10

we do not sort the data, but instead set up an array of the addresses:

0	1	2	3	4

and sort them based on the data to which they refer:

2	4	1	0	3

Internal vs External

- Internal: the data is stored in RAM.
- External: the data is stored on secondary storage (hard drive, tape, floppy disk etc).
- There are two external sorts: natural merge and polyphase. The latter is very complicated and it is usually used for large files. [1]
  - We wouldn’t be looking at polyphase sort in this unit – read out of interest.

Stable vs Unstable

- Stable sorts preserve the prior order of elements where the new order has equal keys.
- For example, if you have sorted on name and then sort on address, people with the same address would still be sorted on name.
- On the whole stable sorts are slower.

Type of Progression

- *Insertion*: examine one element at a time and insert it into the structure in the proper order relative to all previously processed elements.
- *Exchange*: as long as there are still elements out of order, select two elements and exchange them if they are in the wrong order.
- *Selection*: as long as there are elements to be processed, find the next largest (or smallest) element and set it aside.
- *Enumeration*: each element is compared to all others and placed accordingly. [1]
- *Special Purpose*: a sort implemented for a particular one-off situation.

Number of Comparisons

Type	Name	Average O	Worst Case O
Insertion	Straight Insertion	n^2	n^2
	Binary Insertion	n log n	n log n
	Shell*	$n^{1.25}$	-
Exchange	Bubble	n^2	n^2
	Shaker	n^2	n^2
	Quicksort	n log n	n^2
	Merge	n log n	n log n
Selection	Straight Selection	n^2	n^2
	Heap	n log n	n log n

* Based on empirical evidence only.

Number of Comparisons [1]

Type	Name	Average O	Worst Case O
Insertion	Straight Insertion	n^2	n^2
	Binary Insertion	n log n	n log n
	Shell*	$n^{1.25}$	-
Exchange	Bubble	n^2	n^2
	Shaker	n^2	n^2
	Quicksort	n log n	n^2
	Merge	n log n	n log n
Selection	Straight Selection	n^2	n^2
	Heap	n log n	n log n

* Based on empirical evidence.

Number of Moves

Type	Name	Average O	Worst Case O
Insertion	Straight Insertion	n^2	n^2
	Binary Insertion	n^2	n^2
	Shell*	$n^{1.25}$	-
Exchange	Bubble	n^2	n^2
	Shaker	n^2	n^2
	Quicksort	n log n	n^2
	Merge	n log n	n^2
Selection	Straight Selection	n log n	n^2
	Heap	n log n	n log n

* Based on empirical evidence only as there doesn't appear to be algorithm analysis for this algorithm

Number of Moves

Type	Name	Average O	Worst Case O
Insertion	Straight Insertion	n^2	n^2
	Binary Insertion	n^2	n^2
	Shell*	$n^{1.25}$	-
Exchange	Bubble	n^2	n^2
	Shaker	n^2	n^2
	Quicksort	n log n	n^2
	Merge	n log n	n^2
Selection	Straight Selection	n log n	n^2
	Heap	n log n	n log n

* Based on empirical evidence only

Algorithm Choice

- Looking at the tables, ‘clearly’ heap sort is the fastest, followed by mergesort and quicksort.
- So why is quicksort the algorithm used by spreadsheets, the STL, in C etc??
- There can be several reasons:
  - The first is that quicksort is an *internal* sort and the other two are *external* sorts. Therefore it requires less I/O, but there are versions of merge sort which try to cut down on I/O.
  - Obtaining and releasing memory is time consuming.
  - The next reason hidden in the use of big O notation. When running quicksort, merge and heap sort on my PC, I found that although they are all O(n log n) for random data, quicksort ran twice as fast as heap sort and almost 5 times faster than merge sort!
  - There are lots of very complicated ways to optimise quicksort.
  - On the flip side, merge sort is very suited to parallel programming.

Readings

- Textbook Chapter Searching and Sorting Algorithms.
- Reference book, Introduction to Algorithms. For further study, see part of the book called Sorting and Order Statistics. It contains a number of chapters on sorting.

Bubble sort, Merge sort, Heap sort and Quicksort

Bubble Sort

- Bubble sort is the most commonly coded of the simple sorts.
- It is a stable exchange sort.
- Whilst not particularly fast—O(n^2)—it is very simple to code and easy to understand.
- For anything less than 1000 items, bubble sort is fine.
- Its name derives from the fact that large numbers ‘bubble’ to the ‘top’ of the container.

Bubble Sort Algorithm

```
ArrayBubbleSort

    integer target, lastSwap
    boolean swapDone, sortDone

    Initialise lastSwap to 0
    Initialise sortDone to false

    IF array size > 1
        target = size-1
        WHILE not sortDone
            swapDone = false
            FOR index = 0 to target-1
                IF element[index] > element[index+1]
                 Swap them
                 lastSwap = index
                 swapDone = true
                ENDIF
            ENDFOR
            sortDone = not swapDone
            target = lastSwap
        ENDWHILE
    ENDIF

END BubbleSort
```

Merge Sort

- Merge sort uses the divide and conquer algorithmic strategy.
- It has complexity O(nlog n) for all cases.
- It is a simple merge to implement.
- It is most easily implemented using recursion.
- It is an efficient sort to implement for a large amount of data on disk (that does not fit into RAM).

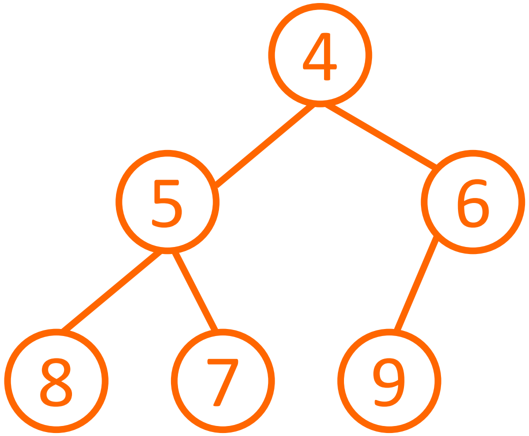
Merge Sort Algorithm

```
•   MergeSort

•       IF there are more than two elements in the container
•           Divide the container into two
•           Merge Sort the first part // call again
•           Merge Sort the second part // call again
•           Merge the two sorted parts into a temp file or array
•           Put merged temp file/array back into array being sorted
•       ELSE IF two elements in the container
•           Swap them if necessary
•       ENDIF

•   END MergeSort
```

The Actual Heap

- Clearly such a structure can be used to sort data.
- However, in actual fact, the data structure used is simply another array.
- This is because we end up doing a lot of data swapping in a heap, which is difficult to code in an actual tree.
- Also it turns out that in an array, the parent-child relationships is mathematical, making swaps particularly easy.

Abstract View vs Actual View



4	5	6	8	7	9

Heap Sort Algorithm

- Heap sort is an unstable selection sort.
- It utilises a greedy algorithmic technique.
- It has complexity O(nlog n).
- But is more complicated to code than a merge sort.

Heap Sort Algorithm

```
•   HeapSort

•       FOR each member of the array
•           Place it at the bottom end of the heap
•           WHILE it is smaller than the parent
•               Exchange it with the parent
•           ENDWHILE
•       ENDFOR

•       index = 0
•       WHILE the heap is not empty
•           Put the top of the heap at index in the array
•           Increment index
•           Delete the top of the heap and rearrange
•       ENDWHILE

•   END HeapSort
```

Put on the heap

Take off the heap

Quicksort

- Quicksort: the name says it all!
- It is the fastest algorithm that uses no extra space.
- It can also be optimised to be very, very fast indeed.
- It is O(nlog n) on average and O(n^2) in the worst case.
- *But* it is difficult to code and difficult to understand unless you actually try it.

Quicksort Algorithm

```
QuickSort
    Quicksort (0, size, array);
END QuickSort

QuickSort (low, high, array)
    IF low < high AND high-low >= 2
        integer pivotIndex
        Split (low, high, array, pivotIndex) // sort is done here
        QuickSort (low, pivotIndex-1, array)
        QuickSort (pivotIndex+1, high, array)
    ELSEIF high-low == 2
        If array[high] < array[low]
            Swap them
        ENDIF
    ENDIF
END QuickSort
```

```
Split (low, high, array, pivotIndex)
    pvalue = array[low]
    integer index1 = low
    integer index2 = high
    WHILE (index1 < index2)
        WHILE (array[index1] <= pvalue && index1 < index2)
          index1++;
        ENDWHILE
        WHILE (array[index2] > pvalue && index2 > index1)
          index2--;
        ENDWHILE
        IF (index1 < index2)
          Swap values at index1 and index2
        ENDIF
    ENDWHILE
    Set pivotIndex to index2-1
    Swap values at low and pivotIndex
End Split
```

Look for a value higher than the pivot value

Look for a value lower than the pivot value

If found, swap them

Now put the pivot value between them

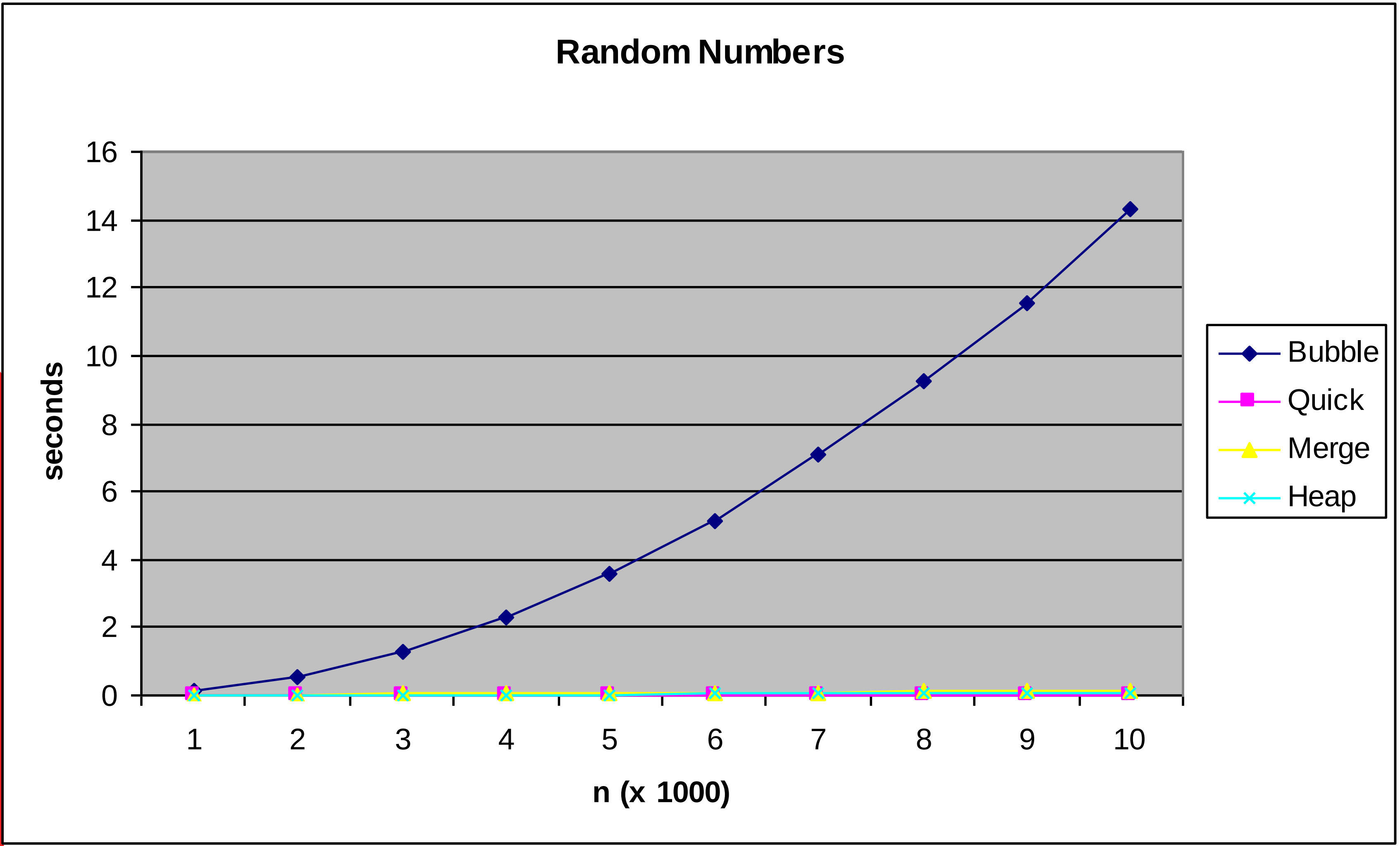
Readings

- Textbook Chapter Searching and sorting Algorithms. Diagrams in the textbook also explain step by step.

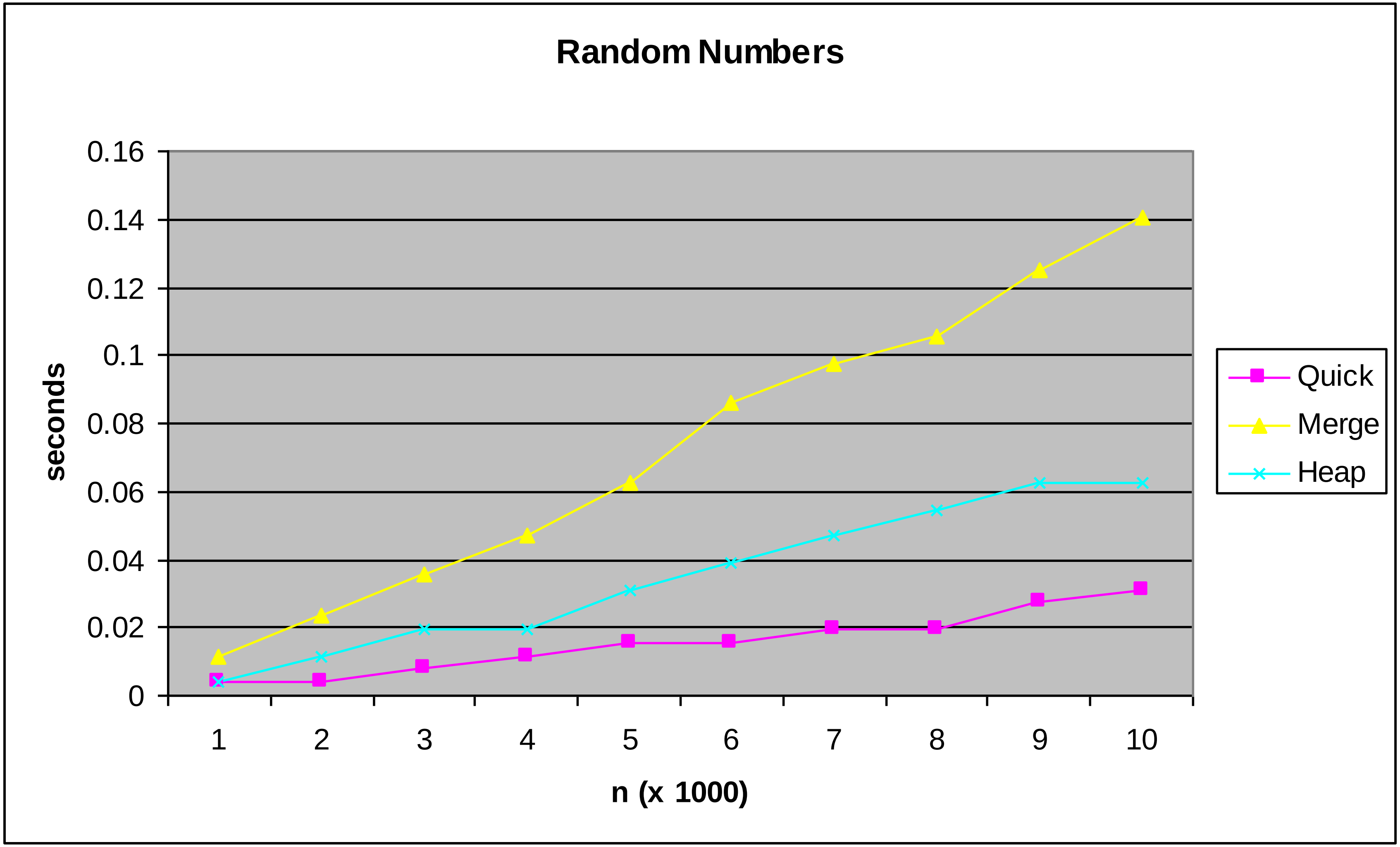
- Reference book, Introduction to Algorithms. For further study, see part of the book called Sorting and Order Statistics. It contains a number of chapters on sorting.

Empirical Comparisons, and the STL Sorts

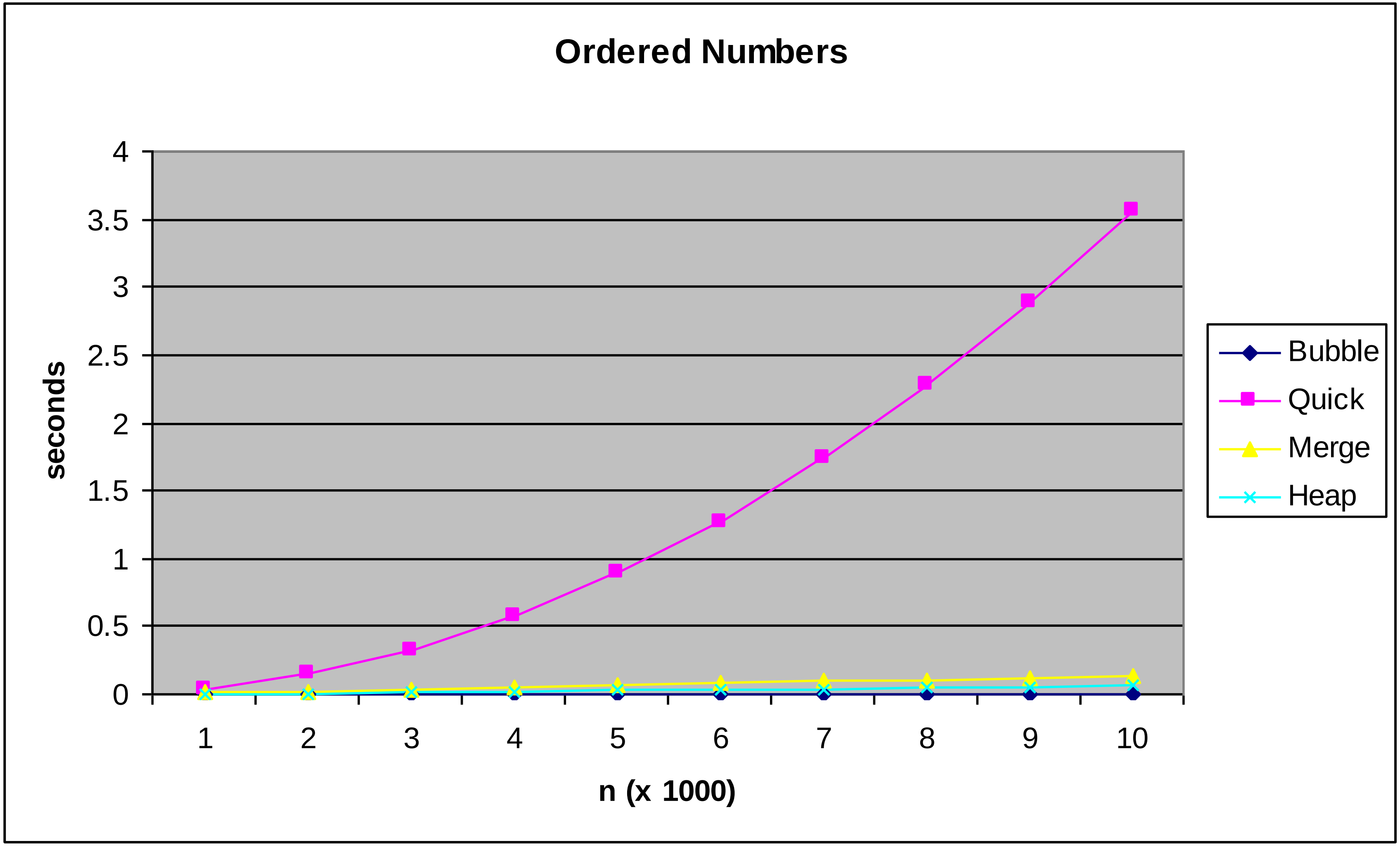
Empirical Comparison 1[1]



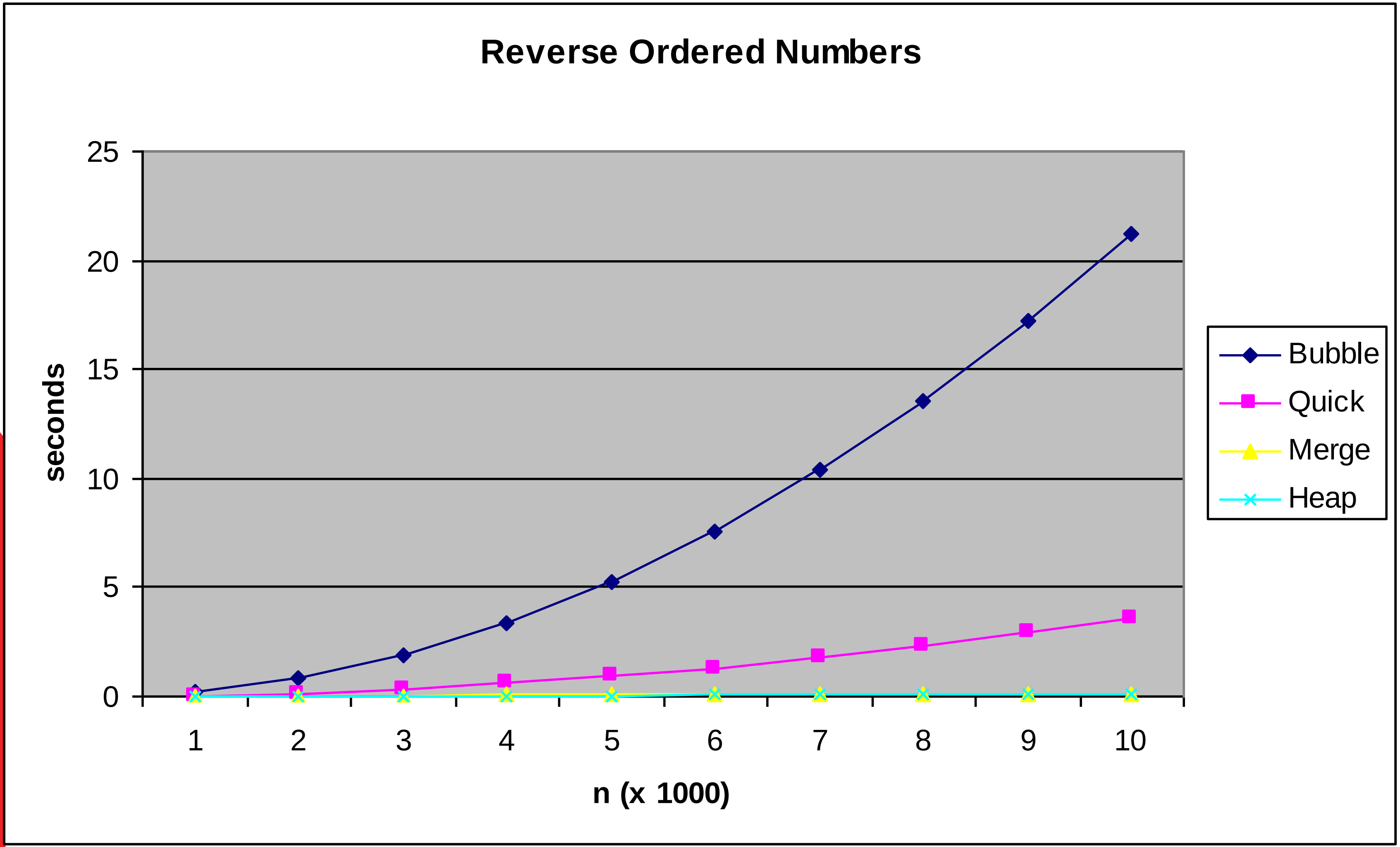
Empirical Comparison 2



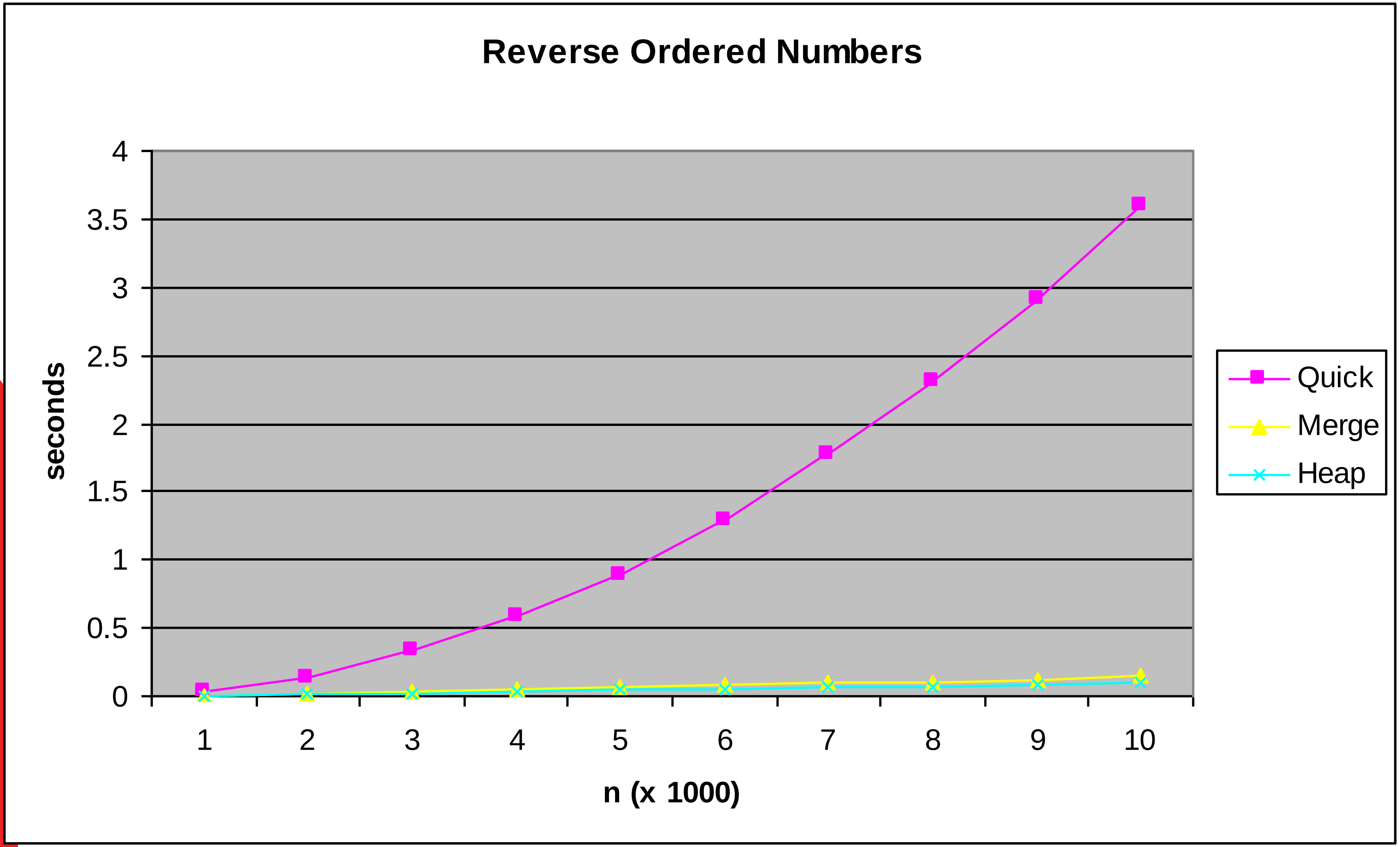
Empirical Comparison 3



Empirical Comparison 4



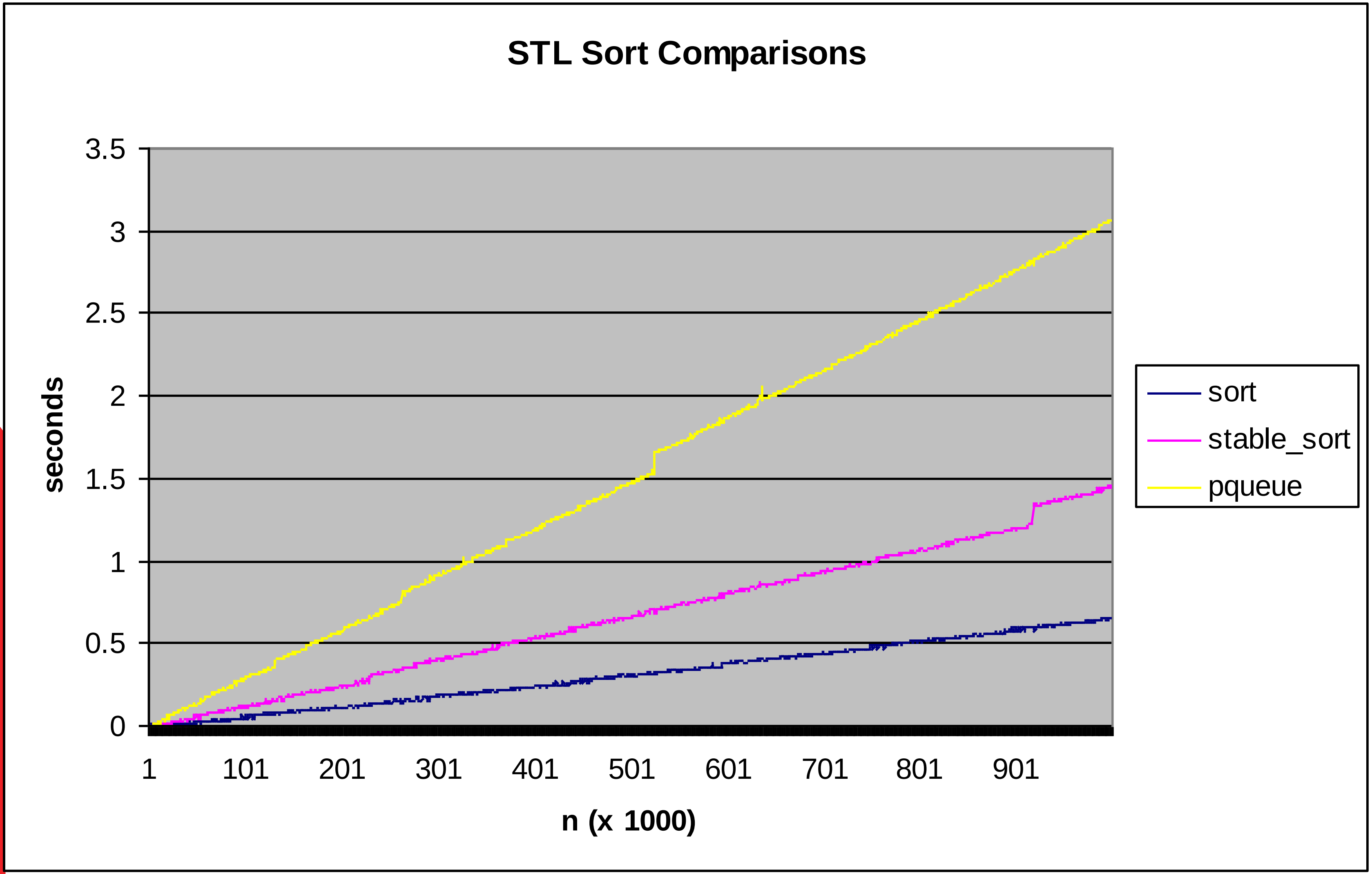
Empirical Comparison 5



STL Sorts

- There are a number of sort routines available in the STL algorithm library, a priority queue, which acts as a heap sort, plus some templates that have sorts of their own.
- **`sort(thing.begin(), thing.end())`** is a quicksort algorithm.
- **`stable_sort(thing.begin(), thing.end())`** does a stable sort, but I could not find definitive information on the algorithm used. However it is described as being like the sort algorithm, in which case, the type of split or partitioning routine will determine stability. But see Silicon Graphics site [1] notes where it is made explicit that stable_sort uses merge sort.
- **`pqueue<something>`** is a heap: put data into it and then pull it out and it is in order. [2]
- These are all very, very fast indeed: much faster than the ones any particular individual can write.
- This is because they have been written, reviewed, optimised etc. by multiple experts.

Empirical Comparison 6



Less Than Operator

- Note that these sorts require that a less than operator (<) be available for the ‘things’ being sorted.
- Therefore if you are sorting your own objects, you must overload a less than operator within the class to which they belong.
- To overload a < operator for a Circle class: [1]

```
bool Circle::operator < (const Circle &other)
{
    return (m_radius < other.m_radius);
}
```

Readings

- Textbook, chapter Standard Template Library, section on Algorithms.