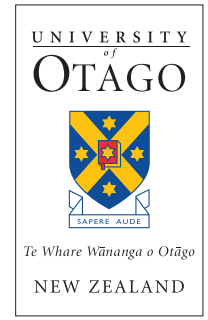
# Cosc 241
# Programming and Problem Solving
# Lecture 22 (16/5/2019)
# Sorting comparisons

Lech Szymanski
lechszym@cs.otago.ac.nz

*Keywords*: sorting algorithms, sorting in Java



# Sorting algorithms

- We've looked at five different sorting algorithms for arrays: selection sort, insertion sort, quicksort, mergesort and heapsort.
- The first two are quadratic, i.e., can only offer $O(n^2)$ performance guarantees, with insertion sort being clearly preferred.
- The last two offer $O(n \log n)$ performance guarantees - mergesort requires an additional $O(n)$ storage whereas heapsort can be carried out in place.
- Quicksort offers worst case $O(n^2)$ guarantees, but in a well-defined way offers "average" performance (i.e., on randomly ordered data) that's $O(n \log n)$.

# Sorting in Java

According to the javadoc for `java.util.Arrays`:

- The sorting algorithm used for arrays of primitive type "is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers $O(n \log n)$ performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations."
- For arrays of reference type: "This implementation is a stable, adaptive, iterative mergesort that requires far fewer than $n \lg n$ comparisons when the input array is partially sorted ..."

# What's left?

There are dozens of interesting investigations one could carry out with respect to how sorting works and how it might be improved. We have time for only a few which I've mentioned before:

- How can we remove the recursion in quicksort which is occasionally problematic due to its depth?
- In a hybrid sorting algorithm some complex sort is used on large (sub-)arrays, which then delegates to a simpler sort (usually insertion sort) on smaller sub-arrays. What's the right dividing line between large and small?

# Removing recursion

- When a method call is made, the complete state of the current method must be stored (since it might all be needed) and then restored when the called method exits.
- This is the main reason why there's a limitation to the depth of recursion that's allowed.
- Sometimes we know just how much state is required (and often, no local state is actually needed) so this is wasteful and we can eliminate recursion by explicit representation of the call stack, storing only information (typically the parameters of what would have been the recursive call) that's needed.

# In quicksort

- Since we actually want to manipulate the same array throughout we really just need to keep track of the sub-array bounds that are needed in the recursive calls.
- So the stack just keeps track of the sub-arrays that still need sorting and when we exhaust it we're done.

# Randomised algorithms

- In a randomised algorithm, the algorithm makes use of random choices in its decisions – the reason is usually to guarantee (as much as possible) some sort of average case behaviour.
- We could randomise quicksort by first shuffling the input array – then only if the shuffle accidentally produced a nearly-sorted (or reverse-sorted) array would we observe the slowdown entailed by that.
- Perhaps simpler is to choose a random element of any sub-array that we're partitioning as the pivot (instead of just the first one).

# Delegating sorting

- “One size fits all” is rarely, if ever, true – equally in algorithms as in real life.
- A frequent dividing line is that more complex algorithms that work well with large data sets may have significant overheads.
- On small data sets a simpler algorithm (that does not scale well) may be better.
- Only in trying to squeeze out the last drop of performance (as in the system sorts provided in a language’s libraries) would we generally worry about these issues.
- Let’s see whether delegating from quicksort to insertion sort can be effective.

# Knuth on optimization

In his 1974 Turing award lecture, Knuth said:

"premature optimization is the root of all evil".

But this is often taken out of context.

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%." (*emphasis mine*)

# Being Goldilocks

- *Never* worrying about the efficiency of code is bad practice - you may well be able to write code that is correct and passes all simple tests, but then fails badly because it does not scale well to the problems it is actually supposed to be dealing with.
- Trying to optimize *every bit of your code* is bad practice. It's time-inefficient (yours), creates large numbers of opportunities for bugs, and makes maintenance difficult.
- You need to find that "just right" spot where: code that doesn't need to be optimised because it's rarely used or needs to deal with only small cases is as simple and straight forward as possible, and code that does need to be optimised is well-tested and encapsulated so that if it creates a problem you know where to look!