Informatics 1

Introduction to Computation

Lecture 14

Laziness, Higher-order, and Sorting

Don Sannella

University of Edinburgh

Part I

The importance of being lazy

Searching for the first odd number

```
ho :: Int -> [Int]
ho n  =  (take 1 . filter odd) [0..n]

comp :: Int -> [Int]
comp n  =  take 1 [ x | x <- [0..n], odd x ]

rec :: Int -> [Int]
rec n  =  helper 0
  where
  helper :: Int -> [Int]
  helper i | i > n       =  []
           | odd i       =  [i]
           | otherwise   =  helper (i+1)
```

Quickcheck

```
prop_odd :: Int -> Bool
prop_odd n  =  a == b && b == c
  where
  a = ho n
  b = comp n
  c = rec n
```

```
[1 of 1] Compiling Main
Ok, one module loaded.
> quickCheck prop_odd
+++ OK, passed 100 tests.
```

Timing

```
> :set +s
> ho 1000000
[1]
(0.00 secs, 64,776 bytes)
> comp 1000000
[1]
(0.00 secs, 64,984 bytes)
> rec 1000000
[1]
(0.00 secs, 65,168 bytes)
```

How it works: rec

```
rec :: Int -> [Int]
rec n  =  helper 0
  where
  helper :: Int -> [Int]
  helper i | i > n       =  []
           | odd i       =  [i]
           | otherwise   =  helper (i+1)
```

```
  rec 1000000
=
  helper 0
=
  helper 1
=
  [1]
```

How it works: ho

```
ho :: Int -> [Int]
ho n  =  (take 1 . filter odd) [0..n]


  ho 1000000
=
  (take 1 . filter odd) [0..1000000]
=
  take 1 (filter odd [0..1000000])
=
  take 1 (filter odd (0 : [1..1000000]))
=
  take 1 (filter odd (1 : [2..1000000]))
=
  take 1 (1 : filter odd [2..1000000])
=
  1 : take 0 (filter odd [2..1000000])
=
  1 : []
```

Part II

Sum of odd squares
three ways

Sum of odd squares

```
ho :: Int -> Int
ho n  =  (foldl (+) 0 . map (^2) . filter odd) [0..n]

comp :: Int -> Int
comp n  =  sum [ x^2 | x <- [0..n], odd x ]

rec :: Int -> Int
rec n  =  helper 0 0
  where
  helper :: Int -> Int -> Int
  helper i a | i > n        =  a
             | odd i        =  helper (i+1) (a + i^2)
             | otherwise    =  helper (i+1) a
```

Quickcheck

```
prop_sqr :: Int -> Bool
prop_sqr n  =  a == b && b == c
  where
  a = ho n
  b = comp n
  c = rec n
```

```
Ok, one module loaded.
> quickCheck prop_sqr
+++ OK, passed 100 tests.
```

Runtimes in ghci

```
> :set +s
> ho 1000000
166666666666500000
(0.43 secs, 596,687,792 bytes)
> comp 1000000
166666666666500000
(0.67 secs, 628,685,832 bytes)
> rec 1000000
166666666666500000
(1.02 secs, 692,881,968 bytes)
```

The Moral

Usually coding involves tradeoffs:

simple and *slow*

vs.

complex and *fast*.

The big win is when you can find a way to be both *simple* and *fast*.

Part III

Sorting
three ways

Insertion sort

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []        =  e
foldr f e (x:xs)    =  x `f` foldr f e xs

   foldr f e [x,y,z] = (x `f` (y `f` (z `f` e)))


isort :: Ord a => [a] -> [a]
isort = foldr insert []
  where
  insert :: Ord a => a -> [a] -> [a]
  insert x []                       =  [x]
  insert x (y : ys) | x <= y        =  x : y : ys
                    | otherwise     =  y : insert x ys
```

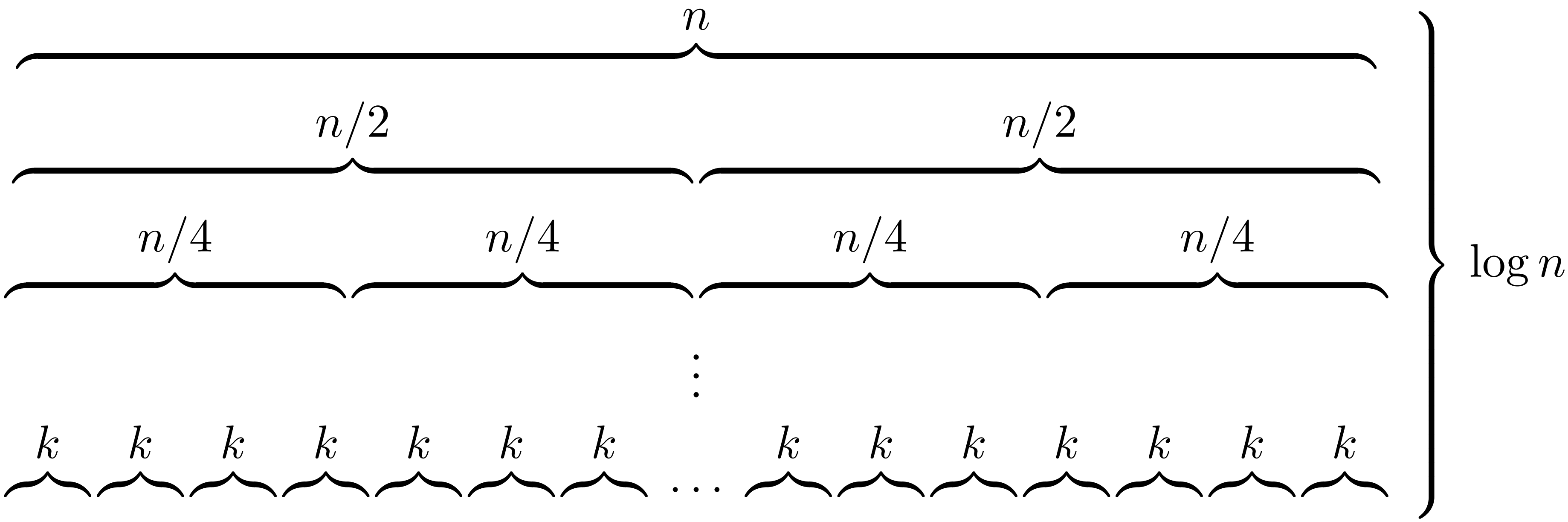
Quicksort

```
qsort :: Ord a => Int -> [a] -> [a]
qsort k xs | length xs <= k  =  isort xs
qsort k (y:xs) =
  qsort k [ x | x <- xs, x < y ]
  ++ [ y ] ++
  qsort k [ x | x <- xs, x >= y ]
```

Merge sort

```
msort :: Ord a => Int -> [a] -> [a]
msort k xs | length xs <= k   =  isort xs
           | otherwise        =  merge (msort k (take m xs))
                                       (msort k (drop m xs))
  where
  m = length xs `div` 2
  merge :: Ord a => [a] -> [a] -> [a]
  merge xs []                          =  xs
  merge [] ys                          =  ys
  merge (x:xs) (y:ys) | x <= y         =  x : merge xs (y:ys)
                      | otherwise      =  y : merge (x:xs) ys
```

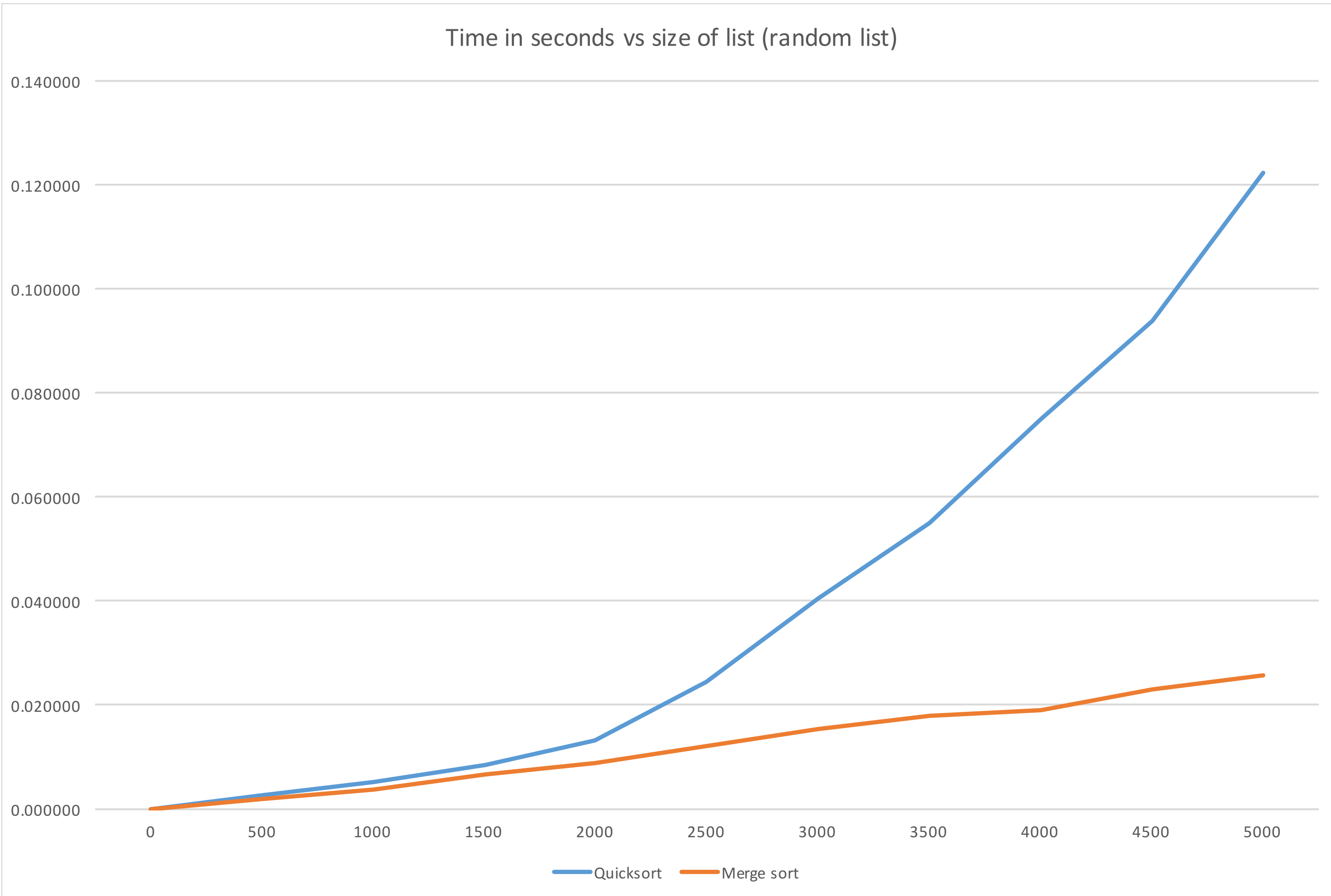
Why quicksort and mergesort are $O(n \log n)$

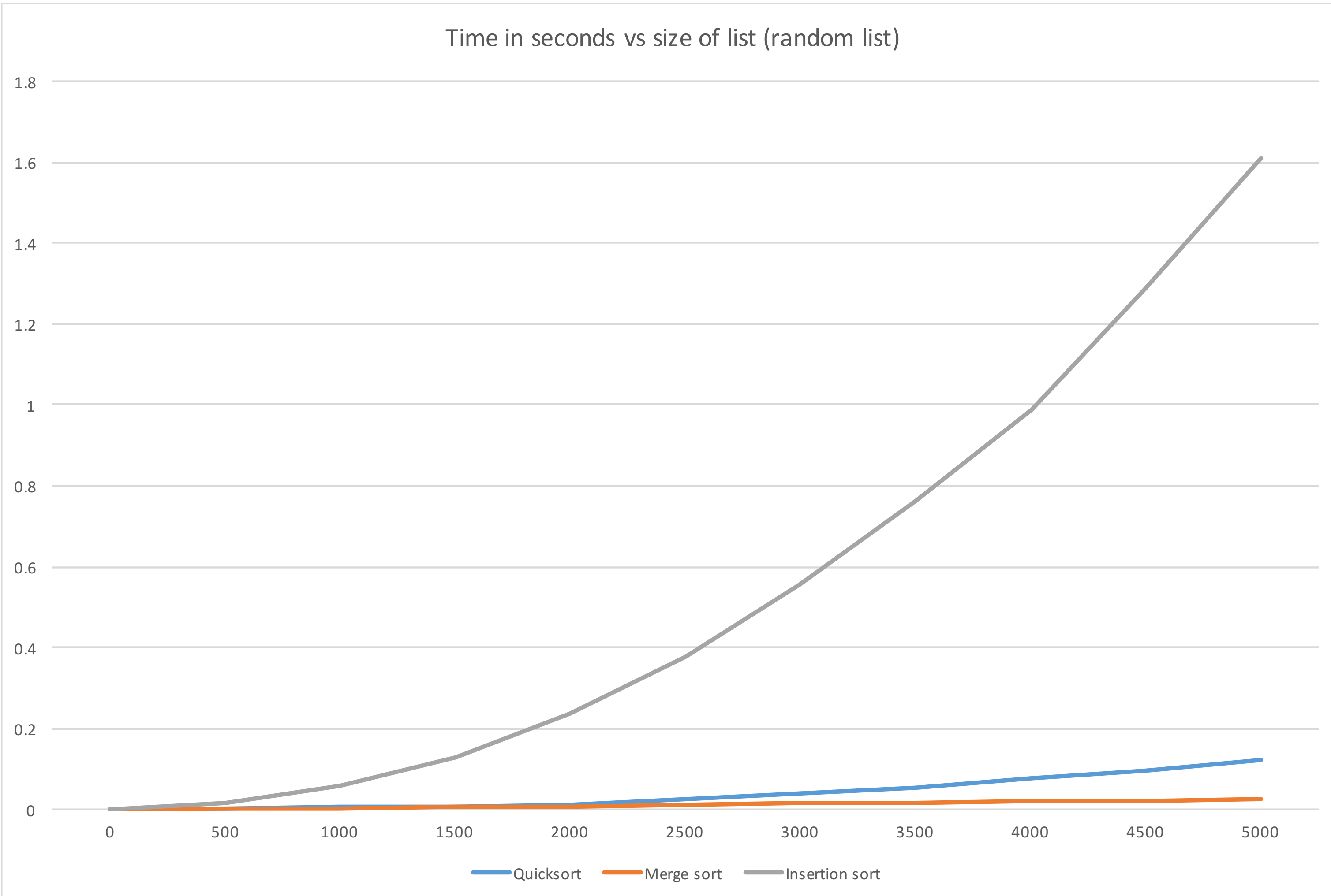


n	number of elements to be sorted
k	cutoff size

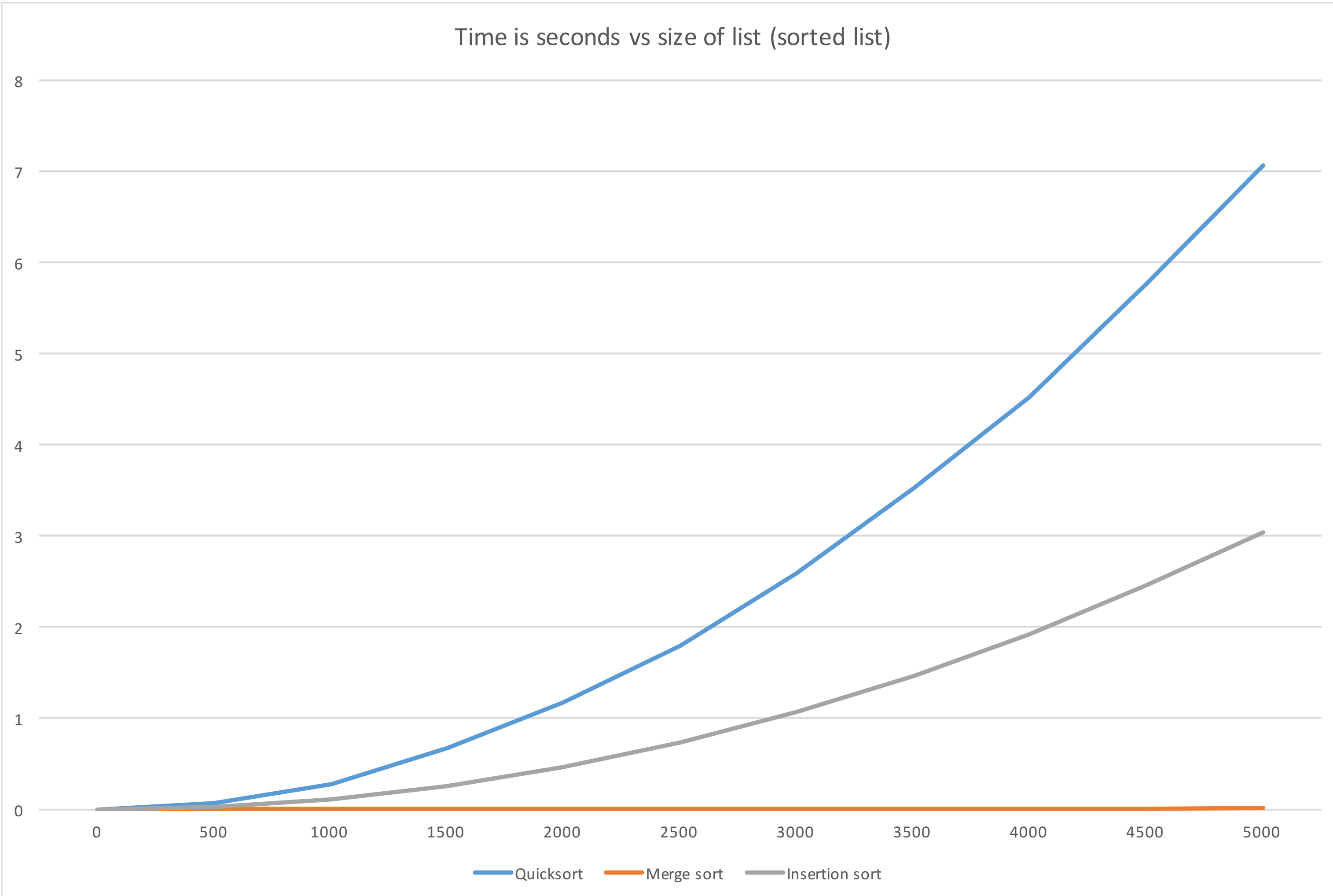
Part IV

A few graphs

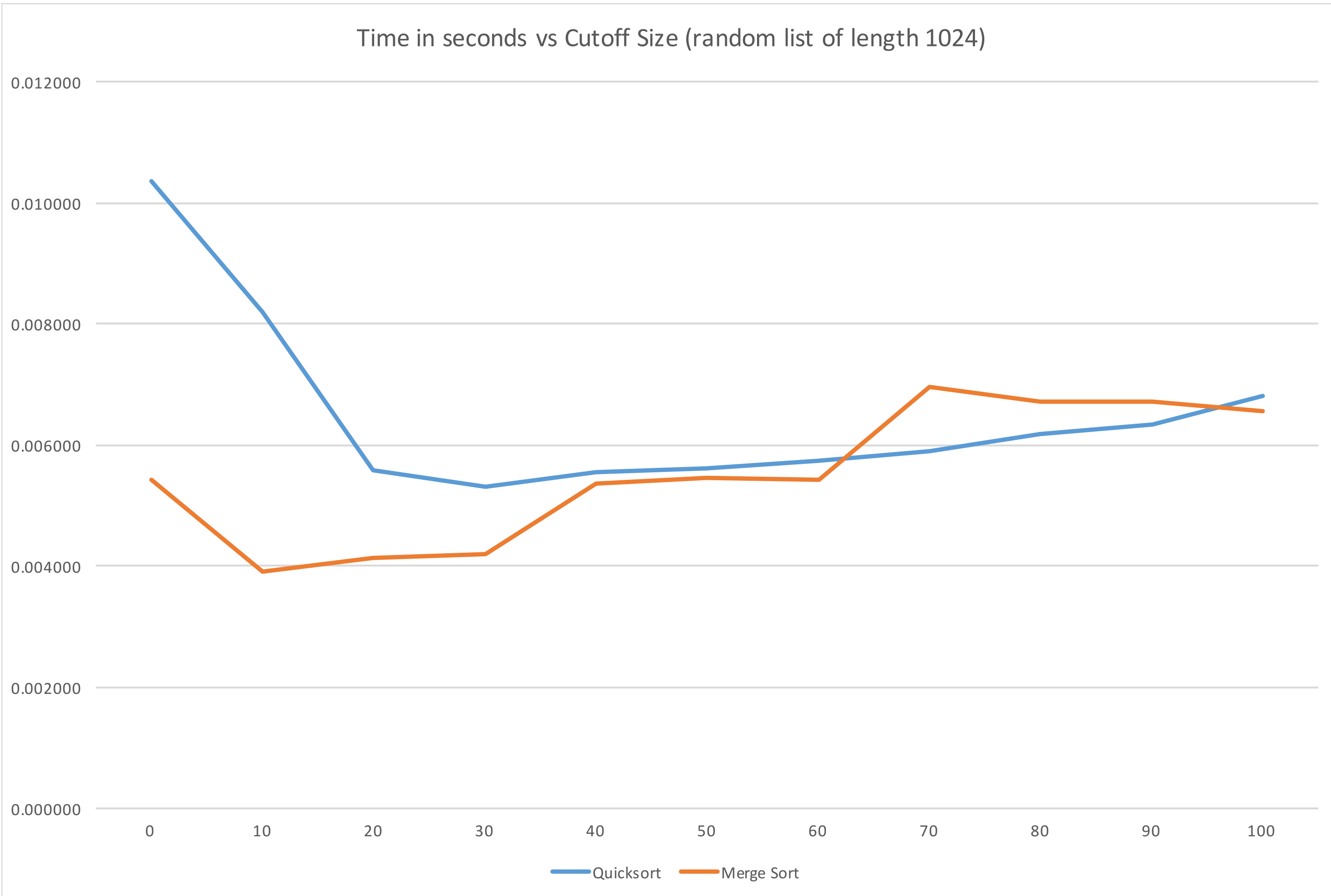

Time in seconds vs size of list (random list)
0.140000
0.120000
0.100000
0.080000
0.060000
0.040000
0.020000
0.000000
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Quicksort
Merge sort



Time in seconds vs size of list (random list)
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Quicksort
Merge sort
Insertion sort



Time is seconds vs size of list (sorted list)
8
7
6
5
4
3
2
1
0
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Quicksort
Merge sort
Insertion sort



Time in seconds vs Cutoff Size (random list of length 1024)
0.012000
0.010000
0.008000
0.006000
0.004000
0.002000
0.000000
0
10
20
30
40
50
60
70
80
90
100
Quicksort
Merge Sort