## CSharp Merge sort

public static partial class Ext **{** public static IEnumerable<int> MergeSort(this int[] source) {  
 if (source.IsEmpty()) {  
 return source;  
 }  
 if (source.Length == 1) {  
 return source;  
 }  
 var splittedArrays = source.SplitInto(2);  
 var left = MergeSort(splittedArrays[0]);  
 var right = MergeSort(splittedArrays[1]);  
 return Merge(left, right);  
 }  
  
 public static IEnumerable<int> Merge(this IEnumerable<int> left, IEnumerable<int> right) {  
 if (!left.Any()) {  
 return right;  
 }  
 if (!right.Any()) {  
 return left;  
 }  
 var leftHead = left.First();  
 var leftTail = left.Skip(1);  
 var rightHead = right.First();  
 var rightTail = right.Skip(1);  
 if (leftHead < rightHead) {  
 return leftHead.ToEnumerable().Concat(Merge(leftTail, right));  
 } else {  
 return rightHead.ToEnumerable().Concat(Merge(left, rightTail));  
 }  
 }  
  
 internal static T[][] SplitInto<T>(this T[] source, int count) {  
 IEnumerable<T[]> SplitInto() {  
 int length = source.Length;  
 count = length < count ? length : count;  
 if (count == 0) {  
 for (int i = 0; i < count; i++) {  
 yield return Array.Empty<T>();  
 }  
 }  
 int maxPartLength = (int)Math.Ceiling((double)length / count);  
 for (int i = 0; i < count; i++) {  
 int sourceIndex = i \* maxPartLength;  
 var diff = length - sourceIndex - maxPartLength;  
 int partLength = diff >= 0 ? maxPartLength : length - sourceIndex;  
 T[] part = new T[partLength];  
 Array.Copy(source, sourceIndex, part, 0, partLength);  
 yield return part;  
 }  
 };  
 return SplitInto().ToArray();  
 }  
**}**

# FSharp Merge sort

let rec merge\_v1List left right =   
 match left with  
 | [] -> right  
 | leftHead::leftTail ->   
 match right with  
 | [] -> left  
 | rightHead::rightTail ->  
 if leftHead < rightHead then leftHead::(merge\_v1List leftTail right) else rightHead::(merge\_v1List left rightTail)  
  
let rec merge\_v2Seq left right =   
 match (left, right) with  
 | (seq, \_) when Seq.isEmpty seq -> right  
 | (\_, seq) when Seq.isEmpty seq -> left  
 | (leftSeq, rightSeq) ->   
 let leftHead = Seq.head leftSeq  
 let leftTail = Seq.tail leftSeq  
 let rightHead = Seq.head rightSeq  
 let rightTail = Seq.tail rightSeq  
 if leftHead < rightHead   
 then Seq.append (Seq.singleton leftHead) (merge\_v2Seq leftTail rightSeq)  
 else Seq.append (Seq.singleton rightHead) (merge\_v2Seq left rightTail)  
  
let rec mergeSort\_v1List (array: 'a array) =   
 match array with  
 | [||] -> Array.toList array  
 | [|x|] -> Array.toList array  
 | \_ ->   
 let splitSortToList = Array.splitInto 2 >> Array.map mergeSort\_v1List >> Array.map Seq.toList  
 let [| left; right |] = splitSortToList array   
 merge\_v1List left right  
   
let rec mergeSort\_v2Seq (array: 'a array) =   
 match array with  
 | [||] -> Seq.empty  
 | [|x|] -> Seq.singleton x  
 | \_ ->   
 let splitSort = Array.splitInto 2 >> Array.map mergeSort\_v2Seq  
 let [| left; right |] = splitSort array   
 merge\_v2Seq left right

Here I wrote two versions of merge sorting algorithms according to two versions of merge function.  
It turned out that the version that uses lists (merge\_v1List) is drastically faster than the one that uses seq. 917.9 ns vs 5,968.3 ns for small test sets.

# CShart quick sort

public static IEnumerable<int> QuickSort(  
 this IEnumerable<int> source,  
 Func<IEnumerable<int>, Func<int, bool>, (IEnumerable<int> left, IEnumerable<int> right)> splitByFunc) {  
 if (!source.Any()) {  
 return new int[] { };  
 }  
 var head = source.First();  
 var tail = source.Skip(1);  
 var splitted = splitByFunc(tail, x => x < head);  
 var lessSorted = QuickSort(splitted.left, splitByFunc);  
 var greaterOrEqualSorted = QuickSort(splitted.right, splitByFunc);  
 return lessSorted.Concat(head.ToEnumerable()).Concat(greaterOrEqualSorted);  
}  
  
 public static (IEnumerable<T> left, IEnumerable<T> right) SplitBy\_v1LinkedList<T>(this IEnumerable<T> source, Func<T, bool> preficate) {  
 return source.Aggregate((left: new LinkedList<T>(), right: new LinkedList<T>()),  
 (acc, x) => {  
 if (preficate(x)) {  
 acc.left.AddLast(x);  
 return acc;  
 } else {  
 acc.right.AddLast(x);  
 return acc;  
 }  
 }  
 );  
}  
  
public static (IEnumerable<T> left, IEnumerable<T> right) SplitBy\_v2List<T>(this IEnumerable<T> source, Func<T, bool> preficate) **{** return source.Aggregate((left: new List<T>(), right: new List<T>()),  
 (acc, x) => {  
 if (preficate(x)) {  
 acc.left.Add(x);  
 return acc;  
 } else {  
 acc.right.Add(x);  
 return acc;  
 }  
 }  
 );  
**}**

Here I have two versions of SplitBy function which is used in QuickSort function. Version that uses List is slightly faster than the one that uses LinkedList (700.2 ns vs 736.2 ns). My guess is that it is to do with higher allocation for LinkedList as every node in it is essentially an object.

# FSharp quick sort

# let public splitBy predicate seq = Seq.fold (fun state x -> if predicate(x) then let newFst = List.append (fst state) [x] (newFst, snd state) else let newSnd = List.append (snd state) [x] (fst state, newSnd)) ([], []) seq let rec quickSort (a: 'a seq) = match a with | seq when Seq.isEmpty seq -> List.Empty | seq -> let head = Seq.head seq let tail = Seq.tail seq let (less, greaterOrEqual) = splitBy (fun x -> x < head) tail let lessSorted = quickSort less let greaterOrEqualSorted = quickSort greaterOrEqual List.append (List.append lessSorted [head]) greaterOrEqualSorted

# Benchmarks

**[**MemoryDiagnoser**]**public class SortBenches **{** private int[] \_array;  
 private FSharpList<int> \_list;  
   
 [Params(100, 200, 300, 400)] public static int N { get; set; }   
   
 [GlobalSetup]  
 public void GlobalSetup()  
 {  
 Random rand = new Random();  
 \_array = Enumerable.Range(0, N).Select(x => rand.Next()).ToArray();   
 }  
  
 [Benchmark]  
 [BenchmarkCategory("CSharp")]  
 public void CSharpMergeSortBench() =>Ext.MergeSort(\_array).ToArray();  
  
 //mergeSort\_v1List is chosen because it uses merge\_v1List which won with enormous gap in MergeBenches  
 [Benchmark]  
 [BenchmarkCategory("FSharp")]  
 public void FSharpMergeSortBench() =>MergeSort.mergeSort\_v1List(\_array);  
  
 //SplitBy\_v2List is chosen because it's faster according to SplitByBenches  
 [Benchmark]  
 [BenchmarkCategory("CSharp")]  
 public void CSharpQuickSortBench() => Ext.QuickSort(\_array, Ext.SplitBy\_v2List).ToArray();  
  
 [Benchmark]  
 [BenchmarkCategory("FSharp")]  
 public void FSharpQuickSortLinkedListBench() => QuickSort.quickSort(\_array);  
**}**

Library BenchmarkDotNet https://benchmarkdotnet.org was used for benchmarking. The results are:

/\*BenchmarkDotNet=v0.11.5, OS=Windows 10.0.17763.503 (1809/October2018Update/Redstone5)

Intel Core i5-3570 CPU 3.40GHz (Ivy Bridge), 1 CPU, 4 logical and 4 physical cores

.NET Core SDK=2.2.107

[Host] : .NET Core 2.2.5 (CoreCLR 4.6.27617.05, CoreFX 4.6.27618.01), 64bit RyuJIT

DefaultJob : .NET Core 2.2.5 (CoreCLR 4.6.27617.05, CoreFX 4.6.27618.01), 64bit RyuJIT

| Method | N | Mean | Error | StdDev | Allocated |

|------------------------------- |---- |--------------:|--------------:|--------------:|------------:|

| CSharpMergeSortBench | 100 | 4,877.64 us | 41.9686 us | 39.2575 us | 2435.44 KB |

| FSharpMergeSortBench | 100 | 45.80 us | 0.8943 us | 1.0982 us | 74.09 KB |

| CSharpQuickSortBench | 100 | 65.50 us | 0.2191 us | 0.1942 us | 70.7 KB |

| FSharpQuickSortLinkedListBench | 100 | 213.06 us | 2.5355 us | 2.3717 us | 592.36 KB |

| CSharpMergeSortBench | 200 | 30,531.86 us | 112.0274 us | 99.3094 us | 9509.95 KB |

| FSharpMergeSortBench | 200 | 104.10 us | 0.5256 us | 0.4659 us | 165.38 KB |

| CSharpQuickSortBench | 200 | 144.99 us | 1.5900 us | 1.4873 us | 146.15 KB |

| FSharpQuickSortLinkedListBench | 200 | 465.25 us | 0.9925 us | 0.8798 us | 1333.2 KB |

| CSharpMergeSortBench | 300 | 98,278.15 us | 742.3274 us | 694.3736 us | 20648.33 KB |

| FSharpMergeSortBench | 300 | 167.87 us | 0.4872 us | 0.4068 us | 261.84 KB |

| CSharpQuickSortBench | 300 | 223.92 us | 1.5402 us | 1.4407 us | 221.27 KB |

| FSharpQuickSortLinkedListBench | 300 | 1,272.03 us | 30.1845 us | 30.9972 us | 4010.09 KB |

| CSharpMergeSortBench | 400 | 226,668.94 us | 1,826.5497 us | 1,708.5557 us | 36660.55 KB |

| FSharpMergeSortBench | 400 | 237.33 us | 0.9780 us | 0.8166 us | 363.68 KB |

| CSharpQuickSortBench | 400 | 329.79 us | 2.0641 us | 1.9307 us | 306.16 KB |

| FSharpQuickSortLinkedListBench | 400 | 1,148.27 us | 4.9538 us | 4.6338 us | 3430.16 KB |

// \* Hints \*

Outliers

SortBenches.CSharpQuickSortBench: Default -> 1 outlier was removed (66.61 us)

SortBenches.CSharpMergeSortBench: Default -> 1 outlier was removed (30.93 ms)

SortBenches.FSharpMergeSortBench: Default -> 1 outlier was removed, 2 outliers were detected (103.16 us, 105.30 us)

SortBenches.FSharpQuickSortLinkedListBench: Default -> 1 outlier was removed (469.24 us)

SortBenches.FSharpMergeSortBench: Default -> 2 outliers were removed (169.93 us, 171.63 us)

SortBenches.FSharpMergeSortBench: Default -> 2 outliers were removed (241.57 us, 241.98 us)

// \* Legends \*

N : Value of the 'N' parameter

Mean : Arithmetic mean of all measurements

Error : Half of 99.9% confidence interval

StdDev : Standard deviation of all measurements

Gen 0 : GC Generation 0 collects per 1000 operations

Gen 1 : GC Generation 1 collects per 1000 operations

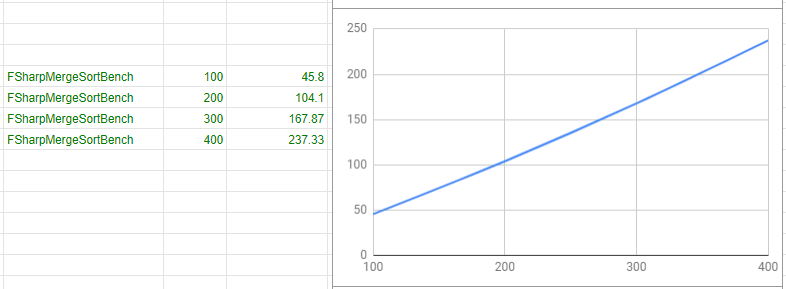
Gen 2 : GC Generation 2 collects per 1000 operations

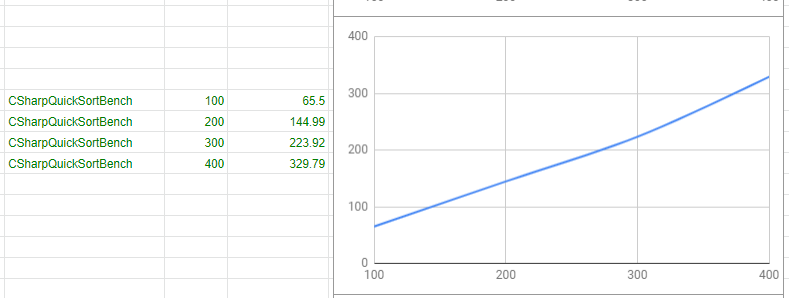
Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)

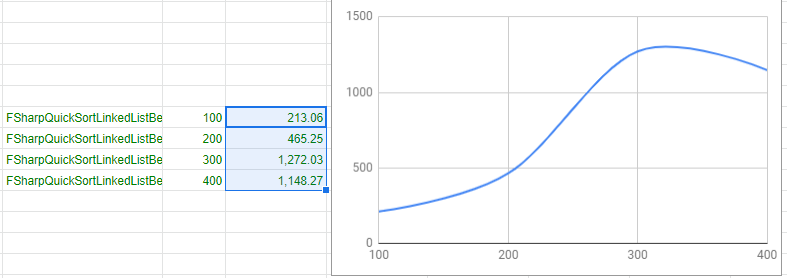
1 us : 1 Microsecond (0.000001 sec)\*/

// \* Diagnostic Output - MemoryDiagnoser \*









The reason why the last plot has significant deviation might be the fact that new test sets are created every iteration. That means that every time we get new random array of integers. So, in sometimes it could generate arrays that are "bad" for quick sorting algorithm. "Bad" array is that that doesn't split in equally sized parts. This problem could be solved by increasing sizes of testing arrays, but it leads to slow benchmarks execution.  
CSharpMergeSortBench shows very high execution time. It might be the result of using LINQ in its implementation. Intense IEnumerator allocation leads to the latency. It's clear from the column Allocated in the table.