THE UGSORT ALGORITHM

Tree, Ian J.

unaffiliated researcher

Author Note

Ian J. Tree, unaffiliated researcher

Eindhoven, the Netherlands

Email: ian.tree@acm.org

Abstract

This paper describes the novel UGSort merge sorting algorithm. The algorithm is simple,

elegant, flexible and easy to implement.

Keywords: sort, algorithm, merge

The UGSort Algorithm

The UGSort algorithm (Unexpectedly Good Sort) is a simple, elegant and flexible algorithm. It is easy to implement in different settings and is efficient.

Revised: 11/09/2023 – Added binary search of partitions for new key.

The Algorithm

The algorithm is a merge sorting algorithm based on partitioning the input keys into an array of double ended queues (dequeues) and then merging the double ended queues into a single queue from which the results of the sort can be output.

Sorting a set of keys KS = $\{k_1, k_2...k_n\}$ with n keys begins with initialising the first element of an array (S) of double ended queues S_0 with the first key available from the input set k_1, both the head and the tail of S_0 will be set to k_1.

Each subsequent key in the input set $k_i = k_2...k_n$ will be added to the partitions by testing against the head and tail keys in each of the available double ended queues in S. The following describes a linear search of the partitions for the first qualifying partition, see the later section "Binary Search" for a description of replacing the linear search with a binary search. If the key k_i is less than or equal to the head value of the queue, then the key is added to the head of the queue. If the key was not consumed by the head of the queue, then if it is greater than or equal to the tail value of the queue then, it is added to the tail of the queue. If the key was not consumed by any of the queues in S then a new queue is added to the end of the array initialised with the unused key k_i.

If a new queue was added to the array S then the number of elements in the array is checked against a computed maximum S_{max} if it exceeds the maximum then a pre-emptive merge is performed. The value of S_{max} is the greater of 100 or ($\frac{sqrt\left(\frac{i}{2}\right)}{2}$), where i is the number of keys processed so far. The pre-emptive merge will reduce the number of queues in the

array S by merging alternate elements in the array with their predecessor. Thus, S_1 is merged with S_0 and removed from the array and so on for each alternate element in the array.

After the last key k_n has been added to the queues in the array S a final merge is performed where all queues are merged into the first queue in the array S_0.

The input set KS is now sorted into ascending sequence in the queue S_0.

Pseudocode

Sort Input: $KS = \{k_1, k_2 ... k_n\}$

Partition Array: $S = \{\}$

Max Partitions: $m = 100$

S_0 = double_ended_queue(k_1)

ForEach k_i in $k_2 ... k_n$

 ForEach S_j in S

  If $k_i <= S_j$.head

   S_j.head = k_i

   Key_consumed = true

   Exit ForEach

  End If

  If $k_i >= S_j$.tail

   S_j.tail = k_i

   Key_consumed = true

   Exit ForEach

  End If

 End ForEach

  If (!Key-consumed)

  S_{j+1} = double_ended_queue(k_i)

If (j+1 > m)

// Perform pre-emptive merge

ForEach S_j in S

Merge S_{j+1} into S_j

Delete S_{j+1}

End ForEach

NewMax = (sqrt(i/2)/2)

If (NewMax > m) m = NewMax

End If

End If

End ForEach

// Perform final merge

ForEach S_j in $S_1...S_m$

Merge S_j into S_0

End ForEach

Binary Search

The linear search of the partitions to locate the correct one to add a new key to may be replaced by a more efficient binary search. The linear search finds the first partition where the new key can be added to the head or the tail of the partition, proceeding from the lowest index partition S_0 to the last partition S_m, the binary search must mirror this constraint.

For each new key k_i begin by testing against the boundary conditions for the complete array of partitions. If the new key is less than or equal to the low key of the first partition S_0 then add the key to S_0 as the new low key. If the new key is greater than or equal to the high key of the first partition S_0 then add the key to S_0 as the new high key. If the key was not consumed by the first S_0 partition then test to see if the key will not by consumed by the last

partition S_m and therefore cause a new partition to be initialised with the key. If the key was not consumed by either of the boundary conditions then start at the mid-point of the array $S_{m/2}$ with a delta of m/4. At each partition test the new key to see if it would qualify for addition the current partition if so then test if it would also qualify for the preceding partition if so subtract the delta from the current partition array index and halve the delta. If the key did not qualify for addition to the preceding partition then add the key to the current partition as the new high or low key as appropriate. If the key did not qualify for the current partition then also test it against the following partition. If the key qualifies for the following partition then add it as the new high or low key to the following partition as appropriate. If it did not qualify for the following partition then add the delta to the partition array index and halve the delta. The search continues until the new key is consumed by a partition.

If a new queue was added to the array S then the number of elements in the array is checked against a computed maximum S_{max} if it exceeds the maximum then a pre-emptive merge is performed. The value of S_{max} is the greater of 100 or $2\sqrt{(\frac{i}{2})}$, where i is the number of keys processed so far. Note that the computation for S_{max} has changed from that used with the linear search.

Binary Search Pseudocode

This snippet of pseudocode replaces the linear search in the reference pseudocode, it is executed for each new key k_i in $k_2...k_n.$

If $k_i <= S_0$.head

    S_0.head = k_i

    Exit

End If

If $k_i >= S_0$.tail

S_0.tail = k_i

Exit

End If

If $k_i > S_j$.head and $k_i < S_j$.tail

S_{j+1} = double_ended_queue(k_i)

If (j+1 > m)

// Perform pre-emptive merge

ForEach S_j in S

Merge S_{j+1} into S_j

Delete S_{j+1}

End ForEach

NewMax = (2*sqrt(i/2))

If (NewMax > m) m = NewMax

End If

Exit

End If

// Perform Binary Search

c = m/2 // Midpoint of partition array

delta = m/4 // Binary chop delta

Loop

If $k_i > S_c$.head and $k_i < S_c$.tail

If $k_i > S_{c-1}$.head and $k_i < S_{c-1}$.tail

c = c – delta

Else

S_{c-1}.head = k_i or S_{c-1}.tail = k_i

                            Exit

        Else

                If $k_i > S_{c+1}$.head and $k_i < S_{c+1}$.tail

                        c = c + delta

                Else

                        S_c.head = k_i or S_c.tail = k_i

                        Exit

        EndIf

EndLoop

Key Stability

        Key stability is the sorting property of preserving the input sequence for keys of equal

value. The UGSort algorithm as presented is not stable. However, it can be made stable by

making minor changes to the algorithm. The tests for addition to the head or tail of a queue

must be changed from less than or equal and greater than or equal to less than and greater

than. The merge processing must always give precedence to the leftmost (lowest index)

queue. These changes will result in a slightly less efficient algorithm.

Best and Worst Cases

        The "Best-Case" input for the algorithm is an input set SK that is pre-sorted into

descending sequence, this results in a single queue being used with all keys being added to

the head of the queue. Pre-sorting into ascending sequence offers the next best case input, it

also uses a single queue but will cost an additional key comparison for each key.

        The "Worst-Case" input for the algorithm is an input set SK that is pre-sorted into

ascending sequence and then reordered by taking alternating keys from the head and the tail

of the sorted queue. For an ordered set of keys KS = $\{k_1, k_{2...}k_n\}$ the worst-case input would be

{k₁,kₙ,k₂,kₙ₋₁...}. Reversing the sequence of a "Worst-Case" set of keys results in a set that will also be a "Best-Case" input.

Sort Properties

As noted in the previous section the UGSort algorithm is not stable but, may be transformed into a stable sort by small modifications. The algorithm is not capable of in-place sorting and is therefore an out-of-place algorithm.

It is possible to introduce parallel processing for the pre-emptive and final merge phases as individual merge operations can be done in parallel, there is no additional co-ordination needed for parallel merges.