K-Way Merge

K-way merge

Given:

K sorted lists (or arrays) of numbers.

Problem:

Combine all K lists into one sorted list.

K-way merge

Given:

K sorted lists (or arrays) of numbers (count of all numbers is N, we can assume each list has N/K numbers)

Problem:

Combine all K lists into one sorted list (of size N)

Use case:

If u r trying to sort a lot of numbers … such that all of them would not fit in the RAM … so u would need to:

- Divide data set into K partitions
  - Partition size should fit in RAM
- Sort partitions individually
- Merge all partitions into one data set. ← Merging K sorted lists, produces final output

K-way merge

Option 0:

1. Combine all the arrays into one array.
2. Sort this array.

Time complexity of this approach: N log N

But this is a naïve approach.

If we had K unsorted arrays, we could have done the same thing.

We should make use of the fact that the given data is already sorted.

K-way merge

Option 1:

1. We take first two lists (each of size N/K), merge them.
2. Then take output from above and merge it with the 3rd list,
3. Then that output merged with the 4th list.

Time complexity of this approach: ?

K-way merge

Option 1:

1. Take first two lists, merge them.
2. Take output from above and merge with 3rd list,
3. Take output from above and merge with 4th list.

Time complexity of this approach:

1.	N/K	+	N/K	=	2N/K elements processed
2.	2N/K	+	N/K	=	3N/K elements processed
3.	…				
4.	(K-1) N/K	+	N/K	=	KN/K = N elements processed

Next: sum them up

K-way merge

Option 1:

1. Take first two lists, merge them.
2. Take output from above and merge with 3rd list,
3. Take output from above and merge with 4th list.

Time complexity of this approach:

1.	N/K	+	N/K	=	2N/K elements processed
2.	2N/K	+	N/K	=	3N/K elements processed
3.	…				
4.	(K-1) N/K	+	N/K	=	KN/K = N elements processed

Sum them up:

$2N/K + 3N/K + 4N/K + \ldots + KN/K$

$N/K (2 + 3 + \ldots + K)$

$N/K (K (K + 1) / 2 - 1) = N (K + 1) / 2$ ➔ $O (N K)$

K-way merge

Option 2:

1. Create pairs (2 lists) and merge them together.
   - Now u will have K/2 lists

2. Repeat step 1.
   - Now u will have K/4 lists

3. How many steps till we get to one list?

K-way merge

Option 2:

1. Create pairs (2 lists) and merge them together.
   - Now u will have K/2 lists

2. Repeat step 1.
   - Now u will have K/4 lists

3. How many steps till we get to one list?
   - $\log_2 K$ steps

Time complexity of this approach: ?

K-way merge

Option 2:

1. Take 2 lists at a time and merge them.
2. Now take 2 lists again and merge them.
3. How many steps till we get to one list?
   - $\log_2 K$ steps

Time complexity of this approach:

At each step, we are processing a total of N elements.

And there are $\log_2 K$ steps.

➔ O (N log K)

Note:

Above steps are essentially what we do in Merge sort … merge sorted sub lists, until u get one final sorted list.

K-way merge

Option 2:

1. Take 2 lists at a time and merge them.
2. Now take 2 lists again and merge them.
3. How many steps till we get to one list?
   - $\log_2 K$ steps

Is there some optimization u could do in this approach?

K-way merge

Option 2:

1. Take 2 lists at a time and merge them.
2. Now take 2 lists again and merge them.
3. How many steps till we get to one list?
   - $\log_2 K$ steps

Is there some optimization u could do in this approach?

What if u had a multi core CPU … nowadays its hard to find one that is not multicore.

K-way merge

Option 2:

1. Take 2 lists at a time and merge them.
2. Now take 2 lists again and merge them.
3. How many steps till we get to one list?
   - $\log_2 K$ steps

Is there some optimization u could do in this approach?

What if u had a multi core CPU … nowadays its hard to find one that is not multicore.

Since the merging of the pairs of “2 lists at a time” is independent of each other, we could potentially think about parallelizing this.

K-way merge

Option 2:

1. Take 2 lists at a time and merge them.
2. Now take 2 lists again and merge them.
3. How many steps till we get to one list?
   - $\log_2 K$ steps

Since the merging of the pairs of "2 lists at a time" is independent of each other, we could potentially think about parallelizing this.

1. Merge 2 lists on each core.
   - For simplicity, we will assume:
     - No parallelization overheads (there are, but we will ignore them).
     - Enough cores available ➔ K/2 cores available.
   - So, all pairs of lists will get merged at the same time, one pair on each core.

   - How many elements r getting merged on one core in this first step?

K-way merge

Option 2:

1. Take 2 lists at a time and merge them.
2. Now take 2 lists again and merge them.
3. How many steps till we get to one list?
   - $\log_2 K$ steps

1. Merge 2 lists on each core.
   - For simplicity, we will assume:
     - No parallelization overheads (there are, but we will ignore them).
     - Enough cores available ➔ K/2 cores available.
   - So, all pairs of lists will get merged at the same time, one pair on each core.

   - How many elements r getting merged on one core in this <u>first step</u>?
     - N/K + N/K = 2N/K

K-way merge

Option 2:

1. Take 2 lists at a time and merge them.
2. Now take 2 lists again and merge them.
3. How many steps till we get to one list?
   - $\log_2 K$ steps

1. Merge 2 lists on each core.
   - For simplicity, we will assume:
     - No parallelization overheads (there are, but we will ignore them).
     - Enough cores available ➔ K/2 cores available.
   - So, all pairs of lists will get merged at the same time, one pair on each core.

2. In the 2nd step, we will merge 2 lists on each core, except each list will now be of size 2N/K
   - How many elements r getting merged on one core in this second step?
     - 2N/K + 2N/K = 4N/K

K-way merge

1. Merge 2 lists on each core.
   - For simplicity, we will assume:
     - No parallelization overheads (there are, but we will ignore them).
     - Enough cores available ➔ K/2 cores available.
   - So, all pairs of lists will get merged at the same time, one pair on each core.

2. In the 2nd step, we will merge 2 lists on each core, except each list will now be of size 2N/K
   - How many elements r getting merged on one core in this second step?
     - 2N/K + 2N/K = 4N/K

3. In the 3rd step, size of each list is 4N/K, so we r merging 8N/K on each core.

K-way merge

1. Merge 2 lists on each core.
   - For simplicity, we will assume:
     - No parallelization overheads (there are, but we will ignore them).
     - Enough cores available ➔ K/2 cores available.
   - So, all pairs of lists will get merged at the same time, one pair on each core.

2. In the 2nd step, we will merge 2 lists on each core, except each list will now be of size 2N/K
   - How many merges r happening on one core in this second step?
     - 2N/K + 2N/K = 4N/K
3. In the 3rd step, size of each list is 4N/K, so we r merging 8N/K on each core.

Total merges happening:

$2N/K + 4N/K + 8N/K + 16N/K + \ldots + 2^{\log K} N/K$

Note that $2^{\log K} = K$.
Proof in a few slides, if u care.

K-way merge

Total merges happening:

$2N/K + 4N/K + 8N/K + 16N/K + \ldots + 2^{\log K} N/K$

$N/K (2 + 4 + 8 + 16 + \ldots + 2^{\log K})$ // sum of powers of 2 (formula coming up)

K-way merge

Formula for sum of first n powers of 2:

$2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^n$

$1 + 2 + 4 + 8 + 16 + \ldots + 2^n = \mathbf{2^{n+1} - 1}$

Examples:

$2^0 + 2^1 + 2^2 = 7$ which is $2^3 - 1$

$2^0 + 2^1 + 2^2 + 2^3 = 15$ which is $2^4 - 1$

K-way merge

Total merges happening:

$2N/K + 4N/K + 8N/K + 16N/K + \ldots + 2^{\log K} N/K$

$N/K (2 + 4 + 8 + 16 + \ldots + 2^{\log K})$ // sum of powers of 2

$N/K (2^{(\log K) + 1} - 1 - 1)$ // the second -1 is because we don't have a 1 in the above sequence (see formula below)

$N/K (2^{(\log K) + 1} - 2)$

$N/K (2 * 2^{(\log K)} - 2)$ // since $2^{(\log K) + 1} = 2 * 2^{(\log K)}$

$2N/K (2^{(\log K)} - 1)$ // since $2^{\log_2 K} = K$ Proof coming up

$2N/K (K - 1)$

$2N (1 - 1 / K)$

If K = 2, then this is N elements being merged.

If K = 16, then we have $2N (1 - 1/16)$ = approaching 2N

K-way merge

Proof $2^{\log_2 K} = K$

Let $2^{\log_2 K} = y$

Take log (base 2) of both sides:

$\log_2 (2^{\log_2 K}) = \log_2 y$

$\log_2 K * \log_2 2 = \log_2 y$ ← Applying this formula: $log (a^b) = b * log\ a$

$\log_2 K * 1 = \log_2 y$

$\log_2 K = \log_2 y$

➔ $y = K$ ➔ $2^{\log_2 K} = K$

K-way merge

Option 3:

We can use a min heap approach to do this.

1. Take the first element from each of the K lists and insert into the min heap.
2. Min heap is of size K
3. Loop
   - Extract the root element. ➔ place it in output array
   - Take the next element from the list of the root element extracted in step above and insert into min heap

Next: Time and space complexity

K-way merge

1. Take the first element from each of the K lists and insert into the min heap. ← O (?)

2. Min heap is of size K

3. Loop (while any of the lists have elements left)
   - Extract the root element. ➔ place it in output array
   - Take the next element from the list of the root element extracted in step above and insert into min heap

4. While min heap ! empty
   - Extract the root element. ➔ place it in output array

K-way merge

1. Take the first element from each of the K lists and insert into the min heap. ← O (K)

2. Min heap is of size K

3. Loop ← Runs N – K times
   - Extract the root element. ➔ place it in output array ← O (log K)
   - Take the next element from the list of the root element extracted in step above and insert into min heap. ← O (log K)

4. While min heap ! empty ← Runs K times
   - Extract the root element. ➔ place it in output array ← O (log K)

Time: O (K) + O ((N – K) (log K + log K)) + O (K log K)

O (K) + O (N log K) - O (K log K) + O (K log K)

O (K) + O (N log K)

O (N log K)

Space: O (K)

External sort

When sorting a huge data set that cannot fit into RAM, external sorting is used.

Partition the data set into *runs*, where each run would fit into RAM.

Sort the partitions. ← Can be done using merge sort or whatever other algorithm.

Now, use the K-way merge to merge these sorted partitions.

External sort

When sorting a huge data set that cannot fit into RAM, external sorting is used.

Partition the data set into *runs*, where each run would fit into RAM.

Sort the partitions. ← Can be done using merge sort or whatever other algorithm.

Now, use the K-way merge to merge these sorted partitions.

Note on a potential optimization that is sometimes used:

We have talked about the locality of reference and cache friendly memory accesses.

When running merge sort, if the size of ur sub array is “small enough” (i.e., fits into the CPU cache), then, u could use some in-place sort algorithm such as Insertion or Quicksort.

And for sizes over this “small enough”, revert to merge sort.

This hybrid approach could be considered for cases where performance is paramount (of course, we all want high performance, but sometimes there is a tradeoff of performance vs complexity of solution/code, and what is “small enough” for a given CPU may not be the case for other SKUs…in other words, a little more complexity / things to worry about)

Power of 3

Given:

An integer.

Problem:

Return true if given integer is a power of 3, false otherwise.

Power of 3

Given:

An integer.

Problem:

Return true if given integer is a power of 3, false otherwise.

Option 1:

Can compute all powers of 3 until :

- You get to the given number, in which case return true
- OR, u get a number > given number … return false.

What would the time and space complexity be for this solution ?

Power of 3

Option 1:

```
bool  IsPowerOf3 (  int  n )
        int pwr3 = 1;
        while  pwr3 <= n
                pwr3 = pwr3 * 3

        return  pwr3 == n
```

Time: $O (\log_3 n)$

Space: O (1)

Power of 3

Option 2:

Notice that all powers of 3 end in either

1 or 3 or 9 or 7

So, u could check if the last digit is one of these, and if so, then invoke option 1, else return false.

Above check would be O (1) time and space.

And there may be more ways to do this.