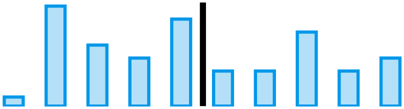
Merge Sort (Divide & Conquer)

(Slides from Alan P. Sexton)

Idea:

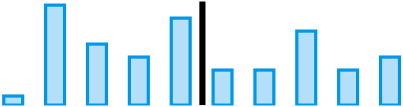
1. Split the array into two halves:



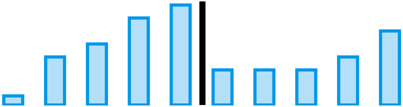
Merge Sort

Idea:

1. Split the array into two halves:



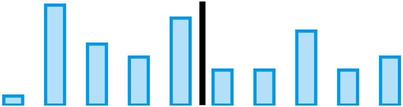
2. Sort each of them recursively:



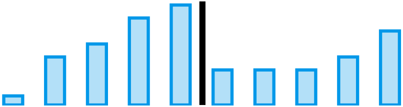
Merge Sort

Idea:

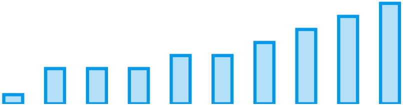
1. Split the array into two halves:



2. Sort each of them recursively:



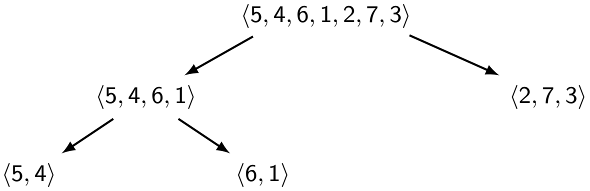
3. Merge the sorted parts:



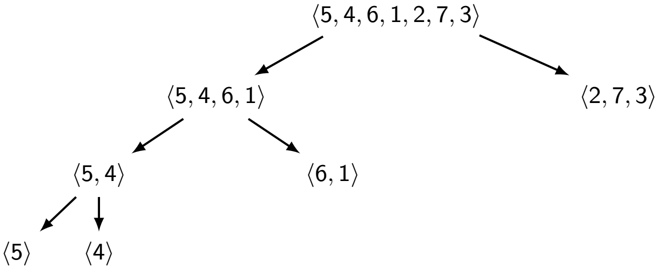
Example: Merge Sort run

$$\begin{tikzcd} & \langle 5, 4, 6, 1, 2, 7, 3 \rangle \arrow[dl] \arrow[dr] & \\ \langle 5, 4, 6, 1 \rangle & & \langle 2, 7, 3 \rangle \end{tikzcd}$$

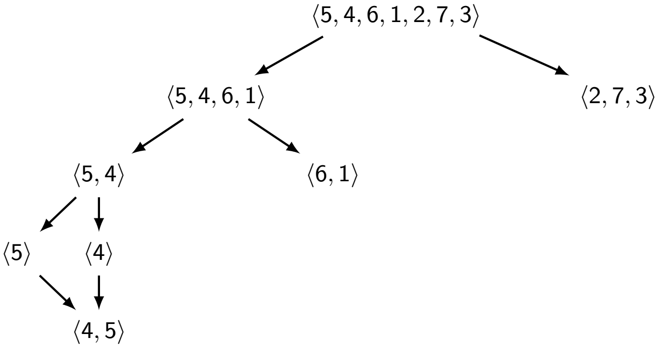
Example: Merge Sort run



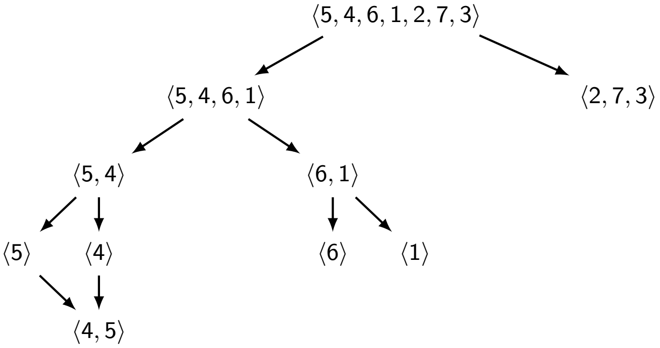
Example: Merge Sort run



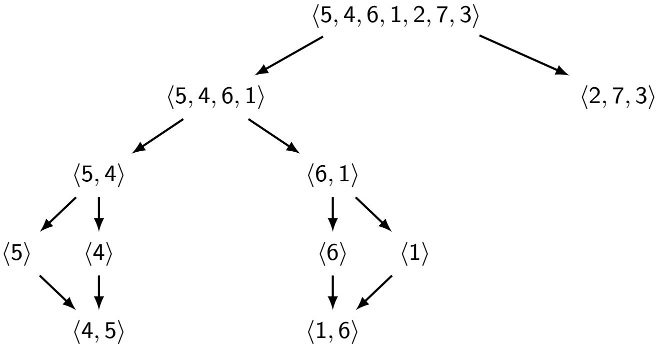
Example: Merge Sort run



Example: Merge Sort run



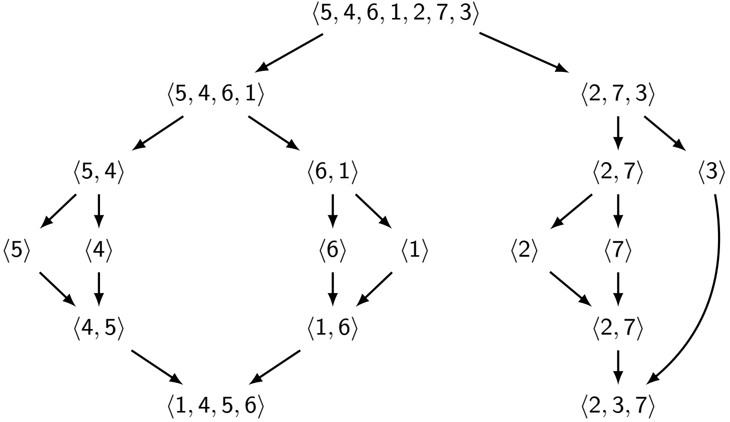
Example: Merge Sort run



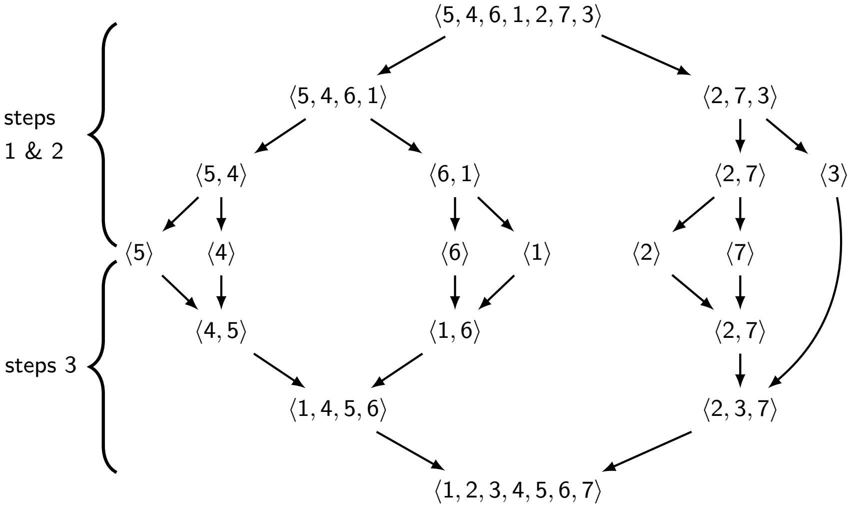
Example: Merge Sort run

$$
\begin{tikzcd}
 & & & & \langle 5,4,6,1,2,7,3\rangle \arrow[dll] \arrow[drr] & & \\
 & & \langle 5,4,6,1\rangle \arrow[dl] \arrow[dr] & & & & \langle 2,7,3\rangle \\
 & \langle 5,4\rangle \arrow[dl] \arrow[d] & & \langle 6,1\rangle \arrow[d] \arrow[drr] & & & \\
\langle 5\rangle \arrow[dr] & \langle 4\rangle \arrow[d] & & \langle 6\rangle \arrow[d] & & \langle 1\rangle \arrow[dll] & \\
 & \langle 4,5\rangle \arrow[dr] & & \langle 1,6\rangle \arrow[dl] & & & \\
 & & \langle 1,4,5,6\rangle & & & &
\end{tikzcd}
$$

Example: Merge Sort run



Example: Merge Sort run

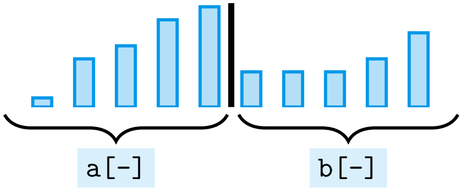


Merging two sorted arrays `a[-]` and `b[-]` efficiently

Idea: In variables `i` and `j` we store the current positions in `a[-]` and `b[-]`, respectively (starting from `i=0` and `j=0`). Then:

1. Allocate a *temporary* array `tmp[-]`, for the result.
2. If `a[i] <= b[j]` then copy `a[i]` to `tmp[i+j]` and `i++`,
3. Otherwise, copy `b[j]` to `tmp[i+j]` and `j++`.

Repeat 2./3. until `i` or `j` reaches the end of `a[-]` or `b[-]`, respectively, and then copy the rest from the other array.



Merging two sorted arrays `a[-]` and `b[-]` efficiently

Merging two sorted arrays is the most important part of merge sort and must be efficient. For example:

Take `a = [1,6,7]` and `b = [3,5]`. Set `i=0` and `j=0`, and allocate `tmp` of length `5`:

1. `a[0]` $\leqslant$ `b[0]`, so set `tmp[0] = a[0]` $(=$ `1`$)$ and `i++`.
2. `a[1]` $>$ `b[0]`, so set `tmp[1] = b[0]` $(=$ `3`$)$ and `j++`.
3. `a[1]` $>$ `b[1]`, so set `tmp[2] = b[1]` $(=$ `5`$)$ and `j++`.

At this point `i` $= 1$, `j` $= 2$ and the first three values stored in `tmp` are `[1,3,5]`.

Since `j` is at the end of `b`, we are done with `b` and we copy the remaining values from `a` into `tmp`. Then, `tmp` stores `[1,3,5,6,7]`.

Merge Sort (pseudocode)

```
mergesort(a, n) {
    mergesort_run(a, 0, n-1)
}

void mergesort_run(a, left, right) {
    if (left < right){
      mid = (left + right) div 2

      mergesort_run(a, left, mid)
      mergesort_run(a, mid+1, right)

      merge(a, left, mid, right)
    }
}
```

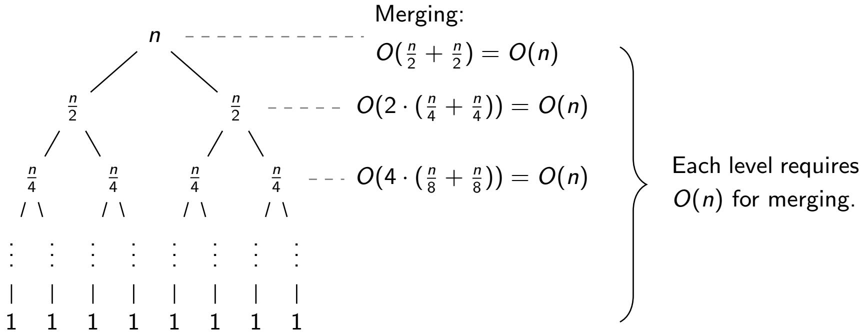
Merging (pseudocode)

```
merge(array a, int left, int mid, int right) {
    create new array b of size right-left+1
    bcount = 0
    lcount = left
    rcount = mid+1
    while ( (lcount <= mid) and (rcount <= right) ) {
        if ( a[lcount] <= a[rcount] )
            b[bcount++] = a[lcount++]
        else
            b[bcount++] = a[rcount++]
    }
    if ( lcount > mid )
        while ( rcount <= right )
            b[bcount++] = a[rcount++]
    else
        while ( lcount <= mid )
            b[bcount++] = a[lcount++]
    for ( bcount = 0 ; bcount < right-left+1 ; bcount++ )
        a[left+bcount] = b[bcount]
}
```

Time Complexity of Mergesort

Merging two arrays of lengths n_1 and n_2 is in $O(n_1 + n_2)$

Sizes of recursive calls:



If $n = 2^k$, then we have $k = \log_2 n$ levels $\implies$ $O(n \log n)$ is the time complexity of merge sort.

(This is the Worst/Best/Average Case complexity.)

Let us analyse the running time of merge sort for an array of size n and for simplicity we assume that $n = 2^k$. First, we run the algorithm recursively for two halves. Putting the running time of those two recursive calls aside, after both recursive calls finish, we merge the result in time $O(\frac{n}{2} + \frac{n}{2})$.

Okay, so what about the recursive calls? To sort $\frac{n}{2}$-many entries, we split them in half and sort both $\frac{n}{4}$-big parts independently. Again, after we finish, we merge in time $O(\frac{n}{4} + \frac{n}{4})$. However, this time, merging of $\frac{n}{2}$-many entries happens twice and, therefore, in total it runs in $O(2 \times (\frac{n}{4} + \frac{n}{4})) = O(2 \times \frac{n}{2}) = O(n)$.

Similarly, we have 4 subproblems of size $\frac{n}{4}$, each of them is merging their subproblems in time $O(\frac{n}{8} + \frac{n}{8})$. In total, all calls of `merge` for subproblems of size $\frac{n}{4}$ take $O(4 \times (\frac{n}{8} + \frac{n}{8})) = O(n)$. ... We see that it always takes $O(n)$ to merge all subproblems of the same size (= those on the same level of the recursion).

Since the height of the tree is $O(\log n)$ and each level requires $O(n)$ time for all merging, the time complexity is $O(n \log n)$. Notice that this analysis does not depend on the particular data, so it is the Worst, Best and Average Case.