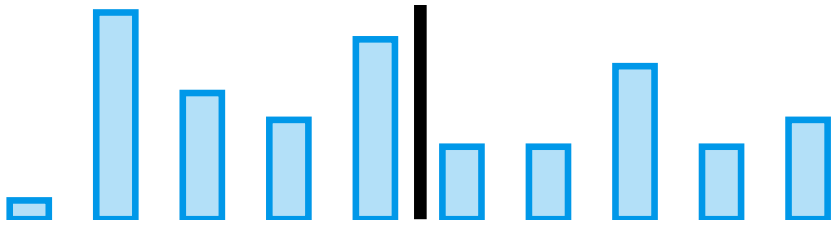
Divide and Conquer Sorting algorithms

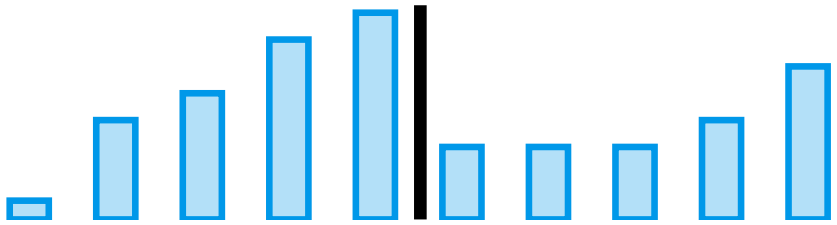
Merge Sort

Idea:

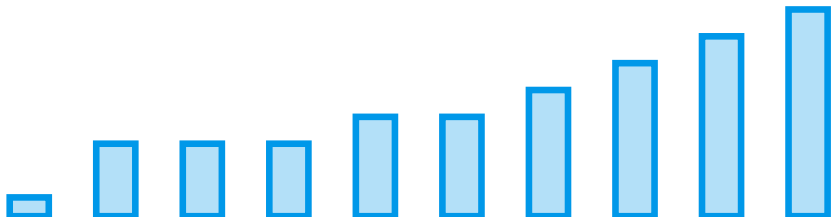
1. Split the array into two halves:



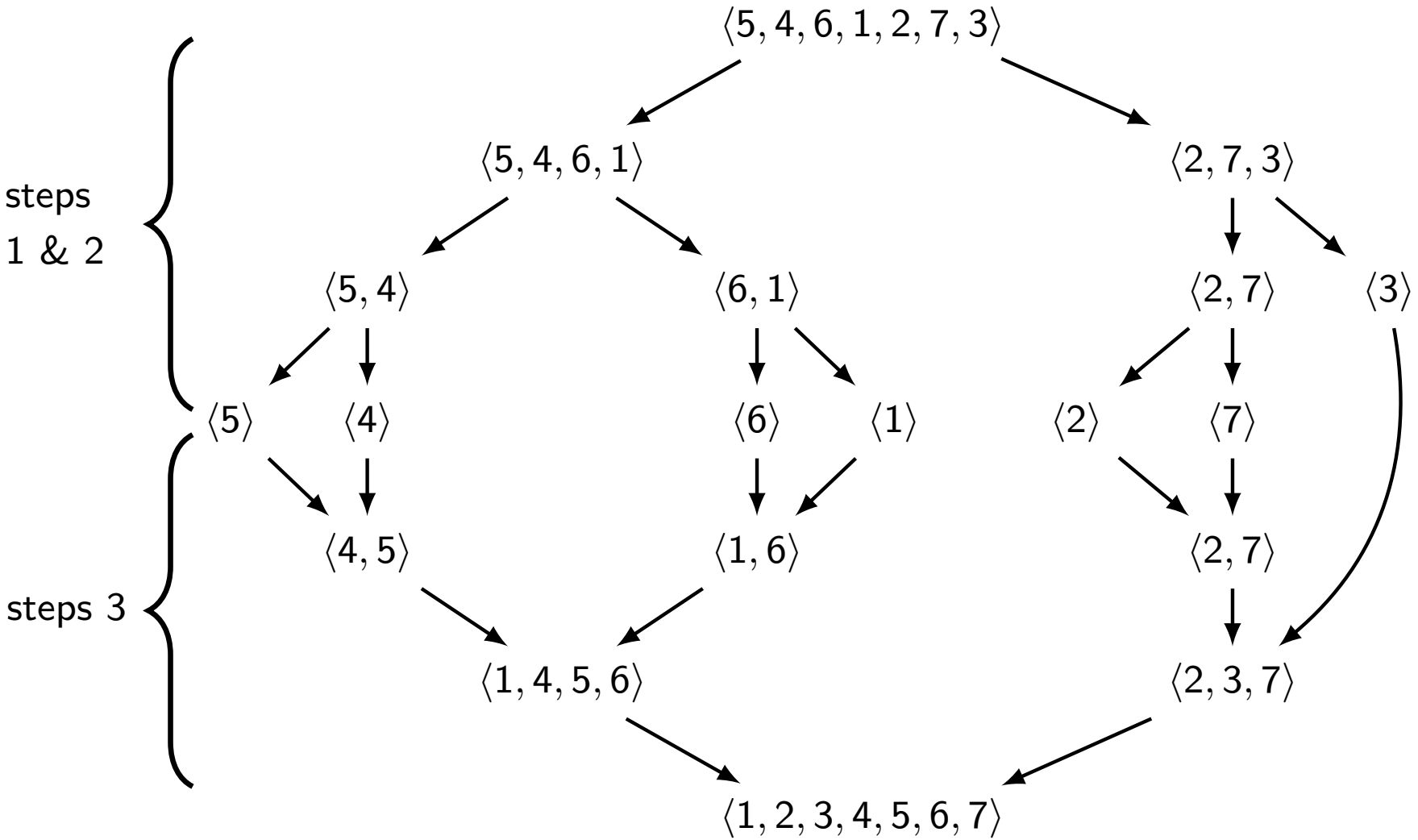
2. Sort each of them recursively:



3. Merge the sorted parts:



Example: Merge Sort run

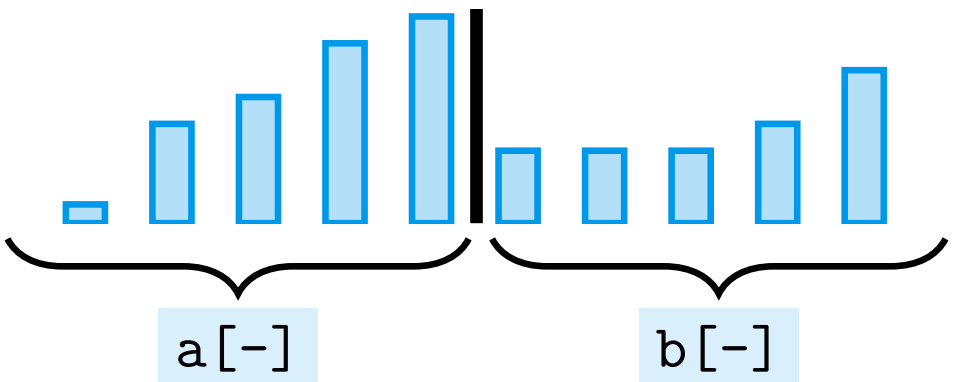


Merging two sorted arrays `a[-]` and `b[-]` efficiently

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 is the most important part of merge sort. Therefore, it is essential to do this operation efficiently.

For example, take `a = [1,6,7]` and `b = [3,5]`. To start with, we set `i=0` and `j=0`, and allocate `tmp` of length `5`. The algorithm will then proceed as follows:

1. Since `a[0] <= b[0]`, set `tmp[0] = a[0]` (= `1`) and `i++`.
2. Since `a[1] > b[0]`, set `tmp[1] = b[0]` (= `3`) and `j++`.
3. Since `a[1] > b[1]`, 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)

```
1  void mergesort(int[] arr) {
2      mergesort_run(arr, 0, arr.length-1);
3  }
4
5  void mergesort_run(int[] arr, int left, int right) {
6      if (right - left <= 1) return;
7
8      int mid = left + right div 2;
9
10     mergesort_run(arr, left, mid);
11     mergesort_run(arr, mid+1, right);
12
13     merge(arr, left, mid, right);
14 }
```

The pseudocode we present here tries to avoid some of the unnecessary allocations of new arrays. Namely, when running recursive calls of merge sort, we do not allocate two new arrays for the two halves, we only compute the `left`-most and `right`-most positions of those halves, with respect to the original array `arr`.

Initially we call `mergesort_run` to sort all elements of the array, that is, we want to sort elements on positions

`0`, `1`, `2`, `3`, ..., `arr.length-1`

In order to sort this, we run merge sort twice, first time for the positions

`0`, `1`, `2`, `3`, ..., `mid`

and the second time for positions

`(mid+1)+0`, `(mid+1)+1`, `(mid+1)+2`, ..., `arr.length-1`.

(In further recursive calls are those `left` and `right` bounds recomputed accordingly.)

Merging (pseudocode)

```
1  void merge(int[] arr, int left, int mid, int right) {
2      int[] tmp = new int[right-left+1];
3      int i = 0;
4      int j = 0;
5
6      while (left+i <= mid && (mid+1)+j <= right) {
7          if (arr[left+i] <= arr[(mid+1)+j]) {
8              tmp[i+j] = arr[left+i];
9              i++;
10         } else {
11             tmp[i+j] = arr[(mid+1)+j];
12             j++;
13         }
14     }
15
16     // Copy the rest          (one of those loops won't run)
17     while (left+i  <= mid)
18         { tmp[i+j] = arr[left+i];   i++; }
19     while ((mid+1)+j <= right)
20         { tmp[i+j] = arr[(mid+1)+j]; j++; }
21
22     // Copy the sorted array in tmp back into arr
23     for (i=0; i<right-left+1; i++)
24         arr[left+i] = tmp[i];
25 }
```

We do not allocate new arrays in recursive calls but only recompute `left` and `right` bounds in array `arr`. Therefore, to `merge` we will be relying on those bounds.

This code assumes that the two halves are stored on positions

`left`, `left+1`, `left+2`, ..., `right`

where the first array occupies positions *until* `mid` and the second array occupies positions *from* `mid+1`.

Consequently, instead of referring to entries of the first array as `a[i]` (as we did on slide 3), we refer to them as `arr[left+i]` and, similarly, `b[j]` translates as `arr[(mid+1)+j]`.

In other words, the values of `a[-]` are the following:

`arr[left]`, `arr[left+1]`, `arr[left+2]`, ..., `arr[mid]`

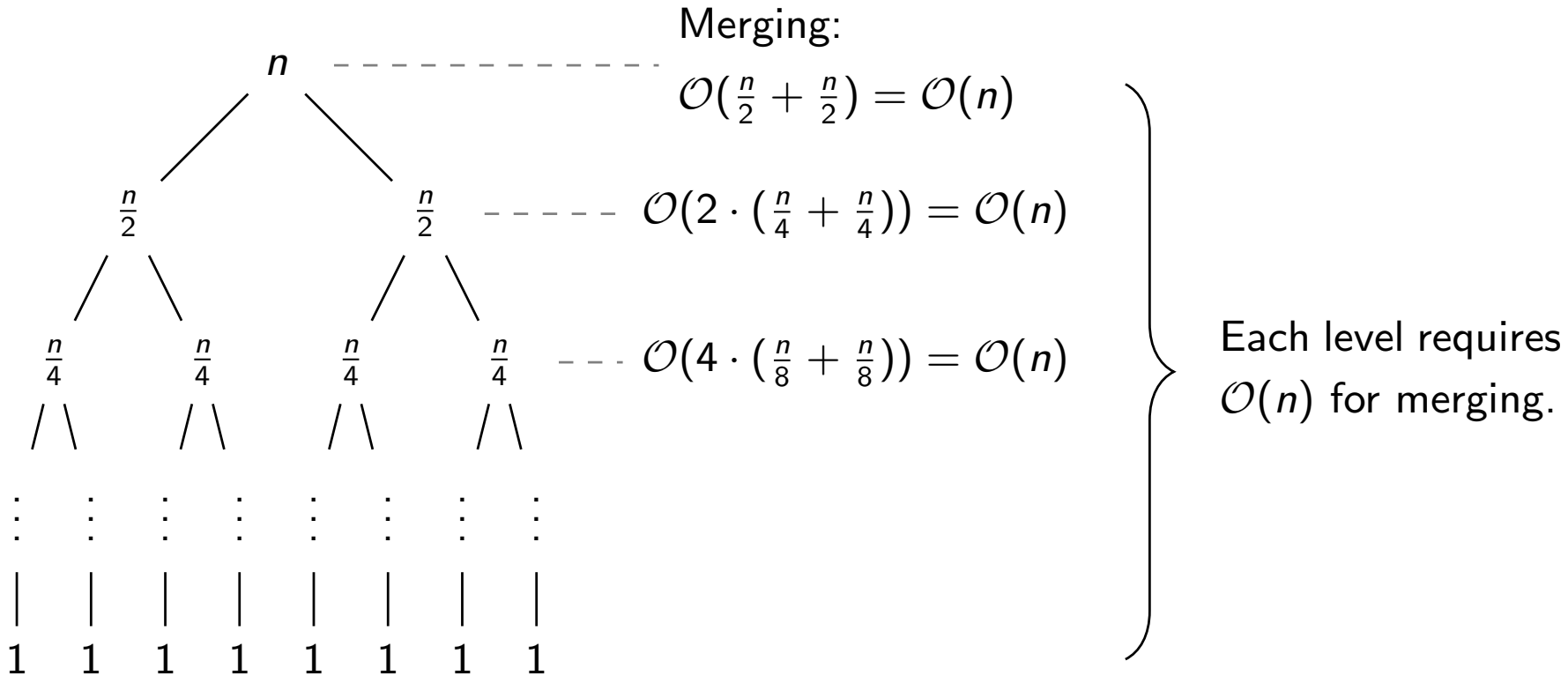
and the values of `b[-]` are

`arr[mid+1]`, `arr[(mid+1)+1]`, `arr[(mid+1)+2]`, ..., `arr[right]`.

Time Complexity of Mergesort

Merging two arrays of lengths n_1 and n_2 is in $\mathcal{O}(n_1 + n_2)$!

Sizes of recursive calls:



If $n = 2^k$, then we have $k = \log_2 n$ levels $\implies$ $\mathcal{O}(n \log n)$ 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 $\mathcal{O}(\frac{n}{2} + \frac{n}{2})$.

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 $\mathcal{O}(\frac{n}{4} + \frac{n}{4})$. However, this time, merging of $\frac{n}{2}$-many entries happens twice and, therefore, in total it runs in $\mathcal{O}(2 \times (\frac{n}{4} + \frac{n}{4})) = \mathcal{O}(2 \times \frac{n}{2}) = \mathcal{O}(n)$.

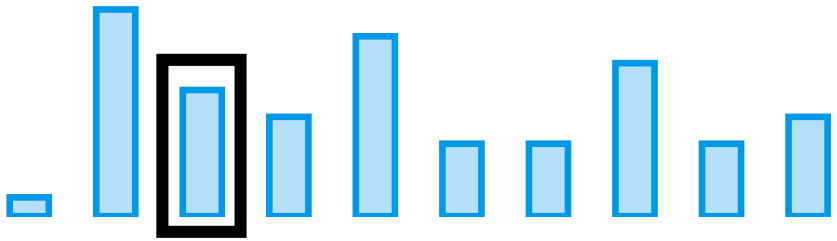
Similarly, we have 4 subproblems of size $\frac{n}{4}$, each of them is merging their subproblems in time $\mathcal{O}(\frac{n}{8} + \frac{n}{8})$. In total, all calls of `merge` for subproblems of size $\frac{n}{4}$ take $\mathcal{O}(4 \times (\frac{n}{8} + \frac{n}{8})) = \mathcal{O}(n)$. ... We see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

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

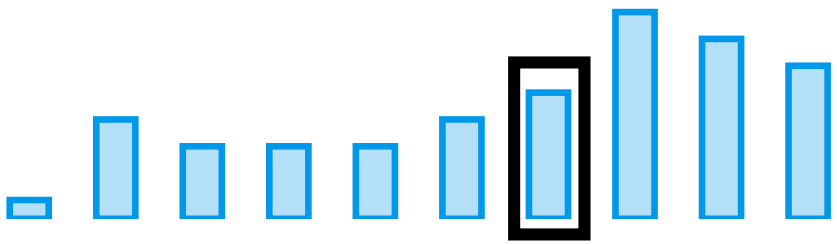
Quick Sort

Idea:

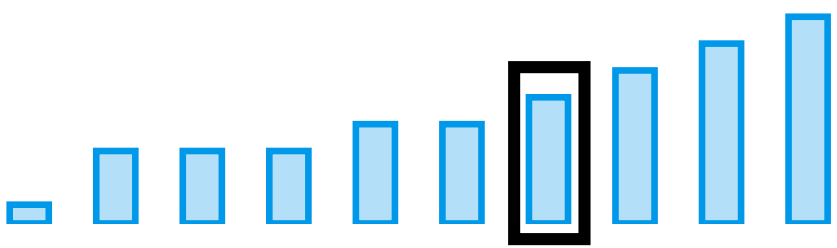
1. Select an element of the array, which we call **pivot**.



2. Partition the array so that the *"small entries"* ($\leq$ pivot) are on the left, then the pivot, then the *"large entries"* ($>$ pivot).



3. Recursively (quick)sort the two partitions.



For the time being it is not important how is the pivot selected. We will see later that there are different strategies which select pivot and they might affect the time complexity of quicksort.

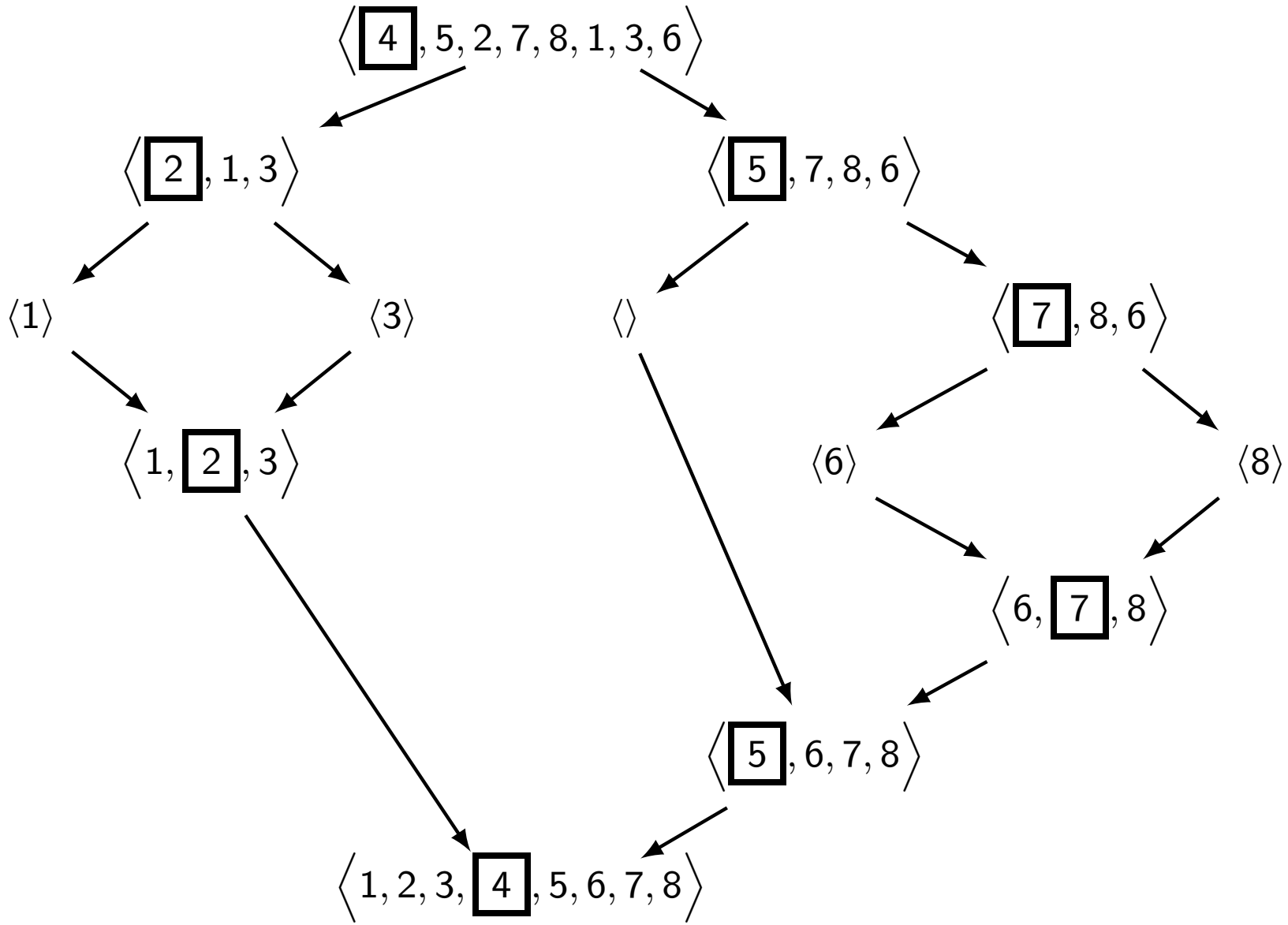
Remark: In order for quicksort to be a *stable* sorting algorithm (see the last slide of week 7 for the definition), it is useful to allow the *large entries* to be also $\geq$ pivot.

On the other hand, it is easier to understand how quicksort works if we require the large entries to be strictly larger than the pivot.

(It does not matter if there are no duplicates in the array.)

Example: Quick Sort run

Pivot selection strategy: we always choose the leftmost entry.



Quick Sort (pseudocode)

```
1  void quicksort(int[] arr){
2      quicksort_run(arr, 0, arr.length-1);
3  }
4
5  void quicksort_run(int[] arr, int left, int right){
6      if (right - left <= 1) return;
7
8      pivot_index = partition(arr, left, right);
9      quicksort_run(arr, left, pivot_index-1);
10     quicksort_run(arr, pivot_index+1, right);
11 }
```

Where `partition` rearranges the array so that

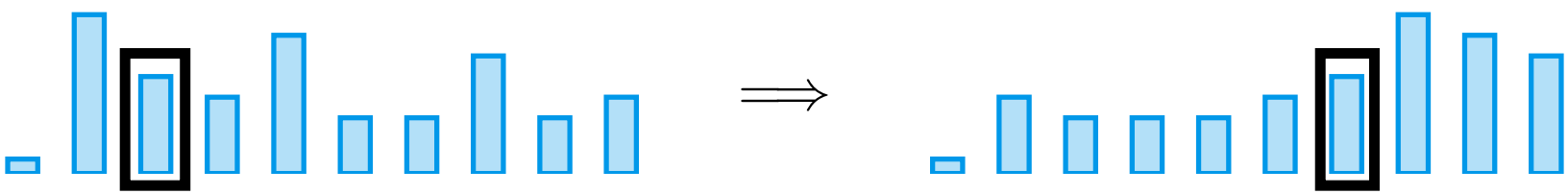
- the small entries are stored on positions `left, left+1, left+2, ..., pivot_index-1`,
- pivot is stored on position `pivot_index` and
- the large entries are stored on `pivot_index+1, pivot_index+2, ..., right`.

Partitioning array `arr`

Idea:

1. Choose a pivot `p` from `arr`.
2. Allocate two temporary arrays: `tmpLE` and `tmpG`.
3. Store all elements *less than or equal to* `p` to `tmpLE`.
4. Store all elements *greater than* `p` to `tmpG`.
5. Copy the arrays `tmpLE` and `tmpG` back to `arr` and return the index of `p` in `arr`.

The time complexity of partitioning is $\mathcal{O}(n)$.



Time Complexity of Quicksort

Best Case: If the pivot is the *median* in every iteration, then the two partitions have approximately $\frac{n}{2}$ elements.

$\Longrightarrow$ The time complexity is as for Merge Sort, i.e. $\mathcal{O}(n \log n)$.

Worst Case: If the pivot is always the *least* element in every iteration, then the second partition contains all elements except for the pivot; it has $n-1$ elements. In the consecutive iterations:

the second partition has $n-1, n-2, n-3, ..., 1$ elements.

$\Longrightarrow$ The time complexity is $\mathcal{O}(n^2)$.

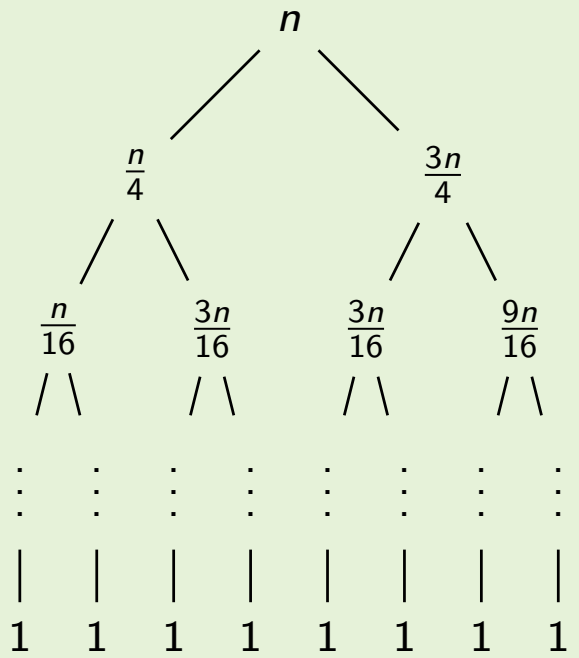
Average Case: Depends on the strategy which chooses the pivots! If there are $\geq 25\%$ many small entries or $\geq 25\%$ many large entries in almost every iteration, then the partitioning happens approximately $\log_{4/3} n$-many times

$\Longrightarrow$ The time complexity is $\mathcal{O}(n \log n)$.

Recall that *median* is the value which you would find exactly in the middle of the array if it was sorted. Therefore, if the pivot is the median, then there are approximately the same number of small and large entries. Consequently, the partitions split the input in half and we can reason similarly as for merge sort to show that Best Case time complexity of quicksort is $\mathcal{O}(n \log n)$. The only difference is that instead of `merge` we call `partition` and, since partitioning n elements runs in $\mathcal{O}(n)$, every level of recursive calls takes $\mathcal{O}(n)$.

For the Average Case time complexity, we assume that the pivot-selecting strategy guarantees that $\geq 25\%$ of entries is small and $\geq 25\%$ is large. Then, although some branches of the tree might be longer than other, it is not difficult to calculate that the height of the tree is again $\mathcal{O}(\log n)$.

(An example of such tree is shown on the right.)



Pivot-selection strategies

Choose pivot as:

1. the middle entry
   (good for sorted sequences, unlike the leftmost-strategy),
2. the median of the leftmost, rightmost and middle entries,
3. use *Quickselect* to choose median in $\mathcal{O}(n)$,
4. a random entry (there is 50% chance for a good pivot).

Remark: In practice, usually 4. or a variant of 2. is used.

Also, for both quicksort and mergesort, when you reach a small region that you want to sort, it's faster to use selection sort or other sort algorithms. The overhead of Q.S. or M.S. is big for small inputs.

Strategies (1) and (2) don't guarantee that the pivot will be such that $\geq 25\%$ entries is small and $\geq 25\%$ is large for *every input* sequence. However, such property is true *on average* (= for a random sequence).

Next, strategy (3) guarantees median in $\mathcal{O}(n)$. Then, selecting the pivot and then partitioning combined is again in $\mathcal{O}(n)$ and so the resulting time complexity of quicksort would still be $\mathcal{O}(n \log n)$.

This strategy is however not used in practice because it is actually slower than just picking the pivot at random (= strategy (4)). Although, we are not guaranteed to have a perfect pivot every single time, we pick it *often* (with 50% probability) which suffices.

Comparison of sorting algorithms

	Selection Sort	Heap Sort	Merge Sort	Quick Sort	Quick Sort 2*
Time Complexity:					
Average C.	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Worst C.	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Space Complexity:					
Average C.	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Worst C.	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Stability	No	No	Yes	Yes	No

"Quick Sort 2" represents a variant of Quick Sort with partitioning done in-place (that is, with no extra temporary arrays).

Recall that a sorting algorithm is **stable** if it doesn't change the respective order of any two elements which are considered equal (see slide 12 of week 7).

The Quick Sort we presented here is stable but it requires $\mathcal{O}(n)$ space for partitioning.

If we are willing to sacrifice stability, there is a variant of Quick Sort which can do partitioning in-place (that is, without extra allocated memory). Then, we can improve the algorithms Average Case *space* complexity to $\mathcal{O}(\log n)$.

So why is quicksort used so much if its Worst Case complexity is as bad as that of selection sort? It is because quicksort's constants hidden by the big-$\mathcal{O}$ are *smaller*. However, if guaranteed $\mathcal{O}(n \log n)$ time complexity is required, it is probably better to use merge sort. Moreover, if we are working with very restricted memory, then it is reasonable to also consider heap sort.

Bonus: in-place partitioning

Invariant: everything to the left of the left barrier `l` is *small*, everything to the right of `r` is *large*.

Three phases of each iteration:

- move `l` as far as possible to the right
- move `r` as far as possible to the left
- swap `a[l]` and `a[r]`

Repeat until `l` and `r` coincide.