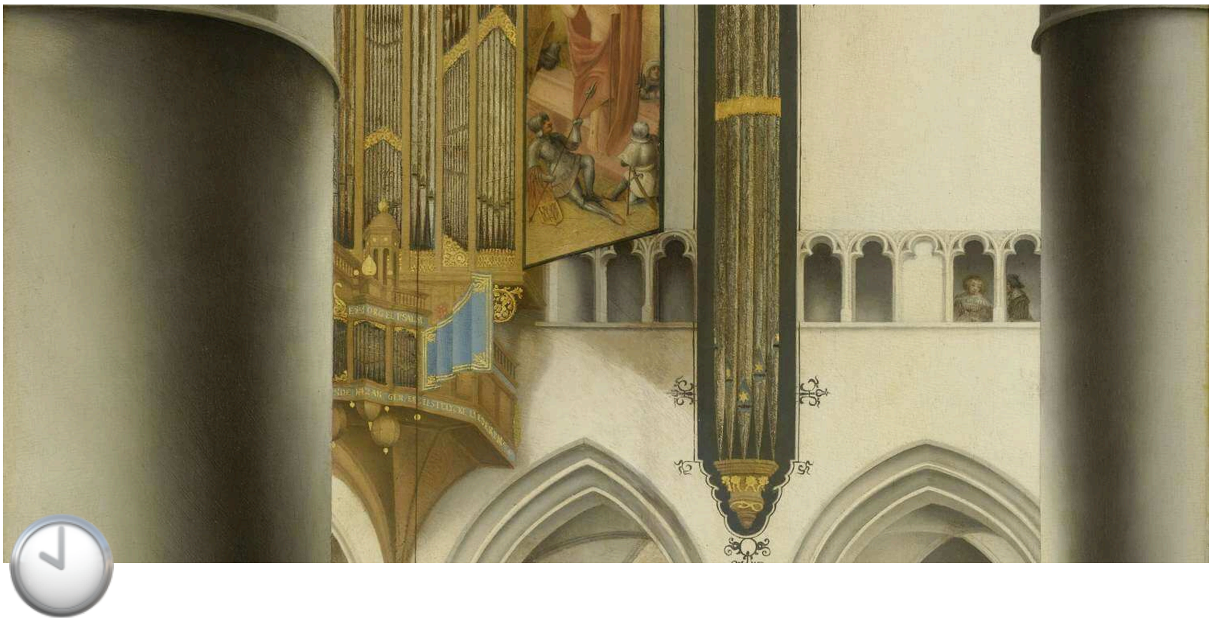


Sorting

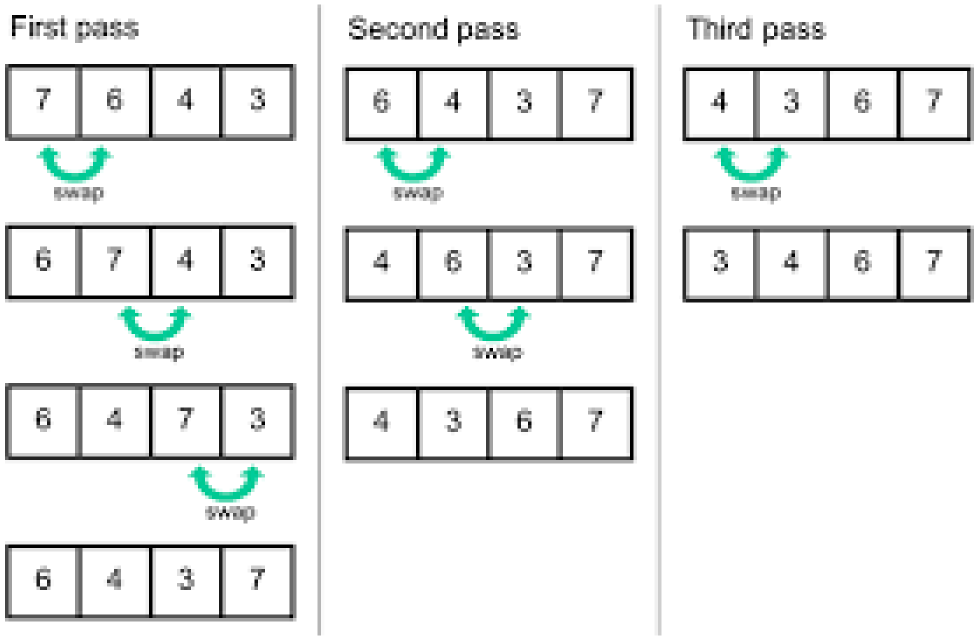
Given a set of records (or objects), we can **sort** them by many criteria. For example, a set of student/mark records that contain marks for different students in different modules could be sorted:

- in decreasing order by mark
- in alphabetic order of their surname first, then by their first name,
    - if there are students with the same surname
- in increasing order of module names, then by decreasing order of mark

Sort algorithms mostly work on the basis of a comparison function that defines the order required between two objects or records.

In some special cases, the nature of the data means that we can sort without using a comparison function.

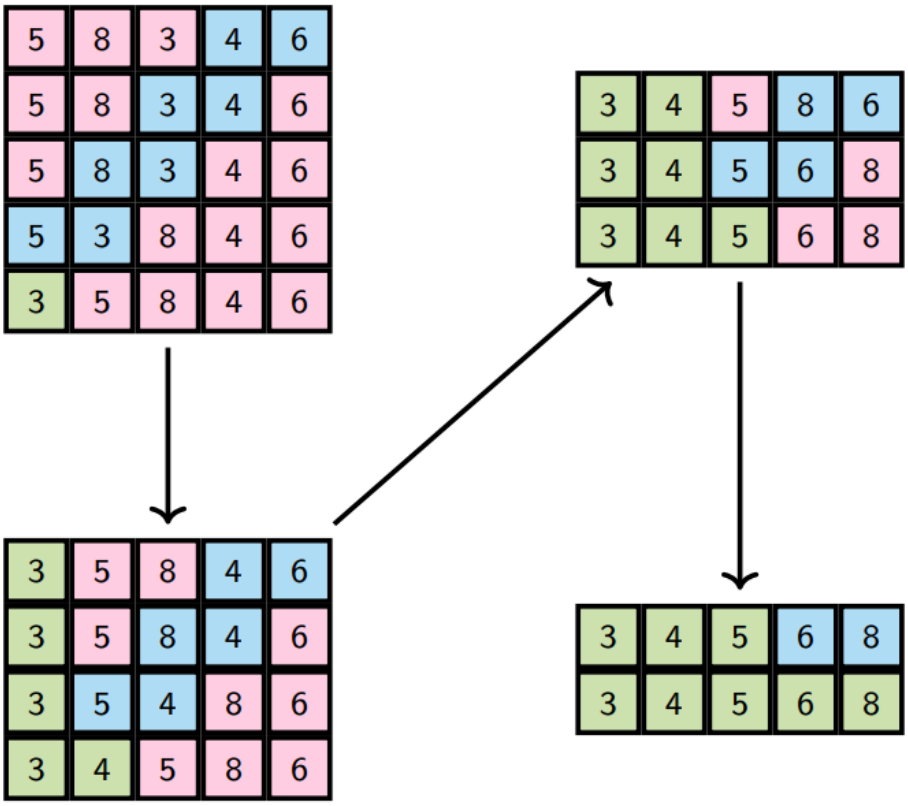
Bubble Sort



- Bubble sort does multiple passes over an array of items, swapping neighbouring items that are out of order as it goes.
- Each pass guarantees that at least one extra element ends up in its correct ordered location at the start of the array
- So consecutive passes shorten to work only on the unsorted part of the array until the last pass only needs to sort the remaining two elements at the end of the array.

```
void bubbleSort(int[] a) {
  for (i = 1; i < a.length; i++)
    for (j = n-1; j >= i; j--)
      if ( a[j] < a[j-1] ) {
        // swap a[j] and a[j-1]
        int temp = a[j]; a[j] = a[j-1]; a[j-1] = temp;
      }
}
```

Example of a Bubble Sort run



5	8	3	4	6
5	8	3	4	6
5	8	3	4	6
5	3	8	4	6
3	5	8	4	6

3	5	8	4	6
3	5	8	4	6
3	5	4	8	6
3	4	5	8	6

3	4	5	8	6
3	4	5	6	8
3	4	5	6	8

3	4	5	6	8
3	4	5	6	8

- *Initial State:*
  - *The algorithm starts with the unsorted grid.*
  - *All numbers are in their initial positions.*
- *Comparisons and Swaps:*
  - *In each step, the algorithm compares adjacent numbers.*
  - *If the number on the left is larger than the one on the right, they are swapped.*
  - *Each box you see represents a step in this process.*
  - *After each pass, the largest number "bubbles" up to its correct position.*
- *Repetition:*
  - *The repetition occurs because the algorithm makes multiple passes through the list.*
  - *In each pass, the same process of comparing and swapping is applied, so the same numbers are checked multiple times.*
- *Sorted State:*

    - *Finally, the grid is sorted in ascending order, with each number in its correct position.*

Bubble Sort Complexity

The outer loop is iterated n −1 times.

The inner loop is iterated n −i times, with each iteration executing a

single comparison.

So the total number of comparisons is:

$$\sum_{i=1}^{n-1}\sum_{j=i}^{n-1} = \sum_{i=1}^{n-1}(n-i)$$

= $(n-1)+(n-2)+.....+1$

= $a=1, l=n-1, m=n-1$

= $S=\frac{m(a+l)}{2} \to S=\frac{(n-1)(1+(n-1))}{2} \to S=\frac{(n-1)n}{2}$

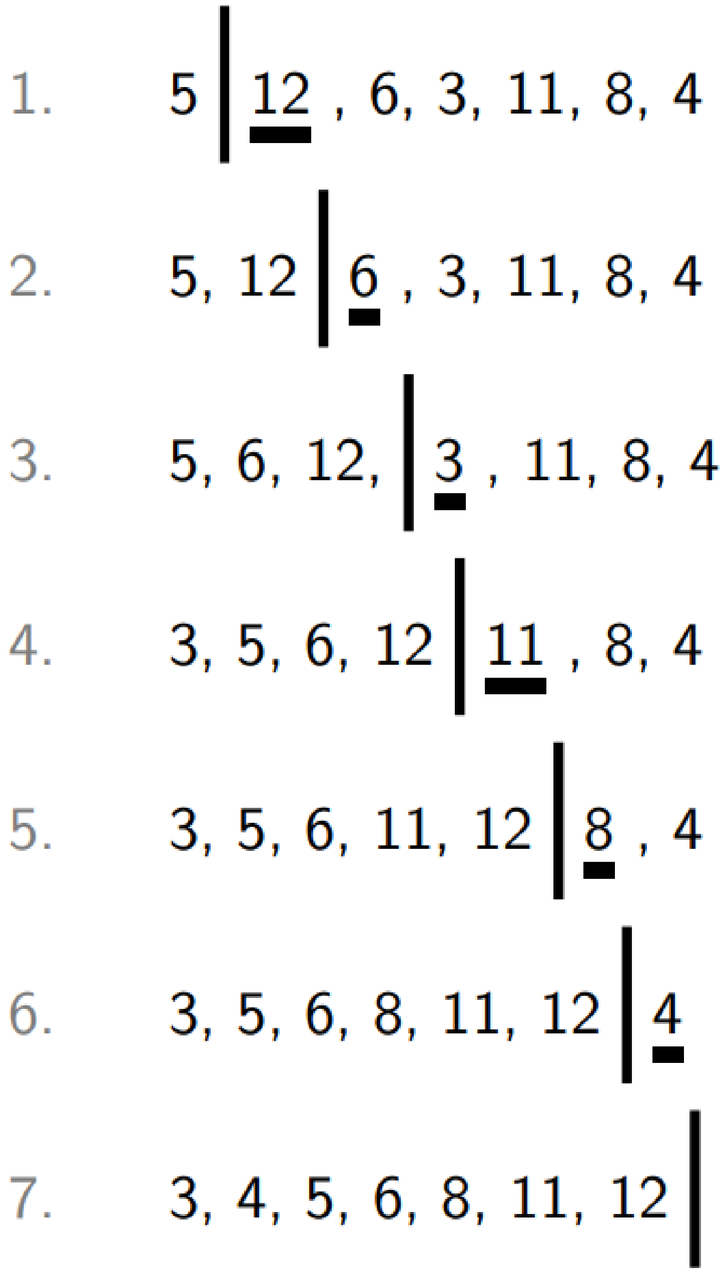
$\to \frac{n(n-1)}{2}$

Thus best, average, and worst-case complexities are all $O(n^2)$

Insertion Sort

- Insertion sort works by taking each element of the input array and *inserting* it into its correct position relative to all the elements that have been inserted so far.
- It does this by partitioning the array into a sorted part at the front and an unsorted part at the end.
- Initially, the sorted part is just the first cell of the array and the unsorted part is the rest.
- In each pass it takes the first element of the unsorted part and inserts it into its correct position in the sorted part, simultaneously growing the sorted part and shrinking the unsorted part by one cell.

Example of an Insertion Sort run



1. 5 | 12 , 6, 3, 11, 8, 4

2. 5, 12 | 6 , 3, 11, 8, 4

3. 5, 6, 12, | 3 , 11, 8, 4

4. 3, 5, 6, 12 | 11 , 8, 4

5. 3, 5, 6, 11, 12 | 8 , 4

6. 3, 5, 6, 8, 11, 12 | 4

7. 3, 4, 5, 6, 8, 11, 12 |

Insertion Sort Complexity

The outer loop is iterated *n* −1 times In the worst case, the inner loop is iterated 1 time for the first outer loop iteration, and 2 times for the 2nd outer iteration, etc.

Insertion Sort has two nested loops: an outer loop and an inner loop.

The outer loop runs n−1n - 1 times, where nn is the number of elements in the array.

In the worst case scenario, the inner loop will have to make more comparisons for every iteration of the outer loop:

1. In the first iteration of the outer loop, the inner loop will compare the first element with the second element. So, it makes 1 comparison.
2. In the second iteration of the outer loop, the inner loop compares the second element with the third element and checks if it needs to be swapped. If not, it will compare the second element with the first element to place it in the correct position. Therefore, it makes 2 comparisons.
3. In the third iteration, the inner loop will make 3 comparisons, and so on.

In general, for the i^{th} iteration of the outer loop, the inner loop makes i comparisons.

Thus, in the worst case, the number of comparisons is:

$\sum_{i=1}^{n-1} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n-1} i$

= $1 + 2 + ... + (n-1)$

$a = 1, l = n-1, m = n-1$

$S = \frac{m(a+l)}{2} \rightarrow S = \frac{(n-1)(1+(n-1))}{2} \rightarrow S = \frac{n(n-1)}{2}$

= $\frac{n(n-1)}{2}$

= $\frac{1}{2}n^2 - \frac{1}{2}n$

The n^2 term will increase quicker and therefore make the $-\frac{1}{2}n$ term insignificant

- In the average case, it is half that
  - because on average the correct position for the insertion in each inner loop will be in the middle of the sorted part), i.e.

Hence average and worst case complexity is at most $Cn^2 = O(n^2)$

Merge Sort

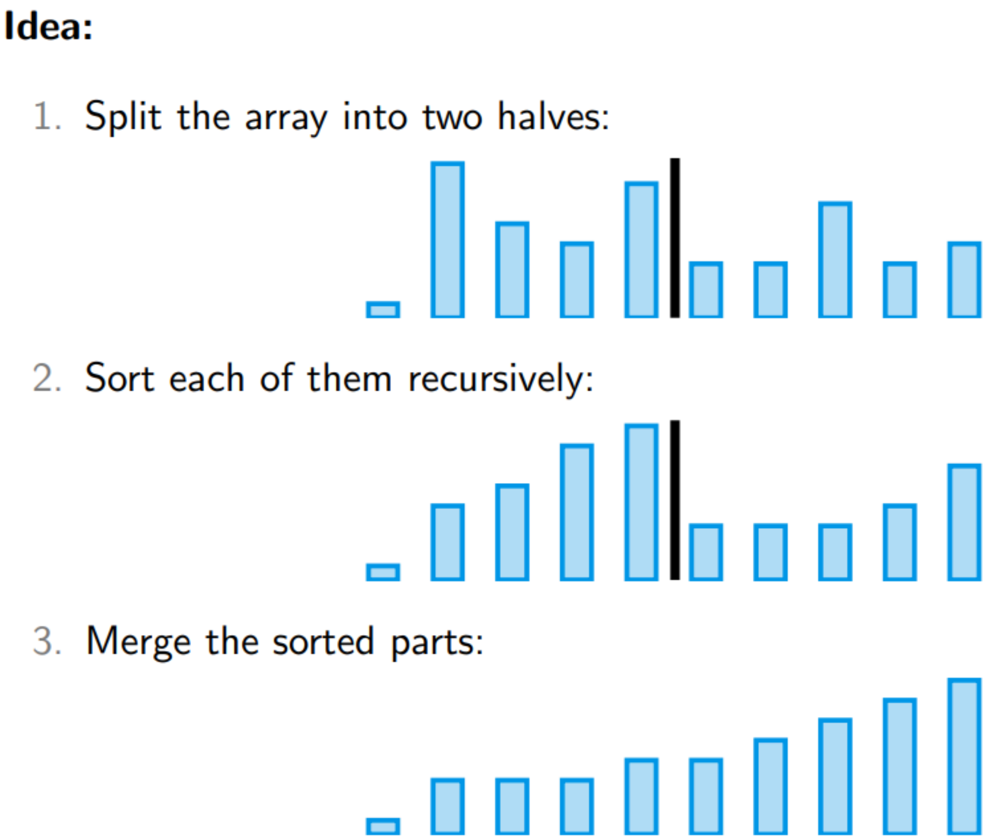
Divide and conquer:

- *Recursively split the problem into smaller subproblems till you are left with much simpler problems*
- *Then put together solutions of these smaller problems into a solution of the big problem*

In the specific case of *Merge sort* we:

- Merge sort is an instance of the use of a *Divide and Conquer*.
- Recursively split the problem into smaller subproblems till you are left with arrays containing only one element.

- Then merge the sorted pieces into bigger and bigger sorted sequences until we get the full array sorted.



Idea:

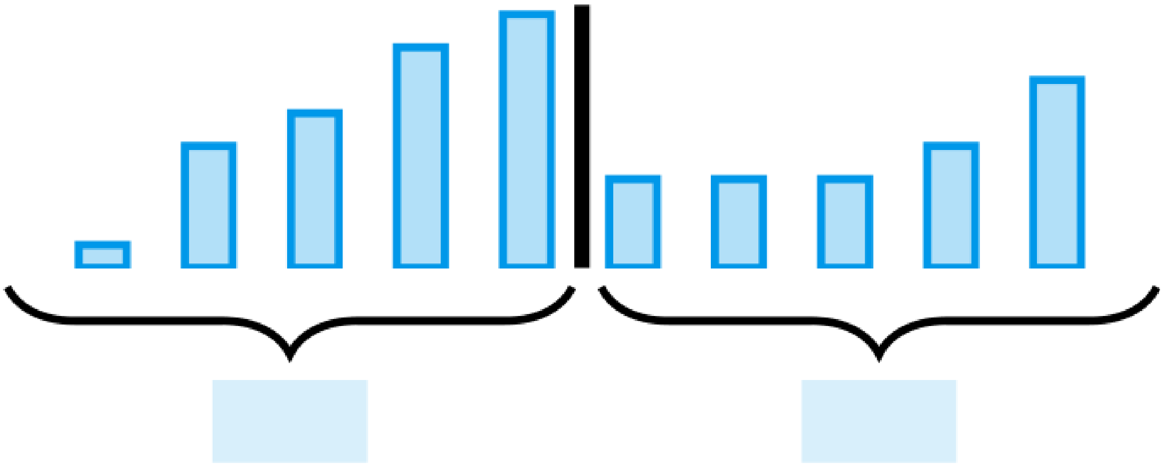
1. Split the array into two halves:
2. Sort each of them recursively:
3. Merge the sorted parts:

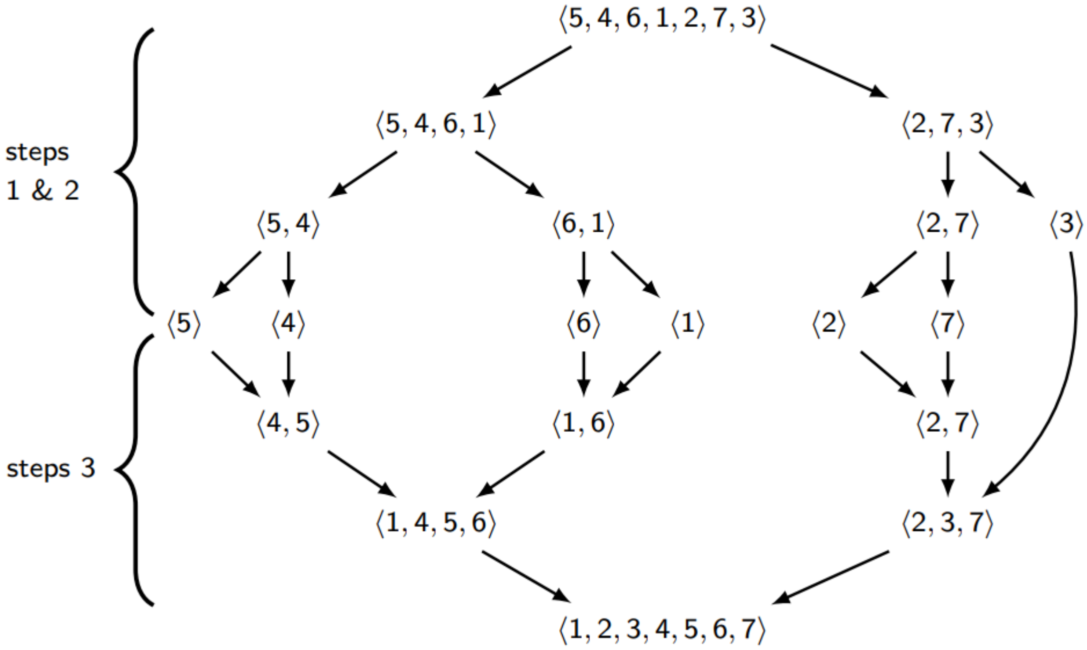
Example: Merge Sort run

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

Idea:

1. In variables i and j we store the current positions in a[-] and b[-], respectively
    a. (starting from i=0 and j=0).
2. Allocate a *temporary* array tmp[-] , for the result.
3. If a[i] <= b[j] then copy a[i] to tmp[i+j] and i++ ,
4. Otherwise, copy b[j] to tmp[i+j] and j++ .
5. Repeat steps 2-4 until i or j reaches the end of a[-] or b[-], respectively
    a. 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$j++$

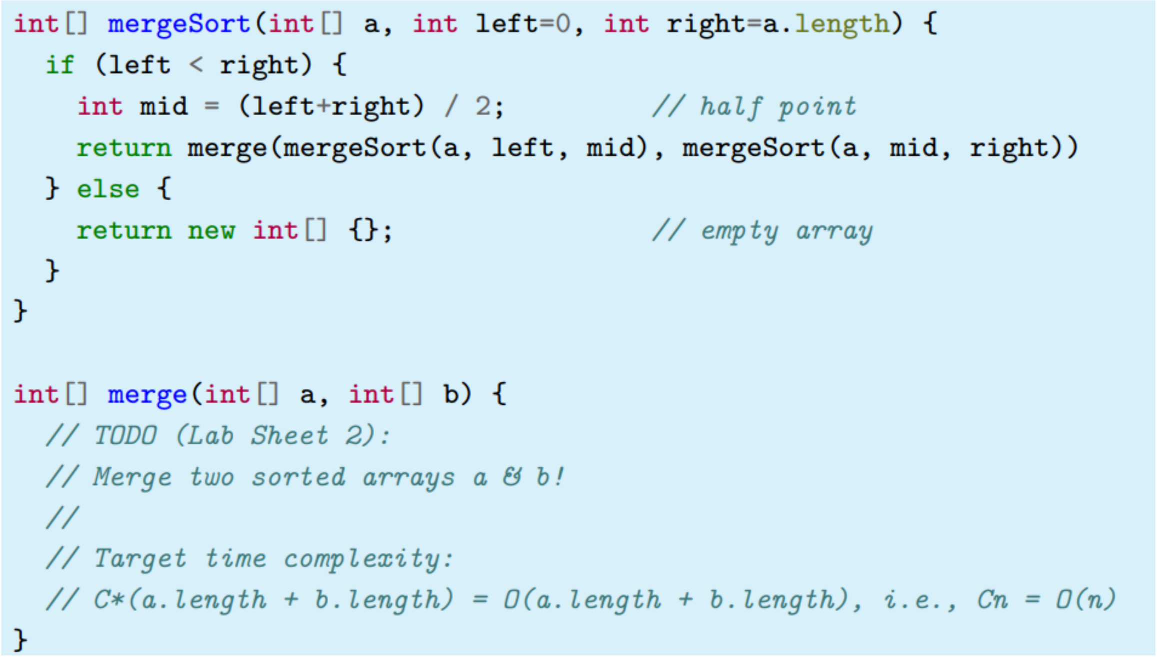
At this point the first three values stored in tmp are $[1, 3, 5]$

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 $[1, 3, 5, 6, 7]$

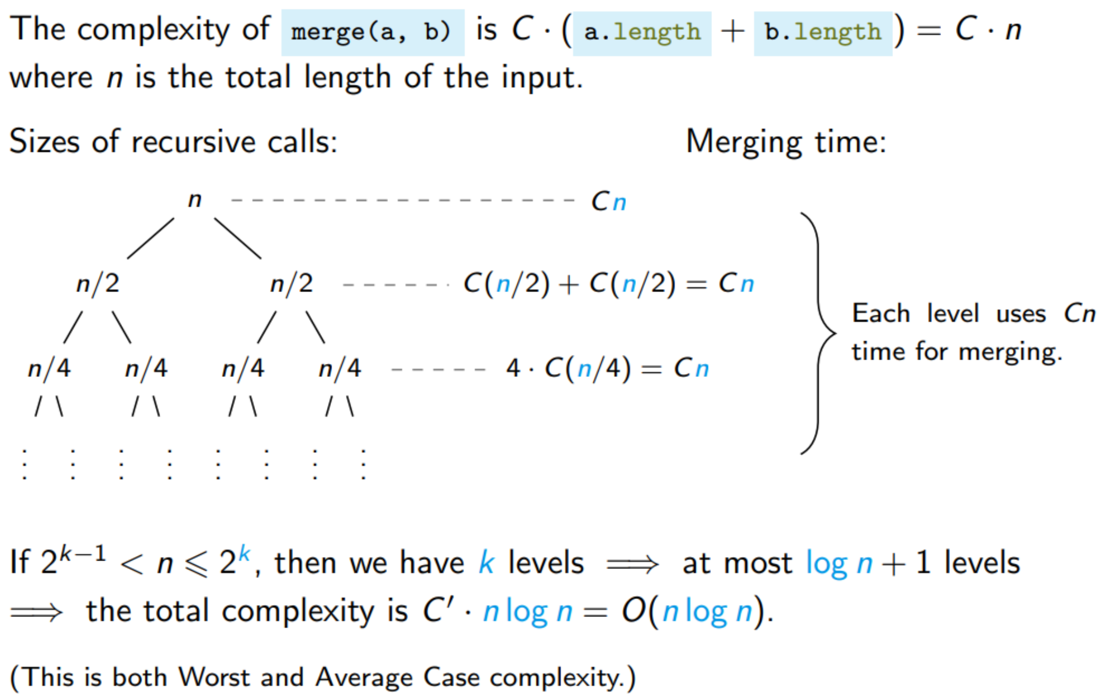
Merge Sort Implementation



```
int[] mergeSort(int[] a, int left=0, int right=a.length) {
  if (left < right) {
    int mid = (left+right) / 2;          // half point
    return merge(mergeSort(a, left, mid), mergeSort(a, mid, right))
  } else {
    return new int[] {};                 // empty array
  }
}

int[] merge(int[] a, int[] b) {
  // TODO (Lab Sheet 2):
  // Merge two sorted arrays a & b!
  //
  // Target time complexity:
  // C*(a.length + b.length) = O(a.length + b.length), i.e., Cn = O(n)
}
```

Time Complexity of Merge sort



The complexity of `merge(a, b)` is $C \cdot ($ `a.length` $+$ `b.length` $) = C \cdot n$ where n is the total length of the input.

Sizes of recursive calls: Merging time:

If $2^{k-1} < n \leqslant 2^k$, then we have k levels $\Longrightarrow$ at most $\log n + 1$ levels

$\Longrightarrow$ the total complexity is $C' \cdot n \log n = O(n \log n)$.

(This is both Worst and Average Case complexity.)

If we say the number of levels as $\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n \Rightarrow \log_2 n + 1$ levels

Merging process takes $O(n)$ time. Therefore since there are $\log_n +1$ levels, the total complexity is $O(nlogn)$

Let us analyse the running time of merge sort for an array of size *n* and simplicity.

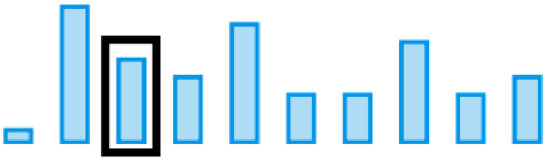
1. We assume that $n = 2^k$.
2. First, we run the algorithm recursively for two halves.
3. Putting the running time of those two recursive calls aside, after both recursive calls finish, we

merge the result in time *O*($\frac{n}{2} + \frac{n}{2}$).

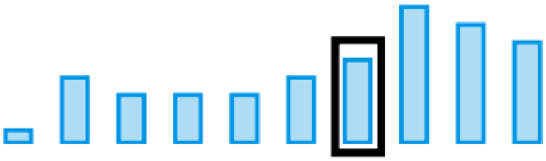
4. Okay, so what about the recursive calls?
5. To sort many entries, we split them in half and sort both $\frac{n}{4}$-big parts independently.
6. Again, after we finish, we merge in time *O*($\frac{n}{4} + \frac{n}{4}$)
7. However, this time, the merging of $\frac{n}{2}$-many entries happens twice and, therefore, in total it runs in *O*(2 × ($\frac{n}{4} + \frac{n}{4}$)) = *O*(2 × $\frac{n}{2}$) = *O*(*n*).
8. Similarly, we have 4 subproblems of size $\frac{n}{4}$, each of them merging their subproblems in time *O*($\frac{n}{8} + \frac{n}{8}$).
9. In total, all calls of merge for subproblems of size $\frac{n}{4}$ take *O*(4 × ($\frac{n}{8} + \frac{n}{8}$)) = *O*(*n*).
10. We see that it always takes *O*(*n*) to merge all subproblems of the same size
    a. those on the same level of the recursion).
11. 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*).
12. Notice that this analysis does not depend on the particular data, so it is the Worst, Best, and Average Case.

Quick Sort

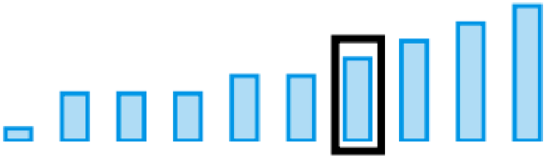
1. Select an element of the array, which we call the **pivot**.



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



3. Recursively (quick)sort the two partitions.

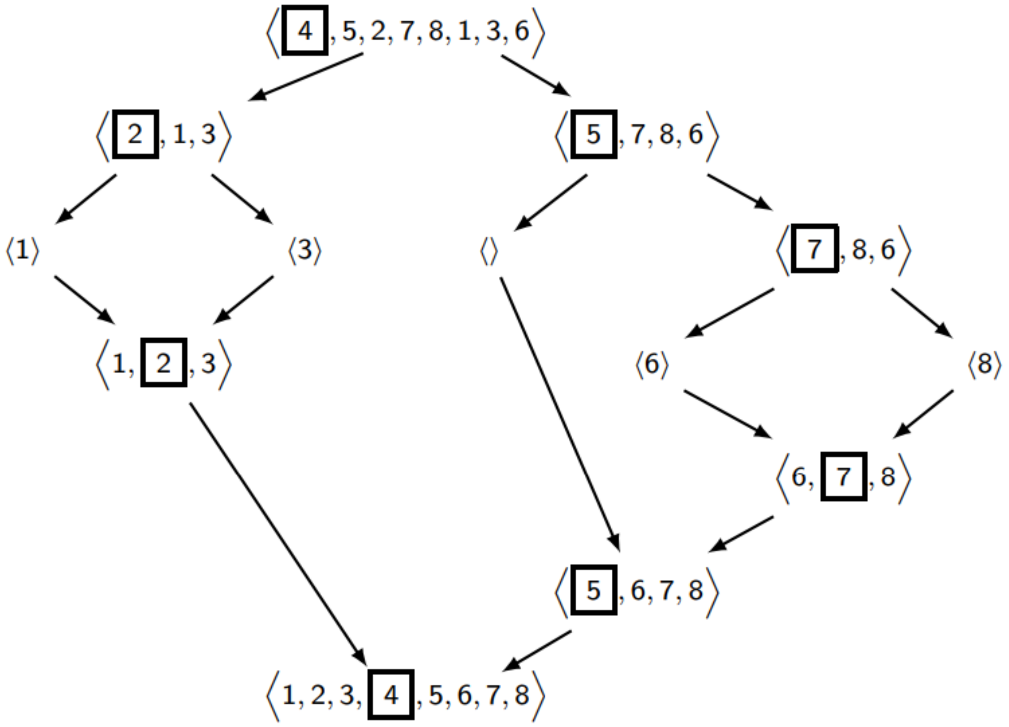


🔥

- ***For the time being it is not important how the pivot is selected.***
- ***We will see later that there are different strategies that select the pivot and they might affect the time complexity of quicksort.***
- ***Remark:***
    - ***For quicksort to be a stable sorting algorithm, it is useful to allow the large entries to also be $\geq$ pivot***
- ***On the other hand, it is easier to understand how quicksort works if we enforce that large entries HAVE to be strictly larger than the pivot.***
- ***Of course, this is only an issue if there are duplicate values in the array.***

Example: Quick Sort run

Initial pivot selection strategy: we always choose the leftmost entry.



This quick sort uses the leftmost value and compares the rest of the value to it, if it's less sends it to the left path, if it's more sends it to the right path.

Quick Sort Implementation of the leftmost entry quicksort solution:

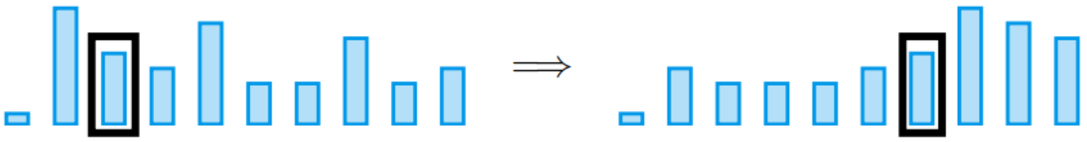
```
void quicksort(int[] a, int left=0, int right=a.length-1){
   if ( left < right ) {
      pivotindex = partition(a, left, right)
      quicksort(a, left, pivotindex-1)
      quicksort(a, pivotindex+1, right)
   }
}
```

Where `partition` rearranges the array so that

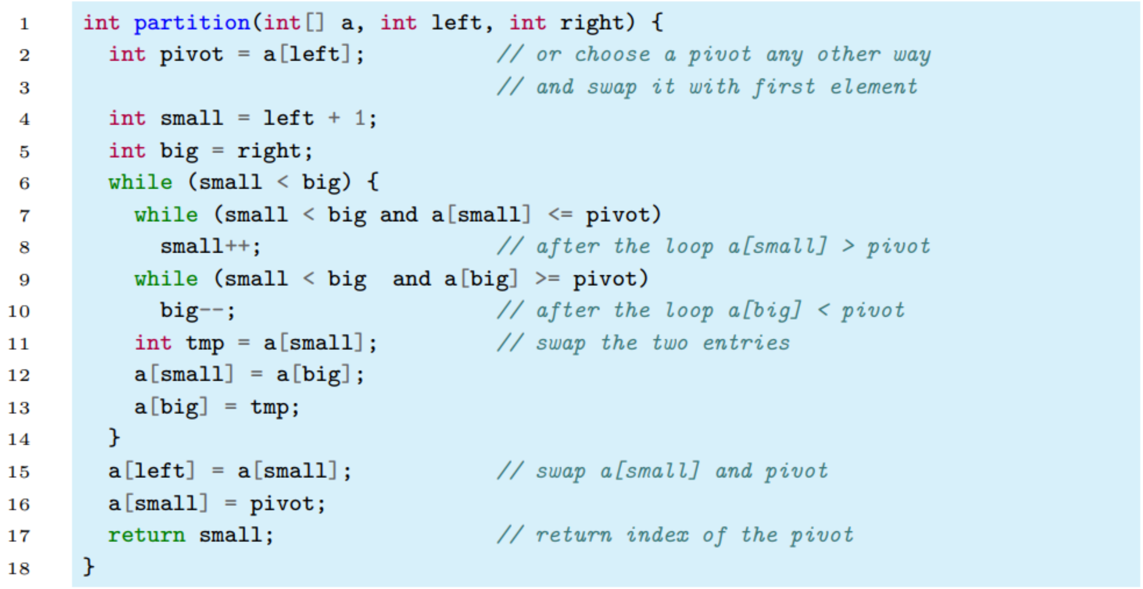
- the small entries are stored on positions `left` , `left+1` , . . . , `pivotindex-1` ,
- pivot is stored on position `pivotindex` and
- the large entries are stored on `pivotindex+1` ,. . . , `right` .

🔥

1. ***Partitioning array** a*
   a. ***Idea:***
      i. Choose a pivot p from a.
      ii. Allocate two temporary arrays: Arr_0 and Arr_2
      iii. Store all elements $\leq$ p to Arr_0
      iv. Store all elements $>$ p to Arr_2 .
      v. Copy the arrays Arr_2 and Arr_2 back to a and return the index of p in a
   b. ***The time complexity of partitioning is O(n).***



Partitioning array `a` in-place



```
int partition(int[] a, int left, int right) {
  int pivot = a[left];          // or choose a pivot any other way
                                // and swap it with first element
  int small = left + 1;
  int big = right;
  while (small < big) {
    while (small < big and a[small] <= pivot)
      small++;                  // after the loop a[small] > pivot
    while (small < big  and a[big] >= pivot)
      big--;                    // after the loop a[big] < pivot
    int tmp = a[small];         // swap the two entries
    a[small] = a[big];
    a[big] = tmp;
  }
  a[left] = a[small];           // swap a[small] and pivot
  a[small] = pivot;
  return small;                 // return index of the pivot
}
```

```
int partition(int[] a, int left, int right) {
```

- The function takes an array `a` and two indices: `left` (start of the partition) and `right` (end of the partition).

Step 1: Choose a Pivot

```
c
CopyEdit
int pivot = a[left];  // Choose the first element as pivot
```

- The pivot is chosen as the first element (`a[left]`).
- The pivot is used to divide the array into two sections:
  - Left side → values **smaller** than the pivot.
  - Right side → values **greater** than the pivot.

Step 2: Initialize Pointers

```
c
CopyEdit
int small = left + 1;
int big = right;
```

- `small`: Moves from left to right, looking for elements **greater than** the pivot.
- `big`: Moves from right to left, looking for elements **smaller than** the pivot.

Step 3: Swap Elements

```
c
CopyEdit
```

```
while (small < big) {
```

- This loop ensures `small` and `big` do not cross each other.

Find a "big" number on the left

```
c
CopyEdit
while (small < big and a[small] <= pivot)
    small++;
```

- `small` moves **right** until it finds an element **greater than** the pivot.

Find a "small" number on the right

```
c
CopyEdit
while (small < big and a[big] >= pivot)
    big--;
```

- `big` moves **left** until it finds an element **smaller than** the pivot.

Swap misplaced elements

```
c
CopyEdit
int tmp = a[small];
a[small] = a[big];
a[big] = tmp;
```

- If `small` and `big` found elements in the wrong places, swap them.

Step 4: Swap Pivot to Correct Position

```
c
CopyEdit
a[left] = a[small];
```

```
a[small] = pivot;
```

- The pivot swaps with `a[small]`, so everything **to the left** is smaller, and everything **to the right** is larger.

Step 5: Return Pivot Index

```
c
CopyEdit
return small;
```

- The function returns the **final position** of the pivot.

Example Walkthrough

Input:

```
c
CopyEdit
a = {5, 8, 1, 3, 7, 9, 2}
left = 0, right = 6
```

Steps:

1. **Pivot = 5**

   Initial: `[5, 8, 1, 3, 7, 9, 2]`

2. Move `small` → stops at `8` (too big)

   Move `big` → stops at `2` (too small)

   Swap **8 and 2** → `[5, 2, 1, 3, 7, 9, 8]`

3. Move `small` → stops at `7`

   Move `big` → stops at `3`

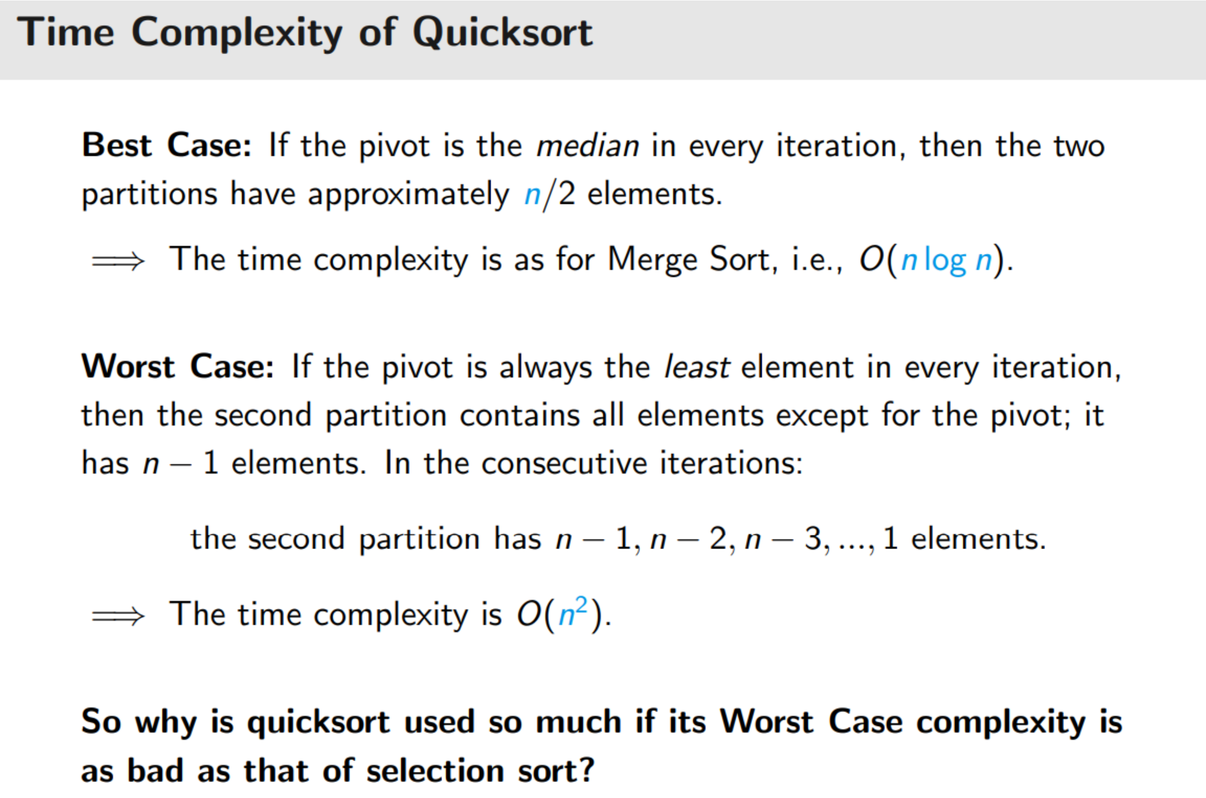
   Swap **7 and 3** → `[5, 2, 1, 3, 7, 9, 8]` (no change)

4. **Swap Pivot (5) and 3**

   `[3, 2, 1, 5, 7, 9, 8]`

5. Pivot is at index **3**, so it returns `3`.

Time Complexity of Quicksort



Time Complexity of Quicksort

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

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

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 $n-1, n-2, n-3, \ldots, 1$ elements.

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

So why is quicksort used so much if its Worst Case complexity is as bad as that of selection sort?

🔥 ***So why is quicksort used so much if its Worst-case complexity is as bad as that of selection sort?***

1. *Quicksort has excellent average-case performance: $O(n \log n)$*
2. *Quicksort is faster in practice compared to Merge Sort due to:*
    - *Better cache locality*
        - *It works in-place*
    - *Fewer memory accesses compared to Merge Sort*
        - *Which needs extra space*
3. *Quicksort is often faster than Merge Sort in real-world scenarios*
    - *Even though both have the same average-case complexity, Quicksort has a smaller constant factor and runs faster in practice.*

The average time complexity of Quicksort

The complexity depends on the strategy which chooses the pivots!

Average Case:

- Assuming that either:
    - We choose pivot randomly
    - That the shuffle is random
        - All permutations of elements are equally likely
- *In practice, the worst-case happens rarely*
    - *The worst case occurs when the pivot always results in the most unbalanced partition (e.g., choosing the smallest or largest element repeatedly).*
    - *Randomized pivot selection or median-of-three strategies make the worst case very unlikely.*
- The average case of the pivot being in the middle 50% of entries is a probability of $\approxeq$ 50%,

  → there are at least 25% many smaller entries and at least 25% many larger entries

  → the partition is, in the worst case, of sizes $\frac{3}{4}n$ and $\frac{1}{4}n$

    - *A good pivot splits the array roughly evenly.*

- *Let's assume the pivot is not perfect, but at least divides the array into 25% and 75% of the elements.*

This means that in the worst case for an average pivot, after each partition:

- *One side has at most $\frac{3}{4}n$ elements.*
- *The other has at most $\frac{1}{4}$ elements.*

→ only $log_{\frac{4}{3}}n = C\log(n)$ recursive calls are required.

- *If each recursive step reduces the problem size from n to at most,* $\frac{3}{4}n$
  - *Then we solve $\log_{4/3} n$ recursive calls until we reach a base case.*
    - *If each recursive step reduces the largest partition to at most $\frac{3}{4}$ of the original array size, then the size of the largest partition follows this pattern:*

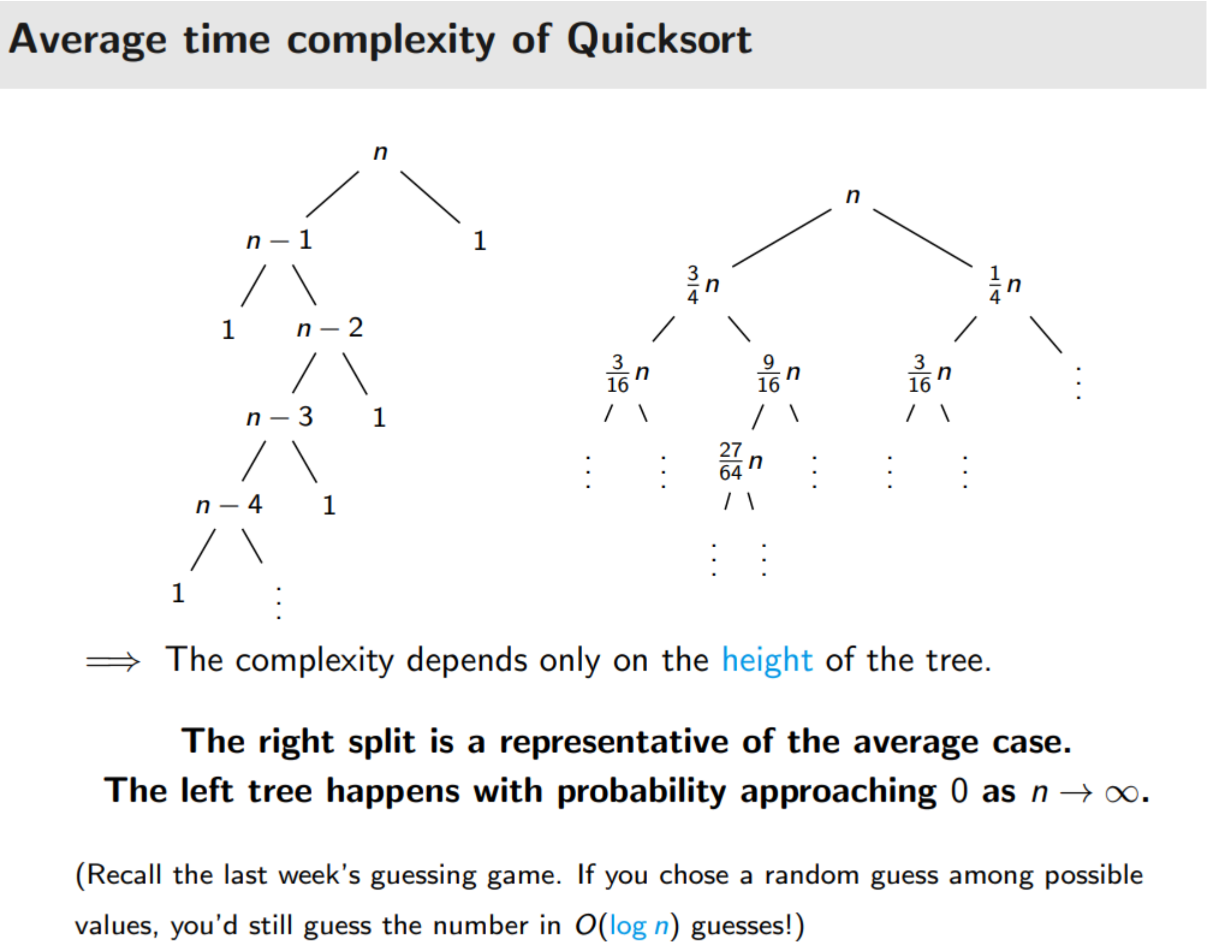
  $n, \ \frac{3}{4}n, \ \left(\frac{3}{4}\right)^2 n, \ \left(\frac{3}{4}\right)^3 n, \ \ldots$

- *The recursion stops when the partition size becomes 1 (base case).*
  - *That means we need to solve:*
    - $n * \left(\frac{3}{4}\right)^k = 1$
  - *Solving for k:*
    - $\left(\frac{3}{4}\right)^k = \frac{1}{n}$
  - *Taking log on both sides:*
    - $k\log\left(\frac{3}{4}\right) = \log\left(\frac{1}{n}\right)$
    - $k = \frac{\log\frac{1}{n}}{\log\frac{3}{4}} * \frac{-1}{-1} = \frac{-1 * \log\frac{1}{n}}{-1 * \log\frac{3}{4}} = \frac{-1 * -\log n}{-1 * \log\frac{3}{4}}$
    - $k = \frac{\log n}{\log\frac{4}{3}}$

→ the time complexity is $O(nlog(n))$

- *Approximation:*
  - *Since logarithms differ only by constant factors, we can write:*
    - $k = O(\log_{\frac{4}{3}} n)$

    - *Since $\log_{\frac{4}{3}} n = \frac{\log n}{\log \frac{4}{3}}$, this is still $O(\log n)$ complexity*
- *The Total Work Done:*
    - *Each level of recursion processes all n elements once.*
    - *Since there are $O(\log n)$ recursive levels, the total work is:*
        - $O(n) \times O(\log n) = O(n \log n)$



Average time complexity of Quicksort

$\Longrightarrow$ The complexity depends only on the height of the tree.

The right split is a representative of the average case.
The left tree happens with probability approaching 0 as $n \to \infty$.

(Recall the last week's guessing game. If you chose a random guess among possible values, you'd still guess the number in $O(\log n)$ guesses!)

Pivot-selection strategies

1. **Choose pivot as:**
    a. The middle entry
        i. good for sorted sequences, unlike the leftmost-strategy
    b. The median of the leftmost, rightmost, and middle entries,
    c. a random entry

        i. there is ⩾a 50% chance for a good pivot

2. **Remark:**
    a. In practice, usually 3. or a variant of 2. is used.
    b. Also, for both quicksort and mergesort
        i. When you reach a small region that you want to sort, It's faster to use selection sort or other sort algorithms.
    c. The overhead of quicksort or mergesort is big for small inputs.
    d. Strategies (1) and (2) don't guarantee that at least 25% of entries are smaller than the pivot and at least 25% is larger than the pivot for *EVERY input* sequence.
        a. However, this property holds *on average*
            a. i..e for a random sequence
    e. Strategy (3), although does not guarantee that we will find a perfect pivot every single time, we pick it *often* (with 50% probability) which suffice

Lower bounds on the complexity of sorting:

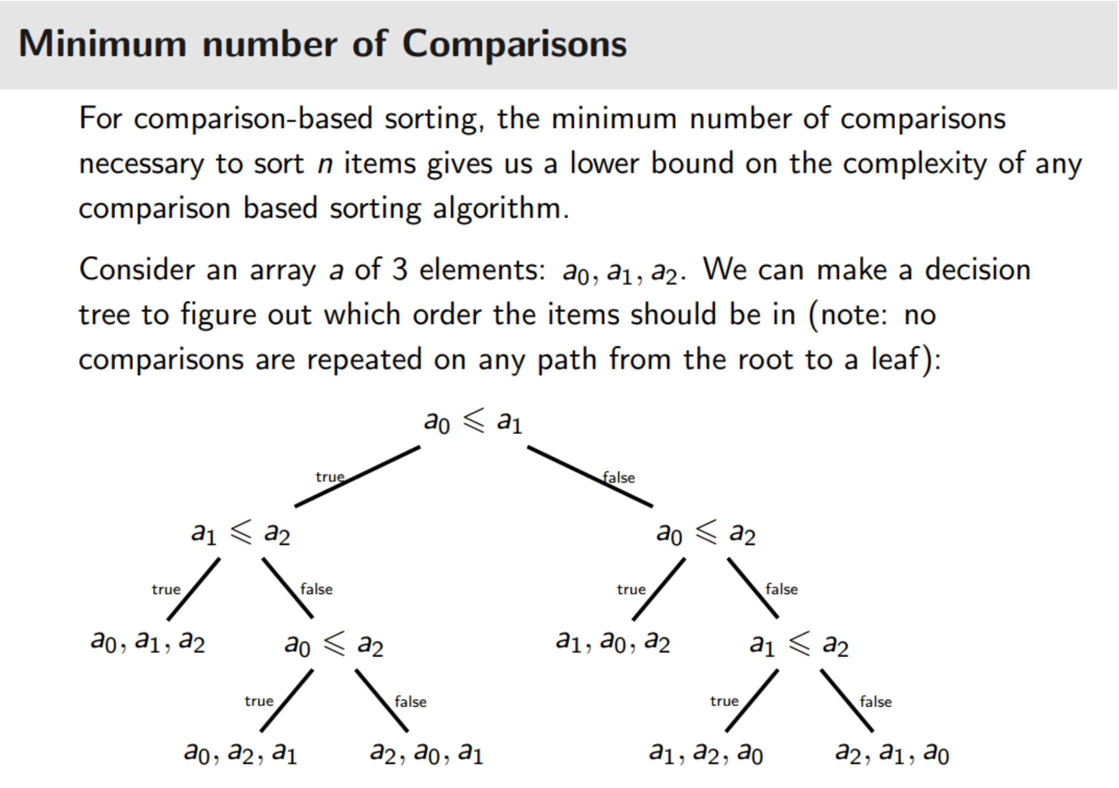
Comparison based strategies

- All algorithms we described so far are comparison-based.
- This means they do not depend on *what we are sorting*
    - but they only compare elements x and y :
    - This way, we may compare $int, float, str(alphabetically), char(\text{by their ASCII value etc } \ldots)$

Comparing objects in Java

- Java provides two interfaces to implement comparison functions:
    - Comparable:
        - A Comparable object can compare itself with another object using its $compareTo(\ldots)$ method.
            - There can only be one such method for any class
                - So this should implement the default ordering of objects of this type.

        - $x.compareTo(y)$ should return
            - a negative int if
                - $x < y$
            - 0 if
                - $x = y$
            - or a positive int if
                - $x > y$
    - Comparator:
        - A comparator object can be used to compare two objects of some type using the $compare(\dots)$ method
            - This does not work on the current object but rather both objects to be compared are passed as arguments.
                - You can have many different comparison functions implemented this way.
        - $compare(x, y)$ should return a
            - negative int if
                - $x < y$
            - 0 if
                - $x = y$
            - or a positive int if
                - $x > y$



Minimum number of Comparisons

For comparison-based sorting, the minimum number of comparisons necessary to sort n items gives us a lower bound on the complexity of any comparison based sorting algorithm.

Consider an array a of 3 elements: a_0, a_1, a_2. We can make a decision tree to figure out which order the items should be in (note: no comparisons are repeated on any path from the root to a leaf):

- ***This decision tree is a binary tree where there is one leaf for every possible ordering of the items.***
  - ***The average number of comparisons that are necessary to sort the items***
    - ***Is the average path length from the root to a leaf of the decision tree***
  - ***The worst-case number of comparisons that are necessary to sort the items***
    - ***Is the the height of the decision tree***
  - *Given n items*
    - *There are n ways to choose the first item*
    - *$n - 1$ ways to choose the second*
    - *$n - 2$ ways to choose the third, $etc\ldots$*
    - *So there are $n(n-1)(n-2)....(3)(2)(1)$*
      - *$n!$ different possible orderings of n items*
  - *Thus the minimum number of comparisons necessary to sort n items is the height of a binary tree with $n!$ leaves*

Minimum number of Comparisons

A binary tree of height h has the most number of leaves if all the leaves are on the bottom-most level, thus it has at most 2^h leaves.

Hence we need to find h such that

$$2^h \geqslant n! \quad \Longrightarrow \quad \log 2^h \geqslant \log n! \quad \Longrightarrow \quad h \geqslant \log n!$$

But

$$\begin{aligned} \log n! &\geqslant \log(\underbrace{n \cdot (n-1) \cdots (n/2)}_{n/2} \cdot (n/2-1) \cdots 1) \\ &\geqslant \log(n/2)^{n/2} = (n/2)\log(n/2) \geqslant C n \log n \end{aligned}$$

Thus we need at least $Cn \log n$ comparisons, for some $C > 0$, to complete a comparison based sort in general.

(You might be tempted to write that we need at least $O(n \log n)$ comparisons, but O is an upper bound! Here we would need to use its 'lower bound brother' $\Omega(n \log n)$.)

🔥

Minimum number of Comparisons

- A binary tree of height h has the most number of leaves if all the leaves are on the bottom-most level
  - Thus it has at most 2^h leaves

◇ Hence we need to find h such that

$$\Rightarrow 2^h \geq n! \\ \text{Take logs on both sides:} \\ \Rightarrow \log 2^h \geq \log n! \\ \Rightarrow h \log 2 \geq \log n! = h \geq \frac{\log n!}{\log 2} \\ \text{ Which simplifies big O notation wise to:} \\ \Rightarrow h \geq \log n! \\ \text{Due to constants being inconsequential in this type of notation} \\ \text{But} \\ \Rightarrow \log n! \geq \underbrace{\log(n \cdot (n-1) \ldots (\frac{n}{2})}_{\frac{n}{2}} \cdot (\tfrac{n}{2} - 1) \ldots 1) \\ \geq \log(\tfrac{n}{2})^{\frac{n}{2}} = (\tfrac{n}{2}) \log(\tfrac{n}{2}) \geq Cn \log n$$

Explanation of the line

Since each term in the second half is at most $\frac{n}{2}$, the entire second half product is smaller than:

$$(\tfrac{n}{2})^{\frac{n}{2}}$$

Taking the log:

$$\log((\tfrac{n}{2} - 1) \cdots 1) \leq \tfrac{n}{2} \log(\tfrac{n}{2})$$

$$\therefore$$

$$\log n! \geq \log(n \cdot (n-1) \cdots (n/2))$$

Explanation of :

$\log((\frac{n}{2} - 1) \cdots 1) \leq \frac{n}{2} \log(\frac{n}{2}) \therefore \log n! \geq \log(n \cdot (n - 1) \cdots (n/2))$:

Writing the Full Factorial Expression

By definition:

$$n! = (n \cdot (n-1) \cdots (\tfrac{n}{2})) \cdot ((\tfrac{n}{2} - 1) \cdots 1)$$

Taking the logarithm on both sides:

$$\log n! = \log(n \cdot (n-1) \cdots (\frac{n}{2})) + \log((\frac{n}{2} - 1) \cdots 1)$$

This equation is exact—it accounts for every term in $n!$

Applying the Inequality

From earlier, we established:

$$\log((\frac{n}{2} - 1) \cdots 1) \leq \frac{n}{2} \log(\frac{n}{2})$$

Substituting this into our equation:

$$\log n! = \log(n \cdot (n-1) \cdots (\frac{n}{2})) + \log((\frac{n}{2} - 1) \cdots 1)$$

$$\log n! \geq \log(n \cdot (n-1) \cdots (\frac{n}{2})) + \frac{n}{2} \log(\frac{n}{2})$$

Since we are **lower bounding** $\log n!$, we can **drop the second term** because it's positive, giving:

$$\log n! \geq \log(n \cdot (n-1) \cdots (n/2))$$

> Thus we need at least $Cn \log n$ comparisons, for some $C > 0$, to complete a comparison based sort in general.
>
> You might be tempted to write that we need at least $O(n \log n)$ comparisons, but O is an upper bound! Here we would need to use its 'lower bound brother' $(nlogn)$

Stability in Sorting

- A *stable* sorting algorithm does not change the order of items in the input if they have the same sort key.
    - In other words if the values are the same the relative order to each other are maintained
        - Thus if we have a collection of student records that is already in order by the student’s first names, and we use a stable sorting algorithm to sort it by students’ surnames,

        - then all students with the same surname will still be sorted by their first names.

1. Using stable sorting algorithms in this way, we can "*pipeline*" sorting steps to construct a particular order in stages.
    a. We can utilise the "same sort key" trick to our advantage
2. In particular, a stable sorting algorithm is often faster when applied to an already sorted, or nearly sorted list of items.
    a. If your input is usually nearly sorted, then you may be able to get higher performance by using a stable sorting algorithm
        i. However, many stable sorting algorithms have higher complexity than unstable ones,
            1. so the complexities involved should be carefully checked.
        ii. Trading lower complexity for higher performance

DIFFERENT SORTING TECHNIQUES STABLITIES ANALYSED:

Bubble Sort Stability

- Consider what happens when two elements with the same value are in the array to be sorted.
- Since only neighbours of values can be swapped, and the swap is only carried out if one is strictly less than the other, no pair of the same values will ever be swapped.
- Hence bubble sort can not change the relative order of two elements with the same value.

Hence bubble sort is *stable*.

Insertion Sort Stability

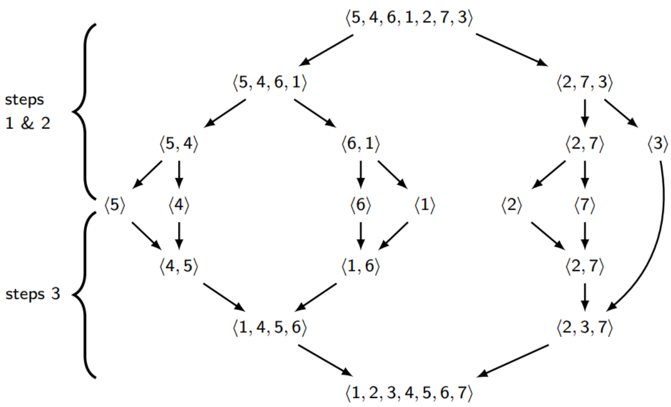
- Consider what happens when two elements with the same value are in the array to be sorted.
- Since the value inserted in each outer loop is the first value in the unsorted part of the array, the first occurrence of two copies of the same value will be taken for insertion before the second.

- Further, in the inner loop, we walk down from the end of the sorted part of the array until we find the first location that is not ***strictly greater*** than the value to be inserted before inserting there.
- That means that we will not insert a later copy of a value before an earlier copy that has already been inserted.

  Hence insertion sort is *stable*.

Stability of Merge sort

- The splitting phase of merge sort does not change the order of any items.
- So long as the merging phase merges the left with the right in that order and takes values from the leftmost sub-array before the rightmost one when values are equal



  - then different elements with the same values do not change their relative order.

  Therefore merge sort is stable.

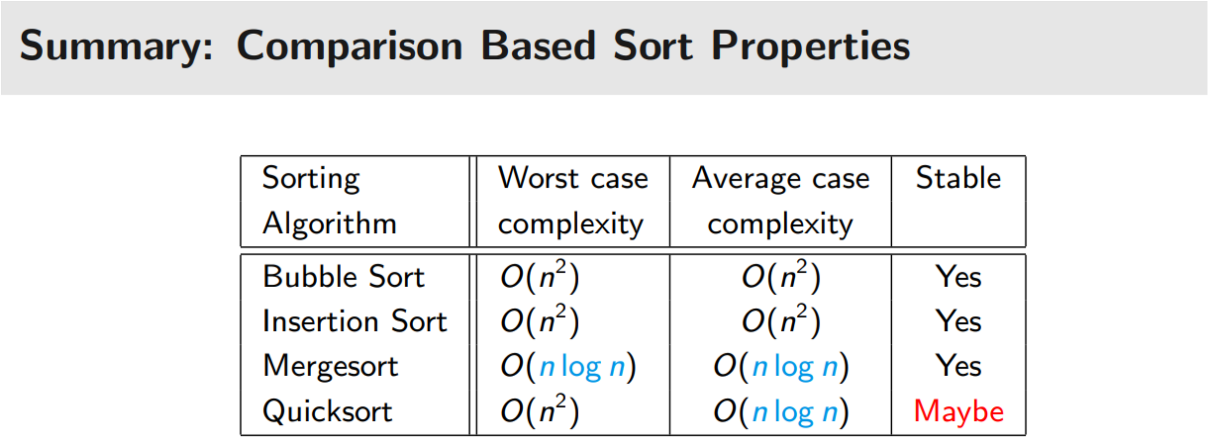
Stability of Quicksort

- The stability of Quicksort depends on the implementation.
- The *in-place* implementation, as we discussed earlier is not stable as it swaps elements without paying attention to where these will end up.

Challenge! Write an implementation of Quicksort that is stable! Can you do that without allocating a new array?

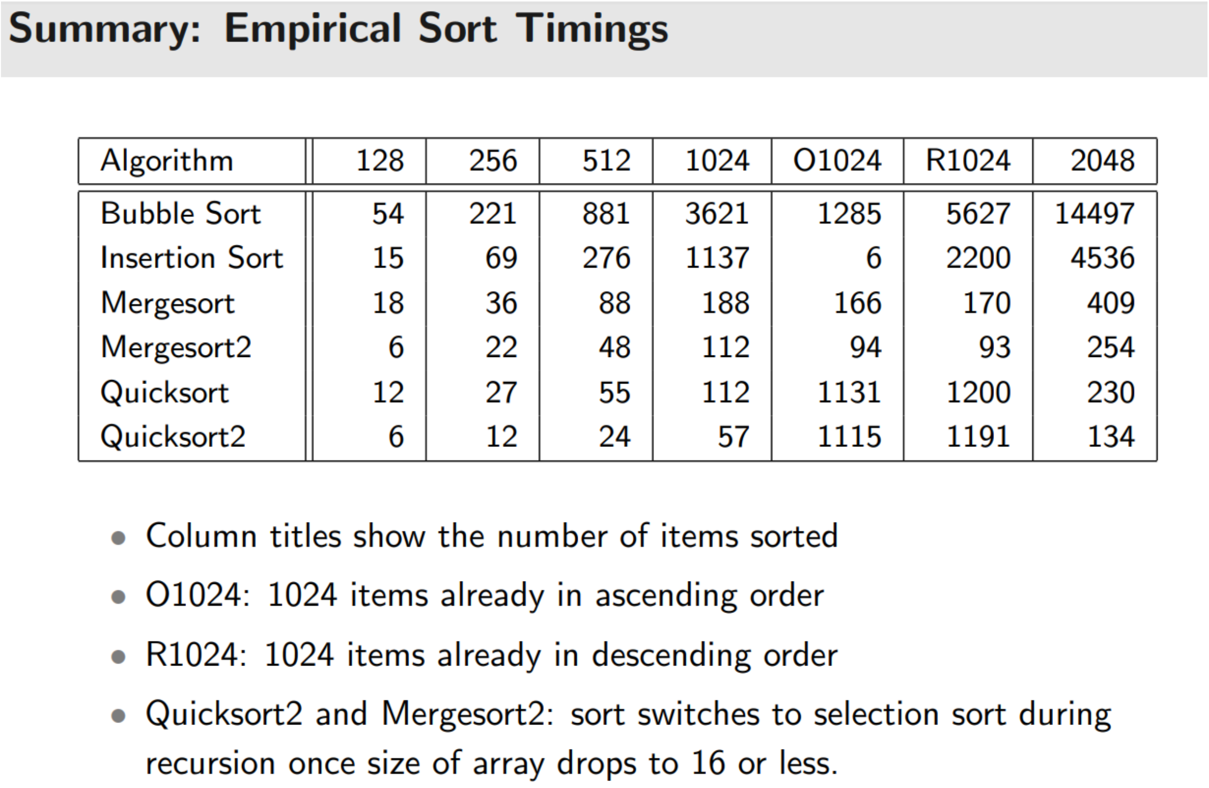
> ***If you can do this (^) you've understood all the stability stuff above***

Sorting summary



Summary: Comparison Based Sort Properties

Sorting Algorithm	Worst case complexity	Average case complexity	Stable
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes
Insertion Sort	$O(n^2)$	$O(n^2)$	Yes
Mergesort	$O(n \log n)$	$O(n \log n)$	Yes
Quicksort	$O(n^2)$	$O(n \log n)$	Maybe



Summary: Empirical Sort Timings

Algorithm	128	256	512	1024	O1024	R1024	2048
Bubble Sort	54	221	881	3621	1285	5627	14497
Insertion Sort	15	69	276	1137	6	2200	4536
Mergesort	18	36	88	188	166	170	409
Mergesort2	6	22	48	112	94	93	254
Quicksort	12	27	55	112	1131	1200	230
Quicksort2	6	12	24	57	1115	1191	134

- Column titles show the number of items sorted
- O1024: 1024 items already in ascending order
- R1024: 1024 items already in descending order
- Quicksort2 and Mergesort2: sort switches to selection sort during recursion once size of array drops to 16 or less.

Non-Comparison Sorts

💡 **Binsort**

- *Binsort is a type of sort that is not based on comparisons between key values*
  - *Instead it simply assigns records to "bins" based on the key value of the record alone.*
- *These bins, in Abstract Data Type terms, are Queue data structures,*
  - *That maintain the order that records are inserted into them.*
- *The final step of the bin sort is to concatenate the queues together to get a single list of records with all records of the first bin followed by all records of the second bin, etc.*
- ***Binsort is a stable sort because values that belong in the same bin are enqueued in the order that they appear in the input.***
- *Binsort does one pass through the input to fill the bins, and one pass through the bins to create the output list, so this is* $O(n)$.

Binsort Example

1. *For example, with a shuffled deck of* 52 *playing cards, you can do a pass through the deck separating each card into one of* 13 *piles by their face value* $(Ace, 2, 3, ..., Jack, Queen, King)$.
2. *Each pile, or bin, would end up with 4 cards with the same face value, but the suits* $(Hearts, Diamonds, Clubs, Spades)$ *within each bin would still be mixed up.*
3. *We can now put all the piles together to make a single pile of* 52 *cards, which are sorted by face value but not by suit.*
4. *If we now do another bin sort on the pile obtained from our first sort, but this time based on suits rather than face values, you end up with 4 piles of 13 cards, with one pile for each suit.*
5. *This time, because of the stability of binsort, each pile* ***WILL*** *be sorted by face value: since the input was sorted by face value, cards are put into each suit pile in face value order.*
6. ***It (^) makes the point that if the sort algorithm is run twice with two separate conditions then the first conditions holds even when the second condition is being run due to the algorithm's stability***

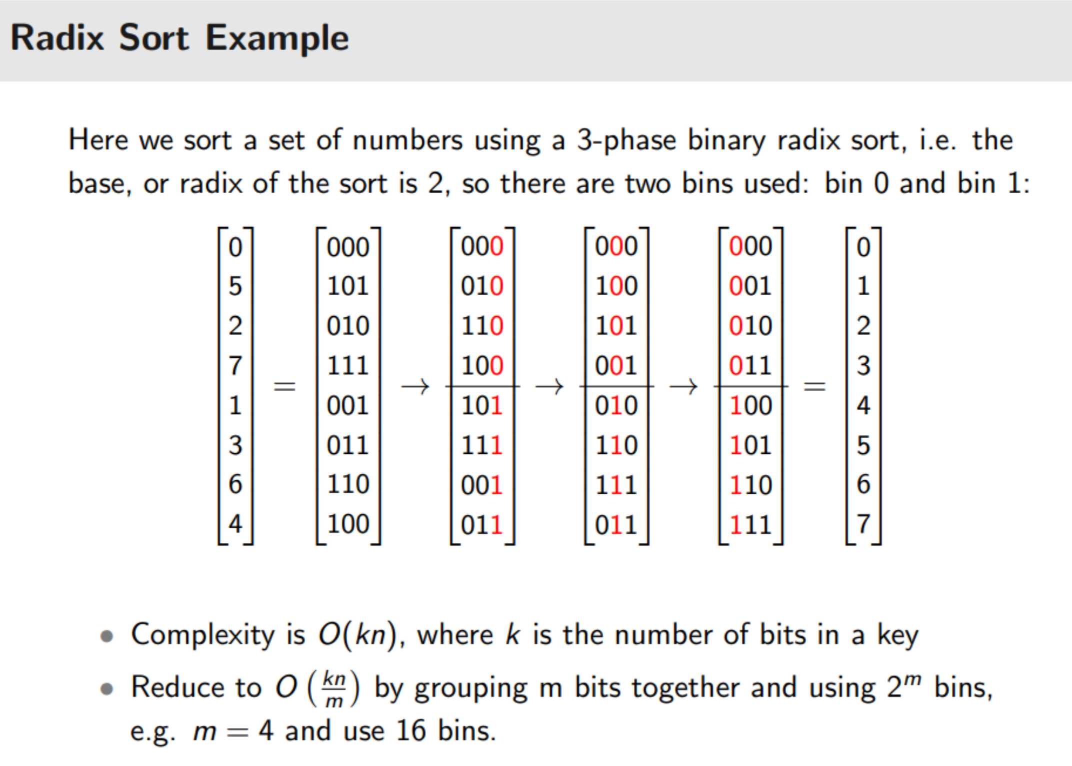
Further Binsort Examples

- *Dates are suitable values to do such "multi-phase" binsorts on:*

    - *Sort first by day*
    - *Then by month*
    - *Then by year*
- *To obtain the list of dates in Year, Month, and Day order.*
- *A variant on binsort is bucket sort, where instead of “scattering” records into bins based just on a value (which could be numeric or categorical),*
    - *They are scattered into buckets based on a range of numeric values or a set of categories.*
    - *Similar enough though. Instead of one condition met a range of conditions are met*

💡 **Radix Sort**

- Radix sort is a multi-phase bin sort where the key sorted in each phase is a different, more significant base power of the integer key.
- For example, in base (or radix) 10, an integer has digits for units, 10s, 100s, 1000s, etc.
- In a radix sort, a bin sort on the unit digit is performed first, then on the 10s digit, then on the 100s digit, etc
- The result final result will be that the keys are sorted first by the most significant digit, then by the next most significant digit, . . . , until finally by the least significant digit.
- That is, they will be sorted into normal integer order.



Radix Sort Example

Here we sort a set of numbers using a 3-phase binary radix sort, i.e. the base, or radix of the sort is 2, so there are two bins used: bin 0 and bin 1:

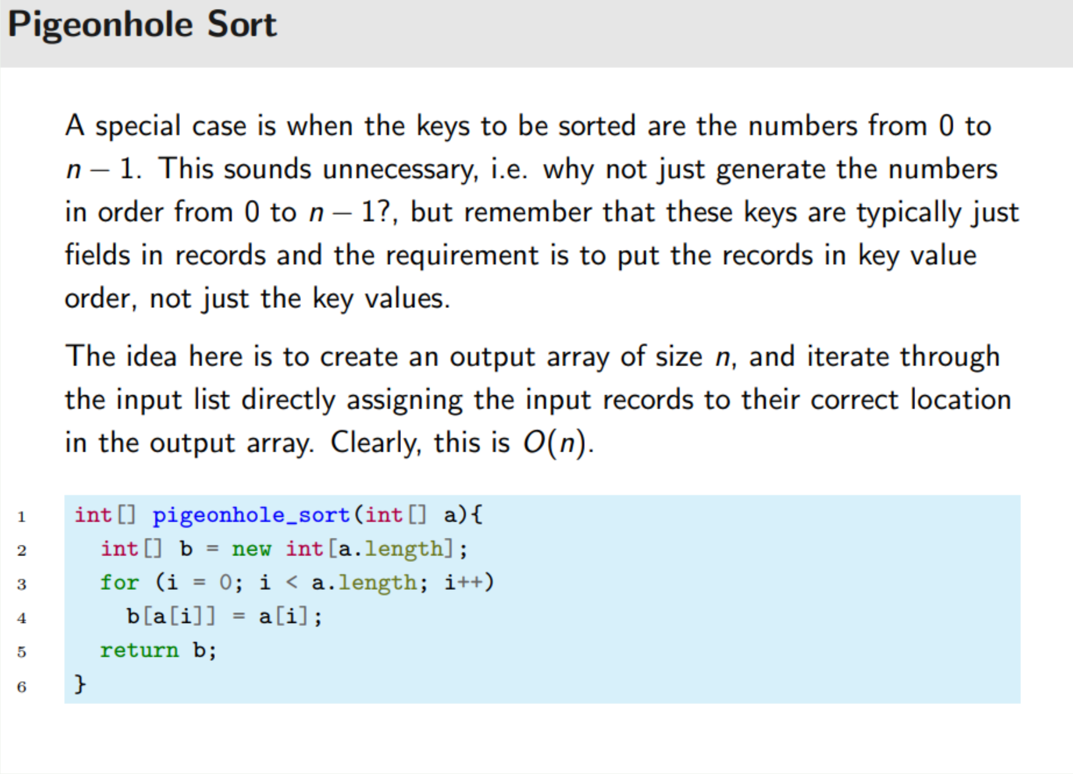
$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \to \left[\begin{array}{c}000\\010\\110\\100\\\hline 101\\111\\001\\011\end{array}\right] \to \left[\begin{array}{c}000\\100\\101\\001\\\hline 010\\110\\111\\011\end{array}\right] \to \left[\begin{array}{c}000\\001\\010\\011\\\hline 100\\101\\110\\111\end{array}\right] = \begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}$$

- Complexity is $O(kn)$, where k is the number of bits in a key
- Reduce to $O\left(\frac{kn}{m}\right)$ by grouping m bits together and using 2^m bins, e.g. $m = 4$ and use 16 bins.

- So it went from $[0, 5, 2, 7, 1, 3, 6, 4] \Rightarrow [0, 2, 6, 4, 5, 7, 1, 3] \Rightarrow [0, 4, 5, 1, 2, 6, 7, 3] \Rightarrow [0, 1, 2, 3, 4, 5, 6, 7]$
- It sorted first based on the unit column, then the tens, then the hundreds
- The radix number defines what the “integer key” in the “more significant base power of the integer key” sentence is.

For a phase 3:

- In base 10 its 1s,10s,100s
- In base 2 its 1s,2s,4s



Pigeonhole Sort

A special case is when the keys to be sorted are the numbers from 0 to $n-1$. This sounds unnecessary, i.e. why not just generate the numbers in order from 0 to $n-1$?, but remember that these keys are typically just fields in records and the requirement is to put the records in key value order, not just the key values.

The idea here is to create an output array of size n, and iterate through the input list directly assigning the input records to their correct location in the output array. Clearly, this is $O(n)$.

```
int[] pigeonhole_sort(int[] a){
  int[] b = new int[a.length];
  for (i = 0; i < a.length; i++)
    b[a[i]] = a[i];
  return b;
}
```

You basically do $arr[i] - min$ in order to compute the location in the new array to place the current original array value. Then simply overwrite the original array with the new array

https://www.youtube.com/watch?v=nVQz0kZNC64

Pigeonhole Sort in-place

We can avoid allocating the extra array and doing the extra copy as follows:

```
void pigeonhole_sort_inplace(int[] a) {
  for (i = 0; i < a.length; i++) {
    int tmp = a[a[i]];  // swap a[a[i]] and a[i]
    a[a[i]] = a[i];
    a[i] = tmp;
  }
}
```

3	0	4	1	2
1	0	4	3	2
0	1	4	3	2
0	1	2	3	4

Every swap results in at least one key in its correct position, and once a key is in its correct position, it is never again swapped, so there are at most $n-1$ swaps, therefore the sort is $O(n)$