Sorting: Divide and Conquer

Sorting

Given a set of records (or objects), we can **sort** them by many different 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 firstname, if there are students with the same surname
- in increasing order of module name, then by decreasing order of mark

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

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

Comparison based strategies

All algorithms we described so far are comparison based. Which means they do not depend on *what we are sorting*, but they only compare elements `x` and `y`:

$$\mathrm{x} < \mathrm{y} \text{ or } \mathrm{x} = \mathrm{y} \text{ or } \mathrm{x} > \mathrm{y}.$$

This way, we may compare `int`, `float`, `str` (alphabetically), `char` (by their ascii value), etc.

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(...)` 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`, 0, or a positive `int` if `x` is less than, equal to, or greater than `y` respectively.

`Comparator`: A `Comparator` object can be used to compare two objects of some type using the `compare(...)` 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`, 0, or a positive `int` if `x` is less than, equal to, or greater than `y` respectively.

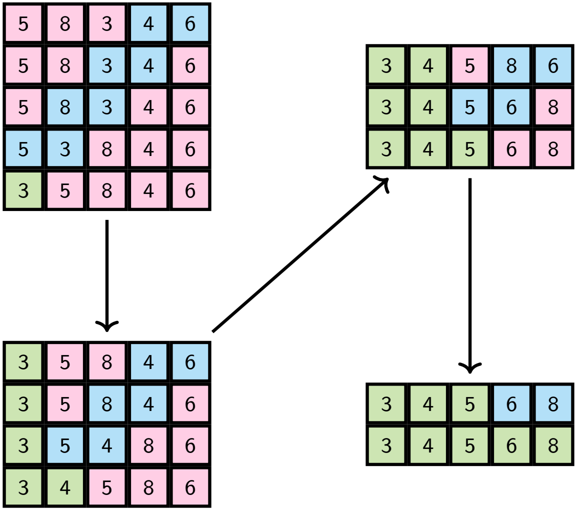
Baseline 1: Bubble Sort

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.

```
function BubbleSort(array $a$)
    while $a$ is not sorted do
        for $i = 0, 1, \ldots, \text{length}(a) - 2$ do
            if $a[i] > a[i + 1]$ then
                Swap $a[i]$ and $a[i + 1]$
```

Example of a Bubble Sort run



Bubble Sort Complexity

- The inner for-loop is iterated $n - 1$ times, with each iteration executing a single comparison.
- After each iteration, one more element is in its correct spot, hence the algorithm finishes in $n - 1$ iterations of the while-loop.

Altogether, we get

$$(n-1) \times (n-1) = n^2 - 2n + 1 = O(n^2)$$

comparisons.

Thus the worst-case complexity of Bubblesort is $O(n^2)$.

Challenge! Find an input on which Bubblesort takes time $O(n^2)$.
On what inputs is it more efficient?

Baseline 2: Selection Sort

Selection sort works by *selecting* the smallest remaining element of the input array and *appending* it at the end of 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 empty and the unsorted part is the whole input array.

In each pass it finds the smallest element of the unsorted part and swaps it with the first element of the unsorted part of the array. Then it moves the split position between the sorted and the unsorted parts of the array on by one cell.

Example of a Selection Sort run

1. | 5, 12, 6, $\underline{3}$, 11, 8, 4
2. 3 | 12, 6, 5, 11, 8, $\underline{4}$
3. 3, 4 | 6, $\underline{5}$, 11, 8, 12
4. 3, 4, 5 | $\underline{6}$, 11, 8, 12
5. 3, 4, 5, 6 | 11, $\underline{8}$, 12
6. 3, 4, 5, 6, 8 | $\underline{11}$, 12
7. 3, 4, 5, 6, 8, 11 | $\underline{12}$
8. 3, 4, 5, 6, 8, 11, 12 |

Selection Sort Complexity

The outer loop is iterated $n - 1$ times.

In the worst case, the inner loop is iterated $n - 1$ times for the first outer loop iteration, $n - 2$ times for the 2nd outer iteration, etc. Thus, the total number of comparisons is:

$$\begin{aligned}\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 &= \sum_{i=0}^{n-2} (n - 1 - i) \\ &= (n-1) + \cdots + 2 + 1 \\ &= \frac{n(n-1)}{2}.\end{aligned}$$

Hence the worst case complexity is $O(n^2)$

Divide & Conquer

Divide and Conquer strategy

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

This is a sketch of a psedocode of a function that uses this strategy:

```
function DivideAndConquer(Input $I$)
    if input $I$ is a base case then
        Process $I$ and return
    else
        Divide the input to two smaller inputs $I_0$ and $I_1$.
        $O_0 := \text{DivideAndConquer}(I_0)$.
        $O_1 := \text{DivideAndConquer}(I_1)$.
        Join the two outputs $O_0$ and $O_1$ into $O$.
        return $O$
```

Merge Sort

Mergesort as Divide and Conquer

Mergesort is an instance of use of the *Divide and Conquer* strategy.

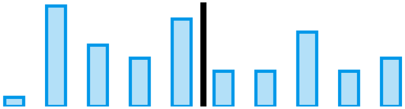
In case of *Mergesort* we:

1. Recursively split the problem into smaller sub-problems till you are left with arrays containing only one element.
2. Then merge the sorted pieces into bigger and bigger sorted sequences until we get the full array sorted.

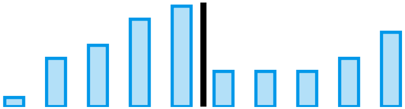
Merge Sort

Idea:

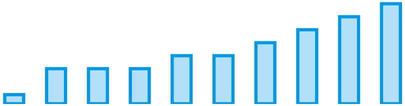
1. Split the array into two halves:



2. Sort each of them recursively:



3. Merge the sorted parts:



Example: Merge Sort run

steps
1 & 2

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

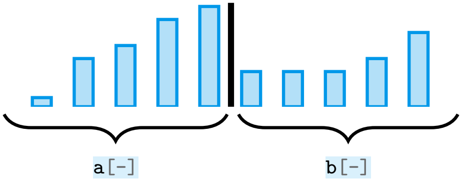
steps 3

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 Implementation

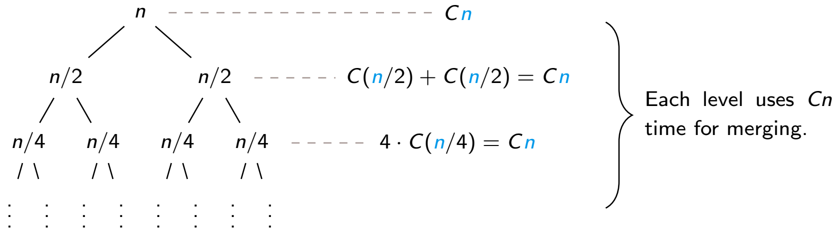
```java
int[] mergeSort(int[] a, int left=0, int right=a.length) {
  if (left < right) {
    int mid = (left+right) / 2;          // half point
    return merge(mergeSort(a, left, mid), mergeSort(a, mid, right))
  } else {
    return new int[] {};                 // empty array
  }
}

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 Mergesort

The complexity of `merge(a, b)` is $C \cdot (\mathtt{a.length} + \mathtt{b.length}) = C \cdot n$ 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 $\implies$ at most $\log n + 1$ levels
$\implies$ the total complexity is $C' \cdot n \log n = O(n \log n)$.

(This is both Worst and 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 $O(2 \times (\frac{n}{4} + \frac{n}{4})) = O(2 \times \frac{n}{2}) = O(n)$.

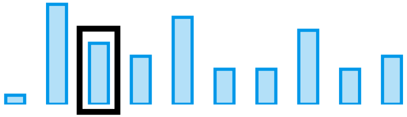
Similarly, we have 4 subproblems of size $\frac{n}{4}$, each of them is merging their subproblems in time $O(\frac{n}{8} + \frac{n}{8})$. In total, all calls of `merge` for subproblems of size $\frac{n}{4}$ take $O(4 \times (\frac{n}{8} + \frac{n}{8})) = O(n)$. ... We see that it always takes $O(n)$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

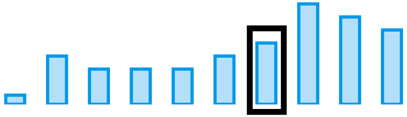
Quick Sort

Quick Sort

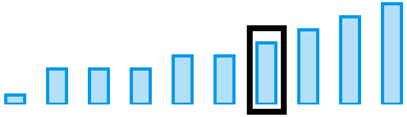
1. Select an element of the array, which we call the **pivot**.



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: In order for quicksort to be a *stable* sorting algorithm, it is useful to allow the *large entries* to also be $\geqslant$ 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. 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.

$$\begin{tikzcd}
 & & & \left\langle \boxed{4}, 5, 2, 7, 8, 1, 3, 6 \right\rangle \arrow[dll] \arrow[drr] & & & & \\
 & \left\langle \boxed{2}, 1, 3 \right\rangle \arrow[dl] \arrow[dr] & & & & \left\langle \boxed{5}, 7, 8, 6 \right\rangle \arrow[dl] \arrow[dr] & & \\
\langle 1 \rangle \arrow[dr] & & \langle 3 \rangle \arrow[dl] & & \langle \rangle \arrow[dddr] & & \left\langle \boxed{7}, 8, 6 \right\rangle \arrow[dl] \arrow[dr] & \\
 & \left\langle 1, \boxed{2}, 3 \right\rangle \arrow[dddrr] & & & & \langle 6 \rangle \arrow[dr] & & \langle 8 \rangle \arrow[dl] \\
 & & & & & & \left\langle 6, \boxed{7}, 8 \right\rangle \arrow[dl] & \\
 & & & & & \left\langle \boxed{5}, 6, 7, 8 \right\rangle \arrow[dll] & & \\
 & & & \left\langle 1, 2, 3, \boxed{4}, 5, 6, 7, 8 \right\rangle & & & &
\end{tikzcd}$$

Quick Sort implementation

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

Where `partition` rearranges the array so that

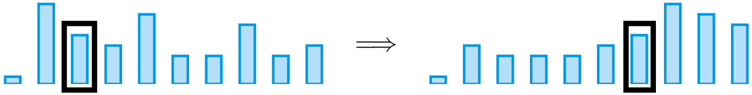
- the small entries are stored on positions `left`, `left+1`, . . . , `pivotindex-1`,
- pivot is stored on position `pivotindex` and
- the large entries are stored on `pivotindex+1`,. . . , `right`.

Partitioning array `a`

Idea:

1. Choose a pivot `p` from `a`.
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 `a` and return the index of `p` in `a`.

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;
  int big = right;
  while (small < big) {
    while (a[small] < pivot)
      small++;                  // after the loop pivot <= a[small]
    while (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;
    if (small < big && a[small] == a[big])
      small++;                  // make a step if both are == pivot
  }
  return big;                   // return index of the pivot
}
```

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?

Average case complexity

Different kinds of complexities

Worst Case complexity

= the worst complexity over all *possible inputs/situations* (complexity upper bound)

Average complexity

= average complexity over all *random choices* (in case of *randomised algorithms*, or if we know the likelihood of each of the inputs!)

Amortized complexity

= average time taken over a sequence of *consecutive* operations

(useful for measuring performance of data structures)

Randomised Binary Search

```
function RandomisedBinarySearch(array $a$, value $x$)
    $l := 0$, $r :=$ length($a$) $- 1$
    while $r > l$ do
        pick $guess \in [l, r]$ at random
        if $a[guess] < x$ then
            $l := guess + 1$
        else
            $r := guess$
    if $a[l] = x$ then
        return $l$ else return $-1$
```

Question: What is the *average complexity* of this algorithm?

Average complexity of Randomised Binary Search

This is not a formal argument, but it provides some intution.

With probability $\leqslant$ 50%, the guess is in the middle 50% of entries.
$\implies$ there are $\geqslant$ 25% many smaller entries and $\geqslant$ 25% many larger entries.
$\implies$ the partition is, in the worst case, of sizes $\frac{3}{4}n$ and $\frac{1}{4}n$.
$\implies$ with probability at least 50%, we reduce the search space to at most 3/4th of the original.
$\implies$ only $\log_{4/3} n = C \log n$ iterations are required.
$\implies$ the time complexity is $O(\log n)$.

Average time complexity of Quicksort

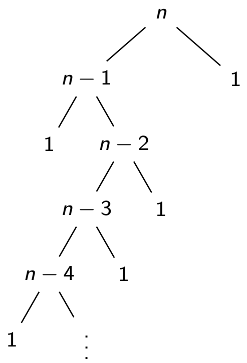
The complexity depends on the strategy which chooses the pivots!

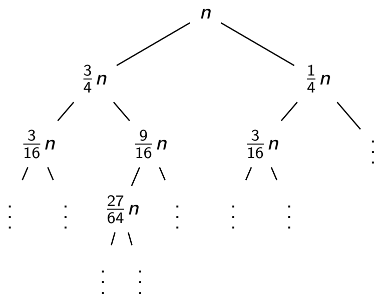
Average Case: Assuming that either:

- we choose pivot randomly, or
- that the shuffle is random (all permutations of elements are equally likely).

With probability $\leqslant 50\%$, the pivot is in the middle 50% of entries.
$\Longrightarrow$ there are $\geqslant 25\%$ many smaller entries and $\geqslant 25\%$ many larger entries.
$\Longrightarrow$ the partition is, in the worst case, of sizes $\frac{3}{4}n$ and $\frac{1}{4}n$.
$\Longrightarrow$ only $\log_{4/3} n = C \log n$ recursive calls are required.
$\Longrightarrow$ the time complexity is $O(n \log n)$.

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

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. a random entry (there is $\geqslant 50\%$ chance for a good pivot).

Remark: In practice, usually 3. 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 $\geqslant 25\%$ entries is small and $\geqslant 25\%$ is large for *every input* sequence. However, this property holds *on average* (= for a random sequence).

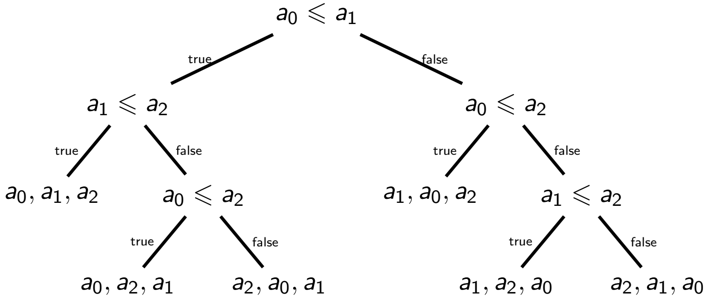
Strategy (3), although it does not guarantee that we will find a perfect pivot every single time, we pick it *often* (with 50% probability) which suffices.

Lower bounds on the complexity of sorting

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):



Minimum number of Comparisons

- 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 will be 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 will be 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. so there are $n(n-1)(n-2)\ldots 3\cdot 2\cdot 1 = n!$ 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

$$\log n! \geqslant \log(\underbrace{n \cdot (n-1) \cdots (n/2)}_{n/2} \cdot (n/2-1) \cdots 1)$$

$$\geqslant \log(n/2)^{n/2} = (n/2)\log(n/2) \geqslant C\, n \log n$$

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’ $\Omega(n\log n)$.)

A Non-Comparison Sort

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.

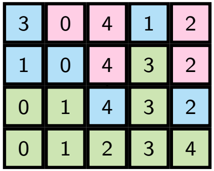
The idea here is to create an output array of size n, and iterate through the input list directly assigning the input records to their correct location in the output array. Clearly, this is $O(n)$.

```
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;
}
```

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;
  }
}
```



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 $O(n)$

Sorting summary

Stability in Sorting

A *stable* sorting algorithm does not change the order of items in the input if they have the same sort key.

Thus if we have a collection of student records which is already in order by the students' 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 firstnames.

Using stable sorting algorithms in this way, we can "*pipeline*" sorting steps to construct a particular order in stages.

In particular, a stable sorting algorithm is often faster when applied to an already sorted, or nearly sorted list of items. If your input is usually nearly sorted, then you may be able to get higher performance by using a stable sorting algorithms. However, many stable sorting algorithms have higher complexity than unstable ones, so the compexities involved should be carefully checked.

Bubble Sort Stability

Consider what happens when two elements with the same value are in the array to be sorted.

Since only neighbouring pairs 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*.

Selection Sort Stability

Consider what happens when two elements with the same value are in the array to be sorted.

For example, consider when the input array contains $2_1, 2_2, 1$, where the subscript indicates the order of appearance of the two copies of the value 2.

In the first pass, we find the smallest element, in this case the 1, and swap it with the first element in the array, the 2_1. This results in $1, 2_2, 2_1$. In other words, the 2 copies of 2 have changed order.

No matter how we change the condition for which element of a set of elements of the same (smallest) value we then select, we can easily produce counterexamples that show that the order of elements with the same value can change.

Hence selection sort is *unstable*.

Stability of Mergesort

The splitting phase of mergesort does not change the order of any items.

So long as 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 (as the pseudocode above does) then different elements with the same values do not change their relative order.

Therefore mergesort is stable.

Stability of Quicksort

Stability of Quicksort depends on the implementation.

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

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

Summary: Comparison Based Sort Properties

Sorting Algorithm	Worst case complexity	Average case complexity	Stable
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	No
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
Selection Sort	12	45	164	634	643	833	2497
Mergesort	18	36	88	188	166	170	409
Quicksort	12	27	55	112	1131	1200	230

- Column titles show the number of items sorted
- O1024: 1024 items already in ascending order
- R1024: 1024 items already in descending order

Challenge! Produce a table like that yourself!
Write a sorting algorithm that beats all of the above.