Sorting

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.

Bubble Sort

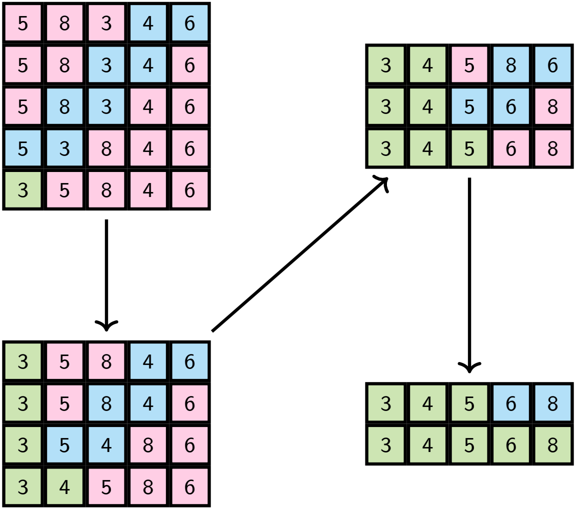
Bubble Sort

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



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:

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

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

Insertion Sort

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, 2 times for the 2nd outer iteration, etc.

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

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

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., $n(n-1)/4$.

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

Merge Sort

Divide and Conquer

Mergesort is an instance of use of a very successive general coding strategy often called *Divide and 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.

Divide and Conquer

Mergesort is an instance of use of a very successive general coding strategy often called *Divide and Conquer*.

Divide and Conquer strategy:

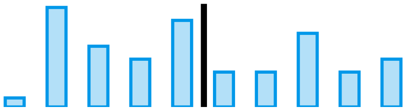
1. Recursively split the problem into smaller sub-problems till you are left with much simpler problems.
2. Then put together solutions of these smaller problems into a solution of the big problem.

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.

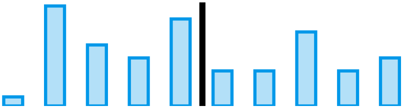
Idea:

1. Split the array into two halves:

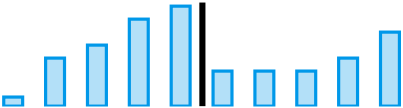


Idea:

1. Split the array into two halves:



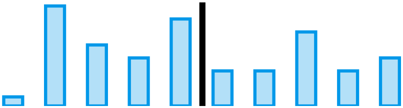
2. Sort each of them recursively:



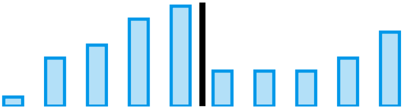
Merge Sort

Idea:

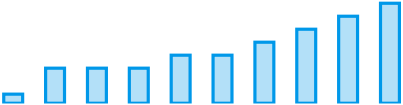
1. Split the array into two halves:



2. Sort each of them recursively:



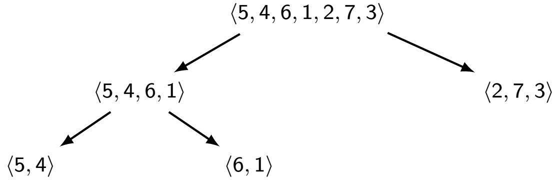
3. Merge the sorted parts:



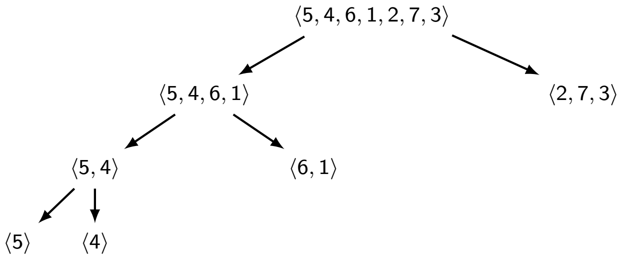
Example: Merge Sort run

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

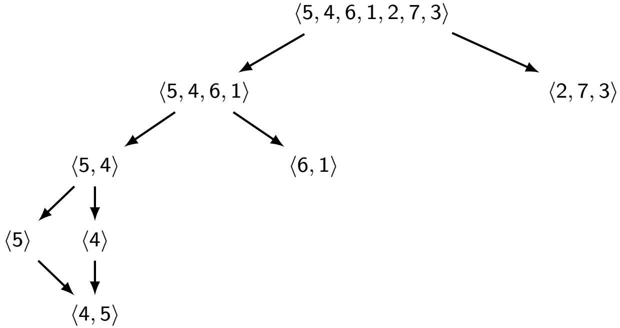
Example: Merge Sort run



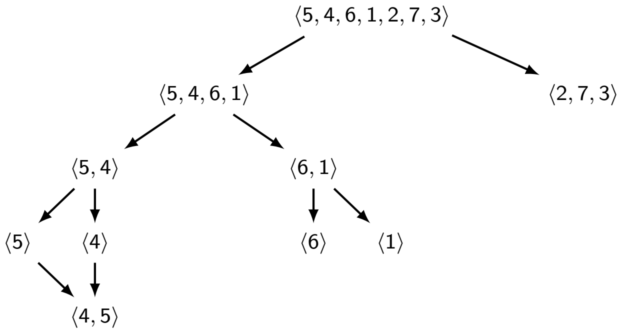
Example: Merge Sort run



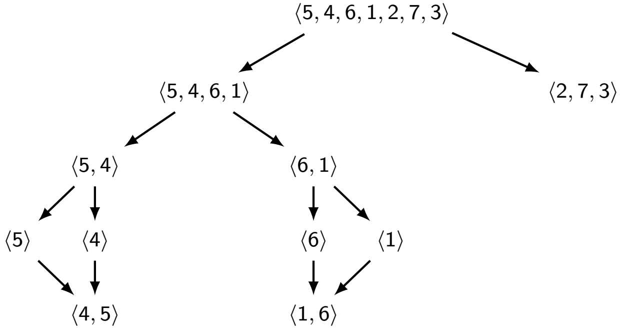
Example: Merge Sort run



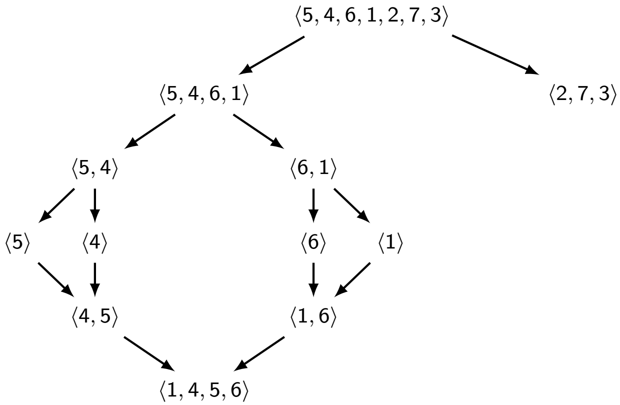
Example: Merge Sort run



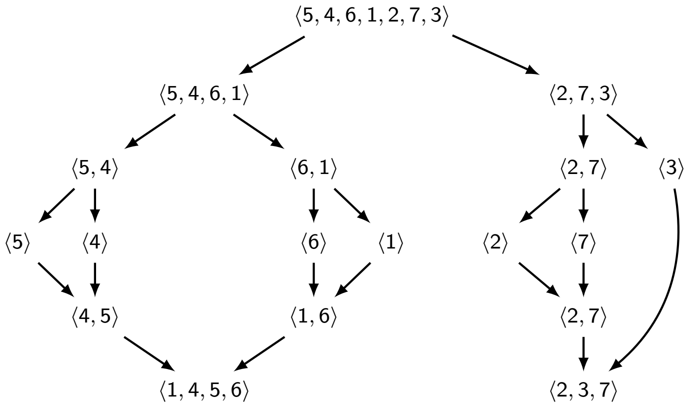
Example: Merge Sort run



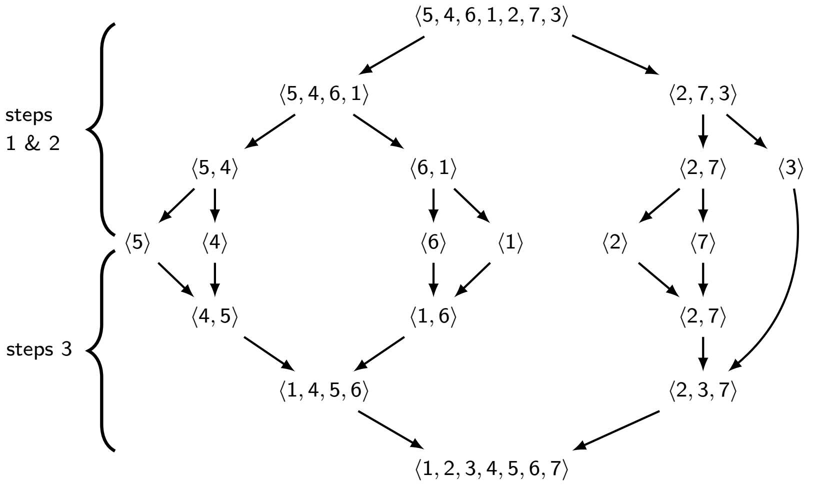
Example: Merge Sort run



Example: Merge Sort run



Example: Merge Sort run

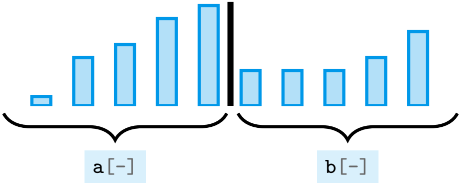


Merging two sorted arrays `a[-]` and `b[-]` efficiently

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 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 Mergesort

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:

n - - - - - - - - - - - - - - - - - - Cn

$n/2$ $\quad$ $n/2$ - - - - - - $C(n/2) + C(n/2) = Cn$

$n/4$ $\quad$ $n/4$ $\quad$ $n/4$ $\quad$ $n/4$ - - - - - $4 \cdot C(n/4) = Cn$

$\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots$

Each level uses Cn time for merging.

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 $O(2 \times (\frac{n}{4} + \frac{n}{4})) = O(2 \times \frac{n}{2}) = O(n)$.

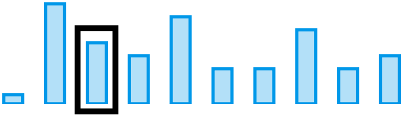
Similarly, we have 4 subproblems of size $\frac{n}{4}$, each of them is merging their subproblems in time $O(\frac{n}{8} + \frac{n}{8})$. In total, all calls of `merge` for subproblems of size $\frac{n}{4}$ take $O(4 \times (\frac{n}{8} + \frac{n}{8})) = O(n)$. ... We see that it always takes $O(n)$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

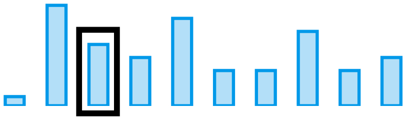
Quick Sort

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

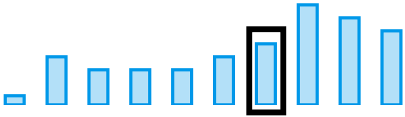


Quick Sort

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

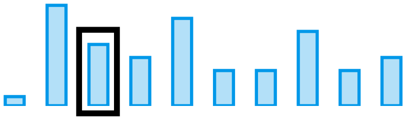


2. Partition the array so that the *"small entries"* ($\leqslant$ pivot) are on the left, then the pivot, then the *"large entries"* ($>$ pivot).

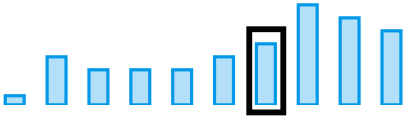


Quick Sort

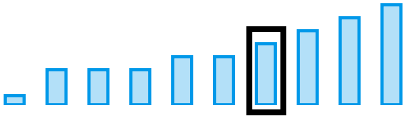
1. Select an element of the array, which we call the **pivot**.



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.

$$\left\langle \boxed{4}, 5, 2, 7, 8, 1, 3, 6 \right\rangle$$

$$\left\langle \boxed{2}, 1, 3 \right\rangle \qquad\qquad \left\langle \boxed{5}, 7, 8, 6 \right\rangle$$

Example: Quick Sort run

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

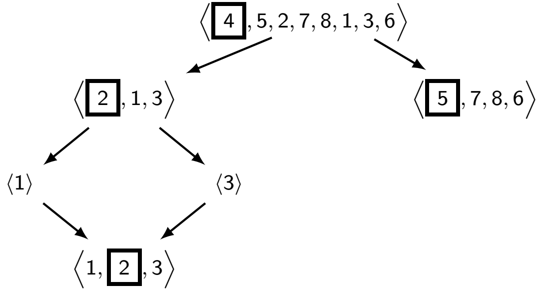
$$\left\langle \boxed{4}, 5, 2, 7, 8, 1, 3, 6 \right\rangle$$

$$\left\langle \boxed{2}, 1, 3 \right\rangle \qquad \left\langle \boxed{5}, 7, 8, 6 \right\rangle$$

$$\langle 1 \rangle \qquad \langle 3 \rangle$$

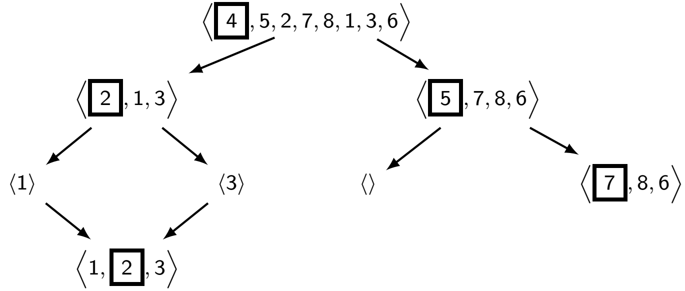
Example: Quick Sort run

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



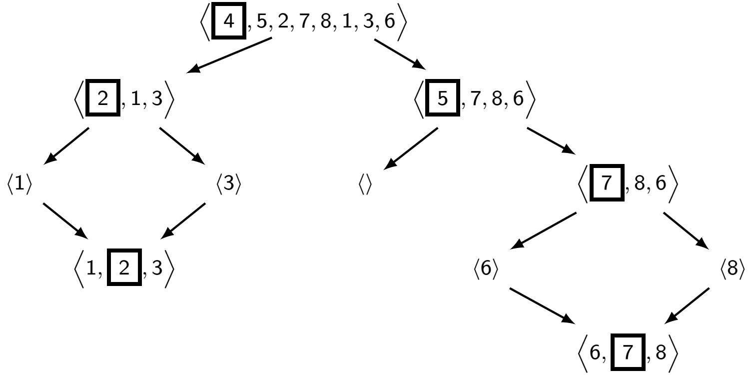
Example: Quick Sort run

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



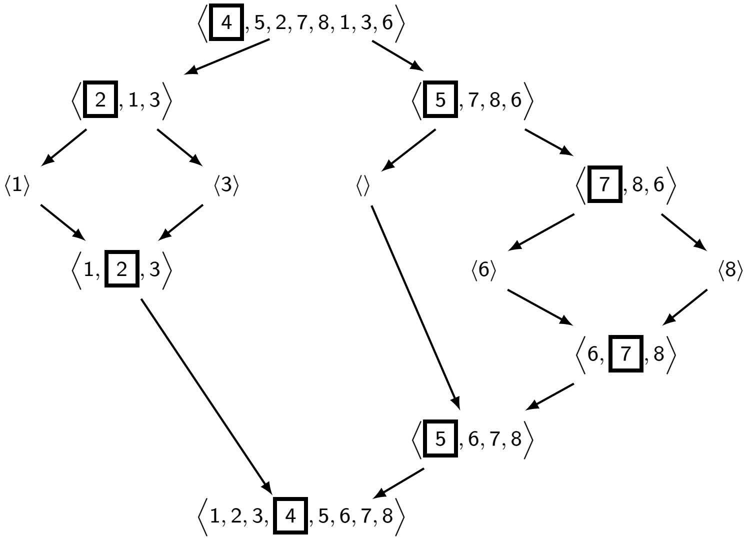
Example: Quick Sort run

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



Example: Quick Sort run

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



Quick Sort implementation

```
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

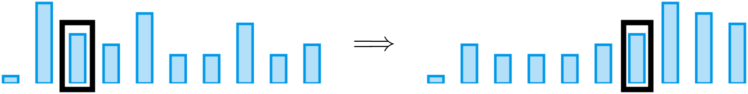
- the small entries are stored on positions `left`, `left+1`, . . . , `pivotindex-1`,
- pivot is stored on position `pivotindex` and
- the large entries are stored on `pivotindex+1`,. . . , `right`.

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 + 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
}
```

Time Complexity of Quicksort

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

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

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 $O(n^2)$.

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 time complexity of Quicksort

The complexity depends on the strategy which chooses the pivots!

Average time complexity of Quicksort

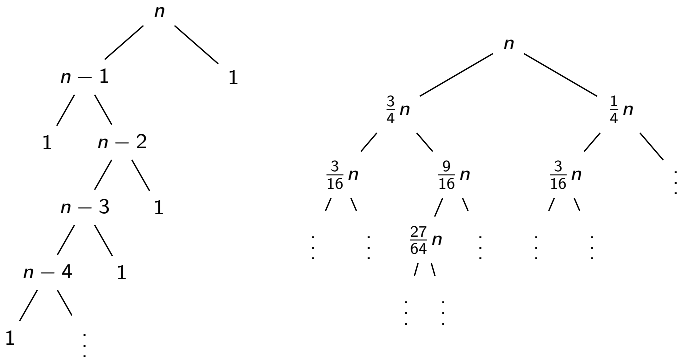
The complexity depends on the strategy which chooses the pivots!

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.
$\implies$ 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 $\frac{1}{4}n$.
$\implies$ only $\log_{4/3} n = C \log n$ recursive calls are required.
$\implies$ 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

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`:

`x` $<$ `y` or `x` $=$ `y` or `x` $>$ `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:

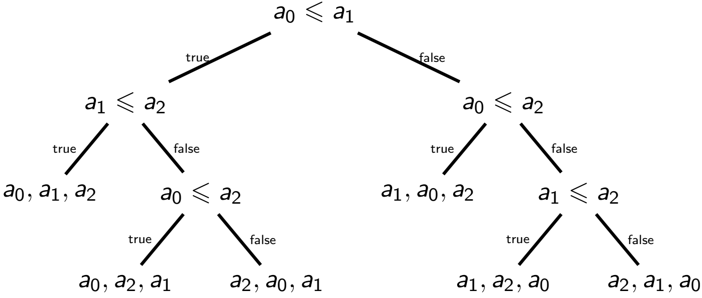
`Comparable`: A `Comparable` object can compare itself with another object using its `compareTo(...)` method. There can only be one such method for any class, so this should implement the default ordering of objects of this type. `x.compareTo(y)` should return a negative `int`, 0, or a positive `int` 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.

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' $\Omega(n \log n)$.)

Stability of sorting algorithms

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*.

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 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?

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 but instead simply assigns records to "*bins*" based of the key value of the record alone.

These bins, in Abstract Data Type terms, are Queue data structures, which maintain the order that records are inserted into them.

The final step of the binsort is to concatenate the queues together in order 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

For example, with a shuffled deck of 52 playing cards you can do a pass through the deck separating out each card into one of 13 piles by their face value (Ace, 2, 3,. . . , Jack, Queen, King). 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. 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.

If we now do another binsort on the pile obtained from our first binsort, 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. 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.

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, Day order.

A variant on binsort is *bucketsort*, 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.

Radix Sort

Radix sort is a multi-phase binsort where the key sorted on 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 binsort on the units 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:

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \rightarrow \begin{bmatrix}00\color{red}{0}\\01\color{red}{0}\\11\color{red}{0}\\10\color{red}{0}\\\hline 10\color{red}{1}\\11\color{red}{1}\\00\color{red}{1}\\01\color{red}{1}\end{bmatrix} \rightarrow \begin{bmatrix}0\color{red}{00}\\1\color{red}{00}\\1\color{red}{01}\\0\color{red}{01}\\\hline 0\color{red}{10}\\1\color{red}{10}\\1\color{red}{11}\\0\color{red}{11}\end{bmatrix} \rightarrow \begin{bmatrix}\color{red}{000}\\\color{red}{001}\\\color{red}{010}\\\color{red}{011}\\\hline \color{red}{100}\\\color{red}{101}\\\color{red}{110}\\\color{red}{111}\end{bmatrix} = \begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}$$

- 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.

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 $O(n)$.

```java
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;
  }
}
```

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 $O(n)$