
Solution:

First a remark: the notation as below only gives an indication of what happens during the execution of the algorithms. It is a step-by-step rendering but does not explain the way the elements are moved in the array. It can be used to check your pictures.

The way to answer this question in an exam is using illustrations for example in the book in Figure 2.2 on page 18.

Illustration of selection sort, where the sorted part of the sequence is written in red.

Bubble sort: The start and the first iteration bringing the maximum element to the right; the magenta indicates which elements are considered in the comparison of 'neighbours'.

The sorted part is written in red. The next (but not last, hence to be completed) iteration:
The recursion tree for merge sort in two parts:

```
[22, 15, 36, 20, 3, 9, 29]
 /    \
[22, 15, 36, 20]  [3, 9, 29]
 /    \
[22, 15]  [36, 20]  [3, 9]  [29]
 /    \
[22]  [15]  [36]  [20]  [3]  [9]
```

and

```
[3, 9, 15, 20, 22, 29, 36]
 /    \
[15, 20, 22, 36]  [3, 9, 29]
 /    \
[15, 22]  [20, 36]  [3, 9]  [29]
 /    \
[22]  [15]  [36]  [20]  [3]  [9]
```

2. Describe a worst-case input array of length $n$ for selection sort and also for bubble sort.

Then, argue that the worst-case time complexity of both these sorting algorithms is in $\Theta(n^2)$.

Solution:

A worst-case input-sequence for selection sort is obtained if the test succeeds every time, that is, if the input is in decreasing order. Then we each time have to do the assignment for $m$.

A best-case input-sequence for selection sort is when the input is sorted. Then the assignment for $m$ is not done, but still the swap has to be done.

A worst-case input-sequence for bubble sort is when the comparison of $A[j]$ and $A[j + 1]$ succeeds every time. This is the case if the input-sequence is in decreasing order.
A best-case input-sequence for bubble sort is when the input is sorted. In the rendering of the algorithm as on the slide we then still have to go through the two nested for-loops. However, in another rendering using a boolean variable intuitively indicating whether we have swapped, a sorted input sequence can be detected in $O(n)$.

For selection sort: We do $n$ times the test for the $i$-for-loop. We do $n - 1$ times the assignment for $m$. We do $n - i + 1$ times the test for the $j$-for-loop. Worst-case: the test $A[j] < A[m]$ always succeeds. Then we do, for a fixed $i$, $n - i$ times the assignment for $m$. Whatever the test gives, we do the swap. So for a fixed $i$ we do $n - i$ times the swap.

The inner for-loop takes time $\sum_{i=1}^{n-1} (n - i) = (n - 1) + \ldots + 1 = \frac{1}{2}(n - 1)n$. Hence the worst-case time complexity of selection sort is in $\Theta(n^2)$.

For bubble sort: The worst-case time complexity is determined by the two nested for-loops. The test for applying the $i$-for-loop is done $n$ times. The body of the $i$-for-loop is executed $n - 1$ times. For a fixed $i$, the test for the $j$-for-loop is done $i + 1$ times. Still for a fixed $i$, the body of the $j$-for-loop is executed $i$ times. In every case we compare two elements. In the worst case we perform a swap. The summation $\sum_{i=1}^{n-1} j = \frac{1}{2}(n - 1)n$ describes the worst-case time complexity of bubble sort. Hence the worst-case time complexity of bubble sort is in $\Theta(n^2)$. (Here we do not exit in case of a sorted input sequence.)

3. Use an invariant to show the correctness of selection sort.

Solution:

We consider the following property $I$: at the start of iteration number $k$ of the $i$-for-loop, we have $A[1 \ldots (k-1)]$ is sorted, and every element in $A[1 \ldots (k-1)]$ is smaller than every element in $A[k \ldots n]$.

We show that the property $I$ is an invariant.

First: the property $I$ holds initially at the start of iteration $k = 1$, because the array $A[1 \ldots 0]$ is empty. So it is sorted, and every element in that part is smaller than every element in the remainder of the array.

Next, assume that $I$ holds after $k$ iterations of the for-loop with $k > 1$. So $A[1 \ldots (k-1)]$ is sorted, and all elements in that part are smaller than all elements in the remainder. We execute a next iteration of the for-loop. The index $m$ is used to indicate the smallest element of the part $A[k \ldots n]$. It is by assumption larger than all elements in $A[1 \ldots (k-1)]$. It is then swapped with the element at position $k$. Hence the part from 1 up to $k$ is sorted. Also, because all element at position 1, $\ldots, (k-1)$ were smaller that the elements in the remainder (by assumption), and $k$ was the smallest part of the remainder, we have that all elements at positions 1, $\ldots, k$ are smaller than all elements at positions $k + 1, \ldots n$.

We conclude that $I$ is an invariant.
Finally, we use \( I \) to show that the output is a sorted array. After exiting the \( i \)-for loop, we have \( i = n \). The property \( I \) for that case is: \( A[1 \ldots (n - 1)] \) is sorted, and every element in \( A[1 \ldots (n - 1)] \) is smaller than every element in \( A[n \ldots n] \). This implies that \( A \) is sorted.

4. An \textit{inversion} in a sequence of integers is a pair \( i \) and \( j \) such that \( i \) occurs before \( j \) and \( i > j \).

(a) Give the inversions in \([2, 3, 8, 6, 1] \).
   The inversions of \([2, 3, 8, 6, 1] \): (2, 1), (3, 1), (8, 1), (8, 6), (6, 1).

(b) Give a permutation of \([1, \ldots, 10] \) with as many inversions as possible.
   A permutation of \([1, \ldots, 10] \) with a maximum number of inversions:
   \([10, 9, 8, 7, 6, 5, 4, 3, 2, 1] \).
   The maximum number of inversions of \([1, \ldots, n] \) is \((n - 1) + (n - 2) + \ldots + (n - (n - 1)) = \frac{n(n - 1)}{2} \).

(c) What can be said about insertion sort and the inversions in the input?
   Consider for instance the array \([2, 3, 8, 6, 1] \). It contains 5 inversions. Every inversion corresponds to a replacement-action in the execution of insertion sort. Also give the execution of insertion sort on this array, clearly indicating the replacements with arrows.

5. Give pseudo-code for a recursive version of insertion sort.

\[\textbf{Algorithm} \quad \text{InsertionSortRecFull}(A):\]
\[\quad \text{InsertionSortRec}(A, A.length)\]

\[\textbf{Algorithm} \quad \text{InsertionSortRec}(A, last):\]
\[\quad \text{if last} \leq 1 \text{ then}\]
\[\quad \quad \text{return}\]
\[\quad \quad \text{InsertionSortRec}(A, last - 1)\]
\[\quad \quad \text{Insert}(A, last)\]

In the following, we assume that the input-array \( A \) is sorted.
**Algorithm Insert** $(A, last)$:

$$key := A[last]$$

$$i := last - 1$$

**while** $i \geq 1$ **and** $A[i] > key$ **do**

$$A[i+1] := A[i]$$

$$i := i - 1$$

$$A[i+1] := key$$

Give and solve a recurrence equation for the worst-case running time.

A recurrence equation for the worst-case running time:

$$T(n) = \begin{cases} 
c_0 & \text{if } n = 1 \\
T(n-1) + I(n) & \text{if } n > 1 
\end{cases}$$

where $I(n)$ is gives the running time for inserting an element (‘the $n$th element’) in sorted array of length $n - 1$. Inserting an element at the right position in a sorted array of length $n - 1$ takes in the worst case order $n$ steps, because then we have to traverse the complete array. So $I(n) = c_1 \cdot n + c_2$ for some constants $c_1$ and $c_2$. Hence we find for the recurrence equation for $T$ the following:

$$T(n) = \begin{cases} 
c_0 & \text{if } n = 1 \\
T(n-1) + c_1 \cdot n + c_2 & \text{if } n > 1 
\end{cases}$$

For simplicity we take 1 for all constants. Because we are interested in the asymptotic complexity this is no severe restriction. Then we get:

$$T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T(n-1) + n + 1 & \text{if } n > 1 
\end{cases}$$

Using the substitution method we find:

$$T(n) = T(n-1) + n + 1$$

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

$$= T(n-3) + (n-2) + (n-1) + n + 3$$

$$= \ldots$$

$$= T(n-i) + (n-i+1) + \ldots + n + i$$

We have $n = i = 1$ if $i = n - 1$. Substituting this yields:

$$T(n) = 1 + 2 + \ldots + n + (n-1)$$

$$= \frac{n(n+1)}{2} + (n-1)$$
Hence $T(n)$ is in $\Theta(n^2)$.

One of the differences with merge sort is that for insertion sort we have a recursive call on the array with only one element less.

6. Give the merge sort tree for $[0, 9, 7, 1, 4, 2, 5, 3, 6, 8]$.

Solution:

We give the recursion tree in two parts.

7. Give an example of an input sequence that has running time in $\Theta(n \log(n))$ for merge sort, but in $\Theta(n)$ for insertion sort.

Solution:

A sorted sequence (in that particular case we do not enter the while-loop).

8. Solve the following two recurrence equations:

(a) (see merge sort)
\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T\left(\frac{n}{2}\right) + n & \text{if } n > 1 
\end{cases} \]

(b) (see tiling example)

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
4T\left(\frac{n}{2}\right) & \text{if } n > 1 
\end{cases} \]

Solutions:

Using the substitution method we find:

\[
T(n) = 2T\left(\frac{n}{2}\right) + n = 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4 \cdot T\left(\frac{n}{4}\right) + 2n = 4 \cdot \left(2 \cdot T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n = 8 \cdot T\left(\frac{n}{8}\right) + 3n = \ldots = 2^i \cdot T\left(\frac{n}{2^i}\right) + i \cdot n
\]

The base case is found if \( \frac{n}{2^i} = 1 \), that is, if \( i = \log n \). We substitute this in the last equation found above: \( T(n) = n \cdot 1 + n \cdot \log n = n + n \cdot \log n \).

Hence \( T \) is in \( \Theta(n \cdot \log n) \).

Using the substitution method we find:

\[
T(n) = 4 \cdot T\left(\frac{n}{2}\right) = 4 \cdot 4 \cdot T\left(\frac{n}{4}\right) = 4 \cdot 4 \cdot 4 \cdot T\left(\frac{n}{8}\right) = \ldots = 2^{2i} T\left(\frac{n}{2^i}\right)
\]

Base case for \( n = 2^i \) so \( i = \log(n) \). Then we find:

\[ T(n) = 2^{2 \log n} \cdot 1 = n^2 \]

Hence \( T \) is in \( \Theta(n^2) \).

9. Is an algorithm with a worst-case time complexity in \( \mathcal{O}(n) \) always faster than an algorithm with a worst-case time complexity in \( \mathcal{O}(n^2) \)?

Solution:

No, it may happen that an algorithm with worst-case time complexity \( n \) \( \mathcal{O}(n^2) \) performs better than an algorithm with worst-case time complexity in \( \mathcal{O}(n) \).
Example: An algorithm $A$ with worst-case time complexity given by the function $f(n) = 100 \cdot n$ is in $\mathcal{O}(n)$, and an algorithm $B$ with worst-case time complexity given by the function $g(n) = n^2$ is in $\mathcal{O}(n^2)$. Eventually, for large inputs, it is the case that $A$ performs better. However, for the small input $n = 2$ we have $f(n) = 200 > 4 = g(n)$. So for a small input an algorithm with worst-case time complexity in $\mathcal{O}(n^2)$ may perform better than one with worst-case time complexity in $\mathcal{O}(n)$.

Other example:

For an input with a specific property an algorithm with worst-case time complexity in $\mathcal{O}(n^2)$ may perform better than one with worst-case time complexity in $\mathcal{O}(n)$. This can happen for example for bubble sort in a rendering different from the one on the slides, where we use a boolean variable indicating we have swapped. Then the output is computed in linear time for the specific case that the input was sorted.